



# Introduction to SwiftUI Data Wrappers

Girish Lukka

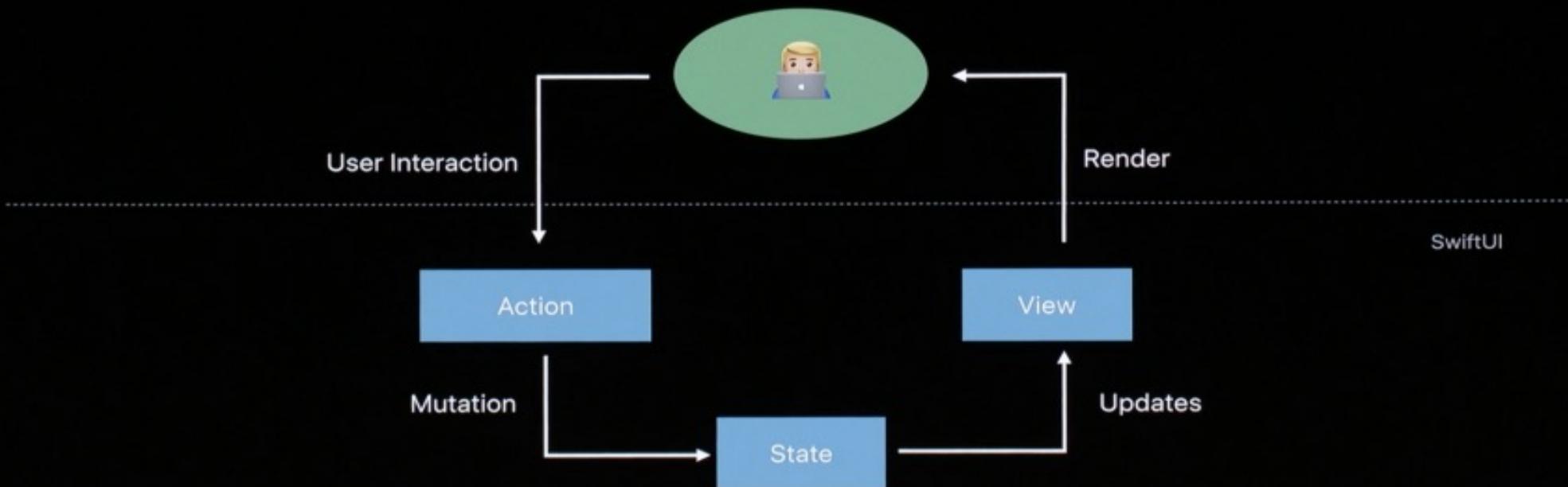
# Topics

- 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

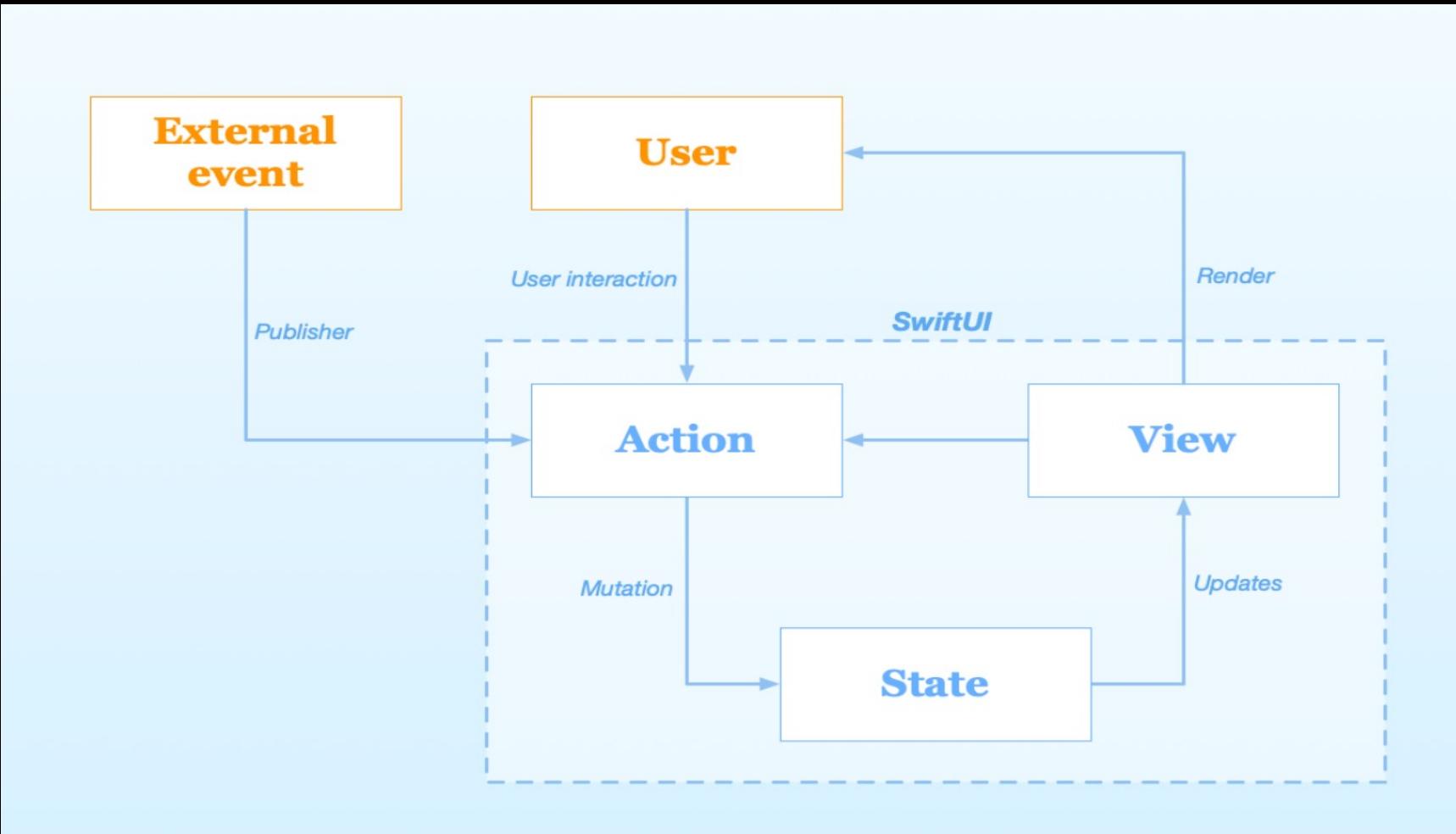
# Overview

- 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.

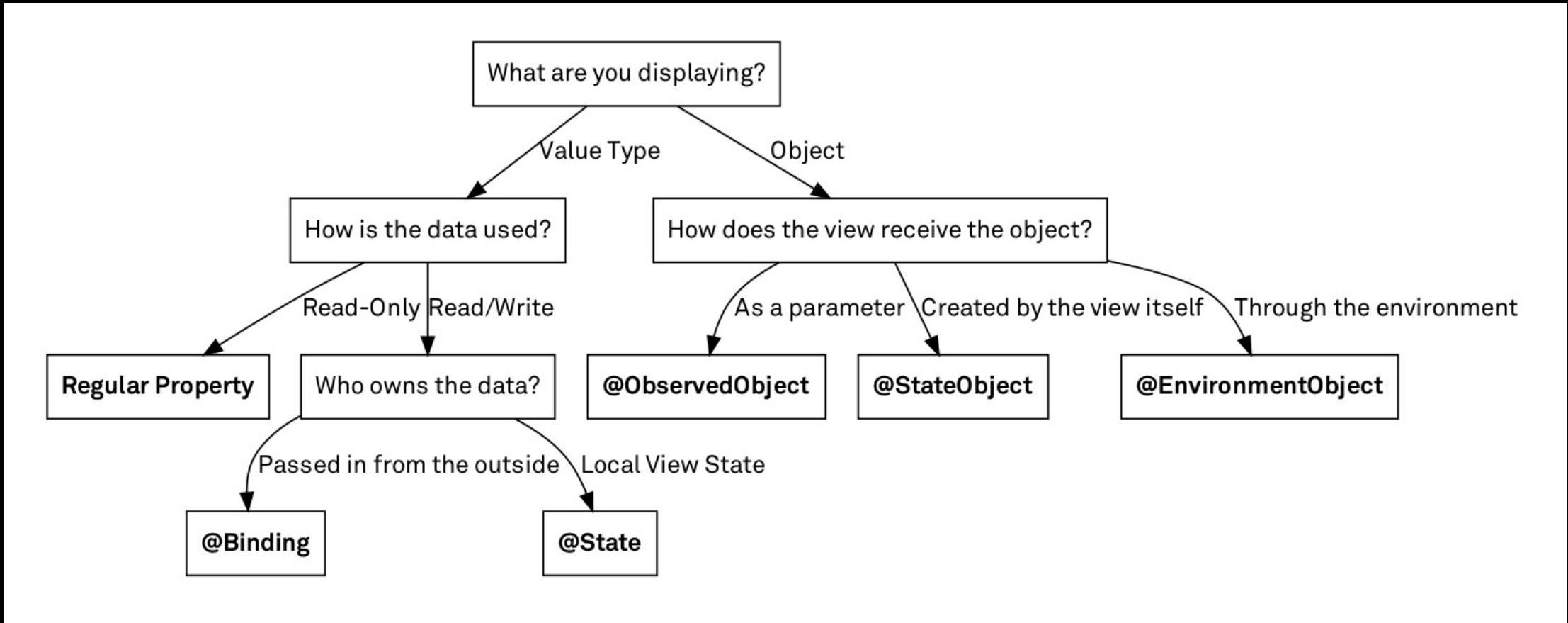
# SwiftUI Data Flow - 2019 WWDC



# SwiftUI Data Flow

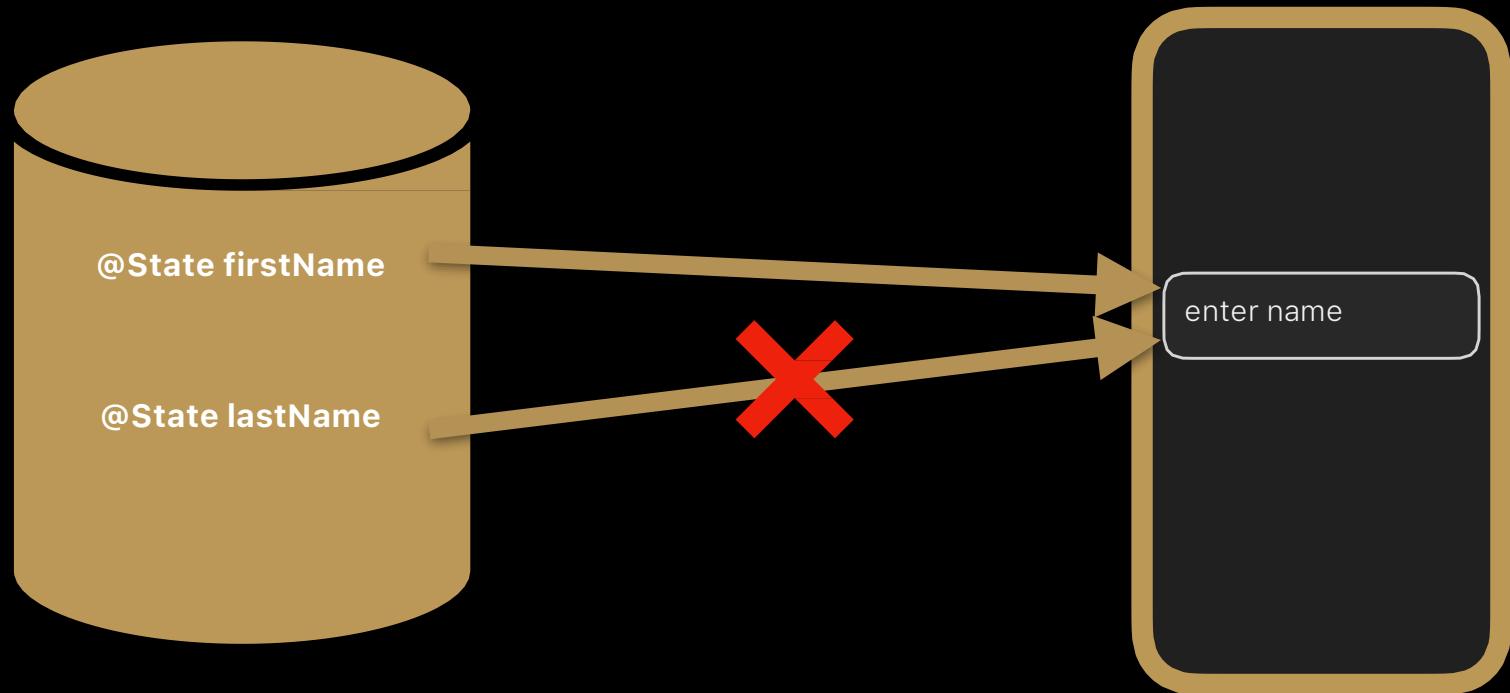


# Common Property Wrappers



# Single Source of Truth

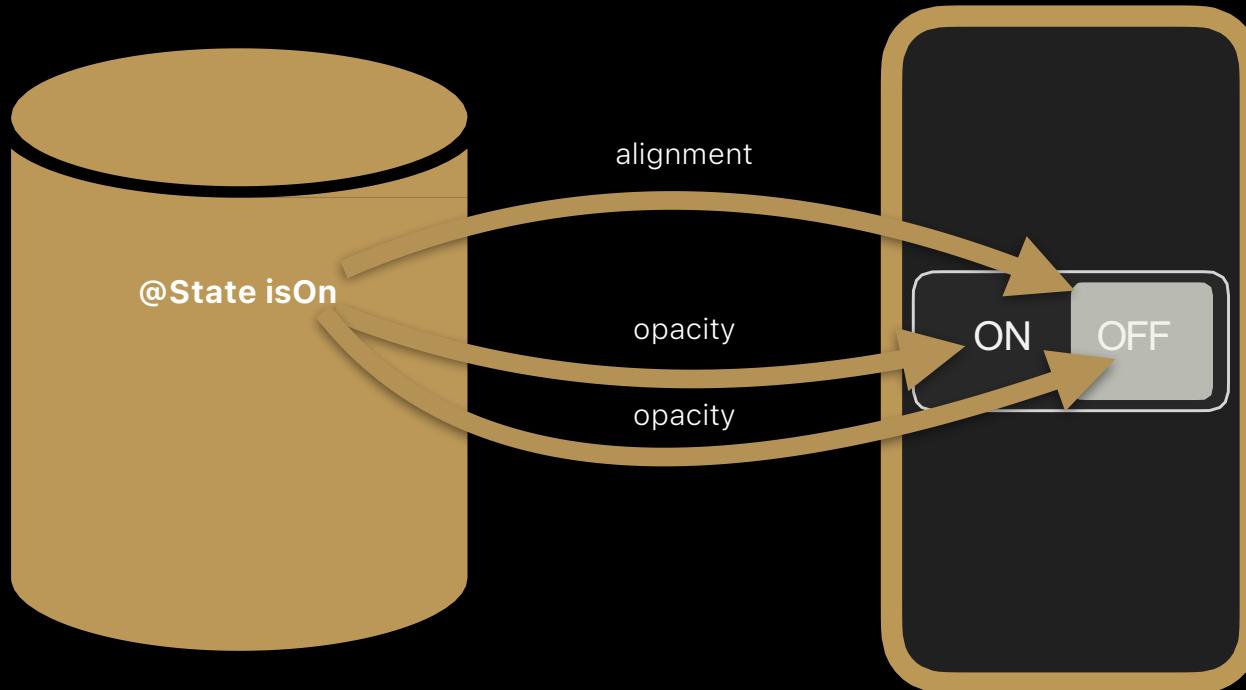
You cannot bind a UI control to more than one piece of data.



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

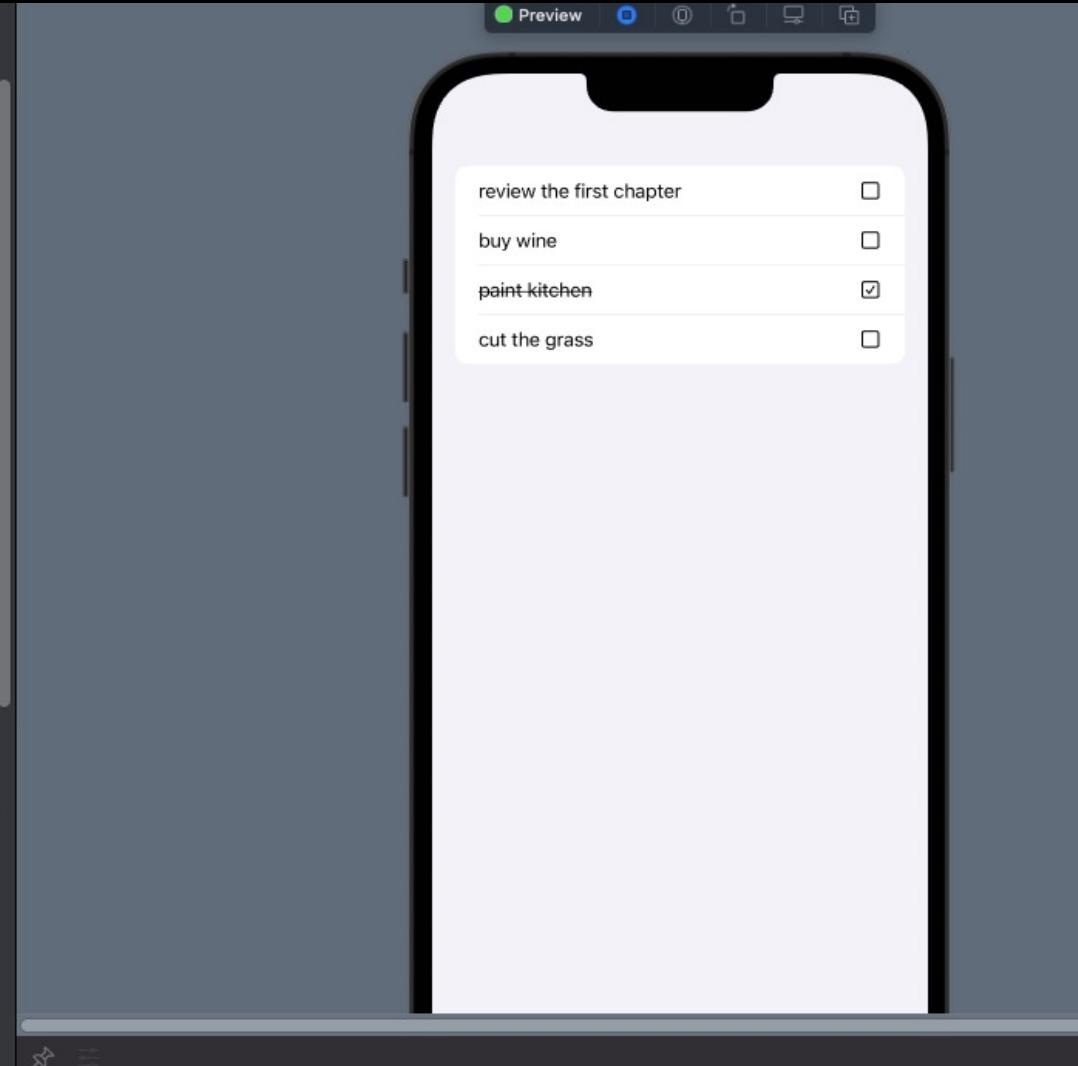
- All views are functions of their state.
- Apple documentation:
  - **SwiftUI manages the storage of any property you declare as a state.**
  - When the state value changes, the view invalidates its appearance and recomputes the body.
  - Use the state as the single source of truth for a given view.
  - 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

```
8 import SwiftUI
9
10 struct Todo: Identifiable {
11     let id = UUID()
12     let description: String
13     var done: Bool
14 }
15
16 struct ContentView: View {
17     @State
18     private var todos = [
19         Todo(description: "review the first chapter", done: false),
20         Todo(description: "buy wine", done: false),
21         Todo(description: "paint kitchen", done: false),
22         Todo(description: "cut the grass", done: false),
23     ]
24
25     var body: some View {
26         List($todos) { $todo in
27             HStack {
28                 Text(todo.description)
29                     .strikethrough(todo.done)
30
31
32                 Spacer()
33                 Image(systemName:
34                     todo.done ?
35                         "checkmark.square" :
36                         "square")
37             }
38             .contentShape(Rectangle())
39             .onTapGesture {
40                 todo.done.toggle()
41             }
42         }
43     }
44 }
```

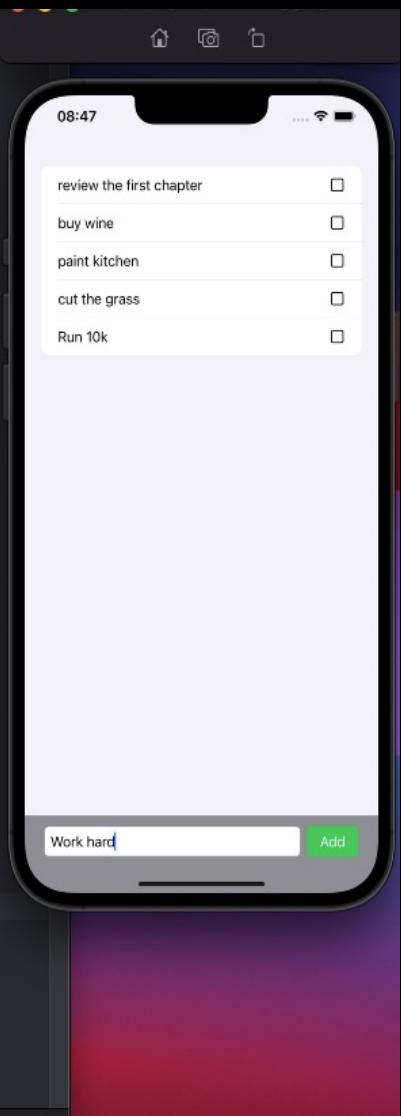


# @Binding

- 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

```
8 import SwiftUI
9
10 struct InputTodoView: View {
11     @State
12     private var newTodoDescription: String = ""
13
14     @Binding
15     var todos: [Todo]
16
17     var body: some View {
18         HStack {
19             TextField("Todo", text: $newTodoDescription)
20                 .textFieldStyle(RoundedBorderTextFieldStyle())
21             Spacer()
22             Button {
23                 guard !newTodoDescription.isEmpty else {
24                     return
25                 }
26                 todos.append(Todo(description: newTodoDescription,
27                                     done: false))
28                 newTodoDescription = ""
29             } label: {
30                 Text("Add")
31                     .padding(.horizontal, 16)
32                     .padding(.vertical, 8)
33                     .foregroundColor(.white)
34                     .background(.green)
35                     .cornerRadius(5)
36             }
37         }
38         .frame(height: 60)
39         .padding(.horizontal, 24)
40         .padding(.bottom, 30)
41         .background(Color.gray)
42     }
43 }
44
45 struct Todo: Identifiable {
46     let id = UUID()
47     let description: String
48     var done: Bool
49 }
50
51 struct ContentView: View {
52     @State
53     private var todos = [
54         Todo(description: "review the first chapter", done: false),
55         Todo(description: "buy wine", done: false),
56         Todo(description: "paint kitchen", done: false),
57         Todo(description: "cut the grass", done: false),
58     ]
59
60     var body: some View {
61         ZStack(alignment: .bottom) {
62             List($todos) { $todo in
63                 HStack {
64                     Text(todo.description)
65                         .strikethrough(todo.done)
66                     Spacer()
67                     Image(systemName:
68                             todo.done ?
69                             "checkmark.square" :
70                             "square")
71                 }
72                 .contentShape(Rectangle())
73                 .onTapGesture {
74                     todo.done.toggle()
75                 }
76             }
77             InputTodoView(todos: $todos)
78         }
79     }
80 }
81
82 struct ContentView_Previews: PreviewProvider {
83     static var previews: some View {
84 }
```



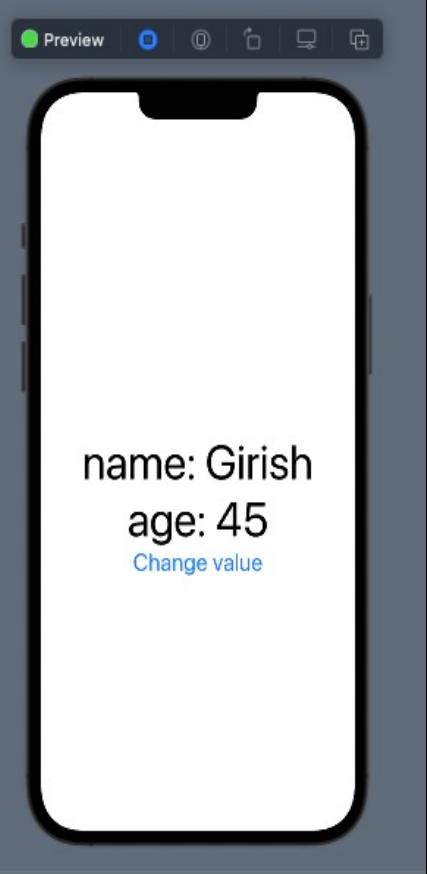
# @Published – see class demo from week2

- **@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



```
5 // Created by Girish Lukka on
6 // 17/03/2022.
7
8 import Foundation
9 import Foundation
10
11 class myObject: ObservableObject{
12     var name: String
13     @Published var age: Int
14
15     init(name:String,age:Int){
16         self.name = name
17         self.age = age
18     }
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

```
5 // Created by Girish Lukka on 17/03/2022.
6 //
7
8 import SwiftUI
9
10 struct StateObjectDemo: View {
11     @StateObject var object1 = myObject(name: "Girish", age: 40)
12
13     var body: some View {
14
15         NavigationView{
16             VStack{
17                 Text("name: \(self.object1.name)")
18                     .font(.system(size: 60))
19                 Text("age: \(self.object1.age)")
20                     .font(.system(size: 60))
21                 Button("Change value"){
22                     self.object1.age = Int.random(in: 20...45)
23                 }.font(.system(size: 30))
24             }
25         }
26     }
27 }
28
29
30 struct StateObjectDemo_Previews: PreviewProvider {
31     static var previews: some View {
32         StateObjectDemo()
33     }
34 }
```

# @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

The image shows three side-by-side Xcode code editors. Each editor has a dark background with light-colored text. The first editor (ColorSet.swift) contains a class definition for ColorSet that implements ObservableObject. It includes @Published properties for foregroundRed and backgroundRed, and computed properties for foregroundColor and backgroundColor. The second editor (ColorChooser.swift) contains a struct definition for ColorChooser that uses @ObservedObject to reference a ColorSet instance. It includes a body view that displays a ZStack with a color swatch and a VStack containing two Chooser subviews: ForeColorChooser and BackColorChooser. The third editor (ColorSetView.swift) contains a struct definition for ColorSetView that uses @StateObject to reference a ColorSet instance. It includes a body view that displays a ZStack with a color swatch, an Image view, and a Button that triggers a sheet presentation.

```
lecture8Demo/lecture8Demo/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
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
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

```
lecture8Demo > lecture8Demo > ColorSet.swift > ColorSet
9 class ColorSet: ObservableObject {
10
11     // ObservableObject
12     // The 6 color components are marked as
13     // @Published
14     var red: Double = 0.0
15     var green: Double = 0.0
16     var blue: Double = 0.0
17     var alpha: Double = 1.0
18
19     // Colors
20     var foregroundColor: Color {
21         get { Color(red: red, green: green, blue: blue) }
22         set { self.red = newValue.red
23               self.green = newValue.green
24               self.blue = newValue.blue
25         }
26     }
27
28     var backgroundColor: Color {
29         get { Color(red: red, green: green, blue: blue, opacity: alpha) }
30         set { self.alpha = newValue.opacity
31               self.red = newValue.red
32               self.green = newValue.green
33               self.blue = newValue.blue
34         }
35     }
36
37     var body: some View {
38         ZStack {
39             Image("ant")
40             .colorMultiply(foregroundColor)
41
42             Text("Change Colors")
43         }
44     }
45
46     func changeColor() {
47         self.backgroundColor = self.foregroundColor
48     }
49 }
```

```
lecture8Demo > lecture8Demo > ColorChooser.swift > body
7 import SwiftUI
8
9 struct ColorChooser: View {
10     @ObservedObject var colorSet: ColorSet
11
12     var body: some View {
13         Form {
14             Section("Foreground") {
15                 Text("Foreground Red:")
16                 Slider(value: $colorSet.red)
17             }
18
19             Section("Background") {
20                 Text("Background Red:")
21                 Slider(value: $colorSet.alpha)
22             }
23         }
24     }
25 }
```

```
lecture8Demo > lecture8Demo > ColorSetView.swift > body
7 import SwiftUI
8
9 struct ColorSetView: View {
10     @ObservedObject var colorSet: ColorSet
11
12     var body: some View {
13         VStack {
14             Text("Change Colors")
15             .frame(width: 170, height: 50)
16             .background(Color.blue)
17             .foregroundColor(.white)
18             .cornerRadius(20)
19
20             ColorChooser()
21
22             Text("Presented: $showChooser")
23         }
24     }
25 }
```

# @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

The screenshot shows three Swift files in Xcode:

- TimerData.swift**: A class `TimerData` that conforms to `ObservableObject`. It has a published property `timeCount` and a private variable `timer`. The `init()` method starts a timer that fires every 1.0 second, calling the `timerDidFire` selector. The `resetCount` method sets `timeCount` back to 0.
- ContentView.swift**: A `struct` that is a `View`. It uses `@EnvironmentObject` to get an instance of `TimerData`. The view contains a `NavigationView` with a `VStack` that displays the current value of `timeCount` and a `Button` to reset it. It also contains a `NavigationLink` to the `SecondView`.
- SecondView.swift**: A `struct` that is a `View`. It uses `@EnvironmentObject` to get an instance of `TimerData`. The view displays the text "Second View" and the current value of `timeCount`.

```
ObservableDemo > TimerData.swift > No Selection
8
9 import Foundation
10
11 class TimerData : ObservableObject {
12     @Published var timeCount = 0
13     var timer : Timer?
14
15     init() {
16         timer =
17             Timer.scheduledTimer(timeInterval: 1.0, target: self, selector: #selector(timerDidFire), userInfo: nil, repeats: true)
18     }
19
20     @objc func timerDidFire() {
21         timeCount += 1
22     }
23
24     func resetCount() {
25         timeCount = 0
26     }
27 }
28
29

ObservableDemo > ObservableDemo > ContentView.swift > No Selection
9 import SwiftUI
10
11 struct ContentView: View {
12
13     @EnvironmentObject var timerData: TimerData
14
15     var body: some View {
16         NavigationView {
17             VStack {
18                 Text("Timer count = \n\((timerData.timeCount))")
19                     .font(.largeTitle)
20                     .fontWeight(.bold)
21                     .padding()
22
23                 Button(action: resetCount) {
24                     Text("Reset Counter")
25                 }
26
27                 NavigationLink(destination: SecondView(timerData: _timerData)) {
28                     Text("Next Screen")
29                 }
30                 .padding()
31
32             }
33         }
34     }
35
36     func resetCount() {
37         timerData.resetCount()
38     }
39
40 }
41
42

ObservableDemo > ObservableDemo > SecondView.swift > No Selection
8
9 import SwiftUI
10
11 struct SecondView: View {
12
13     @EnvironmentObject var timerData: TimerData
14
15     var body: some View {
16         VStack {
17             Text("Second View")
18                 .font(.largeTitle)
19             Text("Timer Count = \n\((timerData.timeCount))")
20                 .font(.headline)
21         }
22     }
23
24
25
26 struct SecondView_Previews: PreviewProvider {
27     static var previews: some View {
28         SecondView()
29             .environmentObject(TimerData())
30     }
31 }
```

# @EnvironmentObject - demo

The image shows a Xcode interface with three open files and their corresponding simulators:

- TimerData.swift**: A Swift class that publishes a timer value. It has an `@Published var timer` property and a `resetCount()` function.
- ContentView.swift**: A SwiftUI view that displays the timer count and provides buttons to reset it or navigate to the next screen.
- SecondView.swift**: A SwiftUI view that also displays the timer count and receives updates from the environment object.

The simulators show the following states:

- TimerData.swift Simulator**: Shows "Timer count = 11".
- ContentView.swift Simulator**: Shows "Timer count = 11" and two buttons: "Reset Counter" and "Next Screen".
- SecondView.swift Simulator**: Shows "Second View" and "Timer Count = 20".

The code snippets are as follows:

```
TimerData.swift:
```

```
import Foundation
import SwiftUI

class TimerData: ObservableObject {
    @Published var timer: Int = 0
    
    init() {
        timer = 0
        Timer.scheduledTimer(timeInterval: 1.0, target: self, selector: #selector(updateTime), userInfo: nil, repeats: true)
    }
    
    @objc func updateTime() {
        timeCount()
    }
    
    func resetCount() {
        timeCount()
    }
}

func timeCount() {
    timer += 1
}
```

```
ContentView.swift:
```

```
import SwiftUI

struct ContentView: View {
    @EnvironmentObject var timerData: TimerData
    
    var body: some View {
        NavigationView {
            VStack {
                Text("Timer count = \n \(timerData.timeCount)")
                    .font(.largeTitle)
                    .fontWeight(.bold)
                    .padding()
                
                Button(action: resetCount) {
                    Text("Reset Counter")
                }
                
                Text("Reset")
                    .onTapGesture {
                        withAnimation {
                            resetCount()
                        }
                    }
                
                NavigationLink(destination: SecondView(timer: timerData)) {
                    Text("Next Screen")
                }
            }
        }
    }
    
    func resetCount() {
        timerData.resetCount()
    }
}
```

```
SecondView.swift:
```

```
import SwiftUI

struct SecondView: View {
    @EnvironmentObject var timerData: TimerData
    
    var body: some View {
        Text("Second View\n Timer Count = \n \(timerData.timeCount)")
    }
}
```

# 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

