**CS193p(1차 유인물, L1~L3 범위) + 스위프트 5.9.2 공식문서**

1차 유인물의 목적은 스위프트의 기본 요소들(변수, 제어흐름, 메소드, 프로퍼티, Arrays and Strings)을 이해하면서 스위프트가 강점으로 가지는(라벨 인자, 패턴 매칭 스위치문, 일급 객체로서의 함수, 객체지향과 함수형 프로그래밍의 혼합…) 부분을 익숙하게 하는 것이다.(4강 듣기 전에 1차 유인물 다 보기)

**스위프트란…**

스위프트는 현대적인 프로그래밍 패턴을 채택해 다른 프로그래밍 언어들에서 흔히 발생하곤 하는 문제들을 피한다.

1. 변수는 항상 사용하기 전에 선언되어야 한다.
2. 인덱스는 경계값 오류를 발생시키기 전에 미리 확인한다.
3. 정수값은 오버플로우를 발생시키기 전에 확인한다.
4. 옵셔널 타입은 Nil 값을 명시적으로 다루도록 보장한다.
5. 메모리는 자동으로 관리된다.
6. 예기치 못한 실패 시 에러 핸들링이 통제된 회복을 허용한다.

스위프트는 현대적 하드웨어 위에서 빠르고(단순한 문법), 안정적(강력한 타입추론, 패턴 매칭)이고, 표현력 있는 소프트웨어를 정확하게 구현하기 위해 만들어졌다. 스위프트는 쓰기 쉽고 읽기도 쉬운 코드를 지향한다.

**Swift Tour**

스위프트는 여타의 언어와 같이 Main()으로 시작하지 않고 문장을 세미콜론으로 마무리 할 필요가 없다. 또한 기본 출력 함수인 print(“”)를 사용하기 위해 라이브러리를 불러올 필요도 없다. 메인 함수가 없으므로 전역으로 선언된 코드는 엔트리 포인트가 된다.

**간단한 변수**

let으로 선언된 변수는 상수, var로 선언된 변수는 변수가 된다. 상수는 컴파일 시점에 알려질 필요가 없지만 반드시 한 가지 값으로만 선언되어야 한다. 타입을 선언하면 반드시 할당되는 값과 타입이 일치해야 하지만 명시적 선언이 강제되지는 않는다. 컴파일러가 값을 통해 자동으로 타입을 추론하기 때문이다. 다만 초기에 할당되는 값으로 타입을 추론하기 어렵다면(ex. 원하는 타입은 double인데 초기값이 정수인 경우) let explicitDouble: Double = 70과 같이 변수명 뒤에 콜론을 붙이고 뒤에 타입을 표기한다.

**명시적 형변환**

묵시적 형변환은 지원하지 않는다. 타입이 다른 값을 연산하려면 반드시 명시적 형변환을 해야 한다. Ex. “The number is ” + String(number)

문자열 안에서 다른 변수를 연산하려면 해당 변수들을 ₩()로 감싸면 된다. 또 문자열의 줄바꿈을 ₩n을 통해 표기하지 않고 화면 그대로 표현하고 싶다면 “””를 통해 문자열을 여닫으면 된다.

**배열과 딕셔너리**

인자를 대괄호[]로 감싸고 컴마로 구분해 선언하면 된다. 딕셔너리는 대괄호로 감싸고 콜론을 통해 키-밸류를 구분한다. 배열의 경우 .append()를 통해 값을 삽입하면 자동으로 길이가 늘어난다.

fruits = []

occupations = [:]

위는 빈 배열, 아래는 빈 딕셔너리, 만약 타입 추론을 하기 힘든 상황이라면

let emptyDictionary: [String: Float] = [:]와 같이 선언해준다.

**제어흐름**

If나 switch의 조건절에 소괄호는 선택사항이지만 대괄호로 이행 사항을 감싸는 것은 필수. 또한 If조건의 조건절은 반드시 불리언 값이어야 한다. 특징적으로 변수에 조건문을 할당하는 것이 가능하다.

let scoreDecoration = if teamScore > 10 {

"🎉"

} else {

""

}

print("Score:", teamScore, scoreDecoration)

// Prints "Score: 11 🎉"

옵셔널 변수는 타입 뒤에 물음표를 붙여 해당 타입이 nil일 수도 있음을 명시한다.

var optionalString: String? = "Hello"

print(optionalString == nil)

// Prints "false"

var optionalName: String? = "John Appleseed"

var greeting = "Hello!"

if let name = optionalName {

greeting = "Hello, \(name)"

}

If let이 함께 쓰인 경우 if가 optionalName이 값을 가지고 있는지 먼저 체크한다. 만약 값을 가지고 있다면 이 값은 name에 할당되며 이는 다시 if의 범위 안에 들어가게 된다. If 범위 안에서 {}내의 greeting = “Hello, \(name)”이 실행되게 된다.

여기서 if를 변수 선언 앞에 둬 optionalName이 nil인 경우에 발생할 수 있는 런타임 오류를 막고 있다.

let nickname: String? = nil

let fullName: String = "John Appleseed"

let informalGreeting = "Hi \(nickname ?? fullName)"

여기서는 ?? 기호를 사용해 nickname이 nil일 경우 디폴트 값인 fullName을 할당하도록 한다.

스위치 문은 타입을 가리지 않고 적용 가능하다.

let vegetable = "red pepper"

switch vegetable {

case "celery":

print("Add some raisins and make ants on a log.")

case "cucumber", "watercress":

print("That would make a good tea sandwich.")

case let x where x.hasSuffix("pepper"):

print("Is it a spicy \(x)?")

default:

print("Everything tastes good in soup.")

}

// Prints "Is it a spicy red pepper?"

var total = 0

for i in 0..<4 {

total += i

}

print(total)

// Prints "6"

For문은 …와 ..<을 통해 인덱스의 범위를 표현한다.

0…10은 0이상 10이하, 0..<10은 0이상 10 미만

**함수 선언과 클로저**

func greet(person: String, day: String) -> String {

return "Hello \(person), today is \(day)."

}

greet(person: "Bob", day: "Tuesday")

->는 함수와 리턴 타입을 구분한다.

매개변수의 이름은 함수의 인자에 붙는 라벨이다. 라벨은 함수를 사용할 때 디폴트로 사용된다. 만약 라벨을 디폴트로 쓰고 싶지 않다면 함수를 선언할 때 소괄호 가장 앞에 \_를 붙여준다.

func greet(\_ person: String, on day: String) -> String {

return "Hello \(person), today is \(day)."

}

greet("John", on: "Wednesday")

함수 선언 시 언더바 사용으로 함수를 사용할 때 매개변수에 라벨링을 하지 않고 사용하고 있다.

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

var min = scores[0]

var max = scores[0]

var sum = 0

for score in scores {

if score > max {

max = score

} else if score < min {

min = score

}

sum += score

}

return (min, max, sum)

}

let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])

print(statistics.sum)

// Prints "120"

print(statistics.2)

// Prints "120"

스위프트는 튜플을 활용해 다중 반환을 지원한다. 반환된 값 중 특정한 값을 호출할 때는 반환값의 라벨명을 이용해도 되고 튜플의 인덱스를 사용해도 된다.

함수의 내장 함수는 내장 함수를 선언하는 바깥 함수의 변수를 가져다 쓸 수 있다. 또한 스위프트에서 함수는 일급 객체이므로 변수에 값을 할당하는 것은 물론 반환형과 매개변수로 사용할 수도 있다.

클로저는 일종의 함수로 생성 당시 자신의 주변 환경을 기억하는 함수이다. 일례로 앞에서 살펴본 내장 함수는 외장 함수의 변수를 가져다 쓸 수 있었다.

하나의 내장 함수가 실행될 때 내부적으로 다음과 같은 과정을 거친다.

실행 컨텍스트 스택에 해당 함수의 실행 컨텍스트 스택이 쌓인다. 이때 이 함수의 변수 객체, 스코프 체인(전역->외장->내장), 그리고 this에 바인딩할 객체가 생성된다. 이 스코프 체인에는 전역 스코프를 가지는 전역 객체와 외장 스코프를 가지는 외장 함수 활성 객체, 그리고 자신의 스코프를 가지는 내장 함수 활성객체가 순서대로 바인딩된다. 이러한 스코프 체인의 바인딩을 렉시컬 스코프라고 한다.

내장 함수가 외장 함수의 변수를 가져다 쓸 수 있는 이유는 실행 컨텍스트에서 렉시컬 스코프 체인을 검색할 수 있도록 지원하기 때문이다(자바스크립트의 경우 자바스크립트 엔진)

이는 외부 함수가 종료되고 내장 함수가 외부 함수보다 더 오래 유지되는 경우에도 적용된다. 이때 종료된 외부 함수가 가지고 있던 변수를 자유 변수라고 한다. 앞에서 클로저란 생성 당시 자신의 주변 환경을 기억하는 함수라고 했다. 실행 컨텍스트를 통해 더 엄밀히 말하면 클로저는 생성 당시 자신의 렉시컬 환경을 참조할 수 있는 변수인 것이다. 이때 참조되는 자유 변수는 실제 외부 함수의 변수로 원래 외부 함수 객체에서 새롭게 복제된 것이 아닌 원래 객체라는 점에 주의하자. 클로저라는 말은 이렇듯 내부 함수가 자유 변수를 비롯한 생성 당시 렉시컬 환경에 둘러 쌓여 갇혀 있다(closed)는 맥락에서 이해하자.

스위프트의 함수는 클로저이다.

**객체와 클래스**

class Shape {

var numberOfSides = 0

func simpleDescription() -> String {

return "A shape with \(numberOfSides) sides."

}

}

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

새 객체를 생성하려면 클래스명 뒤에 소괄호를 친다.

.은 다른 언어에서와 마찬가지로 객체 속성에 접근하기 위한 연산자이다.

생성자는 클래스 내의 init()메서드이다. 객체 속성에 파라미터 값을 부여할 경우

init(name: String) {self.name = name}과 같이 사용한다. 자신의 속성을 가리킬 때 this.가 아닌 self를 사용한다. 생성자를 만들 때는 init()메서드로 객체를 정의하지만 사용할 때는 위에서 학습한 것처럼 클래스명()로 생성한다.

class Square: NamedShape {

var sideLength: Double

init(sideLength: Double, name: String) {

self.sideLength = sideLength

super.init(name: name)

numberOfSides = 4

}

func area() -> Double {

return sideLength \* sideLength

}

override func simpleDescription() -> String {

return "A square with sides of length \(sideLength)."

}

}

let test = Square(sideLength: 5.2, name: "my test square")

test.area()

test.simpleDescription()

위는 상속의 예시이다. 오버라이딩의 경우 위와 같이 오버라이딩을 명시하지 않으면 에러가 난다.

**열거형과 구조체**

열거형을 선언하고 열거형 변수를 나열하면 자동적으로 0부터 1씩 점증하는 값을 갖게 된다.

enum Rank: Int {

case ace = 1

case two, three, four, five, six, seven, eight, nine, ten

case jack, queen, king

func simpleDescription() -> String {

switch self {

case .ace:

return "ace"

case .jack:

return "jack"

case .queen:

return "queen"

case .king:

return "king"

default:

return String(self.rawValue)

}

}

}

let ace = Rank.ace

let aceRawValue = ace.rawValue

위 코드는 초기값에 1을 할당했기에 값이 1로 시작하게 된다. 특정 값을 할당하고 싶다면 이를 명시하면 된다.

**동시성**

동시성은 병렬성과 구분되는 개념으로 둘의 차이를 잘 알고 있어야 한다. 동시성은 한 번에 많은 일을 다루는 것이고 병렬성은 한 번에 많은 일을 진행하는 것이다.

스위프트는 자바 스크립트처럼 async와 awiat 예약어를 사용해 비동기 함수를 생성한다.

func fetchUsername(from server: String) async -> String {

let userID = await fetchUserID(from: server)

if userID == 501 {

return "John Appleseed"

}

return "Guest"

}

스위프트에서 async 는 async let처럼 let과 함께 사용할 경우 병렬성을 허용하기도 한다.

func connectUser(to server: String) async {

async let userID = fetchUserID(from: server)

async let username = fetchUsername(from: server)

let greeting = await "Hello \(username), user ID \(userID)"

print(greeting)

}

위와 같은 코드는 async let이 붙은 비동기 함수들끼리 다른 비동기 함수들과 병렬적으로 실행되도록 허용한다.

만약 동기 코드 내부에서 비동기 함수를 동기 함수처럼 호출해 사용하고 싶으면 Task{}로 비동기 함수를 감싸서 사용한다.

Task {

await connectUser(to: "primary")

}

// Prints "Hello Guest, user ID 97"

액터는 클래스와 유사하지만 비동기 처리에서 차이가 있다. 액터는 동일한 액터 객체가 여럿 있을 때 서로의 비동기 함수 상호작용을 안전하게 처리한다. 비동기 함수가 서로 얽혀 있는 경우에도 동일 엑터 객체들의 비동기 함수는 안전하게 상호작용할 수 있음을 보장하는 것이다.

actor ServerConnection {

var server: String = "primary"

private var activeUsers: [Int] = []

func connect() async -> Int {

let userID = await fetchUserID(from: server)

// ... communicate with server ...

activeUsers.append(userID)

return userID

}

}

let server = ServerConnection()

let userID = await server.connect()

위 상황에서 여러 개의 connect()함수가 각기 다른 액터들에 의해 호출된다면 위에서 말한 것처럼 안전한 상호작용이 보장된다. 다만 이를 위해 동작중인 액터 함수가 끝나기까지 대기 시간이 생길 수 있음을 유념해야 한다.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*프로토콜과 확장(super important)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

프로토콜 예약어를 사용해 프로토콜을 선언할 수 있다.

protocol ExampleProtocol {

var simpleDescription: String { get }

mutating func adjust()

}

클래스, 열거형, 구조체 모두 프로토콜을 ‘채택(adopt)’할 수 있다.

class SimpleClass: ExampleProtocol {

var simpleDescription: String = "A very simple class."

var anotherProperty: Int = 69105

func adjust() {

simpleDescription += " Now 100% adjusted."

}

}

var a = SimpleClass()

a.adjust()

let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {

var simpleDescription: String = "A simple structure"

mutating func adjust() {

simpleDescription += " (adjusted)"

}

}

var b = SimpleStructure()

b.adjust()

let bDescription = b.simpleDescription

클래스의 메소드 구현과 구조체 내 메소드 구현의 차이에 주목하자.

**에러 관리**

어떤 형식으로든 에러 프로토콜을 채택하기만 하면 에러를 표현할 수 있다.

아래는 열거형으로 에러를 표현한다.

enum PrinterError: Error {

case outOfPaper

case noToner

case onFire

}

에러 발생 시 throws로 에러가 발생할 수 있는 코드임을 알리고 이를 반환형에 애로우 포인터로 연결해 선언한다.

func send(job: Int, toPrinter printerName: String) throws -> String {

if printerName == "Never Has Toner" {

throw PrinterError.noToner

}

return "Job sent"

}

에러 선언은 throw로 한다.

do {

let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")

print(printerResponse)

} catch {

print(error)

}

// Prints "Job sent"

이후 try catch와 유사하게 do catch한다. do문 안에서 try를 에러가 발생 가능한 함수 앞에 둔다는 점을 제외하면 try-catch와 큰 차이가 없다.

let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")

let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")

위 표현과 같이 간단하게 표현할 수도 있다. 다만 try?문은 발생한 에러를 폐기하고 이를 nil로 반환한다는 점에 주의하자. 옵셔널 변수처럼 nil값과 정상값을 허용하는 옵셔널 try라고 생각하면 된다.

var fridgeIsOpen = false

let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(\_ food: String) -> Bool {

fridgeIsOpen = true

defer {

fridgeIsOpen = false

}

let result = fridgeContent.contains(food)

return result

}

if fridgeContains("banana") {

print("Found a banana")

}

print(fridgeIsOpen)

// Prints "false"

defer{} 키워드는 키워드 이하 명령을 어떤 일이 있어도 실행 후 함수를 리턴시킨다. 이는 오류 발생 시에도 마찬가지이다. 함수에 대한 일관된 세팅이 필요할 때 이 함수를 사용한다.

**제네릭스**

자바의 제네릭스와 비슷하다. 특수한 차이가 있다면 사용하면서 알아가자.

**The Basics**

**Implicitly unwrapped optionals**

옵셔널 타입 뒤에 ?가 아닌 !를 달아 언래핑을 암묵적으로 하게 만드는 선언이다.

반드시 nil이 아닌 유의미한 값이 들어갈 것임을 약속하는 선언이라고도 한다.

그럴 거면 상수 타입으로 선언하면 되는데 왜 굳이 옵셔널 타입으로 선언하는가 하는 의문이 들었다.

알아본 결과 앱을 만들다 보면 미리 선언은 했지만 클라이언트가 사용하기 이전까지는 값이 정해지지 않은 변수들이 생기기 마련이다. 단순히 생각해봐도 상호작용이 이루어지지 않은 버튼이라든가 입력되지 않은 창 같은 것들이 그렇다. 이런 상호 작용 이전의 모듈들은 모두 상호작용 이전에 정의되어 실행을 대기하고 있어야 한다.

이런 인터페이스로 받을 값을 상수 타입으로 선언 시 실제 프로그램에서 사용 하기까지 만일을 위해 nil체크 로직이 빈번히 들어갈 수 있다. 이런 코드는 가독성을 떨어뜨린다. 코드를 간결하게 만들기 위해 암묵적으로 값에 접근하는 implicitly unwrapped optionals를 선언하면 중간 로직에서 nil체크를 할 필요도 없고 사용 시에도 일반 변수처럼 사용하면 된다.

하지만 위 장점은 사용 시점에 해당 옵셔널 타입에 반드시 값이 할당된다는 약속을 지켰을 때에만 유효하다. 사용 시점을 어기고 값이 할당되지 않은 런타임에 해당 변수에 접근하는 코드가 있다면 에러가 발생할 수 있다.

**Basic Operators**

**nil-coalescing operator**

아래 코드는 옵셔널 변수를 언래핑하는 3항 연산식이다. a가 옵셔널 변수일 때

a != nil ? a! : b

그리고 아래는 위의 3항식을 nil-coalescing operator로 바꾼 것이다.

var isValueOrDefault = a ?? defaultValue

여기서 옵셔널 a가 nil이면 디폴트값이 저장되고 값이 있을 경우 a가 저장된다.

**Strings and Characters**

스위프트에서 스트링은 값 타입이다. 상수로 선언된 스트링을 합쳐 변수로 선언한 새 스트링을 초기화할 수 있다. Copy on Write 정책이 적용되기 때문에 값이 같은 스트링을 복사해 사용할 때 메모리를 효율적으로 사용한다.

**유니코드 음소 단위**

let precomposed: Character = "\u{D55C}" // 한

let decomposed: Character = "\u{1112}\u{1161}\u{11AB}" // ᄒ, ᅡ, ᆫ

// precomposed is 한, decomposed is 한

위 코드처럼 유니코드는 하나의 음절로 표현할 수도 있고 음소단위로 표현할 수도 있다. 아래의 디컴포즈드도 실제 캐릭터 타입으로 출력시에는 ‘한’으로 출력된다. 스위프트는 유니코드를 정규화 해 처리하기 때문에 두 문자는 같은 문자로 간주된다.

**Substring과 String**

메소드나 범위 지정 연산을 통해 스트링에서 서브스트링을 구하게 되면 서브스트링의 길이가 스트링보다 짧을 지라도 같은 메모리 공간을 공유한다. 서브스트링이 스트링 메모리의 일부를 드러내는 것이다.

하지만 이 서브 스트링을 새로운 스트링에 할당하게 되면 새 스트링은 새 메모리 영역을 차지하게 된다. 스트링과 서브스트링은 모두 StringProtocol을 채택한다.

**Collection Types**

**Arrays**

배열의 타입이 호환 가능하면 +로 두 개의 배열을 합칠 수 있다.

**Optional Tuple Return Type**

튜플 리턴 시에 (Int, Int)?와 같이 표현하면 두 개의 튜플 값이 모두 없을 시 nil을 반환한다.

**Function**

**In-out parameter**

C에서 포인터 변수와 비슷하다. 타입 앞에 inout으로 선언하고 변수형 인자 앞에 &를 붙여 참조형으로 전달한다.

**Function Types**

함수 타입은 함수에 들어갈 파라미터 형과 리턴형으로 구성된다. 동일한 파라미터 형과 리턴형을 가지는 함수가 있다면 해당 함수 타입의 변수에 이 함수를 할당할 수 있다.

func example(\_ a:int, \_ b: int) -> int {

return = a + b

}

var funcVal: (int, int) -> int = example

과 같이 할당할 수 있다.

위에서는 변수에 할당했지만 함수 타입을 파라미터에 인자로 전할 수도 있다.

var funcEx(\_ fir: int,\_ sec: int,\_ c: (int, int) -> int) -> int {

…

}

funcEx(a, b, funcVal)

마찬가지로 함수 타입을 리턴타입으로 쓸 수도 있다.

위와 같이 선언한 함수들은 모두 전역 함수이다. 앞서 스위프트는 메인 함수 없이 전역 함수 단위로 실행한다고 배웠었다. 만약 특정 함수를 특정 조건 하에서 실행하고 싶다면 전역 함수로 감싼 내장 함수를 전역 함수 범위 안에 지역 범위로 선언해서 특정 조건을 만족 시 반환하게 하면 된다.

**Closures**

이전에 클로저는 익명 함수라고 배웠다. 공식 문서는 이렇게 정의한다.

클로저는 독립적으로 기능하며 기능 단위로 전달될 수 있는 블록이다.

또 ‘캡처’의 관점에서 클로저의 범위를 나눠 정의할 수도 있다. 여기서 캡처라는 개념은 생소한데 이후에 설명이 나오니 일단은 암기하고 넘어가면 된다.

* 전역 함수는 이름이 있고 아무 것도 캡처하지 않는 클로저이다.
* 중첩 함수는 이름이 있고 자신을 감싸는 함수의 변수를 캡처하는 클로저이다.
* 클로저는 이름이 없고 간소화된 함수 표현이다. 자신을 둘러싼 맥락의 변수를 캡처할 수 있다.

이름과 캡처가 세 종류의 클로저를 구분하는 핵심이 된다.

스위프트의 클로저 표현식은 간결하고 빠르게 작성되지만 있어야 할 것은 다 있는 최적화된 표현을 지향한다. 다음과 같은 최적화가 지원된다.

1. 사용 맥락에서 파라미터와 리턴 타입의 추론 지원
2. 단일 표현 클로저에서 암묵적인 리턴 지원
3. 간소화된 인자 이름
4. 클로저 맥락 추적

**Capturing Values**

클로저는 자신이 정의된 맥락을 둘러싸고 있는 변수와 상수들을 캡처할 수 있다. 그리고 더 이상 원본의 컨텍스트가 존재하지 않는 상황에서도 자신의 범위 바깥에 있는 맥락에서 그러한 변수, 상수를 가져와 수정하고 참조할 수 있다.

가장 단순한 형태의 캡처링 클로저는 중첩 함수라고 할 수 있다. 중첩 함수는 자신을 둘러싼 함수에 선언된 변수나 해당 함수에 인자로 건네어 진 변수의 값을 읽고 참조, 수정할 수 있다. 자신을 둘러싼 함수의 값을 가져와 사용하기 때문에 독립적으로는 사용할 수 없으나 내장 함수로서는 기능할 수 있게 된다. 만약 자신을 둘러싼 함수가 반환값으로 중첩함수를 반환한다면 외장 함수의 종료 이후에 중첩 함수가 동작한다. 그러나 이 경우에도 중첩함수를 자신의 맥락에 존재하는 변수들을 사용할 수 있기 때문에 정상적으로 동작한다. 이 모든 과정의 메모리 관리는 언어적 차원에서 핸들링한다. 추가적으로 클로저는 참조 타입이다.

**Enumeration**

열거형을 스트링에 부여된 정수값으로 한정하는 다른 언어들과 달리 스위프트의 열거형은 보다 유연하다. 열거형 케이스는 스트링, 캐릭터, 정수와 부동 소수점 수에 이르기까지 다양한 값을 가질 수 있다. 또한 타 언어의 클래스가 지원하던 computed variable, 메소드, 생성자 등을 지원한다. 또한 구조체나 클래스처럼 프로토콜을 채택할 수도 있다. 이 모든 확장은 기능성(functionality)을 증가시키기 위함이다.

enum CompassPoint {

case north

case south

case east

case west

}

스위프트 열거형은 문자열을 쓰기만 해서는 선언되지 않는다. 반드시 앞에 case 키워드를 붙여야 한다. 또 열거형 케이스들이 C처럼 0부터 증가하는 정수 값을 갖지도 않는다. 각 case의 기본값은 각 case의 이름인 north, south, east, west가 된다.(String 프로토콜 채택 시)

**Structures and Classes**

스위프트의 구조체나 클래스는 다른 언어의 객체 개념 보다는 functionality에 가깝다. functionality를 어떻게 번역할 지 모르겠지만 내가 느끼기엔 함수성과 기능성이 합쳐진 함수적 기능성 정도로 번역해야 할 것 같다.

스위프트의 클래스와 구조체는 프로그램을 쌓아올리기 위한 블럭 단위이다. 스위프트에서는 개발자가 인터페이스와 그에 대한 구현체를 작성할 필요가 없다. 스위프트는 한 파일 안에서 구조체와 클래스를 작성하면 외부의 인터페이스가 해당 모듈들을 가져다 쓸 수 있게 한다.(제대로 이해한 건지 모르겠지만 잘못 이해한 경우 수정한다.)

상기의 이유로, 또 객체지향 보다는 functionality(함수적 기능성)에 주목하기 위해 스위프트에서 객체는 객체 대신 인스턴스라는 말로 표현하는 게 좋다.

클래스나 구조체는 그 자체로 어떤 구체적인 행동을 할 지 드러내지 않는다. 다만 구체적인 행동이 어떠할 지에 대한 밑그림을 제공할 뿐이다.

사실 지금까지 다뤄온 거의 대부분의 타입들, 정수, 부동소수점 수, 불리언, 문자열, 어레이, 딕셔너리 등등은 전부 구조체로 구현된 밸류 타입들이다. 스위프트에서 모든 구조체와 열거형은 밸류 타입이다. 이 말은 개발자가 작성하든, 스위프트가 제공하는 걸 사용하든 속성이 존재하는 모든 인스턴스들은 값이 전달될 때마다 복제됐다는 것이다.

반면에 클래스는 참조 타입으로 값을 전달할 때 자신의 포인터도 함께 보낸다. 참조 타입은 주의해서 사용해야 한다. 특정 시점에 값이 변경되고 한참 후에 다시 참조 타입을 사용한다면 나중에는 해당 참조타입의 값이 언제, 어디서 변경되었는지 유추하기 힘들다. 또한 상수로 선언한 참조타입의 속성이 변경되는 것은 참조 타입 자체가 변경되는 게 아니다. 참조 타입의 속성 중 변수로 선언된 속성이 변경되는 것일 뿐이다. 속성의 값이 변경된 후에도 참조되는 인스턴스의 메모리 주소는 동일하다.

**Properties**

일반적인 속성(저장 속성)들은 저장되고 computed properties는 연산한다. computed properties는 클래스, 구조체, 열거형 모두에 적용 가능하다. 하지만 저장 속성은 클래스와 구조체에만 선언할 수 있다. 추가적으로 속성 감시자를 둘 수도 있는데 이는 개발자가 특정한 액션에 반응할 수 있게 한다.

**Stored Properties**

만약에 상수로 선언되어 초기화한 구조체가 있다면 해당 구조체의 속성이 var로 선언되었어도 값을 바꿀 수 없다.

let rageOfFourItems = FixedLengthRange(firstValue: 0, length: 4)

rangeOfFourItems.firstValue = 6 // 컴파일 에러 발생

이는 구조체가 값 타입으로 복제되기 때문이다.

**Lazy Stored Properties**

stored property 앞에 lazy 키워드를 선언해 지연 저장 속성으로 선언할 수 있다. 이렇게 되면 지연 저장 속성은 해당 속성이 처음 사용되기 이전까지 연산되지 않는다.

*지연 속성은 var 속성에만 붙일 수 있다. 이런 지연 속성은 간혹 인스턴스의 생성자가 작동한 뒤에도 여전히 지연된 값으로 남아있을 수 있는데 상수 속성은 이를 허용하지 않기 때문이다. 상수로 선언된 속성은 생성자가 작동하기 전에 반드시 값을 가지고 있어야 한다.*

지연 속성은 초기값이 외부 요인에 의존적인 속성에 적용하면 유용하다. 이런 값을 갖는 속성들은 생성자가 실행된 뒤에도 여전히 값의 할당을 유예할 수 있기 때문이다.

class DataImporter {

/\*

DataImporter is a class to import data from an external file.

The class is assumed to take a nontrivial amount of time to initialize.

\*/

var filename = "data.txt"

// the DataImporter class would provide data importing functionality here

}

class DataManager {

lazy var importer = DataImporter()

var data: [String] = []

// the DataManager class would provide data management functionality here

}

let manager = DataManager()

manager.data.append("Some data")

manager.data.append("Some more data")

// the DataImporter instance for the importer property hasn't yet been created

위 예시 코드에서 lazy var importer의 값은 매니저가 데이터 매니저로 초기화된 뒤에도 여전히 지연된 채로 머문다. DataManager 클래스의 목적은 data 변수에 스트링 값을 저장하는 것이다. 만약 읽어와야 할 파일이 생긴다면 importer를 통해 해당 파일의 스트링을 읽어온다. 이 작업은 상대적으로 많은 리소스와 시간을 필요로 하는 작업이다. 따라서 외부에서 읽을 파일이 없다면 굳이 importer를 초기화하지 않는다.

print(manager.importer.filename)

위와 같이 임포터에 엑세스하게 되면 importer 객체가 처음으로 생성된다. 만약 멀티 스레드가 동시에 초기화되지 않은 지연 속성에 접근하게 되면 해당 속성은 여러 번 초기화될 수도 있으니 조심해야 한다.

**Computed Properties**

연산 속성은 클래스, 구조체, 열거형 내에 모두 선언 가능하며 값을 저장하지 않는다. 대신 게터와 옵셔널 세터를 이용해 다른 속성들을 연산한다.

**Method**

메소드는 인스턴스나 타입의 일부로서 기능하는 함수를 말한다. 클래스, 구조체, 열거형 등 모든 인스턴스가 메소드를 정의할 수 있다.

**Instance Methods**

인스턴스 메소드는 클래스, 구조체, 열거형 안에 선언된 메소드이다. 인스턴스 메소드는 인스턴스의 속성에 접근하거나 인스턴스의 목적에 맞는 함수적 기능성을 제공함으로써 functionality를 지원한다. 함수와 형식은 완전히 같다.

**Modifying Value Types from Within Instance Methods**

원래 값 타입 인스턴스 변수의 속성은 변경할 수 없다. 그러나 mutating 키워드를 붙인 메소드를 선언하면 속성 변수를 바꿀 수 있다. mutating 키워드가 있으면 컴파일러는 상태를 바꿀 인스턴스가 있다는 사실을 알게 된다. mutating 키워드가 호출되면 스위프트는 수정된 속성값을 가진 새 인스턴스를 생성한다. 이는 수정 사항이 원래의 인스턴스에 영향을 미치면 안된다는 스위프트의 철학을 반역한다.

mutating 함수는 상수로 선언된 인스턴스에 적용할 수 없고 참조형 타입인 클래스에서 사용할 수 없다.

enum TriStateSwitch {

case off, low, high

mutating func next() {

switch self {

case .off:

self = .low

case .low:

self = .high

case .high:

self = .off

}

}

}

var ovenLight = TriStateSwitch.low

ovenLight.next()

// ovenLight is now equal to .high

ovenLight.next()

// ovenLight is now equal to .off

위 코드는 mutating 함수로 3상태 스위치를 구현한다.

**Subscripts**

섭스크립트는 콜렉션에서 멤버 요소에 접근하는 간단한 표현이다. 메소드를 사용하지 않고 간단하게 arr[idx]로 배열의 요소에 접근할 때 우리는 [idx]로 섭스크립트를 표현한다. 스위프트에서는 하나의 타입에 여러 개의 섭스크립트를 정의할 수 있다. 또 섭스크립트를 오버로딩해 특정한 인덱스 타입을 넘기고 원하는 값을 찾아올 수도 있다. 섭스크립트는 일차원 적으로 제한되지 않고 확장 가능하다. 원한다면 섭스크립트에 여러 개의 인풋 파라미터를 넣을 수 있도록 정의할 수도 있다.

subscript(index: Int) -> Int {

get {

// Return an appropriate subscript value here.

}

set(newValue) {

// Perform a suitable setting action here.

}

}

섭스크립트 선언은 위와 같이 섭스크립트 키워드를 통해 한다. 선언 방식은 함수와 computed property의 선언과 비슷하다.

subscript(index: Int) -> Int {

// Return an appropriate subscript value here.

}

computed property처럼 읽기 전용인 경우 위와 같이 간소화 할 수 있다.

struct TimesTable {

let multiplier: Int

subscript(index: Int) -> Int {

return multiplier \* index

}

}

let threeTimesTable = TimesTable(multiplier: 3)

print("six times three is \(threeTimesTable[6])")

// Prints "six times three is 18

**Type Subscripts**

위에서는 특정한 타입의 인스턴스에 접근하는 섭스크립트를 알아봤다. 스위프트에서는 타입 그 자체를 호출하도록 섭스크립트를 정의할 수 있다. 이런 섭스크립트를 타입 스크립트라고 한다. subscript 앞에 static 키워드를 붙여 선언하며 클래스에서는 class 키워드를 사용해 서브 클래스가 해당 스크립트를 오버라이드 하도록 할 수도 있다.

enum Planet: Int {

case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune

static subscript(n: Int) -> Planet {

return Planet(rawValue: n)!

}

}

let mars = Planet[4]

print(mars)

**Inheritance**

클래스는 다른 언어와 비슷하다.

**Initialization**

**Customizing initialization**

struct Celsius {

var temperatureInCelsius: Double

init(fromFahrenheit fahrenheit: Double) {

temperatureInCelsius = (fahrenheit - 32.0) / 1.8

}

init(fromKelvin kelvin: Double) {

temperatureInCelsius = kelvin - 273.15

}

}

let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)

// boilingPointOfWater.temperatureInCelsius is 100.0

let freezingPointOfWater = Celsius(fromKelvin: 273.15)

// freezingPointOfWater.temperatureInCelsius is 0.0

생성자에 들어가는 argument의 이름을 달리해서 속성에 다른 값을 부여하도록 생성자를 커스텀 할 수 있다. 자바의 경우에 동일 타입인 파라미터로 선언된 생성자는 한 번에 한 인스턴스만 생성할 수 있는데 스위프트는 인자에 이름을 부여해 기능을 확장한다. 대신 자바와 달리 생성자에서 레이블을 빼고 값만 넣으면 컴파일 에러가 발생한다.

레이블 없이도 사용 가능한 생성자를 만들고 싶다면 함수 선언과 마찬가지로 생성자 선언 시 인자에 \_를 표기한다.

lazy 변수와 비슷하게 생성자 선언 시까지 값이 확정되지 않는 변수들은 옵셔널 타입으로 처리한다. 해당 속성들은 기본값으로 nil을 갖게 된다.

속성 변수가 모두 초기화 되어 있는 구조체나 클래스는 생성자를 선언하지 않아도 디폴트 생성자를 갖게 된다. 디폴트 생성자 호출 시 바로 해당 속성으로 된 인스턴스가 만들어진다.

구조체는 별도 선언 없이도 memberwise 생성자를 갖게 되는데 해당 생성자에 멤버 속성 레이블과 값을 넣어 초기화할 수 있다. 아무 것도 넣지 않으면 기본 속성대로 생성된다.

**Initializer Delegation for Value Types**

생성자는 다른 생성자의 인자로 주입될 수도 있다. 이를 Initializer Delegation이라고 한다. 생성자가 여럿 선언되는 것을 피하기 위해 이렇게 사용한다.

struct Size {

var width = 0.0, height = 0.0

}

struct Point {

var x = 0.0, y = 0.0

}

struct Rect {

var origin = Point()

var size = Size()

init() {}

init(origin: Point, size: Size) {

self.origin = origin

self.size = size

}

init(center: Point, size: Size) {

let originX = center.x - (size.width / 2)

let originY = center.y - (size.height / 2)

self.init(origin: Point(x: originX, y: originY), size: size)

}

}

세 가지 방법으로 생성할 수 있다. 세 번째의 경우 두 번째 생성자와 타입은 같지만 레이블을 다르게 해서 하위 생성자가 가지는 속성에 대한 추가 연산을 하게 해줬다. 여기서 self.init()은 오직 값 타입의 생성자 안에서만 선언할 수 있는 키워드로 다른 타입의 생성자를 자신의 생성자 안에서 참조하기 위한 키워드이다.

**Class Inheritance and Initialization**

구조체와 달리 클래스는 생성자가 종료되기 전까지 슈퍼 클래스를 비롯한 모든 인스턴스의 속성이 초기화되어야 한다. 이를 위해 클래스에는 두 가지 생성자가 제공된다. 지정 생성자(designated initializer)와 간편 생성자(convenience initializer)가 그것이다.

지정 생성자는 클래스를 위한 최우선의 생성자이다. 클래스에 지정된 모든 속성을 초기화하고 상속한 클래스가 있다면 슈퍼 클래스의 생성자도 호출해 모든 슈퍼클래스의 값을 초기화한다. 모든 클래스는 최소한 한 개 이상의 지정 생성자를 가지고 있어야 한다. 몇몇 경우에는 슈퍼 클래스의 지정 생성자를 상속받는 것으로 만족된다.

간편 생성자는 보조적 생성자이다. 몇 개의 파라미터 값이 디폴트로 제공되는 지정 생성자를 호출하기 위해 사용한다.

**Syntax for Designated and Convenience Initializers**

지정 생성자는 일반 생성자와 같은 형식으로 선언하지만 간편 생성자는 앞에 convenience 키워드가 붙는다.

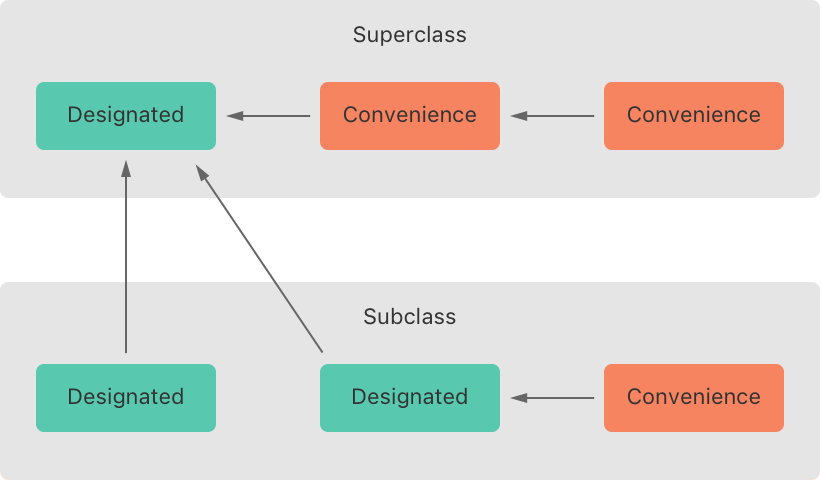
convenience init(<#parameters#>) {

<#statements#>

}

지정 생성자와 간편 생성자의 관계를 간단하게 하기 위해 스위프트는 delegation call을 위한 다음의 세 가지 규칙을 생성자 사이에 적용한다.

1. 지정 생성자는 반드시 직속 슈퍼 클래스의 지정 생성자를 호출해야 한다.
2. 간편 생성자는 반드시 같은 클래스에서 다른 생성자를 호출해야 한다.
3. 간편 생성자는 종국에는 반드시 지정 생성자를 호출해야 한다.



그림으로 표현하면 위와 같다.

간단하게 말하면 모든 지정 생성자는 결과적으로 반드시 슈퍼클래스의 지정 생성자를 호출하고 모든 간편 생성자는 결과적으로 반드시 지정 생성자를 호출한다.

**Two-Phase Initialization**

스위프트의 클래스 초기화는 2단계에 걸쳐 진행된다. 첫 번째 단계에서는 클래스의 저장 속성들이 초기화되고 1단계가 확정된다. 이후 해당 인스턴스를 사용할 준비가 되었을 때 2단계가 진행되어 커스텀한 속성값들이 들어가게 된다. 1, 2단계는 연속적으로 진행된다.

1단계가 완전히 끝나기까지 해당 인스턴스의 속성에 접근하는 것이 금지되기 때문에 초기화를 안전하게 진행할 수 있다. 또 다른 생성자에 의해 예기치 못하게 속성의 값이 수정되는 것을 방지할 수도 있다.

스위프트는 2단계 초기화를 에러 없이 진행하기 위해 4단계의 안전 검사를 수행한다.

1. 지정 생성자는 슈퍼 클래스의 생성자를 호출하기 전에 자신의 모든 속성에 대한 초기화를 완료해야 한다.

* 객체의 메모리는 해당 객체의 저장 속성이 모두 초기화된 후에야 완전히 초기화된 것으로 인식된다. 이 규칙을 만족시키기 위해 지정 생성자는 슈퍼 클래스 생성자를 호출하기 전에 반드시 자신의 모든 속성을 초기화해야 한다.

1. 지정 생성자는 **상속한 속성**에 새 값을 할당하기 전에 슈퍼클래스의 생성자를 호출해야 한다. 그렇게 하지 않으면 지정 생성자에 할당된 새 값이 슈퍼클래스의 생성자에 의해 슈퍼클래스의 것으로 대치된다.
2. 간편 생성자는 아무런 값도 할당하지 않고 우선 다른 생성자를 호출해야 한다. 그렇게 하지 않으면 간편 생성자가 할당한 속성 값이 지정 생성자에 의해 대치된다.
3. 생성자는 1차 생성 작업이 완료되기 전까지 어떤 인스턴스 메소드도 호출할 수 없고 어떤 인스턴스 속성 값도 읽을 수 없다. 또 self 키워드로 자신의 인스턴스 값을 호출하는 것도 안된다.

클래스 초기화 단계를 보다 구체적으로 정리하면 다음과 같다.

**1단계**

* 지정 생성자나 간편 생성자가 호출된다.
* 새 클래스 인스턴스를 저장할 메모리가 할당된다. 아직 메모리가 초기화되지는 않는다.
* 지정 생성자가 모든 속성에 값이 할당되었는지 확인하고 나면 메모리가 초기화된다.
* 슈퍼 클래스에도 동일한 일을 시키기 위해 지정 생성자가 슈퍼 클래스의 생성자를 호출한다.
* 이 과정을 상속 클래스의 최상단에 도달할 때까지 반복한다.
* 상속 사슬의 최상단에 도달하면, 그리고 상속의 최상단 클래스의 모든 속성이 값을 갖고 있다는 게 확인되면, 인스턴스의 메모리는 완전하게 초기화된다. 여기까지 마치면 1단계가 완료된다.

**2단계**

* 다시 상속 사슬의 최상단으로 돌아가서, 각 사슬의 지정 생성자는 인스턴스를 커스터마이즈할 수 있는 옵션을 가진다. 생성자는 이제 self에 접근할 수 있고 스스로의 속성을 수정할 수도, 인스턴스 메소드를 호출할 수도 있다.
* 마지막으로 사슬의 간편 생성자가 스스로를 커스터마이즈할 옵션을 갖고 스스로를 세팅한다.(self, properties, instance method 접근 허용)

1단계는 아래에서 위로(간편 -> 지정 -> 슈퍼 지정), 2단계는 위에서 아래로(슈퍼 지정 -> 지정 -> 간편) 이뤄진다.

**Initializer Inheritance and Overriding**

스위프트는 서브 클래스가 슈퍼 클래스의 생성자를 디폴트로 상속받지 않는다. 이런 스위프트의 접근 방식은 보다 구체화된 서브 클래스가 슈퍼클래스로부터 단순한 생성자를 상속받거나 완전히 초기화되지 않은 불완전한 서브 클래스 인스턴스를 생성하는 일을 방지한다. 특정 상황에서는 슈퍼클래스의 생성자가 상속되지만 해당 생성자를 사용하기에 안전하고 적절한 상황이어야만 한다.

커스텀 서브클래스에 슈퍼 클래스가 가진 것과 동일한 하나 이상의 생성자를 제공하고 싶다면 서브 클래스 안에 그런 생성자를 구현하면 된다. 슈퍼 클래스의 지정 생성자에 해당하는 생성자를 서브 클래스에 구현하는 건 슈퍼클래스의 지정 생성자를 오버라이딩하는 것이다. 따라서 override 키워드를 서브 클래스의 생성자 앞에 붙여야 한다.

class Vehicle {

var numberOfWheels = 0

var description: String {

return "\(numberOfWheels) wheel(s)"

}

}

let vehicle = Vehicle()

print("Vehicle: \(vehicle.description)")

// Vehicle: 0 wheel(s)

class Bicycle: Vehicle {

override init() {

super.init()

numberOfWheels = 2

}

}

let bicycle = Bicycle()

print("Bicycle: \(bicycle.description)")

// Bicycle: 2 wheel(s)

클래스 생성의 2단계에서 서브 클래스의 생성자가 커스텀 구현을 하지 않은 상태이고, 슈퍼 클래스의 지정 생성자가 인자가 없는 동기(synchronous) 생성자라면, 그리고 서브 클래스의 모든 속성에 값을 할당한 뒤라면, super.init() 을 명시하지 않아도 된다.

class Hoverboard: Vehicle {

var color: String

init(color: String) {

self.color = color

// super.init() implicitly called here

}

override var description: String {

return "\(super.description) in a beautiful \(color)"

}

}

let hoverboard = Hoverboard(color: "silver")

print("Hoverboard: \(hoverboard.description)")

// Hoverboard: 0 wheel(s) in a beautiful silver

위 코드를 보면 호버보드의 속성인 color에 값을 할당하는 서브 클래스 생성자를 호출하기만 했는데 상위 클래스인 Vehicle의 numberOfWheels가 0으로 세팅되어 리턴됐음을 알 수 있다.

**Automatic Initializer Inheritance**

위에서 살펴본 것과 같이 서브 클래스는 슈퍼클래스의 생성자를 디폴트로 갖지 않는다. 하지만 몇몇 조건이 충족될 시 슈퍼클래스의 생성자는 자동으로 상속된다. 위에서 살펴본 바로는 많은 경우에 슈퍼 클래스의 생성자를 오버라이드할 필요가 없었다. 서브 클래스에 적용된 모든 속성에 디폴트 값을 할당했다고 가정했을 때, 다음과 같은 2가지 규칙이 적용된다.

1. 서브클래스에서 어떠한 지정 생성자도 정의하지 않으면, 서브클래스는 자동으로 슈퍼 클래스의 모든 지정 생성자를 상속받는다.
2. 서브클래스가 슈퍼클래스의 모든 지정 생성자를 제공하거나 구현하면 슈퍼클래스의 모든 간편 생성자를 자동으로 상속받는다.

2번째 규칙을 만족하면 서브클래스의 간편 생성자로도 슈퍼클래스의 지정 생성자를 구현할 수 있다.

**Designated and Convenience Initializers in Action**

실제 지정 – 편의 – 자동 생성자가 위계 관계 안에서 어떻게 동작하는지 살펴보자.

서열의 최상단에 Food 클래스가 존재한다. Food 클래스는 base class로 음식의 이름을 저장하는 단순한 클래스이다. name이라는 하나의 변수를 가지며 문자열을 받으며 두 개의 생성자를 가진다.

class Food {

var name: String

init(name: String) {

self.name = name

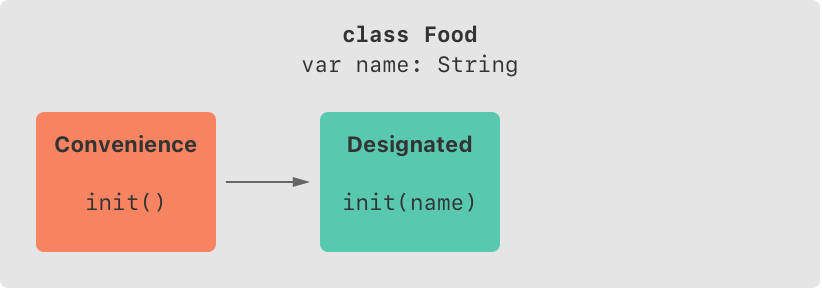
}

convenience init() {

self.init(name: "[Unnamed]")

}

}



푸드 클래스의 생성자 사슬을 표현한 그림이다.

클래스는 구조체와 달리 default memberwise properties를 가지지 않는다. 그래서 푸드 클래스는 하나의 name 인자를 받는 생성자를 제공한다. 이는 구체적인 name을 가진 새 Food 인스턴스를 만들 수 있게 한다.

let namedMeat = Food(name: "Bacon")

// namedMeat's name is "Bacon"

Food 클래스는 상위 클래스가 없기 때문에 super.init()을 가지고 있지 않다.

let mysteryMeat = Food()

// mysteryMeat's name is "[Unnamed]"

위 코드에서는 Food 클래스가 제공하는 간편 생성자로 인스턴스를 만든다. [Unnamed]를 place holder로 두어 init()을 호출하게 했다.

다음 클래스는 RecipeIngredient로 푸드 클래스를 상속해 레시피의 양에 해당하는 quantity 프로퍼티를 더한다.

class RecipeIngredient: Food {

var quantity: Int

init(name: String, quantity: Int) {

self.quantity = quantity

super.init(name: name)

}

override convenience init(name: String) {

self.init(name: name, quantity: 1)

}

}

텍스트, 스크린샷, 포스트잇 노트, 직사각형이(가) 표시된 사진

자동 생성된 설명

RecipeIngredient는 우선 자신의 속성을 초기화 시키고 슈퍼 클래스의 지정 생성자를 호출해 슈퍼클래스에 name값을 넘겨 슈퍼 클래스를 초기화한다. RecipeIngredient의 간편 생성자는 이름을 전달받아 양을 1로 초기화하는 인스턴스를 만들어 양이 1인 여러 개의 인스턴스를 만들 때 코드 중복을 피한다.

let oneMysteryItem = RecipeIngredient()

let oneBacon = RecipeIngredient(name: "Bacon")

let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)

class ShoppingListItem: RecipeIngredient {

var purchased = false

var description: String {

var output = "\(quantity) x \(name)"

output += purchased ? " ✔" : " ✘"

return output

}

}

위에서 쇼핑리스트 아이템은 자신의 속성을 모두 가지고 시작한다. 생성자를 설정하지 않기 때문에 purchased 는 초기 생성 시 항상 false이다. 이렇게 모든 값이 세팅되어 있기 때문에 ShoppingListItem은 자동으로 슈퍼클래스의 모든 지정, 편의 생성자를 상속받는다.

텍스트, 스크린샷, 폰트, 직사각형이(가) 표시된 사진

자동 생성된 설명

따라서 다음과 같이 사용할 수 있다.

var breakfastList = [

ShoppingListItem(),

ShoppingListItem(name: "Bacon"),

ShoppingListItem(name: "Eggs", quantity: 6),

]

breakfastList[0].name = "Orange juice"

breakfastList[0].purchased = true

for item in breakfastList {

print(item.description)

}

// 1 x Orange juice ✔

// 1 x Bacon ✘

// 6 x Eggs ✘

**Failable Initializers**

가끔은 클래스, 구조체, 열거형에 실패할 수 있는 생성자를 정의하는 것도 유용하다. 이 실패는 유효하지 않은 파라미터 값, 반드시 필요한 외부 리소스의 부재, 또는 초기화를 실패 시키는 다른 상황들 때문에 촉발될 수 있다.

이런 실패 가능한 상황들에 대처하기 위해 실패 가능한 생성자를 하나 이상 정의할 수 있다. 실패 가능 생성자는 init 키워드 뒤에 ? 키워드를 붙여 생성한다. 실패 가능한 생성자는 일반적인 생성자와 같은 파라미터 타입과 이름을 가질 수 없음에 주의하자.

실패 가능 생성자는 인스턴스를 옵셔널 값으로 초기화할 수 있다. 생성자가 실패했을 경우를 대비해 실패 가능 생성자 안에 return nil을 선언하자. 엄밀히 말하면 생성자는 값을 리턴하지 않는다. 생성자의 역할은 생성자가 끝나는 시점에 self가 완전하고 올바르게 초기화될 수 있도록 보장하는 데 있다. 생성 실패 시에는 이를 확실히 하고자 return nil을 사용하지만 생성 성공 시에는 return을 사용하지 않아야 한다.

let wholeNumber: Double = 12345.0

let pi = 3.14159

if let valueMaintained = Int(exactly: wholeNumber) {

print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")

}

// Prints "12345.0 conversion to Int maintains value of 12345"

let valueChanged = Int(exactly: pi)

// valueChanged is of type Int?, not Int

if valueChanged == nil {

print("\(pi) conversion to Int doesn't maintain value")

}

// Prints "3.14159 conversion to Int doesn't maintain value"

struct Animal {

let species: String

init?(species: String) {

if species.isEmpty { return nil }

self.species = species

}

}

let someCreature = Animal(species: "Giraffe")

// someCreature is of type Animal?, not Animal

if let giraffe = someCreature {

print("An animal was initialized with a species of \(giraffe.species)")

}

// Prints "An animal was initialized with a species of Giraffe"

let anonymousCreature = Animal(species: "")

// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {

print("The anonymous creature couldn't be initialized")

}

// Prints "The anonymous creature couldn't be initialized"

**Failable Initializers for Enumerations**

하나 이상의 파라미터를 통해 적절한 열거형 case를 선택할 때도 failable Initializer를 사용할 수 있다.

enum TemperatureUnit {

case kelvin, celsius, fahrenheit

init?(symbol: Character) {

switch symbol {

case "K":

self = .kelvin

case "C":

self = .celsius

case "F":

self = .fahrenheit

default:

return nil

}

}

}

let fahrenheitUnit = TemperatureUnit(symbol: "F")

if fahrenheitUnit != nil {

print("This is a defined temperature unit, so initialization succeeded.")

}

// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")

if unknownUnit == nil {

print("This isn't a defined temperature unit, so initialization failed.")

}

// Prints "This isn't a defined temperature unit, so initialization failed."

만약 열거형이 raw value를 가지면 자동으로 실패 가능 생성자를 받게 된다.

enum TemperatureUnit: Character {

case kelvin = "K", celsius = "C", fahrenheit = "F"

}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")

if fahrenheitUnit != nil {

print("This is a defined temperature unit, so initialization succeeded.")

}

// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")

if unknownUnit == nil {

print("This isn't a defined temperature unit, so initialization failed.")

}

// Prints "This isn't a defined temperature unit, so initialization failed."

생성자를 정의하지 않았지만 열거형에 없는 값으로 초기화하자 값이 nil로 되어있는 것을 알 수 있다.

**Propagation of Initialization Failure**

실패 가능 생성자는 클래스, 구조체, 열거형 내의 다른 실패 가능 생성자로 전이될 수 있다. 유사하게, 서브 클래스의 실패 가능 생성자는 슈퍼 클래스의 실패 가능 생성자로 전이될 수 있다. 두 경우 모두에서 실패 생성자의 전이는 다른 생성자의 진행을 멈추고 즉각적인 실패를 야기한다. 실패가 전이된 시점에서 추가적인 생성자 실행코드는 실행되지 않는다.

class Product {

let name: String

init?(name: String) {

if name.isEmpty { return nil }

self.name = name

}

}

class CartItem: Product {

let quantity: Int

init?(name: String, quantity: Int) {

if quantity < 1 { return nil }

self.quantity = quantity

super.init(name: name)

}

}

if let twoSocks = CartItem(name: "sock", quantity: 2) {

print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")

}

// Prints "Item: sock, quantity: 2"

if let zeroShirts = CartItem(name: "shirt", quantity: 0) {

print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")

} else {

print("Unable to initialize zero shirts")

}

// Prints "Unable to initialize zero shirts"

if let oneUnnamed = CartItem(name: "", quantity: 1) {

print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")

} else {

print("Unable to initialize one unnamed product")

}

// Prints "Unable to initialize one unnamed product"

**Overriding a Failable Initializer**

서브클래스에서 슈퍼클래스의 실패 가능 생성자를 오버라이딩할 수도 있다. 슈퍼클래스의 실패 가능 생성자를 서브클래스의 실패 불가능 생성자로 오버라이딩하는 것도 가능하다. 이렇게 하면 슈퍼클래스의 생성자가 실패를 용인해도 서브 클래스의 생성자가 이를 용인하지 않아 생성자 실패가 불가능하게 만들 수 있다.

class Document {

var name: String?

// this initializer creates a document with a nil name value

init() {}

// this initializer creates a document with a nonempty name value

init?(name: String) {

if name.isEmpty { return nil }

self.name = name

}

}

class AutomaticallyNamedDocument: Document {

override init() {

super.init()

self.name = "[Untitled]"

}

override init(name: String) {

super.init()

if name.isEmpty {

self.name = "[Untitled]"

} else {

self.name = name

}

}

}

class UntitledDocument: Document {

override init() {

super.init(name: "[Untitled]")!

}

}

슈퍼클래스를 초기화 시키고 슈퍼 클래스의 nil유무에 따라 조건문으로 속성 값을 할당해 실패가 불가능하게 만들었다.

**The init! Failable Initializer**

init? 대신 init!을 사용하면 적절한 타입의 옵셔널 인스턴스를 암묵적으로 언래핑하는 생성자를 정의할 수 있따.

**Required Initializers**

required 키워드를 이용해 생성자를 선언하면 해당 생성자를 정의한 클래스를 상속하는 모든 하위 클래스가 반드시 해당 생성자를 구현해야 한다.

class SomeClass {

required init() {

// initializer implementation goes here

}

}

class SomeSubclass: SomeClass {

required init() {

// subclass implementation of the required initializer goes here

}

}

상속 시에도 계속해서 다른 서브클래스가 이를 재구현하도록 required로 선언해야 한다.

**Setting a Defualt Property Value with a Closure or Function**

만약에 저장 속성 값이 커스텀화 된 값이나 특정한 세팅을 필요로 하면 해당 속성에 클로저를 할당하거나 전역 함수를 통해 클래스의 속성 값을 할당할 수 있다.

class SomeClass {

let someProperty: SomeType = {

// create a default value for someProperty inside this closure

// someValue must be of the same type as SomeType

return someValue

}()

}

클로저 끝에 소괄호를 둬 스위프트가 클로저를 바로 호출하게 한 것에 주목하자. 소괄호를 빼면 변수에 소괄호를 달아 따로 해당 클로저를 실행해야 하며 someValue가 자동으로 리턴되어 someProperty에 할당되지 않는다. 또 클로저로 할당되는 값을 제외하면 클래스 내의 다른 속성들은 초기화되지도 않은 상태라는 점에 주의하자. 이런 이유 때문에 클로저 안에서 해당 값들에 접근하거나 인스턴스 메소드를 호출하거나, self를 호출하는 건 불가능하다.

사각형, 실내 게임 및 스포츠, 보드게임, 게임이(가) 표시된 사진

자동 생성된 설명

struct Chessboard {

let boardColors: [Bool] = {

var temporaryBoard: [Bool] = []

var isBlack = false

for i in 1...8 {

for j in 1...8 {

temporaryBoard.append(isBlack)

isBlack = !isBlack

}

isBlack = !isBlack

}

return temporaryBoard

}()

func squareIsBlackAt(row: Int, column: Int) -> Bool {

return boardColors[(row \* 8) + column]

}

}

위 코드를 보면 클로저를 통한 할당이 어떻게 이루어지는지 알 수 있다. 함수를 통해 반복적인 작업(위에서는 흑, 백을 교차시켜 체스판 배열을 완성하는 것)을 대체시키고 이를 자동으로 변수에 할당하게 만들어 코드를 단순화했다.

let board = Chessboard()

print(board.squareIsBlackAt(row: 0, column: 1))

// Prints "true"

print(board.squareIsBlackAt(row: 7, column: 7))

// Prints "false"

**스위프트 공식문서(About Swift ~ Initialization) 12/9**