swift 문법

이름짓기 규칙(대소문자구분)

Lower Camel case : function, method, variable, constant ex). someVariable

Upper Camel case : type(class, struct,, enum, extension …) ex). Person, Point

%d대신에 \(변수)로 “ “안에 넣어줘서 출력

상수의 선언

let 이름: 타입 = 값

변수의 선언

var 이름: 타입 = 값

(만약 값의 타입이 명확하다면 타입 생략가능)

--------------------------------------------------------------------------------------------------------------------------------------------------

변수 타입의 종류 : Bool, Int, UInt, Float, Double, Character, String => Character, String에는 큰 따옴표를 사용한다.

Any : Swift의 모든 타입을 지칭하는 키워드

AnyObject : 모든 클래스 타입을 지칭하는 프로토콜

nil : 없음을 의미하는 키워드-NULL과 비슷 (Any type에는 할당 불가능)

Array : 멤버가 순서(index)를 가진 리스트 형태의 컬렉션 타입(여러가지 표현법이 존재한다)   
ex). var integers: Array<int> = Array<int> () or [] or [int]() 등등

내부적으로 여러가지 함수기능들을 제공한다.

Dictionary : key값과 value값으로 이루어진 컬렉션 타입(여러가지 표현법이 존재한다)  
ex). var anyDictionary: Dictionary<String, Any> = [String, Any]()

Set : 중복되지 않는 멤버가 순서없이 존재하는 컬렉션(여러가지 표현법은 없다)

ex). var integerSet: Set<Int> = Set<Int>()

--------------------------------------------------------------------------------------------------------------------------------------------------

1. 함수선언의 기본형태

func 함수이름(매개변수1이름:매개변수1타입, 매개변수2이름:매개변수2타입, ……)->반환타입{

return 반환값

}

매개변수에 기본적으로 전달될 값을 미리 지정해 둘 수 있다. (기본값을 갖는 매개변수는 매개변수 목록 중 뒤쪽에 위치하는 것이 좋다)  
2. 전달인자 래이블

함수를 호출할 때 함수 사용자의 입장에서 매개변수의 역할을 좀 더 명확하게 표현하고자 할 때 사용한다.

ex). func greeting(to friend: String, from me: String){

print(“Hello \(friend)! I’m \(me)”);

}

그리고 호출을 greeting(to: “haha”, from: “yagom”)

3. 가변 매개변수

전달 받을 값의 개수를 알기 어려울 때 사용한다.

ex). func sayHelloToFriends(me: String, friends: String…)->String{  
 return “Hello \(friends)! I’m \(me)”

}  
print(sayHelloToFriends(me: “yagom”, friends: “hana”, “eric”, “wing”))

4. 데이터 타입으로서의 함수

스위프트는 함수형 프로그래밍 패러다임을 포함하는 다중 패러다임 언어이므로 스위프트의 함수는 일급객체이다. 그러므로 함수를 변수, 상수 등에 할당이 가능하고 매개변수를 통해 전달할 수 도 있다.

ex). func runAnother(function: (String, String) -> Void){

function(“jenny”, “mike”)

}

--------------------------------------------------------------------------------------------------------------------------------------------------

1. if조건문

if 조건 { } else if 조건 { } else{ }와 같이 C언어와 유사함

조건은 Bool타입의 값만 위치해야 한다.

2. switch구문

스위프트의 switch구문은 다른 언어에 비해 굉장히 강력한 힘을 발휘한다.

기본적으로 사용하던 정수타입의 값만 비교하는 것이 아니라 대부분의 스위프트 기본 타입을 지원하며, 다양한 패턴과 응용이 가능하다.

default구문은 반드시 작성해야 합니다(매우 한정적인 enum case등이 아니면)

명시적 break를 하지 않아도 자동으로 case마다 break가 됩니다.

fallthrough키워드를 사용하여 break를 무시할 수 있습니다.

ex). switch 비교값 {

case 패턴:

default:

}

---------------------------------------------------------------------------------------------------------------------------------------------------

반복문

1. for-in구문(python과 유사)

ex). for integer in arr{  
 print(integer)

}

2. while구문

while 조건 {

}

3. repeat-while구문

기존 언어의 do-while구문과 형태/동작이 유사합니다.

repeat [  
]while 조건

---------------------------------------------------------------------------------------------------------------------------------------------------

옵셔널

값이 있을 수도, 없을 수도 있음을 표현

nil이 할당 될 수 있는지 없는지 표현

ex). func someFunction(someOptionalParam:Int?){  
} -> nil이 할당가능함

func someFunction(someOptionalParam:Int){  
}->nil할당 불가능

옵셔널을 사용하는 이윺?

1. 명시적 표현 (nil의 가능성을 코드만으로 표현가능, 문서/주석 작성 시간 절약)

2. 안전한 사용 (전달받은 값이 옵셔널이 아니라면 nil을 체크하지 않고 사용가능, 예외 상황을 최소화 하는 안전하고 효율적인 코딩)

옵셔널은 enum+general로 만들어진 형태

옵셔널 문법과 선언

원래는 옵셔널을 적용하기 위해서는 let value: Optinoal<Int> = nil과 같이 선언을 해줘야 하지만

let value: Int? = nil과 같이 선언도 가능(?띄어쓰기 불가능)

!를 사용하는 경우 : 암시적 추출 옵셔널 -> 처음 선언할 때 부터 옵셔널 강제 추출을 가정하는것

(기존 변수처럼 사용가능, nil할당가능 그러나 nil을 가지고 산수 불가능)

?를 사용하는 경우 : 옵셔널

(기존변수처럼 사용 블가능 – optional type과 일반 Int type은 엄연히 다른 타입이므로, nil할당가능)

옵셔널을 꺼내는 2가지 방법

1. Optional Binding – 옵셔널 바인딩

2. Force Unwrapping – 강제 추출

옵셔널 바인딩 : nill체크 + 안전한 값 추출 -> if let방식 사용

옵셔널 강제추출 : 옵셔널 물음표가 벗겨지고 안에있던 값이 나옴 -> 그냥 뒤에 !를 붙임으로써 사용

스위프트 대부분 타입은 구조체로 이루어져 있다.

**구조체 값(value) 타입이다.**

타입이름은 대문자 카멜케이스를 사용하여 정의합니다.

구조체 문법

struct 이름{

}

구조체 내부 변수, 함수의 선언부 앞에 static을 붙이면 타입 자체가 사용하는 공용 프로퍼티로써 사용하게 됨, 구조체 내부변수 초기화 필수!

static변수 혹은 함수는 무조건 구조체 인스턴스가 아닌 그냥 구조체 이름을 사용하여 호출(static이 있으면 type method, 없으면 인스턴스 method)

let타입으로 구조체상수를 정의했을 경우 아무리 내부변수가 var형태이더라도 구조체 생성후 수정이 불가능하다.

만약 class같이 이미 존재하는 타입을 변수명으로 사용하고 싶을 경우 앞뒤에 강세표(맥북의 경우 ~+option)를 붙여줌으로써 변수같이 사용할 수 있다.

구조체 내부프로퍼티는 self. 키워드를 통해서도 접근할 수 있다. (예를들어 내부함수를 작성할때)

구조체 내부변수의 값을 내부 함수를 통해 값을 변경해주고 싶으면 mutating키워드 사용

**클래스는 참조(reference) 타입이다.**

타입이름은 대문자 카멜케이스를 사용하여 정의한다.

Swift의 클래스는 다중 상속이 되지 않는다.

클래스 문법

class 이름{

}

static으로 선언한 경우 재정의 불가 타입 메서드이다(override불가능)

class func ~은 재정의 가능한 타입 메서드이다.

class의 경우 struct와 다르게 var, let둘중 뭐로 선언하여도 내부의 var값을 변경할 수 있다. (let은 불가)

타입 메서드중 앞에 static이 붙게되면 상속시 재정의 불가능

열거형 문법

enum 이름{  
 case 이름1

…

}

타입이 명확하다면 .케이스 처럼 표현해도 된다. ex), day = .tue

Swift의 열거형 또한 C언어와 마찬가지로 값을 가질 수 있다. (rawValue) -> case마다 다른값을 가져야함(자동으로 1이 증가된 값이 할당된다)

enum 이름:타입 으로 선언해줌으로써 rawValue를 가질 수 있음

enum이름.case이름.rawValue를 통해서 값에 접근가능함

case가 없으면 생성이 되지 않을 수 있기 때문에(nil) 변수에 값을 할당할 때는 옵셔널로 선언해준다.

ex). let apple: Fruit? = Fruit(rawValue:0)

열거형 내부에 method를 추가할 수 있다.

클래스 vs 구조체

Class : 단일상속, 참조 타입 (Apple프레임워크의 대부분의 큰 뼈대는 모두 클래스로 구성)

Struct : 상속불가, 값 타입 (Swift의 대부분의 큰 뼈대는 모두 구조체로 구성)

구조체는 언제 사용하나?

-다른 객체 또는 함수 등으로 전달될 때 참조가 아닌 복사를 원할 때

-자신을 상속할 필요가 없거나, 자신이 다른 타입을 상속받을 필요가 없을 때

스위프트에서의 데이터타입들은 모두 구조체로 구현이 되어있다.

중요한 것 클래스에서는 내부 인스턴스를 바꾸는 함수 앞에 mutating이 필요없고, 구조체의 경우 필요하다.

클로저

실행가능한 코드 블럭

함수와 다르게 이름정의는 필요없지만, 매개변수 전달과 반환 값이 존재 할 수 있다.

함수는 이름이 있는 클로저이다.

일급객체로 전달인자, 뱐수, 상수 등에 저장 및 전달이 가능하다.

클로저 문법

{ (매개변수 목록) -> 반환타입 in

실행코드

}

예제]

let divide:(Int, Int)->Int

divide = { (a: Int, b:Int) in

return a/b

}

func calculate(a: Int, b: Int, method: (Int, Int)->Int)->Int{  
 return method(a,b)

}

다른 예제) calculated = calculate(a:50, b:10, method: { (left: Int, right: Int)->Int in return left\*right})

클로저 고급-다양한 클로저 표현

1. 후행 클로저 – 굳이 파라미터를 추가해주지 않아도 뒤에 클로저를 추가해줌으로써 파라미터로써 하나가 더 전달될 것을 암시.

ex). result = calculate(a: 10, b: 10) { (left: Int, right: Int) -> Int in

return left+right

}

2. 반환타입 생략 – 함수의 method매개변수는 Int타입을 반환할 것이라는 사실을 컴파일러도 알기 때문에 굳이 클로저에서 반환타입을 명시해 주지 않아도 된다. 그러나 in키워드는 생략불가.

3. 단축 인자이름 – 클로저의 매개변수 이름이 굳이 불필요하다면 단축 인자이름을 활용할 수 있다. 단축 인자 이름은 클로저의 매개변수의 순서대로 $0, $1, $2…처럼 표현한다.

4. 암시적 반환 표현 – 클로저가 반환하는 값이 있다면 클로저의 마지막 줄의 결과값은 암시적으로 반환값으로 취급된다.

축약전

result = calculate(a: 10, b: 10, method: { (left: Int, right: Int)->Int in

return left+right

})

축약후

result = calculate(a: 10, b: 10) { $1+$2}

프로퍼티의 종류

1. 인스턴스 저장 프로퍼티

2. 타입 저장 프로퍼티

3. 인스턴스 연산 프로퍼티

4. 타입 연산 프로퍼티

5. 지연 저장 프로퍼티

정의와 사용

- 프로퍼티는 구조체, 클래스, 열거형 내부에 구현할 수 있다.

- 다만 열거형 내부에는 연산 프로퍼티만 구현할 수 있다.

- 연산 프로퍼티는 var로만 선언할 수 있다.

- 연산 프로퍼티를 읽기전용으로는 구현할 수 있지만, 쓰기 전용으로는 구현할 수 없다.

- 읽기전용으로 구현하려면 get블럭만 작성해주면 된다. 읽기전용은 get블럭을 생략할 수 있다.

- 읽기, 쓰기 모두 가능하게 하려면 get블럭과 set블럭을 모두 구현해주면 된다.

- set블럭에서 암시적 매개변수 newValue를 사용할 수 있다.

저장프로퍼티와 연산프로퍼티의 기능은 함수, 메서드, 클로저, 타입등의 와부에 위치한 지역/전역 변수에도 모두 사용가능하다.

프로퍼티 감시자

- 프로퍼티 감시자를 사용하면 프로퍼티의 값이 변경될 때 원하는 동작을 수행할 수 있다.

- 값이 변경되기 직전에 willSet블럭이, 값이 변경된 직후에 didSet블럭이 호출된다.

- 둘 중 필요한 하나만 구현해 주어도 무관하다.

- 변경되려는 값이 기존 값과 똑같더라도 프로퍼치 감시자는 항상 동작한다.

- willSet블럭에서는 암시적 매개변수 newValue를, didSet블럭에서는 oldValue를 사용할 수 있다.

- 프로퍼티 감시자는 연산 프로퍼티에는 사용할 수 없다.

- 프로퍼티 감시자는 함수, 메소드, 클로저, 타입 등의 지역/전역 변수에 모두 사용 가능하다.

상속

- 상속은 클래스, 프로토콜 등에서 가능하다.

- 열거형, 구조체는 상속이 불가능하다.

- 스위프트의 클래스는 단일상속이므로, 다중상속을 지원하지 않는다.

상속 문법

class 이름: 상속받을 클래스 이름{

}

class내부 값, 함수에 final키워드를 사용하여 재정의를 방지할 수 있다. (override)

final class == static

재정의 불가 타입 메서드 - static

재정의 가능 타입 메서드 - class

자식class에서 func앞에 override를 붙여줌으로써 override가능  
super키워드를 통해 부모 접근가능

class func도 override키워드를 통해 재정의 가능

인스턴스의 생성과 소멸

- 프로퍼티 초기값

- 이니셜라이저 init

- 디이니셜라이저 deinit

1. 프로퍼티 초기값

스위프트의 모든 인스턴스는 초기화와 동시에 모든 프로퍼티에 유효한 값이 할당되어야 한다,

프로퍼티에 미리 기본값을 할당해두면 인스턴스가 생성됨과 동시에 초기값을 지니게 된다.

2. 이니셜라이저(initializer)

프로퍼티 초기값을 지정하기 어려운 경우에는 이니셜라이저 init을 통해 인스턴스가 가져야 할 초기값을 전달할 수 있다.

- 프로퍼티의 초기값이 꼭 필요 없을 때

내부 인스턴스를 옵셔널로 선언해준다.

init을 파라미터개수별로 여러개로 만들어줄 수 있다. -> 중복되는 경우가 생기므로 다른 방법으로 표현할 수 있다. (만들어져 있던 이니셜라이저를 호출하고 이니셜라이저 앞에 convenience를 붙여준다)  
암시적 추출 옵셔널은 인스턴스 사용에 꼭 필요하지만 초기값을 할당하지 않고자 할 때 사용(인스턴스 뒤에 !를 붙여서 사용)

3. 실패가능한 이니셜라이저

실패하면 nil을 반환해준다

init?(~)로 선언

4. 디이니셜라이저(클래스타입에서만 사용가능)

deinit은 클래스의 인스턴스가 메모리에서 해제되는 시점에 호출된다. 인스턴스가 해제되는 시점에 해야할 일을 구현할 수 있다.

옵셔널 체이닝과 nil병합

있는지 없는지 확인하는 작업을 계속계속 단계별로 접근해야함 -> 코드가 복잡해짐

옵셔널 체이닝

ex). optionalChaining(owner: Person?){  
 if let guardJob = owner?home?`guard`?job{

~

}

else{

}

}

nil병합 연산자

- 앞에 제시된 값이 nil이라면 뒤의 값을 넣어줘라.

ex). guardjob = yagom?.home?.guard?.job ?? “슈퍼맨”

타입 캐스팅

스위프트의 타입캐스팅은 인스턴스의 타입을 확인하는 용도

변수앞에 Double()과 같이 표현해주는 것은 타입캐스팅이 아닌 인스턴스를 하나더 생성해주는 개념

is를 사용하여 타입을 확인가능

업캐스팅의 경우 아무문제가 없음

다운캐스팅의 경우 조건부 다운캐스팅, 강제 다운캐스팅으로 나뉨

-조건부 다운캐스팅(as?)

캐스팅이 성공하게 되면은 옵셔널로 반환(값 or nil)

-강제 다운캐스팅(as!)

캐스팅이 실패하게 되면 런타임오류

assert와 guard

-assert

애플리케이션이 동작 도중에 생성하는 다양한 결과값을 동적으로 확인하고 안전하게 처리할 수 있도록 확인하고 빠르게 처리할 수 있다.

assert함수는 디버깅 모드에서만 동작한다.

앞의 조건이 참이면 그냥 넘어가고, 거짓이면 오류메세지 출력

ex). assert(someInt == 0, “someInt!=0”)

-guard(early exit)

guard를 사용하여 잘못된 값의 전달 시 특정 실행구문을 빠르게 종료한다.

디버깅모드 뿐만 아니라 어떤 조건에서도 동작한다.

guard의 else블럭 내부에는 특정 코드블럭을 **종료하는 지시어(return, break등)가 꼭 있어야 한다.**

타입캐스팅, 옵셔널과 자주 사용된다.

if문과는 다르게 위의. guard에서 생성한 값을 밑에서도 사용가능함.

딕셔너리와 종종 같이 사용됨.

ex). func someFunction(info: [String, Array]){

guard let name = info[“name”] as? String else{

return

}

guard let age. = info[“age”] as? Int, age>=0 else{

return

}

print(“\(name): \(age)”)

}

someFunction(info: [“name”, “jenny”, “age”: “10”] -> 프린트안됨

someFunction(info: [“name”, “mike”] -> 프린트안됨

someFunction(info: [“name”, “jenny”, “age”: 10] -> jenny:10

프로토콜

프로토콜은 특정 역할을 수행하기 위한 메서드, 프로퍼티, 기타 요구사항등의 청사진을 정의한다.

쉽게말하면 기능을 특정화, 강요하는 느낌

문법

protocol 프로토콜이름{

}

사전에 정의한 느낌

struct Person: Talkable{ -> Person구조체는 Talkable프로토콜을 채택했습니다.

}

프로토콜은 클래스와 다르게 다중상속이 가능합니다.

protocol 프로토콜 이름: 부모 프로토콜 이름 목록 {  
}

프로토콜이 요구하는 것중에 하나라도 구헌하지 않으면 오류발생

클래스에서 상속과 프로토콜 채택을 동시에. 하려면 상속받으려는 클래스를 먼저 명시하고 그 뒤에 채택할 프로토콜 목록을 작성한다.

익스텐션

- 익스텐션은 스위프트의 강력한 기능중에 하나이다.

- 익스텐션은 구조체, 클래스, 열거혀으 프로토콜 타입에 새로운 기능을 추가할 수 있는 기능이다.

- 기능을 추가하려는 타입의 구현된 소스 코드를 알지 못하거나 볼 수 없다 해도, 타입만 알고 있다면 그 타입의 기능을 확장할 수도 있다.

<<추가할 수 있는 기능>>

- 연산타입프로퍼티/연산인스턴스프로퍼티

- 타입메서드/인스턴스메서드

- 이니셜라이저

- 서브스크립트

- 중첩타입

- 특정 프로토콜을 준수할 수 있도록 기능 추가

문법

extension 확장할 타입 이름: 프로토콜1, 프로토콜2, … { ->프로토콜 없어도 됨

}  
ex). extension Int{

var isEven: Bool {

return self%2==0

}

var isOdd: Bool {

return self%2==1

}

}

오류 처리

Error프로토콜과 (주로)열거형을 통해서 오류를 표현합니다.

enum 오류종류이름: Error{

case invalidInput

case insufficientFunds(monetNeeded: Int)

case outOfStock

}

오류 발생의 여지가 있는 메서드는 throws를 사용하여 오류를 내포하는 함수임을 표시한다.

func receiveMoney(\_ money: Int) throws { -> 함수 파라미터앞에 \_가 있을 경우 라벨없이 그냥 호출가능해짐

guard money>0 else{

throw VendingMachineError.invalidInput

}

self.deposited += money

}

오류를 던질 수도 있지만 오류가 던져지는 것에 대비하여 던져진 오류를 처리하기 위한 코드도 작성해야 한다. 오류발생의 여지가 있는 throws함수(메서드)는 try를 사용하여 호출해야한다. try와 do-catch, try?와 try!등에 대해 알아보자.

<<do-catch>>

오류 발생의 여지가 잇는 throws함수는 do-catch구문을 활용하여 오류발생에 대비한다.

가장 정석적인 방법으로 모든 오류 케이스에 대응한다.

문법

do {  
 try machine.recieveMoney(0)

} catch VendingMachineError.invalidInput{  
 print(“입력이 잘못되었습니다”)

} catch VendingMachineError.insufficientFunds(let moneyNeeded) {  
 print(“\(moneyNeeded)원이 부족합니다”)

} catch VendingMachineError.outOfStock {

print(“수량이 부족합니다”)

}

이렇게 표현할 수도 있고, 하나의 catch 블럭에서 switch구문을 사용하여 오류를 분류해볼 수도. 있다. (catch문 내부로는 자동으로 error변수가 들어오게 되므로 switch error{}를 통해 오류처리)

딱히 케이스별로. 오류처리를 할 필요가 없다면 catch구문을 간략화 해도됨 (print(error)와 같이)

<try?, try!>

1. try?

별도의 오류처리 결과를 통보받지 않고 오류가 발생했으면 결과값을 nil로 돌려받는다.

정상동작 후에는 옵셔널 타입으로 정상 반환값을 돌려 받는다.

2. try!

오류가 발생하지 않을 것이라는 강력한 확신을 가질 때 try!를 사용하면 정상동작 후에 바로 결과값을 돌려받는다.

오류가 발생하면 런타임 오류가 발생하여 애플리케이션이 동작 중지된다.

고차 함수

고차 함수는 ‘다른 함수를 전달 인자로 받거나 함수 실행의 결과를 함수로 반환하는 함수’를 뜻한다.

스위프트의 함수(클로저)는 일급객체이기 때문에 함수의 전달인자로 전달할 수 있으며 함수의 결과값으로 반환할 수도 있다.

고차함수의 종류 [map, filter, reduce]

map, filter, reduce함수는 스위프트 표준 라이브러리의 컨테이너 타입(Array, Set, Dictionary등)에 구현되어 있다.

1. map

map함수는 컨테이너 내부의 기존 데이터를 변형(transform)하여 새로운 컨테이너를 생성한다.

doubledNumbers = numbers.map({ (number: Int) -> Int in

return number\*2

})

이를 매개변수, 반환타입, 반환 키워드를 생략하고 후행클로저를 적용하게 되면

doubledNumbers = numbers.map{ $0 \*2 }와 같이 표현가능

2. filter

filter함수는 컨테이너 내부의 값을 걸러서 새로운 컨테이너로 추출해준다.

let evenNumbers: [Int] = numbers.filter{ (number: Int) -> Bool in

return number%2 == 0

}

이를 매개변수, 반환타입, 반환 키워드를 생략하고, 후행클로저를 적용하게 되면

let oddNumbers: [Int] = numbers.filter{

$0%2!=0

}

3. reduce

reduce함수는 컨테이너 내부의 콘텐츠를 하나로 통합해준다.

let sum: Int = someNumbers.reduce(0, { (first: Int, second: Int) -> Int in

return first+second

})

-> 초기값이. 0이고 someNumbers내부의. 모든 값을 더해줌.

first – second일시 someNumbers내부의 모든값을 빼줌.

축약형태

let sumFromThree = someNumbers.reduce(3){$0+$1}

추가적으로 알아가야 할 문법과 개념들

1. Generics(제네릭)

2. Subscript(서브스크립트)

3. Accress Control(접슨수준)

4. Automatic Reference Counting(ARC)

5. Nested Types(중첩타입)

6. CustomOperators(사용자정의 연산자)

7. Error Handling(오류 처리)

8. Opaque Types(불명확 타입)

9. Protocol Oriented Programming(프로토콜 지향 프로그래밍)