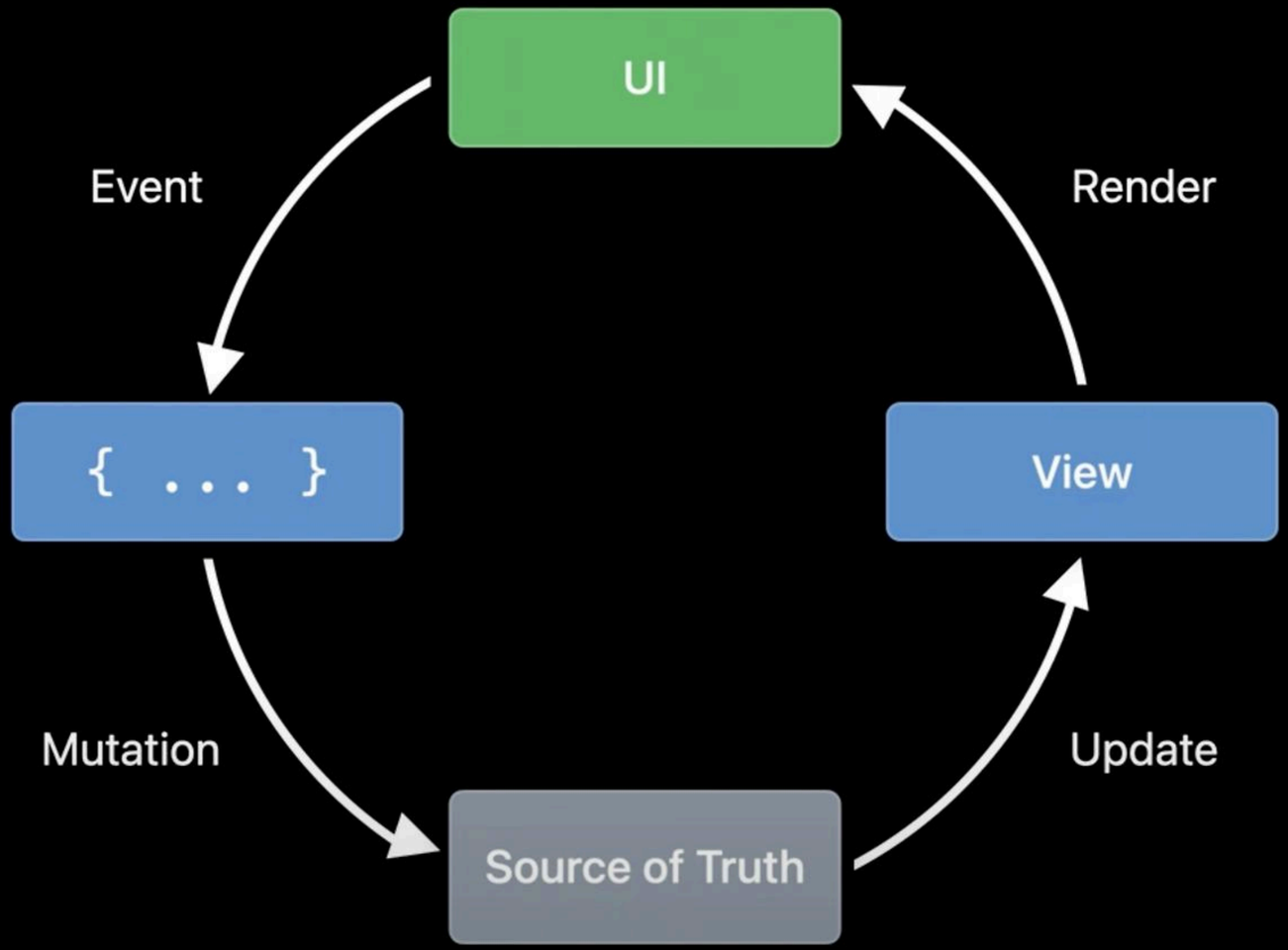
SwiftUI의 상태관리

김형석

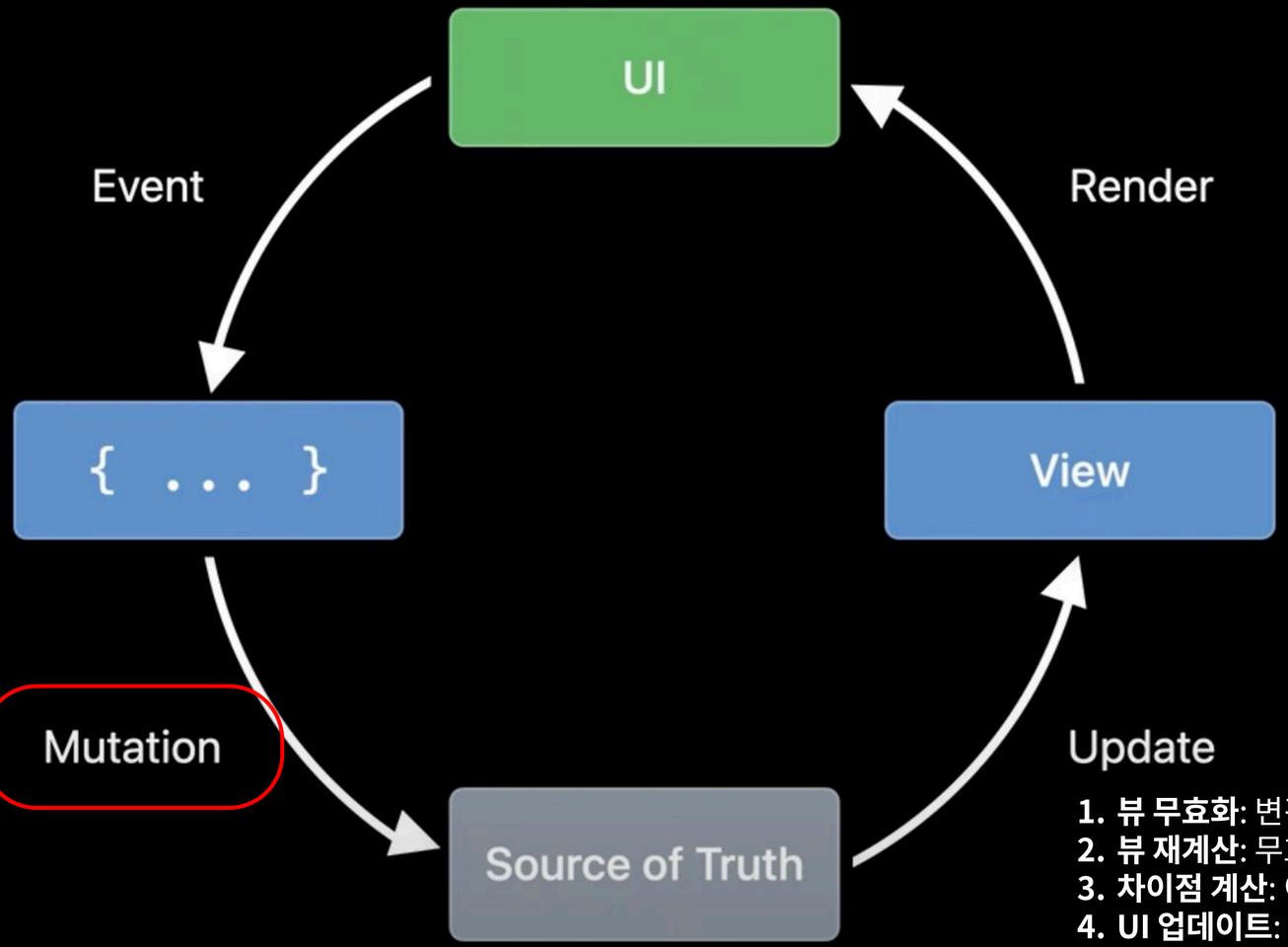
상태 기반 UI 뷰 업데이트 사이클

- **정체성:** SwiftUI가 앱의 여러 업데이트에서 요소들을 동일하거나 구별되는 것으로 인식하는 방법
- **의존성**: 인터페이스가 업데이트되어야 할 시점과 그 이유를 이해하는 데 도움을 줌
- **수명**: 뷰와 데이터를 시간에 따라 추적하는 방식

의존성의 변화는 어느 단계에 해당되는 것일까요?



Mutation

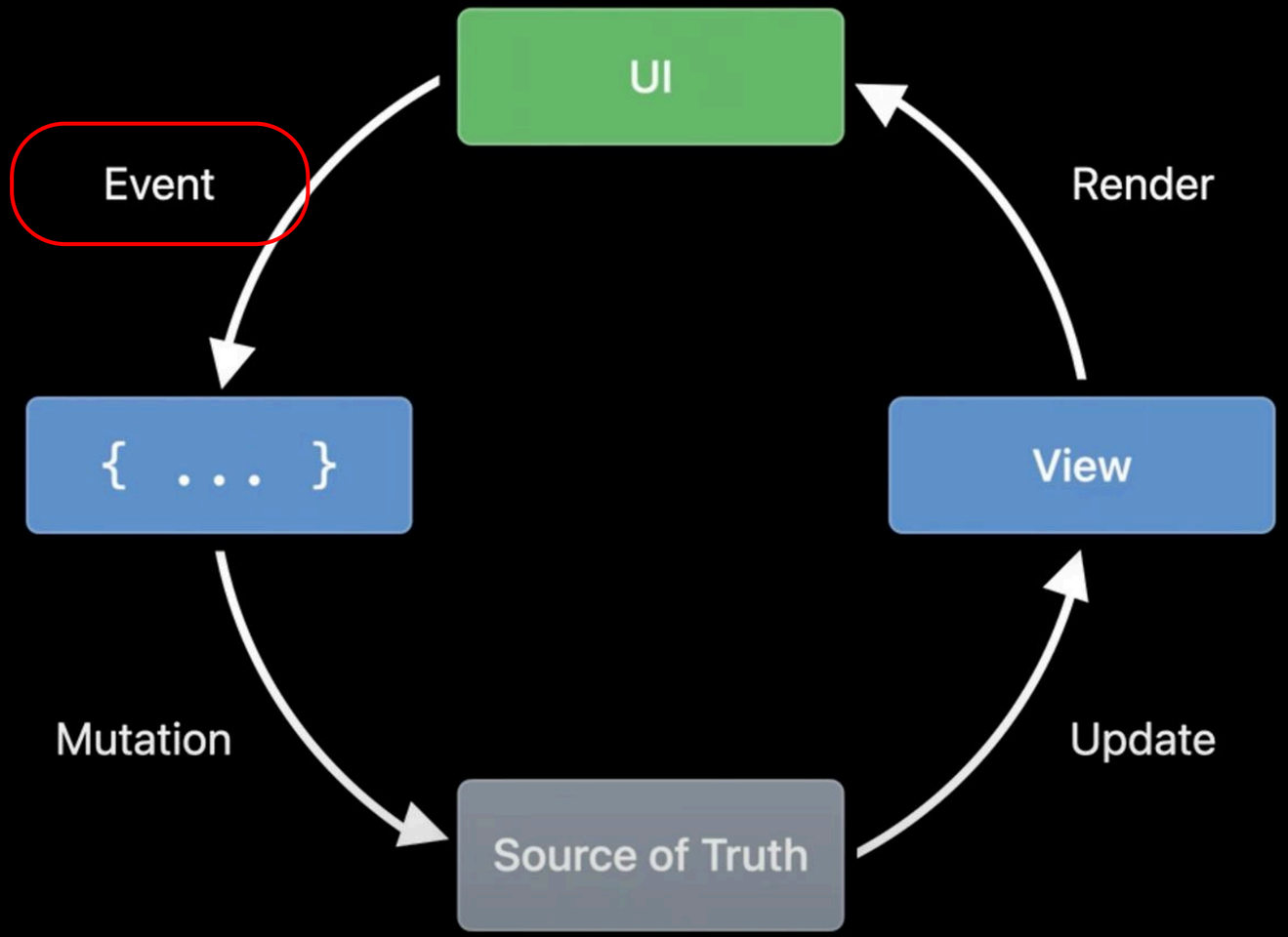


의존성 변경 →

1. **뷰 무효화**: 변경된 상태에 의존하는 뷰들을 무효화
2. **뷰 재계산**: 무효화된 뷰의 body를 재계산
3. **차이점 계산**: 이전 뷰 상태와 새로운 뷰 상태의 차이를 계산
4. **UI 업데이트**: 필요한 부분만 효율적으로 업데이트

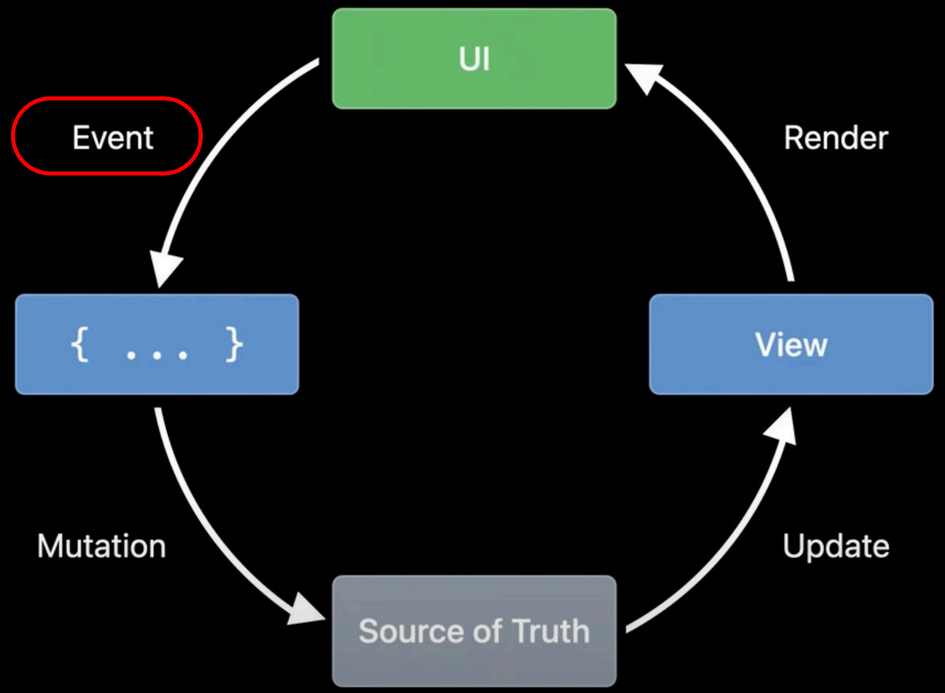
상태 변화 감지: Property Wrapper를 통해 상태 변화를 감지

Event에는 어떤 요소가 있을까요?

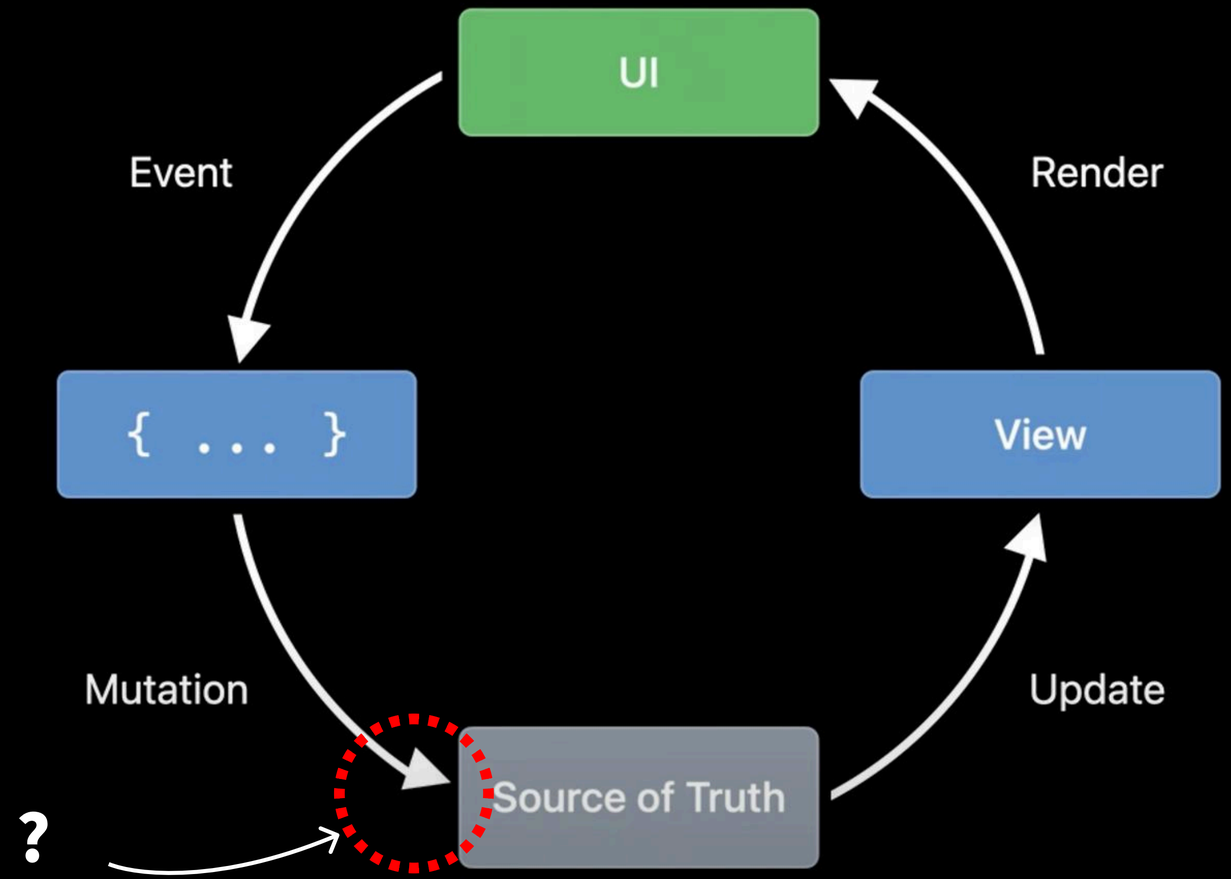


Event 요소

- **사용자 상호작용:** 버튼탭과 같은 직접적인 액션
- **시스템 이벤트:** 타이머, NotificationCenter...
- **퍼블리셔:** onReceive, onChange, onOpenURL, onContinueUserActivity...



의존성 변화를 전파하는 수단



ObservableObject

Combine의 퍼블리셔 메커니즘과 SwiftUI의 뷰 업데이트 시스템을 연결하는 다리 역할

Combine 프레임워크의 프로토콜

SwiftUI의 상태 관리 시스템과 연동되어 데이터 변화를 UI에 반영하는 핵심 역할

objectWillChange Publisher를 통해 객체의 변화를 감지할 수 있는 기능을 제공

ObservableObjectPublish는 Combine의 PassthroughSubject를 내부적으로 사용하는 단순한 퍼블리셔

- 값 저장 없이, 이벤트 전달만 수행
- Void를 출력하여 변경 발생 신호만 전달

```swift
public protocol ObservableObject: AnyObject {
    /// The type of publisher that emits before the object has changed.
    associatedtype ObjectWillChangePublisher: Publisher = ObservableObjectPublisher where ObjectWillChangePublisher.Failure == Never

    /// A publisher that emits before the object has changed.
    var objectWillChange: ObjectWillChangePublisher { get }
}

// 기본 구현 제공
extension ObservableObject where ObjectWillChangePublisher == ObservableObjectPublisher {
    public var objectWillChange: ObservableObjectPublisher {
        if let publisher = objc_getAssociatedObject(self, &ObservableObjectPublisherKey) as? ObservableObjectPublisher {
            return publisher
        }
        let publisher = ObservableObjectPublisher()
        objc_setAssociatedObject(self, &ObservableObjectPublisherKey, publisher, .OBJC_ASSOCIATION_RETAIN)
        return publisher
    }
}

// ObservableObjectPublisher의 주요 구현
public final class ObservableObjectPublisher: Publisher {
    public typealias Output = Void
    public typealias Failure = Never

    private let subject = PassthroughSubject<Void, Never>()

    public init() {}

    public func receive<S>(subscriber: S) where S: Subscriber, Failure == S.Failure, Output == S.Input {
        subject.receive(subscriber: subscriber)
    }

    public func send() {
        subject.send()
    }
}
```

모든 ObservableObject는 objectWillChange라는 publisher를 갖고 있어야 합니다. 여기서 objectWillChange라는 기본값으로 ObservableObjectPushlisher 타입이 지정된 연관 타입이며, 객체 변경 직전에 이벤트를 발행합니다

ObservableObject

```swift
@propertyWrapper
public struct Published<Value> {
    public var wrappedValue: Value {
        get { storage.value }
        set {
            // 여기가 핵심: 값이 변경되기 전에 publisher에 알림
            storage.publisher.send()
            storage.value = newValue
        }
    }

    public var projectedValue: Publisher<Value, Never> {
        return storage.publisher.eraseToAnyPublisher()
    }

    private let storage: PublishedStorage<Value>

    public init(wrappedValue: Value) {
        storage = PublishedStorage(value: wrappedValue)
    }
}

private class PublishedStorage<Value> {
    var value: Value
    let publisher = PassthroughSubject<Value, Never>()

    init(value: Value) {
        self.value = value
    }
}

// @Published 속성 래퍼가 ObservableObject와 연동되는 방식
extension ObservableObject {
    public static func _willChange<T>(
        _ object: ObservableObject,
        _ keyPath: KeyPath<Self, T>,
        _ value: T
    ) {
        // 이 메서드는 @Published 프로퍼티가 변경될 때 내부적으로 호출됨
        if let publisher = object.objectWillChange as? ObservableObjectPublisher {
            publisher.send()
        }
    }
}
```

```swift
// SwiftUI 내부 뷰 업데이트 메커니즘의 간략한 버전
extension View {
    public func environmentObject<T: ObservableObject>(_ object: T) -> some View {
        self.environment(\.environmentObjects, [ObjectIdentifier(T.self): object])
    }

    @_transparent
    public func onReceive<P: Publisher>(
        _ publisher: P,
        perform action: @escaping (P.Output) -> Void
    ) -> some View where P.Failure == Never {
        ReceiveView(content: self, publisher: publisher, action: action)
    }
}

// 내부 구현에 가깝게 단순화한 코드
struct ObservedObjectView<Content: View, ObjectType: ObservableObject>: View {
    @ObservedObject var object: ObjectType
    let content: Content

    init(object: ObjectType, @ViewBuilder content: () -> Content) {
        self._object = ObservedObject(wrappedValue: object)
        self.content = content()
    }

    var body: some View {
        content.onReceive(object.objectWillChange) { _ in
            // 값이 변경될 때 뷰 업데이트 트리거
        }
    }
```

SwiftUI는 ObservableObject의 objectWillChange publisher를 내부적으로 구독하여 값이 변경될 때 UI를 업데이트

wrappedValue의 setter에서 값이 실제로 변경되기 전에 퍼블리셔의 send() 메서드를 호출하여 변경 알림을 발행

@Published로 표시된 프로퍼티가 변경될 때, 해당 프로퍼티의 퍼블리셔뿐만 아니라 ObservableObject의 objectWillChange 퍼블리셔도 ObservableObject._willChange 정적 메서드를 통해 함께 발행

Property Wrapper와 의존성 관리

데이터와 UI 간의 의존성을 정의해 데이터를 바탕으로 UI를 일관되게 유지하는 역할 수행

@State: 뷰 내부 UI에 국한된 일시적인 상태 처리에 사용됩니다. 상태 변화가 이벤트 소스가 됩니다.

@Binding: 상위 뷰의 상태에 대한 참조를 제공하여 읽기/쓰기를 가능하게 합니다. 원본 상태가 변할 때 UI가 업데이트됩니다.

@StateObject: ObservableObject의 생명주기를 뷰와 연결합니다. 객체가 뷰의 생명주기 동안 유지됩니다.

@ObservedObject: 외부에서 제공된 ObservableObject를 관찰합니다. 이때 뷰의 종속성, 즉 뷰 업데이트의 소유권을 관리하는 책임이 생성됩니다. 관찰 대상의 변화가 UI 업데이트를 트리거합니다.

@EnvironmentObject: 뷰 계층 전체에서 공유되는 객체를 관리합니다. 공유 객체의 변화가 의존하는 모든 뷰의 업데이트를 트리거합니다.

SSOT

SSoT 원칙 준수를 위한 기초원칙

1. 데이터 모델과 UI(ViewModel)의 분리

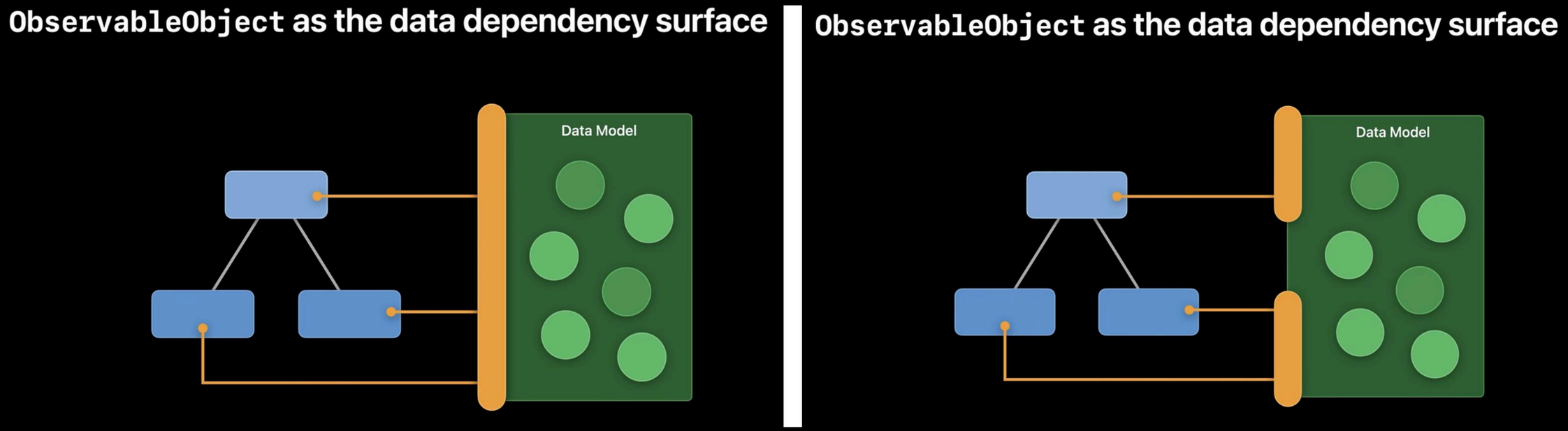
UI와 별도로 데이터 모델을 설계하고 관리하는 것을 지향

Data 자체는 불변성과 예측 가능성이 더 높은 Value type
로 유지

부수효과, 생명주기관리, 상태 관리와 같이 실시간 처리가 필요로 하는 로직들
은 Reference type인 ViewModel이나 Data store로 설계하여 분리

```swift
struct User {
    let id: UUID
    var name: String
    var email: String
}

// ViewModel (Reference Type)
class UserProfileViewModel: ObservableObject {
    @Published var displayName: String = ""
    @Published var isEmailVerified: Bool = false

    private let user: User // 데이터 모델 참조

    init(user: User) {
        self.user = user
    }

    func formatDisplayName() {
        // UI를 위한 데이터 가공
        displayName = "👤 " + user.name
    }
}
```

데이터 모델과 UI(ViewModel)의 분리



단일 ObservableObject로 구성해 전체 데이터 모델을 중앙집중화할 경우, 애플리케이션 내 가능한 상태변화를 쉽게 이해할 수 있게 되겠죠. 반면, 여러 개의 ObservableObject로 나눌 경우, 보다 세분화된 데이터 노출이 가능

2. 데이터의 생명주기관리

설계 간에 데이터를 누가 소유해야하는지에 대한 고민 필요

데이터의 생애주기 관리 및 소스로서의 권한에 대한 올바른 배분 필요

Apple은 **공통 조상을 통해 데이터를 공유하는 것**을 지향

대표적인 데이터 관리 범위

View 범위: ObservableObject 프로토콜을 뷰의 생명주기와 연결시키는 StateObject 속성 래퍼, 혹은 State를 적용시켜야 합니다.

Scene 범위: SceneStorage 속성 레퍼를 통해 Scene 별로 데이터 저장

*Scenes는 고유한 뷰 트리를 가지므로 데이터의 중요한 조각을 트리의 루트에 연결할 수 있으며, 각 Scene인스턴스는 독립적인 SoT를 가질 수 있습니다.

App 범위: AppStorage를 통해 앱 전체에서 유지

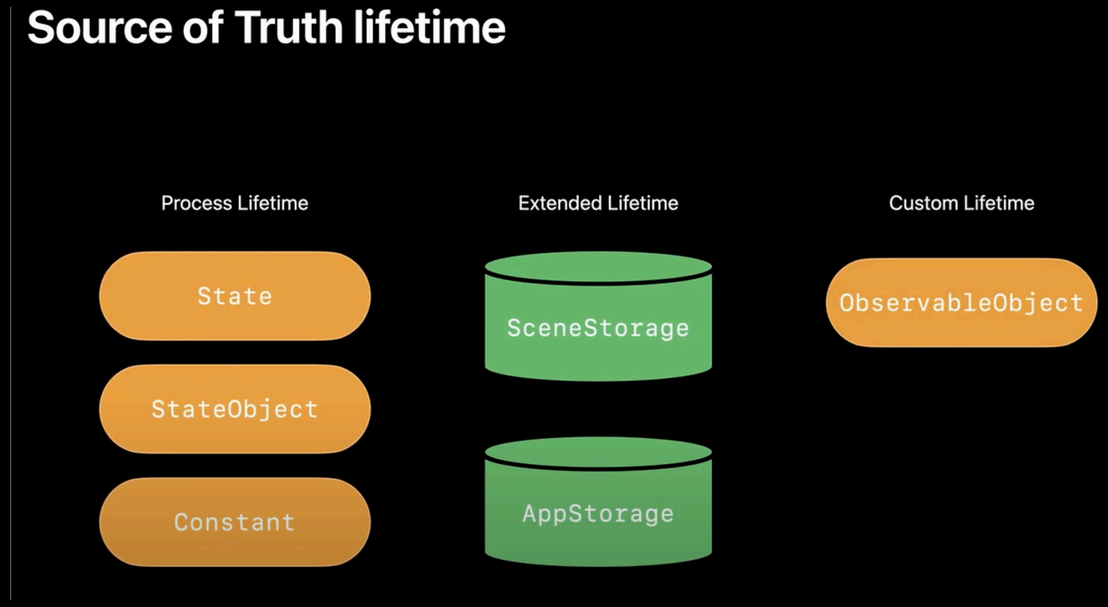
```swift
struct BookClubSettings: View {
    @AppStorage("updateArtwork") private var updateArtwork = true
    @AppStorage("syncProgress") private var syncProgress = true

    var body: some View {
        Form {
            Toggle(isOn: $updateArtwork) {
                //...
            }

            Toggle(isOn: $syncProgress) {
                //...
            }
        }
    }
}
```

App-Wide Source of Truth


```swift
struct BookClubSettings: View {
    @AppStorage("updateArtwork") private var updateArtwork = true
    @AppStorage("syncProgress") private var syncProgress = true

    var body: some View {
        Form {
            Toggle(isOn: $updateArtwork) {
                //...
            }

            Toggle(isOn: $syncProgress) {
                //...
            }
        }
    }
}
```

2. 데이터의 생명주기관리

- **Process Lifetime:** 앱이 종료되거나 재시작될 경우, 초기화됩니다.
- **Extended Lifetime:** 앱 범위의 전역저장소로 확장된 생명주기를 갖고 있으며, 저장 및 복원이 자동으로 이루어집니다.
- **Custom Lifetime:** ObservableObject는 서버에 저장하거나, 다른 서비스와 통신하는 등 고수준의 커스텀 동작이 가능토록 하는 도구로 설계되었습니다.



올바른 속성 래퍼 선택

뷰 생명주기 관리

@StateObject

- ObservableObject를 뷰의 생명주기와 연결
  - body 처음 평가되기 직전에 초기화
  - 뷰 생명주기 동안 동일 인스턴스 유지
  - onDisappear 필요없이 뷰가 필요없어지면 자동으로 할당 해제
- 뷰 재생성/업데이트와 무관하게 데이터 일관성 보장

@ObservedObject

- 뷰 구조체 생성 시점에 초기화
- 뷰 재생성 시 새로운 인스턴스 생성 가능
- 데이터 일관성 보장되지 않을 수 있음

```swift
struct ContentView: View {
    // 1. 뷰 구조체 인스턴스가 생성될 때
    @StateObject private var viewModel = ViewModel()  // 아직 초기화되지 않음

    // 뷰가 재생성될 때마다 새로운 인스턴스가 생성될 수 있음
    @ObservedObject var observedViewModel = ViewModel()
    init() {
        print("ContentView initiated")
        // 이 시점에서 StateObject는 아직 초기화되지 않은 상태
    }

    // 2. body가 계산되기 직전
    var body: some View {
      VStack{
        // 이 시점에서 StateObject가 초기화되고 유지됨
            Text(viewModel.data)  // 여기서부터 viewModel 사용 가능
            Text(observedViewModel.data)
        }
    }
}
```

올바른 속성 래퍼 선택

SSoT 설계

@ObservedObject

- 데이터를 소유하지않고 ObservedObject 속성래퍼로 StateObject를 참조할 수 있음

@Binding

- Binding 속성 래퍼를 통해 SSot를 보존하면서 사용자와의 상호작용을 반영할 수도 있음

```
struct ParentView: View {
    @StateObject private var viewModel = ViewModel()  // 부모가 소유권을 가짐

    var body: some View {
        ChildView(viewModel: viewModel)  // 자식에게는 ObservedObject로 전달
    }
}

struct ChildView: View {
    @ObservedObject var viewModel: ViewModel  // 소유하지 않고 관찰만 함

    var body: some View {
        Text(viewModel.data)
    }
}
```

올바른 속성 래퍼 선택

의존성 그래프 단순화

@EnvironmentObject

- 뷰 계층 전체에 공유할 수 있는 EnvironmentObject 속성 래퍼를 사용해 의존성 그래프를 단순화 할수도 있음