
It doesn't work..... why?
It works..... why?
WeKnowMemes

1) print, debugPrint, dump, NSLog

2) Type Annotation vs. **Type Inference**

```
var q = "C"    //String 일까 Character 일까
```

3) **nil** vs. NULL(java, c)

4) 변수 선언 var vs. let

* let을 왜 쓸까?
* 원주율, 환율을 변수로 선언한다면?

5) 세미콜론

```
* let cat = "😺"; print(cat)
```

6) 부동소수점 Float와 Double

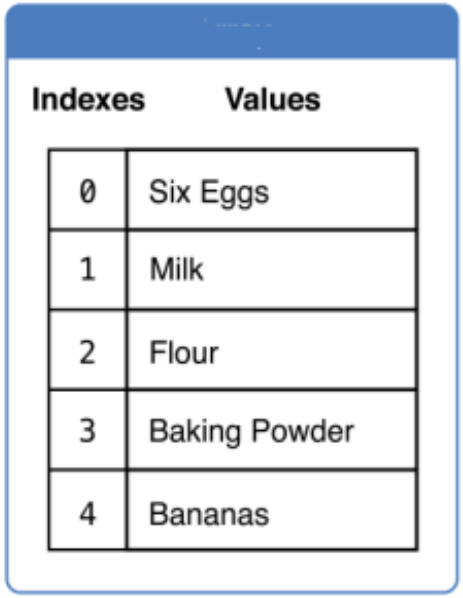
* Float 0.001을 1000번 더하면?

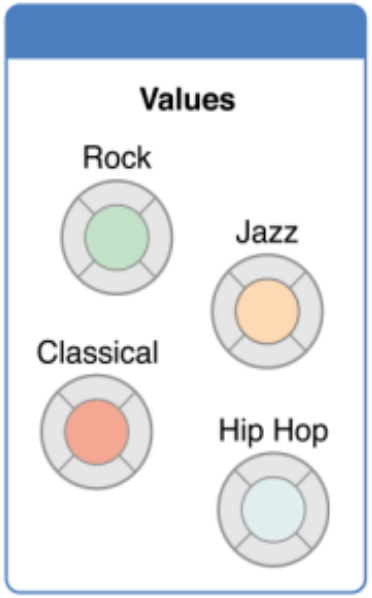
7) Compound Assignment Operators

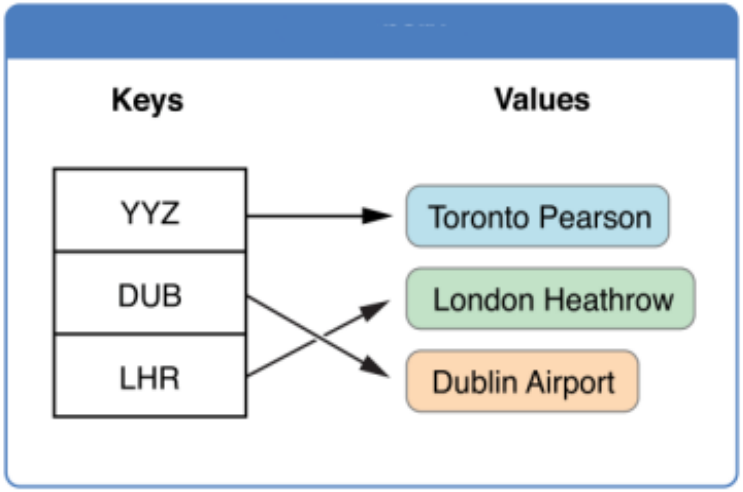
```
var a = 2
var b = (a += 2) // Error?
print(b)
```

8)
```
let lineIndentation = """
    This string starts with a line break.
       It also ends with a line break.
    """
```

9)







10)
```
var arrayOfStrings: [String] = [] // type annotation + array literal
var arrayOfStrings = [String]() // invoking the [String] initializer
var arrayOfStrings = Array<String>() // without syntactic sugar(syntax sugar)

let uniqueUnordered = Array(Set(array))
let uniqueOrdered = Array(NSOrderedSet(array: array))
```

11) 조건문

Nil-Coalescing Operator(nil 병합 연산자) : (a **??** b) == (a != nil ? a! : b)

Short-Circuit Evaluations (left-associative)

```
if (enteredDoorCode && passedScan) || hasDoorKey || knowsPassword {
    //...
}
```

```
if 0...10 ~= number {
}
```

```
if 200..<300 ~= response.statusCode { ... }
```

```
if #available(iOS 10, macOS 10.12, *) {
   // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
   // Fall back to earlier iOS and macOS APIs
}
```

switch 에서 **default**는 반드시 써야 하는가?
fallthrough는 필요한가?

```
switch anotherCharacter {
   case "a": // empty body is Error ?
   case "A":
```

조건절 패턴 매칭
1) 호환성체크 패턴
2) 옵셔널 바인딩 패턴
3) Case 조건 패턴

12) 반복문

```
for i in 0..<3 {
    print(i)
}
```

```
for i in 0...2 {
    print(i)
}
```

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}
```

```
for (index, name) in names.enumerated()
```

```
let arr = [1,2,3,4]
```

```
func collectionTestA() {
    for i in arr {
        if (i == 2) {
            return
        }
        print(i)
    }
}
```

```
func collectionTestB() {
    arr.forEach {
        if $0 == 2 {
            return
        }
        print($0)
    }
}
```

13) **Any** vs. **AnyObject**

The **Protocol** to which all class types implicitly **conform**

14) Struct

값타입의 속성들을 인스턴스에서 값을 수정하려면?

```
struct Counter {
    var value = 0
    mutating func next() {
        value += 1   // Left side of mutating operator isn't mutable: 'self' is immutable
    }                // 따라서 Struct, Enum 속성값을 바꿀때는 mutating 키워드 필요
}
```

```
var counter = Counter()
counter.next() // counter.value += 1 이 경우는 ?
```

All structures have an automatically generated **memberwise initializer**

언제 struct를 쓰면 유리할까?

15) Enum

init, method 사용 가능

associated values

```
enum Action {
    case jump
    case kick
    case move(distance: Float)
}
performAction(.move(distance: 3.3))
if case .move(let distance) = action {
    print("Moving: ₩(distance)")
}
```

```
enum MetasyntacticVariable: Int {
    case foo // rawValue is automatically 0
    case bar = 7
    case baz // rawValue is ☐
    ...

enum MarsMoon: String {
    case phobos // rawValue is ☐
    case deimos // rawValue is ☐
    ...
```

16) Class

구조체와의 차이: https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let constantDog = Dog(name: "Rover", age: 5) // This instance is a constant.
constantDog.name = "Fido" // Error ?, Dog이 Struct(Value Type)이면 Error ?
constantDog.age = 6 // Error ?
constantDog = Dog(name: "Fido", age: 6) // Error?

var variableDog = Dog(name: "Spot", age 7) // This instance is a variable.
variableDog.name = "Ace" // Error?
variableDog.age = 8 // Error?
variableDog = Dog(name: "Ace", age: 8) // Error?
```

17) Call by value, Call by Reference

18) Optional == ?

간보는 변수 : 있을수도 없을수도,

```
var firstName: String? = "Bob"
var myBool: Bool? = false
if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName is bob and myBool was false!")
}
```

값을 추출하는 방법은 Optional Binding과 강제추출(Wrapped !)

```
var number: Int?
if let unwrappedNumber = number {
    print("number: ₩(unwrappedNumber)") // Will not enter this line
} else {
    print("number was not assigned a value")
}
```

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: ₩(unwrappedNumber)")
```

19) Function (1/3)

```
func magicNumber(number1: Int, number2: Int)        magicNumber(10, number2: 5)

func magicNumber(one number1: Int, two number2: Int)

func magicNumber(one number1: Int = 5, two number2: Int = 10)

func sum(_ numbers: Int...) -> Int

func updateFruit(fruit: inout Int)      updateFruit(fruit: &apples)

func errorThrower() throws -> String

func maths(number: Int) -> (times2: Int, times3: Int)    let resultTuple = maths(5)

func loadData(id: String, completion:(String) -> ()) {     loadData(id: "123") { result in
    completion("this is trailing closure syntax")               print(result)
}                                                           }

func changeValue(num: Int) {
    num = 10    // Error ?
}
```

class, struct, enum, protocol에서는 method
method는 instance method, type method(static func / class func)가 있다.

19) Function (2/3)

```
func greeting(for person: String) -> String {
    "Hello, " + person + "!"     // Implicit Return is Error?
}
```

Every function has a specific **function type :** parameter types and the return type

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {        (Int, Int) -> Int
     return a + b
}
```

```
var mathFunction: (Int, Int) -> Int = addTwoInts
"Define a variable called mathFunction,
 which has a type of 'a function that takes two Int values,
 and returns an Int value.'
 Set this new variable to refer to the function called addTwoInts."
print(mathFunction(2, 3)) // result?
```

함수타입은 일반적으로 right-associative

```
Int -> Int -> Int   ==   Int -> (Int -> Int)
```

19) Function (3/3)

Functions are a **first-class** type

*** as Parameter Types**

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print(mathFunction(a, b))
}
printMathResult(addTwoInts, 3, 5)  // result?
```

*** as Return Types**

```
func stock(_ money: Int) -> Int {
    return money - 1
}
func bitcoin(_ money: Int) -> Int {
    return money - 2
}
func investment(highRisk: Bool) -> (Int) -> Int {
    return highRisk ? bitcoin : stock
}

let myChoice = investment(highRisk: true)
// myChoice now refers to the □□□□ function
var currentMoney = 3
print(myChoice(currentMoney))   // result?
```

* higher-order function?

19) Closure (1/4)

self-contained blocks - pass around and used in your code
closing over 유래 – capturing values in context (Context = Scope)

```
{ (parameters) -> return type in
    statements   // body
}
```

Closure 최적화(왜 어려울까?)

```
let _ = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Inferring Type From Context

```
let _ = names.sorted(by: { s1, s2 in return s1 > s2 } ) // (String, String) -> Bool
```

Implicit Returns from Single-Expression Closures

```
let _ = names.sorted(by: { s1, s2 in s1 > s2 } )
```

Shorthand Argument Names

```
let _ = names.sorted(by: { $0 > $1 } )  // $0, $1, $2...   let squared = { $0 * $0 }(12)
```

Operator Methods

```
let _ = names.sorted(by: >)
```

19) Closure (2/4)

Trailing Closures

Closure표현식을 함수의 인자들 중 마지막 인자로 호출하여 넘길때 사용.
이때, 마지막 인자 **라벨은 사용하지 않음, Closure 표현식이 "길"때 유용**

```
let _ = names.sorted() { $0 > $1 }
```

함수의 유일한 인자로 후행 클로저만 제공 되면 함수 뒤의 **괄호를 생략**할 수 있음

```
let _ = names.sorted { $0 > $1 }
```

함수에서 여러 후행 클로저가 올때, **첫번째 인자 라벨은 생략**하지만, 나머지 클로저들은 인자라벨을 남겨둔다

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}

loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
} onFailure: {
    print("Couldn't download the next picture.")
}
```

19) Closure (3/4)

Capturing Value → Closures Are Reference Types

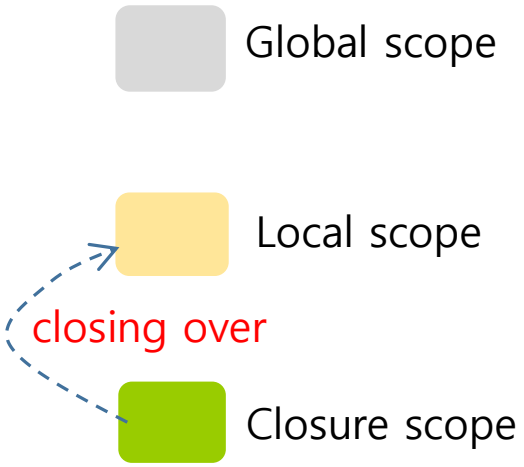
클로저의 body 안에서 상수나 변수의 값을 캡쳐(저장), 심지어 함수 범위를 벗어나더라도!
대표적인 예 : 전역함수, 중첩함수 (클로저의 간단한 형태) // 왜 중첩함수를 쓸까?

```
func makeTotal(amount: Int) -> () -> Int {
    var runningTotal = 0

    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer // () 주의!!
}
```

```
let byTen = makeTotal(amount: 10)
print(byTen())   // result?
print(byTen())   // result?
...
let bySeven = makeTotal(amount: 7)
print(bySeven())  //result?
Print(byTen())  //result?
...
```

```
func addScore(points: Int) -> Int {
    let score = 42
    let calculate = {
        return score + points
    }
    return calculate()
}

let value = addScore(points: 11)
print(value)
```

closing over



19) Closure (4/4)

Escaping Closure

closure가 함수가 return 될때까지 대기 한 후 실행(escape)이 된다(즉, 함수 밖에서도 실행!)
@escaping 사용을 하는 closure는 self 를 명시적으로 작성해야 한다
Swift 3.0부터는 closure가 기본이 non-escaping.
대표적인 예: 비동기 처리(completeHandler)

```
var completionHandlers: [() -> Void] = []
func withEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
func withNonescapingClosure(closure: () -> Void) {
    closure()    // 함수 안에서 끝나는 클로저
}

class SomeClass {
    var x = 10
    func doSomething() {
        // 클로저 안의 self가 class의 instance를 참조하면 강한 참조 사이클을 유발할 수 있다.
        // An escaping closure must refer to self explicitly, or use a capture list
        withEscapingClosure { self.x = 100 }

        // self implicity
        withNonescapingClosure { x = 200 }
    }
}
```

```
let instance = SomeClass()
instance.doSomething()
print(instance.x) // result ?

completionHandlers.first?()
print(instance.x) // result ?
```

Capture List : **break** the strong reference

```
withEscapingClosure { [self] in x = 100 }
```

```swift
var completionHandlers = [() -> Void]()
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
    print(">> end of function with escape")
}

func someFunctionWithNonescapingClosure(closure: () -> Void) {
    closure()
    print(">> end of function with none escape")
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure {
            self.x = 100
            print(">> end of escape closure")
        }
        someFunctionWithNonescapingClosure {
            x = 200
            print(">> end of non-escape closure")
        }
        print(">> end of doSomething()")
    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)
// Prints "200"

completionHandlers.first?()
print(instance.x)
// Prints "100"
```

```
>> end of function with escape
>> end of non-escape closure
>> end of function with none escape
>> end of doSomething()
200
>> end of escape closure
100
```

* Strong Reference Cycles in Escaping Closures

```
class NetworkManager {
    var onFinishRequest: ((String) -> Void)?

    func makeRequest(completionHandler: @escaping (String) -> Void) {
        onFinishRequest = completionHandler
        // Do lengthy task...
        self.finishRequest()
    }

    func finishRequest() {
        onFinishRequest?("some data")
    }
}
```

```
class API {
    var manager = NetworkManager()
    var data = ""
    init() {
        print("init API")
    }

    func getData() {
        manager.makeRequest(completionHandler: { data in
            self.data = data
            print("called completion handler with : \(data)")
        })
    }

    deinit {
        print("deinit API")
    }
}
```

```
var api:API? = API()
api?.getData()  // 이 줄을 제거하면?
api = nil
// 기대하는 결과가 아래와 같은가?
init API
called completion handler with : some data
deinit API
```

해결책1

```
manager.makeRequest(completionHandler: { [weak self] data in
    self?.data = data
    ...
})
```

해결책2

```
func finishRequest() {
    onFinishRequest?("some data")
    onFinishRequest = nil
}
```

20) Property
클래스, 구조체, 열거형의 연관된 값

Stored property
일반적인 메모리 할당(값저장), 클래스, 구조체

Computed property : get/set
read-only는 값이 고정되지 않기 때문에 get만 사용하고 □□□로 선언!
연산프로퍼티 값을 계산하는데 쓴다. 클래스, 구조체, 열거형

Type property : static/class
그자체가 타입이며 객체 인스턴스 변수가 아님

Property Observers
willSet(oldValue) and didSet(newValue) are entirely legal, but oldValue, newValue는 피하라.

Lazy Stored Properties
인스턴스 초기화가 완료될 때까지 초기값을 가져올 수 없어 항상 □□□ 로 선언해야한다.

```
lazy □□□ veryExpensiveVariable = expensiveMethod()

lazy □□□ veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer:arc4random_uniform(5))
    return str
}()
```

* Global/Local 상수와 변수는 lazy와 같은 방식일까?

21) Protocol (1/3)

- adopted by a **class, structure, or enumeration**
- Any type that satisfies the **requirements** of a protocol is said to **conform** to that protocol

*** 프로퍼티 요구사항**

stored property, computed property는 사용할 수 없다
단, getter/setter를 지정만 하는것으로는 사용 가능, 이때 prefixed with the **var** keyword

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }     // static var, class var 가능
    var doesNotNeedToBeSettable: Int { get }
}
```

*** 메소드 요구사항**

```
protocol SomeProtocol {
    func someTypeMethod()   // instance method, type method 가능, default value 불가
}
```

*** Mutating 메소드 요구사항**

```
protocol SomeProtocol {
    mutating func toggle()  // struct, enum 타입에서만 사용 가능
}
```

*** 초기화(init) 요구사항**

```
protocol SomeProtocol {
    init(parameter: Int)
}
```

→

```
class SomeSuperClass: SomeProtocol {
    required init(parameter: Int) {  // required !!
    } ...
```

21) Protocol (2/3)

Protocols as Types : Protocol is fully-fledged type, Existential type(실존타입)
→ 구현체는 없지만 여러곳에서 사용 가능(delegate, 다형성에 유리)

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

Delegation is a design pattern : Hand-Off (=Delegate)
→ iOS 대표적인 예: UITableView class uses the UITableViewDelegate

```
struct IceCream {
    var size: Int = 5
    var hasChocolateChips: Bool = false
}

class IceCreamFactory {
    func makeIceCream() {
        var ic = IceCream()
        ic.hasChocolateChips = true
    } ...
```

```
// 아이스크림을 가게, 온라인으로 판매한다면?
protocol IceCreamDelegate {
    func produced(_ ic: Icecream)
}

class IceCreamFactory {
    var delegate: IceCreamDelegate?
    func makeIceCream() {
        var ic = IceCream()
        ic.hasChocolateChips = true
        delegate?.produced(ic)
    } ...

class IceCreamShop: IceCreamDelegate {
    func produced(_ ic: IceCream) {
        print("Yay! A new icecream arrived \(ic.size)")
    } ...

let shop = IceCreamShop()
let factory = IceCreamFactory()
factory.delegate = shop
factory.makeIceCream()  // resut?
```

21) Protocol (3/3)

Protocol Collection

```
let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Protocol Inheritance

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {  // ...
```

Class-Only Protocols

```
protocol SomeClassOnlyProtocol: AnyObject {
    // class-only protocol definition goes here
```

Protocol Composition

```
func doStuff(object: MyProtocol & AnotherProtocol) { // ...
```

Checking for Protocol Conformance

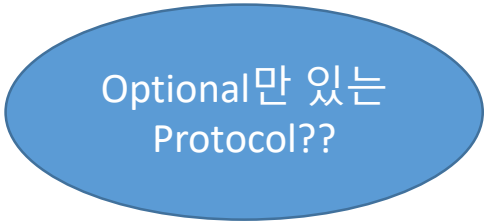
is : returns true/false

as? : returns an optional value of the protocol's type, 준수하지 않으면 nil

as! : forces the downcast to the protocol type, 실패시 a runtime error

Optional Protocol Requirements

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
var dataSource: CounterDataSource?
if let amount = dataSource?.increment?(forCount: 5) { ...
```



감동적인 개발 신입이 되는 방법

- 회사안에서 나만의 go to 멘토 찾기
- 일관성있게 꾸준히 노력해라 - 엄청난 컴파운딩 효과
- 실수를 통해 배우자
- 회사와 팀에 대한 이해를 한다
- 팀이 맡고 있는 프로젝트에 대해 이해 한다
- 팀&프로젝트에 관련 문서들을 받는다(프로젝트의 아키텍쳐를 이해한다)
- 제품을 이해하고 직접 써 본다 (제품에 대한 이해력 향상 - 용어, 개념, 장단점)
- 다양한 모듈의 이슈들을 받아서 해결한다(직접 해봐야 한다)
- 팀의 프로세스를 더 잘 이해한다.
  (어떻게 일하는가, 어떻게 정보는 공유하는가, 프로세스 개선점)
- 잡 디스크립션을 지금 & 다음 레벨 둘다 받는다
- 긍정적으로 능동적인 커뮤니케이션!

22) Initialization (1/11)

Classes, structures의 Stored Property들은 반드시 초기화!

// Enum? Optional의 기본값은?

■ Initializers

```
init() {
    // perform some initialization here, Objective-C와는 달리 return이 없다
}
```

■ Default Property Values

```
class Person {
    var name = "BBora"
}
```

- 기본값 초기화는 인스턴스를 만들때마다 동일한 값을 가질때 유리
  따라서 위 예제는 실무에서는 적합한가?
  쇼핑구매아이템에서의 var isPurchased = false?
- 그외 장점
  코드 가독성을 좋게하는 간단.
  상속관계에서 복잡할때 먼저 초기화 되어 명확!

■ Customizing Initialization

- 함수와 같이 파라미터, 라벨을 써서 초기화
- init이라는 이름만 가지므로 파라미터는 반드시 사용하여 초기화 해야 함.
  단, init (_ name: String) {.... } 과 같이 라벨을 생략할때
  var person = Person("BBora") 가능

22) Initialization (2/11)

■ **Default Initializers**

struct, class에서 1)기본값들로 Property들이 초기화를 제공하고,
2)사용자 정의 생성자가 하나도 없을때 !

```
Class IceCream() {
    var name = "민트초고봉봉"          →  let ic = IceCream()
    var hasChobochip = true
}
```

그러나 init(isCake: Bool) { ...} 과 같은 사용자 정의 생성자가 있을때는 불가능(let ic = ~~IceCream()~~)

컴파일러 : Missing argument for parameter 'isCake' is call

■ **구조체 타입에서의 memberwise (멤버단위) 생성자**

기본생성자와 다른점은 초기값이 없더라도 허용

```
struct Size {
    var width = 0.0, height = 0.0   // var width, height 로 선언해도 가능
}                                   // 단, 인스턴스 생성시 파리미터가 반드시 필요하다

let zeroByZero = Size() // 다음 각각에 대해 width = ?, height = ?
let twoByTwo = Size(width: 2.0, height: 2.0)
let zeroByTwo = Size(height: 2.0)
```

■ **Initializer Delegation for Value Types**

- Value types (struct, enum) 에서 Initializers can call other initializers, avoids duplicating code
- 사용자 정의 (custom) 생성자를 정의했다면, 더이상 memberwise를 호출할 수 없다.
- 상속이 안되니깐 Simple!(Only delegate to another initializer)
- 사용자 정의 생성자를 쓸때, 다른 생성자와 동일한 Value Type에 접근할때는 **self.init** 사용한다.

22) Initialization (3/11)

■ Initializer Delegation for Value Types (Cont.)

```
struct Size {
   var width = 0.0, height = 0.0
}
```

```
struct Point {
   var x = 0.0, y = 0.0
}
```

```
struct Rect {  // 몇개 custom initializers?
   var origin = Point()
   var size = Size()

   init() {} //빈 Body가 허용이 되는가?

   init(origin: Point, size: Size) {
      self.origin = origin
      self.size = size
   }
   //Rect가 클래스 타입이면 아래 생성자는 허용이 되는가?
   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) //delegate!!
   }
}
```

```
let basicRect = Rect()
//Default zero-initialized
//   origin is (0.0, 0.0) and its size is (0.0, 0.0)

let originRect = Rect(origin: Point(x: 2.0, y: 2.0), size: Size(width: 5.0, height: 5.0))
//Memberwise initializer
//   origin is (?, ?) and its size is (?, ?)

let centerRect = Rect(center: Point(x: 4.0, y: 4.0), size: Size(width: 3.0, height: 3.0))
//  origin is (?, ?) and its size is (?, ?)
```

22) Initialization (4/11)

■ Initializer Delegation for Value Types (Cont.)

Tip!

기본 생성자와 memberwise, 또한 사용자 정의 초기화까지 모두 사용하길 원할때는 원래 구현영역에 코드를 추가하지 말고 **Extenstions**을 사용해라

앞의 예제처럼 init() 과 init(origin:size:) 초기화를 정의하지 않고도 동일한 효과를 얻을 수 있다.

```
struct Rect {
    var origin = Point()
    var size = Size()
}

extension Rect {
    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)
    }
}
```

22) Initialization (5/11)

Class Inheritance and Initialization

■ **Designated Initializers**

- 지정된 초기화는 클래스에 대해 우선으로 초기화
- 모든 클래스는 적어도 하나의 designated 초기화(개수 최소화)

```
init(<parameters>) {
    <statements>
}
```

■ **Convenience Initializers**

- 편리한 초기화는 지정된 초기화의 일부 매개변수를 기본값으로 설정하여 지정된 초기화를 호출하기 위해 정의할 수 있다.
- 편리 초기화는 필요에 따라 사용하는것이 좋다.

```
convenience init(<parameters>) {
    <statements>
}
```

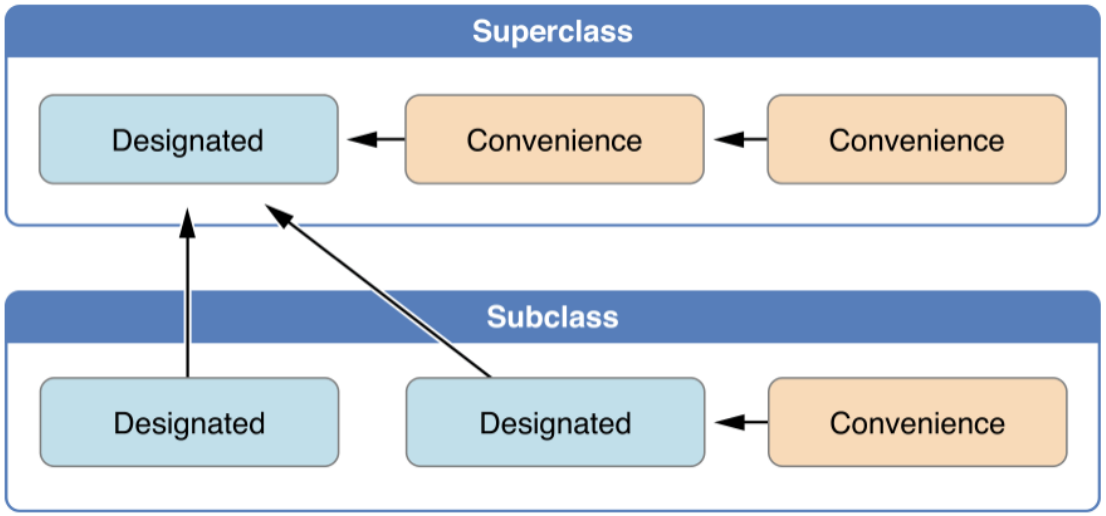
■ **Initializer Delegation** for **Class Types**

Rule 1. 지정된 초기화는 반드시 바로 위의 상위 클래스의 지정된 초기화를 호출해야한다.
Rule 2. 편리한 초기화는 반드시 같은 클래스의 다른 초기화를 호출한다.
Rule 3. 편리한 초기화는 궁극적으로 반드시 지정된 초기화를 호출한다.

Remember!
Designated initializers must always ***delegate up;*** **super.init**을 사용한다.
Convenience initializers must always ***delegate across;*** **self.init**을 사용한다

22) Initialization (6/11)

Class Inheritance and Initialization (Cont.)



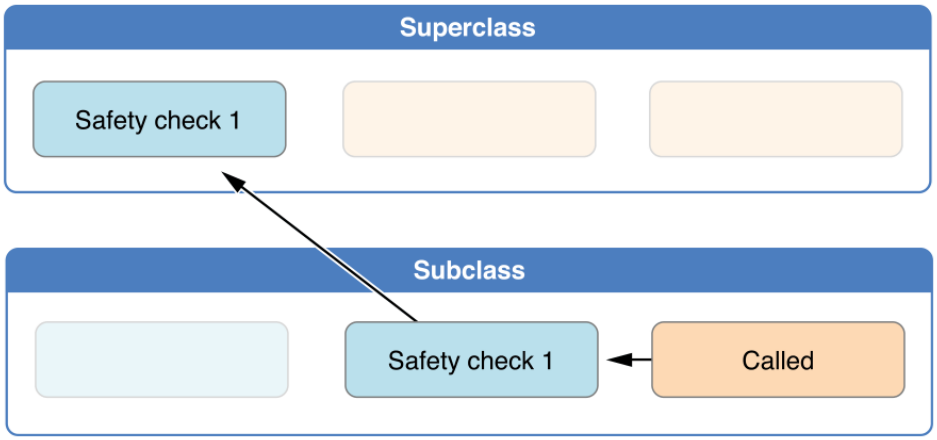

```
// 다음 4개의 생성자들 중
// Designated Initializer는 어느것?
init(content: String, sender: String, recipient: String) {
    self.content = content
    //Rule 1:  Calling designated Initializer from immediate superclass  (and Safety Check 1, Next Page!)
    super.init(sender: sender, recipient: recipient)
}

convenience init() {
    self.init(content: "")  //Rule 2:  Calling another initializer in same class
}

convenience init(content: String) {
    self.init(content: content, sender: "Myself")  //Rule 2:  Calling another initializer in same class
}

convenience init(content: String, sender: String) {
    //Rule 2 and 3:  Calling the Designated Initializer in same class
    self.init(content: content, sender: sender, recipient: sender)
}
```

22) Initialization (7/11)

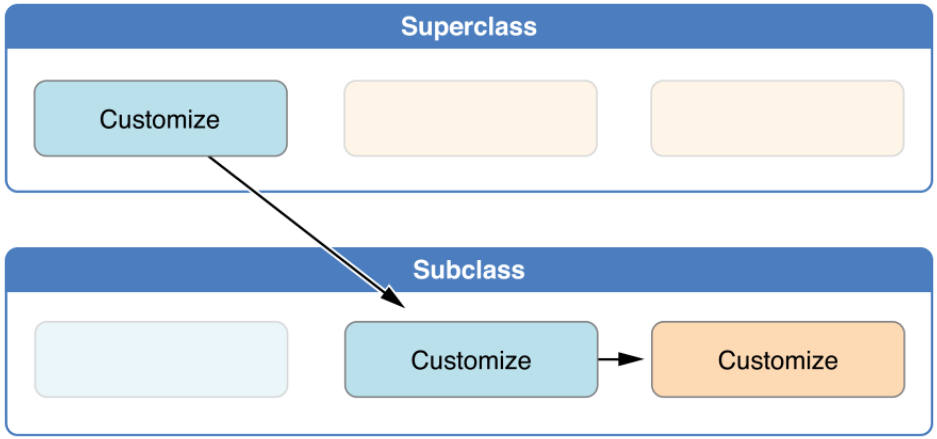
Class Inheritance and Initialization(Cont.)

■ Two-Phase Initialization

- 1단계: 클래스에 의해서 각 저장 속성의 초기값이 할당.
- 2단계: 새로운 instance가 사용준비가 되기 전에 Stored Property를 초기화 할 수 있는 기회를 획득.
- Swift의 컴파일러는 2개의 단계를 처리할때 다음 4개의 안전 사항을 체크한다.



Phase 1



Phase 2

Safety check 1 : Designated 초기화는 슈퍼클래스 초기화에 위임하기 전에 반드시 자신의 모든 속성이 초기화되는지 확인한다.(super.init 호출전에 자신의 속성 초기화)

Safety check 2 : Designated 초기화는 상속받은 속성에 값을 할당하기 전에 슈퍼클래스 초기화에 위임을 해야한다(super.init 호출후에 상속 속성 초기화. 그렇지 않으면 서브클래스에서 덮어쓰는 위험성)

Safety check 3 : Convenience 초기화는 반드시 어떤 속성에든 값을 할당하기 전에 다른 초기화에 위임해야 한다(self.init 이후 초기화. 그렇지 않으면 Designated 초기화에 의해 덮어쓰일 위험성)

Safety check 4 : 첫번째 단계 초기화가 끝나기 전에는모든 인스턴스 메소드를 호출할수 없고, 모든 인스턴스 속성 값을 읽을수 없고, self 값도 참조할 수 없다(슈퍼클래스의 Designate 초기화가 끝난 이후 허용 = 1단계가 완료된 후 허용)

22) Initialization (8/11)

Class Inheritance and Initialization(Cont.)

■ **Automatic Initializer Inheritance**

서브클래스는 **기본적으로** 슈퍼클래스의 초기화를 상속하지 않는다(Objective-C와의 차이).

슈퍼클래스의 지정 초기화를 서브클래스에서 제공하려면 동일하게 직접 구현하고 **override** 키워드 사용한다.

슈퍼클래스의 지정 초기화와 서브클래스의 편의 초기화가 동일한 매개변수이면 override 를 써야 한다

서브클래스에서 슈퍼클래스의 편의 초기화를 재정의 하면, 슈퍼클래스의 편의 초기화가 호출되지 않는다(슈퍼클래스의 편의 초기화에 의해 덮어써질 위험성) -> **Don't override**

- 지정 생성자 → 지정 생성자(override 가능)
- 지정 생성자 → 편의 생성자(override 가능)
- 편의 생성자 → 편의 생성자(override 불가능, 실질적으로 컴파일 에러)

그러나 특정 조건을 충족하면 override나 구현 노력 없이도 자동으로 초기화는 상속된다.

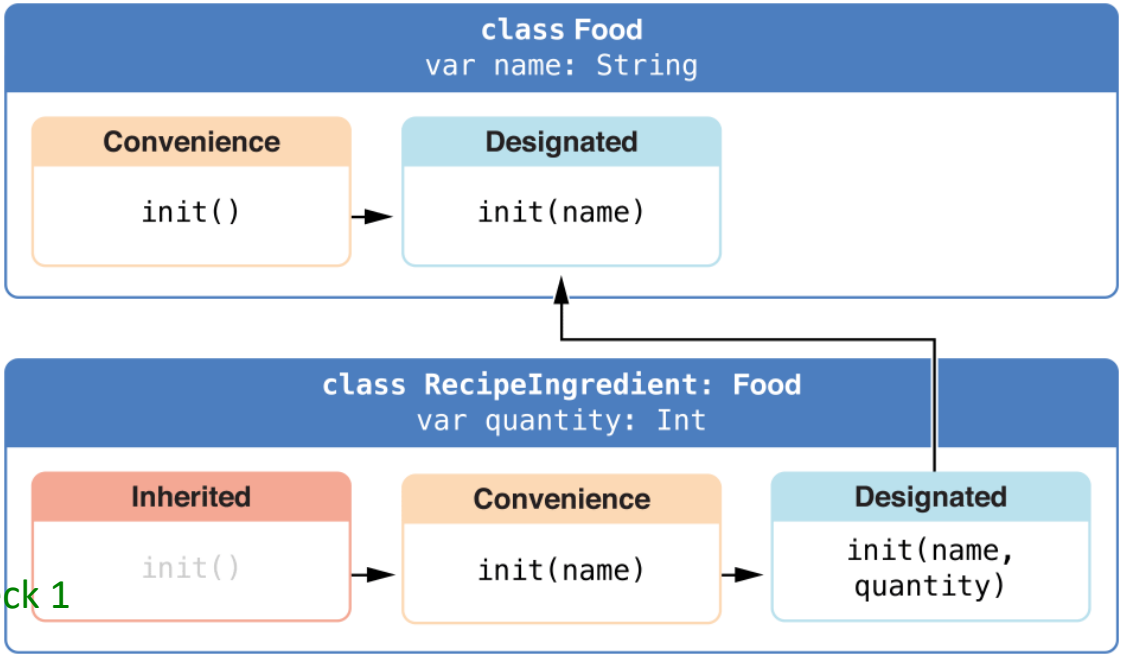
Rule 1 : 서브클래스가 Designated 초기화를 전혀 정의하지 않았으면, 슈퍼클래스의 모든 Designated 초기화가 자동으로 상속

Rule 2 : 서브클래스가 슈퍼클래스의 모든 Designated 초기화를 구현하거나(Rule 1 포함) 슈커플래스의 사용자 정의 생성자를 모두 구현을 하였을때(?), 슈퍼클래스의 모든 Convenience 초기화가 자동으로 상속

22) Initialization (9/11)

Class Inheritance and Initialization(Cont.)

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: “[Unnamed]")
    }
}

class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity //Safety Check 1
        super.init(name: name)
    }
    □ □ □ convenience init(name: String) {  // 1)구현하지 않으면 let rif = RecepeIngredient() 허용?
        self.init(name: name, quantity: 1)      // 2) Automatic…Rule 2?
    }
}
```



```
let namedMeat = Food(name: "Bacon")
let mysteryMeat = Food()
// 아래에서 name? and quantity?
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

```
//아래 클래스의 생성자는?
class ShoppingListItem: RecipeIngredient {
    var purchased = false
}
```

22) Initialization (10/11)

■ Failable Initializers

적절하지 않은 초기화 매개변수 값이나, 필요한 외부 리소스가 없을때 유용.
실패가능(init)과 실패불가능(init?) 초기화 2개를 같은 이름과 같은 매개변수 타입으로 정의불가.
실패할 경우 **nil을 리턴**

```
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 - \(giraffe.species)")
}

let anonymousCreature = Animal(species: "")
if anonymousCreature == nil {
    print("The anonymous creature couldn't be initialized")
}
```

22) Initialization (11/11)

■ Required Initializers

반드시 초기화를 상속 구현

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}

class SomeSubclass: SomeClass {
    required init() {   //You don't write the override modifier
        // subclass implementation
        // of the required initializer goes here
    }
}
```

```
class MyView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        //…
    }

    required init?(coder: NSCoder) {
        //super.init(coder: coder)
        //@IBOutlet을 통한 View접근 불가
    }

    override func awakeFromNib() {
        super.awakeFromNib()
        //@IBOutlet 완료 이후 호출됨
    }
}
```

■ Setting a Default Property Value with a Closure or Function

```
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
    }()
}
```