Introduction to SwiftUI Data Wrappers

Girish Lukka

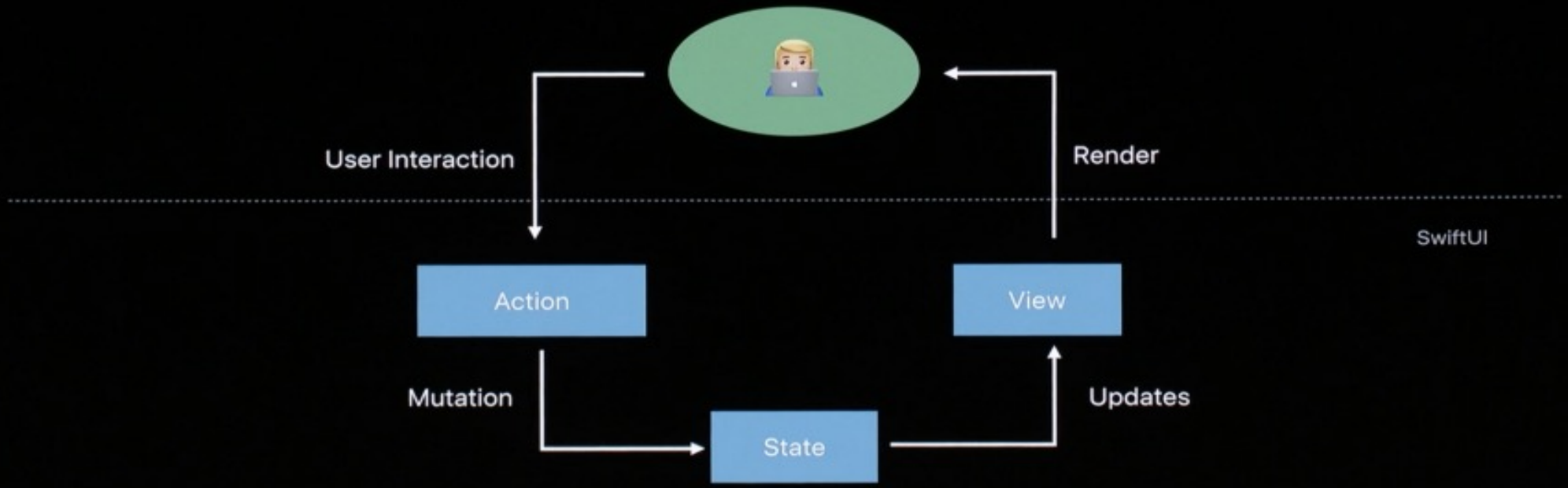
Topics

- Passing Data between Views using a property
- **Passing Data between Views using @State and @Binding**
- Passing data via the view’s environment
- Passing data via @ObservedObject and @StateObject

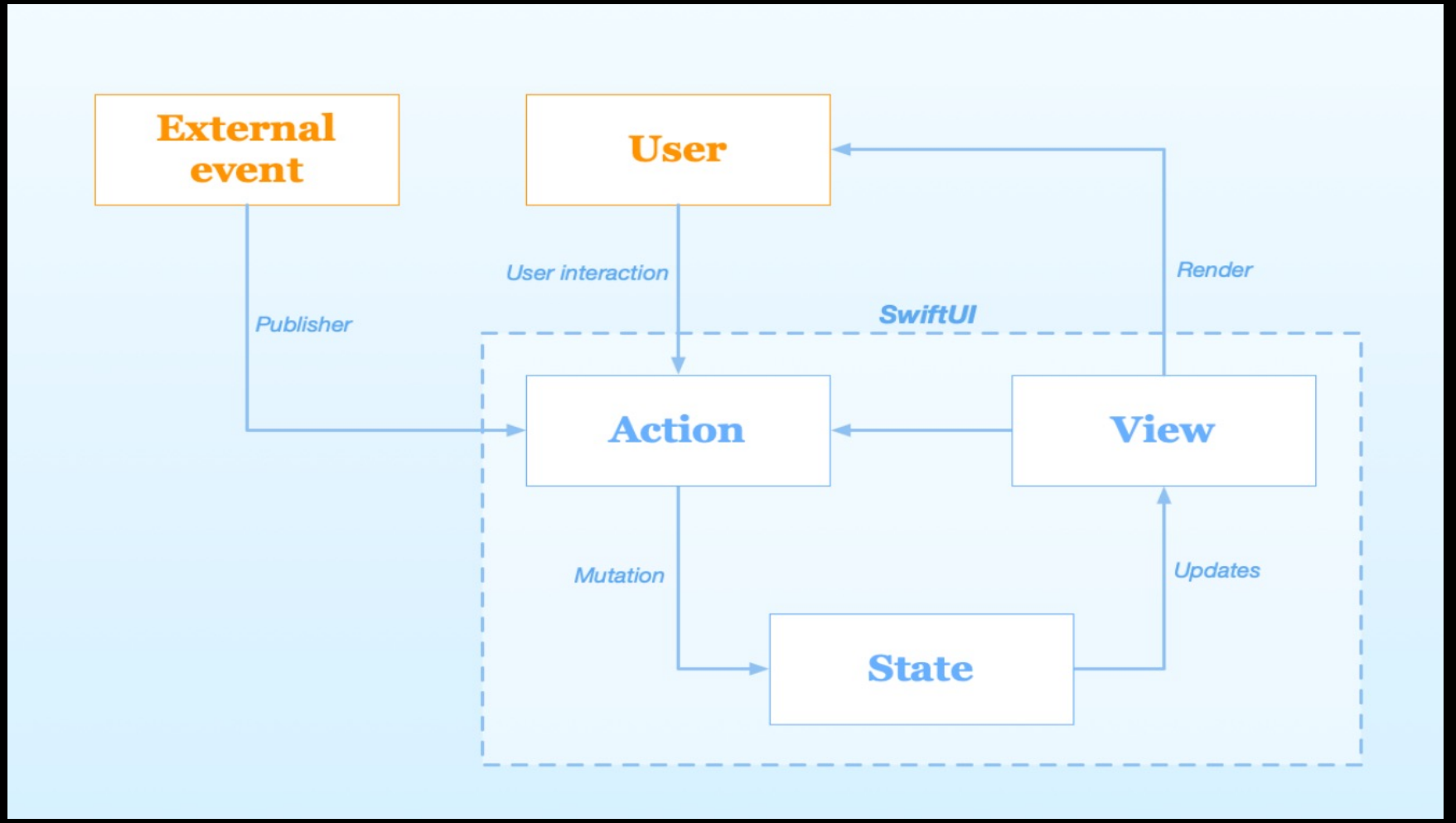
Overview

- All applications have to manage the data creation and flow.
- There should be a single source so that consistency and accuracy is maintained.
- Apple coined and formalized the phrase “ What is the Source of Truth?”
- Essentially – use a single place to store a piece of data and have everywhere else read the same piece of data.
- Should not have to try to keep two values in sync or manually update one value when another one changes.
- Challenge is where should the data be stored – usually uppermost in the hierarchy – first view closest to ContentView that needs the data.

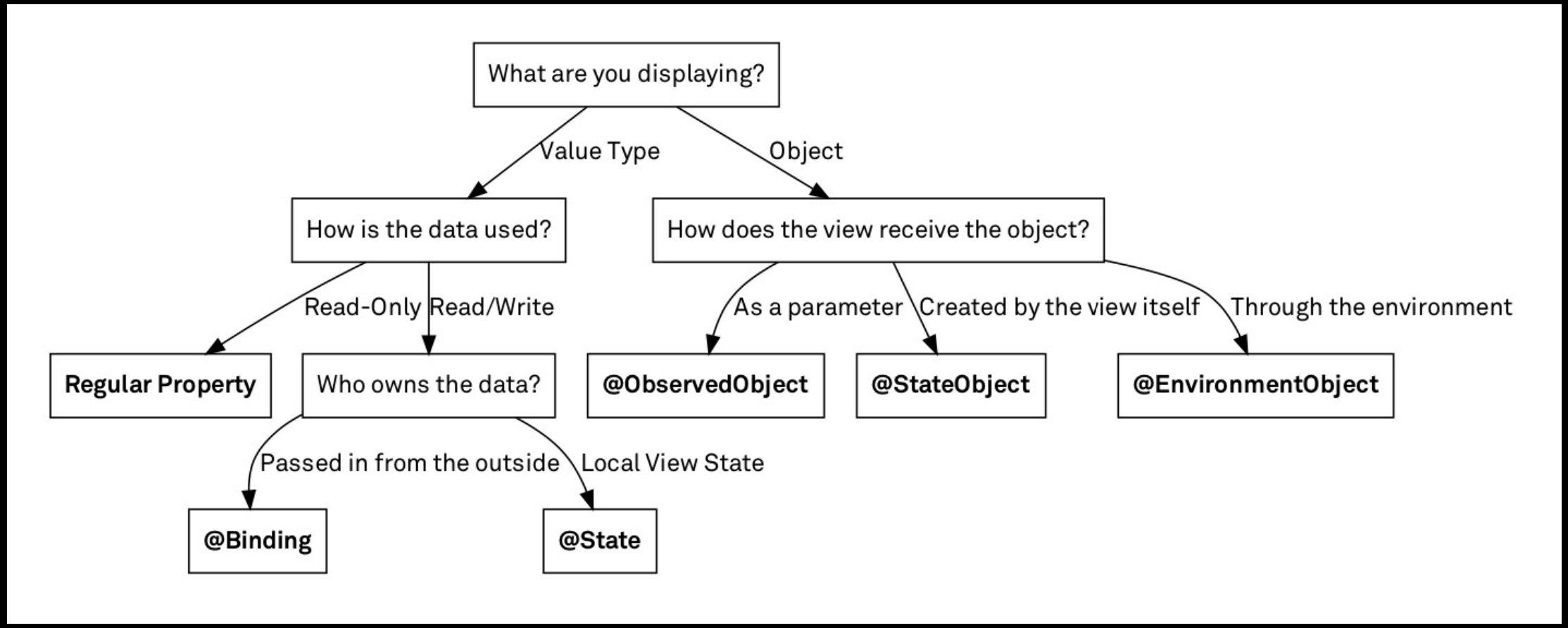
SwiftUI Data Flow - 2019 WWDC



SwiftUI Data Flow

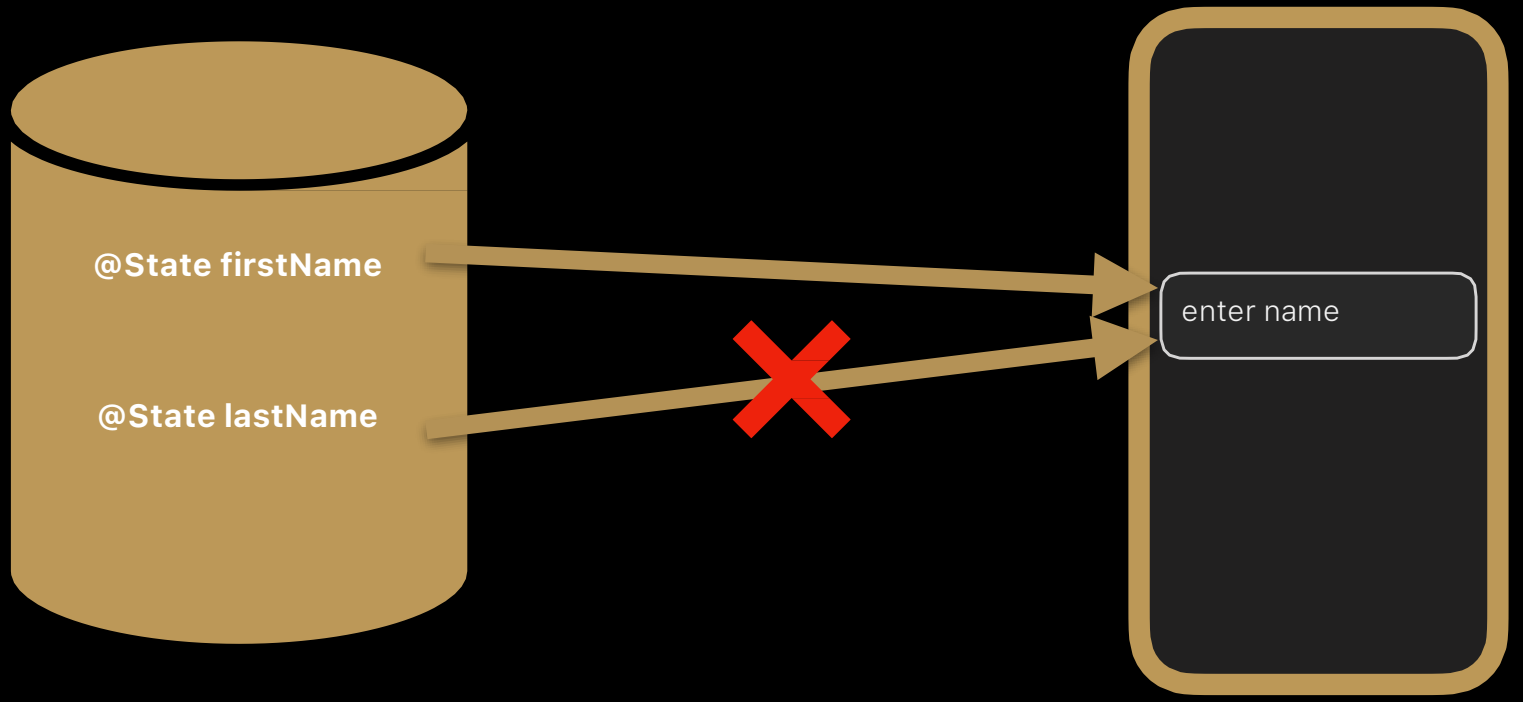


Common Property Wrappers



Single Source of Truth

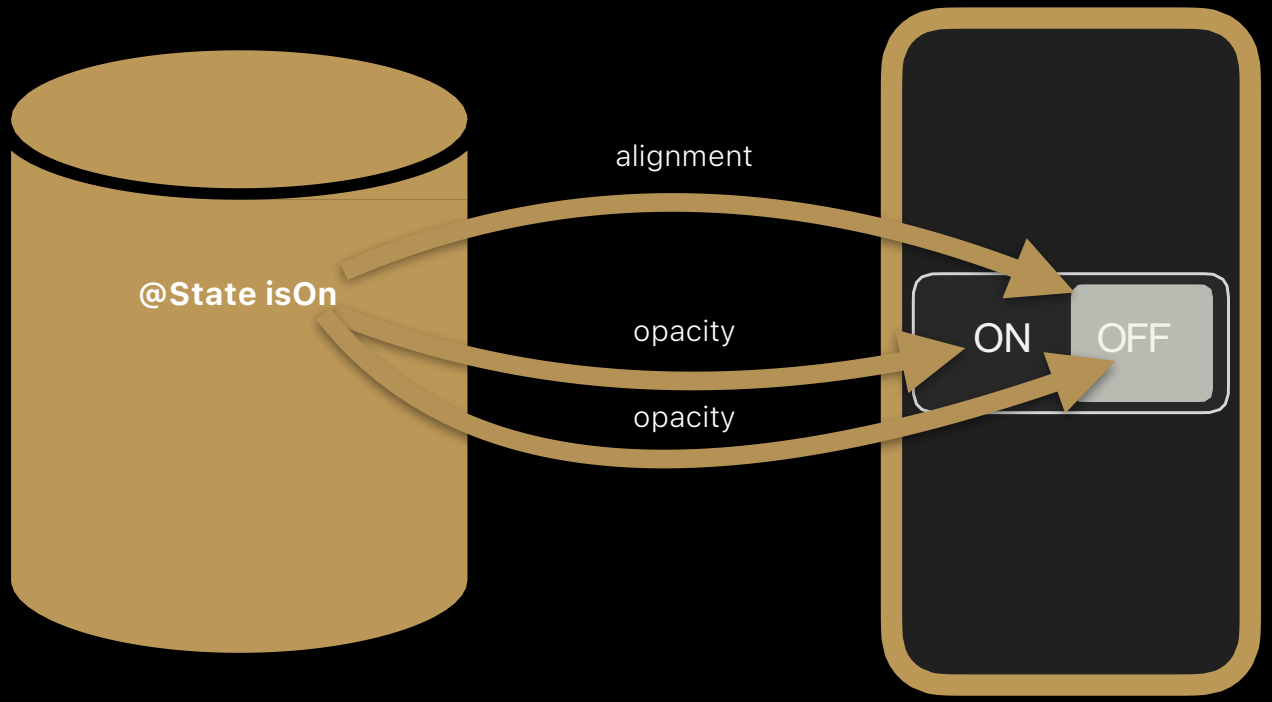
You cannot bind a UI control to more than one piece of data.



This forces you to have your data as a “single source of truth” for your UI.

Single Source of Truth for Many

You CAN have a single source of truth for **many** views though. For example, the button that changed the state property value and that changed the opacity on two views and the alignment of other views.



This forces you to have your data as a “single source of truth” for your UI.

How to Approach Wrappers for Swift Properties

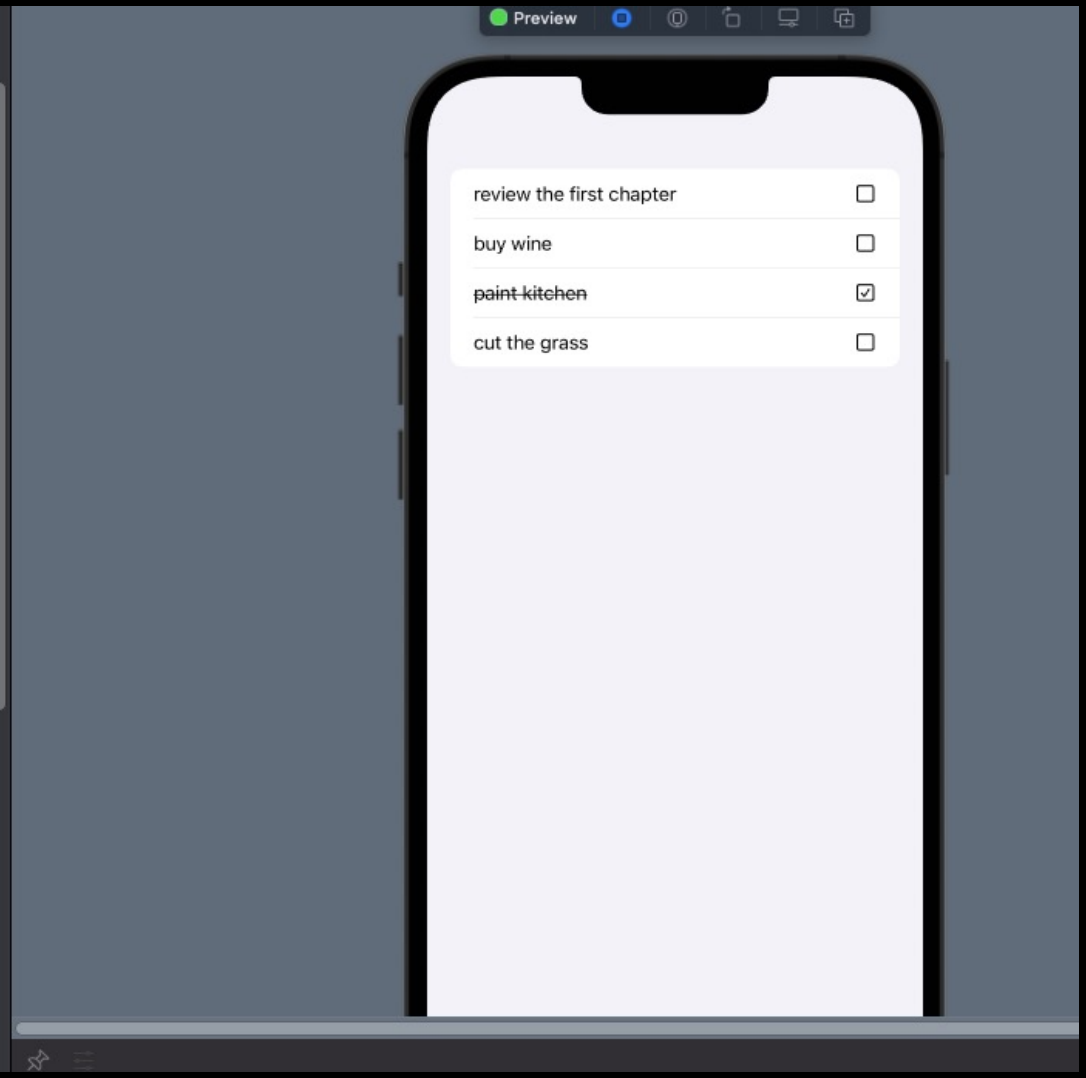
- All views are functions of their state.
- Apple documentation:
  - **SwiftUI manages the storage of any property you declare as a state**.
  - When the state value changes, the view invalidates its appearance and recomputes the body.
  - Use the state as the single source of truth for a given view.
  - A State instance isn't the value itself; it's a means of reading and writing the value

@State

- @State: To change state variables that belong to the same view. These variables should not be visible outside the view and should be marked as Private.
- The view displaying the variable will refresh each time there is a change
- **Code demo – static todo list**
  - Note that every property that's used in the body function of a View is immutable, and cannot be changed.
  - SwiftUI provides a way of passing a value as a reference, even it is a struct.
  - The original container must be decorated with a $, and the internal variable must also have this $.
  - todo is now mutable!

@State – demo: StaticTodoList

```swift
import SwiftUI

struct Todo: Identifiable {
    let id = UUID()
    let description: String
    var done: Bool
}

struct ContentView: View {
    @State
    private var todos = [
        Todo(description: "review the first chapter", done: false),
        Todo(description: "buy wine", done: false),
        Todo(description: "paint kitchen", done: false),
        Todo(description: "cut the grass", done: false),
    ]

    var body: some View {
        List($todos) { $todo  in
            HStack {
                Text(todo.description)
                    .strikethrough(todo.done)


                Spacer()
                Image(systemName:
                        todo.done ?
                      "checkmark.square" :
                        "square")
            }
            .contentShape(Rectangle())
            .onTapGesture {
                todo.done.toggle()
            }
        }
    }
}
```

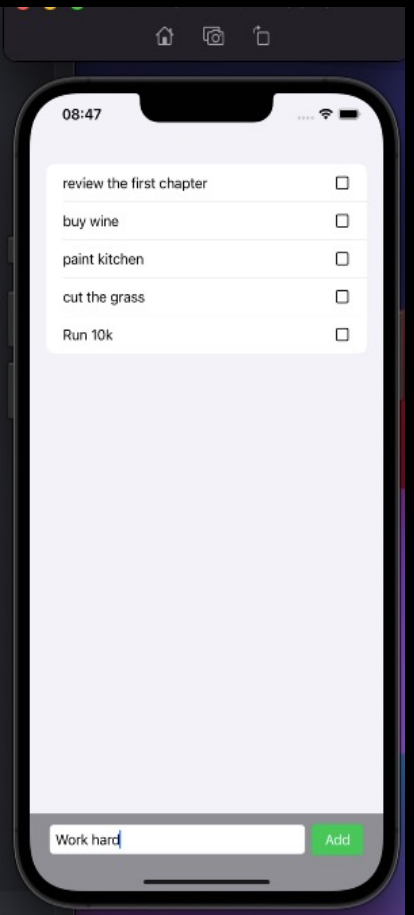


@Binding

- The @Binding property wrapper is used for properties that are passed by another view.
- The view that receives the binding is able to read the bound property, respond to changes made by external sources (like the parent view), and it has write access to the property.
- Meaning that updating a @Binding updates the corresponding property on the view that provided the @Binding.
- use @Binding if:
  - You need read- and write access to a property that's owned by a parent view.
  - The wrapped property is a value type (struct or enum).
  - You don't own the wrapped property (it's provided by a parent view).

@Binding – demo: DynamicTodoList

```swift
import SwiftUI

struct InputTodoView: View {
    @State
    private var newTodoDescription: String = ""

    @Binding
    var todos: [Todo]

    var body: some View {
        HStack {
            TextField("Todo", text: $newTodoDescription)
                .textFieldStyle(RoundedBorderTextFieldStyle())
            Spacer()
            Button {
                guard !newTodoDescription.isEmpty else {
                    return
                }
                todos.append(Todo(description: newTodoDescription,
                                  done: false))
                newTodoDescription = ""
            } label: {
                Text("Add")
                    .padding(.horizontal, 16)
                    .padding(.vertical, 8)
                    .foregroundColor(.white)
                    .background(.green)
                    .cornerRadius(5)
            }
        }
        .frame(height: 60)
        .padding(.horizontal, 24)
        .padding(.bottom, 30)
        .background(Color.gray)
    }
}

struct Todo: Identifiable {
    let id = UUID()
    let description: String
    var done: Bool
}

struct Todo: Identifiable {
    let id = UUID()
    let description: String
    var done: Bool
}

struct ContentView: View {
    @State
    private var todos = [
        Todo(description: "review the first chapter", done: false),
        Todo(description: "buy wine", done: false),
        Todo(description: "paint kitchen", done: false),
        Todo(description: "cut the grass", done: false),
    ]

    var body: some View {
        ZStack(alignment: .bottom) {
            List($todos) { $todo in
                HStack {
                    Text(todo.description)
                        .strikethrough(todo.done)
                    Spacer()
                    Image(systemName:
                            todo.done ?
                          "checkmark.square" :
                            "square")
                }
                .contentShape(Rectangle())
                .onTapGesture {
                    todo.done.toggle()
                }
            }
            InputTodoView(todos: $todos)
        }
        .edgesIgnoringSafeArea(.bottom)
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
```



@Published – see class demo from week2

- **@Published** is the most useful property wrappers in SwiftUI.
- It is allowing to create the observable objects that automatically announce when changes occur.
- SwiftUI automatically observes changes & re-invokes the body property of views that rely on the data.
- Whenever an object property marked **@Published** is changed, all views using that object will be reloaded to reflect those changes

@StateObject - *Only available in iOS 14+, iPadOS 14+, watchOS 7+, macOS 10.16+*

- The @StateObject property is used for similar reasons as @State, except it's applied to ObservableObjects. An ObservableObject is always a reference type (class) and informs SwiftUI whenever one of its @Published properties will change
- SwiftUI’s @StateObject property wrapper is designed to fill a very specific gap in state management: when you need to create a reference type inside one of your views and make sure it stays alive for use in that view and others you share it with.

@StateObject – Lecture4Demo: StateObject

```swift
//  Created by Girish Lukka on
    17/03/2022.
//

import Foundation
import Foundation

class myObject: ObservableObject{
    var name: String
    @Published var age: Int

    init(name:String,age:Int){
        self.name = name
        self.age = age
    }
}
```

```swift
//  Created by Girish Lukka on 17/03/2022.
//

import SwiftUI


struct StateObjectDemo: View {
    @StateObject var object1 = myObject(name: "Girish", age: 40)

    var body: some View {

        NavigationView{
            VStack{
                Text("name: \(self.object1.name)")
                    .font(.system(size: 60))
                Text("age: \(self.object1.age)")
                    .font(.system(size: 60))
                Button("Change value"){
                    self.object1.age = Int.random(in: 20...45)
                }.font(.system(size: 30))
            }

        }
    }
}
    struct StateObjectDemo_Previews: PreviewProvider {
        static var previews: some View {
            StateObjectDemo()
        }
    }

```

@ObservedObject

- SwiftUI provides @ObservedObject property wrapper so that views can watch the state of an external object, and be notified when something important has changed.
- It is similar in behavior to @StateObject, except it must *not* be used to create objects – use @ObservableObject only with objects that have been created elsewhere, otherwise SwiftUI might accidentally destroy the object.

@ObservedObject – Lecture4Demo: ColorSetView

lecture8Demo › lecture8Demo › ColorSet.swift › C ColorSet

```swift
class ColorSet: ObservableObject {

    // ObservableObject
    // The 6 color components are marked as
        @Published so any changes
    // get published to the views that are
        observing

    @Published var foregroundRed = 0.0

    @Published var backgroundRed = 1.0


    // Computed variables to create the Red
        color from the components

    var foregroundColor: Color {
        return Color(red: foregroundRed,
            green: 0.0, blue: 0.0)
    }

    var backgroundColor: Color {
        return Color(red: backgroundRed,
            green: 0.0, blue: 0.0)
    }
}

```

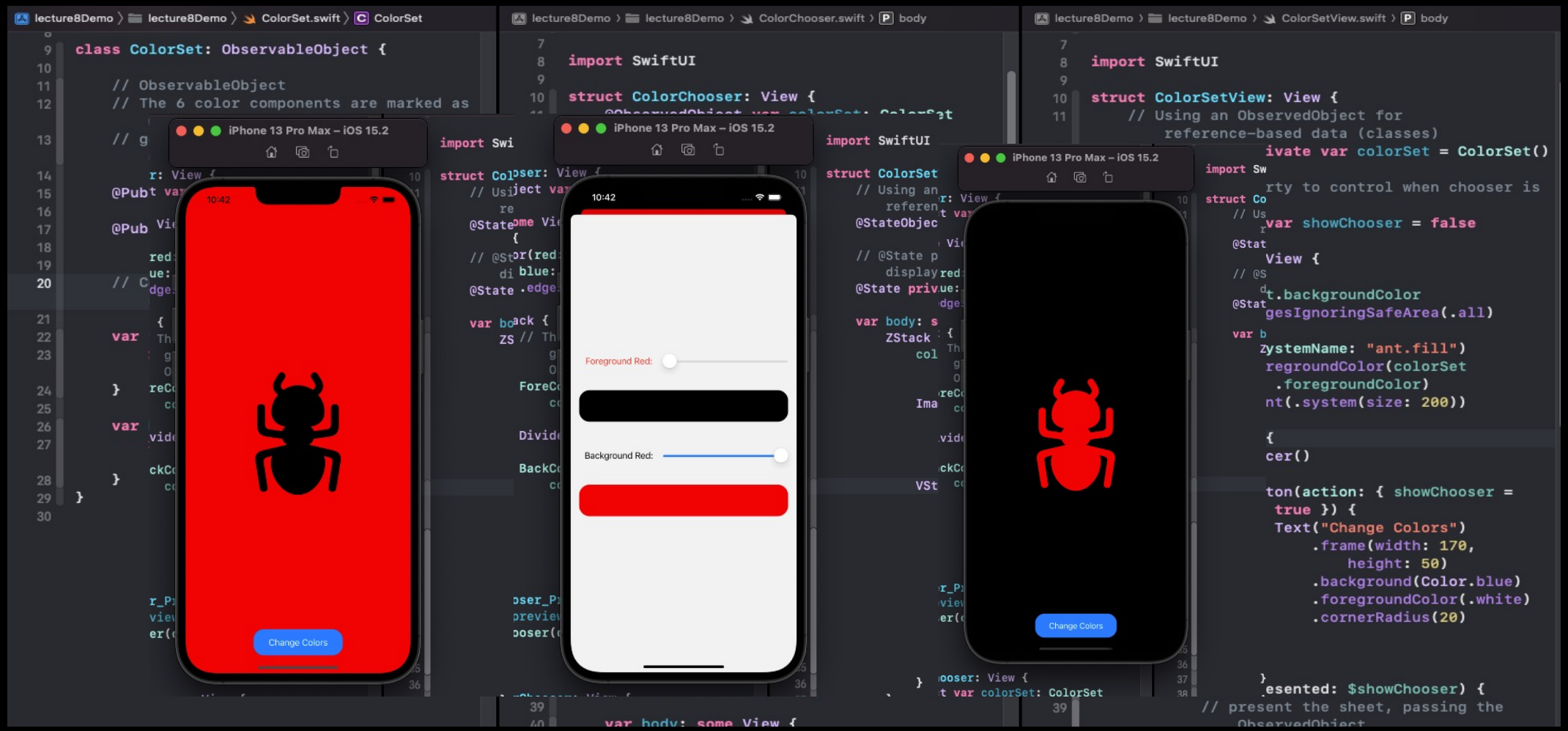
lecture8Demo › lecture8Demo › ColorChooser.swift › P body

```swift

import SwiftUI

struct ColorChooser: View {
    @ObservedObject var colorSet: ColorSet

    var body: some View {
        ZStack {
            Color(red: 0.95, green: 0.95,
                blue: 0.95)
                .edgesIgnoringSafeArea(.all)

            VStack {
                // The 2 Chooser subviews also
                    get passed the
                    ObservedObject
                ForeColorChooser(colorSet:
                    colorSet)

                Divider()

                BackColorChooser(colorSet:
                    colorSet)

            }
        }
    }
}

struct ColorChooser_Previews: PreviewProvider {
    static var previews: some View {
        ColorChooser(colorSet: ColorSet())
    }
}

struct ForeColorChooser: View {
    @ObservedObject var colorSet: ColorSet

    var body: some View {
```

lecture8Demo › lecture8Demo › ColorSetView.swift › P body

```swift

import SwiftUI

struct ColorSetView: View {
    // Using an ObservedObject for
        reference-based data (classes)
    @StateObject private var colorSet = ColorSet()

    // @State property to control when chooser is
        displayed
    @State private var showChooser = false

    var body: some View {
        ZStack {
            colorSet.backgroundColor
                .edgesIgnoringSafeArea(.all)

            Image(systemName: "ant.fill")
                .foregroundColor(colorSet
                    .foregroundColor)
                .font(.system(size: 200))

            VStack {
                Spacer()

                Button(action: { showChooser =
                    true }) {
                    Text("Change Colors")
                        .frame(width: 170,
                            height: 50)
                        .background(Color.blue)
                        .foregroundColor(.white)
                        .cornerRadius(20)
                }
            }
        }
        .sheet(isPresented: $showChooser) {
            // present the sheet, passing the
                ObservedObject
```

@ObservedObject – Lecture4Demo: ColorSetView



@EnvironmentObject

- Think of this as a global object—sometimes you might want to keep track of certain things throughout your app that you might not necessarily need or feel the need to pass through to every view.
- However, it's important to know that EnvironmentObject isn't a single source of truth; it's data—it's merely referencing it from the source, and should the source change, EnvironmentObject will trigger a state change (which is what we want).

@EnvironmentObject – demo: TimerDemo

ObservableDemo 〉 TimerData.swift 〉 No Selection

```swift

import Foundation

class TimerData : ObservableObject {

    @Published var timeCount = 0
    var timer : Timer?

    init() {
        timer =
            Timer
            .scheduledTimer(timeInterval:
            1.0, target: self, selector:
            #selector(timerDidFire),
            userInfo: nil, repeats: true)
    }

    @objc func timerDidFire() {
        timeCount += 1
    }

    func resetCount() {
        timeCount = 0
    }
}

```

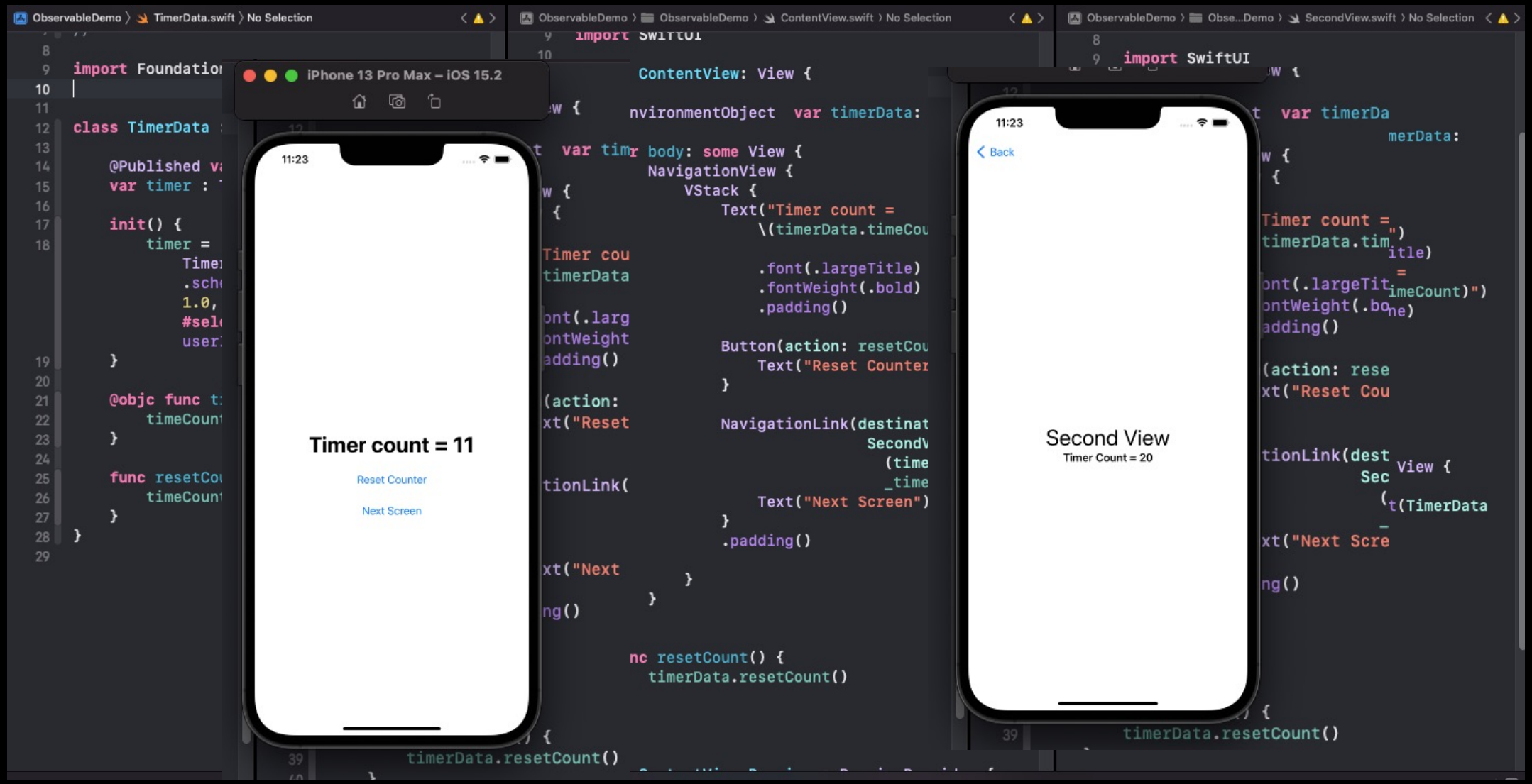
ObservableDemo 〉 ObservableDemo 〉 ContentView.swift 〉 No Selection

```swift
import SwiftUI

struct ContentView: View {

    @EnvironmentObject  var timerData: TimerData

    var body: some View {
        NavigationView {
            VStack {
                Text("Timer count =
                    \(timerData.timeCount)")

                    .font(.largeTitle)
                    .fontWeight(.bold)
                    .padding()

                Button(action: resetCount) {
                    Text("Reset Counter")
                }

                NavigationLink(destination:
                                SecondView
                                  (timerData:
                                  _timerData)) {
                    Text("Next Screen")
                }
                .padding()

            }
        }
    }

    func resetCount() {
        timerData.resetCount()
    }

}

```

ObservableDemo 〉 Obse...Demo 〉 SecondView.swift 〉 No Selection

```swift

import SwiftUI

struct SecondView: View {

    @EnvironmentObject var timerData:
        TimerData

    var body: some View {
        VStack {
            Text("Second View")
                .font(.largeTitle)
            Text("Timer Count =
                \(timerData.timeCount)")
                .font(.headline)
        }.padding()

    }
}

struct SecondView_Previews:
    PreviewProvider {
    static var previews: some View {
        SecondView()
            .environmentObject(TimerData
            ())
    }
}

```

@EnvironmentObject - demo



Sources of truth – Paul Hudson

- Truth:
- @Published
- @State
- @StateObject
- Secondary sources for data:
- @Binding
- @Environment, @EnvironmentObject
- @ObservedObject
- Another good source for deeper discussion - TrozWare

Comparison @StateObject v @ObservedObjec

```
class User: ObservableObject {
    @Published var name = "John Doe"
}

struct ContentView: View {
    @ObservedObject private var user: User

    var body: some View {
        Text("Hello, \(user.name)")
    }
}

struct RootView: View {
    @State private var user = User()

    var body: some View {
        ContentView(user: user)
    }
}
```

```
class User: ObservableObject {
    @Published var name = "John Doe"
}

struct ContentView: View {
    @StateObject private var user = User()

    var body: some View {
        Text("Hello, \(user.name)")
    }
}
```

How to choose

- @StateObject is used to create a single instance of an object that is shared among multiple views. It's initialized when the view is first created, and it remains in memory until the parent view is destroyed. The main advantage of @StateObject is that it's fast and efficient, since it's only created once and can be shared among multiple views.
- @ObservedObject is used to track changes to an object that's passed in as a dependency. It's similar to @StateObject, but it allows you to create an instance of the object outside of the view and pass it in as a dependency. This is useful when you want to share an instance of an object between multiple views, but you don't want to keep it in memory when the views are no longer in use.

Summary - Data flow mechanism options