**SwiftUI Fundamentals by Sean Allen**

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

**1단계. 날씨앱 만들기(0:00 ~ 1:03:00)**

ContentView 안에 연산 프로퍼티로 구현된 body가 최상위 View의 역할을 한다. 이때 body는 단 한 개의 뷰만을 리턴해야 한다.

**Modifiers**

UI 개발에 있어서 선언적 패러다임의 핵심 중 하나는 Modifier 메소드이다. 수정자는 정확하고 간결한 방식으로 UI가 어떻게 보여야 할지 정의한다.

1. 메소드 체이닝: 각각의 수정자는 ‘.’구문을 이용해 뷰에 접근, 뷰를 수정하고 새 뷰를 반환한다.
2. 함수적 접근: 이러한 체이닝 방식은 연속되는 함수를 통해 데이터를 변화시키는 함수적 패러다임과 유사하다. SwiftUI에서는 각각의 수정자가 뷰를 변환한다.
3. 불변성: 수정자를 뷰에 적용할 때, 사실 오리지널 뷰는 수정되지 않는다. 수정자는 수정을 통해 새로운 뷰를 생성하는 것이다. SwiftUI의 뷰들은 불변하는 값타입의 구조체이다.
4. 순서의 중요성: 수정자를 적용하는 순서는 중요하다. 수정자는 하향식으로 적용되기 때문에 위에서의 변경 사항이 아래의 인풋으로 들어간다.

**Behind the Scenes**

개념적으로는, 수정자는 원래의 뷰를 감싸 새로운 렌더링이나 행위를 더한 뒤 내보내는 장식 요소 정도로 생각할 수 있다.

내부적으로는, 각각의 수정자가 뷰 계층의 새로운 레이어를 만들어내는 것으로 볼 수 있다. .padding()과 같은 수정자는 기존 뷰를 패딩 공간을 더한 뷰로 감싼 레이어를 더한다.

**강의에서 배운 Modifiers**

.rederingMode(.original) -> cmd+shift+l, 라이브러리 이미지 호출 뒤 이 수정자를 붙여 해당 이미지를 원본 컬러에 맞게 렌더링했다.

.resizable() -> 이미지를 리사이징할 수 있게 한다. 원래 SwiftUI의 Image는 이미지의 원래 사이즈를 보여주고 리사이징은 허용하지 않는다. resizable은 이미지를 주어진 공간에 맞게 늘이거나 줄이기 위해 사용한다. .aspectRatio()나 .frame()과 함께 많이 쓰인다.

.aspectRatio(contentMode: .fit) -> 이미지를 구체적인 차원 안에서 비례적으로 리사이징 하도록 보장한다. 인자값으로 .fit이나 .fill이 들어가는데 .fit은 원본 비율을 맞추면서 전체 이미지를 리사이징하는데 프레임을 벗어나는 경우 더 이상 확장하지 않는다. 따라서 빈 공간이 생길 수 있다. .fill은 비례를 유지하면서 프레임을 가득 채운다. 따라서 프레임 범위 초과하거나 일부가 보이지 않게될 수 있다.

.frame(width: 100, height: 100) -> 이미지가 담긴 프레임의 사이즈를 정한다.

**Extract Subview(Ctrl + right click)**

ContentView의 바디 안에서 섭뷰를 추출해 구조체로 만들 수 있다.

**커스텀 컬러**

Assets에 커스텀 컬러를 만들어 관리할 수 있다.

**Preview Content**

자주 사용되는 버튼이나 뷰들은 따로 이곳에 모아두고 사용한다.

**@State, @Binding**

프로퍼티 래퍼로 애플리케이션의 데이터 플로우나 상태를 관리하기 위해 사용한다. @State와 @Binding은 SwiftUI의 반응형 프레임워크에서 특별히 중요한 역할을 한다.

@State의 목적은 SwiftUI의 뷰에서 상태관리가 가능한 지역 변수를 만드는 데 있다. 오너십이 @State 변수를 가진 view에 종속된다. SwiftUI는 이 값이 바뀔 때 뷰가 바뀌도록 업데이트한다.

@Binding은 데이터를 저장하는 프로퍼티와 수정된 데이터를 보여주는 뷰 사이에 2 방향의 연결을 생성한다. 자식 뷰에 상태를 전달시키기 위해 존재한다. @Binding은 @State와 다르게 오너십이 존재하지 않는다. @Binding은 부모 뷰에 의해 참조되는 데이터로 기능한다. @Binding은 뷰가 다른 뷰와 상태를 공유해야 할 때 사용한다. 자식 뷰에 상태를 바인딩할 때는 바인딩할 변수 앞에 $를 붙인다. 자식 뷰에서 부모 뷰에 상태를 전달할 변수에도 마찬가지로 $를 붙인다.

**앱 발전시키기**

시간될 때 API 붙여서 앱 완성하기

**2단계. SwiftUI 프레임워크**

사실 1단계에서 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.

**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. 수정자는 계층적으로 동작하며 따라서 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. 뷰빌더늬 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 캐릭터를 입힐 수는 있지만 전반적으로는 그렇다고 한다. 대신 SwiftUI는 UI 개발에 비해 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.

**에셋에 아이콘 저장하기**

**LazyVGrid, LazyHGrid**