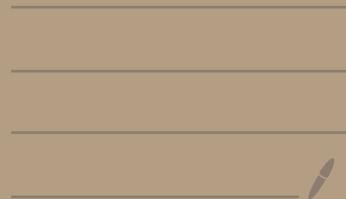


Swift



2강

이름짓기 규칙

- Lower Camel Case : function, method, variable, constant

ex) SomeVariableName

- Upper Camel Case : type(class, struct, enum, extension ~~)

ex) Person, Point, Week

- Swift는 대소문자를 구분한다.

콘솔로그

- print : 단순 문자열 출력

- dump : 인스턴스의 자세한 설명(description 프로퍼티) 가시 출력
↳ 복잡한 Class instance, Struct instance 같은 예제는
문제 때문에 출력

```
class Person {
```

```
    var name: String = "yagom"
```

```
    var age: Int = 10
```

```
let yagom: Person = Person()
```

```
print(yagom) → ~~~.Person
```

```
dump(yagom) → ~~~.Person
```

```
- name: "yagom"
```

```
- age: 10
```

문자열 보간법

- String Interpolation

- 프로그래밍 실행 중 문자열 내에 변수 또는 상수의 실질적인 값을 표현하기 위해 사용

- \()

3강

상수와 변수

- 상수 선언 키워드 : let

- 변수 선언 키워드 : var

- 상수의 선언

```
let 이름: 타입 = 값
```

값의 타입이 명확하다면

타입은 생략 가능

ex) let 이름 = 값
var 이름 > 값

- 변수의 선언

```
var 이름: 타입 = 값
```

```
let Constant : String = "값은 변할 수가 нет"
var Variable : String = "값은 변할 수가 있"
```

· 상수 선언 후 나중에 값 변경 불가

```
let sum : Int
let inputA : Int = 100
let inputB : Int = 200
```

```
sum = inputA + inputB
```

· 물론 변수도 가능함. 대신 같은 값을 활용하고 사용해야
개념을 얻을

ex) var test : String

```
print(test) → error
```

47b

· Swift의 기본 데이터 타입

· Bool, Int, UInt, Float, Double, Character, String

· Bool

```
var someBool : Bool = true
someBool = false
// someBool = 0
// someBool = 1
```

· UInt

```
var someUInt : UInt = 100
[SomeUInt = 100 → error(B5)]
```

[SomeUInt = SomeInt → error
(cannot assign value of type
'Int' to type 'UInt')]

* Swift는 Data type이 엄격한 언어.

Data type끼리 자료형이 맞지 않아.

이유? → 앞서 언급한 Data type 고유의 error를

引发해 버려

· Int

```
var someInt = -100
[SomeInt = 100.1 → error(Float)]
```

· Float 32bit

```
var someFloat : Float = 3.14
someFloat = 3 → error X
```

· Double 64bit

```
var someDouble : Double = 3.14
someDouble = 3
[SomeDouble = SomeFloat → error]
```

· Character 한 글자

· 내보내 사용 (한글이나 표현할 수 있는 모든 것 가능)

var someCharacter : Character = "ㅋㅋㅋ"
[SomeCharacter = "ㅋㅋㅋ" → error]

· String

```
var someString : String = "안녕하세요 ^_^"
someString = someString + "우리집 빵이자고"
print(someString)
[SomeString = SomeCharacter → error]
```

5강

- Any - Swift의 모든 타입을 지정하는 키워드
- Any Object - 모든 클래스 타입을 지정하는 protocol
- nil - 알 수 없는 값이라는 키워드

- Any

```
var someAny : Any = 100  
someAny = "모든 타입은 그걸 가질 수 있다"  
someAny = [1, 2]  
[let someDouble : Double = someAny → error]
```

- nil

```
[someAny = nil → error]  
[someAnyObject = nil  
↳ 어떤 Datatype 이든  
들여올 수 있는 모든 X]
```

· Any Object

```
class SomeClass {}  
var someAnyObject : AnyObject = SomeClass()  
[someAnyObject = 123, nil → error]
```

6강

· 컬렉션 타입

- Array - 순서가 있는 리스트 컬렉션
- Dictionary - 키와 값의 쌍으로
- Set - 순서가 없고, 중복이 허용되는 컬렉션

↳ Int typealias
↳ nil Array 예제

```
.Array  
var integers : Array<Int> = Array<Int>()  
integers.append(1)  
[integers.append(0, 1) → Float error]  
integers.append(100)
```

integers.contains(1) → 1이 있는 Array를 가리킨다. true
integers.contains(2) → false
integers.remove(at: 0) → 0번째 index 삭제 = 1
integers.removeLast() → 마지막 index 삭제 = 100
integers.removeAll() → 빈 배열 = []
integers.count → 0

↳ Array<Double>()

· 빈 Double Array 예제
var doubles : Array<Double> = [Double]()
[doubles.append(1.1)

· 빈 String Array 예제
var strings : [String] = [String]()
[strings.append("")
↳ Array<String>]

· 빈 Character Array 예제

· []는 새로운 빈 Array
var characters : [Character] = []

· let을 사용하여 Array를 선언하면 불변 Array
let immutableArray = [1, 2, 3]

• Dictionary

key가 String type이거나 Value가 Any인 데 Dictionary 타입
var anyDictionary: Dictionary<String, Any> = [String: Any]()
anyDictionary["somekey"] = "Value"
anyDictionary["anotherkey"] = 100
anyDictionary → ["somekey": "Value", "anotherkey": 100]
anyDictionary["somekey"] = "Dictionary" → 값 변경
anyDictionary.removeValue(forKey: "anotherkey") → 값 삭제
 $\uparrow \downarrow$ 유사
anyDictionary["somekey"] = nil

let emptyDictionary: [String: String] = [:]

let initializedDictionary: [String: String]
= ["name": "yagom", "gender": "male"]

[let someValue: String = initializedDictionary["name"] → error 이유 생략해보기]

↳ 이 Dictionary의 key는 해당하는 Value가
있을 수도 있고 없을 수도 있기 때문 → Optional

• Set - 중복 X
- 인덱스 X

(O) ↗ 총액운법 적용

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

integerSet.insert(1)
integerSet.insert(10)
integerSet.insert(99) } 중복 허용 X
integerSet.insert(99)
integerSet → {100, 99, 1}

integerSet.contains(1) → true
integerSet.contains(2) → false

integerSet.remove(100) → 100이 제거됨: {99, 1}

integerSet.removeFirst() → 99이 제거됨

integerSet.count → 1

let setA: Set<Int> = [1, 2, 3, 4, 5] → 자동정렬 X

let setB: Set<Int> = [3, 4, 5, 6, 7]

let union: Set<Int> = setA.union(setB) → 합집합: {2, 4, 5, 6, 7, 3, 1}

let sortedUnion: [Int] = union.sorted() → 정렬: [1, 2, 3, 4, 5, 6, 7]

배열로 변환

let intersection: Set<Int> = setA.intersection(setB) → 교집합: {3, 4}

let subtracting: Set<Int> = setA.subtracting(setB) → 차집합: {2, 1}

7강

함수 선언과 기본 형태

```
func 함수이름 (매개 변수 1 이름 : 매개 변수 1 타입, 매개 변수 2 등...) → 반환타입 {
```

함수 구현부

return 반환값

}

↓

```
func sum(a:Int, b:Int) → Int {
```

return a+b

}

함수의 호출

```
sum(a:3, b:5)
```

반환 값이 있을 때?

① 생략은 가능

```
func printMyName (name:String) → Void {  
    print(name)
```

→

```
printMyName (name:"Ham")
```

}

매개 변수가 있을 때?

```
func 함수이름 () → 반환타입 {  
    함수 구현부  
    return 반환값
```

```
func maximumIntegerValue() → Int {  
    return Int.MAX
```

}

→

```
maximumIntegerValue()
```

매개 변수와 반환값이 없는 함수

```
func hello() → Void { print("hello") }
```

```
func bye() { print ("bye") }  
    ↴ 대체로 생략
```

→

```
hello()
```

```
bye()
```

매개변수 기본값

- 매개변수에 값이 할당되었을 때 기본이 같지 않은 것
 - 기본값을 갖는 매개변수는 매개변수 목록 중에 뒤쪽에 위치하는 것에 종속
- ```
func greeting(friend: String, me: String = "Hoon") {
 print("Hello \(friend)! I'm \(me)!")
}
```
- }
- 매개변수 기본값을 가진 매개변수는 생략 가능
- ```
greeting(friend: "Hoon")
```

전달인자 레이블

- 전달인자 레이블은 함수를 호출할 때 매개변수의 역할을 좀 더 명확하게 하거나 함수 사용자의 입장에서 표현하고자 할 때 사용된다.
- ```
func greeting(to friend: String, from me: String) {
 print("Hello \(friend)! I'm \(me)!")
 ↳ 함수 내부에서만 매개변수 사용
}
```
- 함수를 호출한 때에는 전달인자 레이블을 사용해야 함
- ```
greeting(to: "haha", from: "Hoon")
```

가능 매개변수

- 전달 받을 값의 개수를 알기 어려울 때 사용할 수 있음
 - 가변 매개변수는 함수당 최대한 가질 수 있음
- ```
func sayHelloToFriends(me: String, friends: String...) -> String {
 return "Hello \(friends)! I'm \(me)!"
}

print(sayHelloToFriends(me: "Hoon", friends: "haha", "eric", "wing"))
↓
전달인자나 결과나
내용 볼때면 오류 발생
```
- 가변인자에 아무 값도 넣기고 살피고 싶어 암스런 생활 가능
- ```
print(sayHelloToFriends(me: "Hoon"))
```

데이터 타입으로 허의 함수

- Swift는 칙스팅 프로그래밍의 대체작업을 포함하는 다음 패러다임입니다.
- 데이터의 타입은 일관적이므로 변수, 속성 등에 개성이 가능하고 매개변수를 통해 검증할 수도 있다.

• 함수의 탭 풀현 - 반환 타입을 생략할 수 있음
(매개변수 탭, 매개변수2 탭 ..) → 반환타입

```
var someFunction(String, String) -> Void
```

(?) 일반 만든 greeting 함수
이 모든 생활값 X
String, String

```
var someFunction: (String, String) -> Void = greeting(to: from:)
```

```
someFunction("eric", "Hoon") ⇒ greeting과 결론 동일
```

• ctci 푸드
somefunction = greeting(friend: me:)
somefunction("eric", "Hoon")

• 탭이 다른 함수는 확장 불가
somefunction = sayHelloToFriends(me: friends:)

↳ String 이상 추가한
가능하지 않다

• 함수 탭을 매개변수 탭으로 지정

```
func runAnother(function: (String, String) -> Void) {
    function("jenny", "mike")
}
```

↳ function이라는 매개변수는 알리바로 실행 가능하지 않음

- & String 탭은 매개변수로 가지고 반호같이 있는 함수
function 탭은 매개변수 탭

```
runAnother(function: greeting(friend: me)) → Hello Jenny! I'm mike
```

```
runAnother(function: someFunction) → Hello Jenny! I'm mike
```

9강

조건문

if - else

- if {} 뒤에는 생각 불가
- Swift의 조건문은 항상 블록문에 들어가야 한다.

switch

• 블록 변수

Switch someInteger {

Case 0:
print("zero")

Case 1...100:
print("1~100")

이상 < 0
Case 100:
print("100")

Case 101...Int.MAX:
print("over 100")

default:
print("unknown")

}

정수 외의 대체형의 기본 타입을 사용할 수 있다.

Switch "yagon" {
↳ hashable protocol
Case "jake":
print("jake")

Case "mina":
print("mina")

Case "yagon":
print("yagon")

Default:
print("unknown")

↳ Swift의 switch는
명시적 모든 case가 명시하지 않는 한
default 구문은 꼭 작성 해야 함

또한 Case 다음에 break를 따로 안 적어야 된다.

굳이 break를 쓰기 싫으면 fallthrough를 작성해 주면 된다.

10강

반복문

```
var integers = [1,2,3]
let people = {"yagon":10, "eric":15, "mike":12}
```

· for - in

```
for integer in integers {
    print(integer)
}
```

· Dictionary의 item은 key와 value로 구성된 투플 타입이다.

```
for (name, age) in people {
    print("\(name): \(age)")
}
```

↳ 투플이란?

리스트와 동일하게 여러 객체를 모아서 담는다.

숫자, 문자, 정수, 배열, 투플 안의 투플 전부 가능하다.

하지만 투플 내의 값은 수정, 추가, 삭제가 안된다.

(개인들은 가능)

· While

☞ 소프트는 선택적

```
while (integers.count) > 1 {
    integers.removeLast()
}
```

· repeat - while (do-while과 비슷)

```
repeat {
    integers.removeLast()
} while integers.count > 0
```

옵셔널 (값이 있을 수도, 없을 수도 있음)

• nil의 가능성은 명시적으로 표현

- nil 가능성을 문서화하지 않아도 코드으로 충분히 표현 가능
- 문서/주석 작성 시艰을 절약
- 전달 받은 값이 옵셔널이 아니면 nil 처리를 하지 않아도 안심하고 사용
- 효율적인 코딩
- 예외 사용을 최소화하는 안전한 코딩

```
func someFunction(someOptionalParam: Int?) {
    ~~
}

func someFunction(someParam: Int) {
    ~~
}

someFunction(someOptionalParam: nil) → 사용 가능
someFunction(someParam: nil) → error
```

Optional

enum + general

```
enum Optional<Wrapped> : ExpressibleByNilLiteral {
    case none
    case some(Wrapped)
}

let optionalValue: Optional<Int> = nil
let optionalValue: Int? = nil
```

• Optional에서의 !, ?

!

Implicit Unwrapped Optional

암시적 추출 옵셔널

```
var optionalValue: Int! = 100
                                ← 값이 있는 때

switch optionalValue {
    case .none:
        print("This Optional variable is nil!")
    case .some(let value):
        print("Value is \(value)")   ← 값이 있는 때
}
```

• 기본 변수처럼 사용 가능

optionalValue = optionalValue + 1

• nil 할당 가능

optionalValue = nil

optionalValue에 nil을 넣을 때
자연적으로 접근은 하지 않지만 다른 방법

• 잘못된 접근으로 인한 런타임 오류 발생

optionalValue = optionalValue + 1

?

Optional

```
var optionalValue: Int? = 100
switch optionalValue {
    case .none:
        print("This Optional variable is nil")
    case .some(let value):
        print("Value is \(value)")
```

• nil 할당 가능

optionalValue = nil

• 기본 변수처럼 사용 불가 - 옵셔널과 일반 같은 다른 타입으로 연산 불가

optionalValue = optionalValue + 1

암시적 추출 옵셔널(Implicitly Unwrapped Optionals)

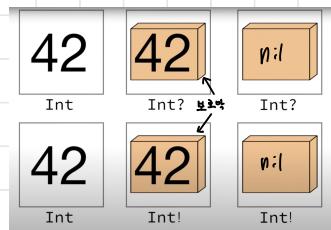
때때로 nil을 할당하고 싶지만, 옵셔널 바인딩으로 매번 값을 추출하기 귀찮거나 로직상으로 nil 때문에 런타임 오류가 발생하지 않을 것 같다는 확신이 들 때 nil을 할당해줄 수 있는 옵셔널이 아닌 변수나 상수가 있으면 좋을 겁니다. 이 때 사용하는 것이 암시적 추출 옵셔널입니다.

optional 값 추출

Optional Unwrapping

· optional Binding - 읍서널 바인딩 (nil 체크 + 인결된 값 추출)

· Force Unwrapping - 강제 추출



Optional Binding - 상자에 값이 있는지 없는지 노크

```
func printName(_ name: String) {
    print(name)
}

var myName: String? = nil
```

```
printName(myName)

↑
error: 전달되는 값의 type이 다르기 때문에
```

if - let

```
func printName(_ name: String) {
    print(name)
}

var myName: String! = nil

if let name: String = myName {
    printName(name)
} else {
    print("myName == nil")
}

printName(name)
```

↑
error: name 상수는 if-let 구문 내에서만 사용 가능.
상수 사용 범위로 빠져나가기 때문에 정의를 오류 발생

if-let

```
var myName: String? = "yagom"
var yourName: String? = nil

if let name = myName, let friend = yourName {
    print("\(name) and \(friend)")
}

// yourName이 nil이기 때문에 실행되지 않습니다
// 두줄이 모두 값이 있어야 함.

yourName = "hana"

if let name = myName, let friend = yourName {
    print("\(name) and \(friend)")
}

// yagom and hana
```

```
func printName(_ name: String) {
    print(name)
}
```

```
var myName: String? = "yagom"

printName(myName!) // yagom
myName = nil

print(myName!)
// 강제추출시 값이 없으므로 런타임 오류 발생
```

```
var yourName: String! = nil
```

printName(yourName)
// nil 값이 전달되기 때문에 런타임 오류 발생

↑
느낌표가 없는 이유: 만약 주석 optional type
처음 선언시 !를 가정하고 선언하는 것과 같다.

구조체

구조체 - Struct

- . Swift에서는 대부분의 type이 구조체로 이루어져 있을 정도로 구조체가 굉장히 중요한 역할을 담당하고 있다.
- . Swift의 구조체는 type을 정의하는 것에 대해서
대문자 camelCase를 사용함

// MARK: 프로퍼티 및 메서드

```
struct Sample {  
    ↳ 프로퍼티  
    var mutableProperty: Int = 100 // 가변 프로퍼티  
    let immutableProperty: Int = 100 // 불변 프로퍼티  
  
    static var typeProperty: Int = 100 // 타입 프로퍼티  
  
    // 인스턴스 메서드  
    func instanceMethod() {  
        print("instance method")  
    }  
  
    // 타입 메서드  
    static func typeMethod() {  
        print("type method")  
    }  
}
```

구조체 사용

```
// 가변 인스턴스 ↳ type ↳ instance  
var mutable: Sample = Sample()  
    ↳ 내부 property 값 변경 가능  
// mutable.mutableProperty = 200  
// mutable.immutableProperty = 200  
    ↳ 불변 property → 값 변경 X  
  
// 불변 인스턴스  
let immutable: Sample = Sample()  
    ↳ 레이저 선언할 때만 가변 property 라고 값 변경 X  
// immutable.mutableProperty = 200  
// immutable.immutableProperty = 200
```

타입 프로퍼티 및 메서드

```
Sample.typeProperty = 300  
Sample.typeMethod() // type method  
    ↳ type 자체에서 사용 가능한 property, method  
// mutable.typeProperty = 400  
// mutable.typeMethod()  
    ↳ instance에서 사용 불가
```

학생 구조체

```
struct Student {  
    var name: String = "unknown"  
    var class: String = "Swift"  
        ↳ 변수 이름과 사용하는 차이점  
    static func selfIntroduce() {  
        print("학생입니다")  
    }  
  
    func selfIntroduce() {  
        print("저는 \(self.class)반 \(name)입니다")  
    }  
}  
Student.selfIntroduce() // 학생입니다  
    ↳ type method  
var yagom: Student = Student()  
yagom.name = "yagom"  
yagom.class = "스위프트"  
yagom.selfIntroduce() // 저는 스위프트반 yagom입니다  
    ↳ instance를 만들기 위해  
let jina: Student = Student()  
    ↓  
// 불변 인스턴스로 프로퍼티 값 변경 불가  
// 컴파일 오류 발생  
// jina.name = "jina"  
jina.selfIntroduce() // 저는 Swift반 unknown입니다
```

Class

Class

구조체와 유사

구조체는 값 + type, 클래스는 참조 + type

Swift의 class는 다중상속이 안됨

//MARK: 프로퍼티 및 메서드

```
class Sample {
    var mutableProperty: Int = 100 // 가변 프로퍼티
    let immutableProperty: Int = 100 // 불변 프로퍼티

    static var typeProperty: Int = 100 // 타입 프로퍼티

    // 인스턴스 메서드
    func instanceMethod() {
        print("instance method")
    }

    // 재정의 가능 타입 메서드 - static
    static func typeMethod() {
        print("type method - static")
    }

    // 재정의 가능 타입 메서드 - class
    class func classMethod() {
        print("type method - class")
    }
}
```

Class 사용

```
var mutableReference: Sample = Sample()

mutableReference.mutableProperty = 200
//mutableReference.immutableProperty = 200
let immutableReference: Sample = Sample()

immutableReference.mutableProperty = 200
//immutableReference.immutableProperty = 200
let immutableReference = mutableReference

// 타입 프로퍼티 및 메서드
Sample.typeProperty = 300
Sample.typeMethod() // type method
```

학생 Class

```
class Student {
    var name: String = "unknown"
    var 'class': String = "Swift"

    func selfIntroduce() {
        print("학생타입입니다")
    }

    func selfIntroduce() {
        print("저는 \(self.class)반 \(name)입니다")
    }
}
```

```
Student.selfIntroduce() // 학생타입입니다
```

```
var yagom: Student = Student()
yagom.name = "yagom"
yagom.class = "스위프트"
yagom.selfIntroduce() // 저는 스위프트반 yagom입니다
```

```
let jina: Student = Student()
jina.name = "jina"
jina.selfIntroduce() // 저는 Swift반 jina입니다
```

→ 구조체와 다르게
let으로 선언해도 값 변경 가능

ENUM

- 각각의 case가 고유의 값이 됨
- enum은 type으로 대체가 camelCase로 이름 정의
- case들은 소문자 camelCase

```
enum Weekday {
    case mon
    case tue
    case wed
    case thu, fri, sat, sun
}
(→ day의 type)
var day: Weekday = Weekday.mon
day = .tue
print(day)

switch day {
    case .mon, .tue, .wed, .thu:
        print("평일입니다!")
    case Weekday.fri:
        print("즐금 파티!!")
    case .sat, .sun:
        print("신나는 주말!!")
}
```

//MARK: - 원시값

```
// enum의 rawValue를 정수값을 가질 수도 있습니다
// rawValue를 사용하면 됩니다
// case별로 각각 다른 값을 가져야합니다
|
| I
enum Fruit: Int {
    case apple = 0 ↓ 자동으로 증가
    case grape = 1
    case peach
    // case mango = 0
}

print("Fruit.peach.rawValue == \(Fruit.peach.rawValue)")
// Fruit.peach.rawValue == 2
```

//정수 타입 뿐 아니라

// Hashable 프로토콜을 따르는 모든 타입이 원시값의 타입으로 지정될 수 있습니다

```
enum School: String {
    case elementary = "초등"
    case middle = "중등"
    case high = "고등"
    case university
}

print("School.middle.rawValue == \(School.middle.rawValue)")
// School.middle.rawValue == 중등

print("School.university.rawValue == \(School.university.rawValue)")
// School.middle.rawValue == 대학교
↳ rawValue를 거쳐艰변
↳逆势의 이름을 가지게됨

//MARK: 원시값을 통해 초기화
```

//rawValue를 통해 초기화 할 수 있습니다

//rawValue가 case에 해당하지 않을 수 있으므로

//rawValue를 통해 초기화 한 인스턴스는 음서널 타입입니다

```
//let apple: Fruit = Fruit(rawValue: 0) → 0이 나올 수도 있기 때문
let apple: Fruit? = Fruit(rawValue: 0) → optional

if let orange: Fruit = Fruit(rawValue: 5) {
    print("rawValue 5에 해당하는 케이스는 \(orange)입니다")
} else {
    print("rawValue 5에 해당하는 케이스가 없습니다")
} // rawValue 5에 해당하는 케이스가 없습니다
```

. 메서드

```
enum Month {
    case dec, jan, feb
    case mar, apr, may
    case jun, jul, aug
    case sep, oct, nov
}

func printMessage() {
    switch self {
        case .mar, .apr, .may:
            print("봄입니다!")
        case .jun, .jul, .aug:
            print("여름 더워요~")
        case .sep, .oct, .nov:
            print("가을은 둑사의 계절!")
        case .dec, .jan, .feb:
            print("겨울입니다!")
    }
}
```

Month().Mar.printMessage()

값 type, 참조 type

Class

- 전통적인 OOP 관점에서의 클래스
- 단일상속
- [인스턴스/타입] 메서드
- [인스턴스/타입] 프로퍼티
- 참조 타입
- Apple 프레임워크의 대부분의 큰 빠대는 모두 클래스로 구성

구조체는 언제 사용하나?

- 연관된 몇몇의 값들을 모아서 하나의 데이터타입으로 표현하고 싶을 때
- 다른 객체 또는 함수 등으로 전달될 때
참조가 아닌 복사를 원할 때
- 자신을 상속할 경우가 없거나
사람이 다른 타입을 상속받을 필요가 없을 때
- Apple 프레임워크에서 프로그래밍을 할 때에는 주로 클래스를 많이 사용

Struct

- C 언어 등의 구조체보다 다양한 기능
- 상속 불가
- [인스턴스/타입] 메서드
- [인스턴스/타입] 프로퍼티
- 값 타입
- Swift의 대부분의 큰 빠대는 모두 구조체로 구성

Value vs Reference

- Value**
 - 데이터를 전달할 때 값을 복사하여 전달
- Reference**
 - 데이터를 전달할 때 값의 메모리 위치를 전달

Enum

- 다른 언어의 열거형과는 많이 다른 존재
- 상속 불가
- [인스턴스/타입] 메서드
- [인스턴스/타입] 연산 프로퍼티
- 값 타입

Class / Struct / Enum

Type	Class	Struct	Enum
Subclassing	O	X	X
Extension	O	O	O

```

/* 클래스, 구조체/열거형 비교 */
import Swift
//MARK:- Class vs Struct/Enum
struct ValueType {
    var property = 1
}
class ReferenceType {
    var property = 1
}

let firstStructInstance = ValueType()
var secondStructInstance = firstStructInstance
secondStructInstance.property = 2

print("first struct instance property : \(firstStructInstance.property)") // 1
print("second struct instance property : \(secondStructInstance.property)") // 2

let firstClassReference = ReferenceType()
var secondClassReference = firstClassReference
secondClassReference.property = 2
print("first class reference property : \(firstClassReference.property)") // 1
print("second class reference property : \(secondClassReference.property)") // 2

```

```

struct SomeStruct {
    var someProperty: String = "Property"
}

var someStructInstance: SomeStruct = SomeStruct()
func someFunction(structInstance: SomeStruct) {
    var localVar: SomeStruct = structInstance
    localVar.someProperty = "ABC"
}

someFunction(someStructInstance)
print(someStructInstance.someProperty) // "Property"

```

↗️ **클래스는** **값을** **참조하는** **변수** **다**
 ↗️ **구조체는** **값을** **복제하는** **변수** **다**

Data types in Swift

```

public struct Int
public struct Double
public struct String
public struct Dictionary<Key : Hashable, Value>
public struct Array<Element>
public struct Set<Element : Hashable>

```

Swift LOVES Struct

- 스위프트는 구조체, 열거형 사용을 선호
- Apple 프레임워크는 대부분 클래스 사용
- Apple 프레임워크 사용시 구조체/클래스 선택은 우리의 몫

```

class SomeClass {
    var someProperty: String = "Property"
}

var someClassInstance: SomeClass = SomeClass()
func someFunction(classInstance: SomeClass) {
    var localVar: SomeClass = classInstance
    localVar.someProperty = "ABC"
}

someFunction(someClassInstance)
print(someClassInstance.someProperty) // "ABC"

```

↗️ **클래스는** **값을** **참조하는** **변수** **다**
 ↗️ **구조체는** **값을** **복제하는** **변수** **다**

클로저

코드의 블록

· 일급 시민 (first-class citizen)

· 변수, 상수 등으로 저장, 전달인자로 전달해 가능

· 함수: 이름이 있는 클로저

☞ 반환 type 없을 때: Void

· ((매개변수 모음) → 반환타입) in

실행코드

}

· 함수를 사용한다면

```
func sumFunction(a: Int, b: Int) -> Int {  
    return a + b  
}  
  
var sumResult: Int = sumFunction(a: 1, b: 2)  
  
print(sumResult) // 3  
  
// 클로저의 사용  
var sum: (Int, Int) -> Int = { (a: Int, b: Int) -> Int in  
    return a + b  
} // 매개변수  
  
sumResult = sum(1, 2)  
print(sumResult) // 3
```

// 함수는 클로저의 일종이므로

// sum 변수에는 당연히 함수도 할당할 수 있습니다

sum = sumFunction(a:b:)

↑

sumResult = sum(1, 2)
print(sumResult) // 3

클로저는 주로 함수의 전달인자로 많이 사용됨

클로저의 활용으로 사용

// MARK: - 함수의 전달인자로서의 클로저

```
let add: (Int, Int) -> Int  
add = { (a: Int, b: Int) -> Int in  
    return a + b  
}  
  
let subtract: (Int, Int) -> Int  
subtract = { (a: Int, b: Int) -> Int in  
    return a - b  
}  
  
let divide: (Int, Int) -> Int  
divide = { (a: Int, b: Int) -> Int in  
    return a / b  
}  
  
func calculate(a: Int, b: Int, method: (Int, Int) -> Int) -> Int {  
    return method(a, b)  
}  
var calculated: Int  
  
calculated = calculate(a: 50, b: 10, method: add)  
print(calculated) // 60  
  
calculated = calculate(a: 50, b: 10, method: subtract)  
print(calculated) // 40  
calculated = calculate(a: 50, b: 10, method: divide)  
print(calculated) // 5  
  
calculated = calculate(a: 50, b: 10, method: { (left: Int, right: Int) -> Int  
})  
return left * right  
}  
print(calculated) // 500
```

method라는 이름으로 클로저를 넘겨준다.

클로저 고급

• 후행 클로저

• 반환타입 생략

• 단축 인자이름

• 양식적 반환 표현

```
func calculate(a: Int, b: Int, method: (Int, Int) -> Int) -> Int {  
    return method(a, b)  
}  
  
var result: Int  
  
// MARK: - 후행 클로저  
// 클로저가 함수의 마지막 전달인자라면  
// 마지막 매개변수 이름을 생략한 후  
// 함수 소괄호 외부에 클로저를 구현할 수 있습니다  
  
result = calculate(a: 10, b: 10) { (left: Int, right: Int) -> Int in  
    return left + right  
}  
  
print(result) // 20
```

↳ 마지막 인자: 클로저

반환타입 생략

```
calculate(a:b:method:) 함수의 method 매개변수는 Int 타입을 반환할 것이라는 사실을  
컴파일러도 알기 때문에 굳이 클로저에서 반환타입을 명시해 주지 않아도 됩니다. 대신 in 키워드는 생  
략할 수 없습니다  
  
result = calculate(a: 10, b: 10, method: { (left: Int, right: Int) in  
    return left + right  
})  
  
print(result) // 20
```

→ Int 생략

단축 인자이름

클로저의 매개변수 이름이 굳이 불필요하다면 단축 인자이름을 활용할 수 있습니다. 단축 인자이름은 클
로저의 매개변수의 순서대로 \$0, \$1, \$2 ... 처럼 표현합니다.

```
result = calculate(a: 10, b: 10, method: {  
    return $0 + $1  
})  
    a+b  
  
print(result) // 20
```

// 단언히 후행 클로저와 함께 사용할 수 있습니다

```
result = calculate(a: 10, b: 10) {  
    return $0 + $1  
}  
  
print(result) // 20
```

암시적 반환 표현

클로저가 반환하는 값이 있다면 클로저의 마지막 줄의 결과값은 암시적으로 반환값으로 취급합니다.

```
result = calculate(a: 10, b: 10) {  
    $0 + $1  
}  
  
print(result) // 20  
  
// 같은하게 한 줄로 표현해 줄 수도 있습니다  
result = calculate(a: 10, b: 10) { $0 + $1 }  
  
print(result) // 20
```

축약 전과 후 비교

```
// 축약 전  
result = calculate(a: 10, b: 10, method: { (left: Int, right: Int) -> Int  
})  
    return left + right  
}  
  
// 축약 후  
result = calculate(a: 10, b: 10) { $0 + $1 }  
  
print(result) // 20
```

* 너무 축약을 해버리면 다른 사용들은 제대로 못할수
있다.

Property

```
// 인스턴스 저장 프로퍼티
var name: String = ""          `class`
var class: String = "Student"
var koreanAge: Int = 0

// 인스턴스 연산 프로퍼티
var westernAge: Int {
    get {
        return koreanAge - 1
    }

    set(inputValue) {
        koreanAge = inputValue + 1
    }
}

// 타입 저장 프로퍼티
static var typeDescription: String = "학생"

/*
// 인스턴스 메서드
func selfIntroduce() {
    print("저는 \(self.class)의 \(name)입니다")
}

*/
// 읽기전용 인스턴스 연산 프로퍼티
// 간단히 위의 selfIntroduce() 메서드를 대체할 수 있습니다
var selfIntroduction: String {
    get {
        return "저는 \(self.class)의 \(name)입니다"
    }
}

/*
// 타입 메서드
static func selfIntroduce() {
    print("학생입니다")
}

*/
// 읽기전용 타입 연산 프로퍼티
// 읽기전용에서는 get을 생략할 수 있습니다
static var selfIntroduction: String {
    return "학생입니다"
}
```

```
// 타입 연산 프로퍼티 사용
print(Student.selfIntroduction)
// 학생입니다

// 인스턴스 생성
var yagom: Student = Student()
yagom.koreanAge = 10

// 인스턴스 저장 프로퍼티 사용
yagom.name = "yagom"
print(yagom.name)
// yagom

// 인스턴스 연산 프로퍼티 사용
print(yagom.selfIntroduction)
// 저는 Swift인 yagom입니다

print("제 한국나이는 \(yagom.koreanAge)살이고, 미국나이는 \(yagom.westernAge)살입니다.")
// 제 한국나이는 10살이고, 미국나이는 9살입니다.

struct Money {
    var currencyRate: Double = 1100
    var dollar: Double = 0
    var won: Double {
        get {
            return dollar * currencyRate
        }
        set(특별히 매번 newValue를 안드면 newValue로 사용) {
            dollar = newValue / currencyRate 사용
            dollar = newValue / currencyRate
        }
    }
}

var moneyInMyPocket = Money()

moneyInMyPocket.won = 11000

print(moneyInMyPocket.won)
// 11000.0

moneyInMyPocket.dollar = 10

print(moneyInMyPocket.won)
// 11000.0
```

지역변수 및 전역변수

저장 프로퍼티와 연산 프로퍼티의 기능은 함수, 메서드, 클로저, 타입 등의 외부에 위치한 지역/전역 변수에도 모두 사용 가능합니다.

```
var a: Int = 100
var b: Int = 200
var sum: Int {
    return a + b
}

print(sum) // 300
```

COPY

property 감시자

프로퍼티 감시자

프로퍼티 감시자를 사용하면 프로퍼티 값이 변경될 때 원하는 동작을 수행할 수 있습니다. 값이 변경되기 직전에 `willSet` 블록이 호출됩니다. 둘 중 필요한 하나님 구현해 주어도 무관합니다. 변경되려는 값이 기존 값과 똑같더라도 프로퍼티 감시자는 항상 동작합니다.

`willSet` 블록에서 암시적 매개변수 `newValue`를 사용할 수 있고, `didSet` 블록에서 암시적 매개변수 `oldValue`를 사용할 수 있습니다.

프로퍼티 감시자는 연산 프로퍼티에 사용할 수 없습니다.

```
struct Money {
    // 프로퍼티 감시자 사용 → 자동 property
    var currencyRate: Double = 1100
    willSet(newRate) {
        print("환율이 \(currencyRate)에서 \(newRate)으로 변경될 예정입니다")
        // 바로이전에 환율
    }
    didSet(oldRate) {
        print("환율이 \(oldRate)에서 \(currencyRate)으로 변경되었습니다")
        // 비교연산
    }
}

// 프로퍼티 감시자 사용
var dollar: Double = 0
// willSet의 암시적 매개변수 이름 newValue
willSet() {
    print("\(dollar)달러에서 \(newValue)달러로 변경될 예정입니다")
    // 매개변수를 만들지 않았을 때
}

// didSet의 암시적 매개변수 이름 oldValue
didSet() {
    print("\(oldValue)달러에서 \(dollar)달러로 변경되었습니다")
}

// 연산 프로퍼티
var won: Double {
    get {
        return dollar * currencyRate
    }
    set {
        dollar = newValue / currencyRate
    }
}

/* 프로퍼티 감시자와 연산 프로퍼티 기능을 동시에 사용할 수 없습니다
willSet {
    저정도는 값이 바뀔때 사용되는 것에 때문
}
*/
}
```

```
var moneyInMyPocket: Money = Money()

// 환율이 1100.0에서 1150.0으로 변경될 예정입니다
moneyInMyPocket.currencyRate = 1150
// 환율이 1100.0에서 1150.0으로 변경되었습니다

// 0.0달러에서 10.0달러로 변경될 예정입니다
moneyInMyPocket.dollar = 10
// 0.0달러에서 10.0달러로 변경되었습니다

print(moneyInMyPocket.won)
// 11500.0

// 프로퍼티 감시자의 기능은
// 함수, 메서드, 클로저, 타입 등의 외부에 위치한
// 지역/전역 변수에도 모두 사용 가능합니다

var a: Int = 100 {
    willSet {
        print("\(a)에서 \(newValue)로 변경될 예정입니다")
    }
    didSet {
        print("\(oldValue)에서 \(a)로 변경되었습니다")
    }
}

// 100에서 200로 변경될 예정입니다 → willSet
a = 200
// 100에서 200로 변경되었습니다 → didSet
```

