SwiftUI State Management Quick Guide

A simple everyday reference for interviews & real projects

SwiftUI re-renders views when data changes. These keywords tell SwiftUI **who owns the data** and **who can change it**.

@State

Purpose: Local state owned by a single view

Use for: Small values like toggles, counters, text inputs

Example:

```
struct CounterView: View {
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Count: \(count)")
            Button("Increase") { count += 1 }
        }
    }
}
```

Memory tip: View's private notebook

@Binding

Purpose: Child view can edit parent view's state

Use for: Passing edit control down the view tree

Example:

```
struct Parent: View {
    @State private var name = "John"
    var body: some View { Child(name: $name) }
}

struct Child: View {
    @Binding var name: String
```

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

Memory tip: Remote control to someone else's state

@StateObject

Purpose: Create and **own** an ObservableObject inside a view

Simple explanation

SwiftUI sometimes **recreates views**, but you don't want your data object recreated. `@StateObject` makes sure your object **lives as long as the view does**, even when the view redraws.

When to use

- You **create** the model here
- The object must persist for the life of the view

Real-life analogy

> You bought the laptop (data model) — even if you move desks (view reload), you still keep the same laptop.

Example:

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

struct ProfileView: View {
    @StateObject var user = UserData()  // created & owned here

    var body: some View { Text(user.name) }
}
```

Common mistake: Don't use `@StateObject` when passing a model **from parent to child** — use `@ObservedObject` instead.

Memory tip: "I **own** this model"

@ObservedObject

Purpose: View watches an already-created object

Use for: Data passed from parent view

Example:

```
struct DetailView: View {
    @ObservedObject var user: UserData
    var body: some View { Text(user.name) }
}
```

Memory tip: "I am **watching** someone else's model"

@EnvironmentObject

Purpose: Share data across many views **without passing it manually**

Use for: App-wide/shared data (settings, user session)

Injection point:

```
@main
struct MyApp: App {
    @StateObject var user = UserData()

    var body: some Scene {
        WindowGroup {
            Home().environmentObject(user)
        }
    }
}
```

Usage:

```
struct Home: View {
    @EnvironmentObject var user: UserData
    var body: some View { Text(user.name) }
}
```

Memory tip: "App-wide shared model"

ObservableObject & @Published

ObservableObject: A class that can notify SwiftUI when data changes

@Published: Marks properties that trigger UI updates

```
class Settings: ObservableObject {
    @Published var isDarkMode = false
}
```

Memory tip: Shared Google Doc — everyone sees updates

Quick Cheat Sheet

Use Case	Keyword
Local view state	`@State`
Child edits parent state	`@Binding`
Create & own shared object	`@StateObject`
Receive & observe shared object	`@ObservedObject`
App-wide shared object	`@EnvironmentObject`
Notify UI of changes	`@Published`
Class that publishes change events	`ObservableObject`

One-sentence summaries

- **@State** — the view's private value storage
- **@Binding** — two-way connection to someone's state
- **ObservableObject** — a sharable class model
- **@Published** — "refresh UI when this changes"
- **@StateObject** — view **creates & owns** the model
- **@ObservedObject** — view **uses someone else's** model
- **@EnvironmentObject** — model shared across many screens

Use this as a quick daily reference — keep solidifying pairings:

Create = `@StateObject`, Receive = `@ObservedObject`, Share = `@EnvironmentObject`

Visual Folder Analogy (Easy Memory)

- **@State** = Sticky note on your desk (local, temporary)
- **@Binding** = Borrowing a sticky note from a friend
- **@StateObject** = Your laptop you bought (persists even if you move desks)
- **@ObservedObject** = Using a friend's laptop (you don't own it)
- **@EnvironmentObject** = Company shared server

- **ObservableObject** = File folder that alerts everyone when changed
- **@Published** = Notifications sent when file updates

Interview Q&A

Q: Difference between @StateObject and @ObservedObject? - `@StateObject` **creates & owns** the object - `@ObservedObject` **receives & watches** an existing object

Q: Why do we need @Published? - Without `@Published`, SwiftUI won't know to update when values change

Q: What happens if @EnvironmentObject isn't provided? - App **crashes at runtime**

Q: When would you use @Binding instead of @ObservedObject? - For **simple values** from a parent (String, Bool, Int) - `@ObservedObject` is for **complex shared models**

Memory Tricks

- **State = Small & Local**
- **StateObject = Create Model**
- **ObservedObject = Receive Model**
- **Binding = Two-way value link**
- **EnvironmentObject = Global shared data**

Mnemonic:

> **C R S G** = Create, Receive, Simple, Global

Real App Example (Simple Notes App)

```
class NotesModel: ObservableObject {
    @Published var notes: [String] = []
}

struct NotesAppView: View {
    @StateObject private var model =
NotesModel()   // Own model

    var body: some View {
        VStack {
            List(model.notes, id: \.self) {
Text($0) }
            NewNoteEntry(notes:
$model.notes)        // Pass binding
        }
    }
}

struct NewNoteEntry: View {
    @Binding var notes: [String]
    @State private var text =
""                   // Local field

    var body: some View {
        HStack {
            TextField("Note", text: $text)
            Button("Add") {
                notes.append(text)
                text = ""
            }
        }
        .padding()
```

```
    }
}
```

iOS 17+: `@Observable` & `@Bindable` (Observation)

What it is: A modern replacement for `ObservableObject` + `@Published`. - No `@Published` needed. Change notifications are auto-generated. - **Fine-grained updates:** Views only re-render for properties they read. - Works great with `@State` for ownership and `@Bindable` for passing editable models.

Core model

```
import Observation

@Observable
class Cart {
    var items: [String] = []
    var taxRate: Double = 0.12
    var subtotal: Double { items.isEmpty ? 0 : Double(items.count) * 10 }
    var total: Double { subtotal * (1 + taxRate) }
}
```

Owning the model (parent)

With Observation, you usually own the model with `@State`, not `@StateObject`.

```
struct CartScreen: View {
    @State private var cart = Cart()   // Owns lifetime of the
model

    var body: some View {
        VStack(spacing: 12) {
            Text("Items: \(cart.items.count)")  // reading tracks
dependency
            Text("Total: \(cart.total, format: .currency(code:
"USD"))")
            Button("Add Item") { cart.items.append("Apple") } //
mutation triggers update
            CartEditor(cart: cart) // pass model to child
        }.padding()
    }
}
```

Editing in a child with `@Bindable`

`@Bindable` exposes **bindings** to the model's properties via the `$` prefix.

```
struct CartEditor: View {
    @Bindable var cart: Cart   // not owned here; just editing

    var body: some View {
        VStack {
            Stepper("Tax: \(cart.taxRate, format: .percent)",
value: $cart.taxRate, in: 0...0.25, step: 0.01)
            Button("Remove All") { cart.items.removeAll() }
        }
    }
}
```

Binding straight to a field (no wrapper objects)

Because `@Observable` tracks access, you can bind **directly** to properties:

```
TextField("Promo Code", text: $cart.promoCode)
```

(Assuming `var promoCode: String = ""` *exists in* `Cart`*.)*

Ignore properties from observation

```
@Observable
class ImageLoader {
    var url: URL
    @ObservationIgnored var cache = NSCache<NSURL, UIImage>() // won't
trigger updates
    init(url: URL) { self.url = url }
}
```

When to use Observation vs. Combine models

- **Target iOS 17+/macOS 14+?** Prefer `@Observable` + `@Bindable`.
- **Need iOS 16 or earlier?** Use `ObservableObject` + `@Published`, `@StateObject`, `@ObservedObject`.

Migration cheat sheet

Old (Combine)	New (Observation)	Notes
`class Model: ObservableObject { @Published var x }`	`@Observable class Model { var x }`	Drop `@Published`
`@StateObject var model = Model()`	`@State private var model = Model()`	Ownership stays with the view

Old (Combine)	New (Observation)	Notes
`@ObservedObject var model: Model`	`var model: Model` or `@Bindable var model: Model`	Use `@Bindable` if you need to **edit**
`Binding(get:set:)` to model fields	`$model.x`	Direct bindings
`@Published private(set) var x` + method updates	`private(set) var x` + methods	Access control works the same

Interview talking points

- Observation uses **access tracking** to update only views that **read** a changing property.
- `@State` is still for **ownership** in the view; `@Bindable` is for **editing** in children.
- No crash equivalent of missing `@EnvironmentObject` here, but you can still inject models via environment if desired.
- You can mix, but avoid putting **both** `ObservableObject` and `@Observable` in the same model.
- For heavy async/UI changes, prefer marking mutating methods `@MainActor`.

Mini end-to-end example

```
import Observation

@Observable
class Settings {
    var username: String = ""
    var isPro: Bool = false
}

struct Root: View {
    @State private var settings = Settings()

    var body: some View {
        VStack(spacing: 10) {
            Text("Hello, \(settings.username.isEmpty ? "Guest" :
settings.username)")
            SettingsEditor(settings: settings)    // child can edit
            if settings.isPro { Text("⭐ Pro features unlocked") }
        }.padding()
    }
}

struct SettingsEditor: View {
    @Bindable var settings: Settings
    var body: some View {
        VStack {
            TextField("Username", text: $settings.username)
            Toggle("Pro", isOn: $settings.isPro)
```

```
        }
    }
}
```

Rule of thumb: - **Own with** `@State` (parent) - **Edit with** `@Bindable` (child) - **Model with** `@Observable` (iOS 17+)