***스위프트 기초 문법***

상수와 변수

프로그래밍에서 상수와 변수는 값을 메모리에 저장할 수 있는 일정한 저장 공간이라고 이해하면 된다. 각각 변하지 않는 일정한 값, 변할 수 있는 값을 나타낸다

쉽게 말해 상수는 어떠한 값을 넣어도 변하지 않는다

상수는

let 상수명:데이터 타입 = 값

변수는

var 변수명:데이터 타입 = 값

이렇게 사용을 한다.

import Foundation

let a: Int = 100

//a = 200 이러면 에러->상수는 변경 불가능

var b: Int = 200

예를 들어 핸드폰의 전체 용량은 상수로 설정하고 사용 가능 공간을 변수로 설정한다.

스위프트의 기본 데이터 타입

Int : 64bit 정수형

UInt : 부호가 없는 64bit 정수형

Float: 32bit 부동 소수점 (Float 변수에 정수를 저장해도 자동으로 부동 소수점으로 변환)

Double : 64bit 부동 소수점

Bool : true, false 값

Character : 문자

ex)

var someCharacter : Character = "가"

String : 문자열

Any: 모든 타입을 자칭하는 키워드(어떤 타입이든 다 가질 수 있다)

스위프트는 타입 추론을 할 수 있다

따라서 변수나 상수명 뒤에 특정 타입을 명시하지 않아도 컴파일러가 알아서 타입을 추론하여 타입을 결정해준다.

var number = 10

그냥 이렇게 써줘도 'Int형 타입이구나' 라고 추론한다

컬렉션 타입

컬렉션 타입은 데이터들의 집합 또는 묶음이다

스위프트의 컬렉션 타입에는 크게 Array, Dictionary, Set가 있다.

Array

데이터 타입의 값들을 순서대로 지정하는 리스트

우리가 흔히 아는 배열이다

Dictionary

키와 값의 쌍으로 데이터를 저장하는 컬렉션 타입

Set

같은 데이터 타입의 값을 순서 없이 저장하는 리스트

값 중복 허용 안 함

그냥 집합이라고 생각하자

자바와 자료구조 시간에 배운 것이기에 세부적인 사항은 넘어간다

import UIKit

var numbers: Array<Int> = Array<Int>()

numbers.append(1)

numbers.append(3)

numbers.append(2)

numbers[0]

numbers[1]

numbers.insert(4, at:2) // index 2번에 4를 집어넣음

numbers

numbers.remove(at:0)

numbers

var names = [String]()

var names: [String] = []

//var dic: Dictionary<String, Int> = Dictionary<String, Int>()

var dic: [String: Int] = ["윤진용": 1]

dic["철수"] = 3

dic["김지"] = 5

dic

dic.removeValue(forKey: "김민지")

var set: Set = Set<Int>()

//set는 축약형 형태가 따로 없다!

set.insert(10)

set.insert(20)

set.insert(30)

set.insert(30)

set.insert(30)

set.remove(20)

함수

함수는 작업의 가장 작은 단위 그리고 코드의 집합이다

계속해서 중복되는 코드를 줄이기 위해 사용한다

func 함수명(파라미터 이름: 데이터 타입)->반환타입 {

return 반환값

}

func sum(a: Int, b: Int) -> Int {

return a+b

}

sum(a: 5, b: 3)

func hello() -> String { //매개변수 없는 함수

return "hello"

}

hello()

func printName() -> Void {

// 반환 값 없는 함수

}

또는

func printName() {

// 반환 값 없는 함수

}

func greeting(friend: String, me: String = "jinyong"){

print("Hello, \(friend)! I'm \(me)")

}

greeting(friend: "Albert")

//전달인자레이블 사용

func sendMessage(from myName: String, to name: String) -> String{

return "Hello \(name)! I'm \(myName)"

}

sendMessage(from: "Gunter", to: "Json")

//와일드카드(바로 값 넘겨주기)

func sendMessage(\_ name: String) -> String {

return "Hello \(name)"

}

sendMessage("윤진용")

//가변매개변수

func sendMessage(me: String, friends: String. . .) -> String {

return "Hello \(friends)! I'm \(me)"

}

sendMessage(me: "Jinyong", friends: "Jonson", "aiensert", "serius")

가변 매개변수로 선언되어 있으면 배열을 받을 수 있다고 생각하면 쉽다

스위프트의 함수는 일급 객체이다. 그래서 함수를 변수로 할당 할 수도 있고 매개변수에 집어넣을 수도 있다.

조건문

주어진 조건에 따라 다르게 동작하도록 하는 문

if, switch, guard 문이 있다

import UIKit

/\*

if 조건식 {

실행할 구문

}

\*/

let age = 12

if age < 19 {

print("미성년자 입니다.")

}

/\*

if 조건식 {

조건식이 만족하면 해당 구문 실행

} else {

만족하지 않는다면 해당 구문 실행

}

\*/

if age < 19 {

print("미성년자")

} else {

print("성년자")

}

/\*

if 조건식1 {

// 조건식1을 만족할 때 실행할 구문

} else if 조건식1 {

//조건식2를 만족할 때 실행할 구문

} else {

// 아무 조건식도 만족하지 않을 때 실행할 구문

}

\*/

let animal = "cat"

if animal == "dog" {

print("강아지 사료주기")

} else if animal == "cat" {

print("고양이 사료주기")

} else {

print("해당하는 동물 사료가 없음")

}

/\*

switch 비교대상 {

case 패턴1:

// 패턴1 일치할 때 실행되는 구문

case 패턴2, 패턴3:

// 패턴2, 3가 일치할 때 실행되는 구문

default:

// 어느 비교 패턴과도 일치하지 않을 때 실행되는 구문

}

\*/

let color = "green"

switch color {

case " blue":

print("파란색 입니다")

case "green":

print("초록색 입니다")

case "yellow":

print("노랑색 입니다")

default:

print("찾는 색상이 없습니다")

}

let temperature = 30

switch temperature {

case -20...9:

print("겨울 입니다")

case 10...14:

print("가을 입니다")

case 15...25

print("봄 입니다")

case 26...35

print("여름 입니다")

default:

print("이상 기후입니다")

반복문

반복적으로 코드가 실행되게 만드는 구문

프로그램을 작성하다 보면 같은 코드를 여러 번 호출해야 하는 경우가 생긴다

for-in

while

repeat-while

이 있다.

/\*

for 루프상수 in 순회대상 {

// 실행할 구문...: 코드블럭

}

\*/

for i in 1...4 {

print(i)

}

-> 1,2,3,4 차례대로 출력

let array = [1,2,3,4,5]

for i in array {

print(i)

}

이렇게 범위 데이터에 배열을 넣을 수 있다

/\*

while 조건식 {

// 실행할 구문

}

\*/

var number = 5

while number < 10 {

number+=1

}

/\*

repeat {

// 실행할 구문

} while 조건식

\*/

var x = 6

repeat {

x+=2

} while x < 5

print(x)

->적어도 한 번은 반드시 구문을 실행한다.

**옵셔널**

값이 있을 수도 있고 없을 수도 있다

(선택적인)

var name: String=""

이녀석은 값이 있는 걸까 없는 걸까

정확히 말하자면 빈 문자열이라는 값을 갖고 있는 것이다.

값이 없는 경우는 바로

var name: String=nil

이렇게 nil이 되는 것이다

var number:Int? = nil

이 경우도 마찬가지

그렇다고 모든 경우에 nil을 넣을 수 있는 것이 아니라

위의 경우와 같이 값이 있을수도 있고 없을 수도 있는 변수를 정의할 때

타입 뒤에 ? 물음표를 붙여줘야 nil을 지정해줄 수 있다.

이렇게 정의한 변수를 우리는 철수 아니 옵셔널이라고 부른다

옵셔널의 경우 기본값은 nil이다

일반적인 프로그래밍 언어에서는 값이 nil인 변수에 접근하면 런타임 에러가 발생해 프로그램이 종료된다 하지만 스위프트는 안전성 때문에 특이하게 값이 nil인 변수에 접근해도 프로그램이 종료되지 않는다.

import UIKit

var name: String?

-> 이거 실행하면 그냥 nil 나옴

그리고 옵셔널 타입에는 일반값도 넣을 수 있다

var optionalName: String? = "Jinyong"

// var requiredName: String = optionalName

이런 경우 에러가 난다. requiredName은 옵셔널이 아닌 String타입이므로 값이 꼭 있어야 하고, optionalName이 옵셔널 타입이므로 이 녀석이 실행되기 전까지 값이 있나 없나를 모르기 때문

따라서 swift는 안전을 위해 이런 경우를 금지한다

이때 발생하는 에러는 optionalName이라는 변수의 옵셔널 포장지를 벗기라는 에러가 뜨는데

optionalName변수를 출력해 보면 Optional("Jinyong") 처럼 옵셔널 포장지에 싸여있는 것을 볼 수 있다.

이 포장지를 벗겨주려면 옵셔널 바인딩을 이용해야만 한다.

**옵셔널 바인딩**

옵셔널 해제 방법에는 명시적 해제와 묵시적 해제가 있다

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저 명시적 해제하는 두가지 방법부터 알아보도록 하자

import UIKit

var number: Int? = 3

print(number)

print(number!)

실행해 보면

Optional(3)

3

이렇게 나온다.

옵셔널 변수에 느낌표를 붙여주면 강제로 옵셔널이 해제된 것을 볼 수 있는데

이 방법은 상당히 위험하다

옵셔널을 강제 해제 하게 되면, 에러가 발생해 프로그램이 종료될 수도 있다.

그렇다면 조금 더 안전한 방법은 없을까?

옵셔널을 조금 더 안전하게 추출하려면 '비 강제 해제-옵셔널 바인딩'을 하면 된다.

if let result = number {

print(result)

}else{

}

옵셔널 타입의 값을 변수 또는 상수로 할당하는 구문이다.

옵셔널이 벗겨지면 print가 실행되고 옵셔널 추출에 실패하면 else구문이 실행된다

->그래서 묶어준다고 해서 바인딩이라고 부른다

옵셔널 바인딩은 guard문으로도 가능한데

func test(){

let number: Int? = 5

guard let result = number else {return}

print(result)

}

test()

이렇게 하면 추출됨

if를 이용해 옵셔널 바인딩을 하면 옵셔널이 추출된 변수나 상수를 if 블록 내에서만 사용할 수 있지만, guard문으로 옵셔널을 추출하면 guard문 다음 함수 전체 구문에서 사용할 수 있다.

guard문은 guard문 조건을 만족할 때만 guard문을 통과하고(그래서 guard문: 장벽같은 녀석) 만약 통과하지 못한다면 else문으로 나간다(흐름을 종료시킨다) : 이후에 더 자세히 다룰 예정

다음은 묵시적 해제를 이용해 옵셔널을 벗겨볼 것이다.

먼저 컴파일러에 의한 자동해제가 있다

옵셔널 값을 비교연산자를 이용해 다른 값과 비교하면 컴파일러가 자동적으로 옵셔널을 해제시켜주는 것이다.

let value: Int? = 6

if value == 6 {

print("value가 6입니다")

} else {

print("value가 6이 아닙니다")

}

-> value가 6입니다

출력

이처럼 다른 값과 옵셔널 값을 비교해주면 컴파일러가 알아서 해제시켜준다.

마지막으로 묵시적 옵셔널 해제에 대해 알아보자

묵시적 해제는 옵셔널 타입이지만 값을 사용할 때는 자동으로 옵셔널이 해제된다

let string = "12"

var stringToInt: Int? = Int(string) //여기서 string 변수가 한글이나 다른 문자일 수도 있어서 이런 경우 nil을 반환하기 때문에 Int(string)의 반환 타입은 옵셔널이다.

//print(stringToInt + 1)

let string = "12"

var stringToInt: Int! = Int(string)

print(stringToInt + 1)

이렇게 stringToInt 변수의 타입 뒤에 느낌표를 붙여주게 되면, 묵시적 옵셔널 해제가 일어난다. 타입 뒤에 느낌표가 붙여져 있는 옵셔널 변수는 자동적으로 옵셔널 해제가 일어나 일반값처럼 자유롭게 사용할 수 있다.

구조체와 클래스

프로퍼티와 메소드를 이용해 구조화된 틀을 만든다

지겹도록 봐왔다

하나의 새로운 사용자 정의 데이터 타입을 만들어 준다고 생각하면 쉽다

또는 붕어빵 틀을 만든다고 생각하면 쉽다

스위프트에서도 구조체와 클래스 사용 방식도 거의 동일한데

약간의 차이점이 존재한다

클래스의 인스턴스는 참조타입

구조체의 인스턴스는 값 타입이라는 점이다

struct 구조체 이름 {

프로퍼티와 메소드

}

import UIKit

struct User {

var nickname: String

var age: Int

func information(){

print("\(nickname) \(age)")

}

}

var user = User(nickname: "Jinyong", age: 23) //생성자로 값을 초기화: 디폴트 생성자

user.nickname // 프로퍼티에 접근

user.nickname = "alphago" // 프로퍼티 값 변경

user.information() // 구조체 함수(메소드)에 접근

클래스는 클래스 키워드로 정의한다

class 클래스 이름 {

프로퍼티와 메소드

}

import UIKit

class Dog {

var name: String = ""

var age: Int = 0

func introduce(){

print("name \(name) age \(age)")

}

}

var dog = Dog() // 인스턴스 생성

dag.name = "coco"

dog.age = 3

dog.age

dog.introduce()

초기화(initialization)

클래스 구조체 또는 열거형의 인스턴스를 사용하기 위한 초기 과정

->생성자

프로퍼티 초기화 및 필요한 조정 수행 역할

텍스트이(가) 표시된 사진

자동 생성된 설명

위 사진에서 user 상수를 실행할 때, init이 호출되어 초기화 구문이 정의되어 있다면 초기화된다.

방식은

init(매개변수: 타입, ...) {

// 프로퍼티 초기화

// 인스턴스 생성시 필요한 설정을 해주는 코드 작성

}

ex)

class User {

var nickname: String

var age: Int

init(nickname: String, age: Int) {

self.nickname=nickname

self.age = age

}

init(age: Int) {

self.nickname = "albert"

self.age =age

}

}

}

var user = User(nickname: "jinyong", age: 23)

user.nickname

user.age

var user2 = User(age: 27)

user2.nickname

user2.age

생성자와 반대되는 소멸자(Deinitializer)도 있다

메모리 정리 및 삭제 기능을 구현할 수 있다

deinit {

print("deint user")

}

}

인스턴스가 있어야지 사용가능하다

인스턴스가 더이상 필요하지 않다면 인스턴스에 nil을 집어넣어라

그러면 더이상 인스턴스가 필요하지 않다고 판단해

deinit이 호출된다.

프로퍼티

프로퍼티는 클래스, 또는 열거형 등에 변수처럼 있는 값

프로퍼티는 세 가지 종류를 가진다

저장 프로퍼티

연산 프로퍼티

타입 프로퍼티

먼저 저장 프로퍼티는 프로퍼티를 사용하는 가장 간단한 방법이다

struct Dog {

var name: String

let gender: String

}

var dog = Dog(name:"warwar", gender: " Male")

print(dog)

이렇게 프로퍼티에 값을 저장시키는 것을 저장 프로퍼티라고 한다

dog.name = "윤진용"

let dog2 = Dog(name: "warwar", gender: "Male")

dog2.name = "윤진용" // 에러 발생

이 경우에는 dog2 즉 인스턴스 변수가 상수로 선언되어 있어서

프로퍼티 값의 변경이 불가능하다

이러한 결과가 나온 이유는 구조체가 값 타입이기 때문이다.

구조체 인스턴스를 상수로 선언하면 내부 프로퍼티도 상수화되어 변경이 불가능해진다.

그렇다면 클래스는 어떨까?

클래스는 참조타입이라 구조체와는 조금 다른 결과가 나온다.

클래스 인스턴스는 상수로 선언해도 변수로 선언한 프로퍼티 값을 변경할 수 있다.

class Cat {

var name: String

let gender: String

init(name: String, gender: String){

self.name = name

self.gender = gender

}

}

let cat = Cat(name: "json", gender: "female")

cat.name = "jonson"

print(cat.name)

클래스는 참조 타입이기 때문에 인스턴스를 상수로 선언했다 하더라도 내부 변수 타입 프로퍼티 값의 변경이 가능하다. 하지만 당연히 gender같은 상수 타입 프로퍼티는 변경할 수 없다.

다음으로 연산 프로퍼티에 대해 알아보자,

저장 프로퍼티는 구조체와 클래스에서만 사용 가능했지만

연산 프로퍼티는 열거형에서도 사용할 수 있다.

연산 프로퍼티는 값을 직접적으로 저장하지 않는 대신 getter 와 setter 를 이용해서 값을 직접적으로 접근하고 수정할 수 있다.

struct Stock {

var averagePrice: Int

var quantity: Int

var purchasePrice: Int {

get {

return averagePrice \* quantitiy

}

set(newPrice){

averagePrice = newPrice / quantity

}

}

}

var stock = Stock(averagePrice: 2300, quantity: 3)

print(stock)

stock.purchasePrice -> 6900 // get 실행

stock.purchasePrice = 3000 // set 실행

stock.averagePrice -> 1000

참고로 set만 지워서 읽기 전용 프로퍼티로 만들 수 있다

이런 경우 값을 변경할 수 없게 된다

그리고 set을 설정할 때 매개변수 디폴트 값은 newValue이다

그 다음으로 프로퍼티 옵저버를 살펴보겠다

프로퍼티 옵저버는 프로퍼티 값의 변화를 살펴보고 반응한다.

새로운 값이 기존 값과 같더라도 프로퍼티 옵저버는 실행된다.

프로퍼티가 set될 때마다 호출된다고 보면 된다.

프로퍼티 옵저버는 세 가지 경우에만 사용할 수 있다

저장 프로퍼티랑 오버라이딩이 된 저장 연산 프로퍼티에서만 사용할 수 있다.

class Account {

var credit: Int = 0 {

willSet {

print("잔액이 \(credit)원에서 \(newValue)원으로 변경될 예정입니다.")

}

didSet {

print("잔액이 \(oldValue)원에서 \(credit)원으로 변경되었습니다.")

}

}

}

var account = Account()

account.credit = 1000

->잔액이 0원에서 1000원으로 변경될 예정입니다

잔액이 0원에서 1000원으로 변경되었습니다.

willSet이나 didSet이나 모두 매개변수를 지정할 수 있는데 지정하지 않는다면

newValue와 oldValue가 디폴트 매개변수가 된다.

마지막으로 타입 프로퍼티는 인스턴스 생성 없이 객체 내 프로퍼티에 접근 할 수 있도록 하는 거다

프로퍼티 타입 자체와 연결하는 것이다.

저장 프로퍼티와 연산 프로퍼티에서만 사용 가능하다.

static 키워드를 사용해서 정의한다.

struct SomeStructure {

static var storedTypeProperty = "Some value."

static var computedTypeProperty: Int {

return 1

}

}

static이라 그냥 인스턴스 생성 안하고 바로 타입으로 호출할 수 있다. (전역\_)

SomeStructure.computedTypeProperty

SomeStructure.storedTypeProperty

SomeStructure.storedTypeProperty = "hello"

값도 변경할 수 있다.

클래스와 구조체의 차이

클래스와 구조체는 문법과 사용 방법이 거의 동일하다.

공통점으로는

프로퍼티를 선언할 수 있고

메소드도 선언할 수 있고

. 연산자를 사용해 내부 값에 접근할 수 있다

생성자(init)을 이용해 초기 상태를 설정할 수 있으며

extension을 사용해 기능을 확장할 수 있다.

Protocol을 채택해 기능을 설정할 수도 있다.

많은 공통점도 있는 반면에

하지만 명확한 차이점도 있다.

클래스는 참조타입이지만 구조체는 값 타입이다.

클래스는 참조타입이기 때문에 메모리 스택 영역에는 포인터, 즉 인스턴스의 메모리 주소만 할당이 되고 실제 데이터는 힙 영역에 할당이 된다. 힙 영역에 데이터가 저장되기 때문에 ARC로 메모리가 관리된다.

또 상속이 가능하고, 타입 캐스팅을 사용해 런타임 환경에서 클래스 인스턴스의 타입을 확인할 수도 있다. deinit을 사용해 클래스 인스턴스의 메모리 할당을 해제할 수도 있으며 같은 클래스 인스턴스를 여러 개의 변수에 발당한 뒤 값을 변경시키면 모든 변수에 메모리가 복사되어 있기 때문에(참조 값이라) 모든 변수에 전부 영향을 준다.

하지만 구조체는 구조체 변수를 새로운 변수에 할당할 때마다 완전히 새로운 구조체가 할당이 되고(값 타입이라 스택 영역에 직접 값이 저장) 따라서 구조체를 여러 개의 변수에 할당한 뒤 값을 변경시키더라도 다른 변수에 영향을 주지 않는다. (값 자체를 복사해 버리기 때문)

import Foundation

class SomeClass {

var count: Int = 0

}

struct someStruct {

var count: Int = 0

}

var class1 = SomeClass()

var class2 = class1

var class3 = class1

class3.count = 2

class1.count // class1의 count가 2로 변경된 것을 확인할 수 있다.

var struct1 = SomeStruct()

var struct2 = struct1

var struct3 = struct1

struct3.count = 3

struct2.count = 4

struct1.count //0

struct2.count //4

struct3.count //3

상속

부모가 자식에게 메소드 프로퍼티 등의 속성을 상속해 주는 것

부모클래스(슈퍼클래스) --> 자식클래스

오버라이딩을 통해 물려받은 각종 메소드를 자신 고유의 내용으로 재정의 할 수 있다.

import Foundation

class Vehicle {

var currentSpeed = 0.0

var description: String {

return "traveling at \(currentSpeed) miles per hour"

}

func makeNoise(){

print("스피커가 on 되었습니다")

}

} // 베이스 클래스

//서브 클래스

class Bicycle: vehicle {

var hasBasket = false

}

var bicycle = Bicycle()

bicycle.currentSpeed

bicycle.currentSpeed = 15.0

bicycle.currentSpeed

//overriding

class Train: Vehicle {

override func makeNoise() {

super.makeNoise() // Vehicle 클래스의 makeNoise 메소드 호출

print("choo choo")

}

}

let train = Train()

train.makeNoise()

//프로퍼티 재정의->프로퍼티의 getter setter 옵저버를 재정의하는 것

class Car: Vehicle {

var gear = 1

override var description: String {

return super.description + "in gear \(gear)"

}

}

let car = Car()

car.currentSpeed = 30.0

car.gear = 2

print(car.description)

추가로 연산 프로퍼티를 오버라이딩 한 프로퍼티는 getter 와 setter를 가질 수 있다.

자식 클래스에서 재정의 하려는 프로퍼티는 슈퍼 클래스의 이름과 타입이 일치해야 한다.

슈퍼 클래스에서 readonly로 선언된 프로퍼티를 자식 클래스에서 read write 프로퍼티로 선언할 수 없지만

반대로 슈퍼 클래스에서 read write 프로퍼티로 선언된 프로퍼티를 자식 클래스에서 readonly 프로퍼티로 재정의 할 수 있다.

class AutomaticCar: Car {

override var currentSpeed: Double {

didSet {

gear = Int(currentSpeed / 10) + 1

}

}

}

let automatic = AutomaticCar()

automatic.currentSpeed = 35.0

print("AutomaticCar: \(automatic.description)")

상수 저장 프로퍼티나 readonly 연산 프로퍼티는 옵저버를 추가할 수 없는데

이는 둘 다 값을 설정할 수 없기 때문에 willSet이나 didSet을 사용할 수 없기 때문이다.

마지막으로 프로퍼티나 메소드 앞에 final 키워드를 붙이게 되면 그 대상은 상속 후 재정의 할 수 없게 된다.

클래스를 정의할 때도 클래스 키워드 앞에 final를 앞에 붙여주면 상속할 수 없게 된다.

타입캐스팅

인스턴스의 타입을 확인하거나 어떠한 클래스의 인스턴스를 해당 클래스 계층 구조상의 슈퍼 클래스나 서브 클래스로 변환하는 것

스위프트에서는 타입 캐스팅을 is 와 as 라는 연산자를 사용해서 수행한다

import Foundation

class mediaItem {

var name: String

init(name: String) {

self.name = name

}

}

class Movie: MediaItem {

var director: String

init(name: String, director: String) {

self.director = director

super.init(name: name)

}

}

class Song: MediaItem {

var artist: String

init(name: String, artist:String) {

self.artist = artist

super.init(name: name)

}

let library = [ // library는 mediaItem 배열로 타입추론

Movie(name: "기생충", director: "봉준호"),

Song(name: "Butter", artist: "BTS"),

Movie(name: "올드보이", director: "박찬욱"),

Song(name: "Wonderwall", artist: "Oasis"),

Song(name: "Rain", artist: "이적")

]

var movieCount = 0

var songCount = 0

for item in library {

if item is Movie {

movieCount += 1

} else if item is Song {

songCount += 1

}

}

print("Media library contains \(movieCount) movies and \(songCount) songs")

for item in library {

if let movie = item as? Movie {

print("Movie: \(movie.name). dir. \(movie.director)")

} else if let song = item as? Song {

print("Song: \(song.name), by \(song.artist)")

}

}

as?는 다운캐스팅한 값을 옵셔널 타입으로 반환

그래서 위에서 if let으로 옵셔널 바인딩을 통해 꺼내온 것을 알 수 있다.

as!는 다운캐스팅한 값을 강제로 unwrapping 해서 반환 : 항상 성공한다는 확신 있으면 사용 (runtime error 방지)

assert와 guard

assert

특정 조건을 체크하고 조건이 성립하지 않으면 메세지를 출력 하게 할 수 있는 함수

assert함수는 디버깅 모드에서만 동작하고 주로 디버깅 중 조건의 검증을 위하여 사용한다.

guard문

뭔가를 검사하여 그 다음에 오는 코드를 실행할지 말지 결정 하는 것

guard문에 주어진 조건문이 거짓일 때 구문이 실행된다.

import Foundation

var value = 0

assert(value == 0)

value = 2

assert(value == 0, "값이 0이 아닙니다") //runtime error 발생

assertion함수는 특정 조건을 검증하고 조건이 만족하지 않으면 메세지와 함께 error를 발생시키는 함수이기 때문에 runtime error가 발생하는 것

그래서 주로 디버깅 중 조건의 검증을 위해 사용하는 거다

guard문은 주로 잘못된 값이 함수에 들어오는 것을 방지하기 위해 사용한다.

guard문은 어디서든지 많이 쓰이기 때문에 확실하게 알고 넘어가야 한다

guard문은 다음과 같은 형태를 띈다

/\*

guard 조건 else {

// 조건이 false면 else 구문이 실행되고

return or throw or break 를 통해 이후 코드를 실행하지 않도록 한다

}

\*/

func guardTest(value: Int) {

guard value == 0 else { return }

print("안녕하세요")

}

guardTest(value: 2) // 아무것도 실행 안 됨 return

guardTest(value: 0)

앞에서 잠깐 언급한 적이 있었듯이 guard문으로 옵셔널 바인딩도 할 수 있다.

func guardTest(value: Int?) {

guard let value = value else { return }

print(value)

}

이런 경우에 nil일 때만 else로 나가고

옵셔널 포장지가 정상적으로 벗겨지면 print 된다.

guardTest(value: 2) // 2

guardTest(value: nil) // 아무 것도 출력 안 됨

**프로토콜**

프로토콜은 특정 역할을 하기 위한 청사진이다.

import Foundation

/\*

protocol 이름 {

}

\*/

protocol SomeProtocol {

}

protocol SomeProtocol2 {

}

struct someStructure: SomeProtocol, SomeProtocol2 {

}

이때 주의해야 할 점은 클래스는 상속이 먼저 나오고 그 다음 프로토콜을 작성한다

/\*

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {

}

\*/

protocol FirstProtocol {

var name: Int { get set }

var age: Int { get }

}

protocol AnotherProtocol {

static var someTypeProperty: Int { get set }

func printFullName()

}

protocol FullyNames {

var fullName: String { get set }

func printFullName(){

print(fullName)

}

}

struct Person: FullyNames {

var fullName:String

}

protocol SomeProtocol3 {

func someTypeMethod()

}

protocol SomeProtocol4 {

init(someParameter: Int)

}

protocol SomeProtocol5 {

init()

}

class SomeClass: SomeProtocol5 {

required Init() { // 구조체는 따로 required 식별자 필요 없다 오직 클래스에서 init을 채택할 때

}

}

클래스가 상속 받을 수 없는 final이면 required 필요 없다

extension

기존의 클래스, 구조체, 열거형, 프로토콜에 새로운 기능을 추가하는 기능

extension으로 추가할 수 있는 기능

- 연산 타입 프로퍼티, 연산 인스턴스 프로퍼티

-타입 메소드/ 인스턴스 메소드

-이니셜라이저

-서브 스크립트

-중첩 타입

- 특정 프로토콜을 준수할 수 있도록 기능 첨가

import Foundation

/\*

extension SomeType {

// 추가 기능

}

\*/

extension Int {

var isEven: Bool {

return self % 2 == 0

}

var isOdd: Bool {

return self % 2 == 1

}

}

var number = 3

number.isOdd // ture

number.isEven // false

참고로 익스텐션은 연산 프로퍼티를 추가할 수 있지만

저장 프로퍼티는 추가할 수 없다

또 타입에 정의되어 있는 기존 프로퍼티의 프로퍼티 옵저버를 추가할 수 없다

extension String {

func converToInt() -> Int? {

return Int(self)

}

}

var string = "0"

string.convertToInt()

즉 타입 자체에다 기능을 추가하는 것!

추가로 이니셜라이져 서브스크립트 등을 이용해 기능을 확장할 수도 있다.

열거형

연관성이 있는 값을 모아 놓은 것

import Foundation

enum CompassPoint {

case north

case south

case east

case west

}

또는

enum CompassPoint {

case north,south,east,west

}

처럼 한 줄로 작성할 수도 있다.

열거형은 하나의 새로운 타입처럼 사용할 수 있다.

그래서 클래스 처럼 스위프트의 이름 규칙에 따라 열거형의 첫문자를 대문자로 시작해야 한다

var direction = CompassPoint.east

direction = .west

switch direction {

case .north:

print("north")

case .south:

print("south")

case .east:

print("east")

case .west:

print("west")

}

//원시값으로 초기화하기

enum CompassPoint: String {

case north = "북"

case south = "남"

case east = "동"

case west = "서"

}

// rawValue 프로퍼티로 초기값 사용

switch direction {

case .north:

print(direction.rawValue)

case .south:

print(direction.rawValue)

case .east:

print(direction.rawValue)

case .west:

print(direction.rawValue)

}

//생성자로 생성하기

let direction2 = CompassPoint(rawValue: "남")

// 연관값 설정하기

enum phoneError {

case unknown

case batteryLow(String)

}

let error = PhoneError.batteryLow("배터리가 곧 방전됩니다.")

//batteryLow("배터리가 곧 방전됩니다") 출력

연관값을 추출하려면 if case 또는 switch 문 사용

switch error {

case .batteryLow(let message):

print(message)

case .unknown:

print("알 수 없는 에러입니다")

}

**옵셔널 체이닝**

옵셔널 체이닝은 옵셔널에 속해있는 nil일지도 모르는 프로퍼티, 메소드, subscription 등을 가져오거나 호출할 때 사용하는 일련의 과정을 뜻한다.

import Foundation

struct Developer {

let name: String

}

struct Company {

let name: String

var developer: Developer?

}

var developer = Developer(name: "kan")

var company = Company(name: "Jinyong" ,developer: developer)

print(company.developer) // 옵셔널에 감싸진 kan이 나온다

print(company.developer.name) // 옵셔널을 해제해야 name 프로퍼티에 접근할 수 있다

print(company.developer?.name) // 옵셔널 체이닝: 접근한 옵셔널 프로퍼티 값은 항상 옵셔널에 감싸져 있고->Optional("kan")->옵셔널 바인딩을 통해 옵셔널을 언래핑하면 된다

print(company.developer!.name) // 옵셔널 체이닝 느낌표로 접근한 옵셔널 프로퍼티 값은 항상 강제 옵셔널 해제되어있다->kan

try~catch

swift에서의 에러 처리 방식

스위프트에서 에러는 에러 프로토콜을 따르는 타입의 값으로 표기된다

에러 프로토콜을 요구 사항이 없는 빈 프로토콜이지만 오류를 표현하기 위해 이 빈 프로토콜을 채택해야 한다.

스위프트의 열거형은 이런 면에서 해당 오류를 나누고 이에 대한 응답을 나누는 데 굉장히 적합한 구문이다

enum PhoneError: Error {

case unknown

case batteryLow(batteryLevel: Int)

}

throw PhoneError.batteryLow(batteryLevel: 20)

스위프트의 오류 처리 방식

1. 함수에서 발생한 오류를 해당 함수를 호출한 곳으로 떠넘기기

2. do catch 사용해 처리

3. 옵셔널 값으로 오류 처리

4. 오류가 발생하지 않는다고 확신!

1번째 방식

함수 매개변수 뒤에 throws 집어 넣기

func checkPhoneBatteryStatus(batteryLevel: Int) throws -> String {

guard batteryLevel != -1 else { throw PhoneError.unknown }

guard batteryLevel > 20 else {throw PhoneError.batteryLow(batteryLevel: 20)}

return "배터리 상태가 정상입니다"

}

2번째 방식

do catch 사용하기

오류가 던져졌을 때, 그 오류를 처리하는 방식

/

do {

try 오류 발생 가능 코드

} catch 오류 패턴 {

처리 코드

}

/

do {

try checkPhoneBatteryStatus(batteryLevel: -1)

} catch PhoneError.unknown {

print("알 수 없는 에러입니다.")

} catch PhoneError.batteryLow(let batteryLevel) {

print("배터리 전원 부족 남은 배터리: \(batteryLevel)%")

} catch {

print("그 외 오류 발생 : \(error)")

}

3번째 옵셔널 방식

let status = try? checkPhoneBatteryStatus(batteryLevel: -1)

print(status)

//오류가 발생하면 반환값이 nil

4번째 절대 발생하지 않는다 자신만만하기

let status2 = try! checkPhoneBatteryStatus(batterylevel: 30)

print(status2)

// 오류가 발생하면 runtime error 가 발생해 프로그램이 강제 종료되므로 주의

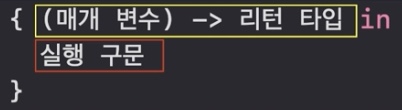
**클로저**

클로저는 코드에서 전달 및 사용할 수 있는 독립 기능 블록이며, 일급 객체의 역할을 할 수 있다.

여기서 일급 객체란 전달 인자로 보낼 수 있고, 변수, 상수 등으로 저장하거나 전달할 수 있으며 함수의 반환 값이 될 수 있는 객체를 뜻한다.

보통 클로저라 하면 이름없는 함수 즉 익명함수를 지칭한다.

unnamed closure 와 named closure(함수) 둘 다 사실 클로저이지만, 보통은 unnamed closure를 우리는 closure라고 한다



클로저는 다음과 같은 형태를 띈다

노란색 박스 안 부분이 클로저 헤드

그리고 붉은 박스 안쪽이 클로저 바디영역이다.

이 둘은 in이라는 키워드에 의해 나누어 진다.

import Foundation

let hello = { () -> () in //파라미터와 리턴 타입 둘 다 없는 경우

print("hello")

}

hello()

클로저의 실행 구문에서 정의한 hello 가 출력된다!

let hello2 = { (name: String) -> String in

return "hello, \(name)" // 주의 클로저에서는 전달 인자 레이블을 사용하지 않는다. 따라서 저녀석은 오직 파라미터 네임일 뿐이다

}

// hello2(name: "Jinyong") 에러발생

hello2("Jinyong")

//클로저를 함수의 파라미터로 전달 가능(일급 객체라 가능)

func doSomething(closure: () -> ()) {

closure()

}

doSomething(closure: { () -> () in

print("hello")

})

//리턴 타입이 클로저인 경우

func doSomething2() -> () -> () {

return { () -> () in

print("hello4")

}

}

doSomething2()()

// 클로저 가독성 좋게 쓰기

func doSomething(closure: () -> ()) {

closure()

}

doSomething(closure: { () -> () in

print("hello")

})

doSomething() {

print("hello2")

//그냥 이렇게 쓴다, 어차피 반환값도 매개값도 없으니 전부 생략, in 도 생략

}

doSomething 함수처럼 단 하나의 클로저만 매개변수로 전달하는 경우 소괄호도 생략 가능하다

doSomething {

print("hello2")

}

매개변수에 클로저가 여러 개 있는 경우 다중 후행 클로저 구문을 사용한다

func doSomething2(success: () -> (), fail: () -> ()) {

}

doSomething2 {

code // 첫번째는 매개변수 레이블 생략

} fail: {

code

}

문법을 최소화 하여 클로저를 단순하게 표현하기

텍스트이(가) 표시된 사진

자동 생성된 설명

#### 고차함수

다른 함수를 전달 인자로 받거나 함수 실행의 결과를 함수로 반환하는 함수

스위프트에선 함수가 일급 객체이다

스위프트에서 제공하는 고차함수에는

map

filter

reduce

이 세가지가 있다

모두 collection 타입에 구현되어있다.

import Foundation

// map

let numbers = [0,1,2,3]

let mapArray = numbers.map { (number) -> Int in

return number \* 2

}

print("map \(mapArray)")

-> [0,2,4,6]

//filter

let intArray = [10,5,20,13,4]

let filterArray = intArray.filter { $0 > 5 }

print("filter \(filterArray)")

//reduce

let someArray = [1,2,3,4,5]

let reduceResult = someArray.reduce(0) {

(result: Int, element: Int) -> Int in

print("\(result) + \(element)")

return result + element

}

print("reduce \(reduceResult)")