**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 더해 반환한다.

**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 범위를 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.

@State는 private와 같이 선언해 멤버와이즈 초기화에 의해 값이 변경되지 않도록 해야 한다. 멤버와이즈 이니셜라이저로 값이 변하는 경우 스토리지 매니저에 의해 충돌이 발생한다.

@State가 오너십을 가지면 @Binding은 자식 뷰에서 부모 뷰에 의해 참조되는 프로퍼티로 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. 원래 @State로 래핑된 프로퍼티는 오너십이 있는 뷰 외에 건드릴 수 없지만 @Binding으로 이어져있는 자식 뷰의 경우는 해당 프로퍼티를 바꿀 수 있다.

**@State와 @Observable을 같이 사용하기**

스테이트는 뷰가 초기화될 때마다 뷰와 함께 초기화된다. 만약 스테이트 프로퍼티가 객체와 같이 많은 연산을 필요로 하는 타입이라면 해당 객체의 클래스명 앞에 @Observable을 붙여서 객체를 한 번만 초기화할 수 있다.

struct ContentView: View {

@State private var library: Library?

var body: some View {

LibraryView(library: library)

.task {

library = Library()

}

}

}

라이브러리 클래스가 옵저버블 클래스이면 .task 수정자 안에서 컨텐트 뷰가 첫 초기화될 때 한 번만 초기화된다.

**앱 발전시키기**

시간될 때 API 붙여서 앱 완성하기