**CS193p(Lecture1) 2023**

**구술적(narrative) 방식 vs 삽화적(vignette) 방식**

구술적 방식은 맥락을 스토리로 만들어 가며 배우는 방식, 삽화적 방식은 한 가지 기능을 구현하기 위해 웹에서 한 사례(현재의 맥락과는 거리가 있는)를 가져와 배우고 익히는 방식, 두 가지 방법을 같이 진행하는 게 이상적

**함수형 프로그래밍**

객체지향은 클래스를 통해 데이터를 캡슐화하고 객체를 정의하는 패러다임이라면 함수형 프로그래밍은 구조체를 통해 행위를 정의하고 어떠한 행위를 하는 구조체인지 정의하는 패러다임이다. 스위프트의 구조체는 객체지향의 클래스처럼 변수와 함수를 갖지만 상속은 하지 않는다. 또 객체지향이 데이터 캡슐화를 기반으로 하는 것과 달리 함수형은 행위의 모듈화를 기반으로 한다.

**Computed Property**

Computed property란 연산되는 속성이라는 말 그대로 어떤 구조체나 클래스에 속해 있는 속성이면서 연산 가능한 클로저를 가지는 속성이다. 일반적으로는 같은 클래스, 구조체 내에 있는 프로퍼티에 접근해 새 프로퍼티를 파생시키기 위해 사용한다.

이런 computed property는 게터를 통해 읽기만 가능한 read-only 속성이 된다. Computed property의 클로저 내용은 사실 “get {}”으로 감싸지는 함수이지만 가독성을 위해 get을 생략하고 {}만 사용하기도 한다. Computed property에 값을 할당하고 싶다면 명시적으로 “set{}” 클로저를 선언해 사용할 수도 있다.

**함수**

모든 함수는 클로저이다. 관례상 이름이 있는 클로저는 함수라고 하고 이름이 없으면 클로저라고 부를 뿐이다. 함수 선언 시 파라미터를 선언하는 방법에는 세 가지가 있다.

1. 와일드 카드를 통한 argument 명 생략

* 호출 시 argument를 명시하지 않아도 된다.

1. Parameter만 선언

* 호출 시 argument는 함수 내부적으로 사용하는 parameter와 동일한 이름을 갖는다.

1. Argument, parameter 모두 선언

* 호출 시 호출하는 쪽은 argument명을 사용하고 함수 내부 이름은 parameter를 사용한다.

인자로 전달되는 Value값(원시값)은 복사된 상수로 전달된다. 따라서 함수 호출 후 외부에서 해당 인자의 값을 바꿔도 함수의 실행에는 아무런 영향을 미칠 수 없다. 또한 value값이 함수 내에서도 상수로 활용되기 때문에 값을 변경할 수 없다.

반면 reference값(객체)으로 전달되는 값들은 주소 그대로 전달된다. 따라서 함수 내부에서 값이 변경되면 실제 객체에도 영향을 미친다.

그 밖에 inout 파라미터가 있는데 hello(name: inout String)와 같이 선언하고 hello(name: &name)과 같이 호출한다. Inout 파라미터는 전달된 밸류값을 참조에 의한 호출로 받아들인다. 따라서 함수 내에서 인자를 수정할 수 있고 그 영향이 밖에 있는 변수에까지 미친다.

함수 선언 시 hello(name: String = “defaultName”)과 같이 타입 옆에 등호를 놓고 값을 할당하면 기본값이 된다. 이 경우 hello()로 호출 시 함수의 인자로 defaultName이 전달된다.

가변 파라미터를 가지는 함수의 경우 hello(of nums: int…){…}와 같이 선언하는데 호출 시 이 경우 가변 파라미터는 int형 배열이 된다. 만약 가변 파라미터 이외에 추가적인 매개변수가 있다면 가변 파라미터 다음에 오는 매개변수는 반드시 argument label을 가져야만 한다. 그렇지 않을 경우 인자가 어디까지 가변 파라미터인지 알 수 없기 때문이다. 또 가변 파라미터는 기본값을 가질 수 없고 하나의 함수는 하나의 가변 파라미터만을 가질 수 있다.

**함수 표기법**

인자명 옆에 ‘:’을 붙인다. 와일드 카드 인자일 경우에도 \_:와 같이 인자가 있음을 명시한다. 만약 인자가 없을 경우 소괄호를 생략해 hello와 같이 표기한다. 함수 표기법은 함수 호출과 다르다.

**함수 타입**

(parameterType) -> ReturnType

이 중 어느 것도 생략하면 안 된다.

둘 다 없을 경우 () -> ()로 표기한다.

**일급 객체 함수의 사용**

**func hello(name: String) {}**

**func hello(\_ name: String) {}**

함수 표기법과 함수 타입은 함수를 일급 객체로 사용하기 위해 필요하다. 함수를 변수나 상수로 이용하기 위한 3가지 방법이 있다.

1. let a = hello

* 함수 이름으로 할당하기. 가장 간단하지만 위 예시처럼 동일한 이름으로 오버로딩 된 함수가 있을 경우 함수를 구분할 수 없어 컴파일 에러 발생

1. let a: (String) -> () = hello

* 함수 타입을 명시하기. 이 경우 다른 파라미터 값으로 오버로딩 된 함수를 구분할 수 있다. 그러나 동일 파라미터로 오버로딩 됐는데 argument와 parameter값이 서로 다른 함수가 있을 경우 컴파일 에러가 발생한다.

1. let a = hello(name:)

* 함수 표기법으로 할당하기. 간편하면서 오버로딩 된 함수도 구분할 수 있다. 하지만 파라미터 타입과 리턴 타입이 없는 함수가 오버로딩 된 경우 2번의 타입 어노테이션 표기를 사용해야 한다.

이후 a에 할당된 함수 사용 시 a()로 사용한다. 이때 함수 타입만 할당 받아 사용하므로 argument label은 사용하지 않는다.

**클로저 표현식**

{(parameters) -> returnType in

실행 구문…

}

(parameters) -> returnType을 클로저 헤드라고 하고

in 실행 구문…을 클로저 바디라고 한다.

**Some 변수**

some 타입은 스위프트에서 굉장히 중요한 역할을 수행한다.

**var** body: **some** View {

VStack {

ScrollView{

cards

}

cardCountAdjusters

}

.padding()

}

강의에서 제시된 body는 some 키워드를 통해 특정 프로토콜(여기서는 View)을 채택하는 어떠한(some) 타입을 변수로 정의한다. some 키워드는 리버스 제네릭이라고도 불린다. 타입을 사전에 정의하지 않고 사용하는 사람이 여러 개의 타입을 추가해가며 사용할 수 있기 때문이다. 위에서는 VStack에 여러 개의 타입을 집어 넣어가며 여러 형태의 뷰를 만들 수 있다. 만약 제네릭처럼 VStack에 들어갈 모든 타입을 사전에 정의하고 사용한다면 사전에 정의해야 할 코드량이 기하급수적으로 늘어난다. 뿐만 아니라 사용하는 시점의 개발자도 뷰를 새로이 정의하고 수정할 때마다 복잡하게 중첩된 제네릭을 번거롭게 수정해야 한다. some 키워드는 코드 작성 시점에서는 타입을 추상화해 생산성을 올리고 컴파일 타임에 타입 추론을 통해 타입을 구체화하는 강력한 기능을 지원한다.

**View와 Window**

흔히들 윈도우를 액자, 뷰를 컨텐츠에 비교한다. 윈도우는 뷰가 들어갈 틀이 된다. 뷰는 스스로 컨텐츠를 보여줄 수도, 다른 뷰를 위한 컨테이너가 될 수도 있다.

뷰는 왼쪽 상단의 모서리를 원점(0, 0)으로 하는 좌표계를 가진다. 좌표계는 오른쪽 아래로 확장된다. 좌표값은 해상도와 상관없는 부동소수점 수이다. 뷰 프레임(절대기준)과 바운드(상대기준)으로 나타낸다.

**Class, Struct, Enum의 차이(블록 단위)**

클래스는 참조 타입이고 열거형과 구조체는 값 타입이라는 점이 가장 큰 차이이다. 또 클래스는 상속이 가능하지만 열거형과 구조체는 불가능하다. 값에 의한 호출과 참조에 의한 호출의 차이를 생각하자.