http://libqa.com/wiki/723

토픽 = Swift

[1.Swift 문법 소개 (50분)](http://www.libqa.com/wiki/741)

- 반복문 = for, for 중첩, for~in, while, repeat~while

- 제어전달문 = break, continue

- 집단자료형 = 배열, 집합, 튜플, 딕셔너리 <== 자세히

- 옵셔널 <== 자세히

- 함수 <== 자세히

- 구조체와 클래스 <== 자세히

- 프로토콜

- 오류처리

[2.스토리보드 사용법을 알아보기 (50분)](http://www.libqa.com/wiki/754)

- 스토리보드 화면 구성

- iOS와 앱 생명주기

- 버튼 생성 및 실행

- 화면전환

- 뷰 컨트롤러 간 데이터 교환

- 메시지 알림창 구현

[3.네트워크 통신과 테이블뷰 생성 시연 및 코드리뷰 (25분)](http://www.libqa.com/wiki/755)

[4.유아(5~7세), 초등 1학년을 위한 더하기빼기 숫자 공부 어플리케이션 시연 및 코드리뷰 (25분)](http://www.libqa.com/wiki/756)

- 자리수 확장하며 더하기/빼기

- 한자리 숫자 더하기/빼기

- 두자리 숫자 더하기/빼기

- 세자리 숫자 더하기/빼기

- 틀린문제 복습하기

- 일별 학습량 관리

- 숙제내기/숙제검사

- 목표설정 및 달성안내

- 통계 화면 제공

- 로컬스토리지 사용 또는 네트워크 이용..으로 데이터저장소를 선택하여 사용할 수 있도록 설정기능 제공

| **작성자** | **dazzilove** |
| --- | --- |
| 작성일 | 2016.09.22 13:07:00 |
| 수정자 | **dazzilove** |
| 수정일 | 2017.01.09 04:13:00 |
| 조회수 | 116 |
| 추천수 | 0 |
| 글상태 | 잠금 |

1.Swift 문법 소개

1.Swift 문법 ﻿소개

* Swift는 타입추론을 이용한다.
* Swift는 줄바꿈에 ;(세미콜론)을 표기하지 않아도 된다.
* null 표기 = nil
* 상수 = let, 변수 = var

1.1.반복문

for

for(초기값; 비교조건; 증가값) {  
 반복 실행할 구문  
}

for~in

/\*=============================

for 루프 상수 in 순회 대상 {

실행할 구문

}

=============================\*/

for index in 1...5 {

print("\(index) times 5 is \(index \* 5)")

}

// 1 times 5 is 5

// 2 times 5 is 10

// 3 times 5 is 15

// 4 times 5 is 20

// 5 times 5 is 25

let base = 3

let power = 10

var answer = 1

for \_ in 1...power {

answer \*= base

}

print("\(base) to the power of \(power) is \(answer)")

// Prints "3 to the power of 10 is 59049"

/\*=============================

for row in Object {

실행할 구문

}

=============================\*/

let names = ["Anna", "Alex", "Brian", "Jack"]

for name in names {

print("Hello, \(name)!")

}

// Hello, Anna!

// Hello, Alex!

// Hello, Brian!

// Hello, Jack!

let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]

for (animalName, legCount) in numberOfLegs {

print("\(animalName)s have \(legCount) legs")

}

// ants have 6 legs

// spiders have 8 legs

// cats have 4 legs

while

while 조건식 {  
 실행할 구문  
}

repeat~while 

repeat {  
 실행할 구문  
}  
while 조건식

1.2.제어전달문

break

for i in 0...6 {  
 if i > 3 {  
 break  
 }  
}

continue

for i in 0...6 {  
 if i < 3 {  
 continue  
 }  
}

1.3.집단자료형

배열, 집합, 튜플, 딕셔너리 <== 자세히

배열

var countries = ["Korea", "China", "Japan", "America"]  
  
print(countries[0])  
print(countries[2])  
  
countries.append("Egypt")  
  
print(countries[4])

집합

import UIKit  
  
var sports : Set<String> = ["tennis", "footboal", "baseboal"]  
var numbers : Set = [1, 2, 3, 4, 5]  
var emptySet : Set<String> = [] // 값이 없는 경우에는 반드시 type을 명기해야 함.  
  
if emptySet.isEmpty {  
 print("집합이 비어있습니다.")  
}  
  
print("집합 numbers 에는 \(numbers.count)개의 아이템이 저장되어 있습니다.")  
  
for sport in sports {  
 print("종목 = \(sport)")  
}  
  
sports.insert("fencing")  
print("Sports 갯수는 \(sports.count)입니다.")  
  
sports.insert("tennis")  
print("Sports 갯수는 \(sports.count)입니다.") // 동일한 값이 두번 등록되지 않는다.  
  
sports.remove("footboal")  
print("Sports 갯수는 \(sports.count)입니다.")  
  
sports.removeAll()  
print("Sports 갯수는 \(sports.count)입니다.")  
  
if sports.isEmpty {  
 print("집합이 비어있습니다.")  
}

튜플

import UIKit  
  
// 튜플은 여러가지 타입의 아이템을 저장 할 수 있다. (튜플의 저장 형태를 확인하자 ==> .0 "a")  
// 정의된 튜플은 추가, 수정, 삭제가 불가하다.  
let tupleValue = ("a", "b", 1, 2, true, false)  
  
// 튜플의 인덱스는 0부터 시작한다.  
tupleValue.0  
tupleValue.1  
tupleValue.2  
tupleValue.3  
tupleValue.4  
tupleValue.5  
  
//튜플은 별도의 선언 구문이 없지만 타입 어노테이션을 사용하기위한 타입은 정의 할 수 있다.  
let tuple1 : (Int, Int) = (1, 2)  
let tuple2 : (String, Int, Int) = ("sample", 1, 2)  
let tuple3 : (Bool, String, Int) = (false, "sample", 1)  
let tuple4 : (Int, String, String, Float) = (1, "sample1", "sample2", 1.0)  
  
// 어노테이션 타입을 정의하지 않았을 경우 유의사항  
// 타입추론원칙에 의해 의도하지 않는 타입으로 추론 될 수 있다.  
// 타입추론 예1  
let tuple5 = ("a")  
// ==> String 또는 Character 타입이 될 수 있다.  
// ==> 별도의 타입 어노테이션을 작성하지 않으면 타입 추론 원칙에 의해 상위 타입인 String으로 추론 된다.  
// 타입추론 예2  
let tuple6 = (1.0)  
// ==> Float 또는 Double 타입이 될 수 있다.  
// ==> 별도의 타입 어노테이션을 작성하지 않으면 타입 추론 원칙에 의해 상위 타입인 Double로 추론된다.  
  
// 튜플의 원소를 개별변수나 상수로 할당 할 수 있음  
let tuple7 : (String, Character, Int, Float, Bool) = ("sample", "t", 1, 2.0, false)  
let (a, b, c, d, e) = tuple7  
print(a)  
print(b)  
print(c)  
print(d)  
print(e)  
  
// 튜플의 활용 : 함수의 리턴타입으로 사용  
func getTupleValue() -> (String, String, Int) {  
 return ("value1", "value2", 5)  
}  
let (returnValue1, returnValue2, returnValue3) = getTupleValue()  
print(returnValue1)  
print(returnValue2)  
print(returnValue3)  
  
// 튜플의 활용 : 함수의 리턴타입으로 사용. 모든 값이 필요하지 않는 경우  
let (returnValue4, returnValue5, \_) = getTupleValue()  
print(returnValue4)  
print(returnValue5)

딕셔너리(Dictionary)

* key:Value로 지정되는 집단 자료형
* 하나의 키는 하나의 데이터에만 연결되어야 한다.
* 하나의 딕셔너리에 키는 중복될 수 없다. 중복선언한 경우 기존키의 값이 변경된다.
* 지정할 수 있는 데이터 타입에는 제한이 없지만, 하나의 딕셔너리에는 하나의 데이터 타입만 정의해야 한다.
* 딕셔너리의 아이템에는 순서가 없으나 키는 내부적으로 순서가 있으므로 for ~ in 구문을 이용한 순회 탐색이 가능하다.
* 딕셔너리에서 사용할 수 있는 키의 타입은 거의 제한이 없지만, 해시(hash) 연산이 가능한 타입이어야 한다.
* 딕셔너리가 아이템을 저장할 때는 입력된 키를 그대로 사용하는 것이 아니라 내부적으로 해시 연산을 거친 값으로 변환한 다음 이를 정렬하여 사용한다.

import UIKit  
  
var characterSet = ["KO":"EUC-KR", "JP":"EUC-JP", "GB":"GB-2312"]  
print(characterSet["KO"])  
print(characterSet["JP"])  
print(characterSet["GB"])  
/\*  
Playground의 메모리 스택 구조  
  
(.0 "JP", .1 "EUC-JP")  
(.0 "GB", .1 "GB-2312")  
(.0 "KO", .1 "EUC-KR")  
  
입력된 순서와는 다르지만(해시 문자열 처리하여 정렬 됨)  
각 아이템이 모두 튜플 형태로 저장되어 있다.  
\*/  
  
  
// 빈 딕셔너리 선언 및 초기화  
Dictionary<String, Int>()  
Dictionary<String, String>()  
Dictionary<String, AnyObject>()  
Dictionary<Character, String>()  
  
// 빈 딕셔너리 선언 및 초기화. 간략한 방식  
[String : Int]()  
[String : String]()  
[String : AnyObject]()  
[Character : String]()  
  
// 딕셔너리 선언과 초기화 분리 1  
var characterSet2 : Dictionary<String, String>  
characterSet2 = Dictionary()  
  
// 딕셔너리 선언과 초기화 분리 2  
var characterSet3 : [String : String]  
characterSet2 = [String : String]()  
  
// 딕셔너리 초기화  
characterSet3 = Dictionary<String, String>()  
characterSet3 = Dictionary() // 타입이 사전에 명시적으로 선언 된 경우에만 가능 함  
characterSet3 = [String : String]()  
characterSet3 = [:] // 타입이 사전에 명시적으로 선언 된 경우에만 가능 함  
characterSet3 = ["KO":"EUC-KR", "JP":"EUC-JP", "GB":"GB-2312"]  
  
// 동적 아이템 추가  
characterSet3["EN"] = "UTF-8"  
  
// 딕셔너리 상태 및 값 확인  
if characterSet3.isEmpty {  
 print("characterSet3 is empty.")  
} else {  
 print("characterSet3 사이즈는 \(characterSet3.count) 입니다.")  
}  
  
// 딕셔너리 순회 조회  
for row in characterSet3 {  
 print("KEY = \(row.key), VALUE = \(row.value)")  
}  
for row in characterSet3 {  
 let (key, value) = row  
 print("KEY = \(key), VALUE = \(value)")  
}  
for (key, value) in characterSet3 {  
 print("KEY = \(key), VALUE = \(value)")  
}  
  
// 아이템 삭제  
characterSet3.removeValue(forKey: "JP")  
print(characterSet3.count)  
  
characterSet3.removeValue(forKey: "NONE")  
print(characterSet3.count)  
  
characterSet3.removeAll()  
print(characterSet3.count)

1.4.옵셔널(Optional)

* 옵셔널은 스위프트에서 도입된 새로운 개념으로 언어 차원에서 프로그램의 안정성을 높이고자 하는 개념이다.
* 오류가 발생할 가능성이 있는 객체를 옵셔널 타입으로 감싸 전달하여 잠재적 오류에 대해 대응 한다.
  + 사실.. 강제해제하여 사용하기 시작하면 별 의미 없는것 같기도 하다.
* 옵셔널 래핑 = ?, 옵셔널 강제 해제 = !
  + 옵셔녈은 오류가능성이 있는 값을 옵셔널 타입이라는 객체로 감싼 후 반환하는 개념으로 이를 옵셔널 래핑(Optional Wrapping) 이라고 한다.
  + 옵셔널 래핑은 선언하는 자료형에 ?를 붙이면 된다.
  + 옵셔널 래핑 된 값을 사용할 때에는 !연산자(Forced-Unwrapping Operator)를 이용하여 옵셔널을 강제 해제하여 사용해야 한다.
* 옵셔널 비강제 해제
  + if 구문 내에서 조건식 대신 옵셔널 타입의 값을 일반 타입의 변수나 상수에 할당하는 구문을 사용하는 방식이다.
* 옵셔널 자동 해제
  + 옵셔널 객체의 값을 비교 연산자를 사용하여 비교하는 경우 옵셔널 객체를 강제 해제 하지 않더라도 한쪽이 옵셔널, 다른 한쪽이 일반 타입이라면 자동으로 옵셔널 타입을 해제하여 비교 연산을 수행한다.
* 옵셔널 체인(Optional Chain)
  + 옵셔널 체인은 옵셔널 타입으로 정의된 값이 하위 프로퍼티나 메소드를 가지고 있을 때, 이 요소들을 if 구문을 쓰지 않고도 간결하게 사용할 수 있는 코드를 작성하기 위해 도입 되었다.
  + 옵셔널 체인은 객체가 nil인 상황에서 안전성 검사를 하지 않고 메소드나 프로퍼티를 호출하더라도 오류가 발생하지 않을 수 있는 문법을 옵셔널 스타일을 이용하여 구현한다.
  + 예) company?.ceo?.name
  + 옵셔널 체인의 특징
    - 옵셔널 체인으로 참조된 값은 무조건 옵셔널 타입으로 반환된다.
      * 옵셔널 체인 구문에서 마지막에 오는 값이 옵셔널 타입이 아닌 일반 값일지라도 옵셔널 체인을 통해 참조했다면 이 값은 옵셔널 타입으로 변경된다.
      * 옵셔널 체인이라는 구문 자체가 nil을 반환할 가능성을 내포하고 있기 때문이다.
    - 옵셔널 체인 과정에서 옵셔널 타입들이 여러 번 겹쳐 있더라도 중첩되지 않고 한 번만 처리된다.
      * Optional(Optional((Optional("a")))) = Optional("a")
  + 옵셔널 체인 구문 = someCompany()?.getCEO()?.name <== 옵셔널 체인이 적용된 객체가 nil 이라도 오류가 발생하지 않음
  + 옵셔널 강제 해재 = someCompany()!.getCEO()!.name  <== 옵셔널 체인이 적용된 객체가 nill 인 경우 런타임 오류 발생

import UIKit  
  
// 옵셔널 래핑  
var optInt : Int? = 3  
var optString : String? = "test"  
var str = "dazzilove"  
  
// 옵셔널 객체 값 확인  
print(optInt) // Optional(3)  
print(optString) // Optional("test")  
  
// 옵셔널 강제 해제  
print(optInt!) // 3  
print(optString!) // "test"  
  
// 옵셔널 타입 연산  
var optInt1 : Int? = 1  
var optInt2 : Int? = 2  
  
// 강제 해제 하지 않고 연산 시 오류 발생  
// let optIntPlusRetuls = optInt1 + optInt2  
/\*  
연산 시도 시 아래와 같은 에러가 발생 한다.  
error: MyPlayground.playground:12:34: error: value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'?  
let optIntPlusRetuls = optInt1 + optInt2  
\*/  
  
// 강제 해제 하여 연산 시 정상 처리 됨  
let optIntPlusRetuls = optInt1! + optInt2! // 3  
  
  
// 옵셔널 객체의 상태를 확인하여 안전한 옵셔널 강제 해제 처리  
var intFromStr = Int(str)  
if intFromStr != nil {  
 print("값 변환에 성공하였습니다. 변환된 값은 \(intFromStr!)입니다.")  
} else {  
 print("값 변환에 실패하였습니다.")  
}  
  
// 옵셔널 비강제 해제  
str = "1234"  
  
intFromStr = Int(str)  
if intFromStr != nil {  
 print("값 변환에 성공하였습니다. 변환된 값은 \(intFromStr!)입니다.")  
} else {  
 print("값 변환에 실패하였습니다.")  
}  
  
if let temp = Int(str) { // 옵셔널 비강제 해제  
 print("값 변환에 성공하였습니다. 변환된 값은 \(temp)입니다.")  
} else {  
 print("값 변환에 실패하였습니다.")  
}  
  
  
// 옵셔널 자동 해제  
let optionalInt : Int? = Int(12345)  
  
if optionalInt != nil {  
 if (optionalInt!) == 12345 { // 옵셔널 강제 해제  
 print("optionalInt == 12345")  
 } else {  
 print("optionalInt != 12345")  
 }  
}  
  
if (optionalInt != nil) {  
 if optionalInt == 12345 { // 옵셔널 자동 해제  
 print("optionalInt == 12345")  
 } else {  
 print("optionalInt != 12345")  
 }  
}

1.5.함수

<== 자세히

* 함수의 정의
  + func 키워드를 사용하여 함수 선언
  + 함수의 이름에는 영어나 숫자, 특수 문자 등 다양한 문자들을 사용할 수 있지만 첫 글자는 반드시 영어 또는 언더바(\_)로 시작 해야 함
  + 함수의 인자값은 기본적으로 함수 내부에서 변수가 아니라 상수로 선언된다. 함수의 인자값을 변수로 사용하고자 할 경우 매개변수 선언 시 var로 선언해주시면 된다.   
    고... 하는데.. Swift3 부터는 안되는것 같다. ㅡㅡㅋ

func 함수명(인자값1:타입, 인자값2:타입, ...) -> 반환값 {  
 실행내용  
 return 반환값  
}

import UIKit

// 리턴값이 없는 함수의 선언과 호출1

func sayHello() {

print("안녕하세요. Hello ^^")

}

sayHello()

// 리턴값이 없는 함수의 선언과 호출2

func sayHi() -> Void {

print("안녕하세요. Hi ^^")

}

sayHi()

// 리턴값이 존재하는 함수의 선언과 호출1

func getUserNameById(userId : String) -> String {

var userName = ""

if userId == "dazzilove" {

userName = "류성희"

} else {

userName = ""

}

return userName

}

print(getUserNameById(userId: "dazzilove"))

// 인자가 n개인 경우의 함수 선언과 호출1

func sayHello(name : String, msg : String) -> Void {

print("\(name)님 \(msg)")

}

sayHello(name: "류성희", msg: "안녕하세요")

// 내부 매개변수와 외부 매개변수

// 함수가 호출될 때 인자값의 목적을 확실하게 표현해주기 위해 외부 매개변수명을 호출해야한다.

// message msg : String ==> message = 외부 매개변수, msg = 내부 매개변수

func sayHello(name : String, message msg : String, visitCount count : Int) {

print("\(name)님 \(msg). \(count)번째 접속 입니다.")

}

sayHello(name:"무지랑", message:"매우 반갑습니다", visitCount:3)

// 매개변수의 기본값

func sayHelloNoMessageDefaultSet(name : String, msg : String = "Hi") {

print("\(name)님 \(msg)")

}

sayHelloNoMessageDefaultSet(name: "류성희")

sayHelloNoMessageDefaultSet(name: "류성희", msg: "Hello")

// 튜플 반환

func getPersonalInfo() -> (String, Int) {

let name = "류성희"

let age = 38

return (name, age)

}

let dazziloveInfo = getPersonalInfo()

print("이름 = \(dazziloveInfo.0), 나이 = \(dazziloveInfo.1)")

let (dazziloveName, dazziloveAge) = getPersonalInfo()

print("이름 = \(dazziloveName), 나이 = \(dazziloveAge)")

// 가변 인자값

func sum(score : Int...) -> Int {

var sum = 0;

for row in score {

sum = sum + row

}

return sum

}

var scoreSum = sum(score : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(scoreSum)

// inout 매개변수

// 상수는 inout 매개변수에 인자값으로 전달 할 수 없음

// 리터럴은 inout 매개변수에 인자값으로 전달 할 수 없음

// 변수는 inout 매개변수에 인자값으로 전달 할 수 있음

// 함수호출 시 매개변수 앞에 &를 표기해줘야 한다.

// 주소를 직접 전달하는 '참조에 의한 전달'은 inpout 키워드를 사용했을 때 적용된다.

// 예외적으로 클래스(Class) 인스턴스는 inout 키워드를 사용하지 않아도 참조에 의한 전달 방식이 적용된다. 때문에 함수의 인자값으로 전달한 클래스 인스턴스는 함수 내부에서 값이 수정되면 원본 객체에도 영향을 미치므로 주의해야 한다.

var count = 20

func countPlus( paramCount : inout Int) -> Int {

return paramCount + 5

}

print(countPlus(paramCount: &count))

print(count + 20) // <========= 이거 안되는데 ㅠㅡ... 다시 확인해보자.

1.6.일급함수(First-Class Function)

* 일급객체(Fist-Class Object)
  + 크리스토퍼 스트래치라는 영국의 과학자가 1960년대에 처음 사용한 개념으로서, 프로그램 언어 안에서 특정 종류의 객체가 일깁의 지위를 가지는가에 대한 의미이다.
* 객체가 다음의 조건을 만족하는 경우 이 객체를 일급 객체로 간주한다.
  + 객체가 런타임에도 생성이 가능해야 한다.
  + 인자값으로 객체를 전달할 수 있어야 한다.
  + 반환값으로 객체를 사용할 수 있어야 한다.
  + 변수나 데이터 구조 안에 저장할 수 있어야 한다.
  + 할당에 사용된 이름과 관계없이 고유한 구별이 가능해야 한다.
* 일급 함수의 특성
  + 변수나 상수에 함수를 할당 할 수 있음
    - 지금까지는 함수를 호출 할 때 함수의 이름 다음에 함수 호출 연산자를 붙어야 했다.
    - 하지만 일급 함수의 특성에 따라 굳이 함수의 이름이 아니더라도 함수가 할당된 변수라면 그 변수에 함수 호출 연산자 ()를 붙여서 함수를 호출 할 수 있다.
    - 변수나 상수에 함수를 할당할 때는 함수가 실행되지 않은 상태로 함수라는 객체 자체만 할당 된다.

func foo(base:Int) -> String {

print("foo 함수가 실행 됩니다.")

return "결과값은 \(base)입니다."

}

var f = foo    // foo 함수가 실행 됩니다. 출력 안됨

f(20)          // foo 함수가 실행 됩니다. 출력 됨

* + - 변수나 상수에 함수의 결과값을 할당하면 변수나 상수의 타입은 함수의 반환 타입을 다르게 되지만, 함수 자체를 할당하는 경우 전혀 다른 타입이 되며, 이 타입을 함수 타입(Function Types)라고 한다. 함수의 형태에 따라 다양한 종류의 함수 타입이 있다.

func foo1(base:Int) -> String {

return "결과값은 \(base)입니다."

}

var f1 : (Int) -> String = foo1

f1(10)

func foo2(name:String, age:Int) -> String {

return "이름은 \(name), 나이는 \(age) 입니다.";

}

var f2 : (String, Int) -> String = foo2

f2("dazzilove", 38)

func foo3() {

print("인자도 반환값도 없는 함수 입니다.")

}

func foo4() -> Void {

print("이것도 인자도 반환값도 없는 함수 입니다.")

}

* + 함수의 반환타입으로 함수를 사용 할 수 있음

func 함수명(인자타입) -> 반환타입

==> 함수를 반환타입으로 사용할 경우

func 함수명(인자타입) -> (인자타입) -> 반환타입

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

return a + b

}

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

return a - b

}

func cal(operand : String) -> (Int, Int) -> Int {

switch operand {

case "+":

return plus

case "-":

return minus

default:

return plus

}

}

var c = cal(operand: "+")

c(5, 10) // 15

c = cal(operand: "-")

c(3, 2) // 1

cal(operand: "-")(10, 7) // 3

cal(operand: "+")(2, 8) // 10

* + 함수의 인자값으로 함수를 사용 할 수 있음

func success(arg: Int) -> String {

return "입력받은값 \(arg)에 대해... 처리가 성공 하였습니다."

}

func fail(arg: Int, addMessage msg : String) -> String {

return "입력받은값 \(arg)에 대해... 처리가 실패 하였습니다. \(msg)"

}

func callbackTest(courseNumber: Int, successCallBack success : (Int) -> String, failCallback fail : (Int, String) -> String) -> String {

if courseNumber == 1 {

success(courseNumber)

} else {

fail(courseNumber, "추가 메시지를 뿌려 줍니다.")

}

return "콜백테스트 완료"

}

callbackTest(courseNumber: 1, successCallBack: success, failCallback: fail)

// "입력받은값 1에 대해... 처리가 성공 하였습니다."

callbackTest(courseNumber: 2, successCallBack: success, failCallback: fail)

// "입력받은값 2에 대해... 처리가 실패 하였습니다. 추가 메시지를 뿌려 줍니다."

1.7.클로저

* 스위프트에서 클로저는 일회용 함수를 작성할 수 있는 구문이다.
* 일회용 함수는 한 번만 사용하면 되므로 굳이 함수의 이름을 작성할 필요없이 생략된다는 점에서 익명(Anonymous) 함수라고 부르기도 한다.
* 비슷한 넘 ==> 자바스크립트 익명 함수, 자바 람다 함수
* 클로저는 자신이 정의되었던 문맥(Context)로부터, 모든 상수와 변수의 값을 캡처하거나 레퍼런스를 저장하는 익명함수이다.
* 스위프트에서 클로저라고 부르는 객체는 대부분 다음 세 가지 경우 중 하나에 해당한다.
  + 전역함수
    - 이름이 있으며, 주변 환경에서 캡처할 어떤 값도 없는 클로저
  + 중첩함수
    - 이름이 있으며, 자신을 둘러싼 함수로부터 값을 캡처할 수 있는 클로저
  + 클로저 표헌식
    - 이름이 없으며 주변 환경으로부터 값을 캡처할 수 있는 경량 문법으로 작성된 클로저
* 클로저 표헌식

{ (매개변수) -> 반환타입 in

실행할 구문

}

\_ = { ( ) -> ( ) in

print("매개변수도, 반환값도 없는 클로저 표현식 입니다.")

}

\_ = { print("매개변수도, 반환값도 없는 클로저 표현식 입니다.") }

// 반환값이 없는 클로저 함수 호출

var f = { (name: String, age: Int) -> Void in

print("name is \(name), age is \(age).")

}

f("RyuSungHee", 38)

// 반환값이 존재하는 클로저 함수 호출

var hasReturnValueFunc = { (count: Int) -> Int in

return count

}

print("return value is \(hasReturnValueFunc(2)).")

// 클로저 표현식 호출

print("return value is \( ({ (count: Int) -> Int in return count })(4) )")

// 클로저 표현식 줄이기 - 반환값 추론이 가능한 경우 반환타입을 지정하지 않아도 된다.

var basicFunc1 = { (a: Int, b: Int ) -> Bool in

return a > b

}

print(basicFunc1(1, 2))

var basicFunc2 = { (a: Int, b: Int ) in

return a > b

}

print(basicFunc2(3, 2))

print( ({ (a: Int, b: Int) -> Bool in return a > b })(5, 6) )

print( ({ (a: Int, b: Int) in return a > b })(9, 6) )

// 클로저 표현식 줄이기 - 인자도 반환값도 없이 클로저 표현식을 실행 합니다.

({ ( ) in print("인자도 반환값도 없이 클로저 표현식을 실행 합니다. 1") })()

({ \_ in print("인자도 반환값도 없이 클로저 표현식을 실행 합니다. 2") })()

// 클로저 표현식과 경량 문법 예제

let value = [1, 5, 3, 7, 2, 8]

func order(a: Int, b: Int) -> Bool {

if(a > b) {

return true

} else {

return false

}

}

value.sorted(by: order) // 8, 7, 5, 3, 2, 1

value.sorted(by: { (a: Int, b: Int) -> Bool in

if(a > b) {

return true

} else {

return false

}

})

value.sorted(by: { (a: Int, b: Int) -> Bool in

return a > b

})

value.sorted(by: { (a: Int, b: Int) -> Bool in return a > b })

value.sorted(by: { (a: Int, b: Int) in return a > b }) // 반환타입에 대해 타입추론

value.sorted(by: { a, b in return a > b }) // 매개변수에 대해 타입추론

value.sorted(by: { return $0 > $1 }) // 매개변수 생략. 매개변수명 대신 $0, $1 과 같은 이름으로 하당 된 내부 상수를 이용할 수 있음

value.sorted(by: { $0 > $1 }) // 컴파일러가 반환타입을 근거로 왼쪽과 같은 구문도 추론하여 컴파일 가능 함

1.6.구조체와 클래스

* 구조체와 클래스의 정의

// 구조체

struct 구조체이름 {

구조체 정의 구문

}

// 클래스

class 클래스이름 {

클래스 정의 구문

}

* 구조체와 클래스의 공통점
  + 값을 저장하는 속성을 변수나 상수로 정의할 수 있다.
  + 함수적인 기능을 제공하는 메소드(Method)를 정의할 수 있다.
  + 속성값에 접근할 수 있는 방법을 제공하는 서브스크립트(Subscript)를 정의할 수 있다.
  + 객체를 원하는 초기 상태로 설정하게 하는 초기화 블록을 정의할 수 있다.
  + 기본 형태로 구현된 객체를 기반으로 함수적 기능을 확장할 수 있는 확장(extends) 기능을 제공한다.
  + 특정 형식의 함수적 표준을 제공하기 위한 프로토콜을 구현할 수 있다.
* 클래스만이 가능한 기능
  + 클래스의 특성을 다늘 클래스가 물려받아 그대로 사용할 수있는 상속이 가능하다.
  + 실행 시 컴파일러가 클래스 인스턴스의 타입을 미리 파악하고 검사할 수 있도록 타입 캐스팅을 제공한다.
  + 클래스 인스턴스에 할당된 리소스를 인스턴스가 소멸할 때 직접 해제할 수 있도록 소멸화 구문을 제공한다.
  + 클래스 인스턴스는 다른 객체로 전달될 때 레퍼런스 참조 형식으로 제공되며, 이때 레퍼런스 참조가 가능한 개수는 제약이 없다.
* 구조체와 클래스의 값 전달 방식
  + 구조체 = 값에 의한 전달
  + 클래스 = 참조에 의한 전달
    - 동일 인스턴스인지 비교할 때 ===
    - 동일 인스턴스가 아닌지 비교할 때 !==

struct Attribute {

var kind = ""

var color = ""

func desc() {

print("인형속성 구조체 입니다.")

}

}

class DogDoll {

var name = ""

var attribute : Attribute = Attribute()

func desc() {

print("강아지인형 클래스 입니다.")

}

}

var defaultAttribute = Attribute()

defaultAttribute.color = "brown"

defaultAttribute.kind = "golden retriever"

let dogDoll = DogDoll()

dogDoll.name = "mong"

dogDoll.attribute = defaultAttribute

print(dogDoll.attribute.color) // brown

print(dogDoll.attribute.kind) // golden retriever

// 구조체는 인스턴스를 생성한 후 이를 변수나 상수에 할당하거나 함수의 인자값으로 전달할 때 값을 복사하여 전달 한다.

defaultAttribute.color = "white"

print(defaultAttribute.color) // white

print(dogDoll.attribute.color) // brown

// 클래스는 값의 복사에 의해 전달되는 구조체와 달리 메모리 주소 참조에 의한 전달 방식을 사용한다. 이를 참조타입(Reference Type) 이라고 한다.

// 참조타입은 변수나 상수에 할당될 때, 또는 함수의 인자값으로 전달될 때 값의 복사가 이루어지지 않는다.

// 대신, 현재 존재하는 인스턴스에 대한 참조가 전달 된다.

class Person {

var doll : DogDoll = DogDoll()

var name = "SiHoo"

var age = 8

func desc() {

print("안녕하세요. \(name)입니다. 제가 가진 강아지 인형의 이름은 \(doll.name)이며 색깔은 \(doll.attribute.color)입니다.")

}

}

let person = Person()

person.doll = dogDoll

person.desc() // 안녕하세요. SiHoo입니다. 제가 가진 강아지 인형의 이름은 mong이며 색깔은 brown입니다.

dogDoll.name = "kong"

person.desc() // 안녕하세요. SiHoo입니다. 제가 가진 강아지 인형의 이름은 kong이며 색깔은 brown입니다.

* 클래스의 저장프로퍼티 lazy 키워드
  + 일반적으로 저장 프로퍼티(var, let)는 클래스 인스턴스가 처음 생성될 때 함께 초기화되지만, 저장 프로퍼티 정의 앞에 lazy라는 키워드가 붙으면 예외이다.
  + lazy 키워드가 붙은 저장 프로퍼티는 호출되기 전까지는 초기화되지 않는다. (선언만 될 뿐 초기화 되지 않는다)

class DumpClass {

init() {

print("DumpClass is init")

}

}

class LazyTest {

lazy var dumpClass = DumpClass()

init() {

print("LazyTest is init")

}

}

let lazyTest = LazyTest() // DumpClass는 초기화 되지 않는다.

lazyTest.dumpClass // 이때에 비로서 DumpClass가 초기화 된다.

* 구조체 내부의 파라미터 수정
  + 구조체와 열거형은 값 타입이며, 기본적으로 값 타입의 속성은 인스턴스 메서드 안에서 수정 할 수 없다.
  + 하지만 특정 메서드 안에서 구조체나 열거형의 속성 값을 변경하고자 할 필요가 있다면 메서드 앞에 mutating 키워드를 지정하여 수정할 수 있음

struct Point {

var x : Int = 0

var y : Int = 0

mutating func move(moveX: Int, moveY: Int) {

self.x = self.x + moveX

self.y = self.y + moveY

}

}

var point = Point()

print("x = \(point.x), y = \(point.y)") // x = 0, y = 0

point.move(moveX: 10, moveY: 20)

print("x = \(point.x), y = \(point.y)") // x = 10, y = 20

* 클래스 상속
  + 단일 상속만 지원 됨

struct Attribute {

var color = ""

func desc() {

print("인형속성 구조체 입니다.")

}

}

class Doll {

var type = ""

var name = ""

var attribute : Attribute = Attribute()

func desc() {

print("인형 클래스 입니다.")

}

}

class CatDoll : Doll {

func wolf() -> String {

return "Mew"

}

override func desc() {

print("\(type)인형입니다. 색깔은 \(attribute.color)이구요. \(wolf())라고 울어요.")

}

}

class DogDoll : Doll {

func wolf() -> String {

return "Bow-wow"

}

override func desc() {

print("\(type)인형입니다. 색깔은 \(attribute.color)이구요. \(wolf())라고 울어요.")

}

}

// 고양이 인형

let catDoll = CatDoll()

catDoll.type = "CAT"

catDoll.name = "NURI"

var catDollAttribute = Attribute()

catDollAttribute.color = "yellow"

catDoll.attribute = catDollAttribute

catDoll.desc() // CAT인형입니다. 색깔은 yellow이구요. Mew라고 울어요.

print(catDoll.wolf()) // Mew

// 강아지 인형

let dogDoll = DogDoll()

dogDoll.type = "DOG"

dogDoll.name = "MONG"

var dogDollAttribute = Attribute()

dogDollAttribute.color = "white"

dogDoll.attribute = dogDollAttribute

dogDoll.desc() // DOG인형입니다. 색깔은 white이구요. Bow-wow라고 울어요.

print(dogDoll.wolf()) // Bow-wow

* + 타입 비교
    - 타입 비교 연산자 = is ==> SUV() is Car // true, Car() is SUV // false
    - 타입 비교 시.. 연산자 왼족에 인스턴스가 아니라 인스턴스가 할당 된 변수가 사용될 경우 주의가 필요하다. 변수가 선언된 타입을 기준으로 비교하는것이 아니라 변수에 할당된 실제 인스턴스를 기준으로 타입을 비교하기 때문이다.

class Fruit {

func desc() { print("This is Fruit") }

}

class Apple : Fruit {

override func desc() { print("This is Apple") }

}

class Banana : Fruit {

override func desc() { print("This is Banana") }

}

Apple() is Fruit // true

Banana() is Fruit // true

Fruit() is Fruit // true

let myFruit : Fruit = Apple()

if myFruit is Apple {

print("myFruit는 Apple 타입 입니다.")

} else {

print("myFruit는 Fruit 타입 입니다.")

}

// Fruit 타입으로 선언되었으나 Apple로 생성되었기에... Apple 타입으로 비교된다.

* + 타입 캐스팅 가능 함
    - 업 캐스팅
      * 객체 as 변환할타입
    - 다운 캐스팅
      * 객체 as? 변환할타입(옵셔널 타입으로 반환)
      * 객체 as! 변환할타입(일반 타입으로 반환)

class Car {

func desc() {

print("CAR Class")

}

}

class SUV : Car {

override func desc() {

print("SUV Class")

}

}

let suv = SUV()

let suvToCar = suv as Car

suvToCar.desc()

let car = Car()

let carOptinalDownCasting = suvToCar as? SUV

if carOptinalDownCasting != nil {

print("\(carOptinalDownCasting!) 다운 케스팅 성공 [1]")

} else {

print("다운 케스팅 실패")

}

let carDownCasting = suvToCar as! SUV

print("\(carDownCasting) 다운 캐스팅 성공 [2]")

* + Any, AnyObject
    - Any, AnyObject는 범용 타입이라고 설명할 수 있다.
    - AnyObject
      * 클래스의 일종이며, 클래스 타입에 상관없이 모든 종류의 클래스 타이을 저장 할 수 있는 범용 타입의 클래스 이다.
      * 클래스 중에서 가장 추상화된 클래스이며, 상속 관계가 직접 성립하는 것은 아니지만 가장 상위 클래스라고 할 수 있다.
      * 모든 클래스의 인스턴스는 AnyObject 클래스 타입으로 선언된 변수나 상수에 할당할 수 있다.
    - Any
      * Any 역시 AnyObject와 비슷한 범용 객체이지만, 클래스에 국한되지 않고 스위프트에서 제공하는 모든 타입에 대한 범용성을 지원한다.
      * 클래스만 할당할 수 있는 AnyObject에 비해 Any는 클래스뿐만 아니라 구조체, 열거형, 함수의 타입도 포함할 수 있다.

// AnyObject

class Apple {

func desc() {

print("This is Apple")

}

}

let apple : AnyObject = Apple()

(apple as! Apple).desc()

class Banana {

func desc() {

print("This is Banana")

}

}

let banana : AnyObject = Banana()

(banana as! Banana).desc()

var listAnyObject = [AnyObject]()

listAnyObject.append(Banana())

listAnyObject.append(Apple())

// listAnyObject.append("ddd") <== 에러!! 클래스만 AnyObject에 대입할 수 있다.

// Any

var listAny = [Any]()

listAny.append(Banana())

listAny.append("ddd")

listAny.append(12345)

listAny.append(1.0)

* 초기화 구문
  + init(), init(매개변수1, 매개변수2, ...)
  + 초기화 블록에서 정의된 매개변수는 함수에서의 외부 매개변수명까지 그 역할을 겸한다. 예) init(a:Int, b:Int) ==> a paramA: Int 와 같이 사용하지 않음
  + init 구문이 작성되지 않으면 구조체와 클래스는 빈 인자값 형식의 초기화 구문을 기본으로 제공한다.
  + 만약 init 구문이 일단 작성되면 작성된 init 구문이 어떤 인자값 형식을 갖는가에 상관없이 그 객체의 기본 초기화 구문은 더이상 제공되지 않는다.
  + 클래스에서 옵셔널 타입으로 지정되지 않은 모든 저장 프로퍼티는 초기값을 가져야 한다.

class dump {

var name : String = "초기값셋팅"

var age : Int =

var car : Car? // 옵셔널 타입으로 초기값을 지정하지 않아도 된다.

}

* + 초기화 구문 오버라이딩
    - 클래스에서는 초기화 구문도 상속 대상에 포함되므로 자식 클래스에서 오버라이딩 할 수 있다. ==> override init()
    - 초기화 구문을 오버라이딩하면 더 이상 부모 클래스에서 정의한 초기화 구문이 실행되지 않는다.
    - 초기화 구문을 오버라이딩할 경우 부모 클래스에서 정의된 초기화 구문을 내부적으로 호출할 수 있다. ==> override init(param:type) {  super.init(param:type) }
  + 초기화 구문 델리게이션(Initailizer Delegation)
    - 연쇄적으로 오버라이딩된 자식 클래스의 초기화 구문에서 부모 클래스의 초기화 구문에 대한 호출이 발생 하는 것
    - 기본 초기화 구문을 제외한 나머지 초기화 구문을 오버라이딩할 때는 반드시 부모 클래스의 초기화 구문을 호출함으로써 델리게이션 처리를 해 주어야 한다.
* ㄴㅇㄹㄴㄹ

1.7.프로토콜

* 다른 객체지향 언어에서의 인터페이스 역할을 함
* 프로토콜이란 특정 객체가 갖추어야 할 기능이나 속성에 대한 설계도이다.

protocal 프로토콜명 {

구현해야할 프로퍼티 명세 1

구현해야할 프로퍼티 명세 2  
 ....

구현해야할 메서드 명세 1

구현해야할 메서드 명세 1

....

}

* 프로퍼티 명세
  + 초기 값 할당 안됨
  + 프로퍼티의 종류, 이름, 변수/상수 구분, 타입, 읽기 전용인지 읽고 쓰기가 가능한지에 대해서만 정의 함

protocol SampleProperty {

var name : String { get, set } // 읽기, 쓰기 가능. 저장 프로퍼티의 경우 반드시 get, set 으로 정의해야 함

var age : Int { get } // 읽기만 가능

}

* 메서드 명세

protocol SampleMethod {

func execute(cmd: String)

func showPort(p: Int) -> String

}

* 구조체의 속성 변경 관련
  + 구조체에서 구현된 메소드의 실행 구문에서 프로퍼티를 수정해야 한다면 메서드 앞에 mutating 키워드를 추가해야 한다.
  + 해당 메소드가 프로토콜에서 선언된 메서드라면 프로토콜에 mutating 키워드가 먼저 작성 되어있어야 한다.
  + 클래스는 참조 타입의 객체이므로 메서드 내부에서 프로퍼티를 수정하더라도 mutating 키워드를 붙일 필요가 없다.

protocol SampleProtocol {

mutating func move(moveX: Int, moveY: Int)

func showPoint()

}

struct Point : SampleProtocol {

var x : Int = 0

var y : Int = 0

mutating func move(moveX: Int, moveY: Int) {

self.x = self.x + moveX

self.y = self.y + moveY

}

func showPoint() {

print("x = \(x), y = \(y)")

}

}

var point = Point()

point.showPoint() // x = 0, y = 0

point.move(moveX: 5, moveY: 7)

point.showPoint() // x = 5, y = 7

point.move(moveX: 5, moveY: 3)

point.showPoint() // x = 10, y = 10

* 클래스에서 상속과 프로토콜 정의
  + 스위프트는 단일 상속만을 허용한다.
  + 따라서.. 첫번째 인자가 클래스인 경우 상속할 클래스를 의미하고... 그 다음 인자 부터는 프로토콜을 의미한다.

class ClassA : ClassB { 블라블라 } // ClassB를 사용한다.

class ClassA : ClassB, ProtocolA { 블라블라 } // ClassB 상속, ProtocolA 구현

class ClassA : ProtocolA { 블라블라 } // ProtocolA 구현

class ClassA : ProtocolA, ProtocolB, ... { 블라블라 } // ProtocolA, ProtocolB, Protocol... 구현

class ClassA : ClassB, ProtocolA, ... { 블라블라 } // ClassB 상속, ProtocolA, Protocol... 구현

* 객체 생성 시 프로토콜을 구현한 메서드만 사용 가능하도록 제한 할 수 있음

protocol A { func doA() }

protocol B { func doB() }

class Impl : A, B {

func doA() { ... }

func doB() { ... }

func desc() { ... }

}

var impl : procotol<A, B> = Impl() // doA(), doB() 는 사용할 수 있지만, desc() 는 사용 할 수 없음

impl.doA()

impl.doB()

* 프로토콜의 상속

protocol A { func doA() }

protocol B { func doB() }

protocol C : A, B { func doC() }

class Impl : C {

func doA() { print("doA") }

func doB() { print("doB") }

func doC() { print("doC") }

}

* 클래스 전용 프로토콜
  + 프로토콜은 문법적으로 구조체에서 확장체에 이르기까지 광범위한 객체들이 구현 할 수 있다.
  + 때때로 클래스만 구현할 수 있는 프로토콜을 정의하고자 할 때 이를 "클래스 전용 프로토콜" 이라 한다.
  + 클래스 전용 프로토콜은  class 키워드를 프로토콜 이름 뒤 콜론으로 구분 된 영역에 위치한다.

protocol A : class { func doA() }

class Impl : A {

func doA() { print("doA") }

}

1.8.오류처리

import Foundation

enum DateParseError : Error {

case OverSizeString

case UnderSizeString

}

struct Date {

var year : Int

var month : Int

var date : Int

}

func parseDate(param: String) throws -> String {

let paramsLength = param.characters.count;

guard paramsLength == 8 else {

if paramsLength > 8 {

throw DateParseError.OverSizeString

} else {

throw DateParseError.UnderSizeString

}

}

let index = param.index(param.startIndex, offsetBy: 8)

return param.substring(to: index)

}

func getParseDate(dateStr: String) -> String {

var result : String = ""

do {

result = try parseDate(param: dateStr)

} catch DateParseError.OverSizeString {

print("입력된 문자열이 너무 깁니다. 줄여주세요.")

} catch DateParseError.UnderSizeString {

print("입력된 문자열이 너무 짧습니다. 늘려주세요.")

} catch {

print("알수없는 오류가 발생하였습니다.")

}

return result

}

getParseDate(dateStr: "20160105") // return 20160105

getParseDate(dateStr: "2016010500") // 입력된 문자열이 너무 깁니다. 줄여주세요.

getParseDate(dateStr: "201601") // 입력된 문자열이 너무 짧습니다. 늘려주세요.

# 2.스토리보드 사용법을 알아보기 (50분)

2.스토리보드 사용법을 알아보기

- [스토리보드 화면 구성](http://www.libqa.com/wiki/763)

- [iOS와 앱 생명주기](http://www.libqa.com/wiki/764)

- [버튼 생성 및 실행](http://www.libqa.com/wiki/758)

- [화면전환](http://www.libqa.com/wiki/759)

- 뷰 컨트롤러 간 데이터 교환

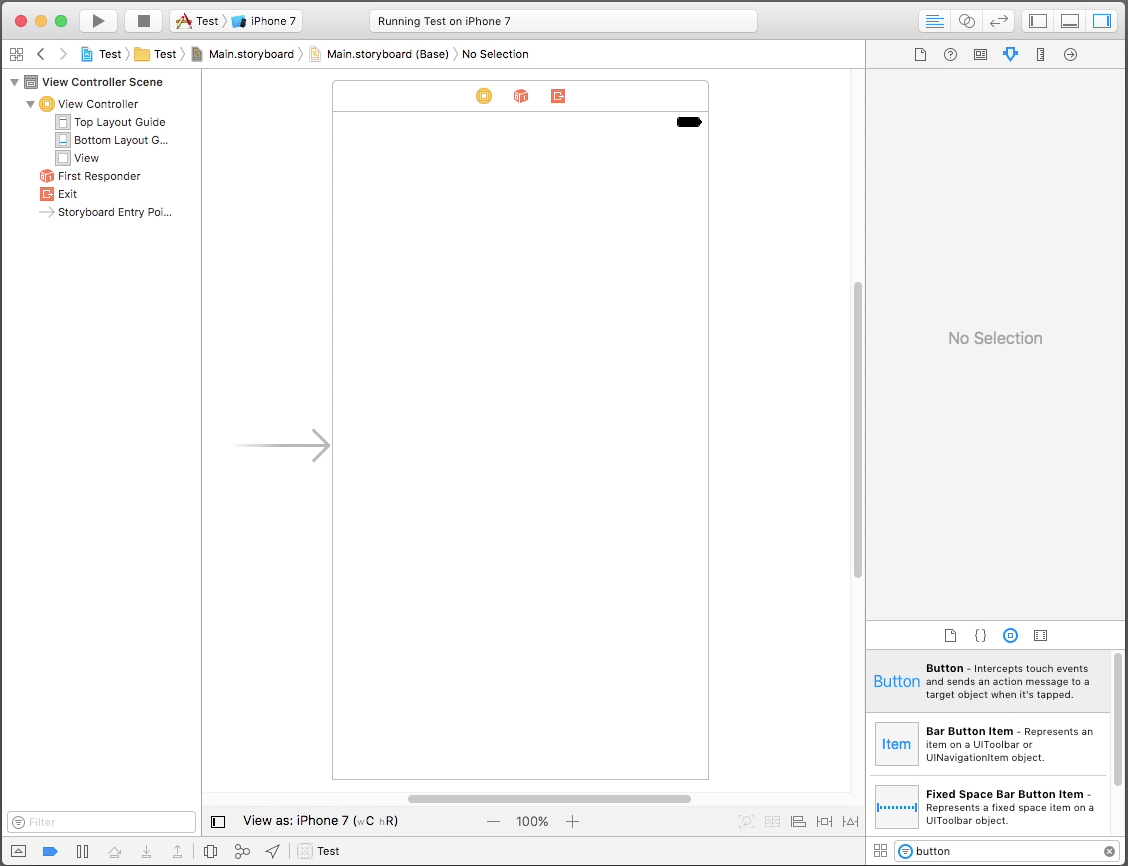
2.3.버튼 생성 및 실행

시작하기 전에...

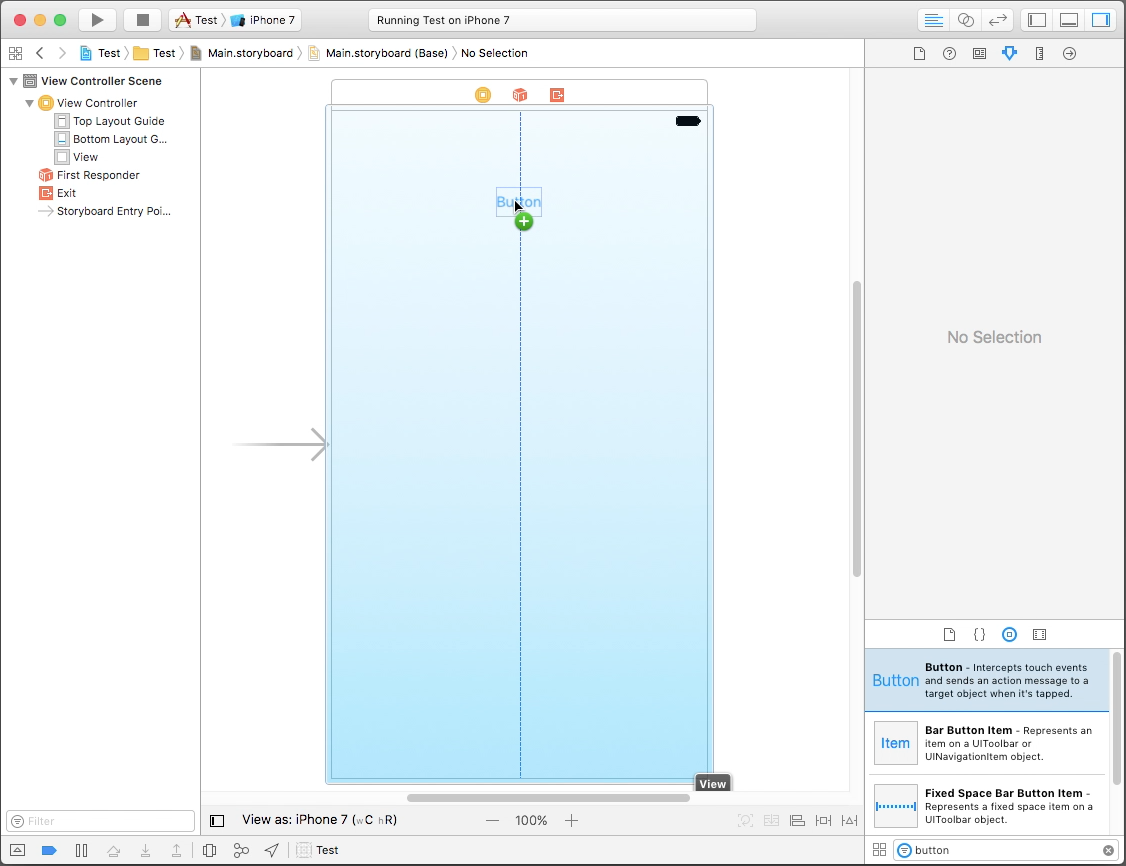
* 아웃렛 변수와 액션 함수
  + 스토리보드에 추가한 객체를 선택하고 내용을 변경하거나 특정 동작을 수행하도록 하기위해   
    해당 객체에 접근할 수 있는 변수인 '아웃렛 변수'와   
    동작을 정의한 함수인 '액션 함수'가 필요 하다.

버튼을 생성하고 실행해보자

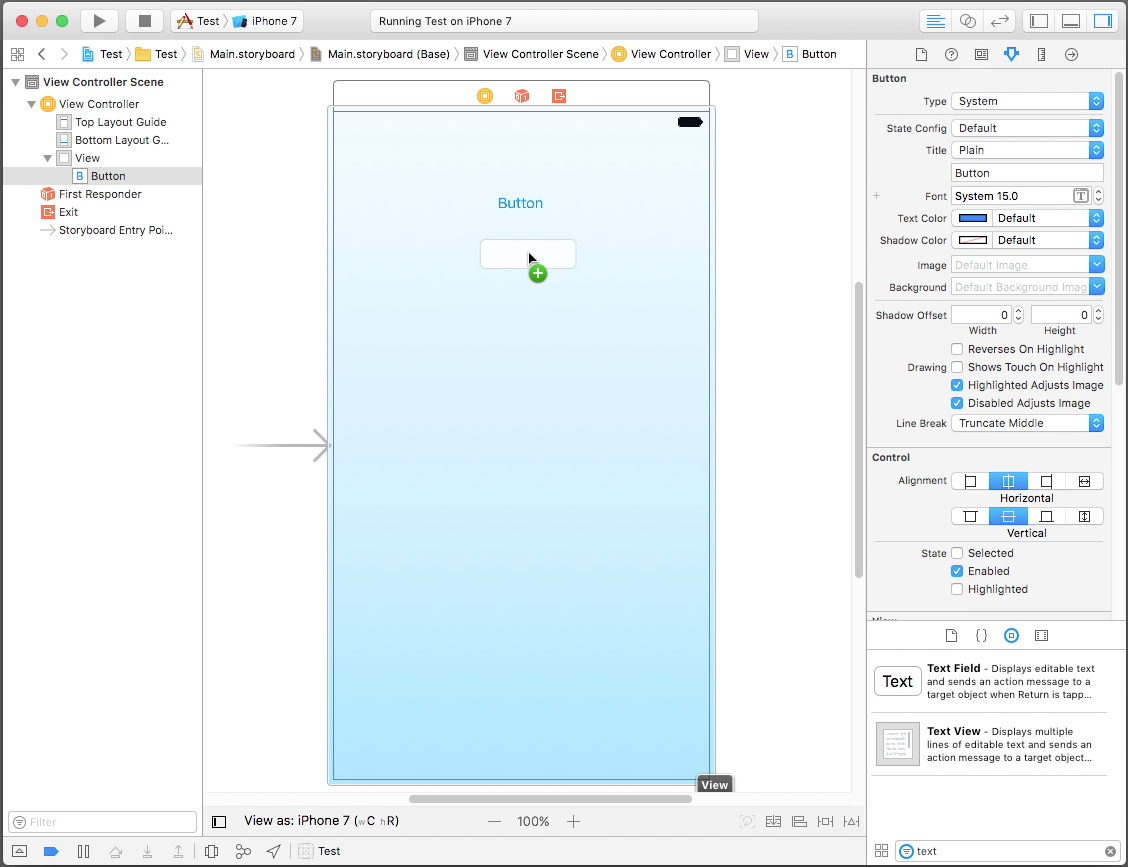
1.스위프트 프로젝트를 새로 생성하고 Main.storyboard 를 선택 합니다.



2.Object Lib 검색란에 "button"을 입력하여 버튼을 스토리보드로 drag&drop 합니다.

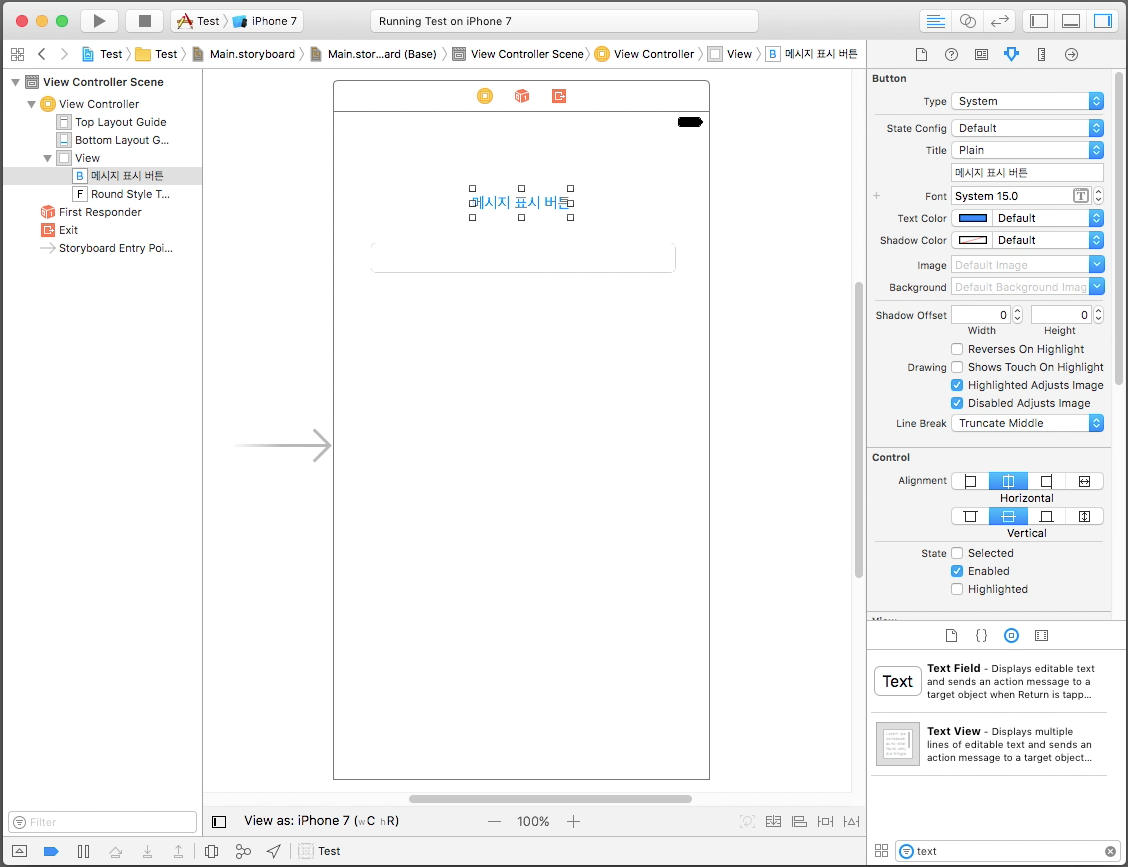


3.Object Lib 검색란에 "text"을 입력하여 버튼을 스토리보드로 drag&drop 합니다.

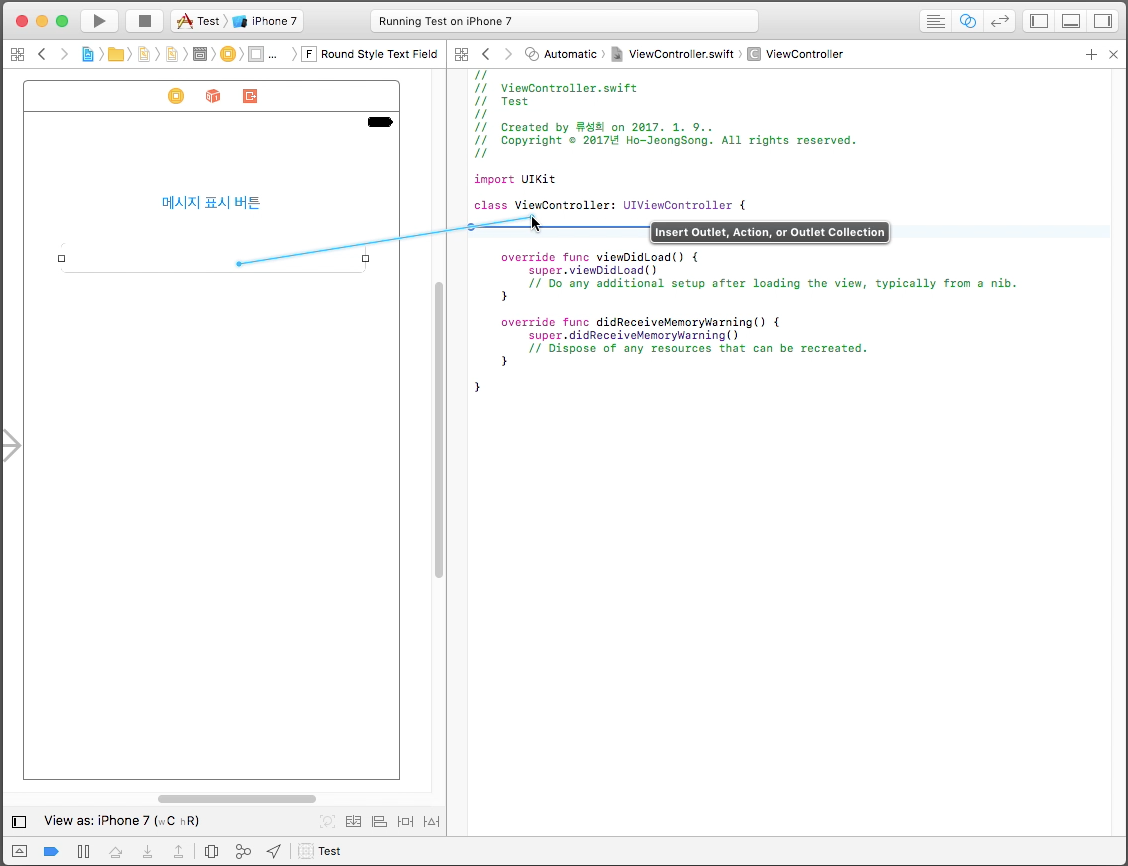


4.버튼을 더블클릭하여 표시할 텍스트를 "메시지 표시 버튼"으로 변경 합니다.

Attributes Inspector의 "Title"을 변경해도 됩니다.

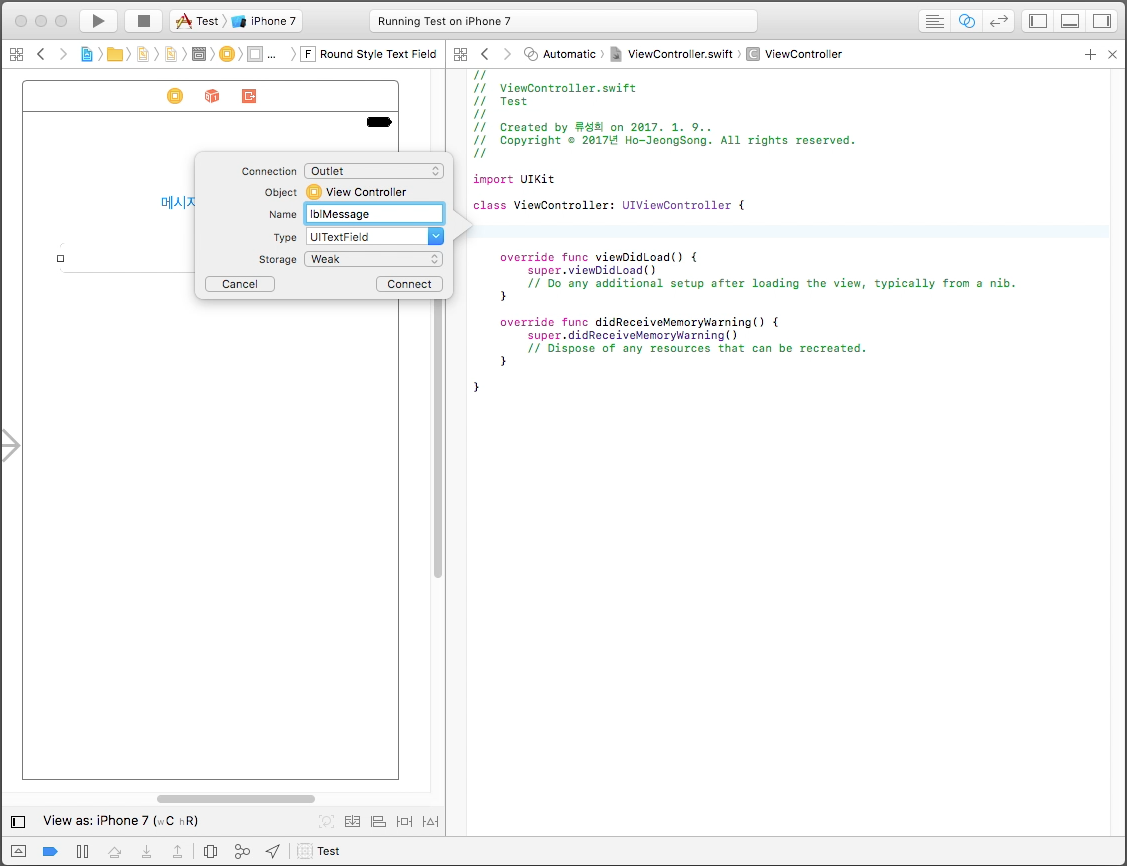


5.텍스트 필드에서 마우스 우클릭한 상태로 오른쪽의 코드편집창으로 drag&drop 합니다.

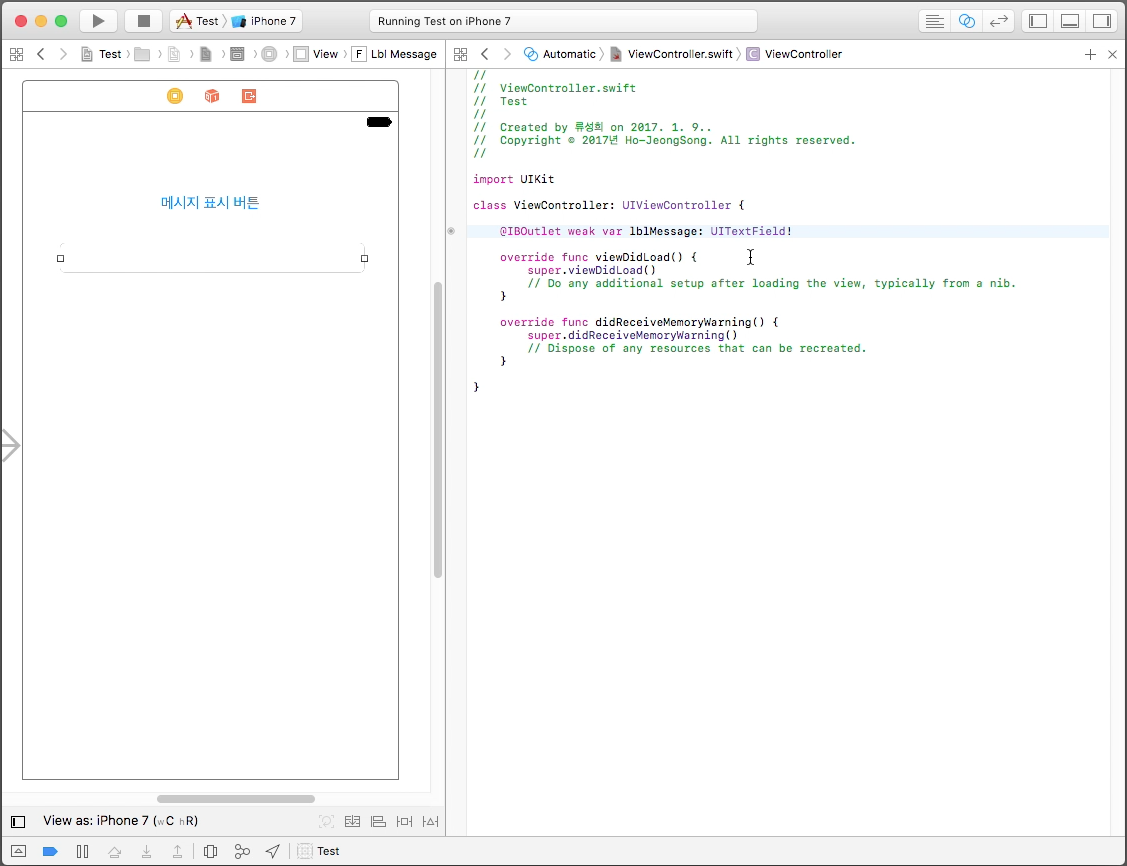


6.오른쪽 코드편집창에서 마우스를 때면 편집 툴이 표시됩니다.

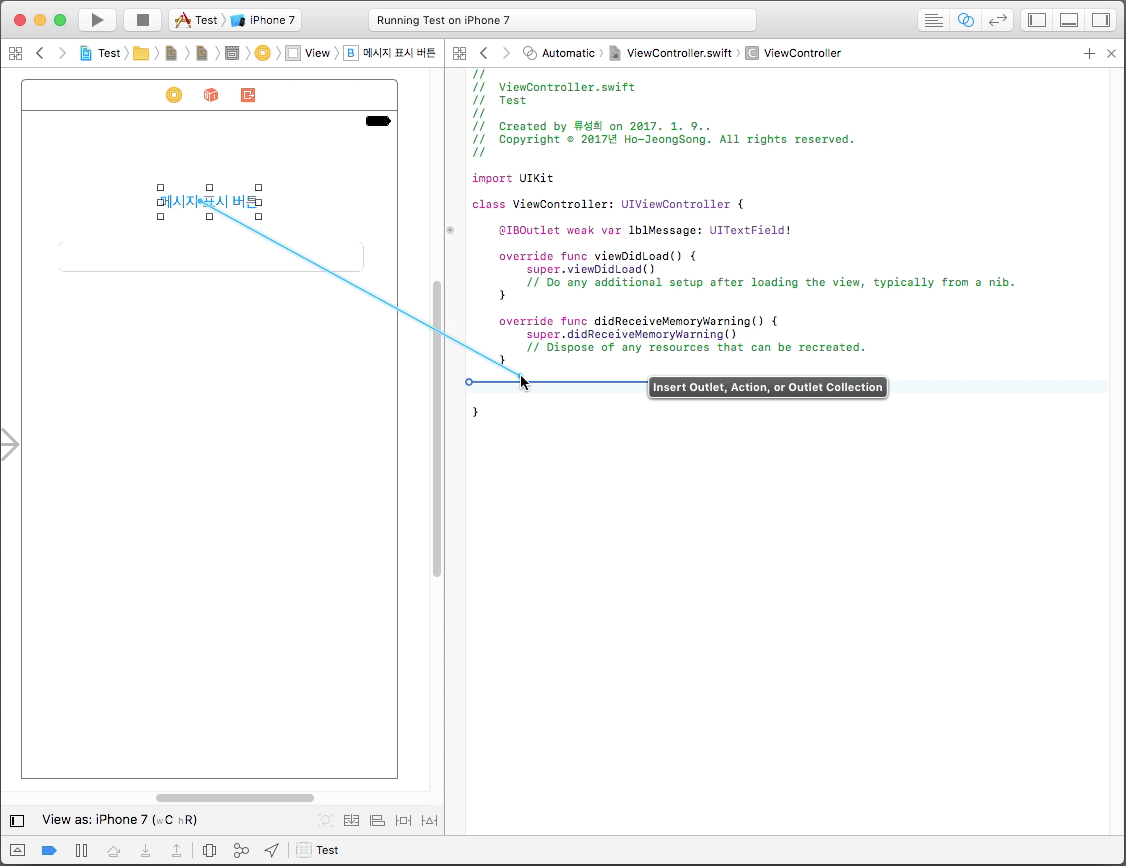
Connection = Outlet, Name = lblMessage, Type = UITextField, Storage = Weak 로 지정 후 Connect 버튼을 클릭 합니다.



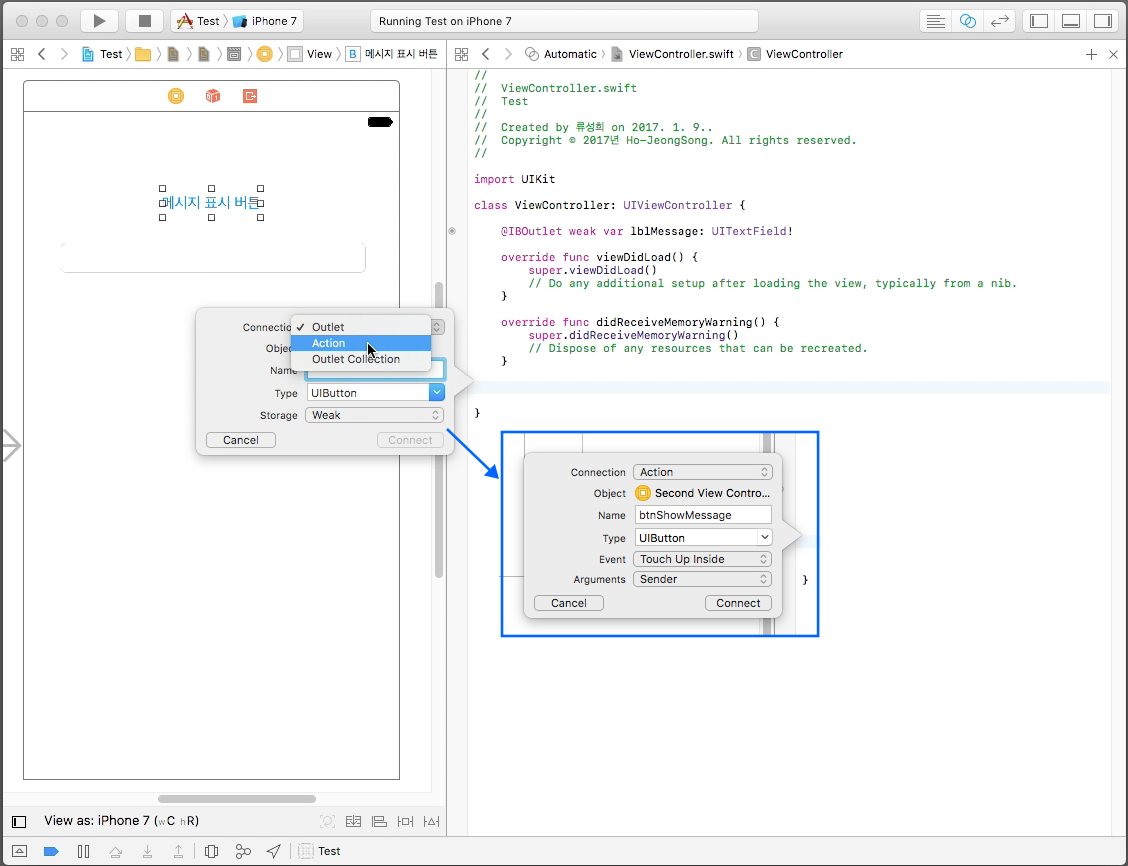
7.오른쪽 코드 편집창에 아웃렛 변수가 선언됩니다.



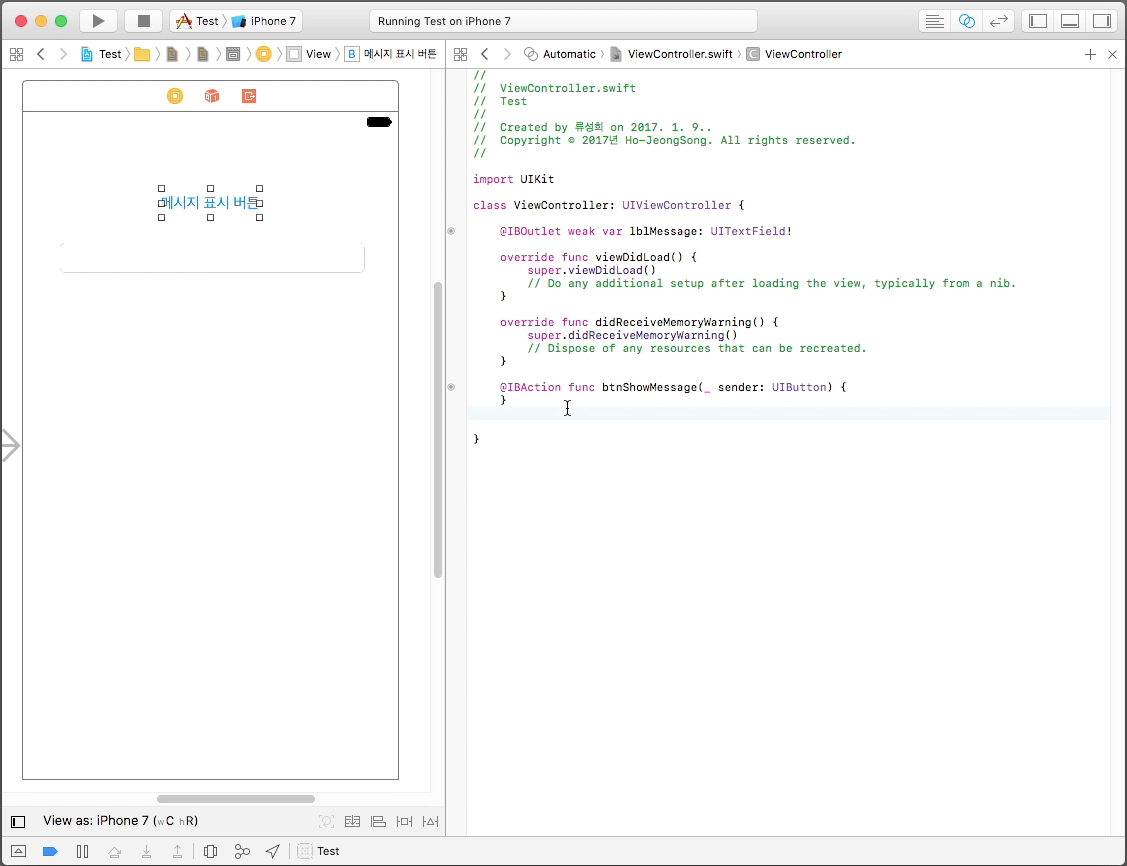
8.버튼의 Action 함수를 추가 합니다. 버튼을 우클릭 한 상태로 코드 편집창으로 드래그 합니다.



9.표시 창에서 Connection = Action, Name = btnShowMessage, Type = UIButton을 입력 후 Connect 버튼을 클릭 합니다.

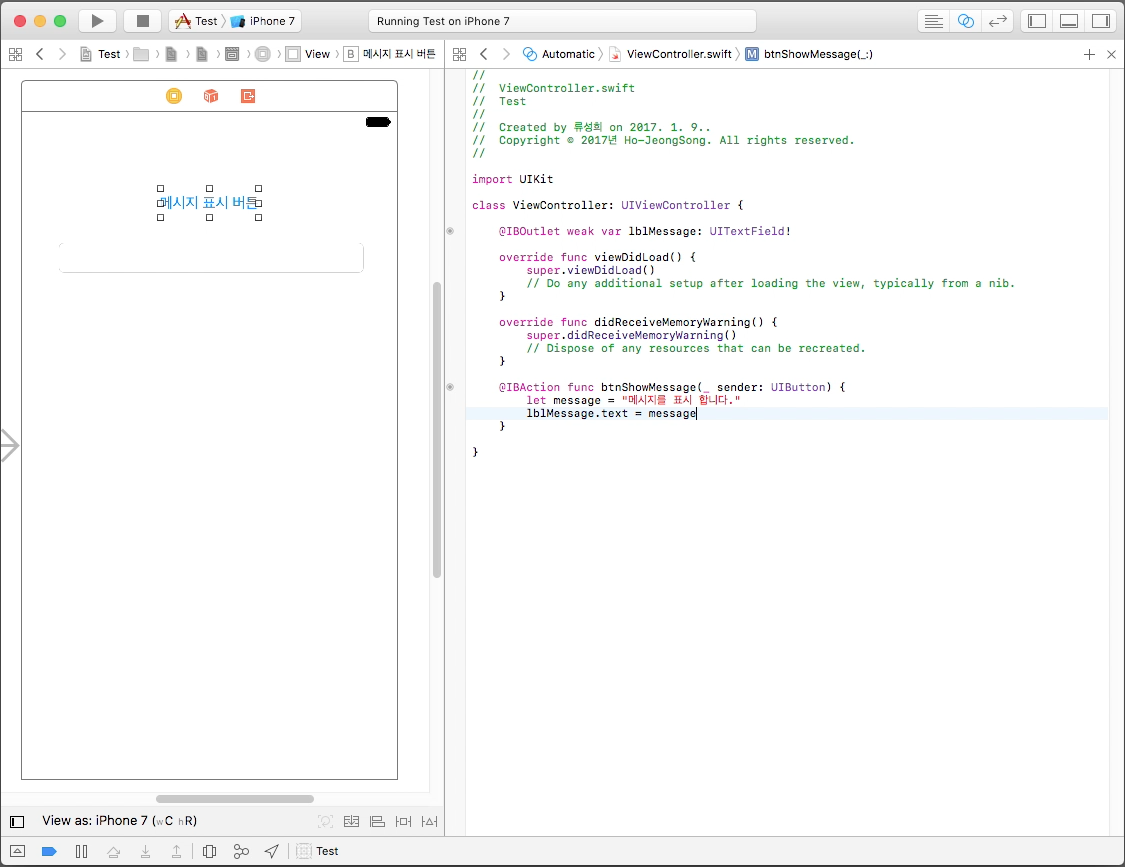


10.오른쪽 코드 편집창에 btnShowMessage 함수가 생성 되었습니다.



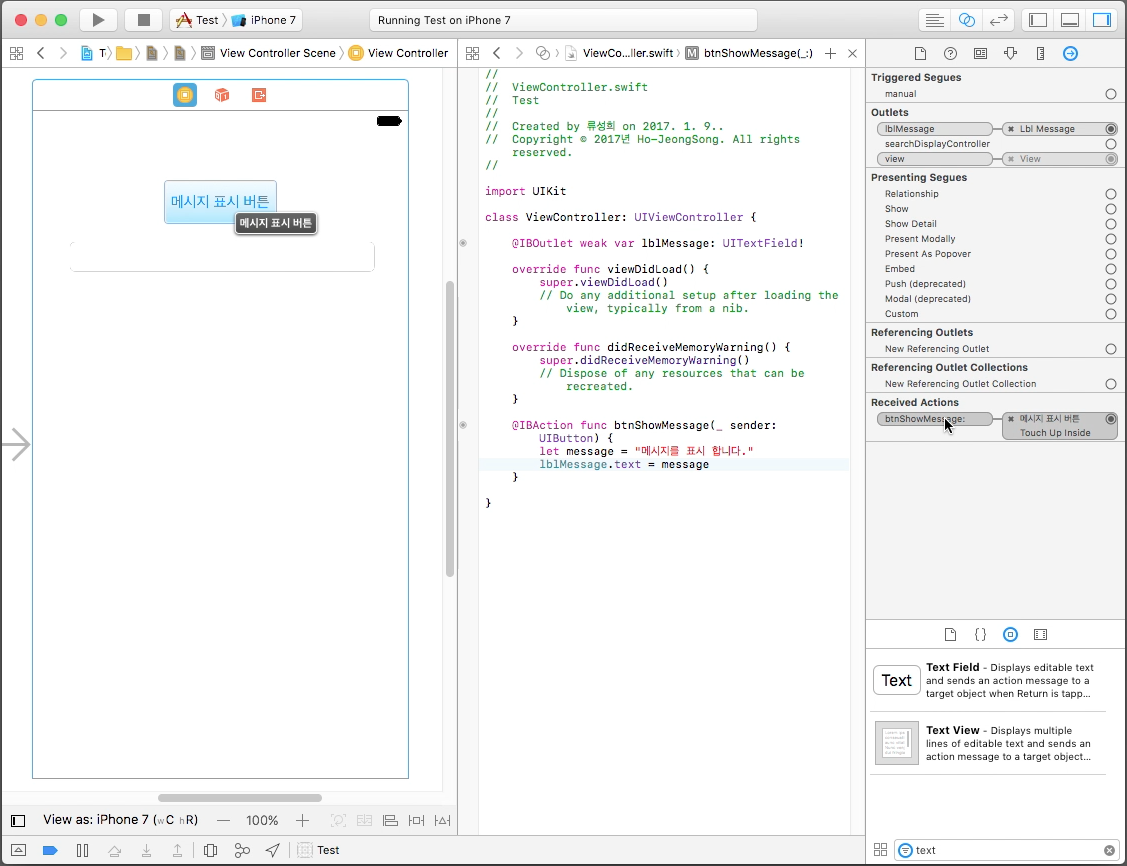
11.생성된 btnShowMessage() 함수에 동작을 선언 합니다.

하단 텍스트 필드에 메시지를 표시하도록 수정하였습니다.

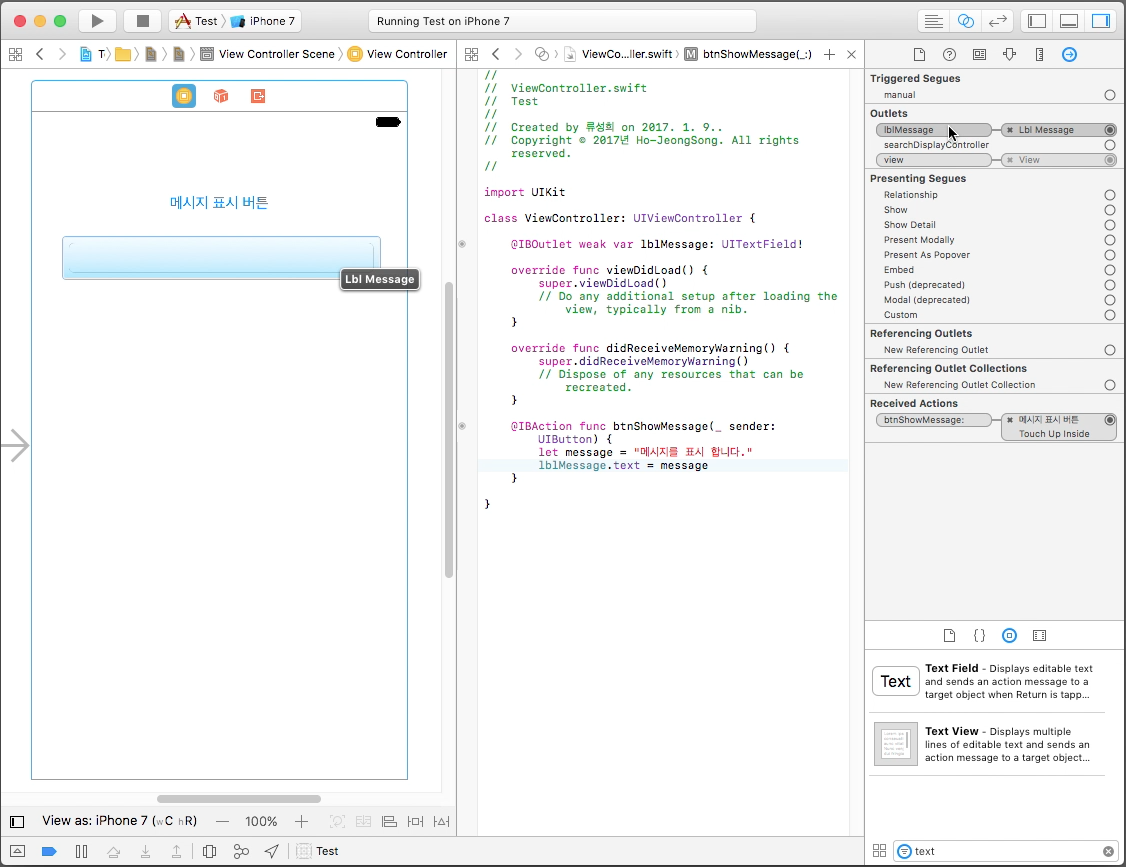


12.왼쪽 스토리보드에서 View Controller를 선택하고 오른쪽 Connections Inspector의 정보를 확인 합니다.

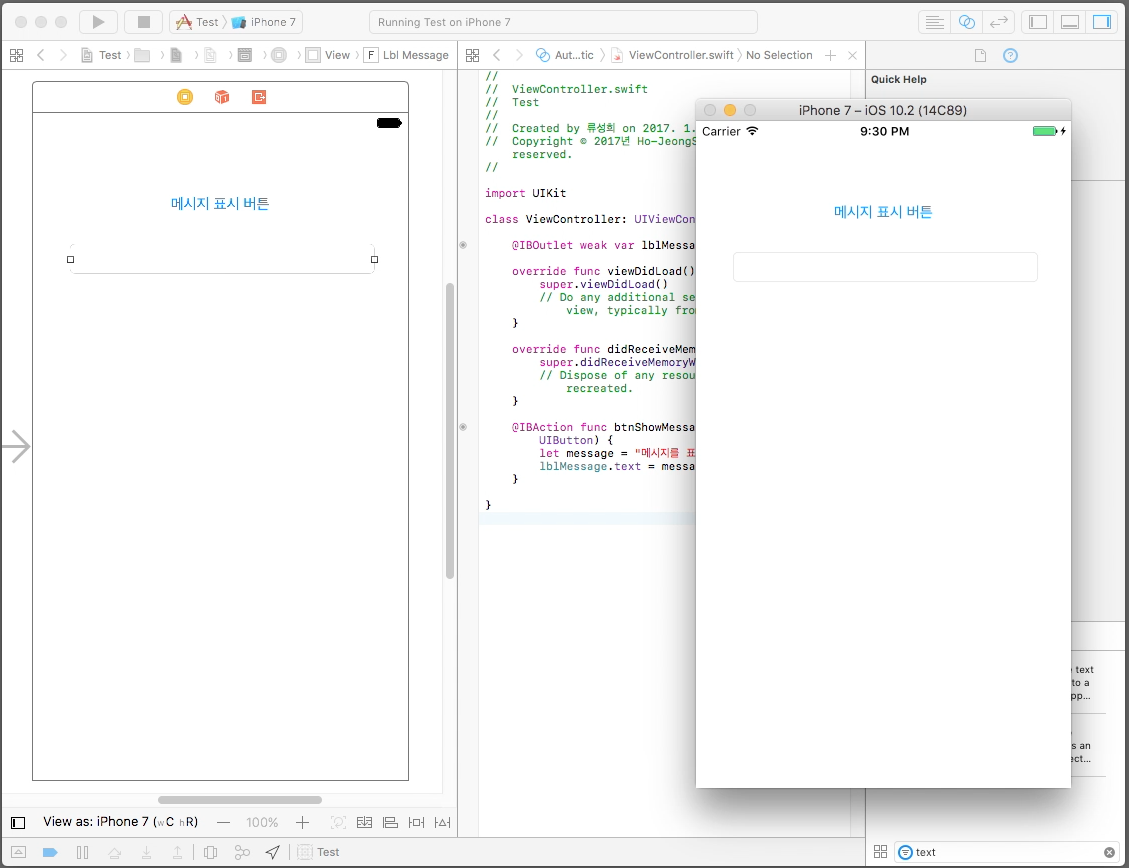
버튼에 대해 Touch Up Inside Action이 연결되어 있는것을 확인 할 수 있습니다.



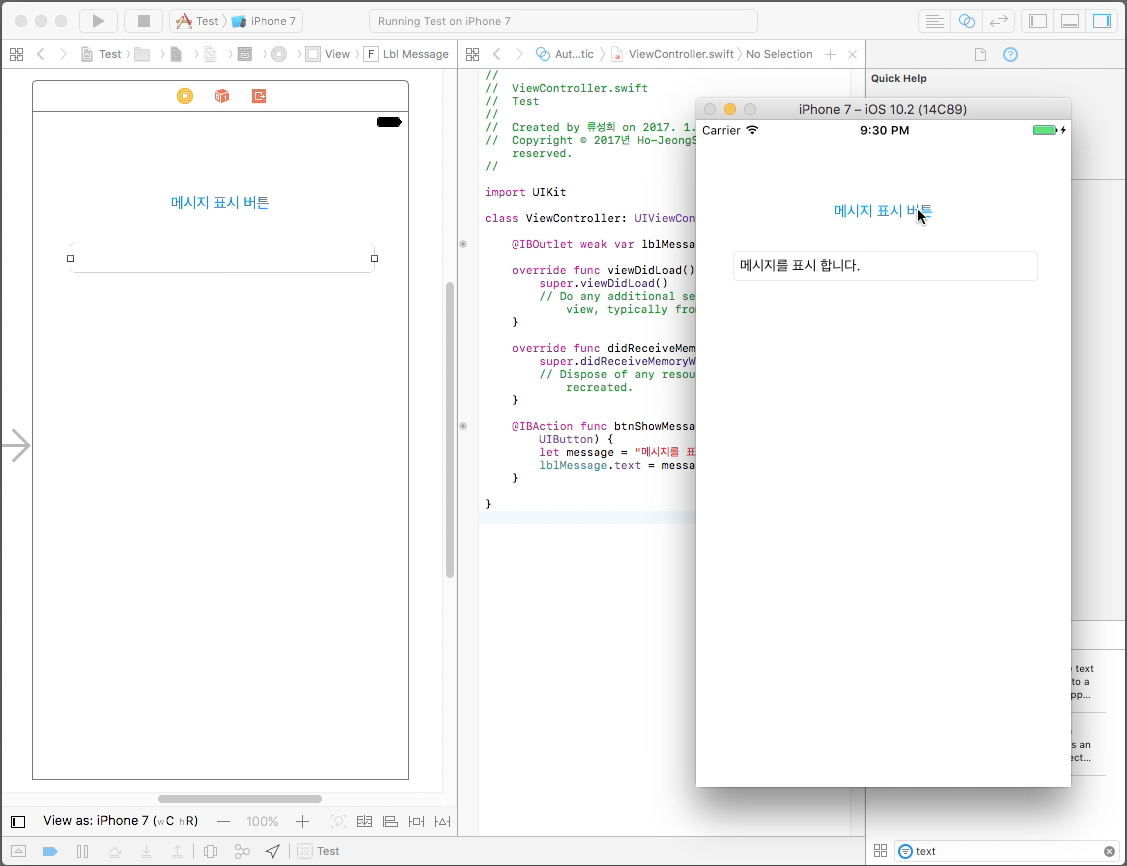
13.동일하게 스토리보드의 텍스트 필드가 Outlet 변수로 연결되어 있는것을 확인 합니다.



14.시뮬레이터를 실행 하고... 초기 로딩된 화면을 확인 합니다.



15.버튼을 클릭하여 텍스트 필드에 메시지가 노출되는것을 확인 합니다.

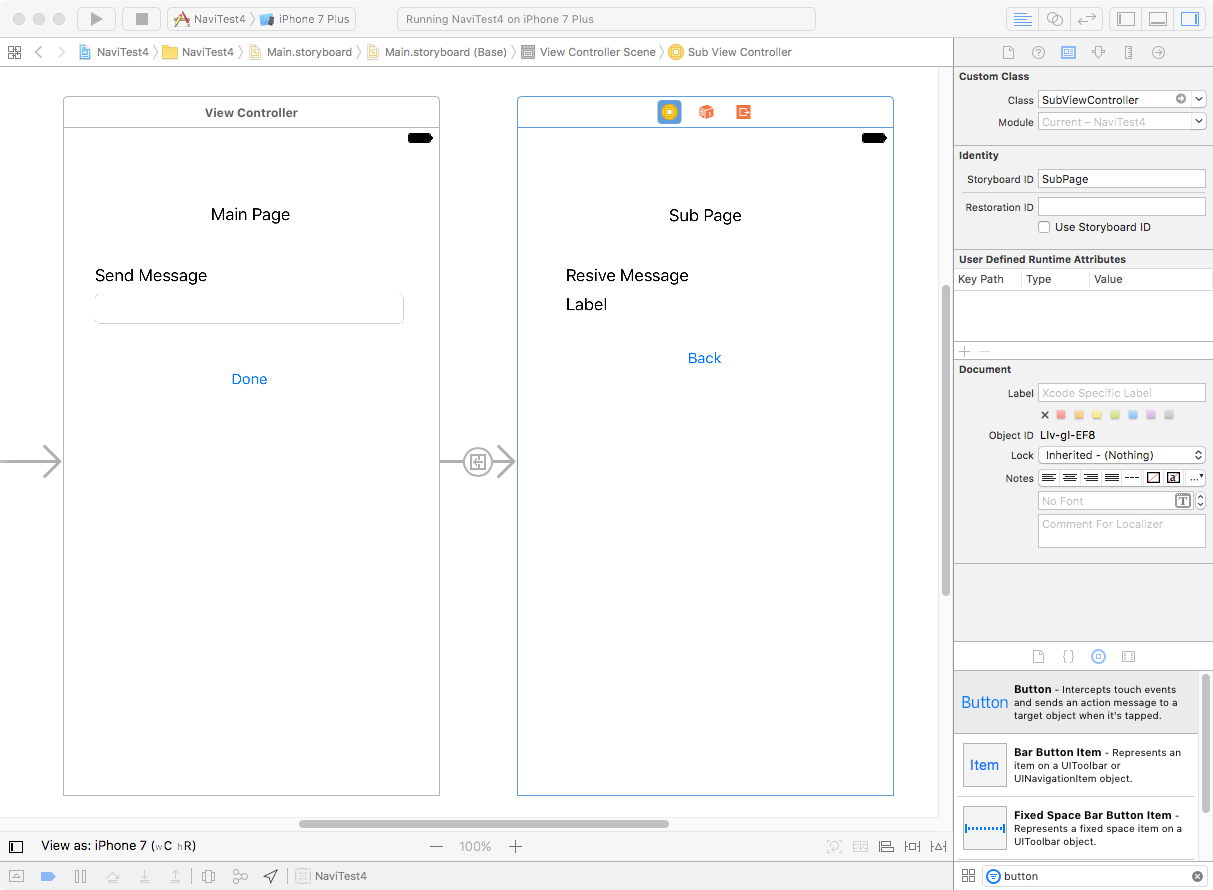


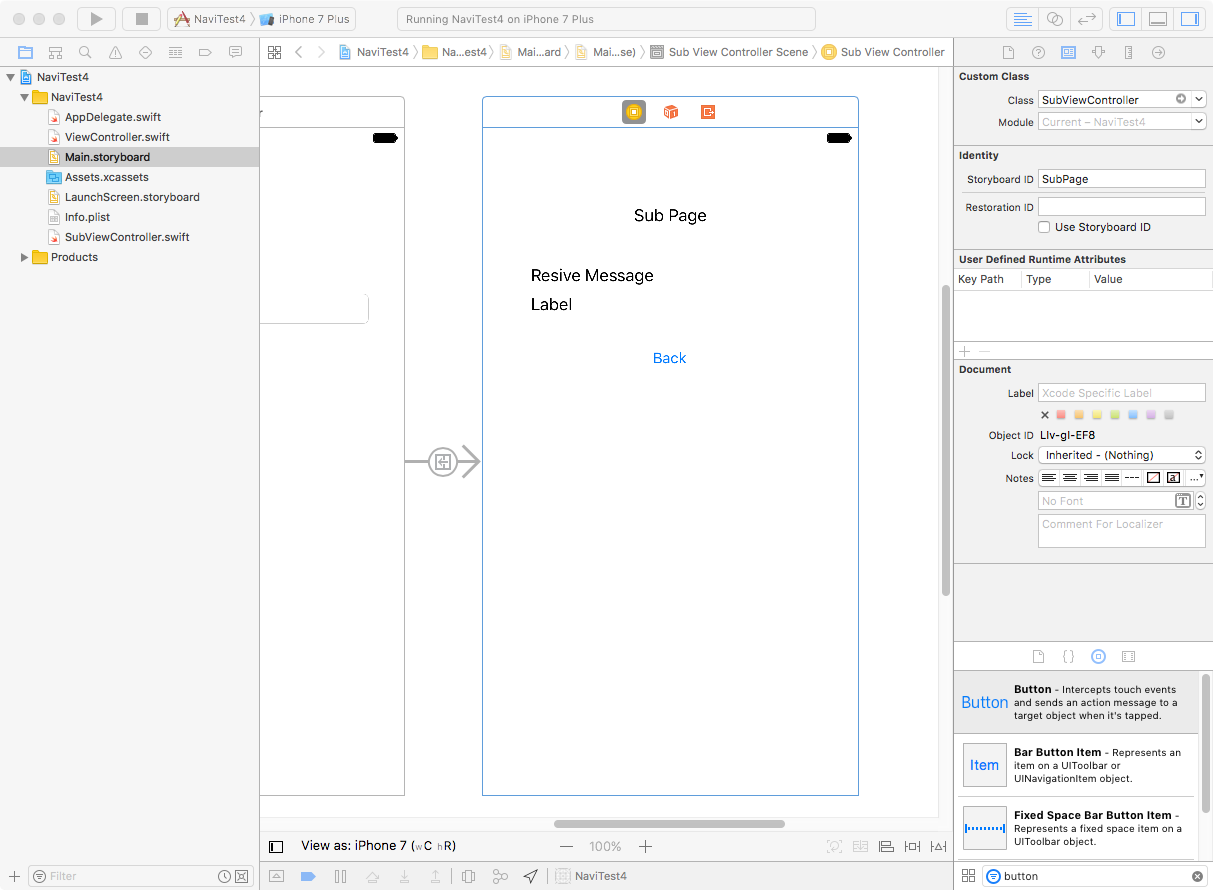
2.4.화면전환

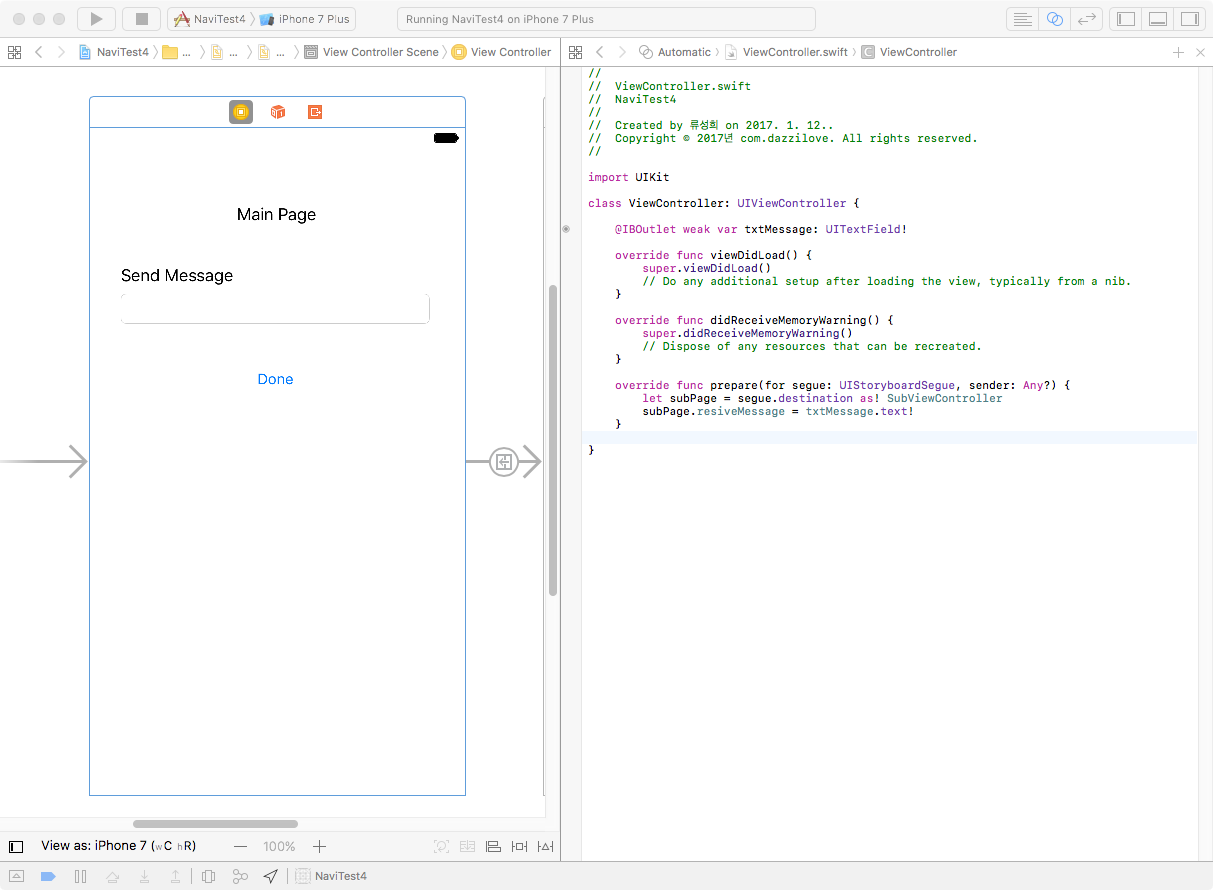
* 세그웨이(Segueway)를 이용한 화면 전환
  + 스토리 보드에서 작업
* 소스코드를 이용한 화면 전환
  + let nextPage = self.storyboard?.instantiateViewController(withIdentifier: "SecondPage")
  + self.present(nextPage!, animated: true, completion: nil)
* 참고 URL = <http://blog.lattecom.xyz/2016/06/06/swift-ios-scene-transition/>

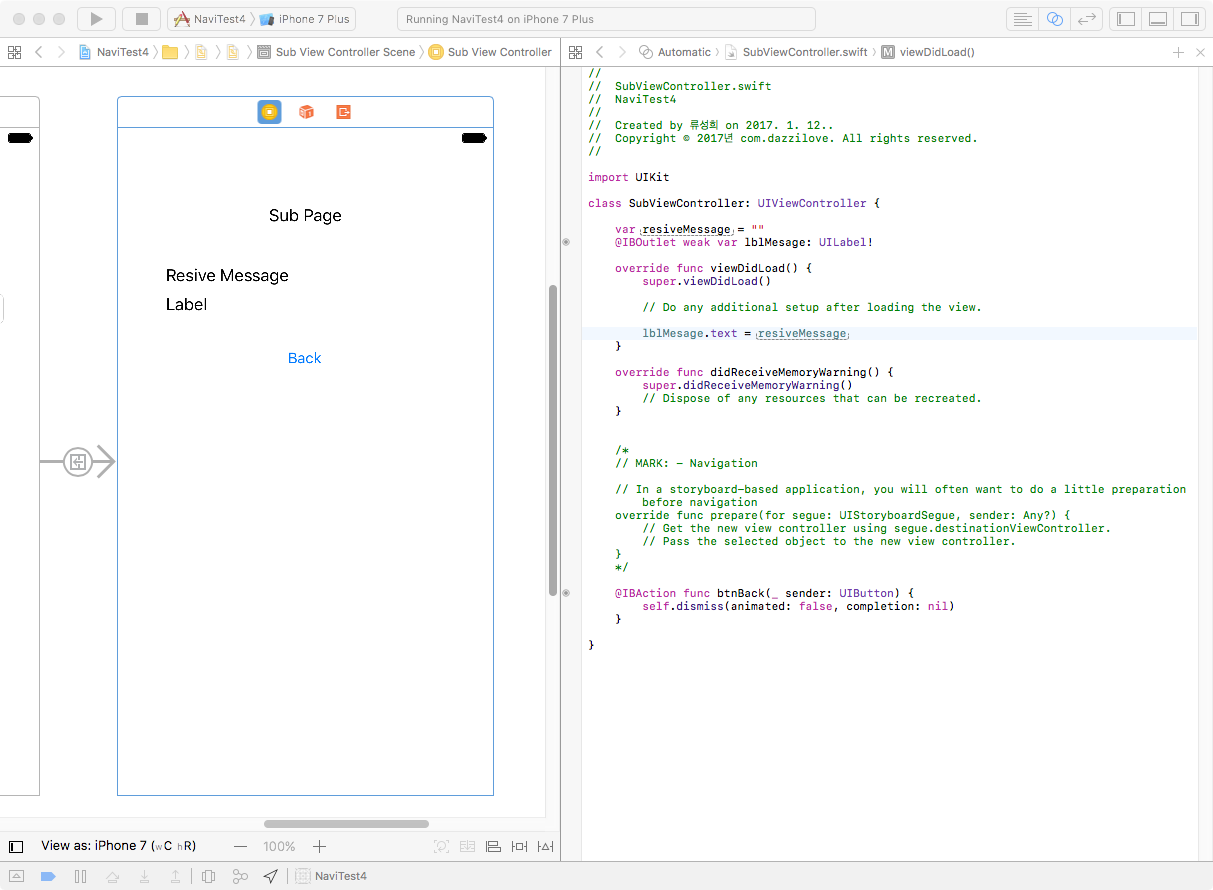
화면 전환용 객체 세그웨이(Segueway) 사용하기

* prepare(for segue: UIStoryboardSegue, sender: Any?)
  + 전환 되는 시점의 뷰 컨트롤러 클래스에 재정의(override)하여 세그웨이가 실행되기 전 특정 코드를 실행할 수 있다.



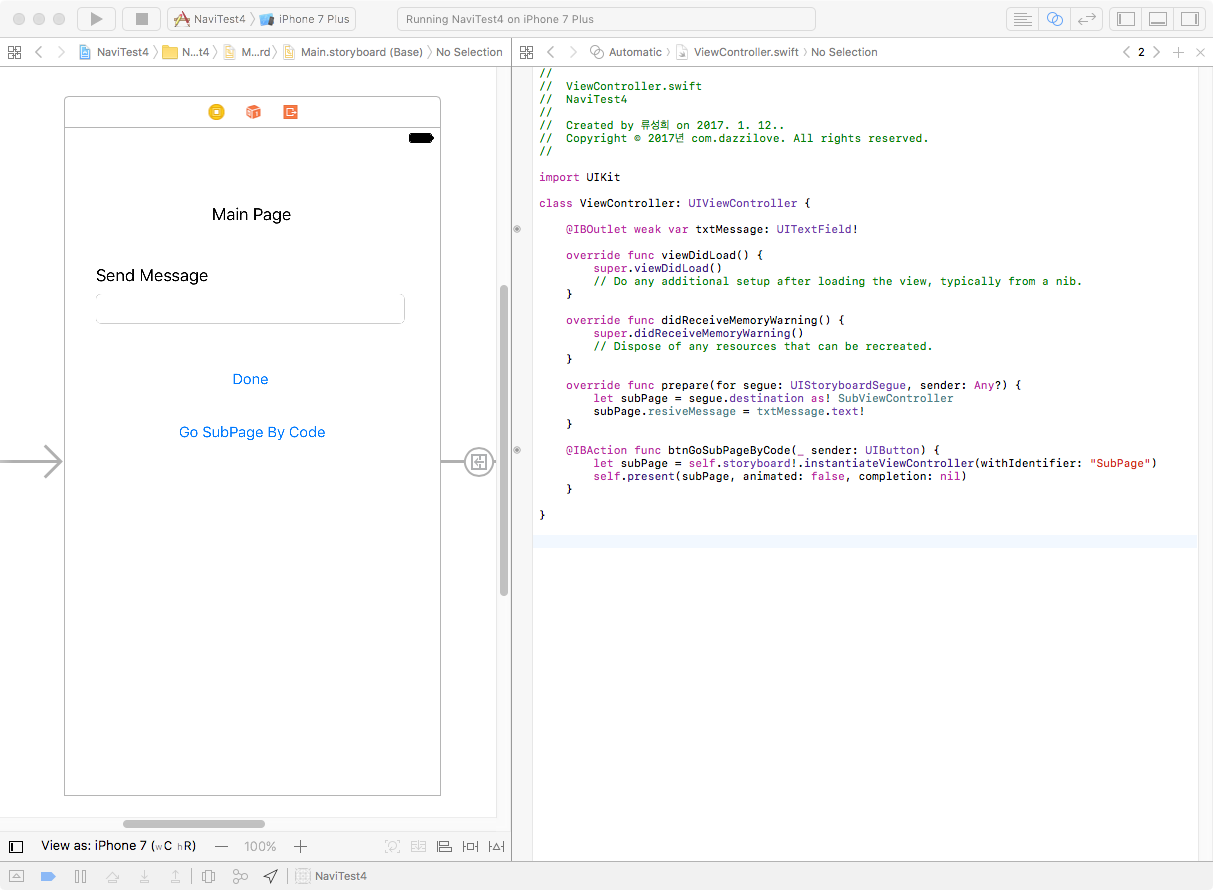




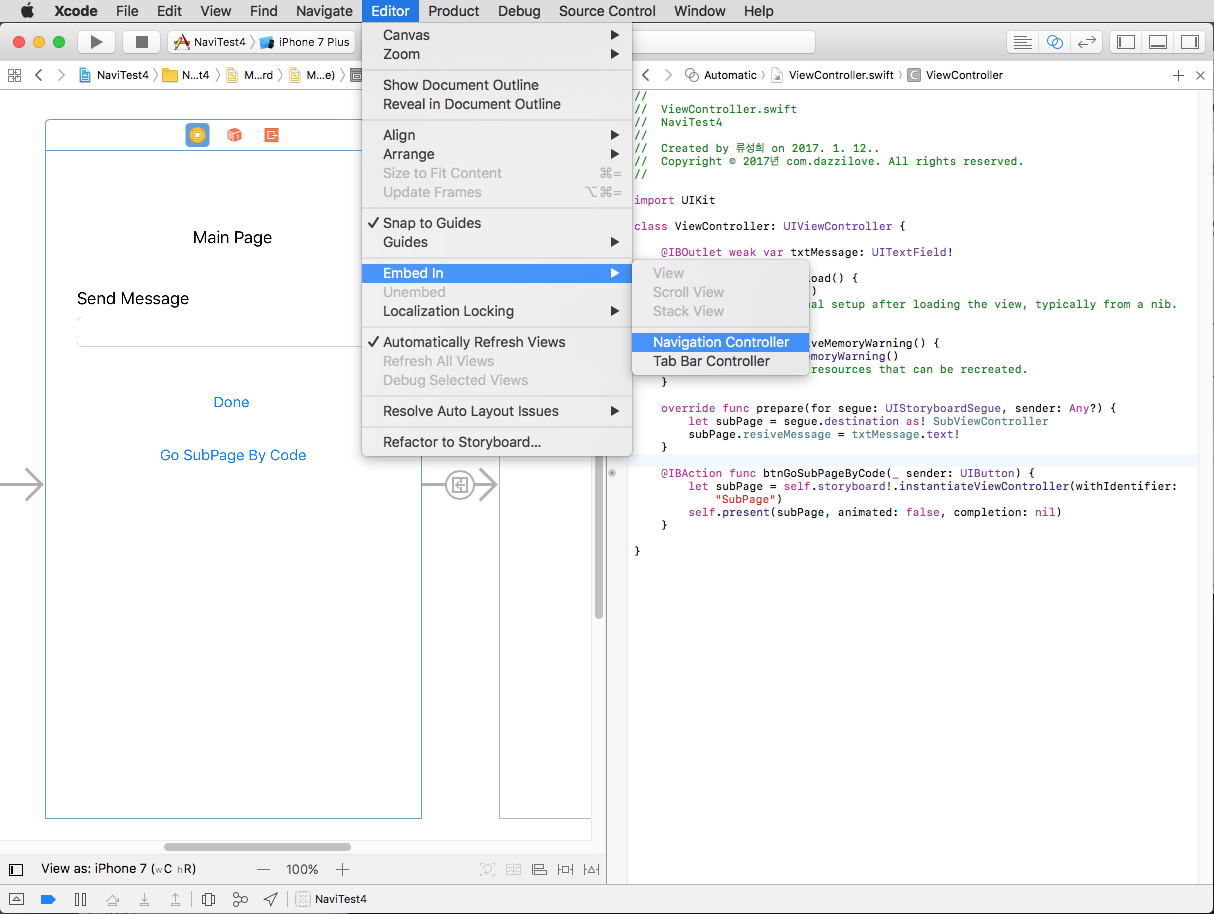


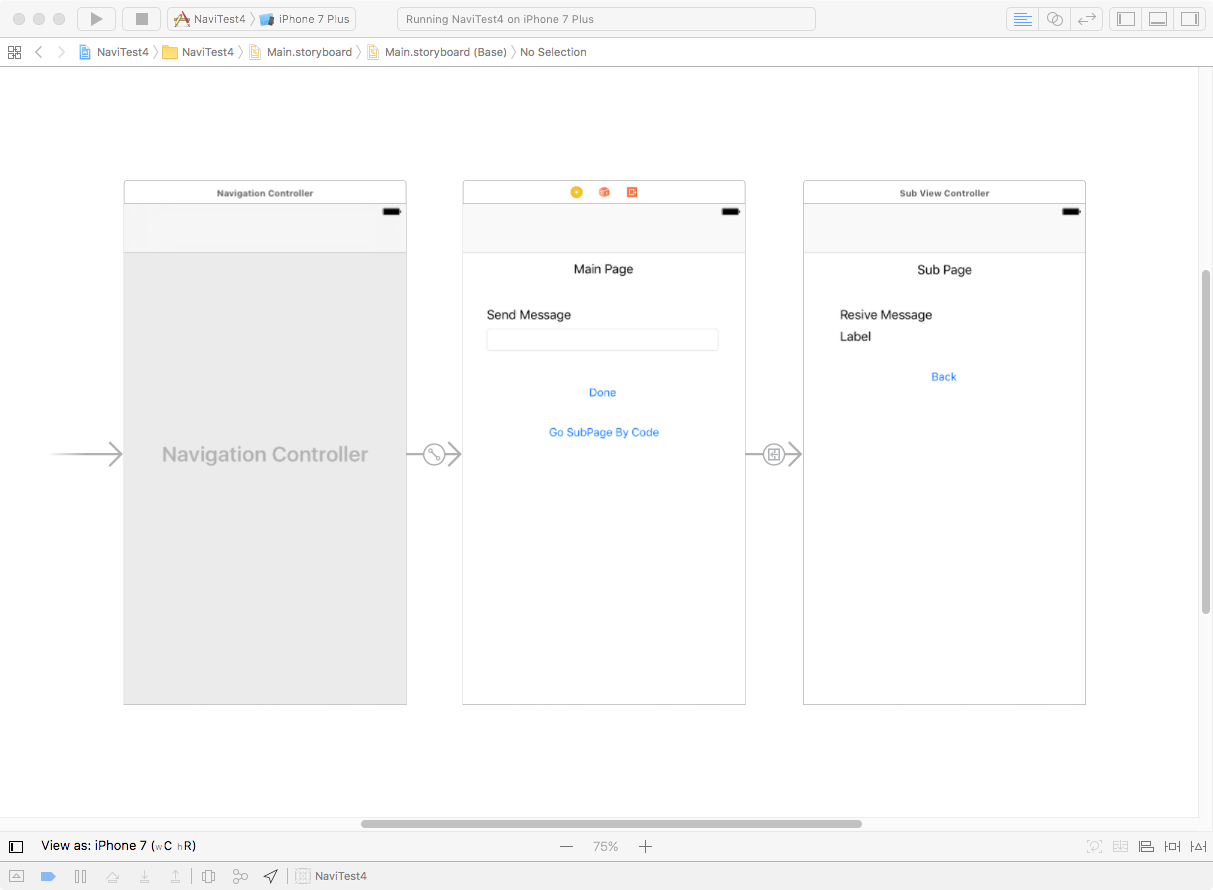
뷰컨트롤러에서 다른 뷰컨트롤러 호출하기

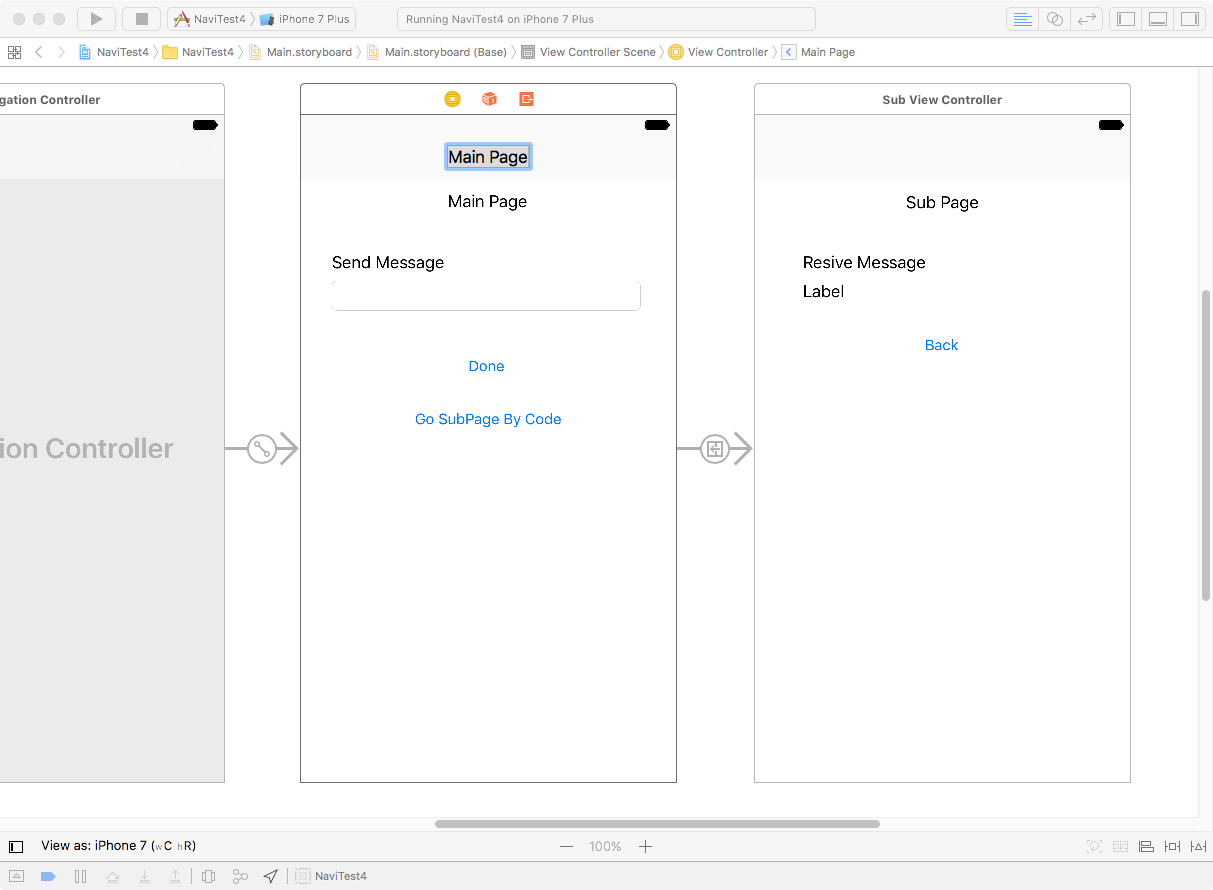
* present(\_ viewControllerToPresent: UIViewController, animated flag: Bool, completion: (() -> Swift.Void)? = nil)
* dismiss(animated flag: Bool, completion: (() -> Swift.Void)? = nil)

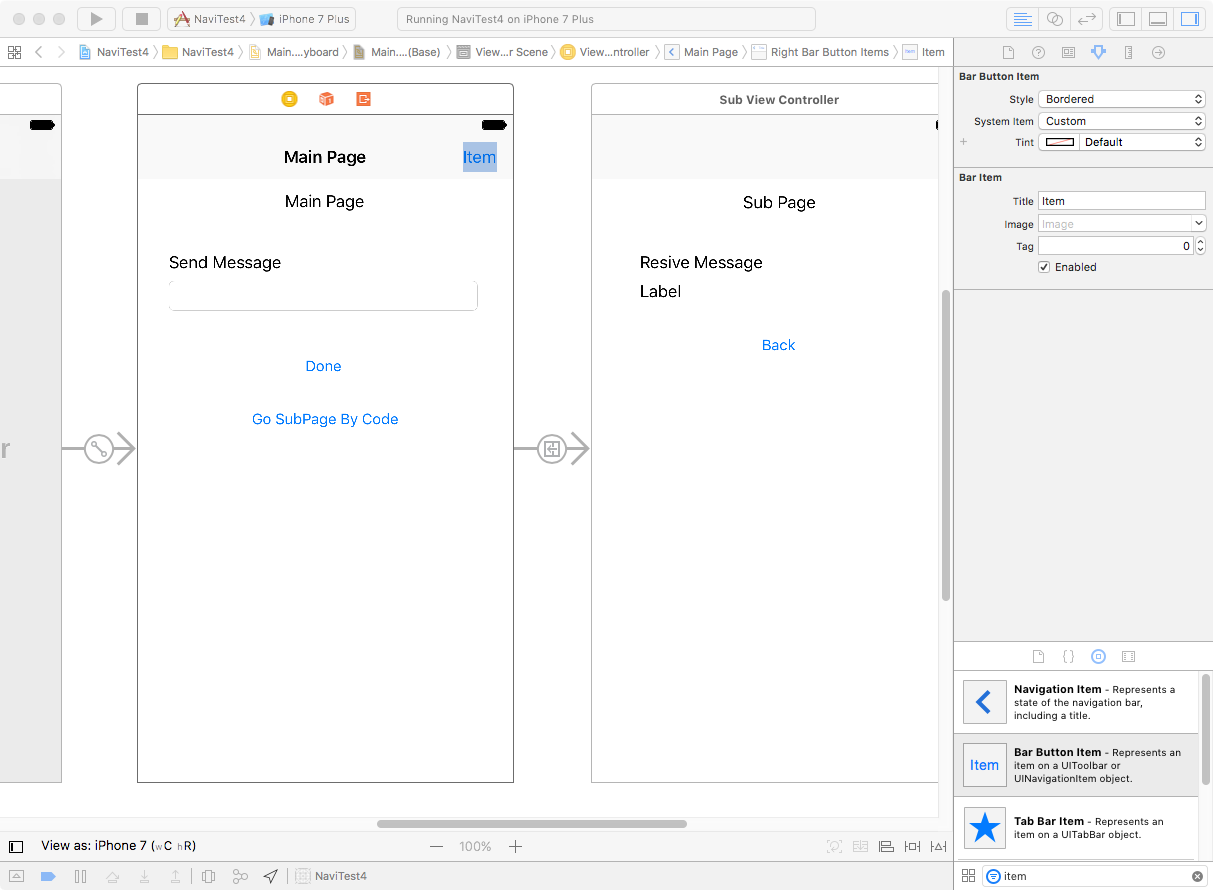


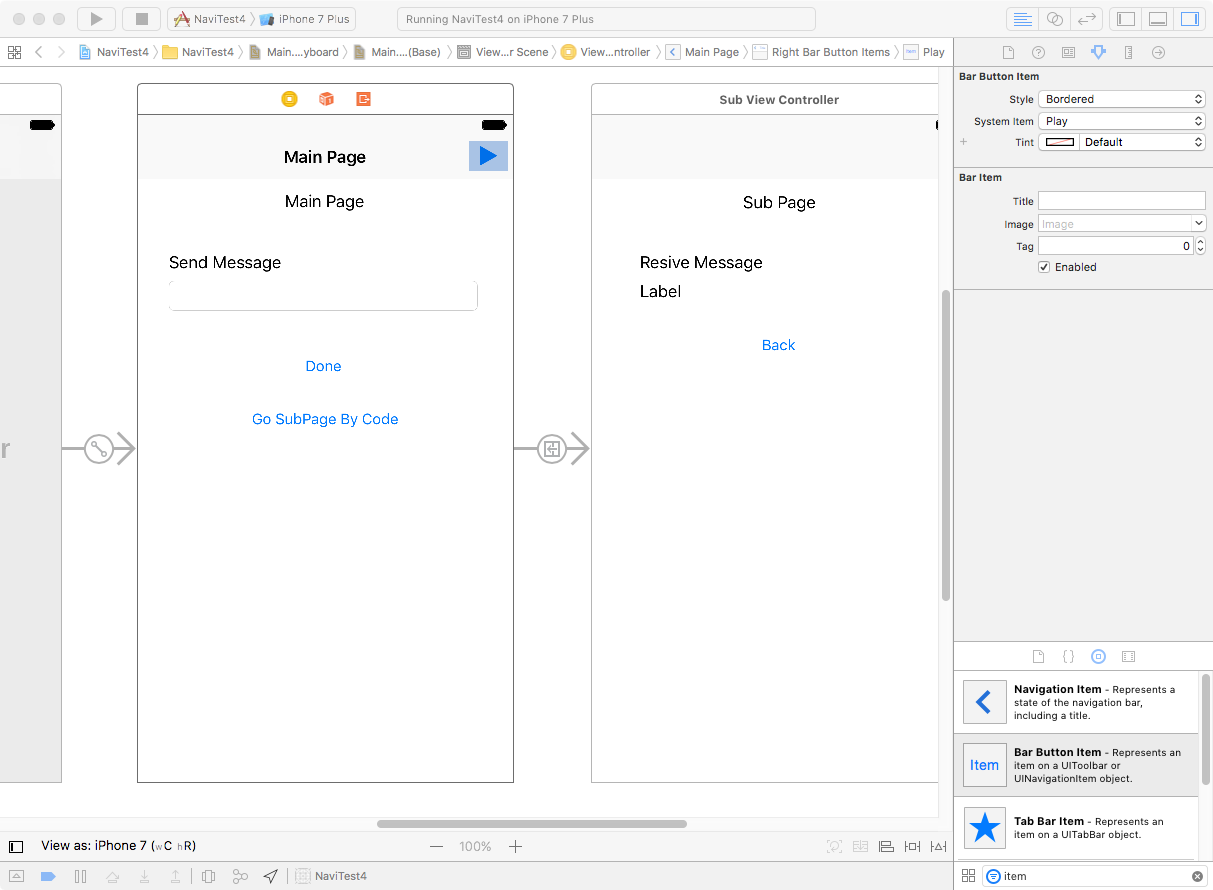
네비게이션 컨트롤러를 이용한 화면 전환

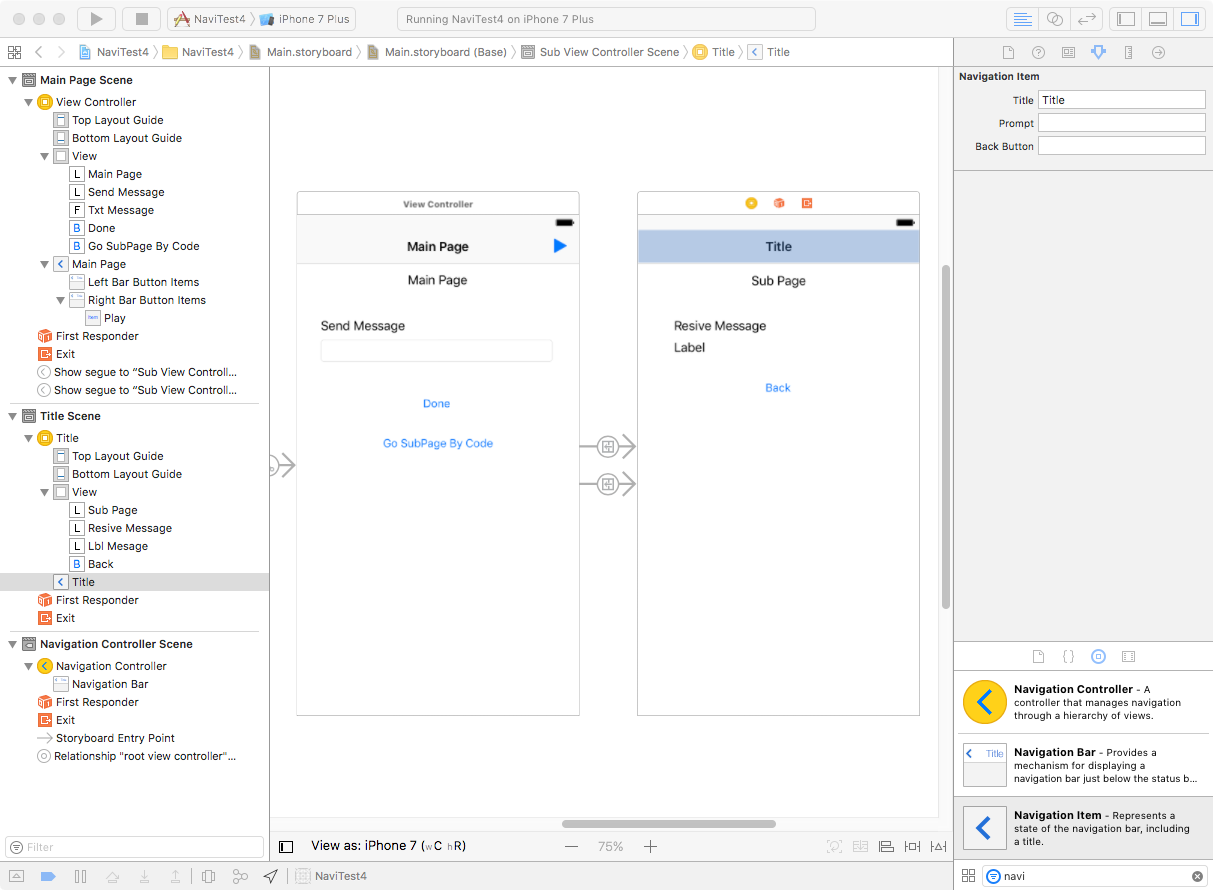
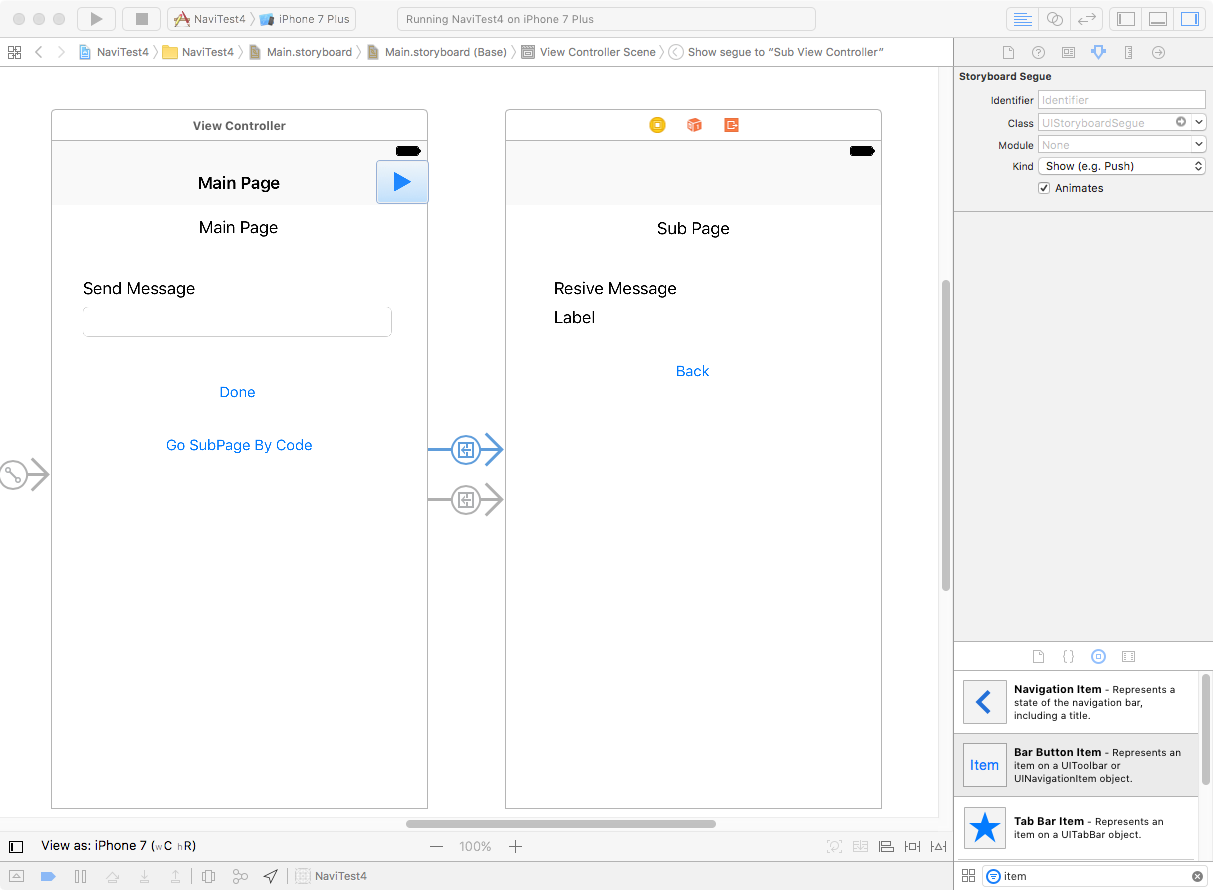


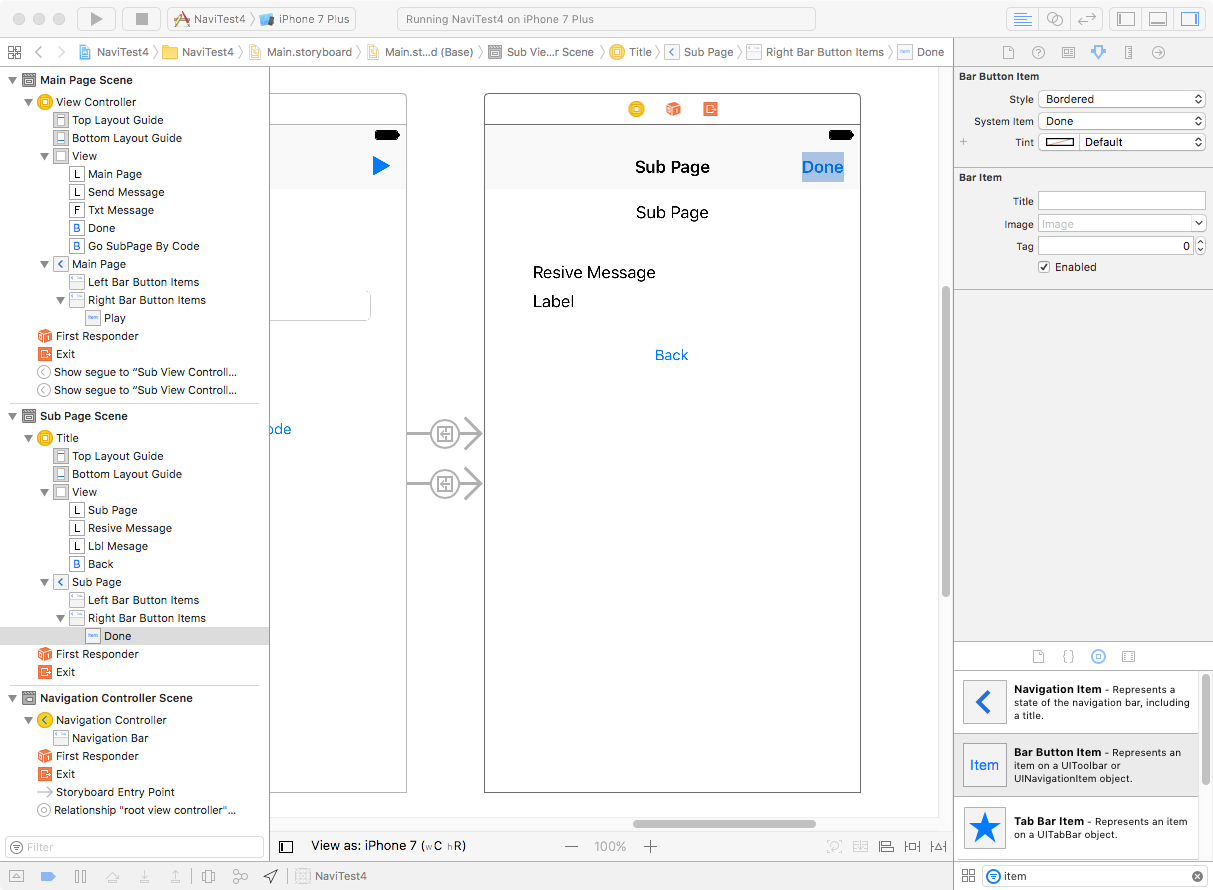


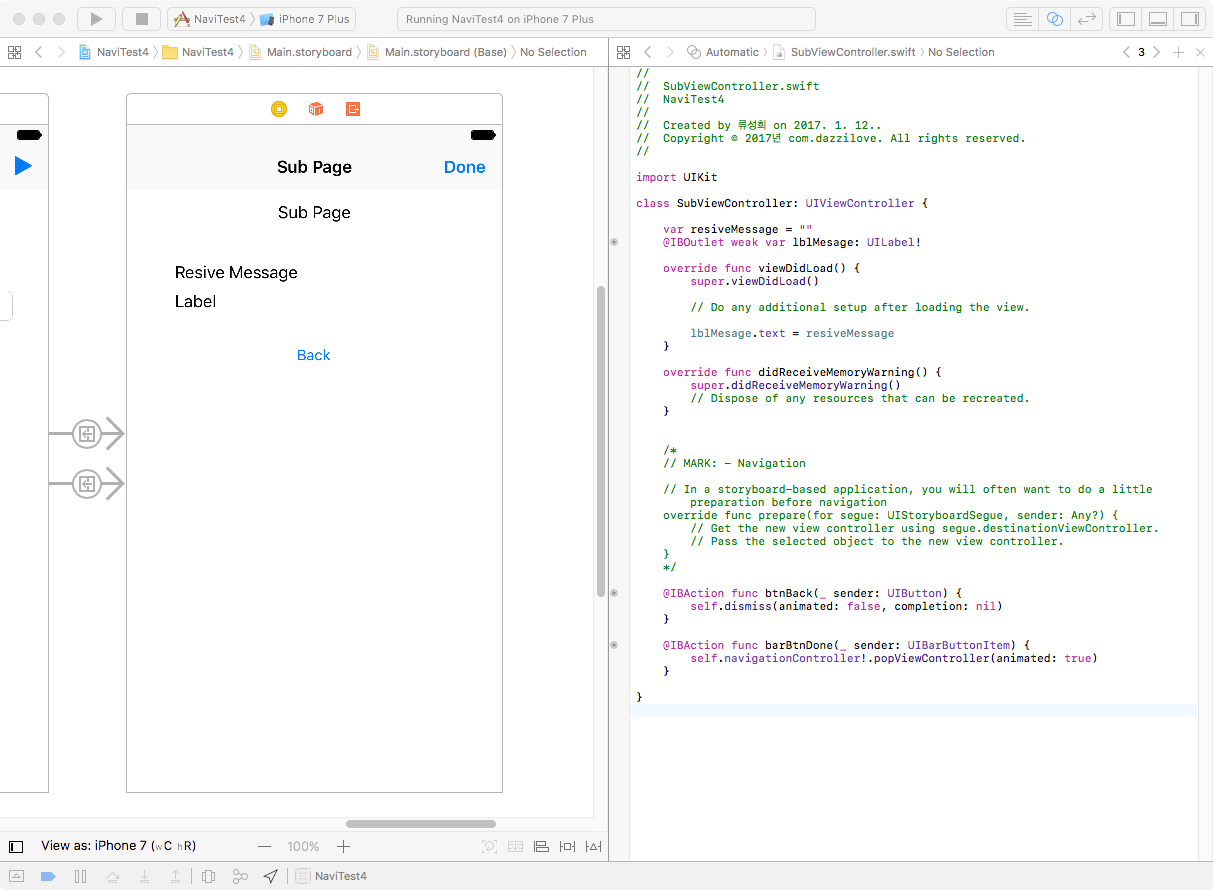


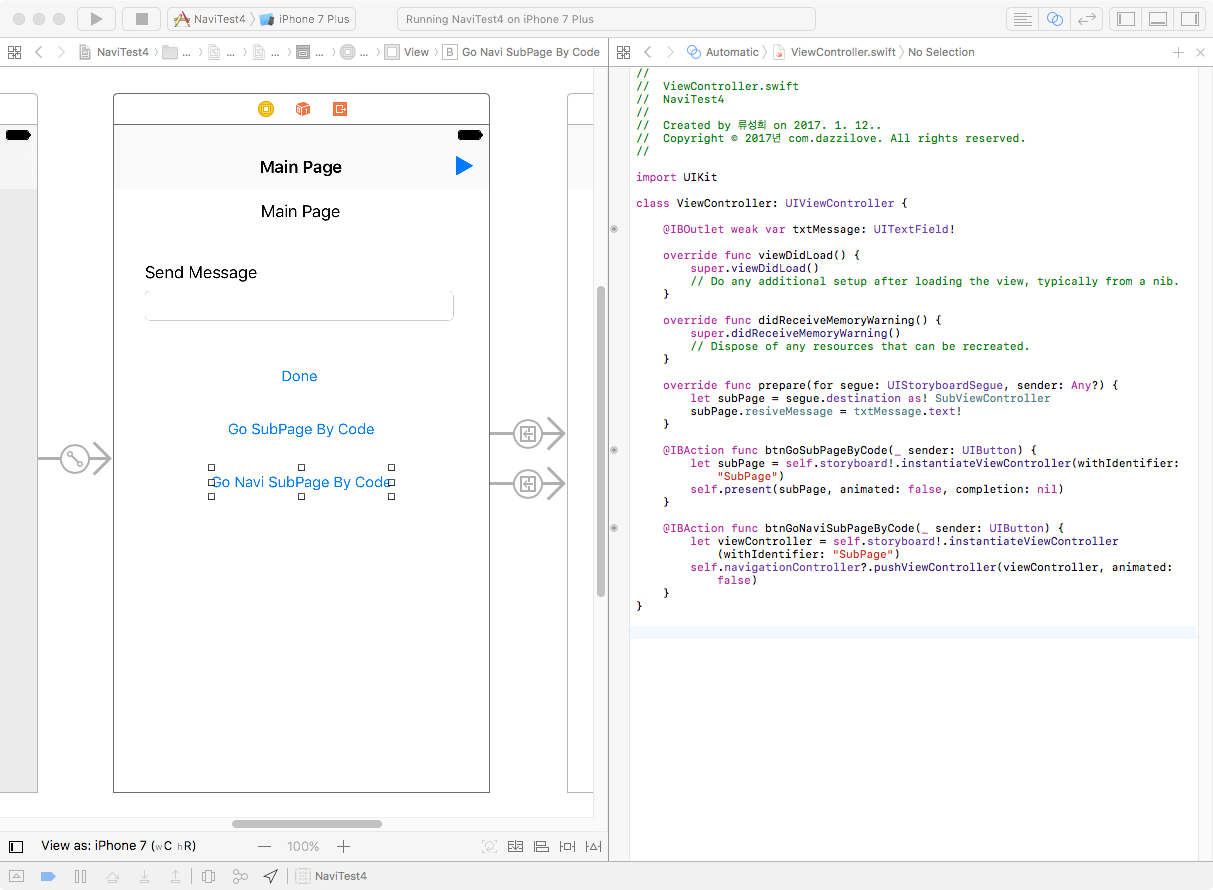




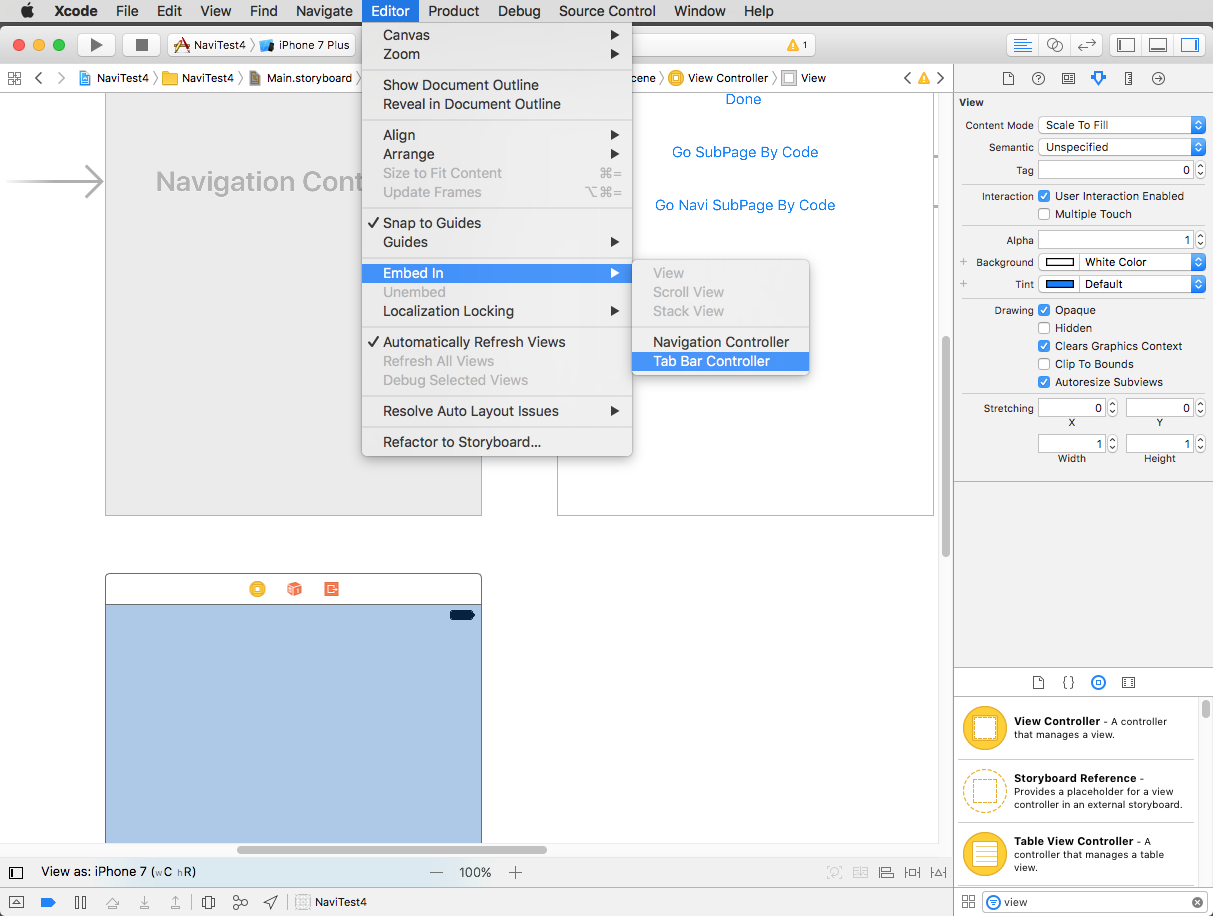


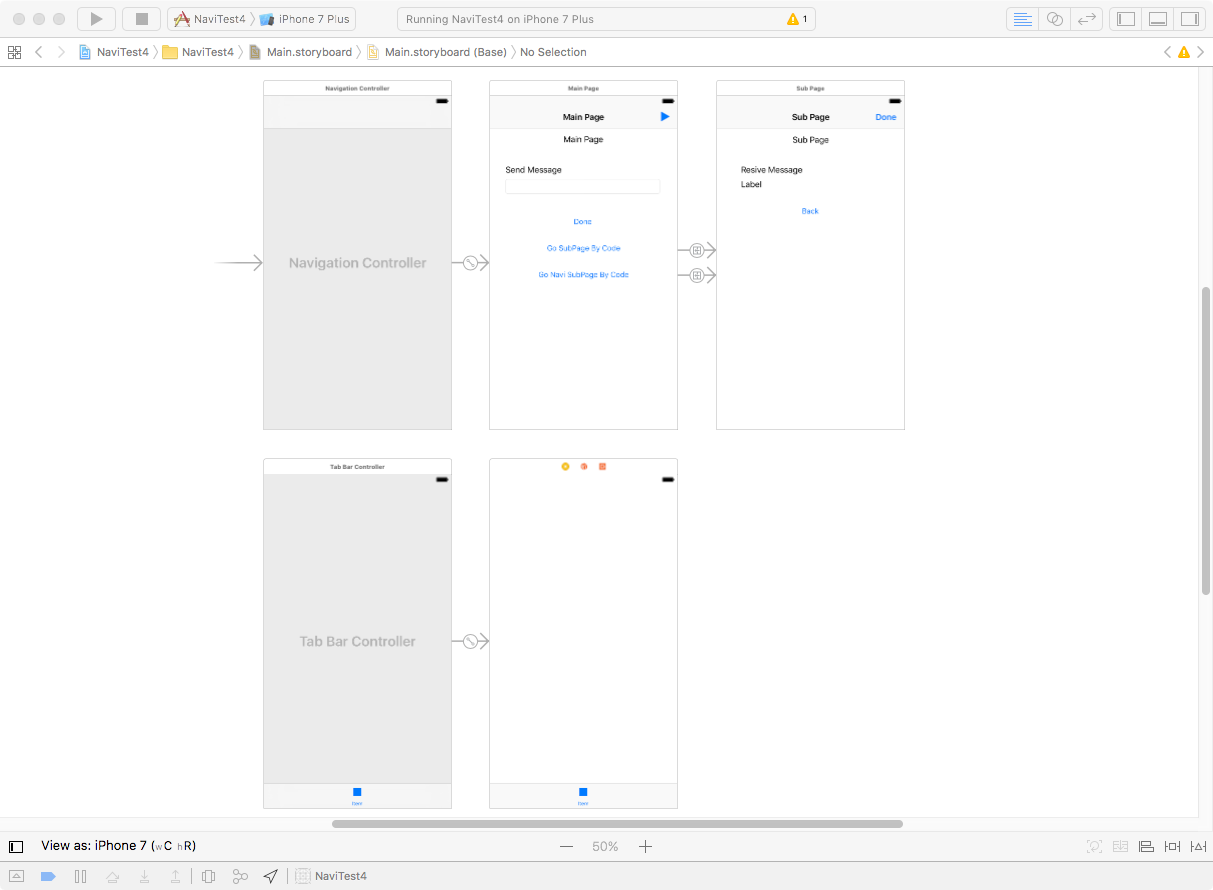


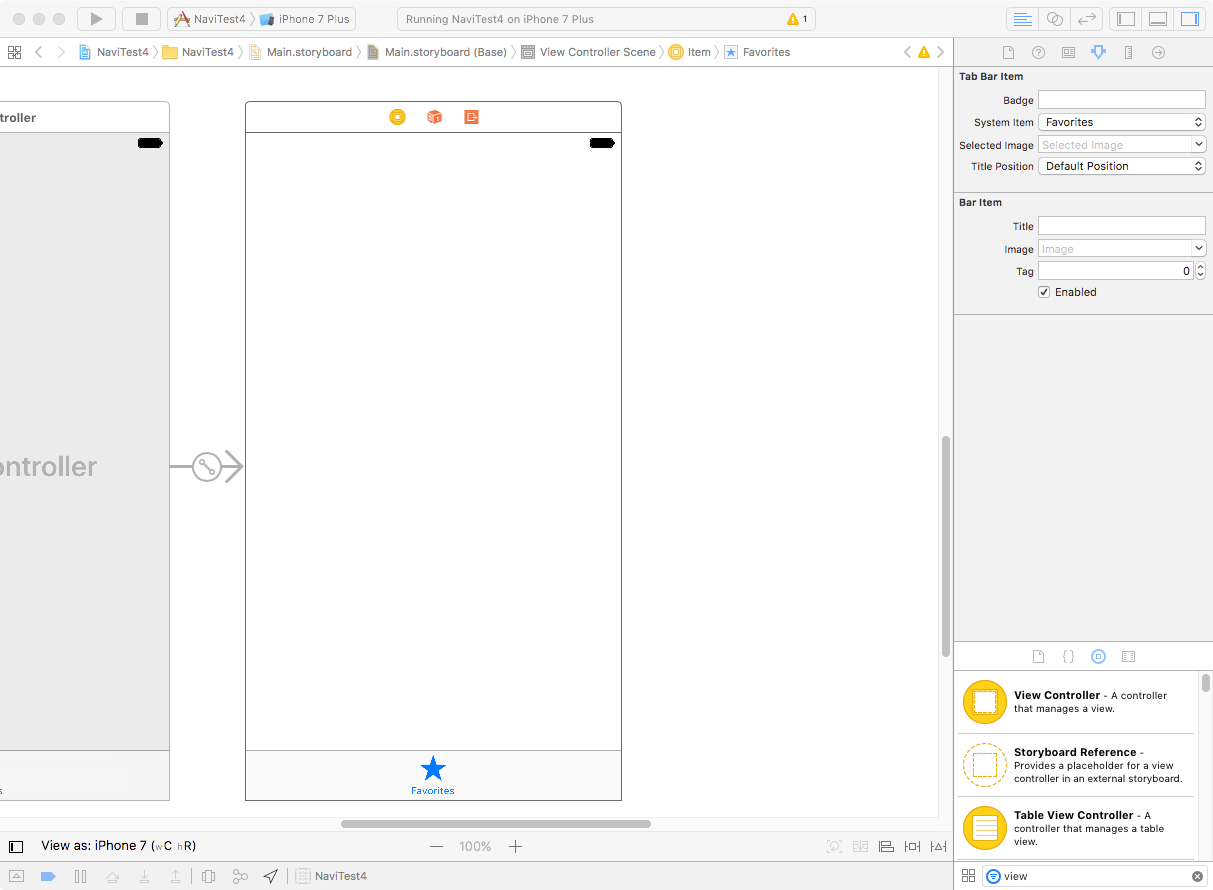


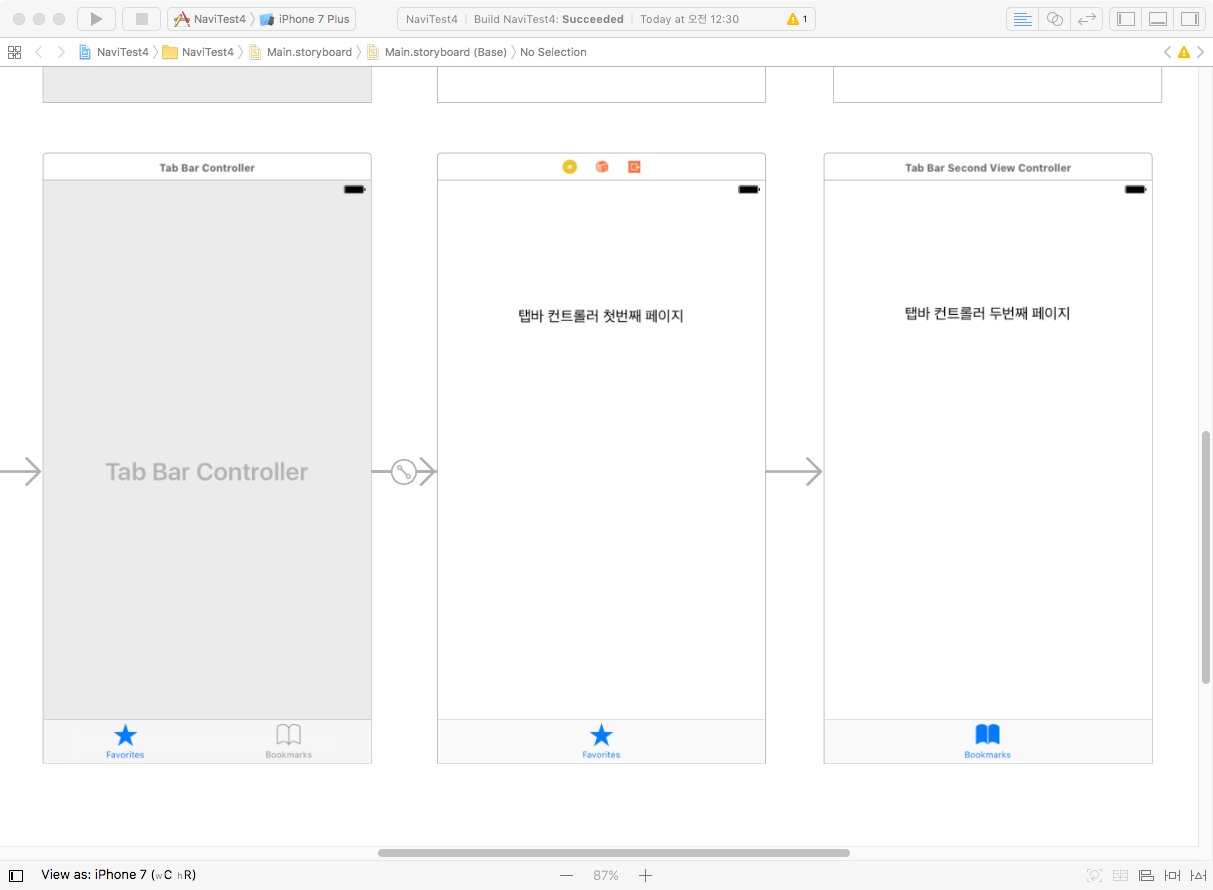


탭바 컨트롤러를 이용한 화면 이동





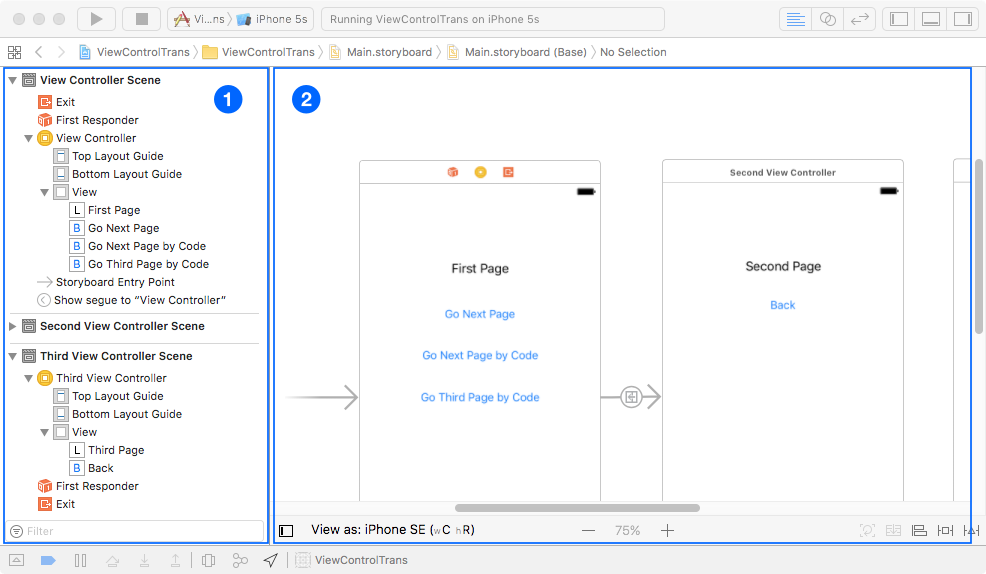




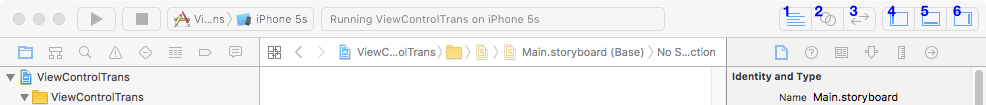
2.1.스토리보드 화면 구성



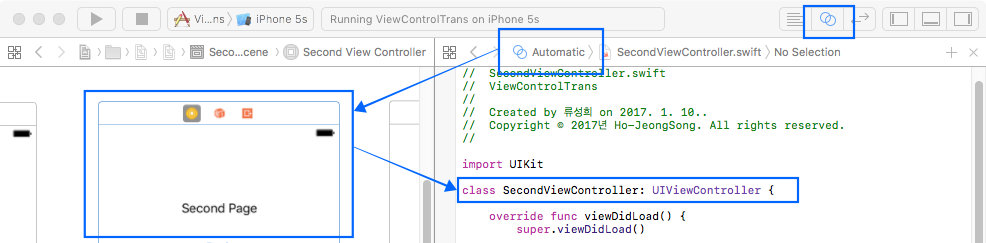
1. 네비게이터 영역
   * Project Navigator, Symbol Navigator, Find Navigator, Issue Navigator, Test Navigator, Debug Navigator, Breakpoint Navigator, Report Navigator
2. 편집기 영역
   * 소스 파일을 열어 편집하거나, 스토리보드를 편집할 수 있는 영역
3. 인스펙터 영역
   * 스토리보드를 편집 할 때 버튼, 컨트롤러 등 객체의 속성을 편집하는 영역
4. 라이브러리 영역
   * 스토리보드에서 사용할 수 있는 버튼, 컨트롤러 등의 모든 객체를 볼 수 있고 사용할 수 있는 영역
5. 디버그 영역
   * 디버그를 진행할 때 변수의 값을 확인하거나 테스트할 목적으로 사용한 입출력 내용이 출력되는 영역
   * 왼쪽 = 변수 영역, 오른쪽 = 콘솔 영역



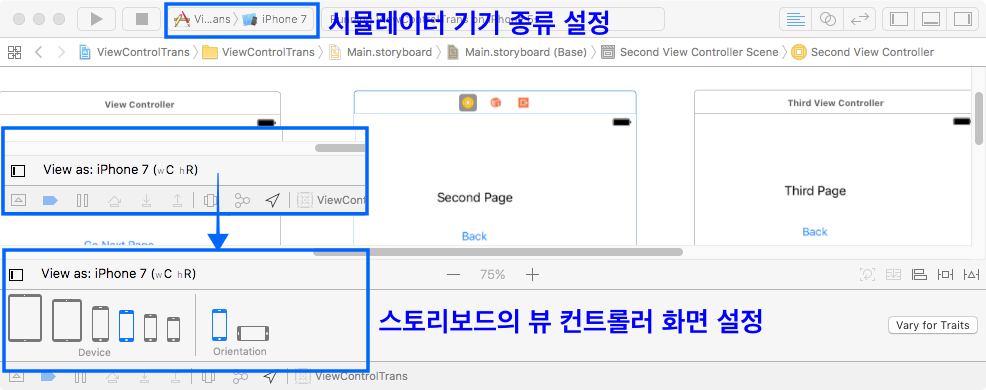
1. 도큐먼트 아웃라인 영역
   * 스토리보드에 존재하는 객체들을 도큐먼트 트리로 노출해준다.
   * 스토리보드에서 객체 선택이 어려운 경우 도큐먼트 트리에서 해당 객체를 선택하면 된다.
   * 스토리보드 왼쪽 하단 아이콘을 이용하여 '숨기기/보이기'가 가능하다.
2. 스토리보드
   * 화면을 구성하는 객체를 배치하는 영역이다.
   * 더블클릭 시 확대/축소가 가능하다.



1. 기본 편집기(Standard editor) 버튼
   * 편집기 영역에 기본 편집기를 연다
2. 보조 편집기(Assistant editor) 버튼
   * 편집기 영역에 보조 편집기를 연다.
3. 버전 편집기(Version editor) 버튼
   * 버전 편집기를 연다. Source Control 기능을 이용할 경우 이전 버전과 현재 버전의 차이를 비교할 수 있다.
4. 내비게이터 영역 감춤/보임(Hide or show the Navigator) 버튼
   * 내비게이터 영역을 감추거나 보여 준다.
5. 디버그 영역 감춤/보임(Hide or show the Debug area) 버튼
   * 디버그 영역을 감추거나 보여 준다.
6. 유틸리티 영역 감춤/보임(Hide or show the Utilities) 버튼
   * 인스펙터 영역과 라이브러리 영역을 합쳐 유틸리티 영역이라 한다.
   * 유틸리티 영역을 감추거나 보여 준다.



* 보조 편집기가 "Automatic" 으로 설정되어 있을 경우 (보통 Automatic이 기본이다)  
  스토리보드에서 선택한 View Controller에 연결된 소스파일이 자동으로 노출 된다.

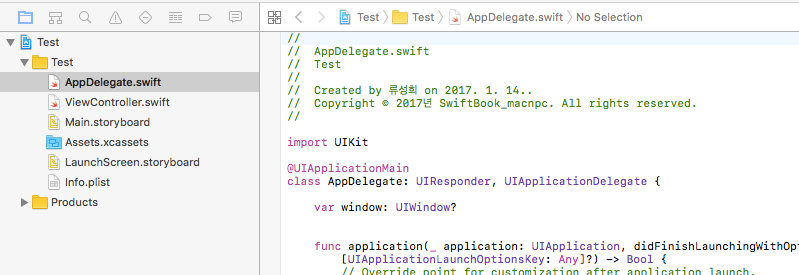


* 스토리보드의 뷰 컨트롤러와 시뮬레이터 기기의 종류를 설정하여 확인 할 수 있다.

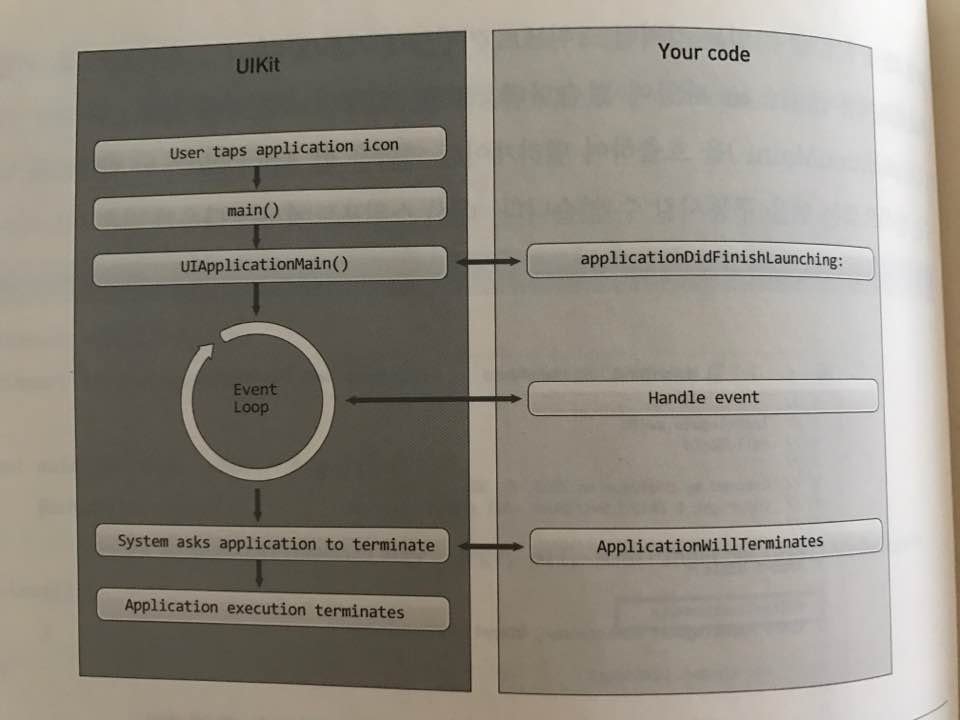
2.2.iOS와 앱 생명주기

앱의 생명주기

* iOS 역시 main() 함수를 호출하여 앱이 실행된다.
* 다만 iOS 앱은 일반 프로그램과는 달리 앤트리 포인트로서의 main() 함수를 직접 작성할 필요가 없다.
* 기본 프로젝트를 생성 할 때 Xcode가 알아서 main() 함수를 생성한다.



* UIApplicationMain() 을 호출하여 델리게이트를 전달하는 방법 대신, 델리게이트 역할을 할 클래스에 @UIApplicationMain 어노테이션을 걸어주는 방식으로 시스템에 델리게이트 클래스 정보를 전달하고, iOS 시스템은 앱을 실행할 때 어노테이션이 표시된 클래스를 찾아 델리게이트로 지정 한다.



import UIKit

@UIApplicationMain

class AppDelegate: UIResponder, UIApplicationDelegate {

var window: UIWindow?

func application(\_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

// Override point for customization after application launch.

return true

}

func applicationWillResignActive(\_ application: UIApplication) {

// Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.

// Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks. Games should use this method to pause the game.

}

func applicationDidEnterBackground(\_ application: UIApplication) {

// Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is terminated later.

// If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.

}

func applicationWillEnterForeground(\_ application: UIApplication) {

// Called as part of the transition from the background to the active state; here you can undo many of the changes made on entering the background.

}

func applicationDidBecomeActive(\_ application: UIApplication) {

// Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.

}

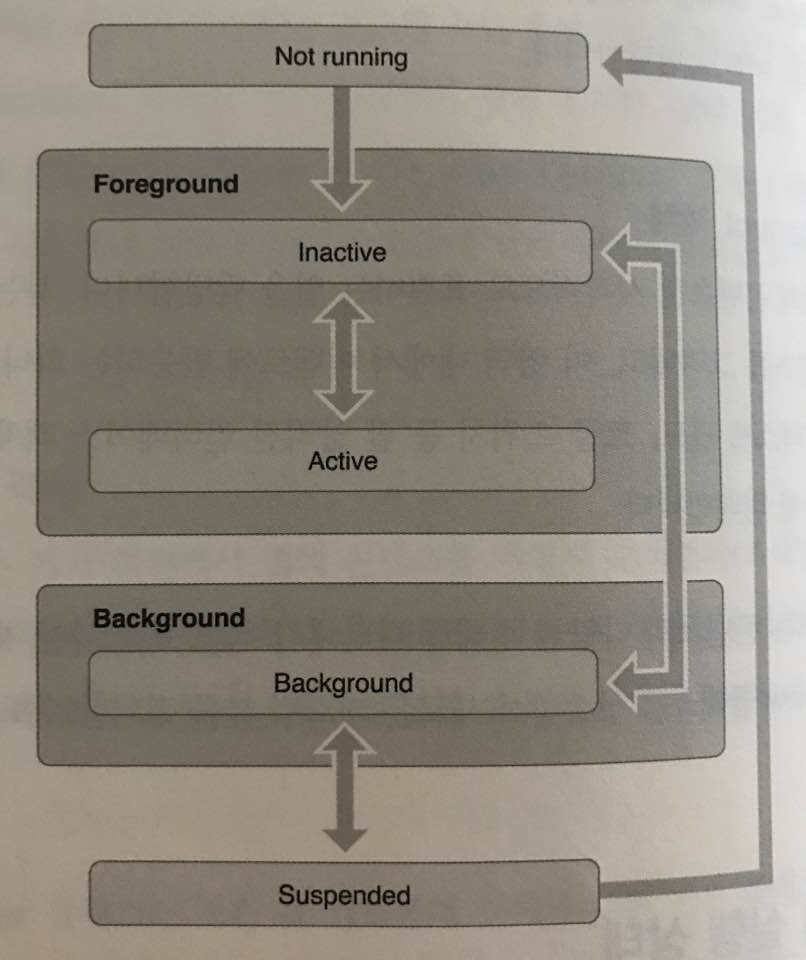
func applicationWillTerminate(\_ application: UIApplication) {

// Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.

}

}

앱의 실행 상태



앱의 상태 변화

* Not Running = 앱이 시작되지 않았거나 실행되었지만 시스템에 의해 종료된 경우
* Inactive = 앱이 전면에서 실행 중이지만, 아무런 이벤트를 받지 않고 있는 상태
* Active = 앱이 전면에서 실행 중이며 이벤트를 받고 있는 상태
* Background = 앱이 백그라운드에 있고, 코드가 실행되고 있는 상태. 대부분의 앱은 일시정지 상태로 가는 도중에 일시적으로 이 상태에 진입하게 됨. 그러나 여분의 실행 시간을 요청하는 앱의 경우 특정 시간 동안 이 상태로 남아있게 되는 경우도 있음
* Suspend = 앱이 백그라운드에 있고, 실행되는 코드가 없는 상태. 이 상태에서는 앱이 메모리에 유지되지만 아무런 실행코드가 없음. 메모리 부족 상황일 때 시스템은 전면에 있는 앱들에 대한 여유 메모리 공간을 확보하기 위해 이 상태에 있는 앱들을 특별한 알림 없이 정리하게 됨

앱의 상태변화에 따라 등록된 앱 델리게이트 객체에는 API 호출이 발생 한다.

* application(\_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool
  + 앱이 처음 실행되는 순간 호출 된다.
* pplicationWillResignActive(\_ application: UIApplication)
  + 앱이 사용자에게 화면으로 표시되기 직전에 호출되어 전체적인 초기화를 수행할 수 있도록 해준다.
* applicationDidEnterBackground(\_ application: UIApplication)
  + 앱이 현재 백그라운드에서 실행 중이고 언제 어느 순간에 일시 중지될 수 있는 상태일 때 호출 된다.
* applicationWillEnterForeground(\_ application: UIApplication)
  + 앱이 백그라운드에서 포그라운드로 옮겨질 때 실행되지만, 아직 활성화 상태는 아닌 경우이다.
* applicationDidBecomeActive(\_ application: UIApplication)
  + 앱이 포그라운드로 올라올 때 실행 된다.
* applicationWillTerminate(\_ application: UIApplication)
  + 앱이 제거될 때 발생 한다.

뷰 컨트롤러의 상태 변화

뷰 컨트롤러의 상태

* Appearing = 뷰 컨트롤러가 스크린에 표시되기 위해 처리되고 있는 상태
* Appeared = 뷰 컨트롤러가 스크린에 완전히 표시된 상태
* Disappearing = 뷰 컨틀롤러가 스크린에서 사라지고 있는 상태
* Disappeared = 뷰 컨트롤러가 스크린에서 완전히 사라진 상태

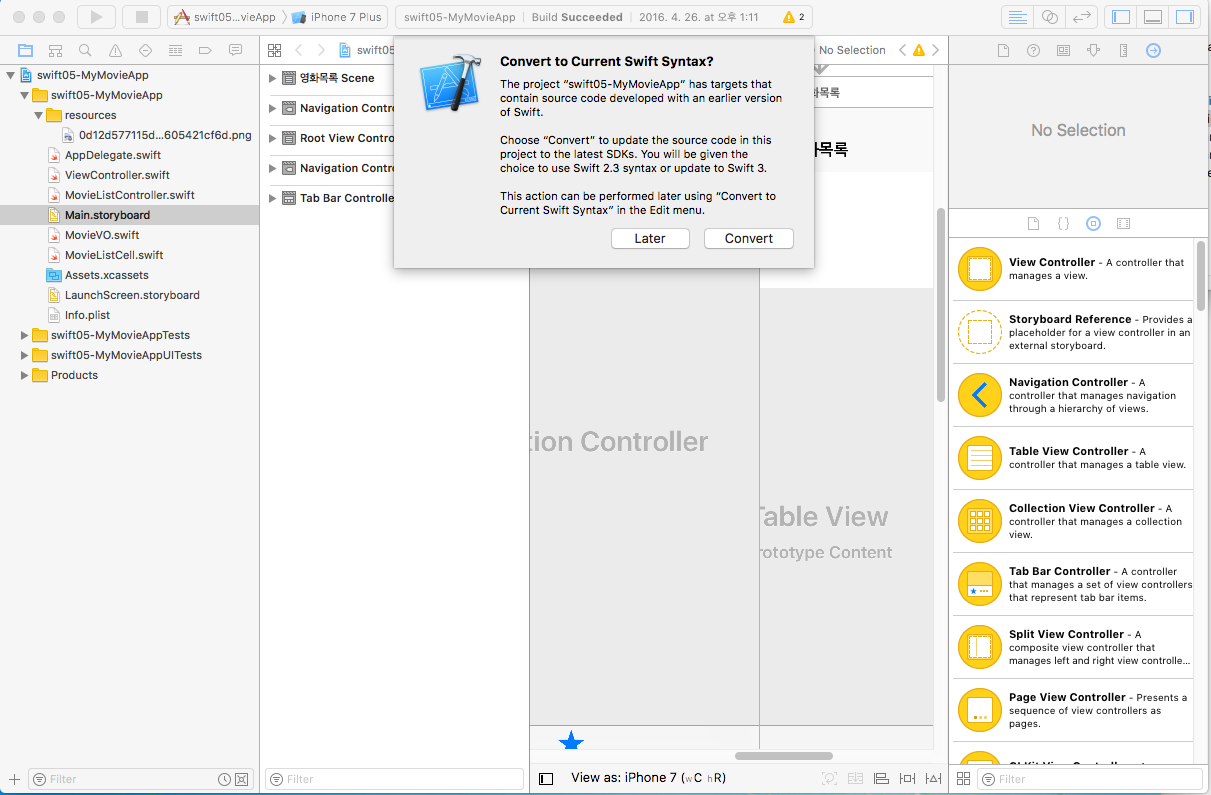
뷰 컨트롤러 상태 변화 중 발생하는 이벤트

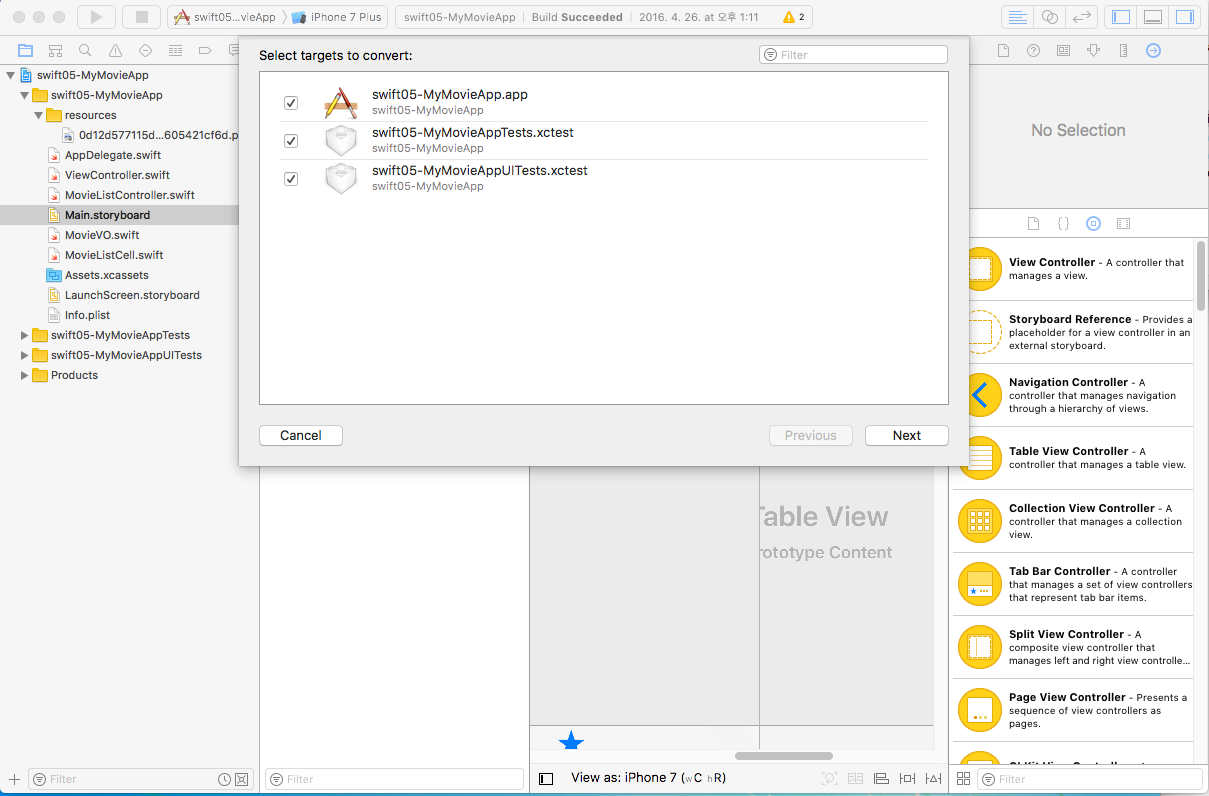
* viewWillAppear(\_ animated: Bool)
  + 화면이 완전히 가려진 상태에서 표시되기 시작하는 상태(=Appearing)로 변화가 일어나는 동안 뷰 컨트롤러는 앱 시스템에 의해 viewWillAppear 메서드가 호출 된다.
  + 다른 화면으로 이동했다가 다시 본래 화면이 표시될 때 데이터를 갱신하여 보여주고 싶다면 이 메서드를 재정의하여 원하는 코드를 작성하면 된다.
* viewDidAppear(\_ animated: Bool)
  + 화면이 표시 처리되는 단계를 넘어서 화면이 완전히 표시되고 나면 viewDidAppear 메서드가 호출 된다.
* viewWillDisappear(\_ animated: Bool)
  + viewDidAppear 메서드가 호출 된 상태에서 다른 액션이 일어나 화면의 전환이 이루어지고나 홈 버튼을 눌러 백그라운드로 내려가는 등 스크린에서 화면이 제거되는 상태가 발생하면 viewWillDisapper 메서드가 호출 된다.
* viewDidDisappear(\_ animated: Bool)
  + viewWillDisappear 메소드가 호출되고, 상태변화가 완료되면 viewDidDisappear 메서드가 호출 된다.

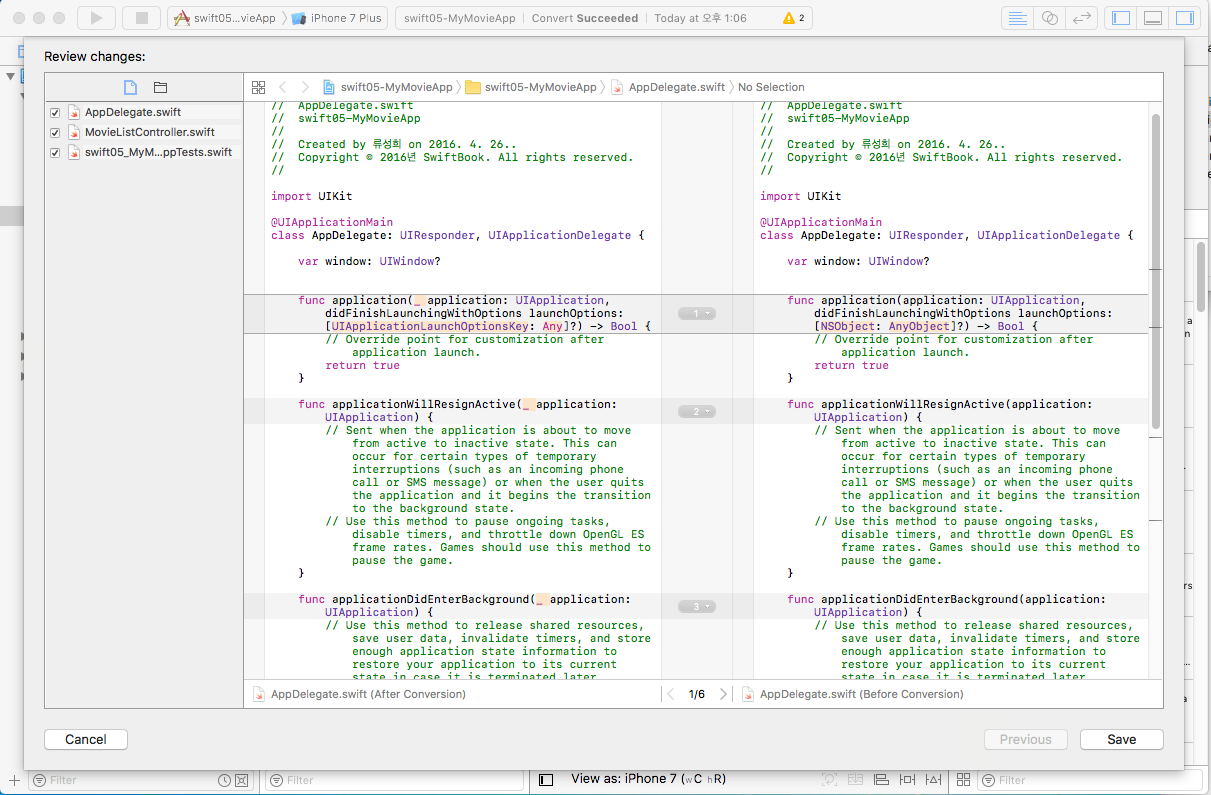
Launch Screen

* 기본 런치 스크린 코드 시연
* 커스텀 런치 스크린 코드 시연

2. Swift2에서 Swift3로 자동 컨버팅







| **작성자** | **dazzilove** |
| --- | --- |
| 작성일 | 2017.01.09 04:09:00 |
| 수정자 | **dazzilove** |
| 수정일 | 2017.01.09 04:09:00 |
| 조회수 | 12 |
| 추천수 | 0 |
| 글상태 | 기본 |

첨부파일

[swift\_ver2\_1.png](http://libqa.com/download?path=/resource/temp/9/201701098733969212648137.png)[swift\_ver2\_2.png](http://libqa.com/download?path=/resource/temp/9/201701098733974683982763.png)[swift\_ver2\_3.png](http://libqa.com/download?path=/resource/temp/9/201701098733977599682069.png)