FOUNDATIONAL PROTOCOLS - 02

Swift Protocols for Reactive UI, Layout, and Concurrency

With Use Cases, Explanations, and Code Examples

Protocols powering SwiftUI internals

- ✓ Sendable
- ✓ ViewBuilder
- ✓ Actor
- ✓ MainActor
- ✓ AsyncIteratorProtocol
- ✓ AsyncSequence
- ✓ ResultBuilder
- ✓ Sequence and Collection
- ✓ Error (Custom Error Types)
- ✓ Layout (SwiftUI 2023+)

Sendable

Explanation:

Ensures a value can be safely passed across concurrency domains (e.g., to another task or thread).

Example:

```
struct User: Sendable {
    let id: Int
}
```

📌 **Use Case**:

Required for data passed between structured concurrency units like tasks and actors.

ViewBuilder

Explanation:

Special result builder used by SwiftUI to compose views declaratively.

Example:

```
@ViewBuilder
func buildSection(show: Bool) -> some View {
    if show {
        Text("Visible")
    } else {
        EmptyView()
    }
}
```

📌 **Use Case**:

Create custom view factories or reusable UI components with branching logic.

Actor

Explanation:

A reference type that protects its mutable state using isolation from concurrency issues. Protects from data races by allowing only one task at a time to access its internal properties.

Example:

```
actor Counter {
   private var count = 0

   func increment() {
      count += 1
   }
}
```

📌 **Use Case**:

Used to isolate and protect shared mutable state in concurrent environments, like caching, session management, or file access.

MainActor

Explanation:

Used to isolate a class or function to run on the main thread.

Example:

```
@MainActor
class ViewModel {
   var title: String = ""
   func updateUI() {
      // always runs on main thread
   }
}
```

📌 **Use Case**:

Protect UI state changes or any main-thread-only logic. @MainActor can be applied to functions, classes, or properties.

AsyncIteratorProtocol

Explanation:

Provides the logic to fetch each async value inside an AsyncSequence.

Example:

```
struct Countdown: AsyncIteratorProtocol {
   var current = 3
   mutating func next() async -> Int? {
      guard current > 0 else { return nil }
      defer { current -= 1 }
      return current
   }
}
```

🧠 Used by:

AsyncSequence.makeAsyncIterator()

→ returns this type to power the loop

📌 **Use Case:**

Emit values one by one — like countdowns or delayed actions

AsyncSequence

Explanation:

A protocol for types that emit values over time, used with for await.

Example:

```
struct Counter: AsyncSequence {
    typealias Element = Int

    func makeAsyncIterator() -> Iterator {
        Iterator()
    }
    struct Iterator: AsyncIteratorProtocol {
        var current = 0
        mutating func next() async -> Int? {
            current < 3 ? current : nil
        }
    }
} // Usage:
for await value in Counter() {
    print(value) // 0, 1, 2
}
```

📌 **Use Case:**

Consume values from streams like timers, APIs, or custom async collections. Think of it as : for-in, but with pauses — works over time

ResultBuilder

Explanation:

Power behind SwiftUI DSL — lets you build structured data using code blocks.

Example:

```
@resultBuilder
struct StringBuilder {
  static func buildBlock(_ components: String...) -> String {
      components.joined(separator: " ")
  }
}

func greet(@StringBuilder builder: () -> String) {
   print(builder())
}
greet {
   "Hello"
   "Swift"
   "6"
}
```

📌 **Use Case:**

Create custom builders like SwiftUI's ViewBuilder.

Sequence and Collection

Explanation:

Foundation protocols that power iteration (for-in) and index-based access.

Example:

```
struct CustomList: Collection {
    let data = ["A", "B", "C"]
    var startIndex: Int { data.startIndex }
    var endIndex: Int { data.endIndex }

    subscript(index: Int) -> String { data[index] }
    func index(after i: Int) -> Int { i + 1 }
}
```

📌 **Use Case:**

Create custom iterable or subscriptable types.
Note: Collection refines Sequence and adds index-based access.

Error (Custom Error Types)

Explanation:

Defines a custom error that can be thrown or caught in Swift's error-handling system.

Example:

```
enum LoginError: Error {
    case invalidCredentials, networkError
}
func login() throws {
    throw LoginError.invalidCredentials
}
do {
    try login()
} catch {
    print("Login failed:", error)
}
```

📌 **Use Case:**

Represent domain-specific or business logic errors.

Layout (SwiftUI 2023+)

Explanation:

Used for building custom layout containers in SwiftUI.

Example:

```
struct MyLayout: Layout {

    func sizeThatFits(proposal: ProposedViewSize, subviews: Subviews, cache: inout ()) ->
CGSize {

        // Compute layout size here

        return CGSize(width: 100, height: 50)

    }


    func placeSubviews(in bounds: CGRect, proposal: ProposedViewSize, subviews:
Subviews, cache: inout ()) {

        // Place subviews

    }

}
```

📌 **Use Case:**

Create advanced layouts beyond HStack/VStack/ZStack. (Available from iOS 16+ / SwiftUI 4.0)

🟧 Thanks for Reading !

Lets Connect:

🧑‍🚀 **Sri Hari YS**
iOS Lead Developer | Swift & SwiftUI Specialist
✉️ srihari.yss@gmail.com
📎 github

Follow, Comment, Reshare