Swift Concurrency & SwiftUI Attribute Reference

This document explains modern Swift attributes you referenced: @Observable, @State, @MainActor, @Environment, .shared, and related concepts. It provides definitions, usage patterns, memory/lifecycle behavior, and examples.

@Observable (Swift Observation Framework)

Summary

@Observable is a Swift macro introduced to replace/modernize the older ObservableObject protocol in SwiftUI. It automatically synthesizes the observation logic for model types.

Key Points

- Part of Swift's new Observation system
- Eliminates @Published for basic use cases
- Works across SwiftUI and non-UI contexts
- Automatically tracks property changes and notifies observers

Example

```swift
@Observable
class CounterModel {
    var value: Int = 0
}
```

Use in SwiftUI:

```swift
struct CounterView: View {
    @State var model = CounterModel()

    var body: some View {
        VStack {
            Text("\(model.value)")
            Button("Increment") { model.value += 1 }
        }
    }
}
```

When to Use

- App state models
- Shared observable logic
- Replaces @StateObject / @ObservedObject in many cases

@State

Summary

@State stores local, view-owned state in SwiftUI.

Key Points

- Value-type state owned by a view
- Stored outside of struct lifecycle
- Best used for local UI state

Example

```
struct ToggleView: View {
    @State private var isOn = false

    var body: some View {
        Toggle("Enabled", isOn: $isOn)
    }
}
```

When to Use

- Transient UI state
- Local flags, counters, editing fields

@MainActor

Summary

Ensures execution on the main thread/actor. Required for UI updates.

Key Points

- Guarantees thread-safe access for UI-related state
- Can annotate functions, properties, or entire types

Example

```
@MainActor
class UserViewModel {
    var name: String = ""

    func updateName(_ new: String) {
        name = new
    }
}
```

When to Use

- UI logic
- Shared state accessed from tasks
- ViewModels interacting with SwiftUI

@Environment

Summary

Injects environment values provided by SwiftUI (system or custom).

Key Points

- For dependency injection in views
- Includes system values (dismiss, colorScheme, etc.)
- Works with custom environment keys

Example

```
struct ProfileView: View {
    @Environment(\.dismiss) var dismiss

    var body: some View {
        Button("Close") { dismiss() }
    }
}
```

Custom Keys

```
private struct ThemeKey: EnvironmentKey {
    static let defaultValue = Color.blue
}
extension EnvironmentValues {
    var theme: Color {
        get { self[ThemeKey.self] }
        set { self[ThemeKey.self] = newValue }
    }
}
```

.shared

Summary

.shared is not an attribute but a common Swift singleton-access pattern.

Typical Pattern

```
final class AuthManager {
    static let shared = AuthManager()
    private init() {}

    func login() {}
}
```

Used like:

```
AuthManager.shared.login()
```

When to Use

- Global services
- Network clients
- Cache managers
- Session/identity systems

Notes

- Good for system-level services
- Avoid overusing; consider dependency injection in modular apps

Attribute Comparison Table

Feature	Scope	Ownership	Lifecycle	Typical Use
@Observable	Model layer	Class/struct	Persistent/shared	Global/state models
@State	View only	SwiftUI view struct	Recreated view, stable storage	UI local state
@Environment	View injection	Framework-managed	Inherited from parent	Dependencies/settings
@MainActor	Execution context	Global actor	App lifetime	UI thread enforcement
.shared	Global singleton	Static instance	App lifetime	Services & managers

Example Architecture

```swift
@Observable
class SessionModel {
    var user: User? = nil
}

@MainActor
class AuthService {
    static let shared = AuthService()

    func signIn() async {
        // network
    }
}

struct LoginView: View {
    @Environment(SessionModel.self) var session
    @State private var username = ""

    var body: some View {
        VStack {
            TextField("Username", text: $username)
            Button("Login") {
                Task {
                    await AuthService.shared.signIn()
                }
            }
        }
    }
}
```

Best Practices

Do

- Use @Observable for app models
- Keep @State small/local
- Mark UI-related logic with @MainActor
- Prefer dependency injection over global .shared when scaling

Avoid

- Overusing singletons
- Mixing UI state and business logic
- Updating UI from background tasks without @MainActor

Additional SwiftUI State & Data Flow System

@StateObject

Used for reference-type observable objects **created by the View**.

```
@StateObject var vm = LoginViewModel()
```

- Persistent across View reloads
- Use when the View **owns the lifecycle** of the model

@ObservedObject

Used for reference-type observable objects **passed into the View**.

```
struct DashboardView: View {
    @ObservedObject var vm: DashboardViewModel
```

- Does **not** persist on view rebuild
- Use when parent owns the ViewModel

@EnvironmentObject

Global dependency injection for shared observable objects.

```
@EnvironmentObject var session: SessionStore
```

Defined at app root:

```
.rootView.environmentObject(SessionStore())
```

- Ideal for **session, settings, navigation, theme**

@Binding

Two-way binding between parent and child views.

```
struct InputField: View {
    @Binding var text: String
```

Used like:

```
InputField(text: $username)
```

@AppStorage

Automatic persistence backed by UserDefaults.

```
@AppStorage("themeMode") var themeMode: String = "light"
```

- Re-renders when value changes
- Lightweight persistent settings

@SceneStorage

State restoration between app scenes (like activity windows or navigation restarts).

```
@SceneStorage("selectedTab") var tab = 0
```

Swift Concurrency Attributes

@Sendable

Ensures closure values are thread-safe when crossing concurrency boundaries.

```
func load(@Sendable work: () async -> Void) {}
```

@unchecked Sendable

Used to manually assert safety when the compiler can't verify.

Use very rarely and only with deep understanding.

@globalActor

Declares a global actor for serialized access domain-wide.

```
@globalActor actor NetworkActor {}
```

@TaskLocal

Thread-local-like storage for async tasks.

```
@TaskLocal static var requestId: String
```

Modern vs Legacy Mapping

Old	Modern	Notes
ObservableObject	@Observable	New macro system
@Published	Implicit in @Observable	No need to mark each property
@StateObject / @ObservedObject	Still used	But less often with new observation
EnvironmentObject	@Environment(SomeType.self)	New environment API improves DI

Mental Model

Data Ownership Pyramid

```
@State          Local View State
@StateObject    View-Owned Model
@ObservedObject Parent-Owned Model
@Environment    Dependency Injection
@EnvironmentObject Global App State
```

Swift Concurrency Mental Model

```
@MainActor      UI safety
Task { }        Structured async
DetachedTask    Fire-and-forget
Sendable        Cross-thread guarantees
```

Flow Diagram: View → State → Model → Service

```
View
 ├─ @State (UI-local)
 ├─ @Binding (child prop sharing)
 ├─ @Environment / @EnvironmentObject (DI)
 └─ @StateObject (View-owned model)

Model (@Observable)
 └─ Business logic

Services (.shared or DI)
 └─ async, networking, persistence
```

Decision Tree

Should the View own the object?

- Yes → @StateObject

- No → @ObservedObject

Is this simple UI state?

- Yes → @State

Is this global shared state?

- Yes → @EnvironmentObject or Swift 5.9 environment values

Does this need persistence?

- Yes → @AppStorage

Does it update UI?

- Yes → @MainActor

Practical Example (Modern Pattern)

```
@Observable
class SessionModel {
    var user: User? = nil
    func logout() { user = nil }
}

@MainActor
class AuthService {
    static let shared = AuthService()

    func login(username: String, pw: String) async -> User {
        // network...
    }
}

struct LoginView: View {
    @Environment(SessionModel.self) var session
    @State private var username = ""
    @State private var password = ""

    var body: some View {
        VStack {
            TextField("Email", text: $username)
            SecureField("Password", text: $password)
            Button("Sign In") {
                Task {
                    session.user = await
AuthService.shared.login(username: username, pw: password)
                }
            }
        }
    }
}
```

End of document.