Swift 3

문법

- 엔트리 포인트로 사용되는 main함수가 없다.

```
Import UIKit

@UIapplicationMain
Class AppDelegate : UIResponder, UIApplicationDelegate
```

@UIapplicationMain 어노테이션을 사용하여 앱 시작하는 객체를 지정

변수와 상수

- 변수를 선언할 때 : var + 변수명
- 상수를 선언할 때 : let + 상수형

//상수는 한번 초기화하면 바꿀 수 없다.

- var age : Int //명시적 선언, 타입 어노테이션
- var year = 1999 // 선언과 동시에 초기화
- let welcomMessage = "안녕하세요"

자료형

- 정수형
- Int8 : 8bit
- Int16 : 16bit
- Int32 : 32bit
- Int64 : 64bit

- Ex))
- var num : Int
- //int로 선언하면 cpu의 크기(64bit)로 할당

자료형

- Unsigned 정수형
- UInt8 : 8bit
- UInt16 : 16bit
- UInt32 : 32bit
- UInt64 : 64bit

- Ex))
- var num : UInt
- num = 333333
- UInt.max // 18446744073709551615...
- UInt.min // 0

자료형

- 실수형 Double/Float

- Float32 / Float64(Double)

- Ex))
- var num : Double

- num = 32.123

자료형

- Bool
- True / False

- Ex))
- var num : Bool

- num = true

- if num {
- 	print(1)
- }
- // 결과 1이 출력

자료형

- String

- Ex))
- var str : String

- str = "하이" // 오직 큰 따옴표만 허용

- print(str)

자료형

- Character

- Ex))
- var char : Character

- char = "s" //오직 큰 따옴표만 허용

타입 추론

- var str1 = "hello" // str
- var str2 = "h" // char? str?

- str2 = "me" // ???

- 타입 어노테이션을 통한 모호함 해결
- var str2 : String = "h"

변수 결합

- var str = "나의 키는"
- var height = 185
-

- var print_str = str + String(height) // 메서드를 사용
- var print_str = "₩(str) ₩(height)" // 문자열 템플릿을 사용
-

- print(print_str)

문자열 템플릿

- Ex))
- var a = 1
- var b = 3
- var c = 100

- var str = "₩(a+b+c)"
- // str = "104"

연산자

- 단항 연산자 [-] // 부호 switch
- 이항 연산자 [+-*/%]

- Ex))
- var a : Int

- a = 1+2
- a = 1 + 2
- a = 1+ 2 // error

//띄어쓰기는 연산자 좌우로 같아야한다.

연산자

- 비교 연산자 [< > != ==]

- 논리 연산자 [! && ||]

- 범위 연산자 1...n
- Ex))
- for a in 1...5 {
    print(a)
}

연산자

- 반 닫힌 범위 연산자 1..<n
- for a in 1..<5 {
    print(a)
}
// 결과 : 1~4까지 출력
// 파이썬의 range() 제네레이터와 흡사

연산자

- 대입 연산자
- [= += -= *= /= %= <<= >>= &= ^= |=]

- 증감 연산자
- [++ --] // 현재는 사용안함.. 이유 : c언어와 유사(?)

코드 진행

- var i = 1; for i in 1...10{print(i)}
- //한 줄로 진행 -> 세미콜론 사용

- var i = 1
- for i in 1...10{
- 	print(i)
- }
- //여러줄로 진행

반복문

For 루프상수 in 순회대상{
    실행 구문
}

Ex))
var i = 1
for i in 1...10{
    print(i)
}
//순회 대상으로는 배열, 딕셔너리, 집합, 범위 데이터, 문자열

루프 상수 생략

- for _ in 1...5 {
- 	print(1)
- }
- // 루프 안에서 상수가 필요 없을 경우 생략 가능

반복문

- While 조건식 {
  실행할 구문
- }

- Ex))
- var n = 0

- while n<100 {
- n += 1
- print(1)
- }

반복문

- repeat ~ while 구문
- -> do ~ while 구문과 같음

- Ex))
- var n = 1

- repeat {
- 	n *= 2
- }

- while n<1000

- print(n) // 1024 출력

조건문

- If 조건식{
  실행 구문
- }

- Ex))
- var age = 26
- let adult = 19

- if age > adult {
- print("adult!")
- }

조건문

- If ~ else if ~ else 구문

- Ex))
- if age < adult {
- 	print("kid!")
- }
- else if age < old {
- 	print("adult!")
- }
- else{
- 	print("old!")
- }

조건문

- Guard ~else 구문

- var base : Int

- func divide(base : Int)
- {
- guard base != 0 else{
- print("연산할 수 없습니다.");
- return // 반드시 필요
- }
- }

#available 구문

- Api의 사용 여부 체크

- if #available(iOS 9, OSX 10.10, watchOS 1, *){
- //ios 9,osx 10.10, watchOS 1 용 api 사용
- }
- else
- {
- //api 처리 실패시
- }

조건문

- Switch 비교대상 {
  Case 비교 대상1 :
  실행 구문
  Case 비교 대상2 :
  실행 구문
  Default :
  실행 구문
- }

제어 전달문

- Break : break + 레이블
- Continue : continue + 레이블
- Fallthrogh : case 구문 넘기기
- Return : 함수 반환값 return

집단 자료형

- 배열

- Ex))
- var arr = ["a","b", "c"]

- var arr : [String]
- arr = ["a", "b", "c"]

- var arr = [String](repeating : "None", count : 3)
- //배열 아이템과 갯수 초기화

집단 자료형

- 배열의 동적인 추가

- Cities.append("Seoul")
- Cities.append("New York")
- Cities.insert("Tokyo", at : 1)
- Cities.append(contentsOf: ["Dubai", "Sydney"])

집단 자료형

- 집합 : 중복이 없고 순서도 없음.
- 해시 연산이 가능해야함.
- -> 해시 연산 : 고정 길이의 데이터 크기로 변환해주는 연산
- Ex) 10으로 나누는 알고리즘.

Ex)

```
var set = Set<Int>()
set = [1,2,3,4]
```

집단 자료형

- 집합 연산
- Intersection(_:) // 교집합
- SymmetricDifference(_:) // 합집합-교집합
- Union(_:) // 합집합
- Subtract(_:) // 차집합

집단 자료형

- 튜플
- 초기화 되면 변경 및 추가 불가능

- Ex))
- var tup = (1,2,3,4)

- var tup : (Int, Int)
- tup = (1,2)

집단 자료형

- 딕셔너리

- var dict = Dictionary<Int, Int>()
- var dict1 = [String : Int]()
- var dict2 = ["me" : "Moon"]

딕셔너리 항목 변경

```
var dict = Dictionary<Int, Int>()


dict = [1:2, 3:4, 5:6]


dict.updateValue(10, forKey: 1)
//인자1 값, 인자2 변경할 키


print(dict)
```

옵셔널

- 스위프트가 잠재적 오류를 다루는 방법

- 값을 처리하는 과정에서 오류가 발생할 가능성이 있는 값을
- '옵셔널'이라는 타입으로 감싼 후 반환

옵셔널

- Ex))
- "123"문자열을 숫자 123으로 변경
- Int("123") --> 123
- Int("asdf") = ??? Error발생

- 반환값으로 nill값을 반환함

nil

- Null과 비슷한 표현으로, 값이 없다는 것을 표현하기 위해 사용
- 값으로 사용되지 않고, 그 자체로 값이 잘못 되었음을 나타냄

- Ex))
- if nil {
- 	print(1)
- } //error

- Var a = nil // error
- Nil은 값을 가질 수 없으므로 null과는 다르다.(대입이나 비교 불가)

옵셔널

- 스위프트는 오류가 발생하면 오류를 출력하지 않고,
- 옵셔널의 nil값을 반환한다.
- -> 프로그램이 중단되는 경우 배제를 위함.

- 옵셔널 -- nil
- 　　-- nil이 아닌 값

옵셔널 타입 선언

- 타입 뒤에 '?'를 붙여서 선언하면 옵셔널 타입으로 선언된다.

- var a : Int?
- var arr : [String]?
- arr = ["asdf"] / 배열에 값을 할당
-

- print(a) // nil 출력
- print(arr) // optional(["asdf"]) 출력

옵셔널 값 처리

- 옵셔널 값은 대입은 자유로우나 결합이나 사칙연산 불가

- var a : Int?
- var b : Int?

- a=1
- b=2

- a+b // error

옵셔널 강제 해제

- 값 뒤에 '!'를 붙여서 해제

- var a : Int?
- var b : Int?
- var c : Int

- a=1
- b=2

- c = a! + b!

옵셔널

- 옵셔널을 사용할 때 주의점

- a!= 1
- -> a의 옵셔널을 해제 후 1을 대입
- -> a와 1이 같지 않다.

함수

- Func 함수 이름(매개변수 : 타입) -> 반환 타입{
  실행 내용
  Return 반환값
- }
- Ex))
- func sum(a : Int, b : Int)-> Int {
- return (a+b)
- }

- var c : Int
- c = sum(a: 1,b: 2) // 매개 변수 이름을 반드시 지정해주어야 함
- print("합은 ₩(c)")

함수의 반환 값(튜플) 참조

- func char_info()-> (String, Int, Int) {
- var name = "Moon"
- var age = 26
- var height = 176

- return (name, age, hight)
- //튜플로 반환
- }

- char_info().0
- char_info().1
- char_info().2

typealias

- 튜플 타입이 여러 곳에 쓰일 때

- Ex))
- typealias my_info = (String, Int, Int)

- func char_info()-> my_info {
- var name = "Moon"
- var age = 26
- var height = 176
-
- return (name, age, height)
- //튜플로 반환
- }

함수 선언

- func info(A name : String,B age : Int){
- var Name = name
- var Age = age
-
- print("₩(Name)(₩(Age))")
- }

- info(A : "Moon", B : 26)
- //내부 매개변수, 외부 매개변수

함수 선언

```
func sum(val:Int...)->Int{
    var total = 0
    for r in val{
        total += r
    }

    return total
}

print(sum(val :1,2,3))
//가변인자의 사용
```

함수 선언

- func sum(default_int: Int = 1, val:Int)->Int{
- var total = 0
- total = default_int + val
-
- return total
- }

- print(sum(val : 1))
- //함수의 기본값 사용

함수의 내부 값 참조

- func foo(param:inout Int) -> Int{
- param += 1
- return param
- }
- //inout int로 선언
- var count = 30

- foo(param : &count) // 포인터값을 전달

일급 함수

- 일급 함수의 특성
- 1. 객체가 런타임에도 생성 가능
- 2. 인자 값으로 객체를 전달 가능
- 3. 반환 값으로 객체를 사용 가능
- 4. 변수나 데이터 구조 안에 저장 가능
- 5. 할당에 사용된 이름과 관계 없이 고유한 구별 가능

일급 함수

- func foo(base : Int)-> String{
- var Sum = 2
-
- Sum = base + Sum
-
- return "₩(Sum)"
- }

- var fn2 = foo
  //변수에 대입할 수 있음.
- fn2(10)

일급 함수

- Func boo(a : Int) -> String {
  ~~~
  Return "aaa"
- }
- Func boo(a : Int, b : String ) -> String{
  ~~~
  Return "bbb"
- }

- Var func1 = boo // error

일급 함수

- 방법 1 : var t1 : (Int, String) -> String = boo
- // 변수의 형태를 지정해준다.
- 방법 2 : var t2 = boo(a:b:)
- // 함수의 식별자를 이용한다.

일급 함수

- func boo(a : Int, b : Int)->String{
- 	var sum = 0
- 	sum = a+b
- 	return "sum : ₩(sum)"
- }

- func boo(a : Int)->String{
- 	var sum = 0
- 	sum = sum+a
- 	return "sum : ₩(sum)"
- }

- var func1 = boo(a:b:)
- var func2 : (Int)->String = boo

일급 함수

- 함수의 반환 타입으로 함수를 사용 가능

- func desc() -> String{
- 	return "desc func!"
- }

- func pass() -> () -> String{
  //()->String은 desc함수의 식별자
- 	return desc //반환형으로 함수를 사용!
- }

- let p = pass()
- p()

일급 함수

- 함수의 인자 값으로 함수를 사용할 수 있음

- func incr(param : Int) -> Int {
- return param + 1
- }

- func broker(base : Int, function fn : (Int) -> Int) -> Int {
- //내부함수 fn
  return fn(base)
- }

- broker(base:3, function:incr)

일급 함수

```
func success(){
    print("연산 성공!")
}

func fail(){
    print("연산 실패!")
}
func divide(base:Int, succ:()->Void, fa:()->Void)->Int{
    guard base != 0 else{
        fa()
        return 0
    }
    defer { //가장 마지막에 실행하는 구문
        succ()
    }
    return 100
}
divide(base:0, succ:success,fa:fail)
```

함수의 중첩

- 함수 내부에 다른 함수 선언 가능

- func outer(base : Int)->String{
- 	func inner(inc : Int)->String{
- 		return "₩(inc)를 반환합니다"
- 	}
-
- 	let result = inner(inc:base + 1)
- 	return result
- }

- print(outer(base : 3))

함수의 중첩

- func basic(param : Int) -> (Int) ->Int{ //함수를 반환하고 있음
- 	let value = param + 20
-
- 	func append(add : Int) -> Int{ //내부에 함수선언
- 		return value + add
- 	}
-
- 	return append //내부함수를 반환
- }

- let result = basic(param: 10) // basic함수의 반환 = append함수
- result(10) // append함수 호출?? value값은????

함수의 중첩

- value값은 함수 내부 변수이므로 함수 호출 후 소멸
- append함수는 내부함수 이므로 외부함수 소멸시 같이 소멸
- But,
- result라는 변수에 append함수가 저장되고
- result함수에 10인자를 넣어 줄 때, value값을 사용하므로
- '클로저'라는 객체가 사용되어 값을 유지..

클로저(중첩 함수)

- 클로저(중첩 함수)는 두 가지로 이루어진 객체이다.
- (내부 함수와 내부 함수가 만들어진 주변 환경)
- 클로저는 내부 함수가 외부함수의 지역변수나
- 상수를 참조할때 만들어진다.

- -> 클로저란 내부 함수와 내부 함수에 영향을 미치는 주변 환경을 모두 포함한 객체이다.

클로저

- 이전 예제에서,
- let result1 = basic(param: 10)
- let result2 = basic(param: 5)
- 라면

- Func append(add : Int) -> Int { //result1에 할당된 클로저
  Return 30 + add //value값은 param에 10을 넣은 30이 capture된다.
- }
- Func append(add : Int) -> Int { //result2에 할당된 클로저
  Return 25 + add // 25가 capture된다.
- }

클로저(넓은 개념)

- 클로저(Closures)는 코드상에서 사용될 수 있는 독립적인 블록

- 1. 전역 함수 // 이름o, 주변 환경에서 캡처할 값x
- 2. 중첩 함수 // 이름o, 주변 환경에서 캡처할 값o
- 3. 클로저 표현식 // 이름x, 주변 환경에서 값을 캡처할 수 있는 경량 문법

    -> 익명(Anonymous) 함수

클로저 표현식

- { (매개변수) -> 반환 타입 in 실행할 구문 }

- Ex))
- let f = { () -> ()in
- print("1")
- }

- f()

Sorting closure

- 스위프트의 표준라이브러리는 sorted function를 제공을 하는데,
- 이 함수는 제공한 sorting closure 에 기반하여 배열을 정렬
- //인자로 클로저를 받는다.
- var value = [9,1,2,6,5,7]

- value.sort(by:{
- (s1:Int, s2:Int) -> Bool in
- if s1 > s2 {
- return true
- }
- else{
- return false
- }
- })

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{
- (s1:Int, s2:Int) -> Bool in
  Return s1 > s2 // if문 간결화
- })

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{ (s1:Int, s2:Int) -> Bool in Return s1 > s2 })
- // 한줄로 표현

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{ (s1:Int, s2:Int) in Return s1 > s2 })
- // 반환형 생략 (리턴 값으로 추론 가능)

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{ s1, s2 in Return s1 > s2 })
- // 매개변수 타입과 괄호 생략 (입력값으로 추론 가능)

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{ Return $0 > $1 })
- // 내부 상수를 이용하여 매개변수 제거 및 in 키워드 제거

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by:{ $0 > $1 })
- // return 생략

클로저 표현식 간결화

- var value = [9,1,2,6,5,7]

- value.sort(by: >)
- //연산자 함수를 이용
- //연산자만으로 의미하는 바를 정확히 나타낼 수 있을때 사용

구조체와 클래스

- struct 구조체이름{
  구조체 내용
- }

- class 클래스이름{
  클래스 내용
- }

네이밍 방법

- 카멜 표기법

- 1. 구조체와 클래스 이름의 첫 글자는 대문자로, 나머지 글자는 소문자로 작성한다.
- 2. 2개 이상의 복합 단어는 단어별로 끊어 첫 글자는 대문자, 나머지는 소문자로 작성한다/
- 3. 이미 축약된 약어는 모두 대문자로 작성 가능하다.
- 4. 프로퍼티나 메소드를 선언할 때는 소문자로 시작한다.
- 5. 언더바로 단어를 연결하는 방식은 지양한다.

메소드와 프로퍼티

- 프로퍼티 : 구조체나 함수 내부에 선언된 변수/상수
- 메소드 : 구조체나 함수 내부에 선언된 함수

```
class A{
    var B //프로퍼티

    func C() -> String{ // 메소드
        return "hello"
    }
}
```

인스턴스

- 인스턴스 : 구조체나 클래스로 만들어 낸 객체

```
class a{
    var b = 11 //프로퍼티

    func c() -> String{
        return "hello"
    }
}

let d = a() //인스턴스 생성
print(d.b) // 프로퍼티 출력
print(d.c()) // 메소드 출력
```

인스턴스 속성 접근

```
class A {
    var B = 1

    func C() -> String {
        return "Hello"
    }
}
```

- var D = A() //인스턴스 생성
- var E = D.B // 인스턴스 프로퍼티 접근
- var F = D.C() // 인스턴스 메서드 접근

인스턴스 단계적 접근

- struct Struct {
- 	var some = 1
- }

- class A {
- 	var B = 1 // 프로퍼티 할당
- 	var C = Struct() // 구조체 할당
-
- }

- var E = A() // 인스턴스 할당
- var F = E.C.some // 1

인스턴스 초기화

- class A {
- var B : Int
- }

- var C = A()

- print(C.B)// 에러 -> 값이 없음
- // 선언할 때 var B : Int? 로 옵셔널 선언해주면 nil값으로 반환

인스턴스 초기화

- 모든 프로퍼티를 정의할 때 초기값을 주던가, 아니면 옵셔널 타입으로 선언한다.
- 인스턴스를 생성할 때에는 클래스명 뒤에 ()를 붙여준다.

구조체의 값 전달 방식

- struct A{
- 	var height = 100
- }
- var B = A()//초기화를 통한 인스턴스 생성
- var C = B // B의 인스턴스 복사

- print(B.height)//100

- B.height = 200
- print(B.height) //200 -> 200을 대입해주어서
- print(C.height) //100 -> 이전값인 100이 저장

클래스의 값 전달 방식

- class A{
- 	var height = 100
- }
- var B = A()//초기화를 통한 인스턴스 생성
- var C = B // B의 인스턴스 복사

- print(B.height)//100

- B.height = 200
- print(B.height) //200 -> 200을 대입해주어서
- print(C.height) //200 -> 같은 포인터 값을 참조하기때문

함수의 인자, 인스턴스

- class A{
- 	var B = 1234
- }

- let C = A()

- func some(D : A) { // 클래스 타입의 인자
- 	print(D.B)
- }

- some(D : C)
- //함수의 인자로 인스턴스를 넘겨줄 수 있다.

ARC(Auto Reference Counter)

- 인스턴스 선언에 따른 메모리 이슈(낭비) 방지하기 위함
- Object-C 에서는 사용자가 판단하여 인스턴스를 해제
- ARC는 컴파일러가 인스턴스를 참조하는 곳이 모두 몇 군데인지 자동으로 카운트 해주는 객체.
- 인스턴스가 사용되면 카운트를 1씩 줄여가며 0이 되면 자동으로 해제.

인스턴스의 비교

```
class A{
}

let B = A()
let C = B

if (B === C) {
    print("Same instance")
}
else if (B !== C) {
    print("Not Same")
}
// 동일 인스턴스 비교 ===
// 다른 인스턴스 비교 !==
```

지연 저장 프로퍼티

- 인스턴스가 선언될 때가 아닌 프로퍼티가 호출되는 시점에서 생성.
- class OnCreate {
- init(){ // 초기화 메서드 뒤에 나옴.
- print("Create")
- }
- }
- class LazyTest {
- var base = 0
- lazy var late = OnCreate()
- //지연 저장 프로퍼티
- init() {
- print("Lazy Test")
- }
- }
- let A = LazyTest() // Lazy test 출력
- A.late // Create 출력

클로저를 이용한 프로퍼티 초기화

```
class A{
    var B : String = {
        return "Hello"
    }()
}

var C = A()
C.B //hello 출력
C.B //출력 안함
//인스턴스가 초기화 되고 최초 한번만 실행
```

연산 프로퍼티 get

```
import Foundation
struct MyInfo {
    var birth : Int!//태어난 년도
    var thisYear : Int! { //올해년도 구함
        get { // 연산을 통해 thisYear에 값을 대입
            let df = DateFormatter()
            df.dateFormat = "yyyy"
            return Int(df.string(from: Date()))
        }
    }
    var age : Int{ //올해년도 - 태어난년도 +1 = 나이
        get {
            return (self.thisYear - self.birth) + 1
        }
    }
}
```

간단한 get/set 구문

```
class A {
    var B = 123
    var C : Int {
        get {
            return 1 // 무조건 1반환
        }
        set(get_num) { //대입값이 매개변수로 넘어온다.
            print(get_num)
        }
    }
}
var D = A() // 인스턴스 생성
D.C = 2 // set함수 실행
```

Read-only property

- class A {
- 	var B = 123
- 	var C : Int {
- 		get { // get함수만 존재하고 set함수는 존재하지 않음. -> read only..
- 			return 1
- 		}
-
- 	}
- }

- var D = A() // 인스턴스 생성
- D.C = 2 // 프로퍼티에 값 대입 -> set함수 실행 -> 존재하지않음 -> 에러

프로퍼티 옵저버

```
class A {
    var B : Int? {
        willSet(some){ //프로퍼티 값 할당 전 실행
            print(some!)
        }
        didSet{ // 값 할당 후 실행
            print("₩(B! + 1)")
        }
    }
}
```

타입 프로퍼티

- 인스턴스에 관계 없이 클래스나 구조체 자체에 값을 저장
- -> 글로벌 변수처럼 사용됨.
- class A {
- static var sFoo = "origin"
  //class var sFoo = "origin"과 동일
- }

- A.sFoo = "new" // 클래스 프로퍼티 변경

- print(A.sFoo) // new 출력

- var B = A() // 인스턴스 생성
- print(B.sFoo) // 인스턴스를 통한 프로퍼티 참조 ->에러

인스턴스 메소드

- 인스턴스에 소속된 함수
- class A {
- func B() -> () {
- print("method")
- }
- }

- var C = A()
- C.B()

메소드 self

```
class A {
    var count = 0

    func increase() -> () {
        self.count = self.count+1
    }
    func increaseby(num : Int) -> () {
        self.count = self.count + num
    }
    func reset() -> () {
        count = 0 // self를 생략해도 컴파일러는 오류를 발생하지 않음.
    }
}
```

타입 메소드

```
class A {
    class func B() { //static으로 해도 가능
        print("123");
    }
}
```

```
var C = A()
C.B() // 인스턴스로 참조 : 에러
A.B() // 123 출력
```

상속

- 한 클래스가 다른 클래스에서 정의된 프로퍼티나 메소드를 물려받아 사용하는 것
- class A {
- var B = 13
- func C() {
- print("123");
- }
- }

- class D : A { // A클래스를 상속받음
- var E = 11
- }

- var some = D()

- some.E
- some.B
- some.C()

오버라이딩

- 부모 클래스로 상속받은 프로퍼티나 메소드를 필요에 의해 재구성
- class car {
- var engineLevel = 0
- var speed : Int {
- return engineLevel * 50
- }
- }
- class bongo : car{
- override var speed : Int { // override 구문을 사용
- return engineLevel * 40 // 리턴 타입이나 매개변수 타입은 일정해야 함
- }
- }
- var A = car()
- var B = bongo()
- B.engineLevel = 1
- B.speed // 40 출력
- A.engineLevel = 1
- A.speed // 50 출력

오버라이딩을 막는 방법

```
class A{
    var B = 10

    final func C() -> (){ // final 키워드를 사용
        print("Hello")
    }
}

class D : A{// A를 상속받음
    override func C() -> (){ // error!
        print("Hello!")
    }
}
```

타입 캐스팅

- class Vehicle {
- }

- class Car : Vehicle { // 상속 받음
- }

- var A : Car // Car 클래스 타입 변수

- A = Vehicle() // 상위 클래스 할당 -> error

타입 캐스팅

- class Vehicle {
- 	func print_message() -> () {
- 		print("Vehicle Class")
- 	}
- }
- class Car : Vehicle { // 상속 받음
- 	func print_message1() -> () {
- 		print("Car Class")
- 	}
- }
- var A : Vehicle // Vehicle 클래스 타입 변수
- A = Car() // 하위 클래스 할당 -> Vehicle 클래스로 캐스팅
- A.print_message()
- A.print_message1() // Vehicle class 이므로 error

타입 비교 연산

- 타입 비교 연산자 is
- 1. 연산자 왼쪽 인스턴스 타입이 오른쪽과 일치 : true
- 2. 연산자 왼쪽 인스턴스 타입이 오른쪽의 하위 클래스 : true
- 3. 그 외 : false

- class Vehicle {
- }
- class Car : Vehicle { // 상속 받음
- }
- var car = Car()
- var vehicle = Vehicle()
- car is Vehicle // True
- vehicle is Car // False

타입 캐스팅 연산

```
class Vehicle {
    func printSome() -> () {
        print("Vehicle")
    }
}
class Car : Vehicle{ //상속 받음
    func printSome1() -> () {
        print("Car")
    }
}
var anyCar : Vehicle = Car()
// Car 클래스를 받지만 Vehicle 클래스의
// 프로퍼티나 메소드만 호출 가능
anyCar.printSome1() // error!
// Car 클래스의 값을 호출하려면 형변환이 필요
```

타입 캐스팅 연산

- 업 캐스팅
- 객체 as 변환할 타입

- 다운 캐스팅
- 객체 as? 변환할 타입 (결과는 옵셔널)
- 객체 as! 변환할 타입 (결과는 일반)

타입 캐스팅 연산

```
class Vehicle {
    func printSome() -> () {
        print("Vehicle")
    }
}
class Car : Vehicle{ //상속 받음
    func printSome1() -> () {
        print("Car")
    }
}
var anyCar : Car = Car()
var anyVehicle = anyCar as Vehicle
anyVehicle.printSome1()
// 업캐스팅 되어서 printSome1함수는 사용 불가능
```

타입 캐스팅 연산

- class Car : Vehicle{ //상속 받음
- 	func printSome1() -> () {
- 		print("Car")
- 	}
- }
- class SUV : Car{
- 	func printSome2() -> () {
- 		print("SUV")
- 	}
- }
- var someCar : Car = SUV()
- var someSUV = someCar as? SUV
- someSUV?.printSome2()
- //다운 캐스팅 -> 자식 메소드 접근 가능

Any / AnyObject

- class A {
- }

- var num : Any = 123 // 변수
- var num2 : Any = A() // Any에 모든 객체, 자료형 대입 가능
- var obj : AnyObject = A() // 객체

- func B(param : AnyObject) -> AnyObject { //인자값 리턴값
- 	return param
- }
- B(param:A())

- // any는 초기화할 때 자료형이 정해짐.
- // 타입이 동적이어서 컴파일러가 에러를 검출하기 어려움.

초기화 구문

- struct A {
- 	var num : Int
- }
- var B = A(num : 123) // 구조체는 멤버 와이즈 초기화 가능

- class C {
- 	var num1 : Int
- }
- var D = C(num1 : 123) // 불가능

Init 초기화 메소드

```
class A {
    var B : Int
    var C : String

    init(num_B : Int, string_C : String){
        self.B = num_B
        self.C = string_C
    }
}
var D = A(num_B : 123, string_C : "hello")
//var D = A() 에러!

D.B
D.C
```

Init 초기화 메소드

```
class A {
    var B : Int
    var C : String

    init(B : Int = 0, C : String = ""){ // 초기화
        self.B = B
        self.C = C
    }
}
var D = A() // 가능
D.B
D.C
```

초기화 구문의 오버라이딩

```
class A {
    var B = 12
}

class C : A{ //A를 상속받고 있음.
    var D = 0
    override init (){ // A에 초기화 구문이 없어도
    //명시적으로 오버라이딩
        self.D = 123
    }
}
```

초기화 구문의 델리게이션

```
class A {
    var B = 0
    init(num_B : Int){
        self.B = num_B
    }
}
class C : A{
    var D = 0
    override init (num_B : Int){
        super.init(num_B: num_B)
        //슈퍼클래스를 이용하여 A의 init구문을 호출 -> delegation
        self.D = 123
    }
}
var E = C(num_B : 12)
```

옵셔널 체인

```
struct Info{
    var name : String?
    var man : Bool?
}

var boy : Info? = Info(name : "moon", man : true)

if boy != nil{
    if boy!.name != nil{
        print(boy!.name!) // 비로소 moon출력
    }
}
```

옵셔널 체인

```
struct Company{
    var ceo : Info?
    var companyName : String?
}
struct Info{
    var name : String?
    var man : Bool?
}
var msCompany : Company? = Company(ceo : Info(name:"MOON", man:true), companyName : "moonsCompany")
if msCompany != nil{
    if msCompany!.ceo != nil{
        if msCompany!.ceo!.name != nil{
            print(msCompany!.ceo!.name!) // moon 출력
        }
    }
}----------> if문이 반복되는 굉장히 불편한 작업
```

옵셔널 체인

- struct Company{
- 	var ceo : Info?
- 	var companyName : String?
- }
- struct Info{
- 	var name : String?
- 	var man : Bool?
- }
- var msCompany : Company? = Company(ceo : Info(name:"MOON", man:true), companyName : "moonsCompany")
- print(msCompany?.ceo?.name) // 간단하게 출력

옵셔널 체인과 옵셔널 강제해제

- MsCompany?.ceo?.name : 옵셔널 체인
- -> 중간에 속성이 하나라도 nil이면 출력은 nil

- MsCompany!.ceo!.name : 옵셔널 강제해제
- -> 중간에 속성이 하나라도 nil이면 에러

열거형

- enum 열거형 이름{
  - Case 멤버값1
  - Case 멤버값2
  - Case 멤버값3
- }
- Ex))
- enum direction {
- case north
- case south
- case east, west
- }
- let N = direction.north
- let S = direction.south
- let E = direction.east
- let W = direction.west

열거형 객체의 사용

- var directionToHead = Direction.west

- var directionToHead : Direction = Direction.west
- directionToHead = .east
- //타입 어노테이션을 해주면, 열거형 이름 생략가능

- var directionToHead : Direction= .east

열거형

- enum Rank : Int { //타입 어노테이션
- 	case one = 1 // 시작할 정수값 지정
- 	case two, three, four, five // 나머지는 자동할당
- }

- Rank.one.rawValue // 속성값 참조는 rawValue
- Rank.two.rawValue // 2
- Rank.three.rawValue // 3

열거형

```
enum HTTPCode : Int {
    case OK = 200
    case NOT_MODIFY = 304
    case INCORRECT_PAGE = 404
    case SERVER_ERROR = 500
    var value : String {
        return "HTTPCode number is ₩(self.rawValue)"
    }
    func getDescription() -> String{
        switch self {
        case .OK :
            return "응답이 성공했습니다. 코드는 ₩(self.rawValue)입니다."
        case .NOT_MODIFY :
            return "변경된 내역이 없습니다. 코드는 ₩(self.rawValue)입니다."
        case .INCORRECT_PAGE :
            return "존재하지 않는 페이지입니다. 코드는 ₩(self.rawValue)입니다."
        case .SERVER_ERROR :
            return "서버 오류입니다. 코드는 ₩(self.rawValue)입니다."
        }
    }
}
```

익스텐션

- Extension <확장할 기존 객체> {
  //추가할 기능
- }
- Ex))
- extension Double {
- var km : Double { return self * 1_000.0}
- //언더바는 자릿수 구분을 위해 사용
- var m : Double { return self }
- var cm : Double { return self / 100.0 }
- }

- 2.km // 2000
- 5.5.cm // 0.055
- print("마라톤의 총 길이는 ₩(42.0.km + 195.m) 입니다.")

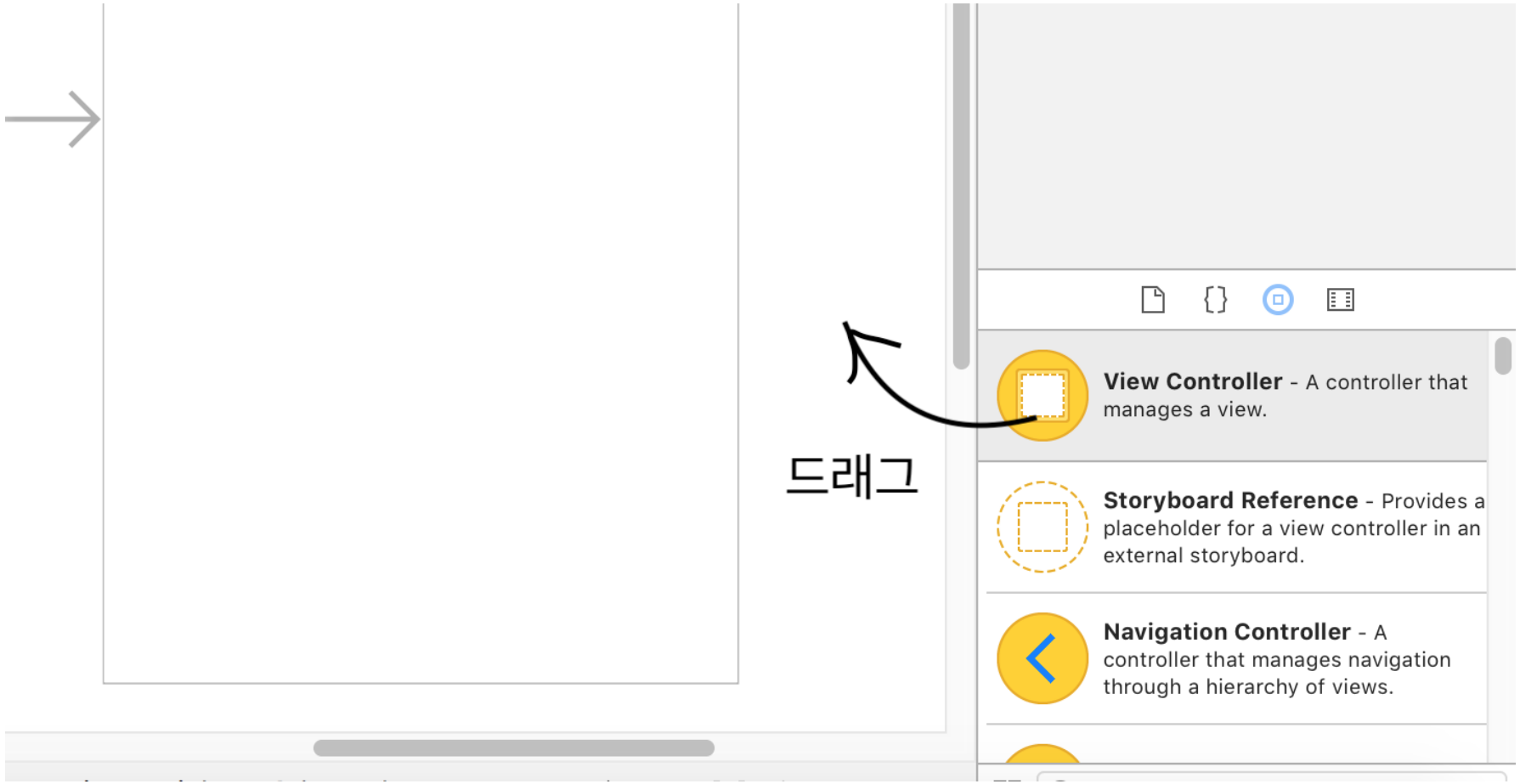
익스텐션과 메소드

- extension Int {
- 	func tell() -> (){
- 		print("This is Integer!")
- 	}
- }

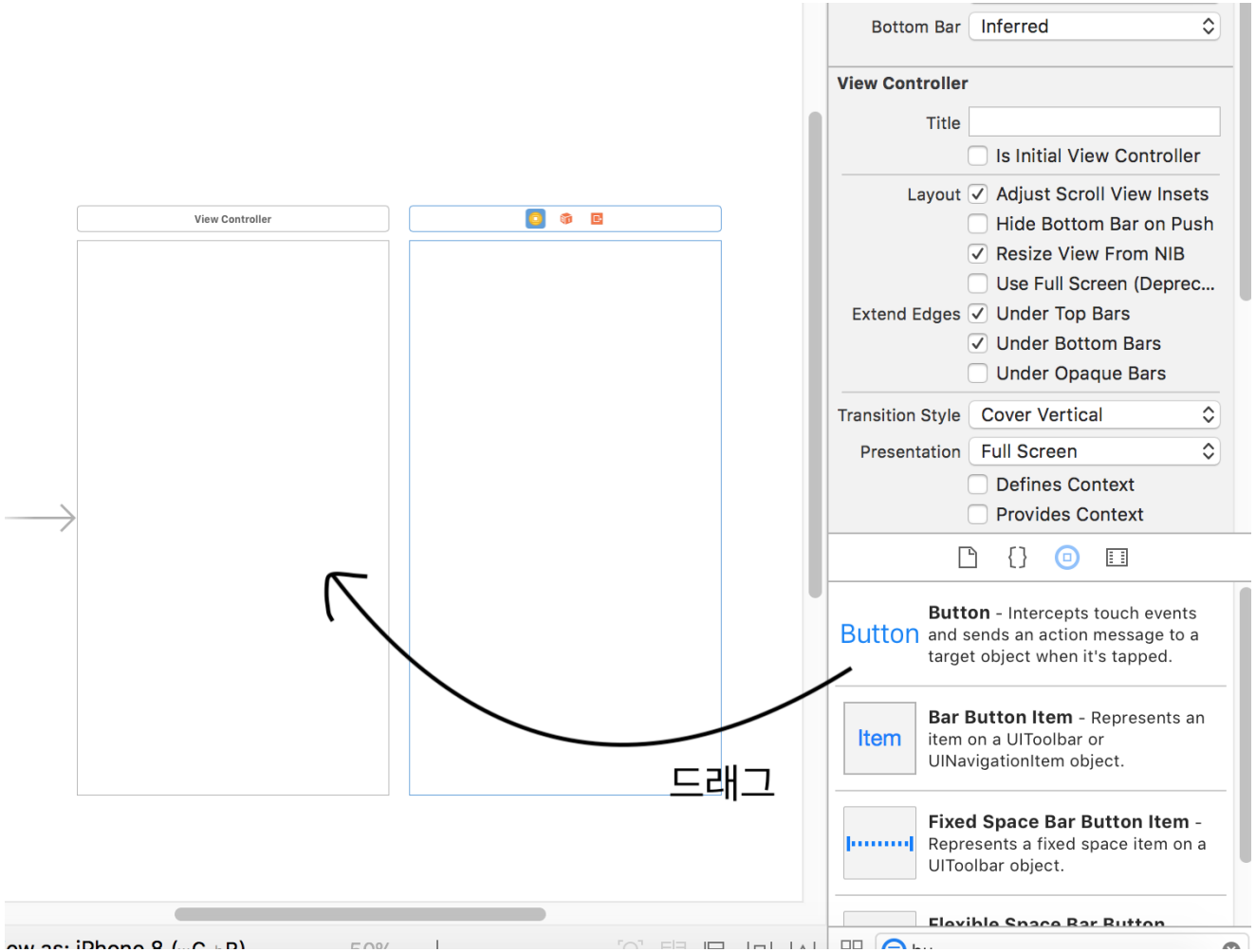
- var A = 123

- A.tell()
- //익스텐션에서 기존 객체에서 사용된 같은 메소드를 재정의 하는 것은 안된다.
- //-> 상속받아서 오버로딩으로만 가능.

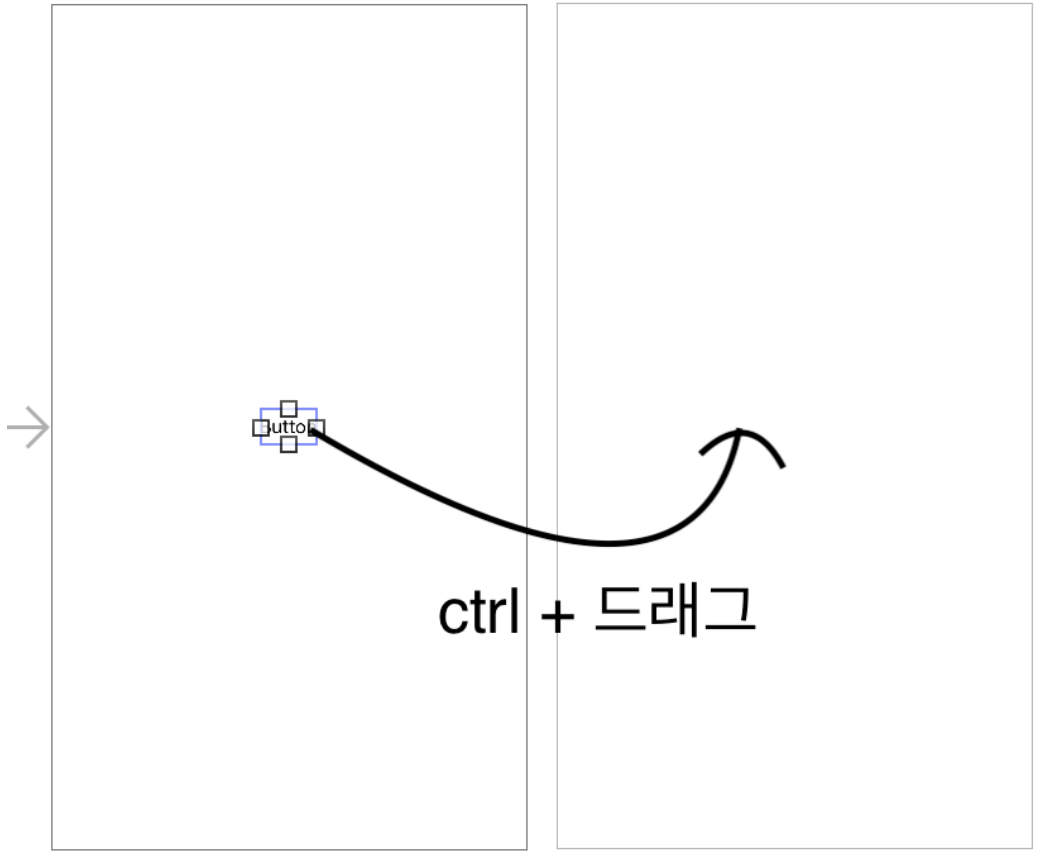
Button event를 이용한 View controller 전환



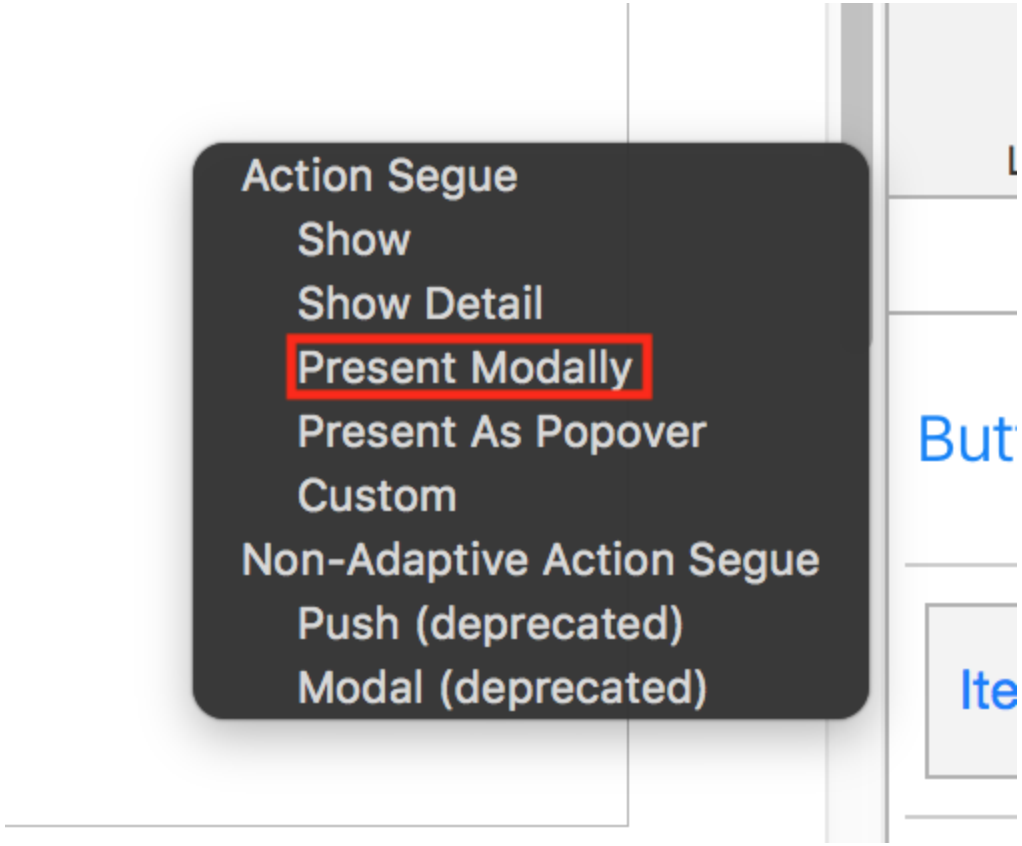
Button event를 이용한 View controller 전환



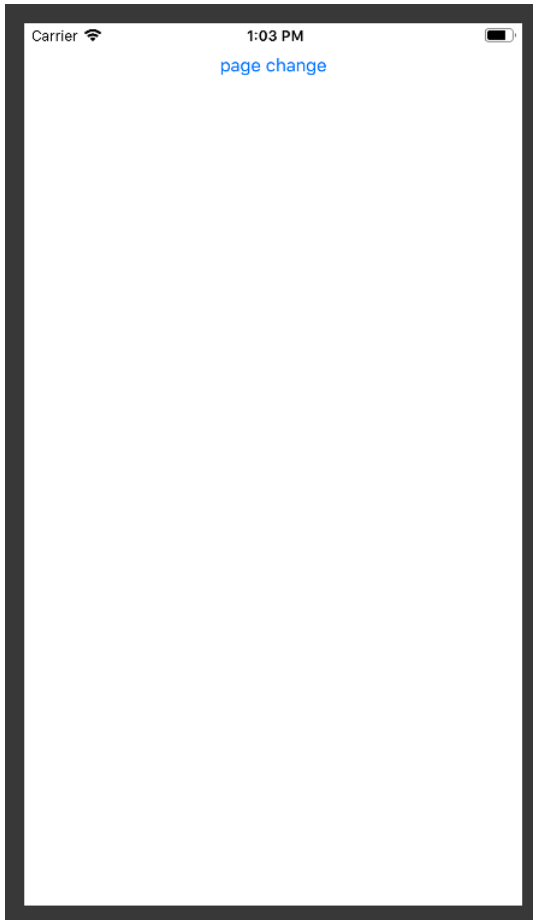
Button event를 이용한 View controller 전환



Button event를 이용한 View controller 전환



Button event를 이용한 View controller 전환



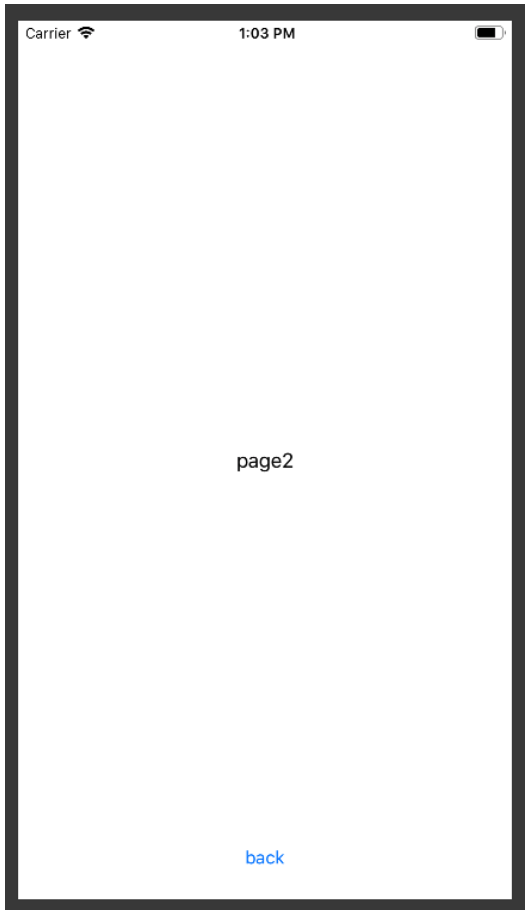


Table view 만들기

드래그를 통해 테이블뷰 생성

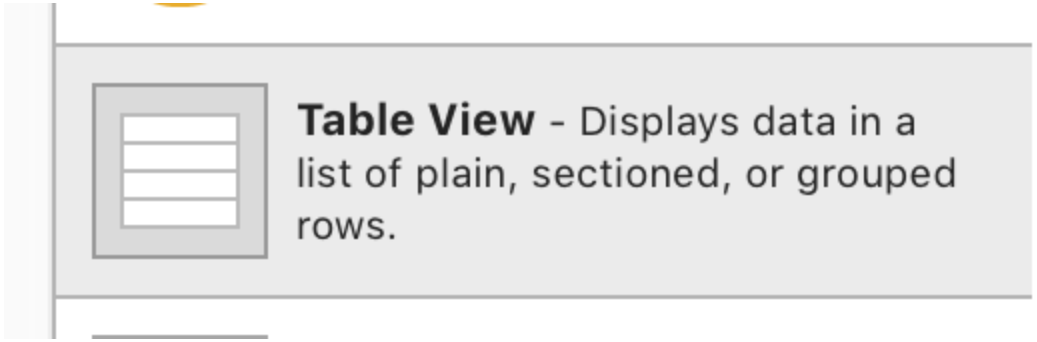


Table view 만들기

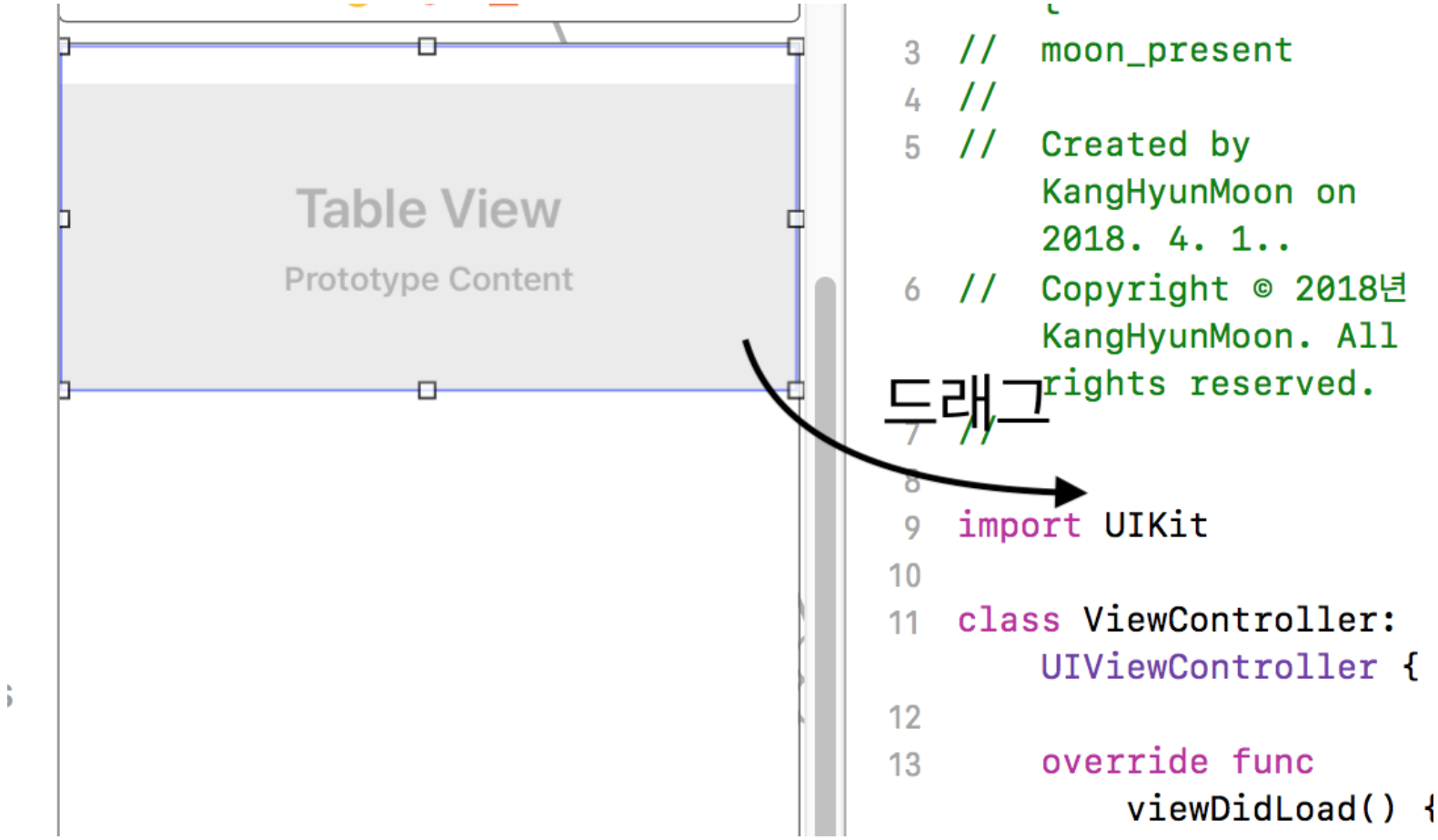


Table view 만들기

- 이름 tableView
- @IBOutlet weak var tableView: UITableView!
- 변수 tableView가 UITableView 클래스를 상속받는다.

Table view 만들기

```
override func viewDidLoad() {
        //뷰컨트롤러가 로드 되고 난 후 한번만 실행
        super.viewDidLoad()
        //슈퍼클래스의 didload deligate

        // Do any additional setup after loading the view, typically from a nib.
        tableView.delegate = self
        tableView.dataSource = self

        // 테두리 제거
        self.tableView?.tableFooterView = UIView()
    }
```

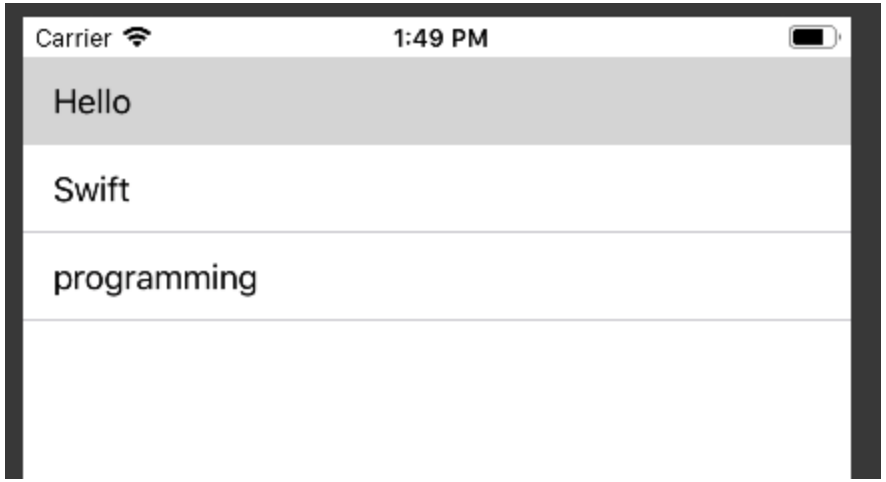
Table view 만들기

```
let titles = ["Hello", "Swift", "programming"]
// 테이블 행수 얻기 (tableView 구현 필수)
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return titles.count // 배열의 갯수 반환
}
// 셀 내용 변경하기 (tableView 구현 필수)
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell: UITableViewCell = UITableViewCell(style: UITableViewCellStyle.subtitle, reuseIdentifier: "Cell")
    cell.textLabel?.text = titles[indexPath.row]
    //셀의 텍스트라벨 설정
    return cell
}
```

Table view 만들기

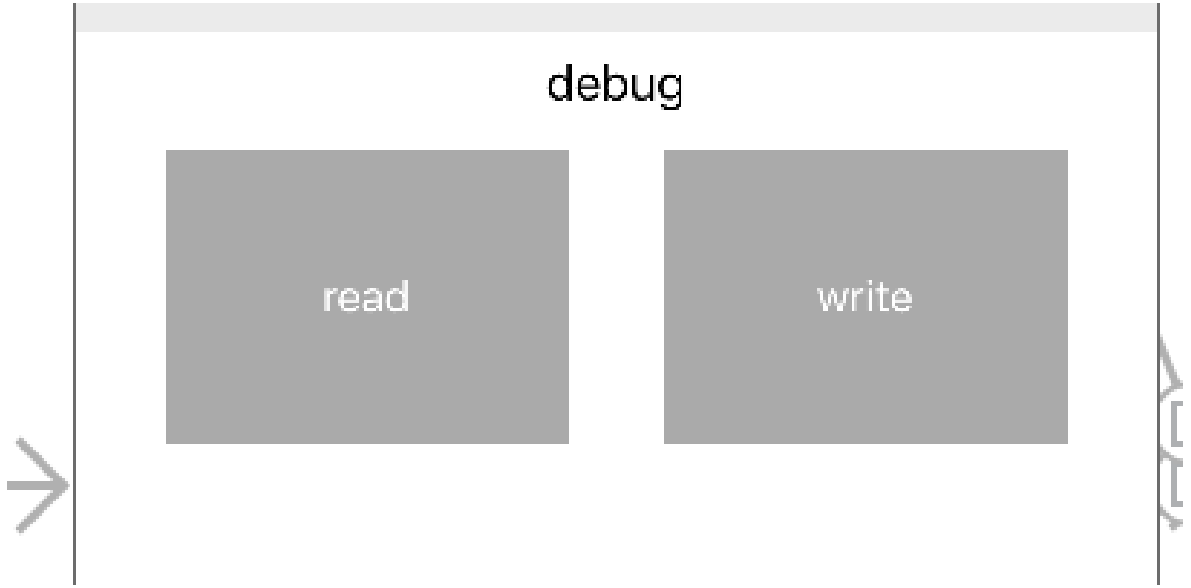
```
//왼쪽 공백 제거
    func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath) {
        if(self.tableView.responds(to: #selector(setter: UITableViewCell.separatorInset)))
        {
            self.tableView.separatorInset = UIEdgeInsetsMake(0.0, 0.0, 0.0, 0.0)
        }

        if(self.tableView.responds(to: #selector(setter: UIView.layoutMargins))){
            self.tableView.layoutMargins = UIEdgeInsetsMake(0.0, 0.0, 0.0, 0.0)
        }

        if(cell.responds(to: #selector(setter: UIView.layoutMargins))) {
            cell.layoutMargins = UIEdgeInsetsMake(0.0, 0.0, 0.0, 0.0)
        }
    }
```

Table view 만들기



"helloworld.txt" 만들기(파일 입출력)

- 다음과 같이 뷰를 꾸민다.



"helloworld.txt" 만들기(파일 입출력)

- @IBOutlet var debugLabel: UIView!
- //디버그 메시지를 보기 위한 아웃렛변수 선언
- @IBAction func fileRead(_ sender: UIButton) {
- }
- @IBAction func fileWrite(_ sender: UIButton) {
- }
- //버튼 두개의 동작을 위한 액션함수 선언
- @IBOutlet weak var textBox: UITextField!
- //데이터 입력받기 위한 텍스트 박스 설정

"helloworld.txt" 만들기(파일 입출력)

- @IBAction func fileWrite(_ sender: UIButton) {
- //write 버튼 누름
- let fileManager = FileManager()
- //filemanager 인스턴스 생성
- let documentsDirectory = fileManager.urls(for: .documentDirectory, in: .userDomainMask).first!
- //document 디렉토리 저장
- //urls(for:in:) 메소드를 통해 특정 경로에 접근

"helloworld.txt" 만들기(파일 입출력)

- urls(for:in:) 메소드는 SearchPathDirectory 와 SearchPathDomainMask를 파라미터로 받고 있는데,
- SearchPathDomainMask의 범위에서 SearchPathDirectory를 찾는 메소드이다.
- SearchPathDomainMask를 userDomainMask로 설정하면, /Users보다 위에 있는 디렉토리는 접근할 수 없다.

"helloworld.txt" 만들기(파일 입출력)

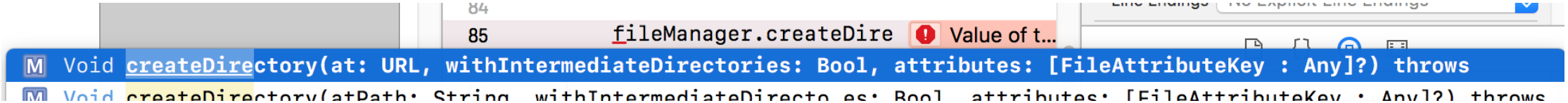
- let dataPath = documentsDirectory.appendingPathComponent("HelloWorld")
- //디렉토리 이름 어팬딩
- //appendingPathComponent(_:)를 사용하면 경로 URL에 추가적인 경로를 붙일 수 있다.
- Ex) **///Users/kanghyunmoon/Library/Developer/CoreSimulator/Devices/8C7BC643-687C-47CB-9051-606593BA115B/data/Containers/Data/Application/AE7C4216-AE85-4BF1-A205-0310A7E3F7E9/Documents/HelloWorld**

"helloworld.txt" 만들기(파일 입출력)

- do {
- 		// 디렉토리 생성
- 		try fileManager.createDirectory(atPath: dataPath.path, withIntermediateDirectories: false, attributes: nil)
-
- 	} catch let error as NSError {
- 		print("Error creating directory: ₩(error.localizedDescription)")
- 		//에러 잡기
- 	}

"helloworld.txt" 만들기(파일 입출력)

- Try~catch 구문 사용 이유



- Throws - > 에러 발생 가능
- 에러검출 코드를 작성.

"helloworld.txt" 만들기(파일 입출력)

```
do {
        var filePath = dataPath.appendingPathComponent("Hello.txt")
       // 파일 이름을 기존의 경로에 추가
        let text = textBox.text

        do {
            try text.write(to: filePath, atomically: false, encoding: .utf8)
        }
    } catch let error as NSError {
        print("Error writing File : ₩(error.localizedDescription)")
    }
  }
}
```

"helloworld.txt" 만들기(파일 입출력)

- @IBAction func fileRead(_ sender: UIButton) {
- let fileManager = FileManager()
- //filemanager 인스턴스 생성
- let documentsDirectory = fileManager.urls(for: .documentDirectory, in: .userDomainMask).first!
- //document 디렉토리 저장
- let dataPath = documentsDirectory.appendingPathComponent("HelloWorld")
- //디렉토리 이름 어팬딩
- print(dataPath)
- var filePath1 = dataPath.appendingPathComponent("Hello.txt")
- print("writing complete")
- let text2 = try? String(contentsOf: filePath1, encoding: .utf8)
- debugLabel.text = text2

"helloworld.txt" 만들기(파일 입출력)

TCP/IP 통신 (PC : Server)

//서버소켓을 다음과 같이 만들어 주었다.

```
int main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET hServSock;
    SOCKET hClntSock;
    SOCKADDR_IN servAddr;
    SOCKADDR_IN clntAddr;
    int szClntAddr;
    char message[BUFSIZE];
    int strLen;
```

TCP/IP 통신

```
if (argc != 2)
    {
        printf("Usage : %s <port>₩n", argv[0]);
        exit(1);
    }
    /* Load Winsock 2.2 DLL */
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
    {
        ErrorHandling((char *)"WSAStartup() error!");
    }
    hServSock = socket(PF_INET, SOCK_STREAM, 0); // 서버 소켓 생성
    if (hServSock == INVALID_SOCKET)
    {
        ErrorHandling((char *)"socket() error");
    }
```

TCP/IP 통신

- memset(&servAddr, 0, sizeof(servAddr));
- servAddr.sin_family = AF_INET;
- servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
- servAddr.sin_port = htons(atoi(argv[1]));

- printf("bind...₩n");
- if (bind(hServSock, (SOCKADDR*)&servAddr, sizeof(servAddr)) == SOCKET_ERROR)
- {
- ErrorHandling((char *)"bind() error"); // 소켓에 주소할당
- }
- printf("fin.₩n");

TCP/IP 통신

- printf("listen...₩n");
- if (listen(hServSock, 5) == SOCKET_ERROR)
- {
- ErrorHandling((char *)"listen() error"); // 연결 요청 대기
- }
- printf("fin.₩n");

- szClntAddr = sizeof(clntAddr);
- printf("accept...₩n");
- hClntSock = accept(hServSock, (SOCKADDR*)&clntAddr, &szClntAddr); // 연결 요청 수락
- if (hClntSock == INVALID_SOCKET)
- {
- ErrorHandling((char *)"accept() error");
- }

TCP/IP 통신

```
while ((strLen = recv(hClntSock, message, BUFSIZE, 0)) != 0) // EOF가 오지 않으면 실행
    {
        printf("recv....₩n");
        send(hClntSock, message, strLen, 0); // 데이터 전송
    }
    closesocket(hClntSock); // 연결 종료
    WSACleanup();
    return 0;
}
void ErrorHandling(char *message)
{
    fputs(message, stderr);
    fputc('₩n', stderr);
    exit(1);
}
```

TCP/IP 통신 (app : Client)

- Swift 내에서 클라이언트 부분
- class TcpSocket: NSObject, StreamDelegate {
- var host:String?
- var port:Int?
- var inputStream: InputStream?
- var outputStream: OutputStream?
- //변수들 선언
- func connect(host: String, port: Int) { //커넥트 매서드
- self.host = host //호스트 저장
- self.port = port //포트 저장

TCP/IP 통신

- Stream.getStreamsToHost(withName:host, port : port, inputStream: &inputStream, outputStream: &outputStream)
- //인자값으로 포인터값을 넘겨주고, 포트와 호스트값으로 스트림 얻음.
- if inputStream != nil && outputStream != nil {
- 	// Set delegate
- 	inputStream!.delegate = self
- 	outputStream!.delegate = self
- 	// Schedule
- 	inputStream!.schedule(in: .main, forMode: RunLoopMode.defaultRunLoopMode)
- 	outputStream!.schedule(in: .main, forMode: RunLoopMode.defaultRunLoopMode)
- 	print("Start open()")
- 	// Open!
- 	inputStream!.open()
- 	outputStream!.open()
- }

TCP/IP 통신

- func send(data: Data) -> Int {
- let bytesWritten = data.withUnsafeBytes { outputStream?.write($0, maxLength: data.count) }
- return bytesWritten!
- }
- //withUnsafeBytes
- //함수 정의
- //func withUnsafeBytes<R>(_ body: //(UnsafeRawBufferPointer) throws -> R) rethrows -> R

TCP/IP 통신

- withUnsafeBytes 의 클로저 입력으로 들어오는 것은 UnsafeRawBufferPointer 라는 특수한 타입이다. 이 타입은 이름처럼 UnsafeBufferPointer 와 비슷하면서도 UInt8 타입 단위로 동작하는 UnsafeRawPointer 와 거의 동일하다.
- -> raw pointer : 형이 없는 포인터

TCP/IP 통신

- 예제
- void *memcpy(void *dst, const void *src, size_t n);
- //memcpy 함수는 반환값으로 반환형이 없는 포인터를 반환한다.
- //det와 src에 있는 자료형에 관계없이 바이트 단위로 값을 긁어오기 때문.
- -> 자료형을 알 필요가 없음.

TCP/IP 통신

- 예제
- *정수형 포인터(int *)의 주소에 1을 더한 포인터 주소는 sizeof(int) 만큼 증가한 값이다. 즉 타입의 크기 단위로 포인터가 이동한다.*
- Ex)
- Int a[2] = {3,4};
- 배열 a의 주소값 : 0x0001 -> 3
- &a + 1 = 0x0005 -> 4
- 0x0002 주소의 값을 참조할 수 없음.

TCP/IP 통신

- Raw pointer의 크기를 지정해줄 수 있다.
- for i in 0..<bytes {
- // 포인터 B 가 가리키는 메모리의 i 번째 바이트를 읽어온다.
- let value = pointerB.advanced(by: i).load(as: Int8.self)
- print("₩(value)")
- }
- 위의 예제에서 load 메소드를 이용해서 int8로 형변환을 해주었고,
- int8 = 1byte다.(1바이트 단위로 포인터가 이동한다.)

TCP/IP 통신

```
func recv(buffersize: Int) -> Data {
        var buffer = [UInt8](repeating :0, count : buffersize) //버퍼 선언
        let bytesRead = inputStream?.read(&buffer, maxLength: buffersize) //값을 읽어옴
        var dropCount = buffersize - bytesRead!
        //버퍼 사이즈보다 읽어온 사이즈가 더 크면
        if dropCount < 0 {
            dropCount = 0// 카운트를 0으로
        }
        let chunk = buffer.dropLast(dropCount)
        //버퍼에 있는 값을 카운터 값으로 자름.

        return Data(chunk)
        //데이터 반환
    }
```

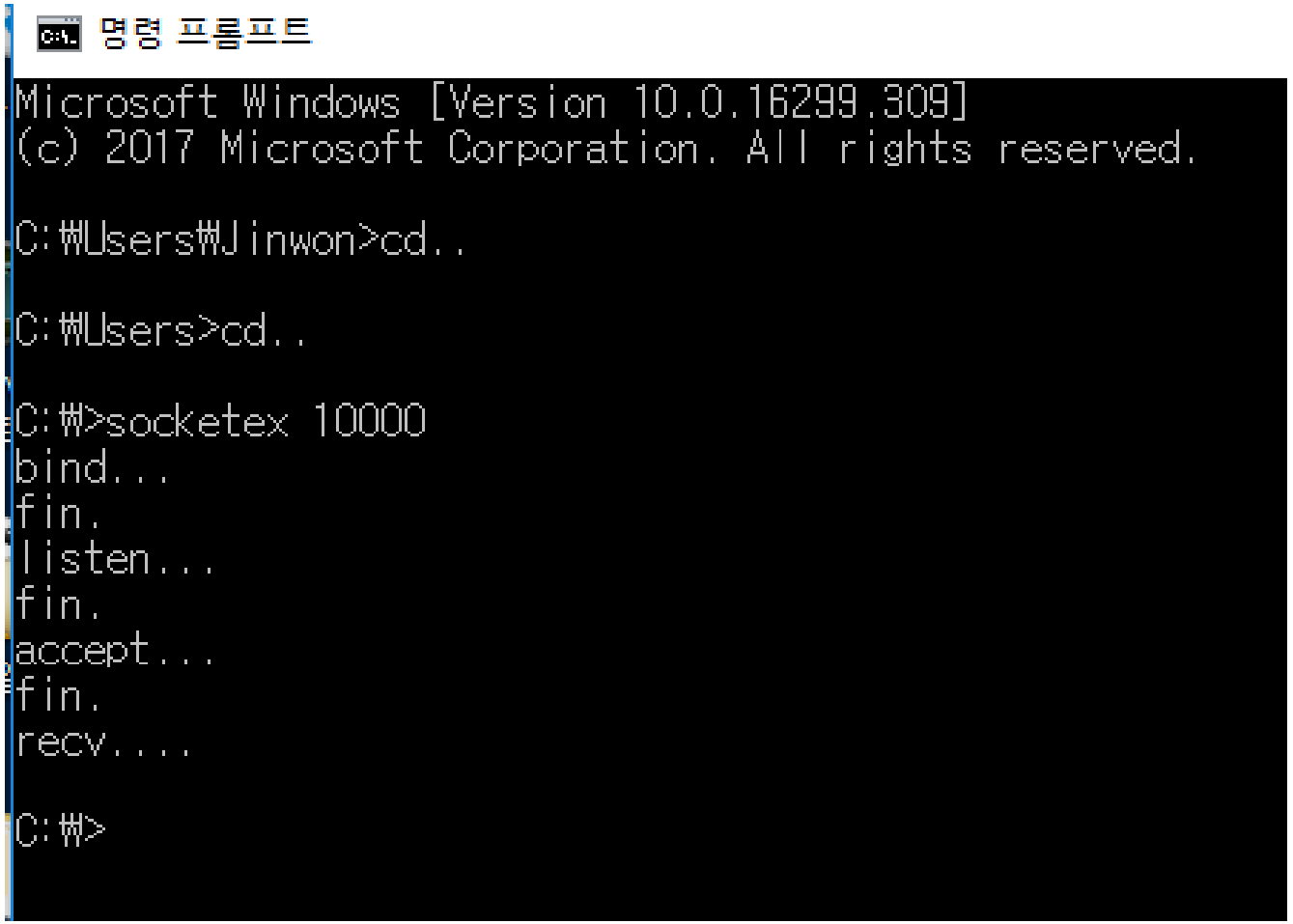
TCP/IP 통신

```
• func disconnect() {
•         inputStream?.close()
•         outputStream?.close()
•     }
```

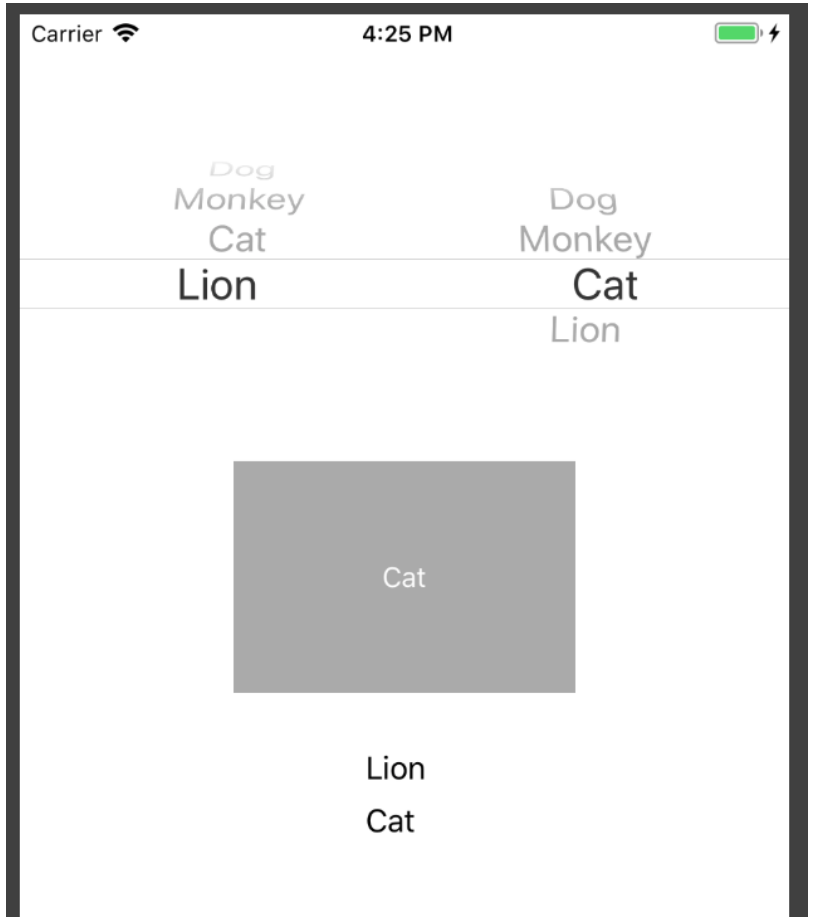
TCP/IP 통신

```
let socket = TcpSocket()
        socket.connect(host: "222.111.165.147", port: 10000)
        let query = "HELLO SWIFT SOCKET!"
        let dataQuery = query.data(using: String.Encoding.utf8, allowLossyConversion: true)
        let sentCount = socket.send(data: dataQuery!)
        print("sentCount : ₩(sentCount)")

        let buffersize = 1024
        let chunk = socket.recv(buffersize: buffersize)
        print("recv")
        var getString : String?
        if(chunk.count > 0){
            getString = String(bytes: chunk, encoding: String.Encoding.utf8)!
            print("received : ₩(getString!)")
        }
        socket.disconnect()
```

TCP/IP 통신


명령 프롬프트
Microsoft Windows [Version 10.0.16299.309]
(c) 2017 Microsoft Corporation. All rights reserved.
C:₩Users₩Jinwon>cd..
C:₩Users>cd..
C:₩>socketex 10000
bind...
fin.
listen...
fin.
accept...
fin.
recv....
C:₩>

Picker view 예제



다음과 같이 버튼과 picker view 레이아웃을 생성

픽커 뷰가 멈추는(선택되는) 항목이 버튼 텍스트에 출력되게 만들고 싶음.

버튼을 누르면 picker view가 숨김상태(안보이는)에서 보이는 상태로 전환.

Picker view 예제

- import UIKit

- class ViewController: UIViewController,UIPickerViewDataSource, UIPickerViewDelegate {
- //UIViewController는 클래스의 상속
- //UIPickerViewDataSource, UIPickerViewDelegate 는 프로토콜 채택
- //-->프로토콜은 중복 채택이 가능
- let animals = ["Dog", "Monkey", "Cat", "Lion"]
- @IBOutlet weak var pickerView: UIPickerView!
- @IBOutlet weak var selectionBtn: UIButton!
- // picker view에 들어갈 animals배열 선언
- // picker view와 버튼의 outlet변수 선언

Picker view 예제

```
class ViewController: UIViewController,UIPickerViewDataSource,
    UIPickerViewDelegate {
```

UIViewControler : 뷰컨트롤러 클래스 상속
UIPickerViewDataSource, UIPickerViewDelegate : 프로토콜 채택

Picker view 예제

```
public protocol UIPickerViewDataSource :
    NSObjectProtocol {


    // returns the number of 'columns' to
        display.
    @available(iOS 2.0, *)
    public func numberOfComponents(in pickerView:
        UIPickerView) -> Int


    // returns the # of rows in each component..
    @available(iOS 2.0, *)
    public func pickerView(_ pickerView:
        UIPickerView, numberOfRowsInComponent
        component: Int) -> Int
}
```

DataSource의 프로토콜
PickerView에 사용되는 내부적인 2개의 함수가 프로토타입으로 구현되어 있다.
-> 채택한 클래스는 반드시 2개의 함수를 구현해 주어야 한다.

Picker view 예제

```
public protocol UIPickerViewDelegate :
    NSObjectProtocol {


    // returns width of column and height of row
        for each component.
    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, widthForComponent
        component: Int) -> CGFloat

    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, rowHeightForComponent
        component: Int) -> CGFloat

    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, titleForRow row: Int,
        forComponent component: Int) -> String?

    @available(iOS 6.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, attributedTitleForRow row:
        Int, forComponent component: Int) ->
        NSAttributedString? // attributed title
        is favored if both methods are
        implemented

    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, viewForRow row: Int,
        forComponent component: Int, reusing
        view: UIView?) -> UIView


    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView:
        UIPickerView, didSelectRow row: Int,
        inComponent component: Int)
}
```

Picker view 예제

- 프로토콜 정의에서 optional로 구현된 것은 구현하지 않아도 무방하나,
- -> nil값을 반환
- 구현하지 않은 메소드들은 default값을 취한다.

Picker view 예제

```
override func viewDidLoad() { //뷰컨트롤러가 실행되면 메소드 실행
        super.viewDidLoad() // 슈퍼클래스의 값을 델리게이트 함.

        pickerView.isHidden = true // 픽커뷰를 숨김

        pickerView.delegate = self
        pickerView.dataSource = self
        // 메시지(델리게이트 함수 호출)처리를 이 클래스에서 함.
        // 특정한 상황에서 델리게이트 함수가 호출될 때, 이 클래스(self)에서
        // 선언된 델리게이트 함수를 사용함.
    }
```

Picker view 예제

```
@available(iOS 2.0, *)
open class UIPickerView : UIView, NSCoding {

    weak open var dataSource:
        UIPickerViewDataSource? // default is
        nil. weak reference

    weak open var delegate:
        UIPickerViewDelegate? // default is nil.
        weak reference
```

픽커뷰 클래스 내부에 선언된
dataSource 클래스와
delegate 클래스

초기값이 없으면 프로토콜은
모든 델리게이트 함수를
default 값을 받는다.

Picker view 예제

```
@IBAction func selectPressed(_ sender: UIButton) {
        //버튼 클릭 이벤트 메서드
        if (pickerView.isHidden){
            pickerView.isHidden = false
        }
        else{
            pickerView.isHidden = true
        }
}
```

Picker view 예제

- //프로토콜(UIPickerViewDataSource)에서 정의된 메서드 구현.
- public func numberOfComponents(in pickerView: UIPickerView) -> Int{
- 	return 2
- }
- // returns the # of rows in each component..
- // 리턴값으로 픽커뷰의 갯수 반환
-
- public func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int{
- 	return animals.count
- } // 컴포넌트의 행의 갯수

Picker view 예제

- //UIPickerViewDelegate 프로토콜 메서드 구현
- func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int){
  if (component == 0) // 컴포넌트가 0인 경우
  {
  label1.text = animals[row]
  }
  else
  {
  label2.text = animals[row]
  }
- selectionBtn.setTitle(animals[row], for: .normal)
- } // 컴포넌트가 셀렉트 되었을때

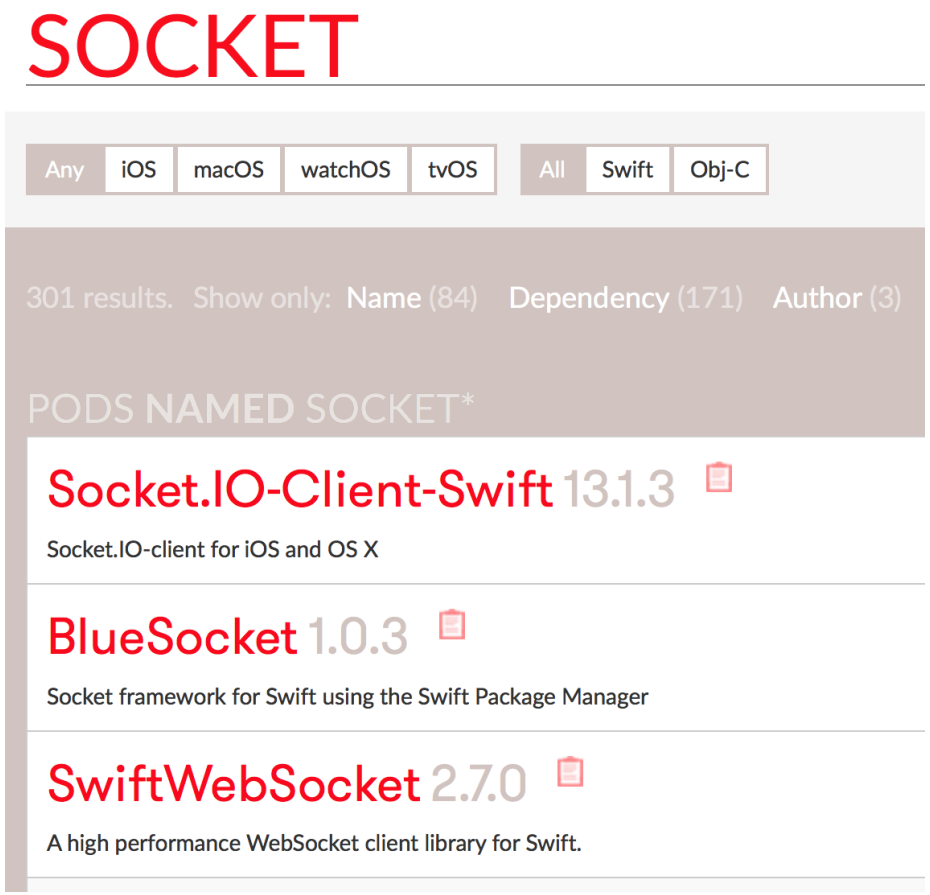
Picker view 예제

- func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?{
- return animals[row]
- } // 컴포넌트의 이름
- func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?{
- return animals[row]
- } // 컴포넌트의 이름 설정

Cocoapod

- Graph를 활용하기 위해 cocoapod을 사용하였음.
- cocoapod은 외부 라이브러리를 내 프로젝트로 가져올 수 있게 만들어줌.
- https://cocoapods.org // 오픈소스 라이브러리 저장소

Cocoapod



'Socket' 키워드를 입력한 결과.
결과값을 여러 분류로 검색이 가능하다.

Cocoapod

- 터미널 실행
- $sudo gem install cocoapods
- //mac은 루비환경 지원
- 자신의 프로젝트가 있는 path로 이동
- $pod init
- //podfile 생성

Cocoapod

Podfile

프로젝트 경로에 만들어진 podfile

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'graph_test3' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!
  pod 'SwiftChart', '~> 1.0'
  # Pods for graph_test3

  target 'graph_test3Tests' do
    inherit! :search_paths
    # Pods for testing
  end

  target 'graph_test3UITests' do
    inherit! :search_paths
    # Pods for testing
  end

end
```

그래프를 위해
SwiftChart 라이브러리를
가져옴.

Cocoapod


Small pie-chart vi
SwiftCh
Easy to use and h
For your Podfile
pod 'SwiftChart', '~> 1.0'
SwiftChart 1.0.1
Line and area chart library

Cocoapod

```
ADMINui-MacBook-Pro:graph_test3 kanghyunmoon$ pod install
Analyzing dependencies
Downloading dependencies
Installing SwiftChart (1.0.1)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `graph_test3.xcworkspace` for this project from now on.
Sending stats
Pod installation complete! There is 1 dependency from the Podfile and 1 total pod installed.
```

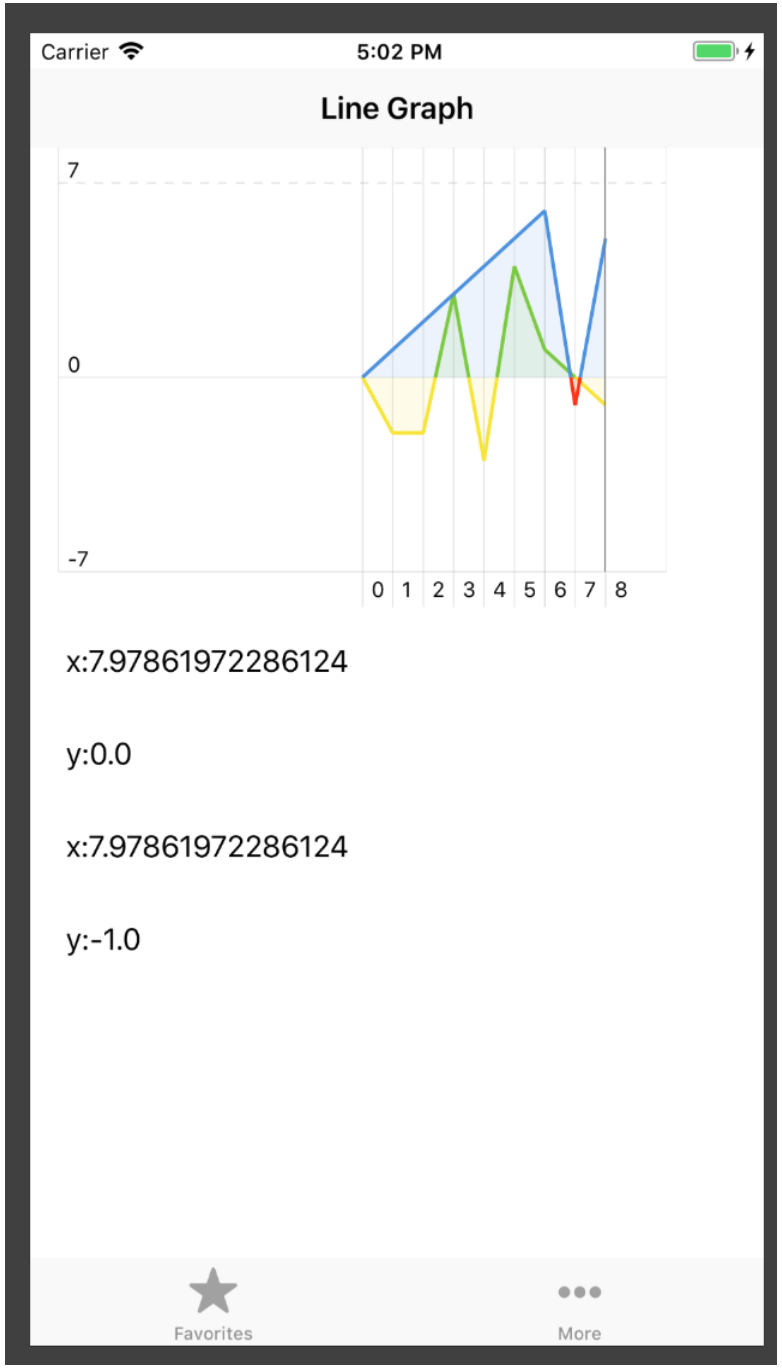
터미널에 pod install을 입력하면 podfile에 적어둔 라이
브러리를 현재 폴더에 있는 프로젝트에 포함시킴.

Cocoapod

```
import UIKit
import CoreGraphics
import Foundation
import SwiftChart
```

install이 끝나게 되면 workspace가 만들어지고, workspace에 import 를 해서 자유롭게 사용가능

Graph



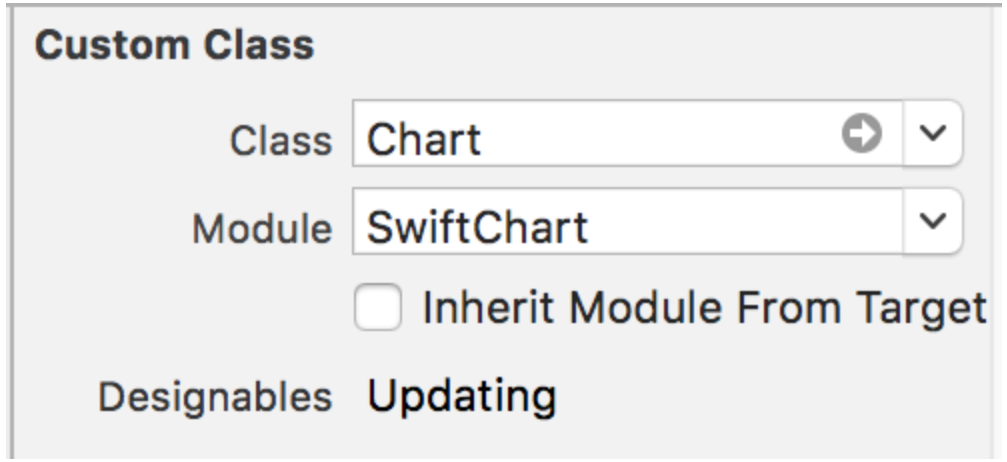
Line graph 2개를 그린다.
그 그래프를 클릭하면,
클릭한 위치의 x좌표값에 따른 y값을 출력한다.

Graph

```
class ViewController: UIViewController, ChartDelegate{

    @IBOutlet weak var chart: Chart!
    @IBOutlet weak var NavigationBar: UINavigationBar!
    @IBOutlet weak var tabBar: UITabBar!
    @IBOutlet weak var viewCell1: UITableViewCell!
    @IBOutlet weak var viewCell2: UITableViewCell!
    @IBOutlet weak var viewCell3: UITableViewCell!
    @IBOutlet weak var viewCell4: UITableViewCell!
```

ChartDelegate 프로토콜 채택
그래프에 관련한 변수 선언

Graph



Graph를 출력할 View를 Chart 클래스로 만들어줌.

Graph

```
func didTouchChart(_ chart: Chart, indexes: Array<Int?>, x: Double,
    left: CGFloat) {
    for (seriesIndex, dataIndex) in indexes.enumerated() {
        if let value = chart.valueForSeries(seriesIndex, atIndex:
            dataIndex) {
            print("Touched series: \(seriesIndex): data index: \
                (dataIndex!); series value: \(value); x-axis value: \(x)
                (from left: \(left))")

            if (seriesIndex == 0)
            {
                viewCell1.textLabel?.text = "x:\(x)"

                viewCell2.textLabel?.text = "y:\(value)"
            }
            else
            {
                viewCell3.textLabel?.text = "x:\(x)"

                viewCell4.textLabel?.text = "y:\(value)"
            }

        }
    }
}
```

ChartDelegate 프로토콜의 메서드 구현
차트가 클릭 되었을 때 실행되는 메서드.
Chart = 터치된 차트
Indexes = 차트에서 선택된 x축의 값
-> 시리즈 순서대로 값이 들어옴.
X : 차트 x축의 좌표(double)
Left : 차트 맨 왼쪽 좌표로부터 떨어진 값.

Graph

```
func didFinishTouchingChart(_
    chart: Chart) {
}

func didEndTouchingChart(_ chart:
    Chart) {
}
//delegate func...
```

나머지 delegate 함수 구현

Graph

```
override func viewDidLoad() {
    super.viewDidLoad()

    chart.delegate = self
    NavigationBar.topItem?.title = "Line Graph"
    // title 설정

    // Chart with y-min, y-max and y-labels formatter
    let data: [Double] = [0, -2, -2, 3, -3, 4, 1, 0, -1]
    let data2: [Double] = [0, 1, 2, 3, 4, 5, 6, -1, 5]
```

Navigationbar 타이틀 설정

그래프값 임의 설정

Graph

```
let series = ChartSeries(data)
series.colors = (
    above: ChartColors.greenColor(),
    below: ChartColors.yellowColor(),
    zeroLevel: 0
)
let series2 = ChartSeries(data2)
series.colors = (
    above: ChartColors.greenColor(),
    below: ChartColors.yellowColor(),
    zeroLevel: 0
)
series.area = true
series2.area = true

chart.add(series)
chart.add(series2)
```

시리즈 2개 구현
기준 레벨 0으로 설정
기준 위 아래 색깔 구현

그래프 영역 색채우기 true

Graph

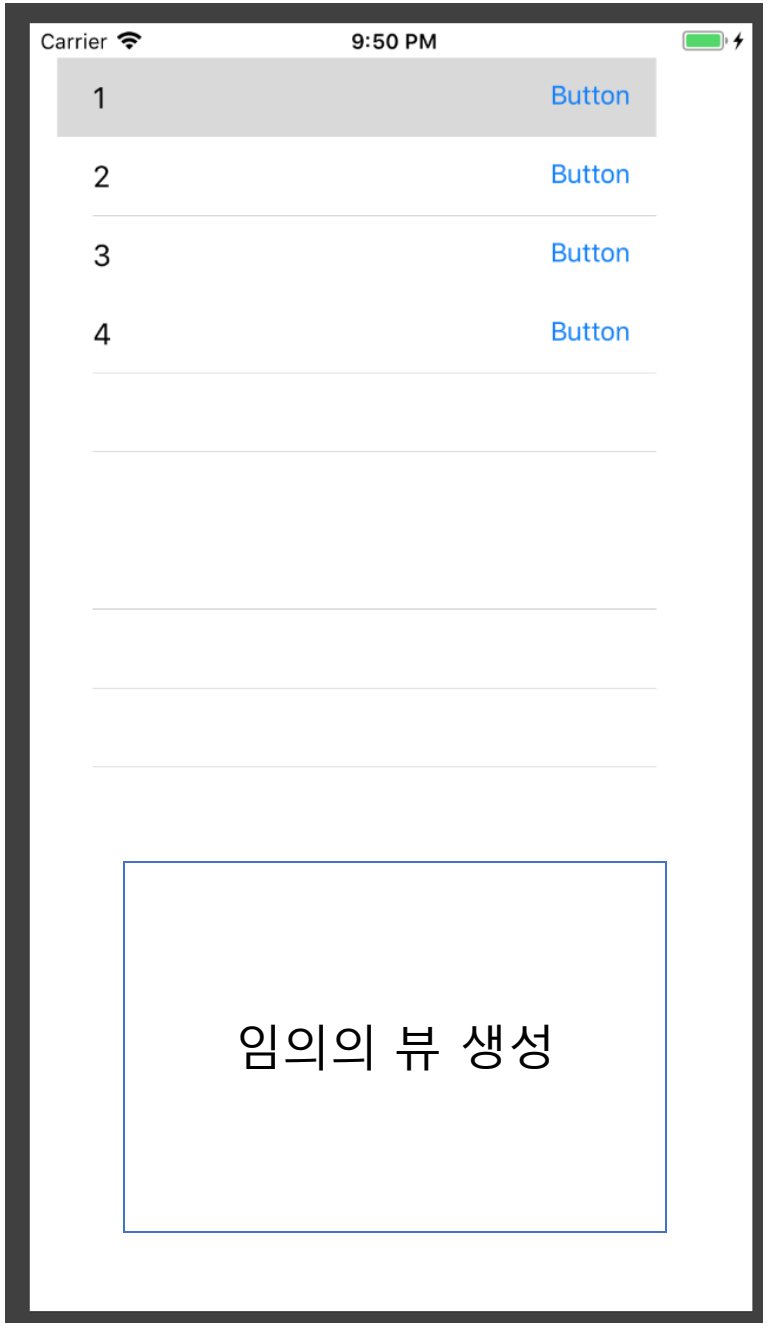
```
        // Set minimum and maximum values for y-axis
        chart.minY = -7
        chart.maxY = 7

        chart.minX = -10
        chart.maxX = 10



        // Format y-axis, e.g. with units
        chart.yLabelsFormatter = { String(Int($1))}

    }
}
```

차트 표시 영역 설정

그래프 y축에 표시될 단위 설정

Ex) chart.xLabelsFormatter = { String(Int($1))+"K"}

Gesture



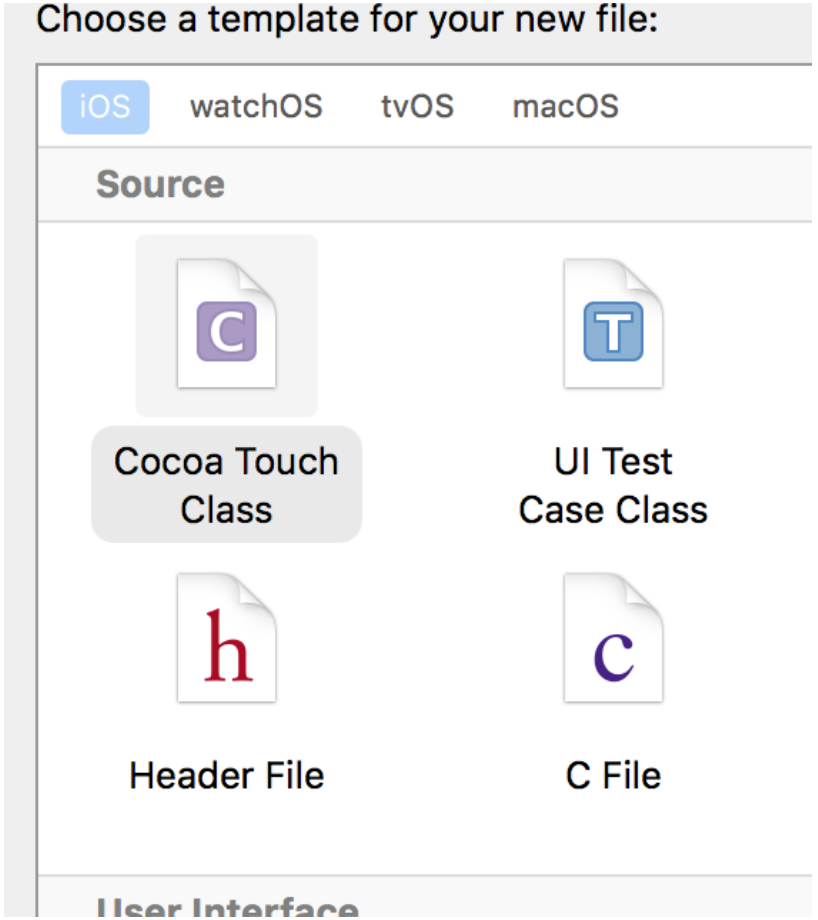
LongGesture : tableView를 길게 누를때 작동
-> 테이블뷰 안의 셀의 버튼을 off

TapGesture : 임의의 View를 짧게 누를때 작동
-> 테이블뷰 안의 셀의 버튼을 on

Gesture

- TableView 안에 Cell이 존재하고, 그 Cell 안에 button 존재.
- cell은 custom으로 설정.(안에 버튼을 부여하기 위함)
- 만들어진 cell은 그 안의 프로퍼티들을 사용하기 위해 새로운 class를 만들어서 사용.

Gesture



Cocoa touch class를 사용하여 클래스 생성.
Class의 이름은 TableViewCell

Gesture

```
import UIKit

class TableViewCell: UITableViewCell{

    @IBOutlet weak var btn: UIButton!

    @IBOutlet weak var cellView: UIView!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code

        btn.setTitle("Hi!", for:
            UIControlState.normal)

        print(12345)


    }

    override func setSelected(_ selected:
        Bool, animated: Bool) {
```

클래스를 만듬.
UITableViewCell 클래스를 상속받고 있음.

셀 안에 구현된 버튼과 셀뷰의 변수를 선언.

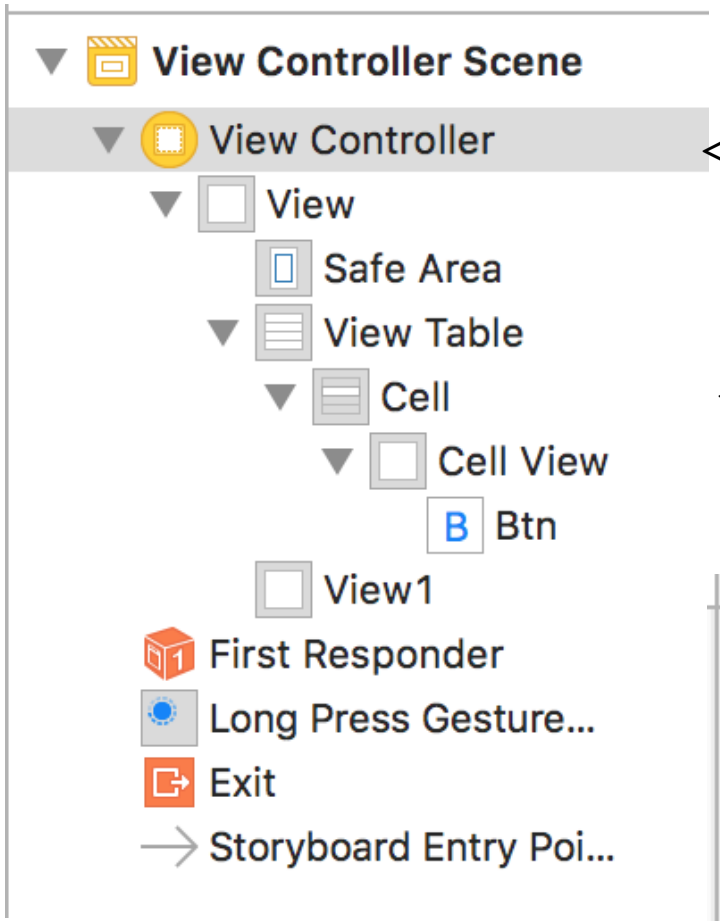
Nib파일에서 코드를 가져올 때 실행 (초기화 함수)

버튼의 title을 메서드를 사용하여 변경

디버깅용 메시지

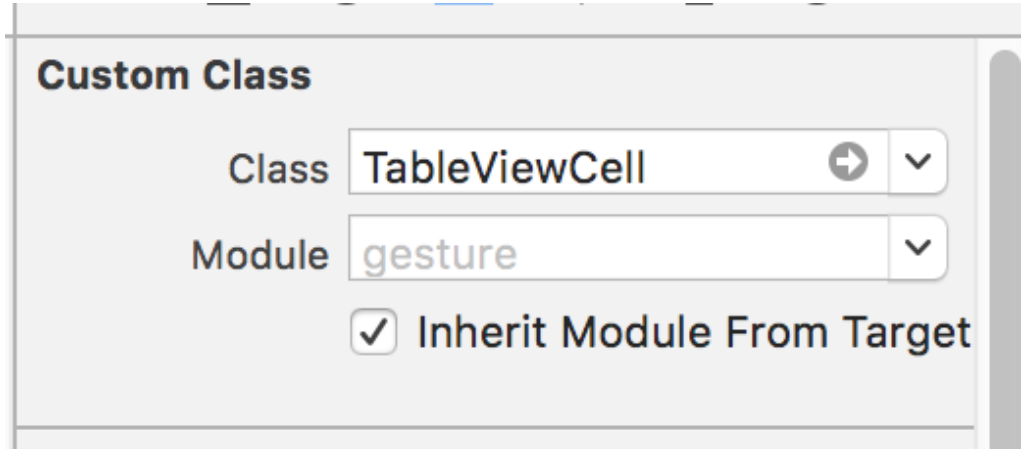
클래스에서 상속받은 나머지 함수

Gesture



<< 메인 뷰컨트롤 클래스

<<방금 본 클래스는 이 계층(custom cell)에서 구현된 클래스.
cell의 class를 TableViewCell로 아까 생성해준 클래스와 연결해주어야함.(연결 안하면 default=UITableViewCell)



Gesture

```
import UIKit

class ViewController: UIViewController,
    UITableViewDataSource,
    UITableViewDelegate{


    @IBOutlet weak var View1: UIView!

    var titles = ["1","2","3","4"]

    func tableView(_ tableView: UITableView,
        numberOfRowsInSection section: Int)
        -> Int {
        return titles.count
    }

    var arr_cell : NSMutableArray = []
    var index = 0
```

컨트롤러 상속, 프로토콜 채택

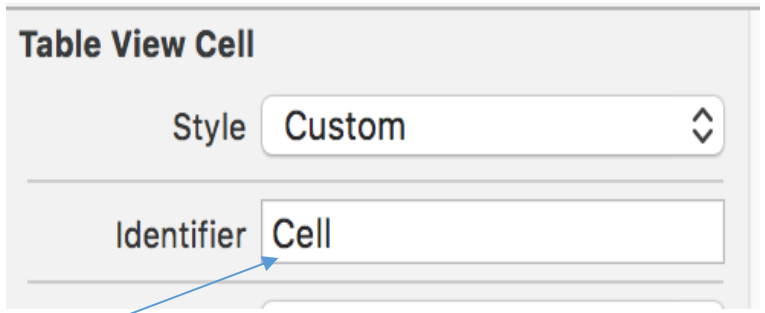
임의의 뷰 변수 생성.
셀의 타이틀 변수 선언.

dataSource의 프로토콜 매서드 구현
(반환 값으로 테이블뷰의 안에 셀의 갯수를 반환)

셀의 인스턴스를 저장하기 위한 배열 선언.

Gesture

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {

    let cell =
        tableView.dequeueReusableCell(withIdentifie
        r: "Cell", for:indexPath)
    cell.textLabel?.text = titles[indexPath.row]

    arr_cell[index] = cell

    index = index+1


    //셀의 텍스트라벨 설정

    return cell
}
```



identifier이 "Cell"인 재사용 가능 셀을 큐에서 하나씩 빼온다.

그 셀의 textLabel의 text를 타이틀로 설정.

이 cell의 인스턴스를 배열에 저장.
인덱스 증가

*reuseable cell:
메모리 절약을 위함
Reuseable cell 객체하나를 사용하여, dataSource를 기반으로 내용을 바꾸어가며 화면에 셀을 표시.
스크롤이 올라가서 화면 밖으로 밀려나면, 다시 재사용 풀(pool)로 들어감.

Gesture

```
@objc func func1()->(){
    print(1)

    for num in 0..<index {
        (arr_cell[num] as!
            TableViewCell).btn.isHidden =
            false
    }

}
```

Gesture을 구현하는 방법 1

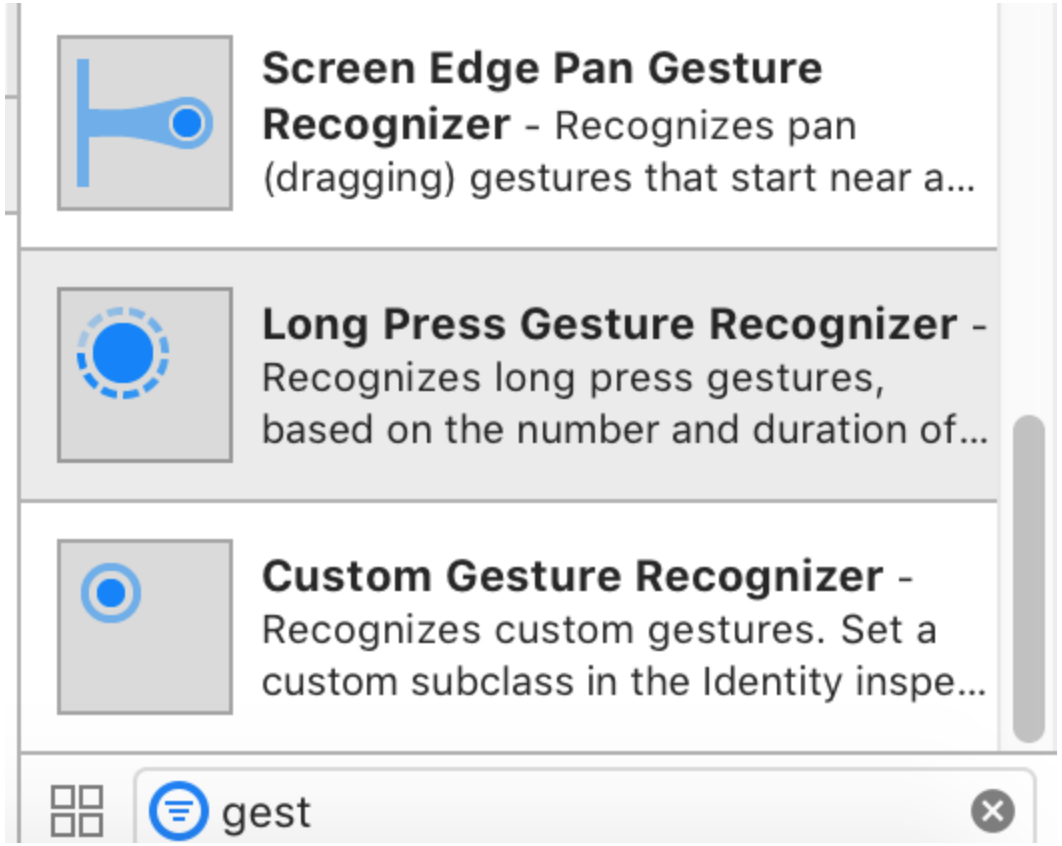
실행할 함수 선언
Func1 => tapGesture
배열에 저장된 셀들을 isHidden
매서드로 보여지는 상태로 설정.

Gesture

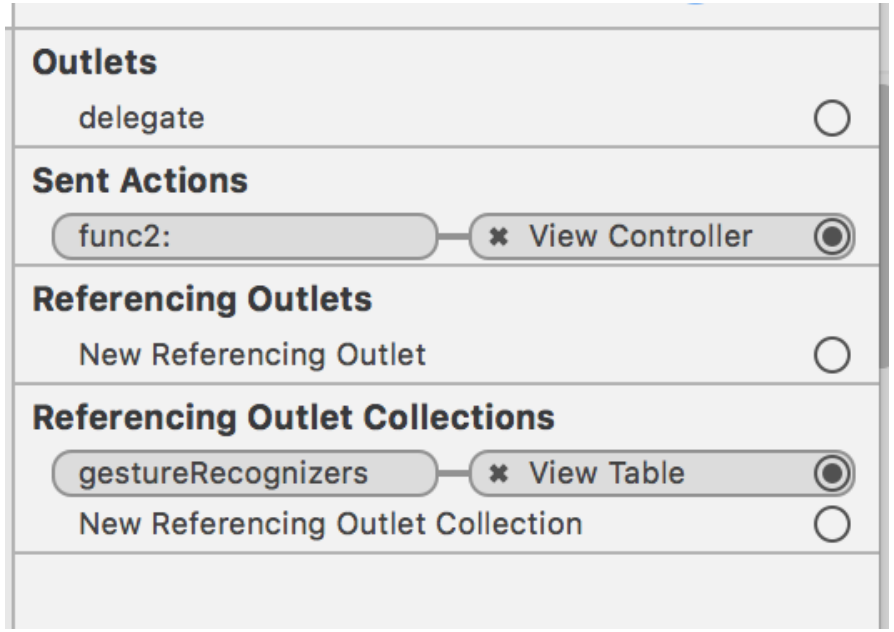
```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading
        the view, typically from a nib.
    viewTable.delegate = self
    viewTable.dataSource = self

    let tapGesture =
        UITapGestureRecognizer(target: self,
        action: #selector(func1))
    View1.addGestureRecognizer(tapGesture)
    View1.isUserInteractionEnabled = true

}
```

Gesture 객체 생성.
(selector로 func1함수를 전달해주고 있다.)
임의의 View1에 gesture를 선언.
유저인터렉션 설정 true

Gesture



방법 2
드래그로 객체를 만들어 준다.
아래 그림은 longGesture가 연결된 모습
Sent Actions로 func2함수를 호출하고 있다.



Gesture

```
@IBAction func func2(_ sender: Any) {
    print(2)

    for num in 0..<index {
        (arr_cell[num] as!
            TableViewCell).btn.isHidden =
            true
    }

}
```

Func2함수

버튼을 감추는 기능.