**SwiftUI Fundamentals by Sean Allen**

[**https://www.youtube.com/watch?v=b1oC7sLIgpI**](https://www.youtube.com/watch?v=b1oC7sLIgpI)

**2단계. SwiftUI 프레임워크(1:18:00~3:38:00)**

사실 1단계의 자식 뷰에서 isNight 변수에 @Binding을 설정한 건 @Binding을 설명하기 위함이지 실제 필요한 건 아니었다. 메인 화면의 로직은 @State 만으로도 충분했다.

**프로젝트 매니저**

프로젝트 폴더 최상위에 프로젝트 매니저가 있다. 여기서 iOS 버전과 아이폰 버전을 확인해 코드에 deprecated된 메소드가 없는지 확인해야 한다.

**Modifiers**

.forgroundStyle(primary: , secondary: , third: )

.simbolRenderingMode(.hierarchical (or .palette))

Color.blue.gradient -> gradient는 자연스럽게 색상에 입체감을 준다.

sheet ->

fullScreenCover ->

**Swift는 어떻게 동작하는가?**

**선언형 vs 명령형**

UIKit 까지도 명령형 프로그래밍이 적용되었기 때문에 선언형 프로그래밍과 명령형 프로그래밍의 차이를 알고 다르게 적용하는 게 상당히 중요하다. 선언형과 명령형의 차이는 다른 문서를 통해 많이 다뤘다. 여기서는 명령형은 로직과 UI를 함께 절차적으로 다루는 것, 선언형은 로직을 정의해두면 시스템에 의해 UI가 데이터 플로우를 반영해 업데이트 되는 것으로 설명한다.

**Modifiers의 계층**

1장에서 따로 정리했던 내용과 겹친다. 수정자가 계층적으로 동작하기 때문에 수정자를 적용하는 순서가 상당히 중요하다. 수정자를 적용할 때마다 ModifiedContent<>라는 래퍼가 해당 뷰 인스턴스를 감싸게 된다. 컨테이너를 통해 여러 개의 뷰를 감싼 뒤 일괄적으로 Modifiers를 적용할 수도 있다.

**@ViewBuilder**

뷰빌더는 컨테이너의 생성자 안에 선언되어 있다. 실제 사용시 코드상에서는 클로저로 표현된다. 뷰빌더의 핵심 아이디어는 여러 개의 뷰를 편집해 하나의 뷰로 만드는 것이다. 조건문을 허용해 조건에 따라 달라지는 커스텀 뷰를 만들 때 특히 유용하다. 명시적으로 Group이나 AnyView 래퍼를 선언할 필요를 줄여 깔끔한 코드를 작성할 수 있게 하기도 한다.

**스위프트가 Class 기반이 아닌 Struct 기반 구조를 선택한 이유**

UIKit는 클래스 기반 명령형 프로그래밍 방식을 채택한다. 예를 들어 UIButton은 UIView를 상속하는데 이런 식으로 여러 계층의 상속이 생기면 너무 많은 기능이 생기고 시스템 차원에서 이를 일일이 다루기 힘들어진다. 스위프트는 상속이 존재하지 않는 구조체 방식을 택해서 구조체를 가볍게 유지하고 구조체 안에 정의해 놓은 내용을 시스템이 관리하게 하는 방식을 택했다.

**SwiftUI가 UI를 그려내는 방식**

SwiftUI의 뷰 계층은 트리 구조로 구성된다. 뷰 트리라고 보면 된다. SwiftUI는 데이터 플로우를 감시하다가 변경이 감지되면 데이터 변경 사항이 변경된 뷰에 접근해 해당 뷰를 업데이트한다. 더 깊게 다루면 복잡하지만 강의에서는 우선 이 정도의 배경지식만 갖춰두라고 조언한다.

**SwiftUI가 바꿔놓은 것**

UIKit 개발에 비해 SwiftUI는 UI 개발을 굉장히 편리하게 만들었다. 하지만 쉽고 규격화된 탓에 모두 비슷해 보이는 UI 디자인을 양산하게 됐다. 몇 가지 방법을 사용해 특징을 입힐 수는 있지만 전반적으로는 그렇다고 한다. 또한 SwiftUI는 상대적으로 어려운 데이터 조작 방식을 가지고 있다. 원하는 로직을 앱이 실행하게끔 하는 일이 UI 구축보다 어렵다. 솔직히 말하면 SwiftUI 자체가 나온지 얼마 안 돼서 가르치는 자신도 전문가라고 하기는 힘들다. 이런 환경에서 개발해야 하기 때문에 한 가지 방식만 답이라고 생각하지 말고 여러 방식으로 접근하고 개발하기를 권장한다.

**Source of Truth**

SwiftUI에서 데이터를 다룰 때 해당 키워드가 자주 언급되어 정리한다.

진실의 근원은 주요하거나 권위 있는 앱 데이터의 근원을 나타낸다. Source of Truth는 뷰가 데이터 변화에 의해 자동으로 업데이트되는, 데이터 주도 디자인의 근간을 이루는 핵심 개념이다. Source of Truth를 따르면 모든 데이터는 그것을 소유하고 관리하는 하나의 최종 근원에서 비롯되어야 한다. 3 개의 특징으로 Source of Truth의 개념을 살펴보자.

1. **단일 오너십:** 각각의 데이터는 반드시 단일한 근원을 가져야 한다. 앱의 한 부분이 이 데이터를 소유하고 업데이트를 할 책임이 있다는 말이다.
2. **일관성:** 하나의 데이터 소스를 가짐으로서 앱 전체에 걸쳐 일관성을 유지할 수 있다. 해당 데이터와 상호작용하고 이를 그려내는 모든 뷰는 같은 소스에 반응한다. 이를 통해 뷰들은 항상 동기화 되어 있을 수 있다.
3. **데이터 플로우:** SwiftUI에서 소스에서 비롯된 데이터 플로우는 뷰로 전달된다. 뷰는 데이터를 관찰하고 데이터가 변경될 때 이를 자동으로 업데이트 한다.

@State, @StateObject 등은 모두 이 개념을 구현하기 위한 래퍼이다.

@Binding, @EnvironmentObject 등은 그 자체로 source of truth는 아니지만 뷰가 source of truth에 접근하고 상호작용할 수 있게 한다.

**Source of Truth 정리**

동적이고 인터랙티브한 앱을 단순한 방식으로 제작할 수 있게끔 한다.

데이터 권한이 있는 유일한 데이터 소스를 만듦으로써, 개발자는 데이터 중복이나 비일관성의 문제에서 자유로울 수 있다.

**@StateObject**

뷰 안에서 데이터의 source of truth를 선언하기 위한 프로퍼티 래퍼이다. ObservableObject를 채택하는 참조형 타입을 사용하기 위해 Swift 2.0 버전에 추가되었다. 복잡한 SwiftUI 앱에서 상태를 정확하고 효율적으로 관리하기 위한 핵심 프로퍼티 래퍼이다.

@StateObject는 ObservableObject를 생성하고 소유하기 위해 사용된다. @StateObject를 선언하는SwiftUI의 뷰는 객체의 생명 주기를 관리하고 초기화해야 할 책임이 생긴다. 또한 해당 뷰는 객체를 소유하므로 데이터가 뷰 업데이트 이후에도 영속적으로 관리된다.

ObservableObject에 @Published로 래핑된 프로퍼티가 있으면, 뷰는 자동으로 변경 사항을 업데이트한다. 이는 SwiftUI 반응형 프레임워크의 일부 기능이다. 특정 데이터가 특정 뷰에만 동기화되어 사용될 때 유용하게 쓸 수 있다.

**@ObservedObject와 @StateObject의 차이**

전자도 뷰가 ObservableObject를 구독하도록 하기는 한다. 그러나 뷰가 그 객체를 소유하지는 않는다. 객체는 반드시 뷰의 바깥에서 생성되어야 한다. @ObservedObject는 단순히 뷰가 객체의 변화를 감지하기만 하는 것이다. 만약에 뷰가 파괴되었다가 다시 생성되면, @ObservedObject 어딘가에서 관리되지 않는 한, @ObservedObject의 상태가 손실될 수 있다.

@StateObject는 뷰가 존재하는 한 ObservedObject를 생성하고 관리한다. 뷰 안에서 상태를 관리하기에 좀 더 안정적이라고 할 수 있다.

class MyViewModel: ObservableObject {

@Published var count = 0

}

struct ContentView: View {

@StateObject var viewModel = MyViewModel()

var body: some View {

VStack {

Text("Count: \(viewModel.count)")

Button("Increment") {

viewModel.count += 1

}

}

}

}

**@StateObject**

어떤 뷰가 특정 개체의 데이터와 상태를 관리해야 할 때 사용한다.

**@ObservedObject**

여러 개의 뷰가 하나의 객체를 공유하고자 할 때 사용한다.

**애플리케이션이 복잡해질수록 데이터와 상태를 효과적으로 관리하기 위해 두 옵션을 잘 구분해야 한다.**

**에셋에 아이콘 저장하기**

**LazyVGrid, LazyHGrid**

둘 다 스위프트의 뷰이다. 유연한 격자를 그리는 것을 목적으로 동작한다.

각자 수직, 수평으로 스크롤 가능한 그리드를 만든다.(스크롤 기능은 원래 없다, ScrollView{}로 래핑해서 쓰인다.)

**왜 Lazy인가?**

이 그리드들은 lazy-loaded 방식을 취한다.

스크롤링이 되지 않은 영역은 그려지지 않고 있다가 필요에 의해 보여져야 할 때 초기화되고 다시 그려진다. eager 방식은 이와 다르게 전체의 뷰를 미리 다 그려놓는다. lazy 방식은 많은 수의 복잡한 계층 구조를 가진 뷰를 다룰 때 유용하다. 렌더링에 대량의 데이터가 필요한 경우에도 유용하다.

**MVVM**

SwiftUI는 발전, 개선 중인 프레임워크이기 때문에 어떤 아키텍처가 보편적으로 가장 적합한지에 대한 논쟁이 많다. MVVM가 많이 쓰이기는 하지만 항상 정답인 것은 아니다. MVVM에 대한 개괄은 CS193p 3강에서 정리했기 때문에 생략한다.

**옵저버**

**옵저버는 구조체나 클래스처럼 자체적으로, 독립적으로 사용 가능한 엔티티가 아니다.**

프로퍼티의 변화에 대응하는 특별한 기능에 가깝다.

**didSet**

didSet은 프로퍼티 옵저버이다. 프로퍼티에 붙여서 사용할 수 있으며 프로퍼티의 값이 바뀌면 특정 코드를 실행하게 할 수 있다.(초기화 제외). 프로퍼티 값이 바뀌면 할당 이후에 바로 해당 코드를 실행한다. didSet 안에서는 oldValue라는 특별한 상수에 접근할 수 있는데 oldValue는 해당 프로퍼티가 새 프로퍼티로 바뀌기 전의 값을 저장한다. 이전 값과 새 값을 비교할 때 유용하다. 하지만 퍼포먼스 이슈가 있기 때문에 신중하게 사용하는 게 좋다.

**willSet**

willSet은 didSet처럼 프로퍼티가 바뀔 때 동작하지만 시점이 다르다. willSet은 프로퍼티가 바뀌기 직전에 동작한다.

class ExampleClass {

var myProperty: String {

willSet(newVal) {

print("About to set myProperty to \(newVal)")

}

didSet {

print("Just set myProperty to \(myProperty) from \(oldValue)")

}

}

}