FOUNDATIONAL PROTOCOLS

Essential Swift Protocols for Clean, Confident iOS Code

With Use Cases, Explanations, and Code Examples

- **Core Swift Protocols**

  - ✓ CaseIterable
  - ✓ CustomStringConvertible
  - ✓ CustomDebugStringConvertible
  - ✓ Equatable
  - ✓ Comparable
  - ✓ Hashable
  - ✓ RawRepresentable
  - ✓ Strideable
  - ✓ Codable
  - ✓ Numeric

- **SwiftUI Protocols**

  - ✓ Identifiable
  - ✓ View
  - ✓ ObservableObject
  - ✓ ViewModifier
  - ✓ DynamicProperty
  - ✓ Shape
  - ✓ Animatable
  - ✓ App
  - ✓ Scene

- **Combine Protocols**

  - ✓ Publisher
  - ✓ Subscriber
  - ✓ Cancellable
  - ✓ PassthroughSubject
  - ✓ CurrentValueSubject

Core Swift Protocols

Powering your logic, models, and data structures

- ✓ CaseIterable
- ✓ CustomStringConvertible
- ✓ CustomDebugStringConvertible
- ✓ Equatable
- ✓ Comparable
- ✓ Hashable
- ✓ RawRepresentable
- ✓ Strideable
- ✓ Codable
- ✓ Numeric

Case Iterable

Explanation:

Lets you access all cases of an enum automatically.

Example:

```swift
enum Day: CaseIterable {
    case monday, tuesday, wednesday
}
print(Day.allCases) // [monday, tuesday, wednesday]
```

📌 **Use Case**:

Used in pickers, toggles, menus, or when you want to iterate over all cases.

Custom String Convertible

Explanation:

Customize how objects are printed when using print()

Example:

```
struct User: CustomStringConvertible {
    var name: String
    var description: String { "User: \(name)" }
}
print(User(name: "Sri")) // User: Sri
```

📌 **Use Case**:

Pretty-print objects for logs or debugging. Improve debug output for model structs and classes.

CustomDebugStringConvertible

Explanation:

Customizes debug output (used in the debugger, not in print()

Example:

```
struct User: CustomDebugStringConvertible {
    var name: String
    var debugDescription: String { "[User: \(name)]" }
}
```

📌 **Use Case**:

Improve visibility of structs/classes when inspecting in Xcode debugger.

Equatable

Explanation:

Enables the == operator, allowing comparisons between values of the same type.

Example:

```
struct Product: Equatable {
    let id: Int
}
print(Product(id: 1) == Product(id: 1)) // true
```

📌 **Use Case**:

Compare values in arrays, conditionals, or tests.

Comparable

Explanation:

Allows custom types to be sorted or compared using <, >, etc.

Example:

```
struct Score: Comparable {
    let value: Int
    static func < (a: Score, b: Score) -> Bool {
        a.value < b.value
    }
}
let scores = [Score(value: 80), Score(value: 90)].sorted()
```

📌 **Use Case:**

Enables <, >, etc. Sort objects in arrays or use in min/max calculations.

Hashable

Explanation:

Enables types to be used in a Set, as keys in a Dictionary, and for view diffing in SwiftUI.

Example:

```
struct Item: Hashable {
    let id: Int
}

let items = Set([Item(id: 1), Item(id: 1)])
print(items.count) // Output: 1
```

📌 **Use Case**:

Used when you need to ensure uniqueness or detect duplicates efficiently.

RawRepresentable

Explanation:

Used to create enums backed by primitive types like String or Int.

Example:

```
enum Status: String {
    case active = "A", inactive = "I"
}
print(Status.active.rawValue) // Output: "A"
```

📌 **Use Case**:

Enables enums with backing values. Store enums in databases or send them over the network.

Strideable

Explanation:

Adds the ability to step through values — commonly used for ranges and time.

Example:

```
for i in stride(from: 0, to: 10, by: 2) {
    print(i)
}
// Output: 0, 2, 4, 6, 8
```

📌 **Use Case**:

Enables stride(from:to:by:) syntax. Create loops that increment in steps — especially with dates and numbers.

Codable (Encodable and Decodable)

Explanation:

Enables automatic encoding and decoding of data, especially for JSON.

Example:

```
struct Profile: Codable {
    let name: String
}

let data = try JSONEncoder().encode(Profile(name: "Sri"))
let decoded = try JSONDecoder().decode(Profile.self, from: data)
```

📌 **Use Case**:

Auto-generates encoding/decoding for structs. Parse data from API or save models locally.

Numeric

Explanation:

Allows arithmetic operations in a type-safe way

Example:

```
func doubleIt<T: Numeric>(_ input: T) -> T {
    input * 2
}
print(doubleIt(10))        // Output: 20 (Int)
print(doubleIt(5.5))       // Output: 11.0 (Double)
```

📌 **Use Case**:

Create generic math functions that work for Int, Float, Double

SwiftUI Protocols

Making Views Reactive, Dynamic, and Composable

- ✓ Identifiable
- ✓ View
- ✓ ObservableObject
- ✓ ViewModifier
- ✓ DynamicProperty
- ✓ Shape
- ✓ Animatable
- ✓ App
- ✓ Scene

Identifiable

Explanation:

Requires each instance to have a unique id so SwiftUI can track, diff, and update views efficiently.

Example:

```
struct Task: Identifiable {
    let id = UUID()
    let title: String
}

// Used in SwiftUI like:
List([Task(title: "Build")]) { task in
    Text(task.title)
}
```

📌 **Use Case:**

Used in List, ForEach, and other dynamic views to distinguish between elements.

View

Explanation:

The foundational protocol for building all user interfaces in SwiftUI.

Example:

```
struct MyView: View {
    var body: some View {
        Text("Hello")
    }
}
```

📌 **Use Case**:

Everything visible in SwiftUI must conform to View. Defines the visual layout and structure of SwiftUI screens.

Observable Object

Explanation:

A class type that notifies SwiftUI views when its properties change.

Example:

```
class Counter: ObservableObject {
    @Published var count = 0
}
// Used in a view with @ObservedObject or @StateObject
```

📌 **Use Case**:

Power view models for reactive UI updates.

ViewModifier

Explanation:

Encapsulates reusable view styling and behavior.

Example:

```
struct CardStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .padding()
            .background(Color.gray.opacity(0.1))
            .cornerRadius(10)
    }
}

Text("Styled").modifier(CardStyle())
```

📌 **Use Case:**

Apply changes to views in a composable way. Create clean, reusable modifiers for padding, backgrounds, shadows, etc.

DynamicProperty

Explanation:

Used for building custom property wrappers that integrate with SwiftUI's rendering lifecycle.

Example:

```
@propertyWrapper struct MyState: DynamicProperty {
   @State private var value = 0
   var wrappedValue: Int { value }
}
```

📌 **Use Case**:

Enable custom property wrappers to behave like built-in ones (@State, @Binding, etc.)

Shape

Explanation:

Protocol for creating vector-based drawing paths.

Example:

```
struct Triangle: Shape {
   func path(in rect: CGRect) -> Path {
      var path = Path()
      path.move(to: rect.origin)
      path.addLine(to: CGPoint(x: rect.maxX, y: rect.minY))
      path.addLine(to: CGPoint(x: rect.midX, y: rect.maxY))
      path.closeSubpath()
      return path
   }
}
```

📌 **Use Case:**

Build custom geometric visuals like triangles, waves, or complex SVG-style paths. Used for drawing paths like circles, lines, etc.

Animatable

Explanation:

Allows a view or shape to animate its changing values smoothly.

Example:

```
struct RotatingView: View {
   @State private var angle = 0.0
   var body: some View {
      Text("Spin")
         .rotationEffect(.degrees(angle))
         .onTapGesture {
            withAnimation { angle += 45 }
         }
   }
}
```

📌 **Use Case:**

Enable fine-grained control over animations for custom drawing or transitions

App

Explanation:

The main entry point for SwiftUI apps. Replaces UIApplicationDelegate.

Example:

```
@main
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

📌 **Use Case**:

Define global app configuration and set the initial view.

Scene

Explanation:

Represents an independent UI instance (like a window or widget). Used to manage multiple views/scenes.

Example:

```
var body: some Scene {
   WindowGroup {
      HomeScreen()
   }
}
```

📌 **Use Case**:

Define how the app's UI is presented — via WindowGroup, DocumentGroup, etc.

Combine Protocols

- Handling Data Streams the Declarative Way

- ✓ Publisher
- ✓ Subscriber
- ✓ Cancellable
- ✓ PassthroughSubject
- ✓ CurrentValueSubject

Publisher

Explanation:

A type that delivers a sequence of values over time to one or more subscribers.

Example:

```
import Combine

let publisher = Just("Hello")
_ = publisher.sink { value in
   print(value) // prints: Hello
}
```

📌 **Use Case**:

Streams of values over time (e.g., network data, timers, form input). React to user input, API calls, timers, and more in a declarative way.

Subscriber

Explanation:

Defines how to receive values from a Publisher. Controls demand and handles completion.

Example:

```
import Combine

final class MySubscriber: Subscriber {
  typealias Input = String ; typealias Failure = Never

  func receive(subscription: Subscription) {
    subscription.request(.unlimited)
  }

  func receive(_ input: String) -> Subscribers.Demand {
    print("Received: \(input)")
    return .none
  }
  func receive(completion: Subscribers.Completion<Never>) {
    print("Completed") } }
```

📌 **Use Case:**

Create custom handlers for reactive streams.

Cancellable

Explanation:

A token representing a subscription, which can be cancelled to stop data flow and release resources.

Example:

```swift
import Combine

var cancellable: AnyCancellable?

let publisher = Just("Clean")
cancellable = publisher.sink { print($0) }

// Later in lifecycle:
cancellable?.cancel()
```

📌 **Use Case:**

Manage memory by canceling Combine pipelines when they are no longer needed.

Subject (PassthroughSubject)

Explanation:

A Combine subject that starts empty and only forwards values you manually send. It does not store the latest value.

Example:

```
import Combine


let subject = PassthroughSubject<String, Never>()

let subscription = subject.sink { print("Value:", $0) }


subject.send("Tapped")  // Output: Value: Tapped
```

📌 **Use Case**:

Send events like button taps or notifications to multiple subscribers in real time.

Subject (CurrentValueSubject)

Explanation:

A subject that always holds the latest value and immediately sends it to new subscribers. Think of it as a mutable wrapper around a value with Combine support.

Example:

```
import Combine

let subject = CurrentValueSubject<Int, Never>(0)
let subscription = subject.sink { print("Value:", $0) }

subject.send(10)
// Output: Value: 0 (initial)
//        Value: 10 (new)
```

📌 **Use Case**:

Track and expose current state (e.g., form field values, toggles, user settings).

🟧 Thanks for Reading !

Lets Connect:

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

“Knowledge shared is knowledge multiplied.”

→ If this helped, follow or comment with your favorite protocol!