**2차 유인물**

**프로토콜**

프로토콜 채택해 프로퍼티의 값을 변경하는 함수를 구현할 경우, 값 타입인 구조체는 상속한 함수에 mutating 표시를 해야 한다. mutating 키워드를 통해 프로퍼티 값이 변경되는 경우, 메모리에서는 @State 프로퍼티의 값을 바꾸거나 ObservableObject의 프로퍼티 값을 바꾸는 경우와 동일한 일이 발생한다. 값타입이지만 이전 구조체를 폐기하고 바뀐 프로퍼티와 함께 스트럭트 전부를 교체하는 게 아니라 필요한 부분만 바꾸는 식으로 최적화해 교체한다.

**Assertions and preconditions**

Assertions는 런타임에 동작해 코드가 의도한대로 동작하는지 검증한다. Assertions의 값이 true이면 코드가 계속 진행되고 아닐 경우 해당 프로그램의 진행이 중지된다. 프로그램에서 앱을 종료시키는 상황을 예상하는 환경에서 통제하기 위해 사용한다.

**nil-coaliscing operator**

변수가 nil인 경우를 검증하는 3항 연산자를 간편하게 표현하는 식으로 1차 유인물에서 학습했었다. a ?? b와 같이 표현하며 a가 nil인 경우 디폴트로 b를 반환한다.

**closures are reference types**

클로저는 참조형이다. 클로저가 변수로 선언된 프로퍼티를 갖고 있을 경우, 해당 클로저를 상수에 할당해도 클로저의 프로퍼티 값을 변경할 수 있다는 것을 학습했었다.(상수로 선언되어 참조되는 클로저에는 변함이 없지만 참조되는 클로저가 자체적으로 변수를 가짐)

**Auto Closures**

var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

print(customersInLine.count)

// Prints "5"

let customerProvider = { customersInLine.remove(at: 0) }

print(customersInLine.count)

// Prints "5"

print("Now serving \(customerProvider())!")

// Prints "Now serving Chris!"

print(customersInLine.count)

// Prints "4"

오토 클로저로 할당된 customerProvider는 호출 시까지 연산을 미룬다. 호출 시에 인자를 넣지 않아도 되는 구문상의 이점도 있어 가독성을 올릴 수도 있다. 클로저에 담긴 내용에 사이드 이펙트가 있을 수 있거나 연산 비용이 큰 경우 오토 클로저를 활용할 수 있다.

**Property Observers**

프로퍼티 옵저버는 프로퍼티의 값을 관찰하다가 변경이 생기면 호출된다. 소유 프로퍼티, 상속 프로퍼티, 컴퓨티드 프로퍼티에 적용할 수 있다. 상속 프로퍼티는 상속한 클래스에서 해당 프로퍼티를 오버라이딩 해 옵저버를 추가할 수 있고 컴퓨티드 프로퍼티는 옵저버를 더하지 않아도 세터에 옵저버를 넣을 수 있다. willSet은 프로퍼티의 값이 바뀌기 직전에 호출되고 didSet은 프로퍼티의 값이 바뀐 직후에 호출된다.

**property wrappers**

만약 프로퍼티에 특정한 조건을 부여하고 이를 일괄적으로 적용하고 싶다면 프로퍼티 래퍼로 감싼 프로퍼티를 선언할 수 있다.

@propertyWrapper

struct TwelveOrLess {

private var number = 0

var wrappedValue: Int {

get { return number }

set { number = min(newValue, 12) }

}

}

위에서 number는 private로 선언되어 초기화 시에만 접근할 수 있다. 여기서 프로퍼티 래퍼는 number가 항상 12보다 작은 값을 갖는 것을 보장한다.

struct SmallRectangle {

@TwelveOrLess var height: Int

@TwelveOrLess var width: Int

}

var rectangle = SmallRectangle()

print(rectangle.height)

// Prints "0"

rectangle.height = 10

print(rectangle.height)

// Prints "10"

rectangle.height = 24

print(rectangle.height)

// Prints "12"

초기화에서 아무런 값을 세팅하지 않으면 number의 기본값인 0을 가지고 12보다 큰 값을 넣으면 12로 고정된다.

**Optional Chaining**

옵셔널 체이닝은 nil 값을 가질 수 있는 여러 개의 변수, 메소드, 섭스크립트 등을 연결해 하나라도 nil이 발생하면 전체 체인이 실패하도록 하는 방식이다.

forced unwrapping 대신 쓰는 것을 고려할 수 있다.

let roomCount = john.residence!.numberOfRooms

// this triggers a runtime error

위 코드에서 residence가 nil일 경우 런타임 에러가 발생한다.

그러나

if let roomCount = john.residence?.numberOfRooms {

print("John's residence has \(roomCount) room(s).")

} else {

print("Unable to retrieve the number of rooms.")

}

// Prints "Unable to retrieve the number of rooms."

와 같이 옵셔널 변수에 ?를 추가하게 되면 스위프트는 해당 변수가 옵셔널 체이닝에 해당하는 것을 알게 된다. 이 경우 residence가 nil값을 가지더라도 런타임 에러가 발생하지 않는다.