

---

# 계산기 만들기

---

# 기초 중의 기초

---

- 객체 지향형 프로그래밍이란?

# 객체지향형 프로그래밍 방법론

---

객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다. 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다.

# 집을 만들어 봅시다.

---

- 만약 컴퓨터로 집을 만든다고 한다면 어떻게 해야할까요?

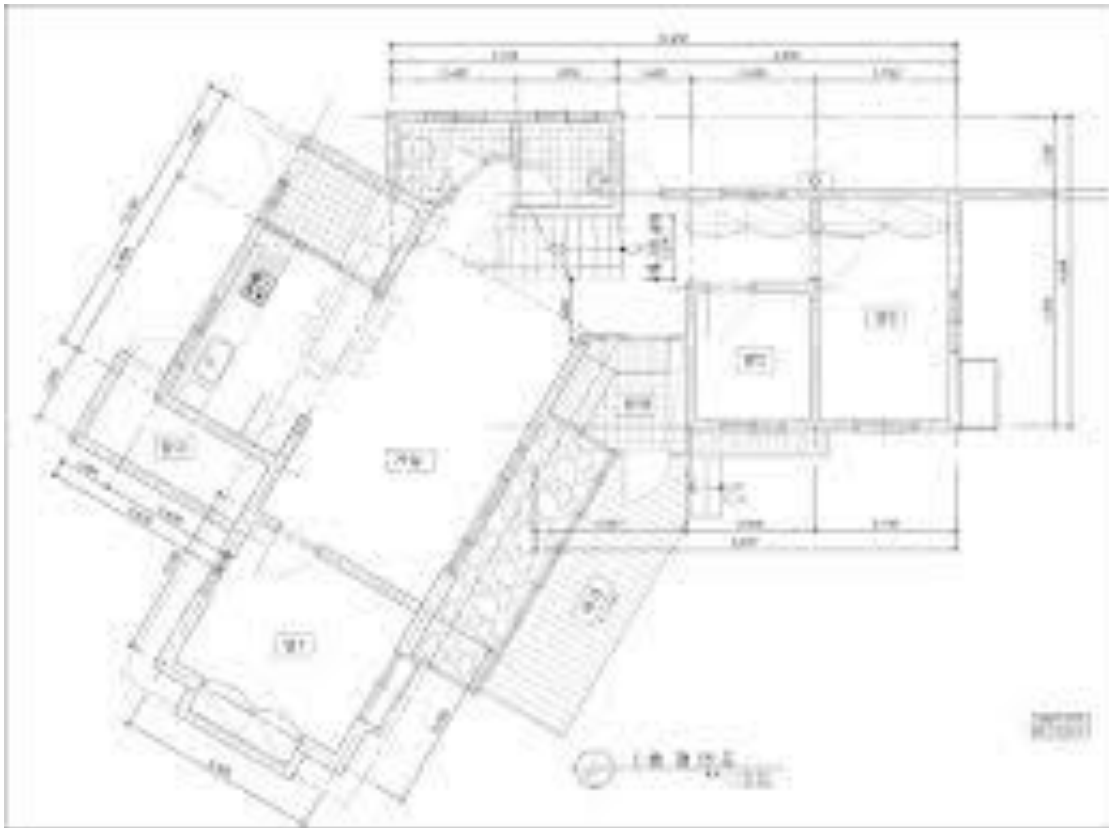
# 기본 구성 요소

---

- **클래스(Class)** - 같은 종류(또는 문제 해결을 위한)의 집단에 속하는 속성(attribute)과 행위(behavior)를 정의한 것으로 객체지향 프로그램의 기본적인 사용자 정의 데이터형(user define data type)이라고 할 수 있다. 클래스는 프로그래머가 아니지만 해결해야 할 문제가 속하는 영역에 종사하는 사람이라면 사용할 수 있고, 다른 클래스 또는 외부 요소와 독립적으로 디자인하여야 한다.
- **객체(Object)** - 클래스의 인스턴스(실제로 메모리상에 할당된 것)이다. 객체는 자신 고유의 속성(attribute)을 가지며 클래스에서 정의한 행위(behavior)를 수행할 수 있다. 객체의 행위는 클래스에 정의된 행위에 대한 정의를 공유함으로써 메모리를 경제적으로 사용한다.
- **메서드(Method), 메시지(Message)** - 클래스로부터 생성된 객체를 사용하는 방법으로서 객체에 명령을 내리는 메시지라 할 수 있다. 메서드는 한 객체의 서브루틴(subroutine) 형태로 객체의 속성을 조작하는 데 사용된다. 또 객체 간의 통신은 메시지를 통해 이루어진다.

# 클래스 vs 객체

---



# 클래스 vs 객체

---



# 클래스 vs 객체

---

- 여러분은 어떤 차이를 느끼시나요?





**Inner Peace**

# 객체지향형 프로그래밍 방법론

---

객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다. 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다.

# 객체지향형 프로그래밍 특징

---

- 추상화
- 캡슐화
- 은닉화
- 상속성
- 다형성

# Swift Class Architecture

```
class ClassName : superClass
{
    var vName1 = "1"
    var vName2 = 4

    func fName1() - > Any
    {

    }

    func fName2(_ ani:Bool)
    {

    }
}
```

<CalssName.swift>

# 변수 & 함수

---

- 변수 : 프로그램에서 데이터의 저장공간을 담당
- 함수 : 프로그램이 실행되는 행동을 담당

# 정리

---

- 객체지향 프로그래밍은..  
변수와, 함수로 이뤄진 클래스파일을 기초로 하며  
이 클래스 파일을 메모리에 적재되면 객체화가 되고  
객체들의 속성과 명령어를 수행 하면서 프로그램이 실행된다.

# Project 1. 계산기 만들기

---

Step 1. UI만들기

Step 2. UI - Class 연결

Step 3. 버튼 액션 만들기

Step 4. 계산기 완성

# Step 1. Make UI







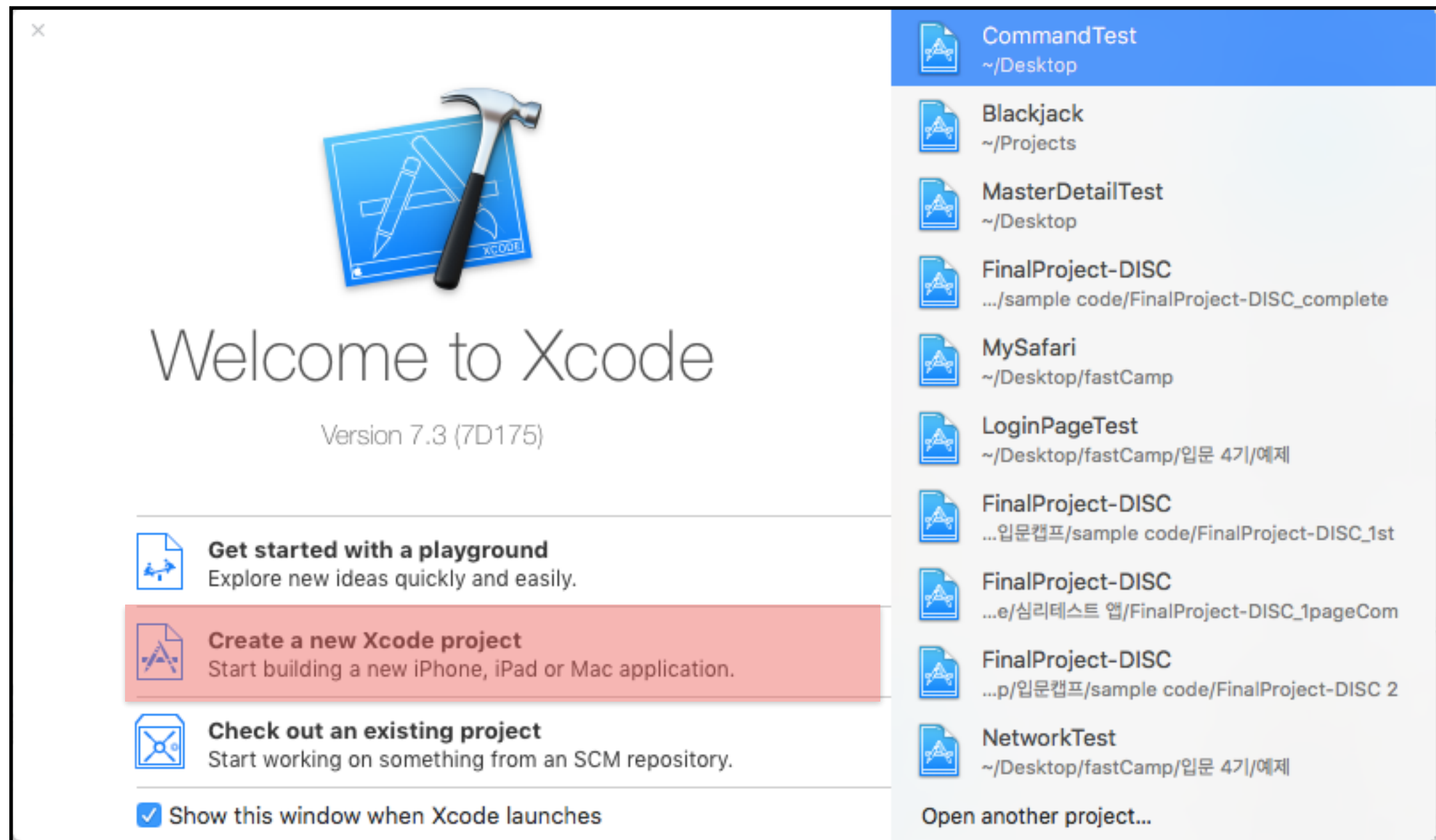
지금 만들라고?  
뭘?  
어떻게?  
나 혼자서?

---

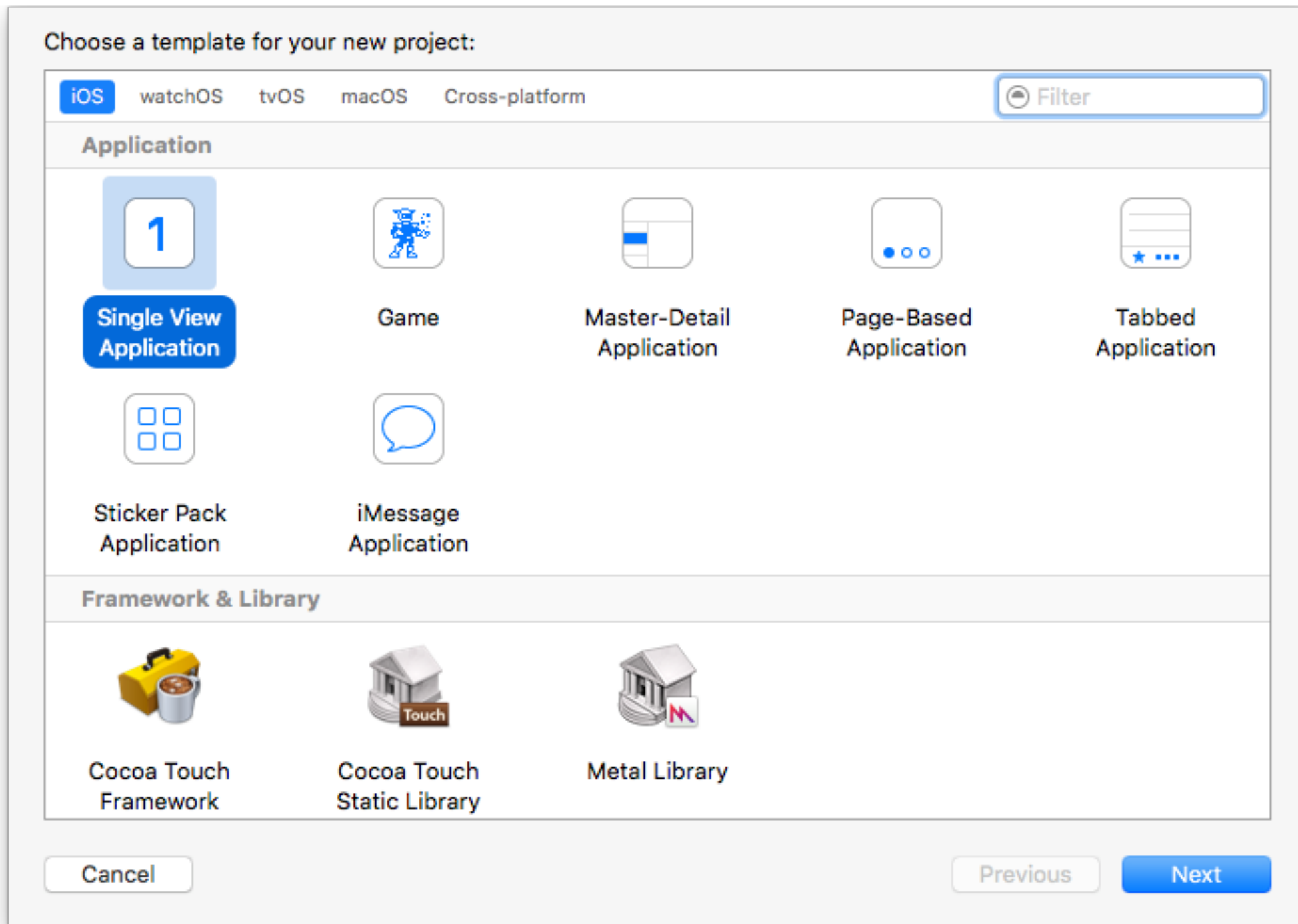
# Xcode 사용법

---

# Xcode - 시작



# 템플릿 선택



# 프로젝트 생성

Choose options for your new project:


Product Name:


Team:

Organization Name:

Organization Identifi...

Bundle Identifier:

Language:  

Devices:  

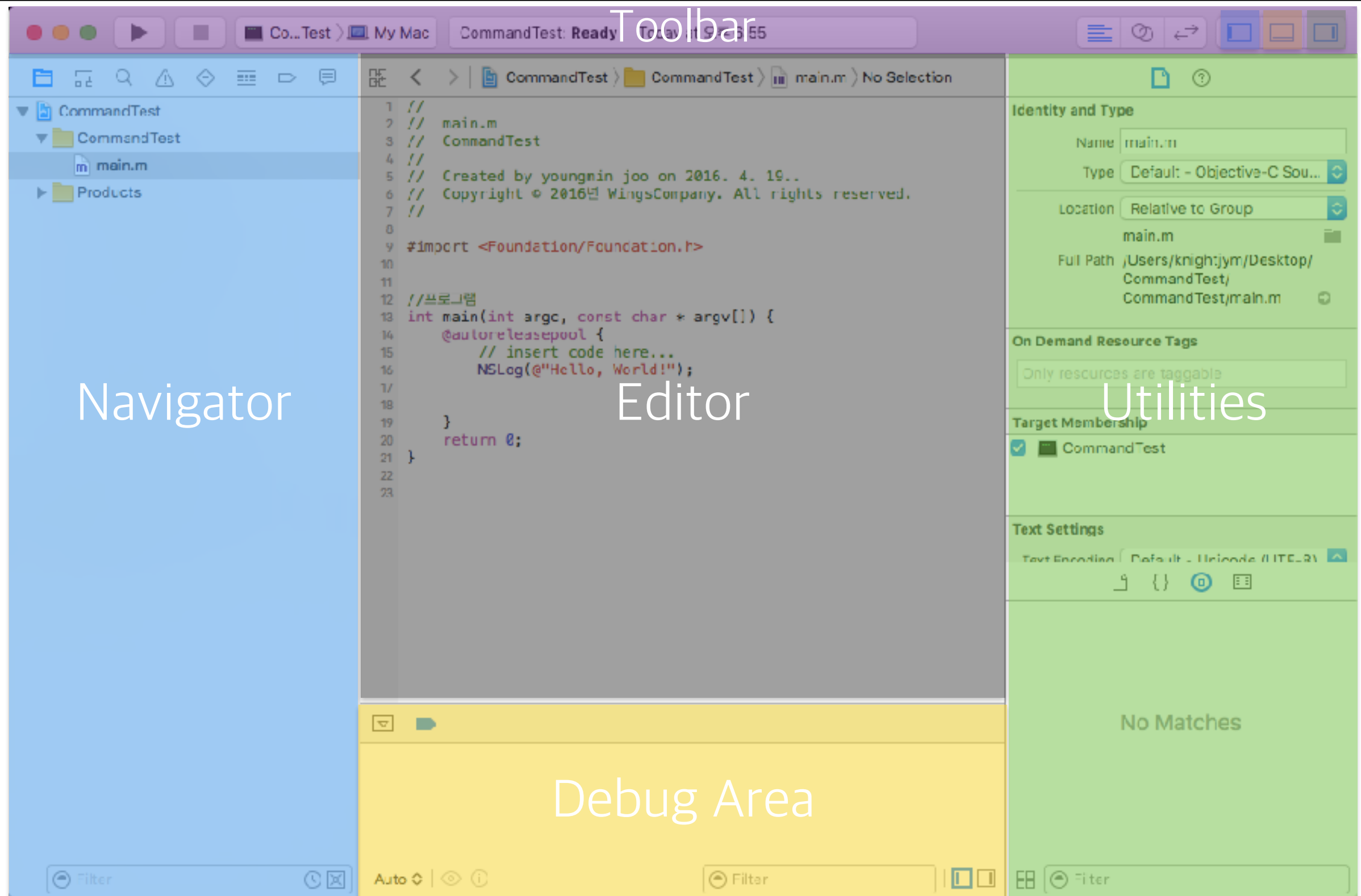
☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests



# Xcode Main Window



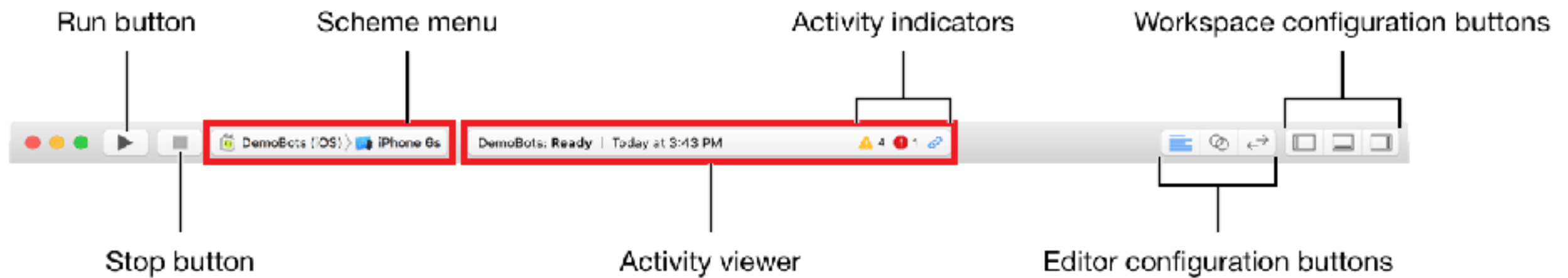
# Xcode Main Window

---

- Navigator : 프로젝트 관리를 위한 도구 모음
- Editor : Project Navigator에서 선택한 파일의 내용을 수정하는 화면
- Debug Area: 프로그램 실행 중 Debugging를 위한 콘솔창
- Utilities : Project Navigator에서 선택된 파일의 상세 정보 및 UI속성 수정등의 작업을 위한 공간

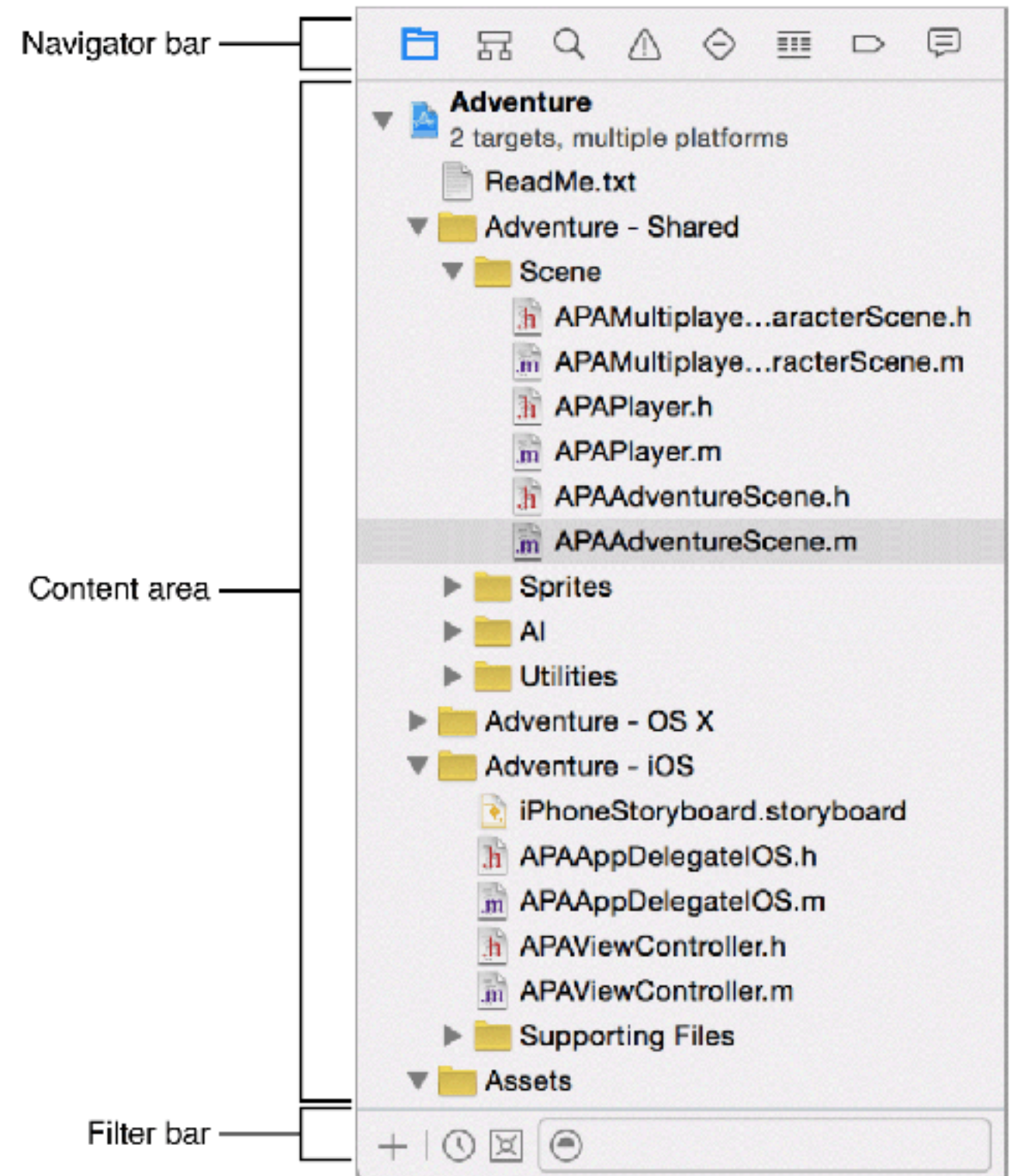
# Toolbar

---

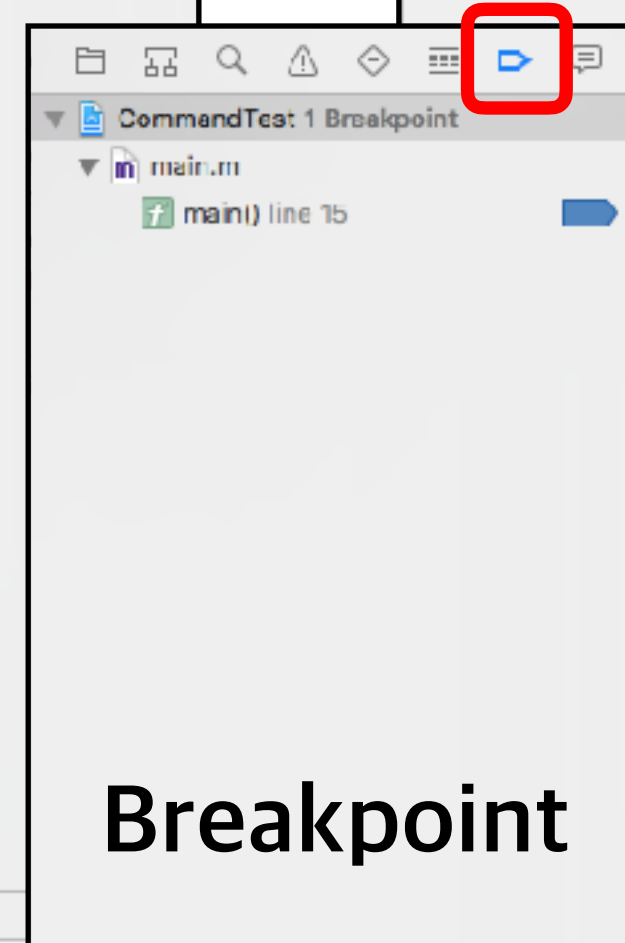
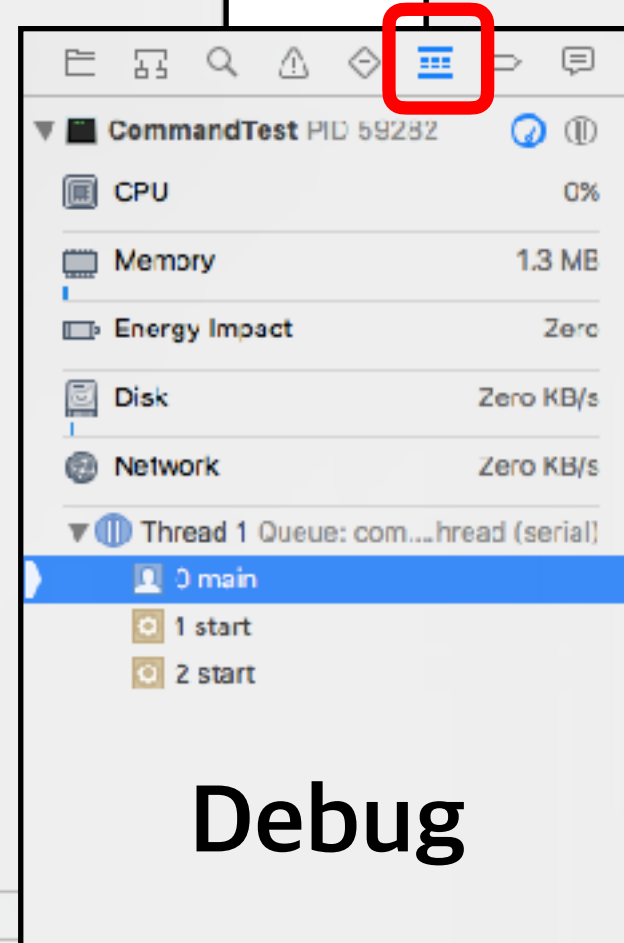
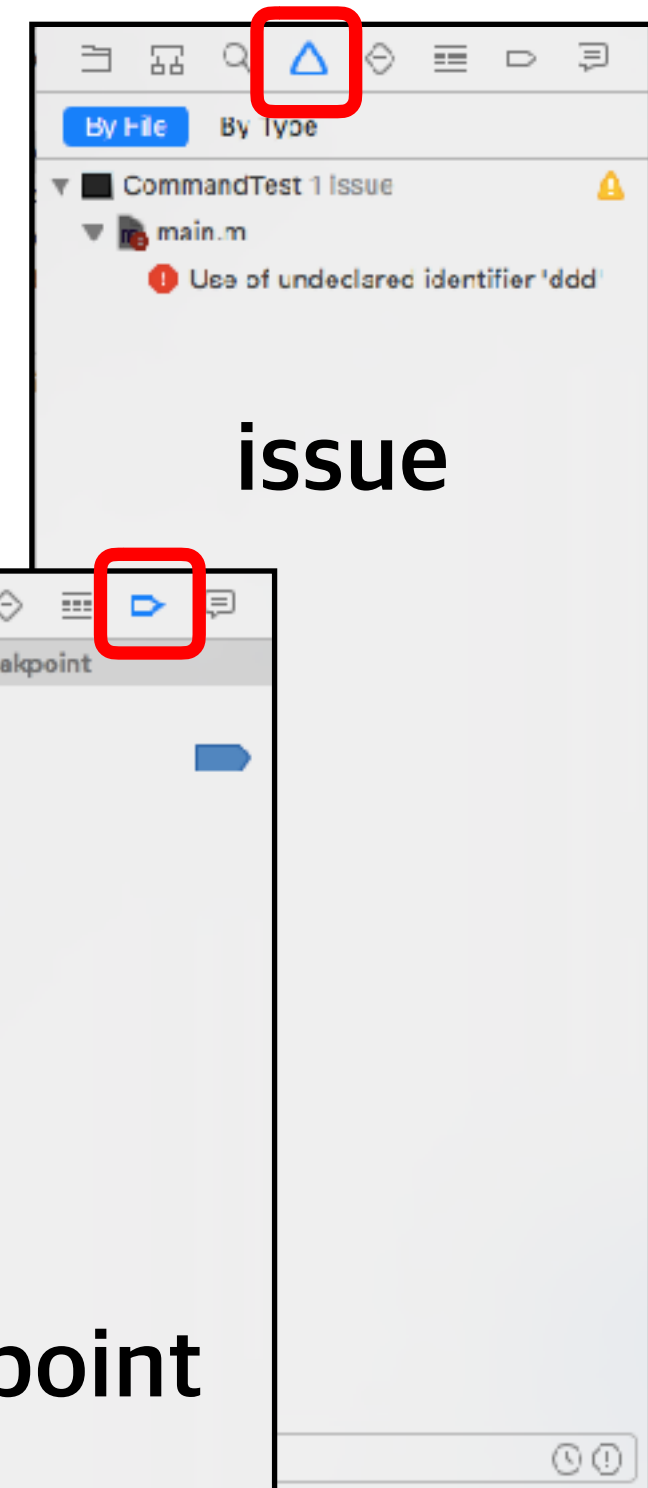
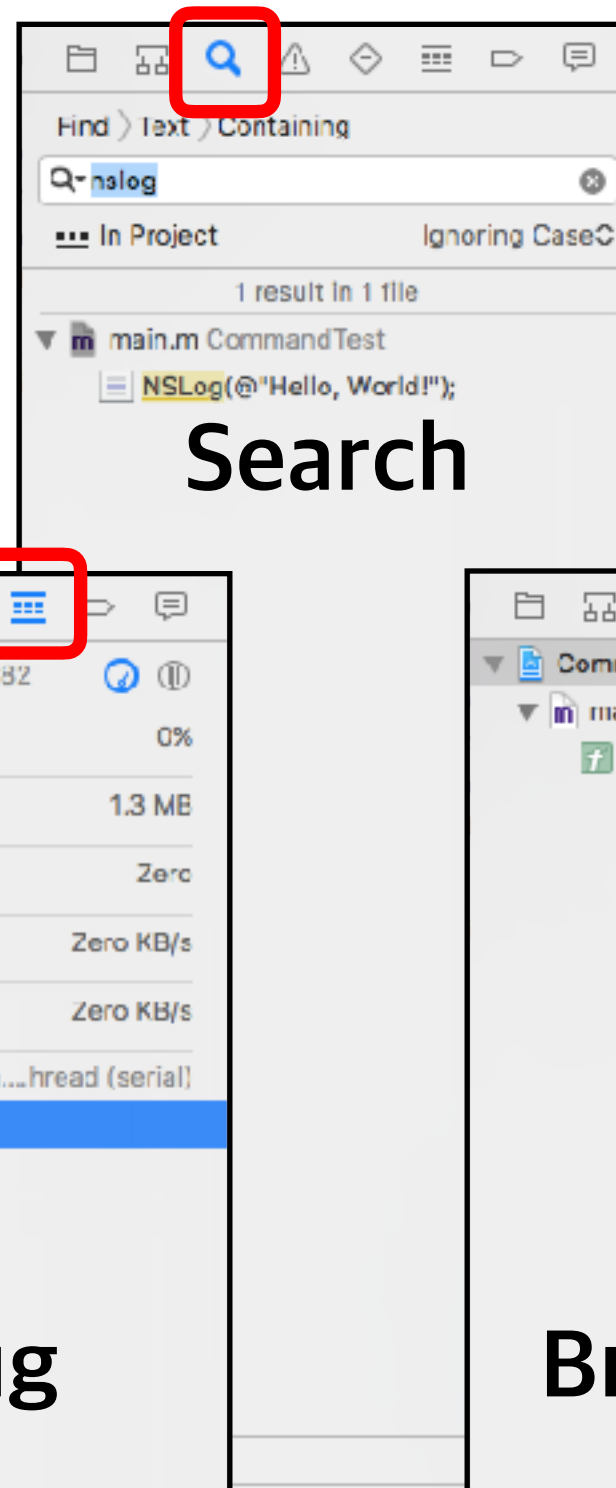
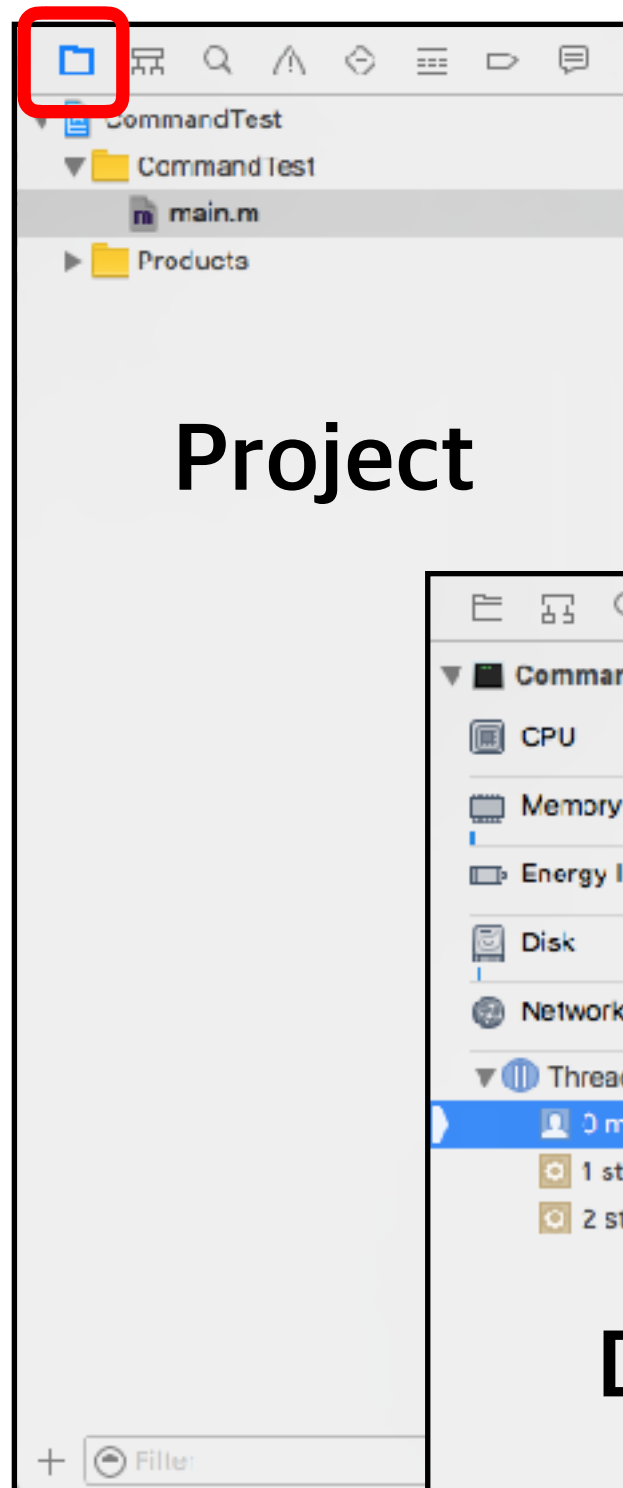




# Navigator Area

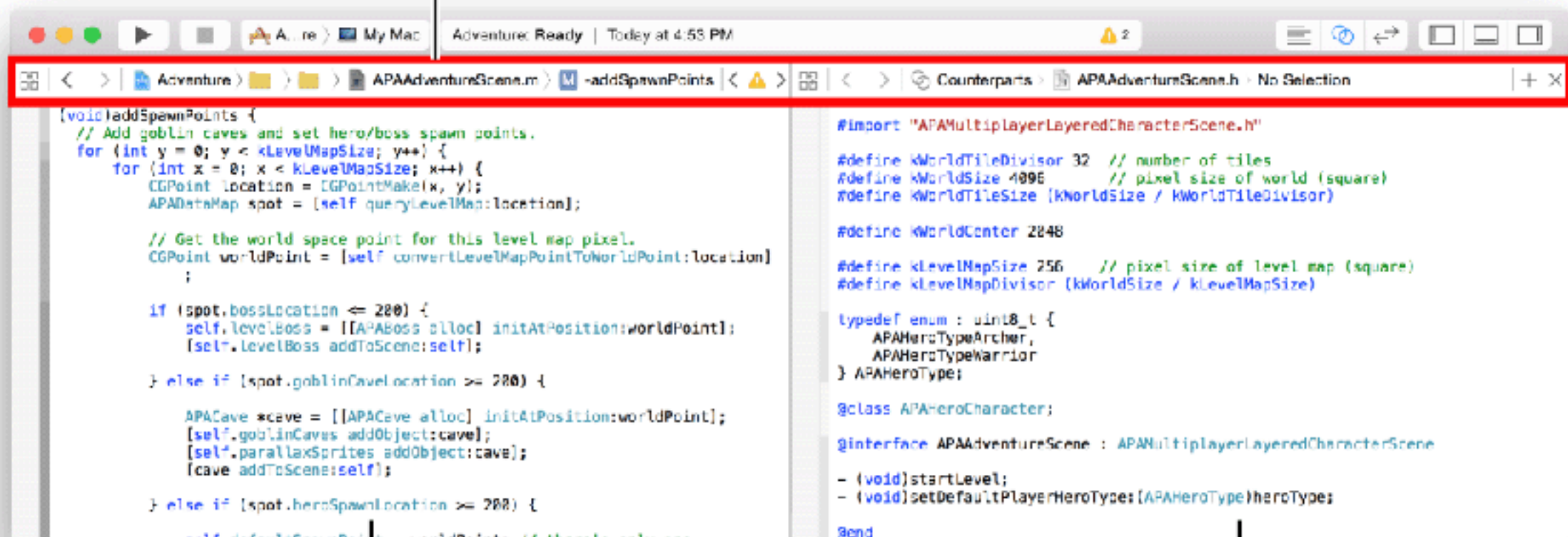


# Navigator bar Menu



# Editor

Jump bars






Standard editor pane

Assistant editor pane

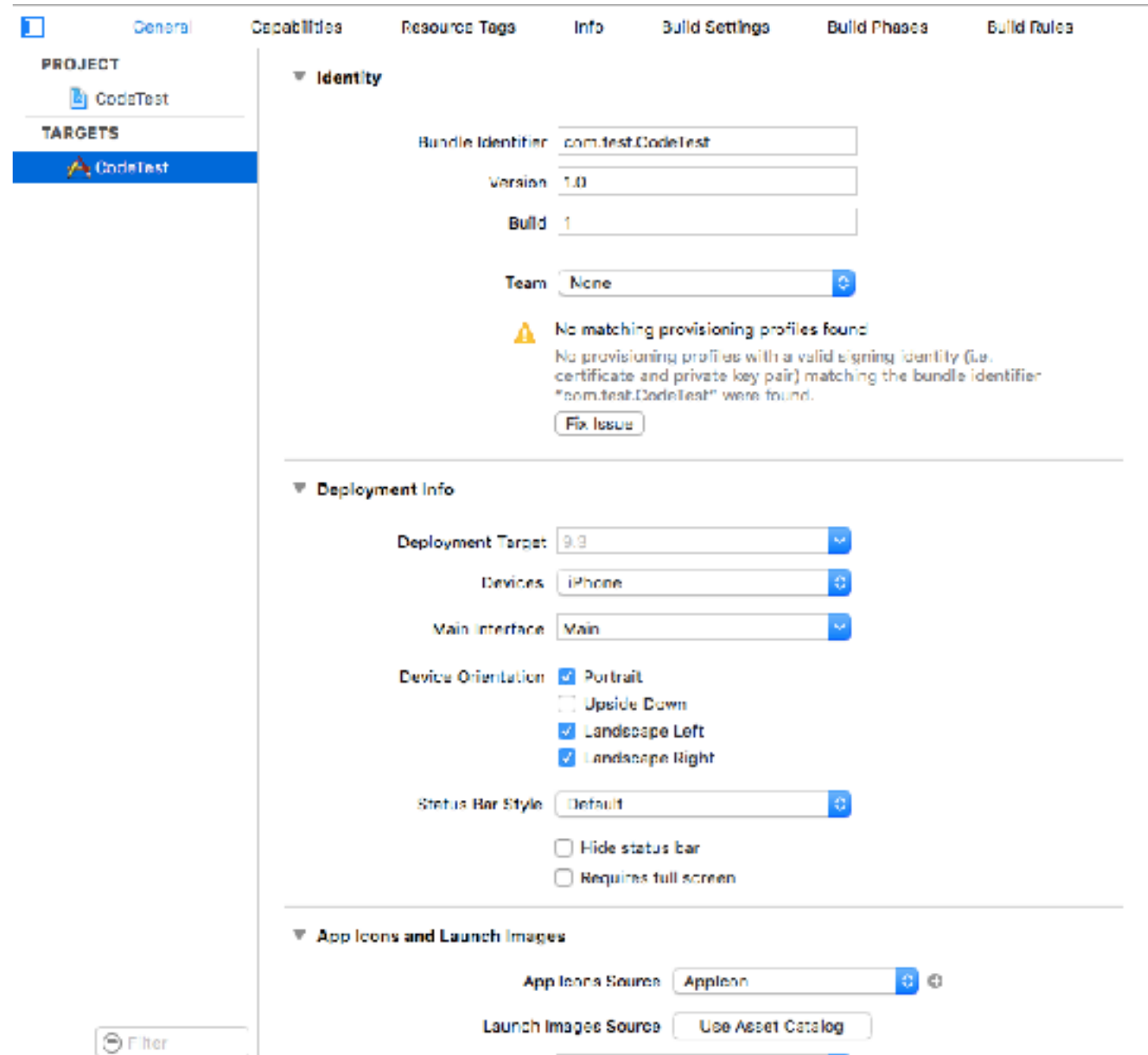
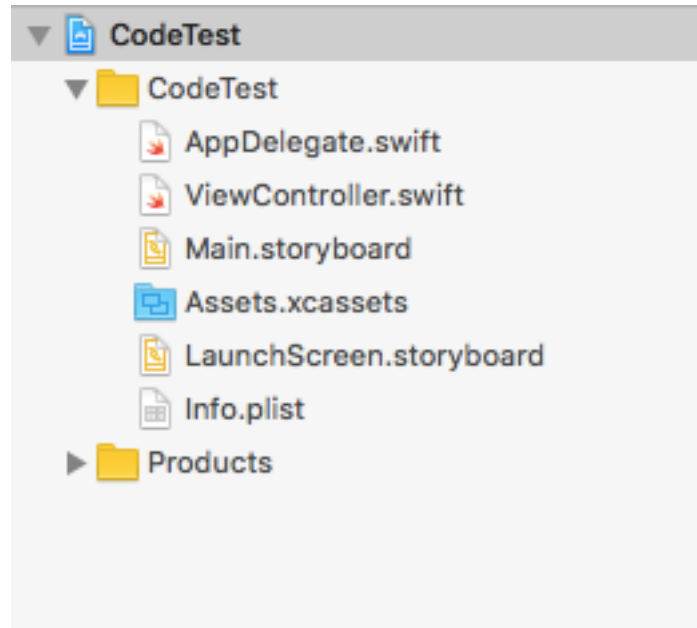
# Editor 상태편집

---

- Standard(  ): fills a single editor pane with the contents of the selected file.
- Assistant(  ): presents a separate editor pane with content logically related to that in the standard editor pane. Use the split controls in the
- Version (  ): shows the differences between the selected file in one pane and another version of that same file in a second pane.

# Project Editor

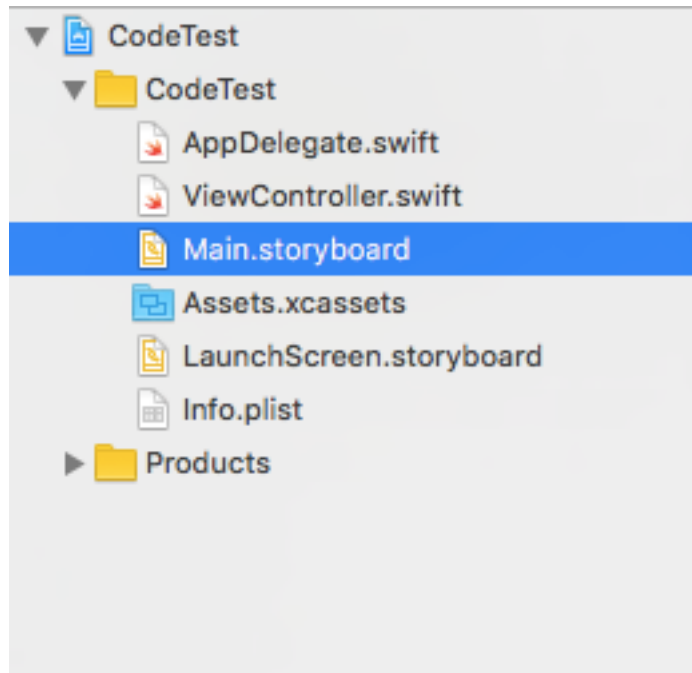
- 프로젝트 설정 변경



# Source Editor

---

- 선택된 파일의 코드를 수정할 수 있다.



```
1 //
2 // ViewController.swift
3 // CodeTest
4 //
5 // Created by youngmin joo on 2016. 5. 3..
6 // Copyright © 2016년 WingsCompany. All rights reserved.
7 //
8
9 import UIKit
10
11 class ViewController: UIViewController {
12
13     override func viewDidLoad() {
14         super.viewDidLoad()
15         // Do any additional setup after loading the view, typically from a nib.
16     }
17
18     override func didReceiveMemoryWarning() {
19         super.didReceiveMemoryWarning()
20         // Dispose of any resources that can be recreated.
21     }
22
23 }
24
25
26
```



# Break point

- debug를 위한 방법
- 여기서 실행을 멈춰라!

```
28 //프로그램
29 int main(int argc, const char * argv[]) {
30     @autoreleasepool {
31         // insert code here...
32
33         printf("여기에서 브레이크 포인트가 실행된다.");
34
35         printf("이코드는 아직 실행되지 않습니다.");
36
37         printf("다음 스텝을 눌러야 실행됩니다.");
38
39     }
40     return 0;
41 }
42
43
44
45
46
```

Thread 1: breakpoint 1.1

CommandTest > Thread 1 > 0 main

▸ A argv = (const char \*\*) 0x7fff5fbff808  
A argc = (int) 1

여기에서 브레이크 포인트가 실행된다. (lldb)

# Break point

The screenshot shows a C program in a debugger. The code is as follows:

```
28 //프로그램
29 int main(int argc, const char * argv[]) {
30     @autoreleasepool {
31         // insert code here...
32
33         printf("여기에서 브레이크 포인트가 실행된다.");
34
35         printf("이코드는 아직 실행되지 않습니다.");
36
37         printf("이코드는 실행되었습니다.");
38
39     }
40
41     return 0;
42 }
43
44
45
46
```

A breakpoint is set at line 35, indicated by a green bar and the text "Thread 1: breakpoint 1.1".

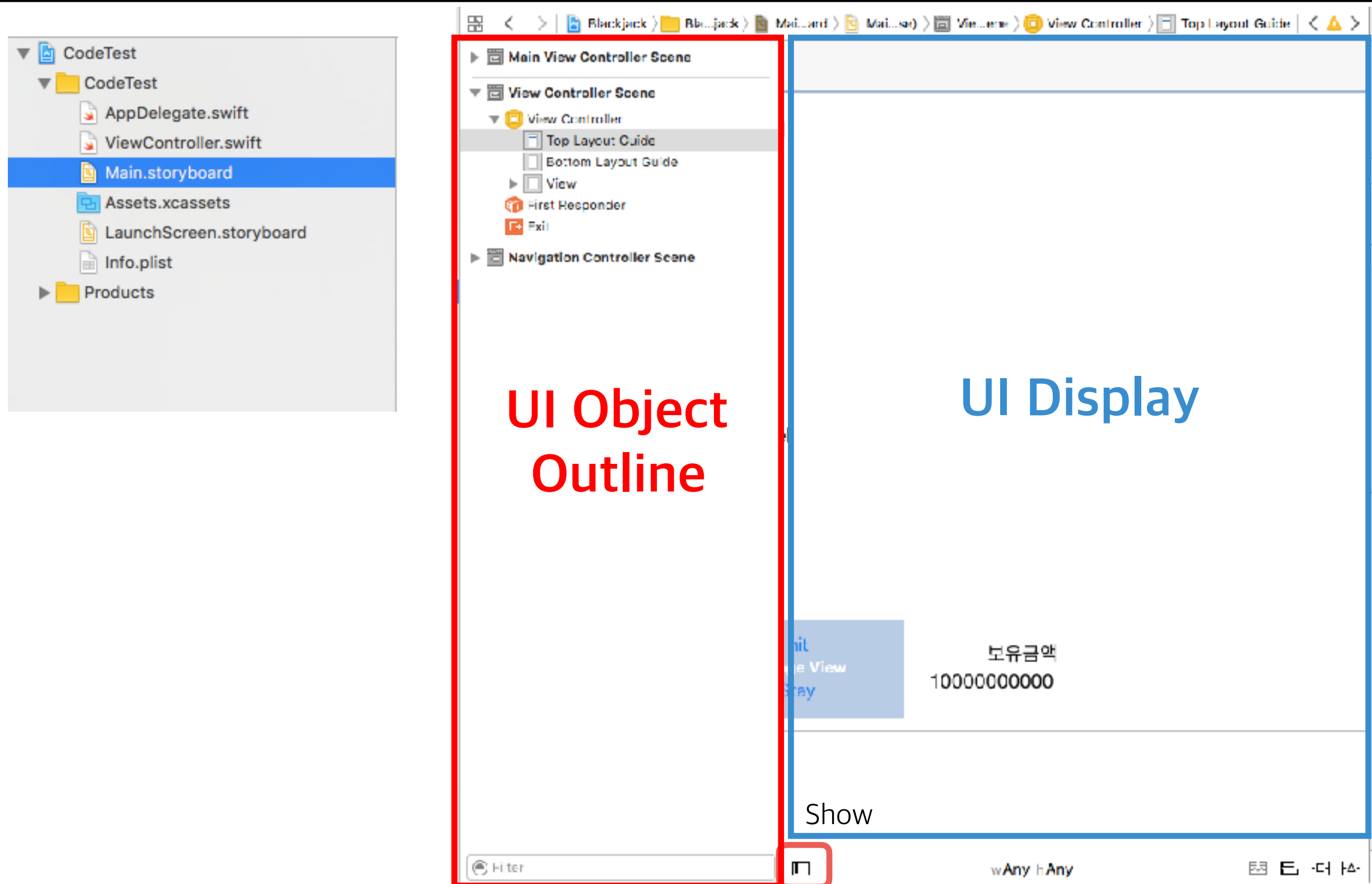
Red text and arrows point to the following buttons in the debugger toolbar:

- break point enable**: A blue button with a right-pointing arrow.
- continue**: A button with a right-pointing arrow and a vertical bar.
- next Step**: A button with a right-pointing arrow and a small upward-pointing arrow.

The debugger status bar shows "CommandTest > Thread 1 > 0 main". The console output shows the message "여기에서 브레이크 포인트가 실행된다. (lldb)".

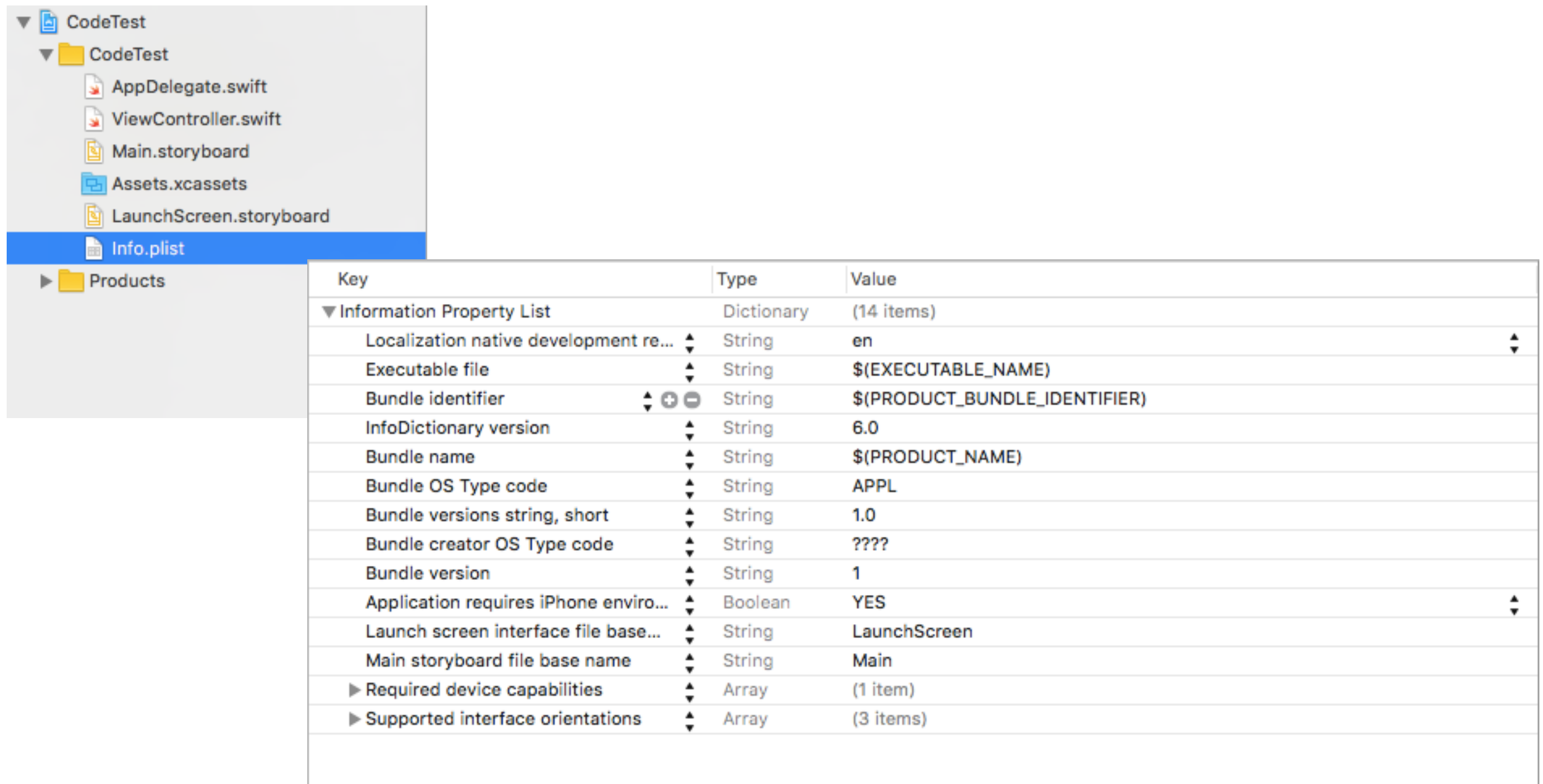


# Interface Builder



# Property list Editor

- property list(plist)파일 편집

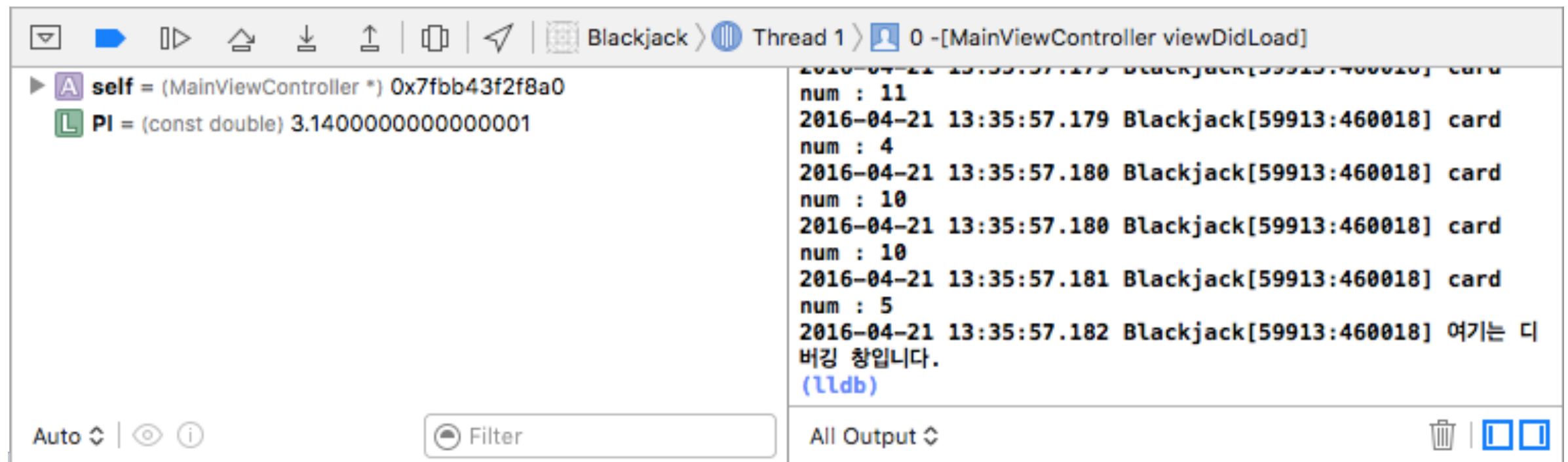


Key	Type	Value
▼ Information Property List	Dictionary	(14 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)

# Debug Area

## Variables View

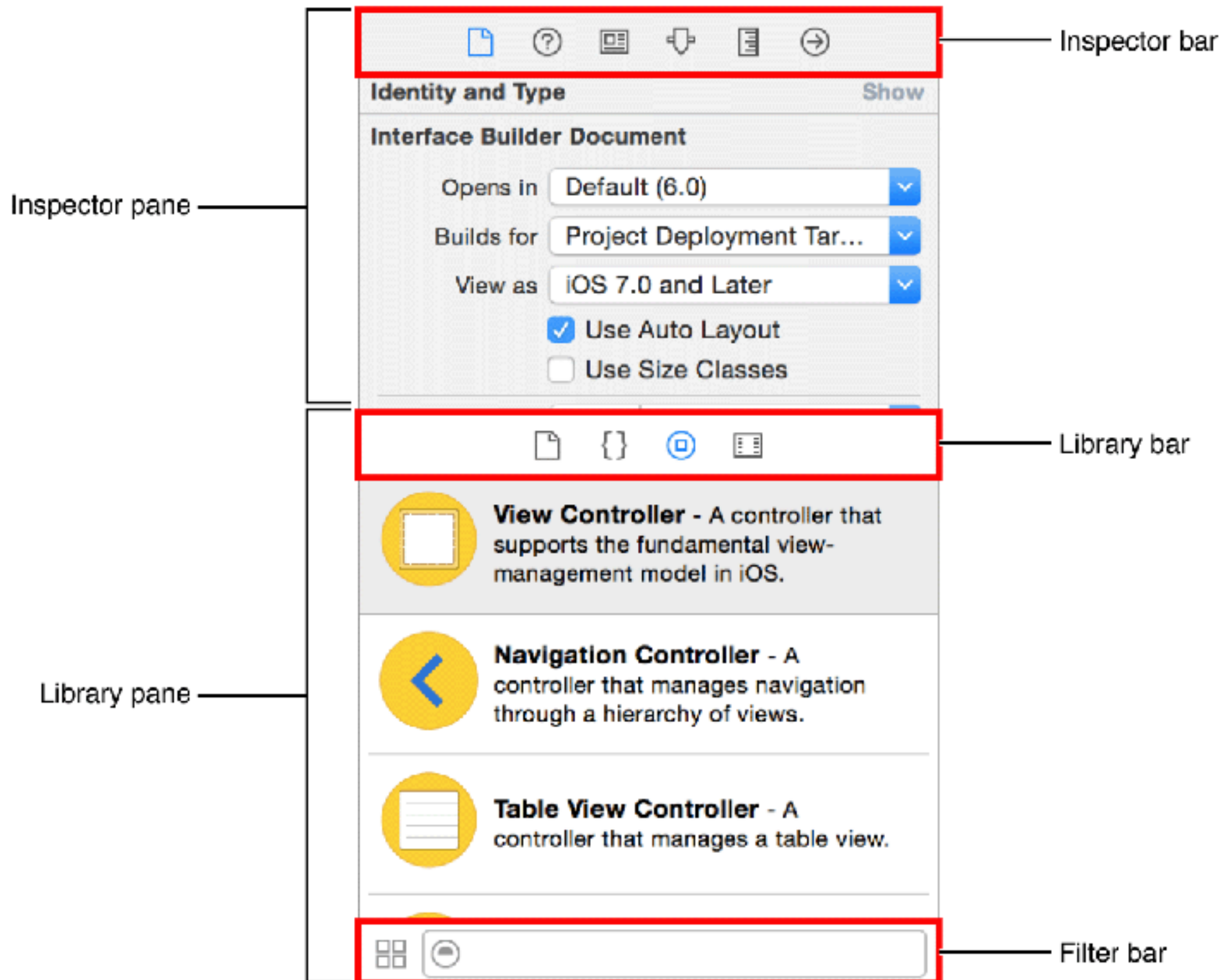
Breaking point로 실행 중 해당 변수의 값을 확인 가능



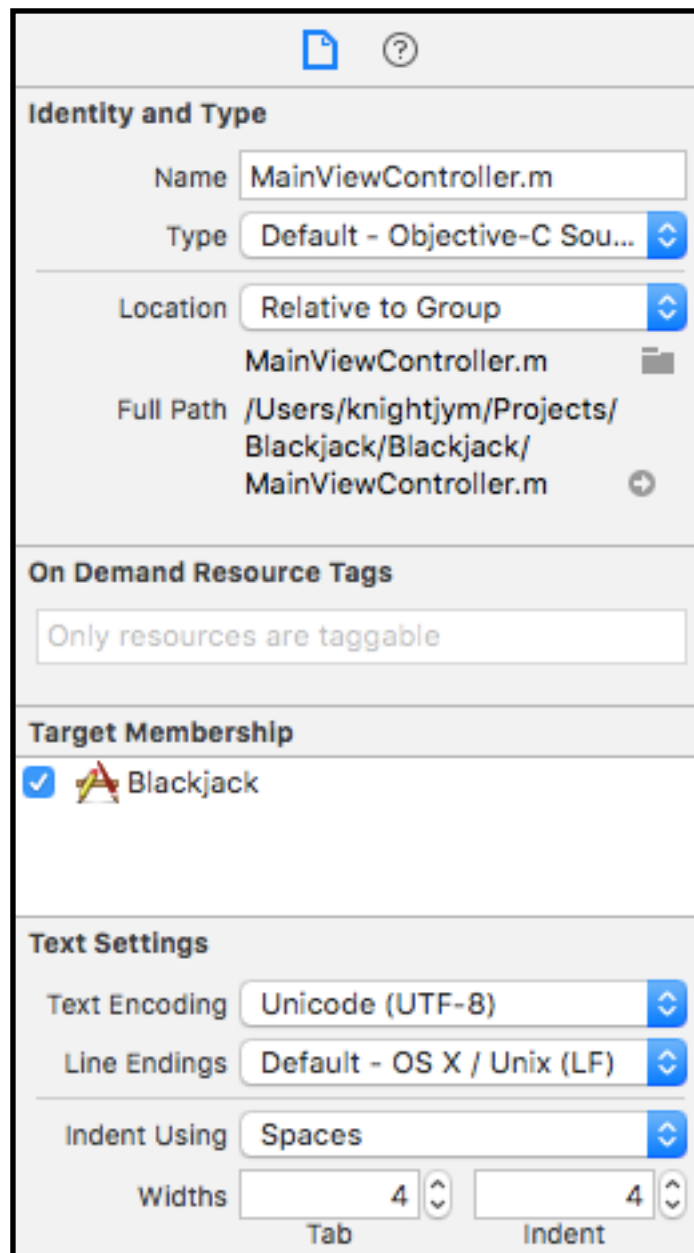
## Console

프로그래머의 log출력과 직접메소드 실행이 가능

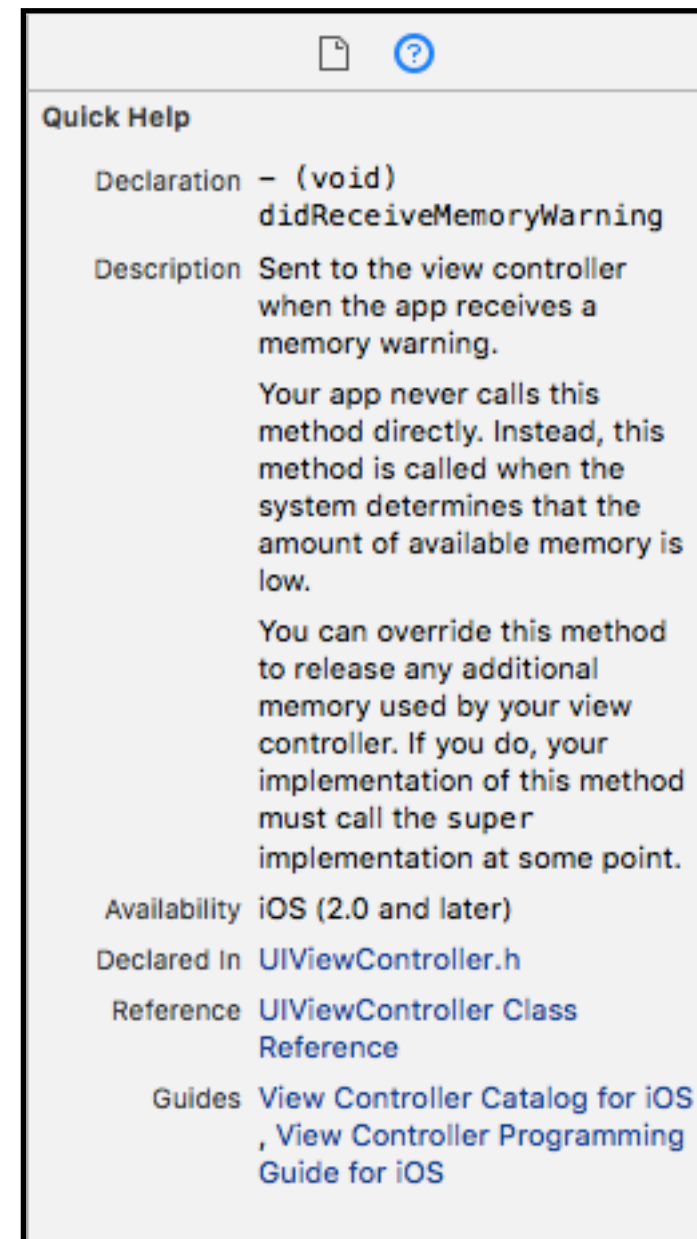
# Utilities



# Utilities - Inspector1

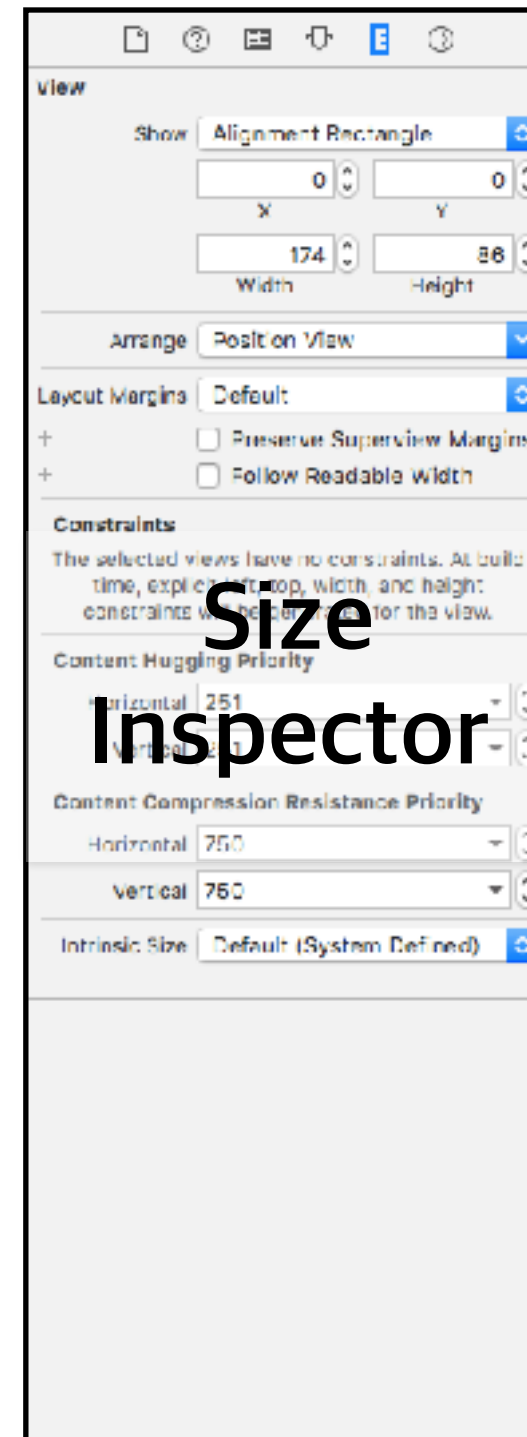
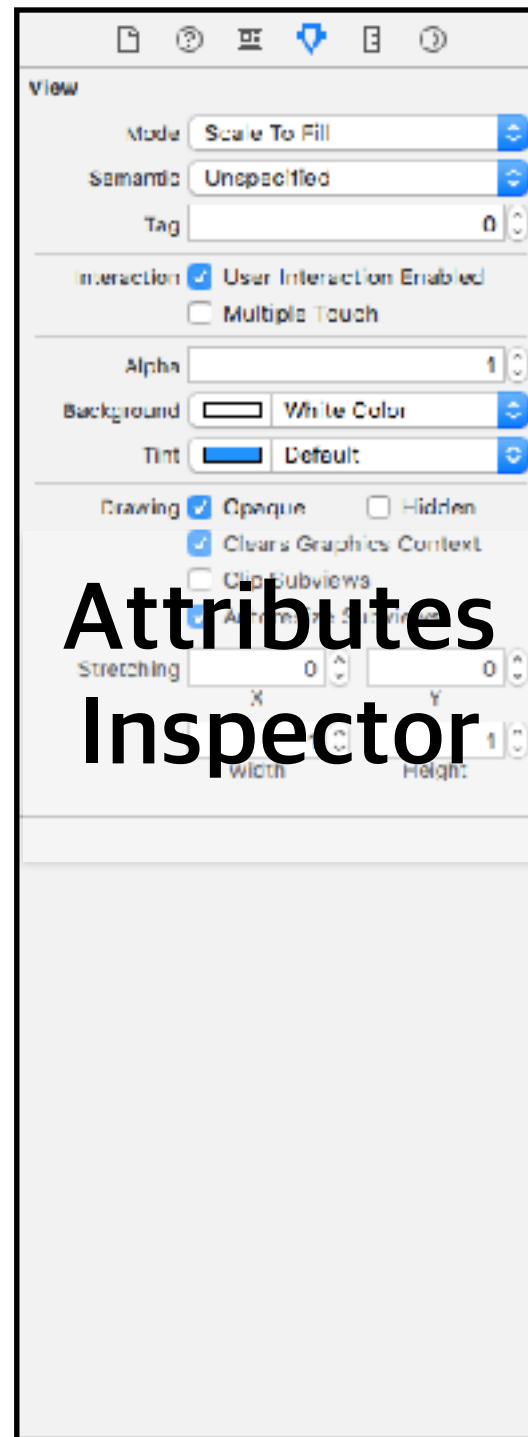
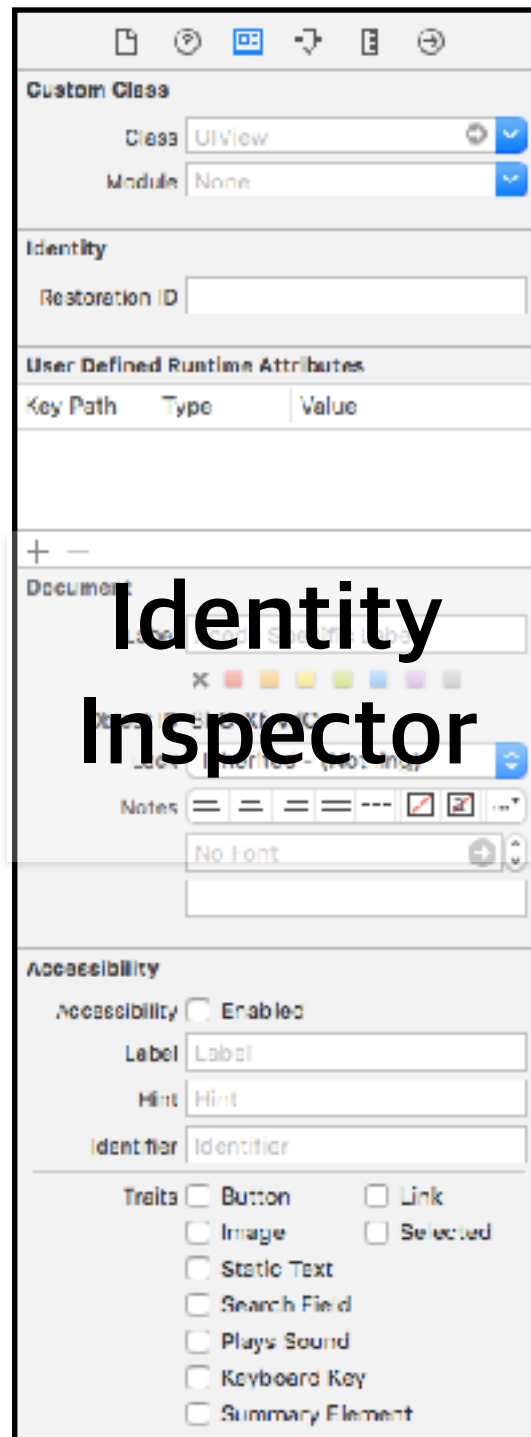


File Inspector



Quick Help Inspector

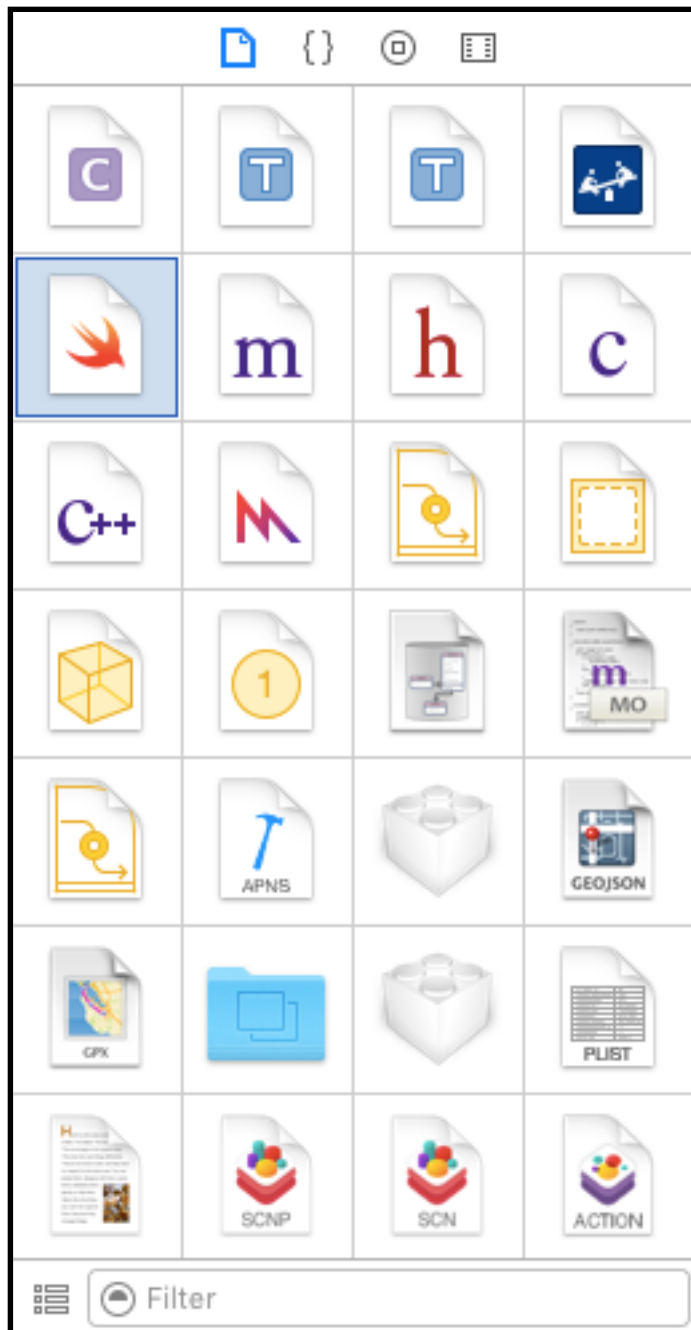
# Utilities - Inspector2 ver UI



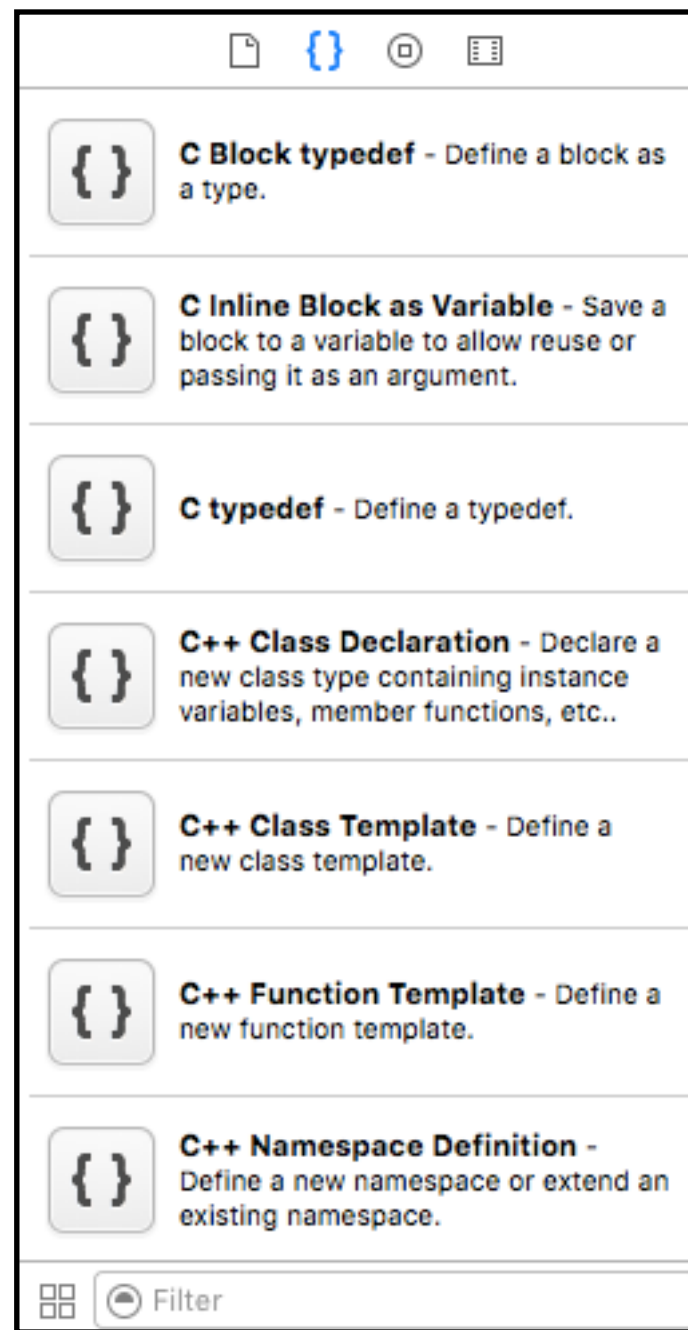


# Utilities - library

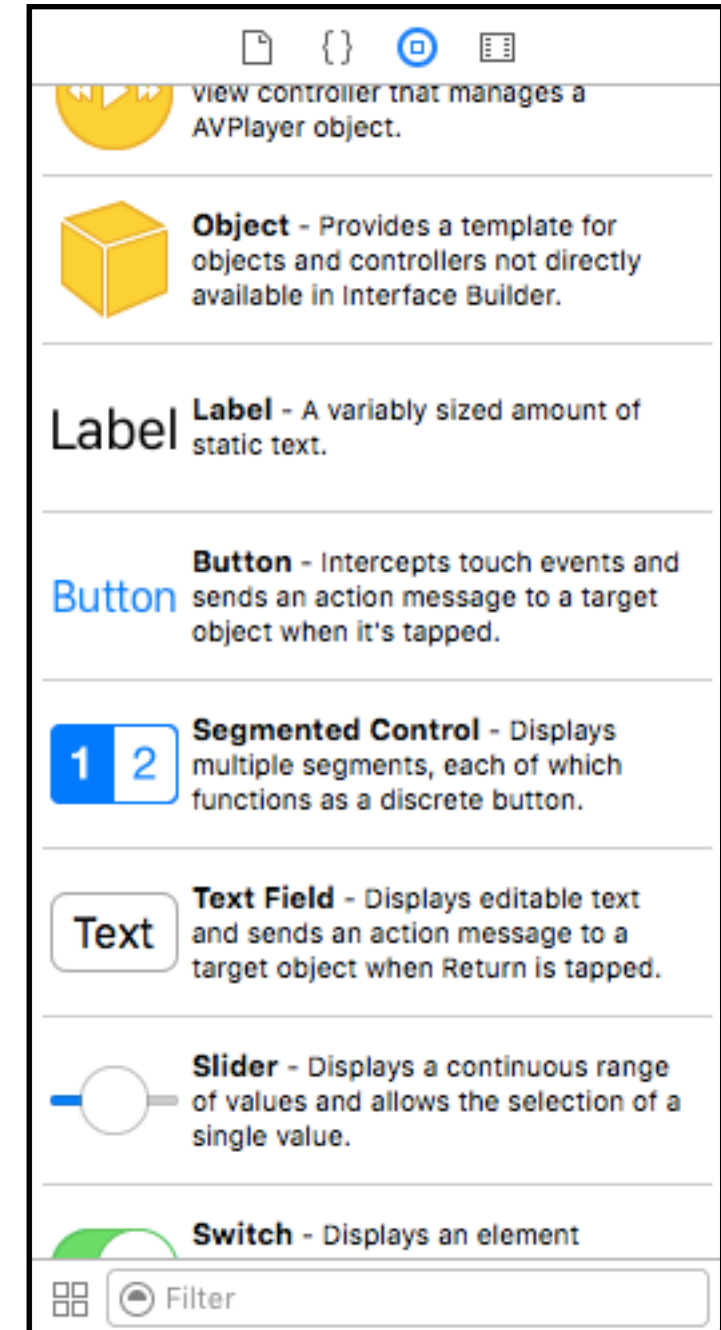
## File Template



## Code Snippet



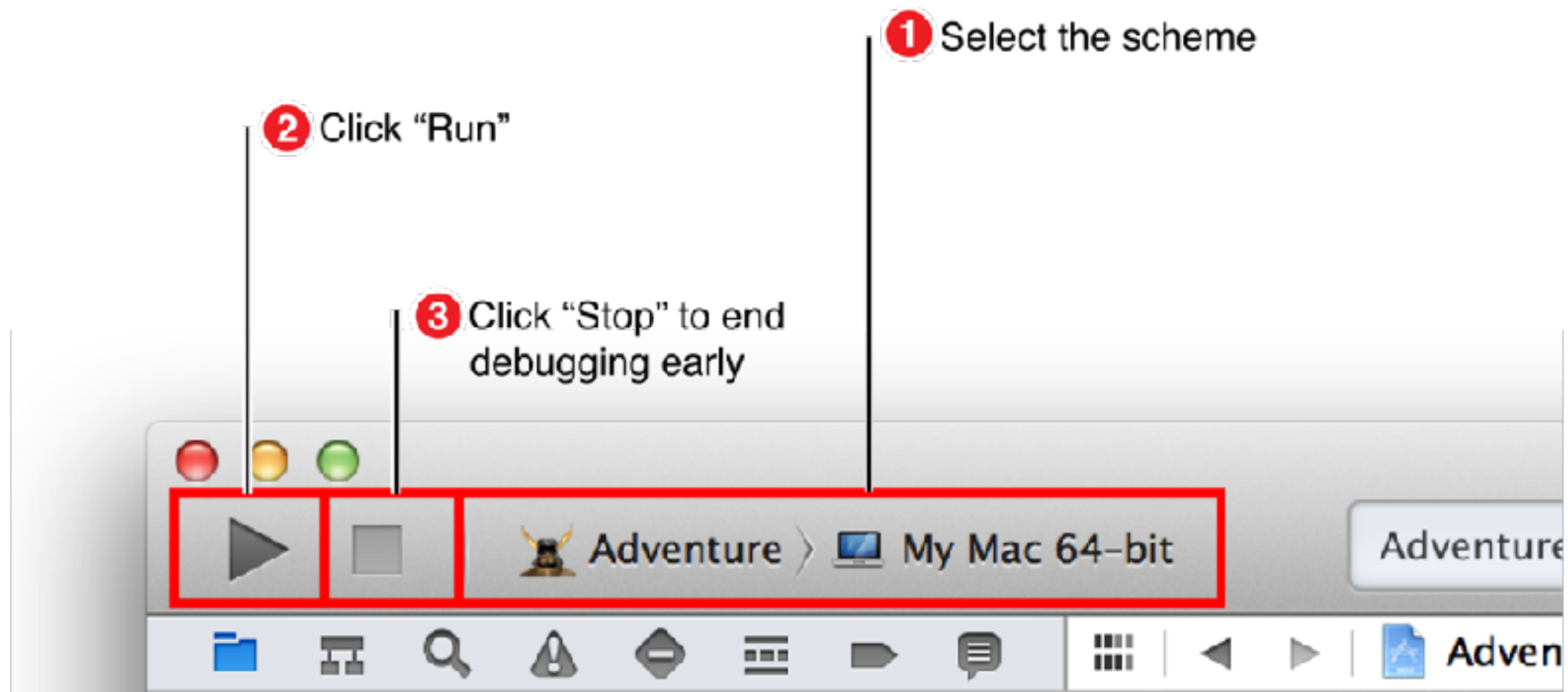
## Object



# 빌드 & 런

---

1. Select an active scheme and destination.
2. Click Run to build and run your code with the active scheme.
3. Use the Stop button to stop an in-progress build or end the current debugging session.



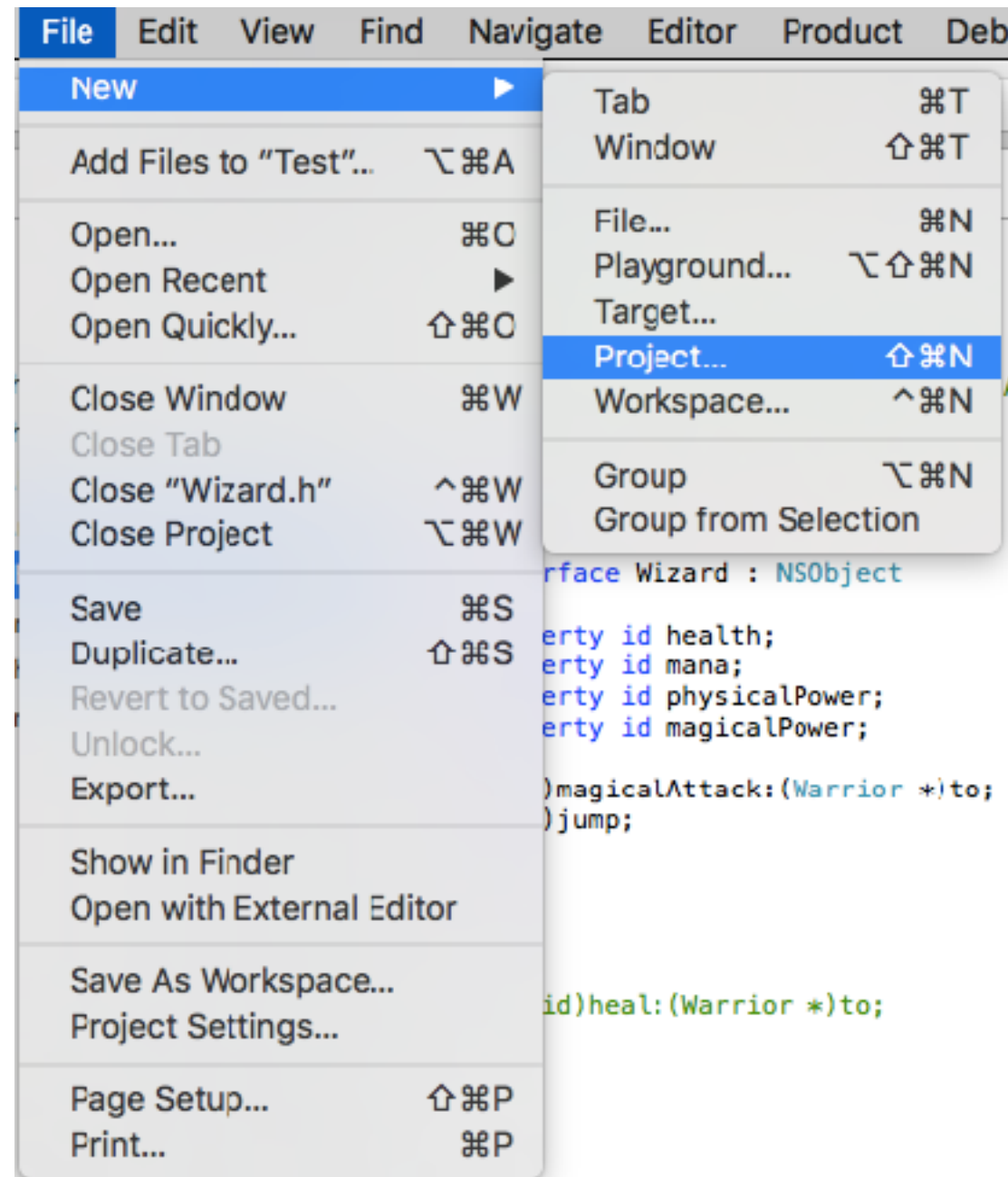


---

# Xcode가지고 놀기

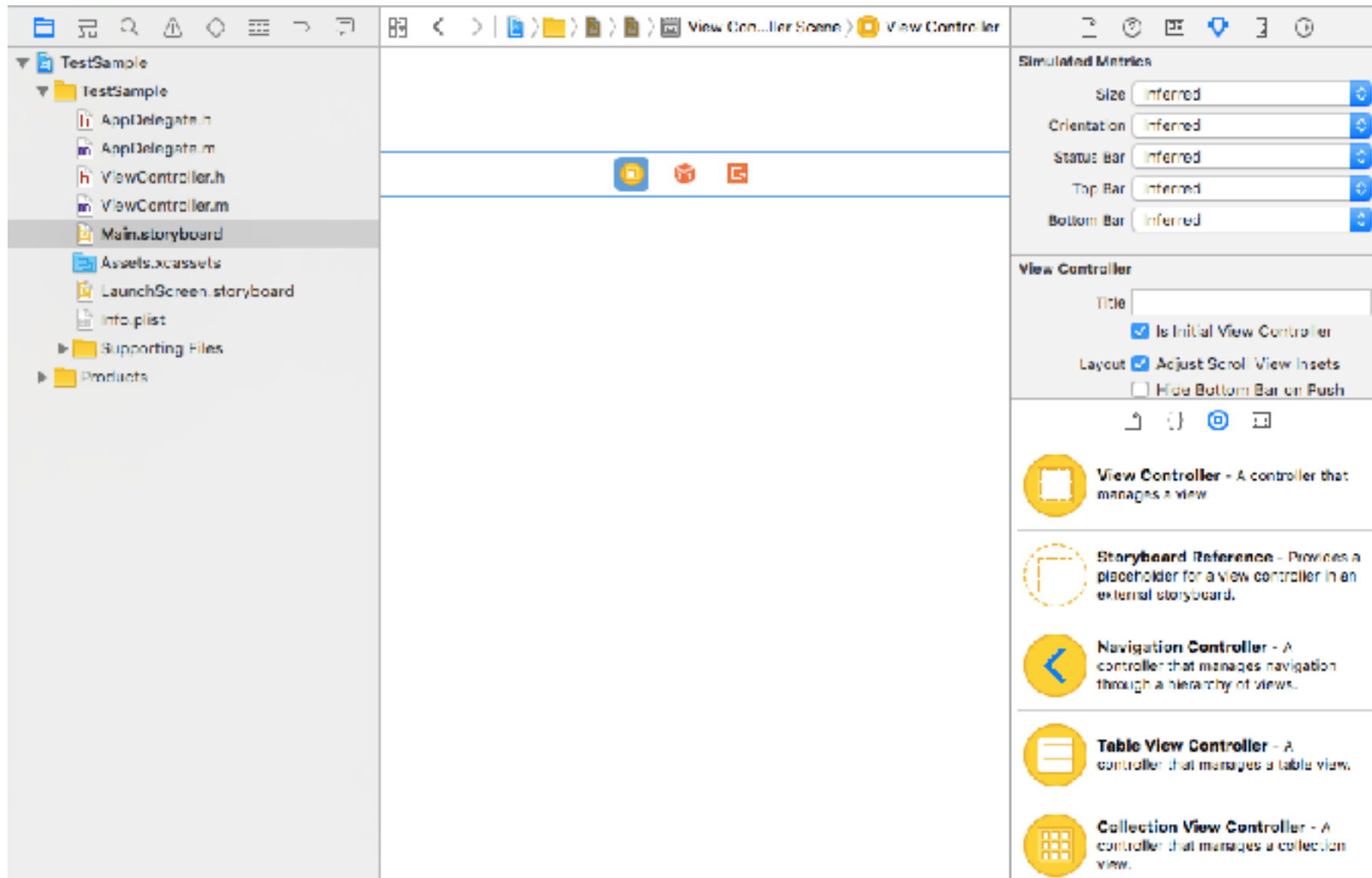
---

# 새프로젝트 만들기



# Storyboard가지고 놀기

- Utilities - library - Object들을 가지고 놀아보세요



# 따라해봐요

---

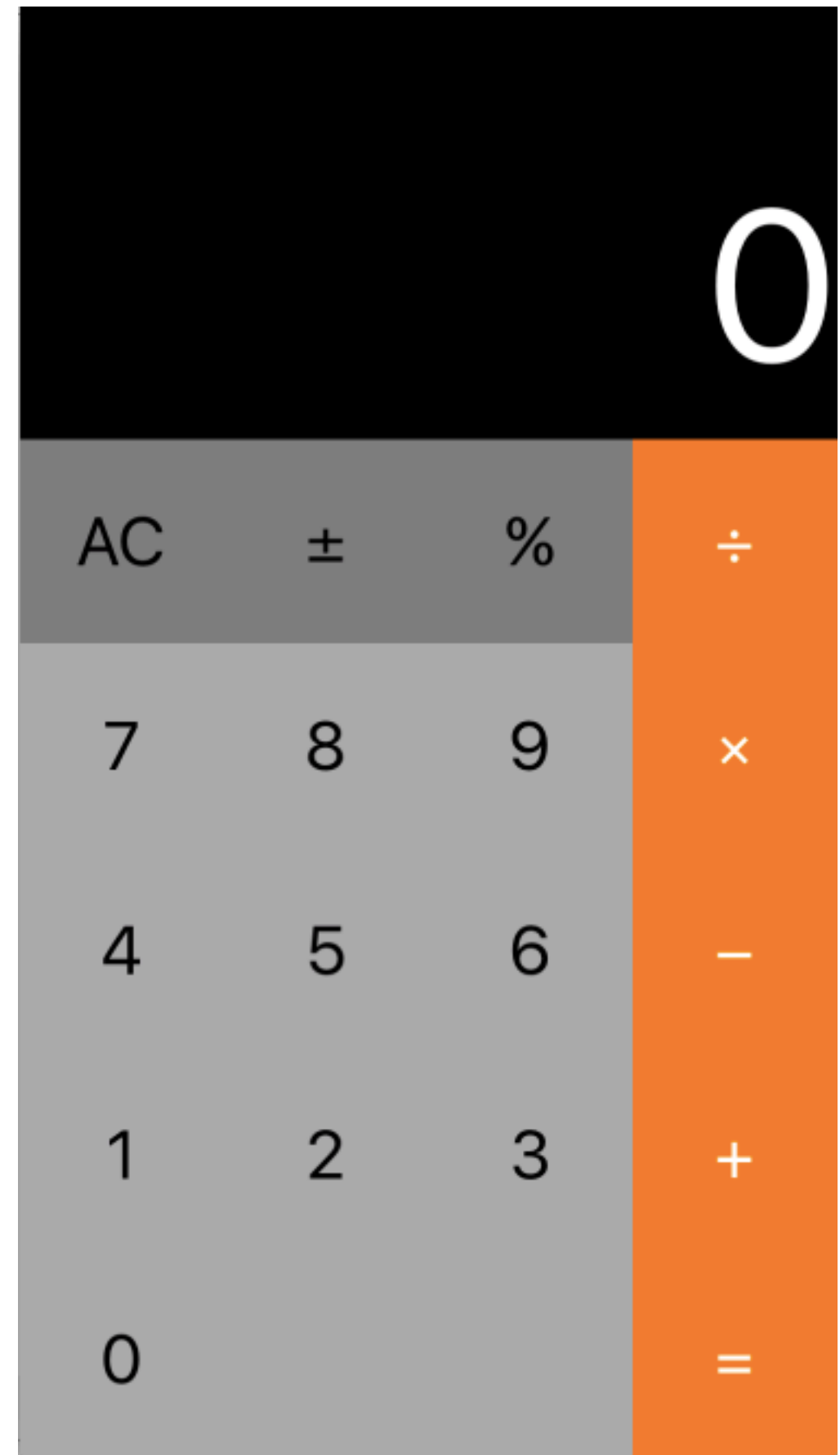
- break point 찍어보기
- build 해보기
- run 해보기

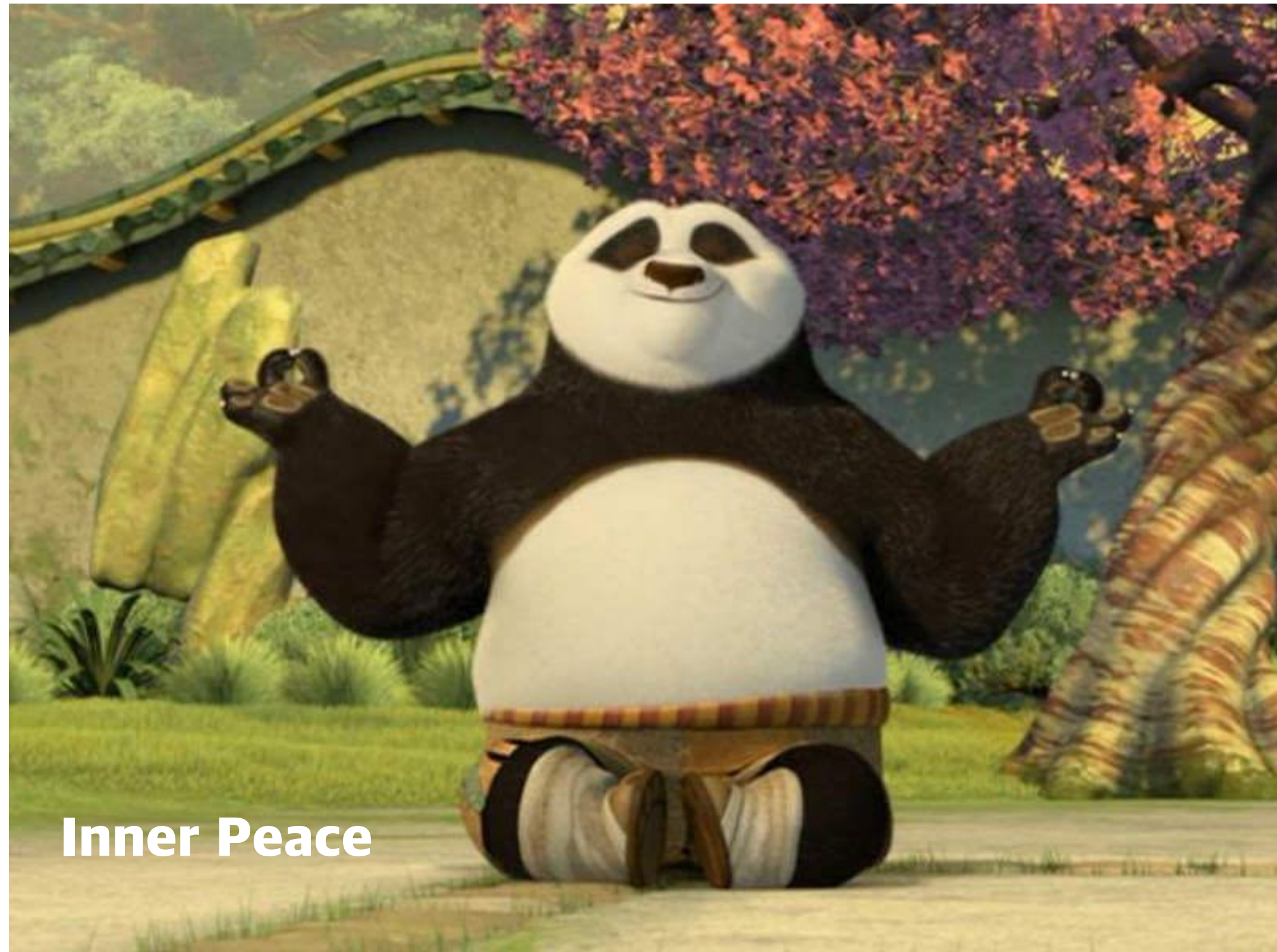
# Hello World

---

- hello world 찍어보기 (Log And UI)

# Step 1. Make UI





**Inner Peace**



---

# 변수 & 함수

---

# Swift Class Architecture

```
class ClassName : superClass
{
    var vName1 = "1"
    var vName2 = 4

    func fName1() - > Any
    {

    }

    func fName2(_ ani:Bool)
    {

    }
}
```

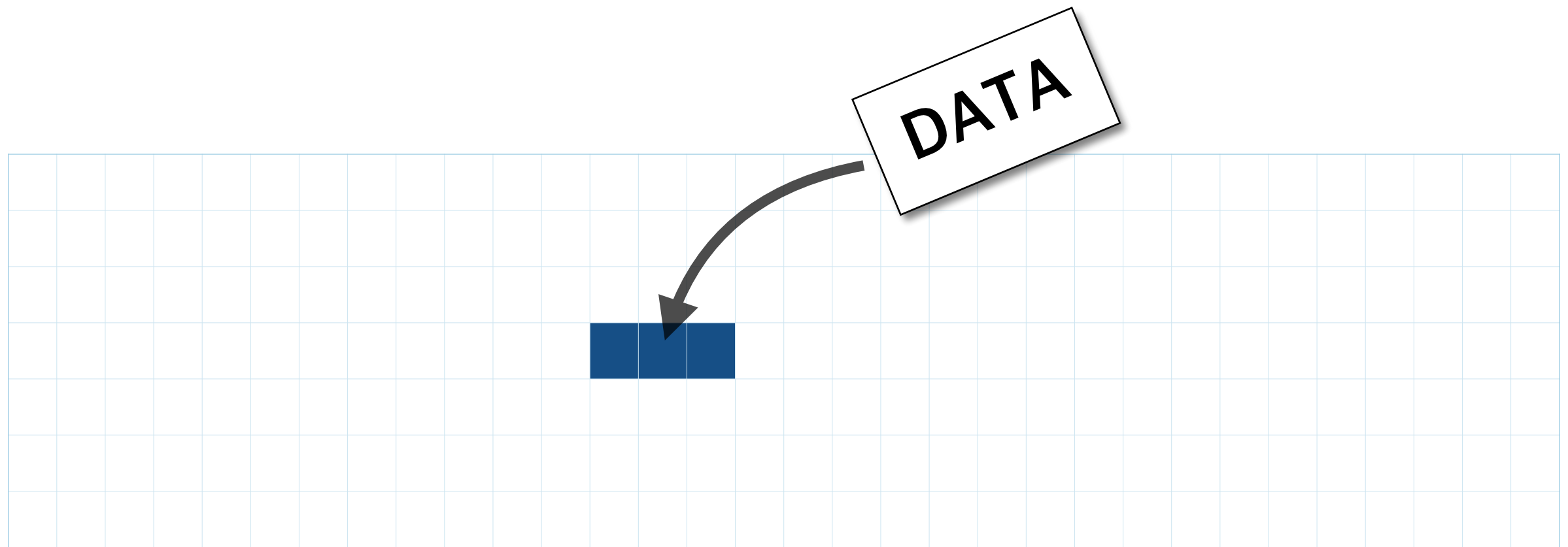
<ClassName.swift>

# 변수 & 함수

---

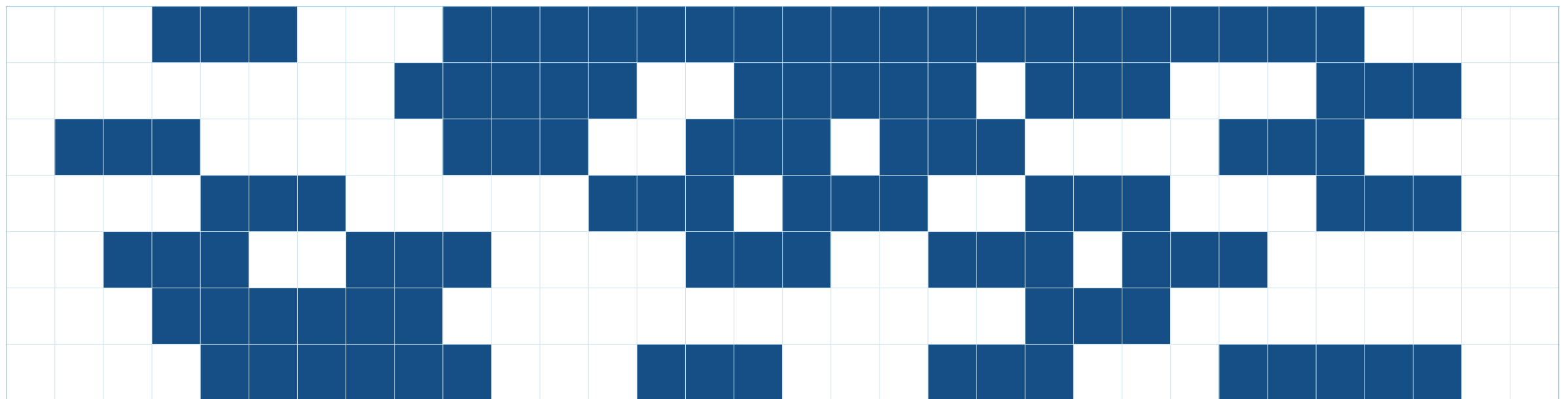
- 변수 : 프로그램에서 데이터의 저장공간을 담당
- 함수 : 프로그램이 실행되는 행동을 담당

# 변수



<메모리>

각 메모리 안에는 어떤 데이터가 들어있을까요?  
조금 전 넣은 데이터는 어디 일까요?



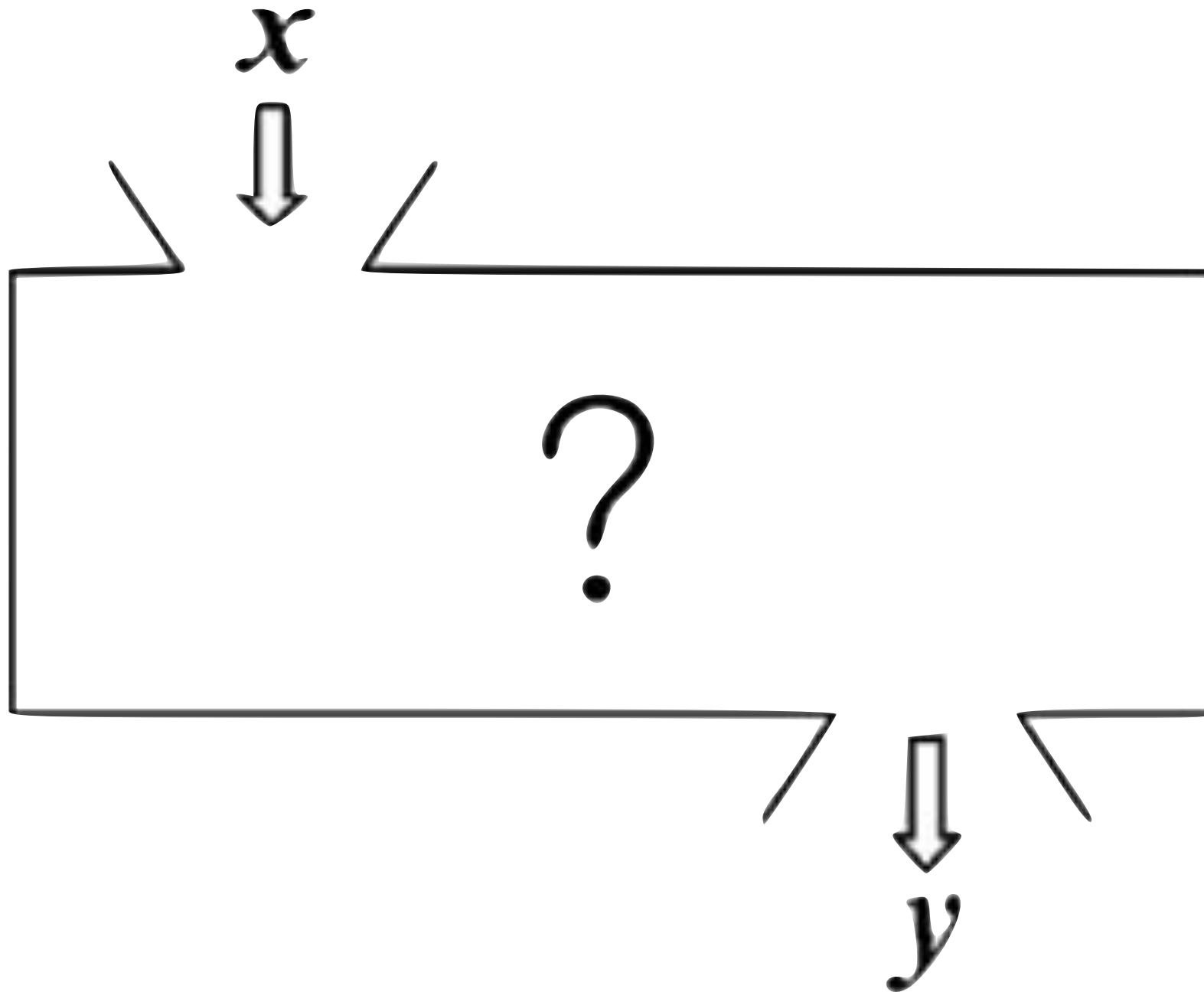
<메모리>

- 변수를 만드는데 있어 필요한 것은?

키워드 + 변수 명(Name) + 변수 타입(Type)

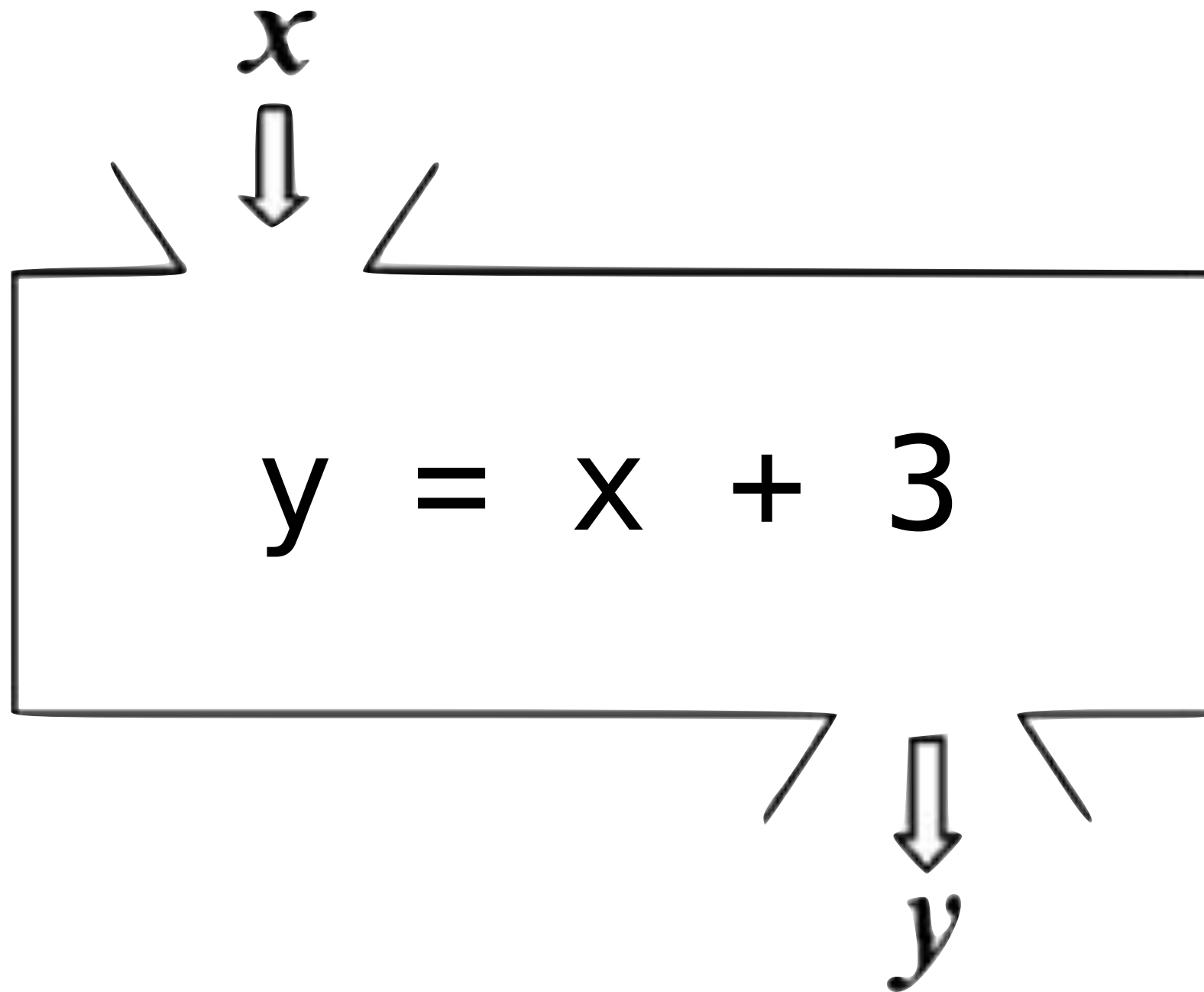
문법 : `var vName:Any`

# 함수

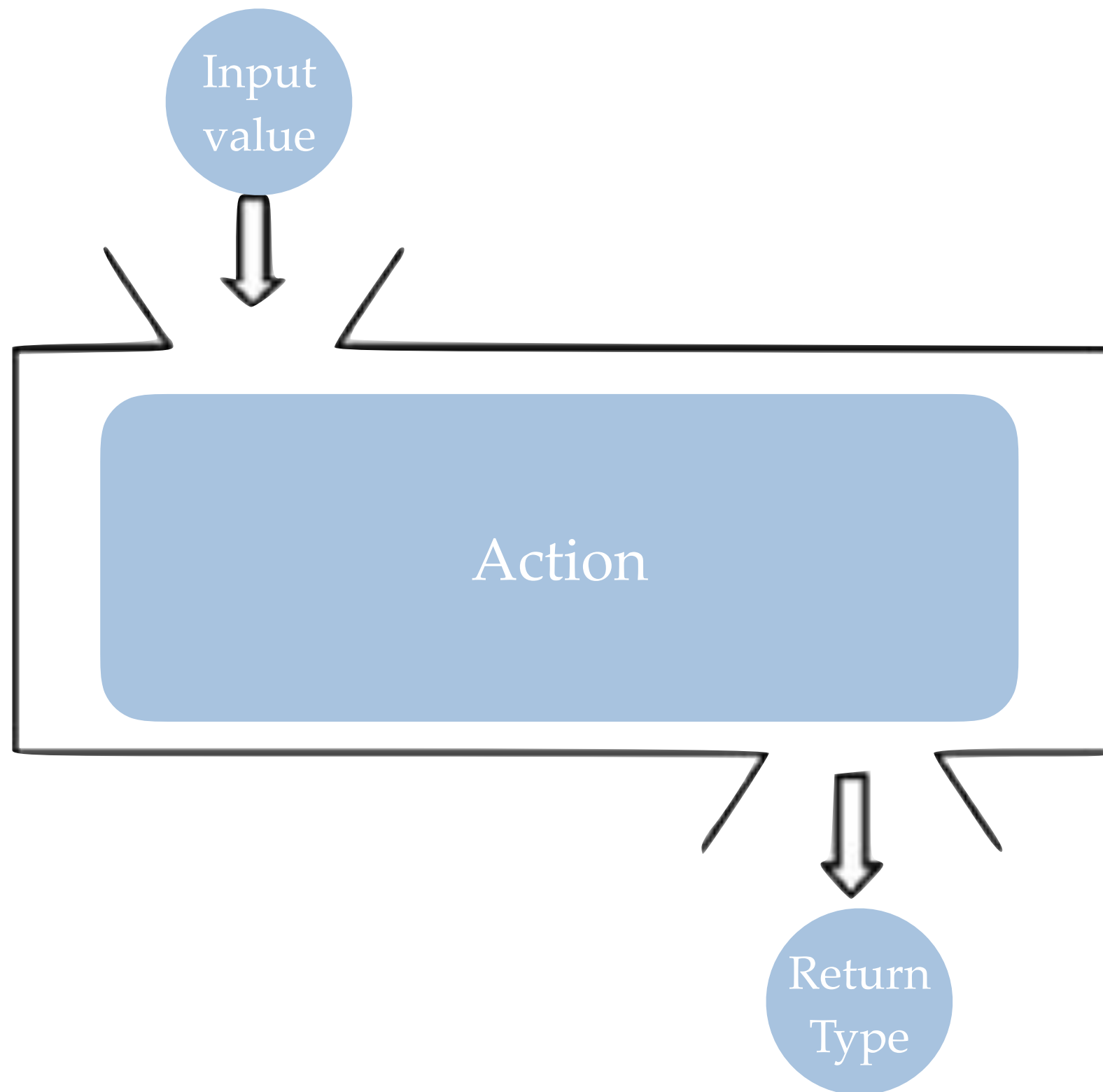




# 함수



# 함수



- 함수 만들기 위해 필요한것?

키워드 + 함수명(Name) + 입력값(Input Value) +  
함수 내용(Action) + 결과타입(Return Type)

문법 : `func vName(_ parameter: Any) -> Any`  
    {  
        //함수 내용  
    }

# 정리 해보아요

---

- 변수 만들기 위해 필요한것?

키워드 + 변수 명(Name) + 변수 타입(Type)

- 함수 만들기 위해 필요한것?

키워드 + 함수명(Name) + 입력값(Input Value) +  
함수 내용(Action) + 결과타입(Return Type)



**Inner Peace**

# Swift 문법 - 변수

---

키워드                      변수타입

`var` `name` `:` `Type` `=` `value`

변수명                      값

# 다양한 형태의 변수 (일단 보고 가실꺼요)

---

//일반 변수 선언

```
var name:String = "joo"
```

//변수 값 재정의

```
var number:Int = 50
```

```
number = 100
```

//상수 선언

```
let PI = 3.14
```

//옵셔널 변수 선언(나중에 배울꺼예요)

```
var address:String?
```

```
address = "서울시 신사동"
```



# 키워드

---

- 변수 : 변할수 있는 값

```
var name:String = "joo"
```

- 상수 : 변할수 없는 고정 값

```
let name:String = "joo"
```

# 키워드

---

- 변수 : 변할수 있는 값

```
var name:String = "joo"  
name = "iOS개발 스쿨" ———— O
```

- 상수 : 변할수 없는 고정 값

```
let name:String = "joo"  
name = "iOS개발 스쿨" ———— X
```

# 변수명

---

- 명명규칙에 따라 작성
- 유니 코드 문자를 포함한 거의 모든 문자가 포함될 수 있다.(한글 가능)
- 변수안에 들어있는 데이터를 표현해 주는 이름으로 작성
- 중복작성 불가 ( 한 클래스, 함수, 구문 안에서)

# 명명규칙

---

- 시스템 예약어는 사용할 수 없다.
- 숫자는 이름으로 시작될 수는 없지만 이름에 포함될 수 있다.
- 공백을 포함 할 수 없다.
- 변수 & 함수명을 lowerCamelCase,  
클래스 명은 UpperCamelCase로 작성한다.

# 변수 타입

---

## 기본형

타입이름	타입	설명	Swift 문법 예제
정수	Int	1, 2, 3, 10, 100	var intName: Int
실수	Double	1.1, 2.35, 3.2	var doubleName: Double
문자열	String	"this is string"	var stringName: String
불리언	Bool	true or false	var boolName: Bool

## 참조형

타입이름	타입	설명	Swift 문법 예제
Custom Type	ClassName	클래스 객체를 다른곳에서 사용할 경우	let customView: UIView
			let timer: Timer

# Int & Uint

---

- 정수형 타입 (Integer)
- Int : +/- 부호를 포함한 정수이다.
- Uint : - 부호를 포함하지 않은(0은 포함) 정수
- 최대값과 최소값은 max, min프로퍼티를 통해 알아볼수 있다.
- Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64의 타입으로 나뉘져 있는데 시스템 아키텍처에 따라서 달라진다.
- 접두어에 따라 진수를 표현할수 있다. (2진법 0b, 8진법0o, 16진법 0x)

# Bool

---

- 불리언 타입 (true, false)



# Float & Double

---

- 부동 소수점을 사용하는 실수형 타입
- 64비트의 부동소수점은 Double, 32비트 부동 소수점은 Float으로 표현한다.
- Double은 15자리,Float은 6자리의 숫자를 표현가능
- 상황에 맞는 타입을 사용하는것이 좋으나 불확실할때는 Double을 사용하는 것을 권장.

# Character

---

- 단어나 문장이 아닌 문자 하나!
- 스위프트는 유니코드 문자를 사용함으로, 영어는 물론, 유니코드 지원 언어, 특수기호등을 모두 사용 할 수 있다.
- 문자를 표현하기 위해서는 앞뒤에 쌍 따옴표(“ ”)를 붙여야 한다.

# String

---

- 문자의 나열, 문자열이라고 한다.
- Character와 마찬가지로 유니코드로 이뤄져 있다.
- 문자열을 다루기 위한 다양한 기능이 제공된다.  
(hasPrefix, uppercased, isEmpty등)

# String 조합

---

1. string 병합: + 기호를 사용

```
var name:String  
name = "주" + "영민"
```

2. interpolation(삽입): \ (참조값)

```
var name:String = "주영민"  
print("my name is \ (name) ")
```

\ ()가 interpolation

# 튜플

---

- 정해지지 않은 데이터 타입의 묶음
- 소괄호 ( ) 안에 타입을 묶음으로 새로운 튜플타입을 만들수 있다. ex) (Int, Int) // (String, Int, String)
- 각 타입마다 이름을 지정해 줄수도 있다.  
ex) (name:String, age:Int)

# 튜플 예시

---

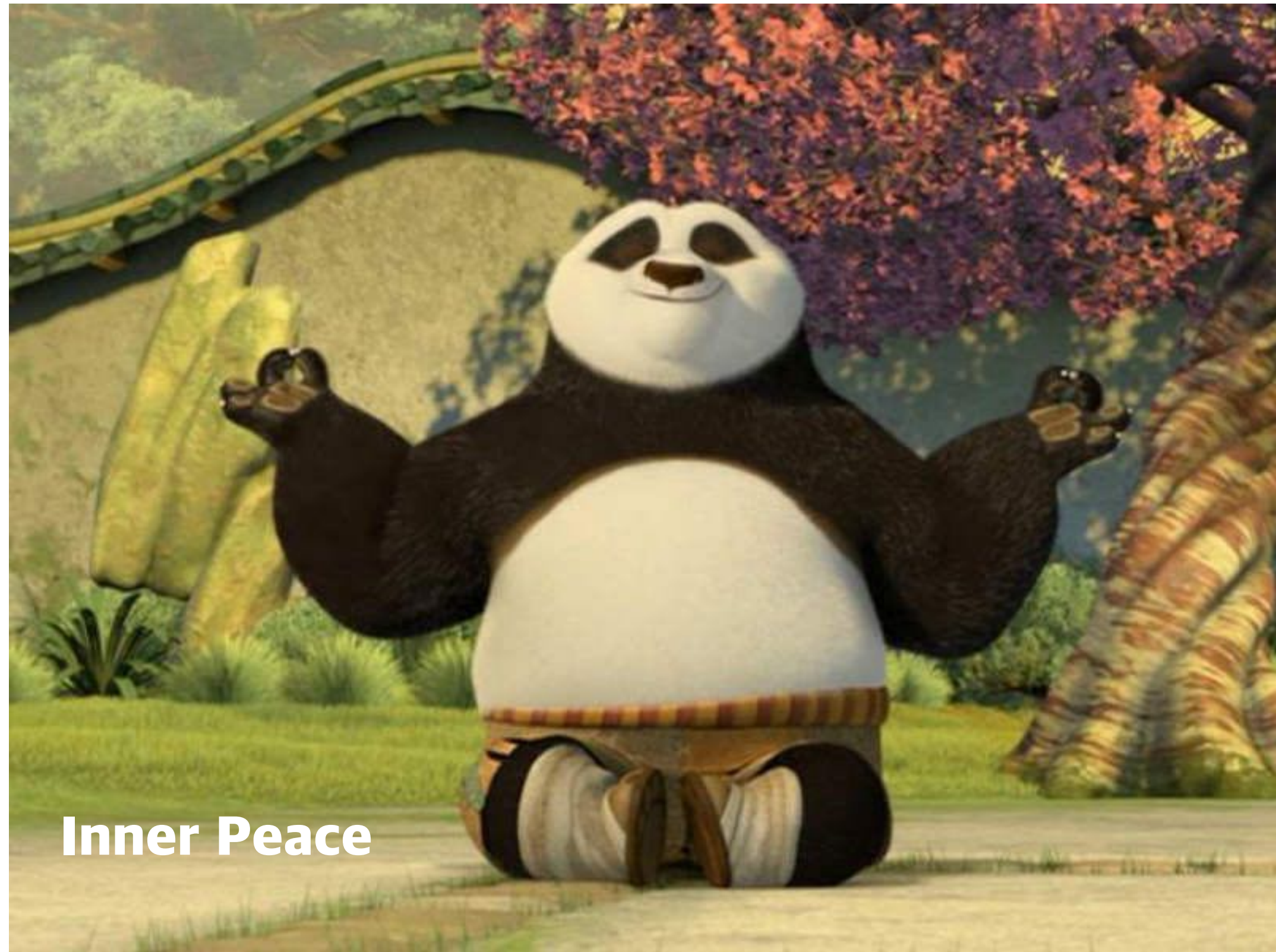
```
var coin:(Int,Int,Int,Int) = (3,1,5,3)
print("10원짜리 : \" + coin.0 + "\")
print("50원짜리 : \" + coin.1 + "\")
print("100원짜리 : \" + coin.2 + "\")
print("500원짜리 : \" + coin.3 + "\")
```

```
var person:(name:String, age:Int, weight:Double)
           = ("joo", 30, 180.2)
print("이름 : \" + person.name)
print("나이 : \" + person.age)
print("몸무게 : \" + person.age)
```

# Any, AnyObject, nil

---

- Any : 스위프트 내의 모든 타입을 나타냄
- AnyObject : 스위프트 내의 모든 객체 타입을 나타낸다.(클래스)
- nil : 데이터가 없음 을 나타내는 키워드



**Inner Peace**



# 캐스팅(형변환)

---

```
var total:Int = 107
```

```
var average:Double
```

```
average = total/5
```

← type Error

# 캐스팅을 해야하는 이유

---

실수 : 107.0

1	1	0	0	1	0	0	0	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

정수 : 107

1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# 캐스팅(형변환)

---

```
var total:Int = 107
```

```
var average:Double
```

```
average = total/5 ← type Error
```

```
average = Double(total)/5 ← casting
```

# 캐스팅(형변환)

---

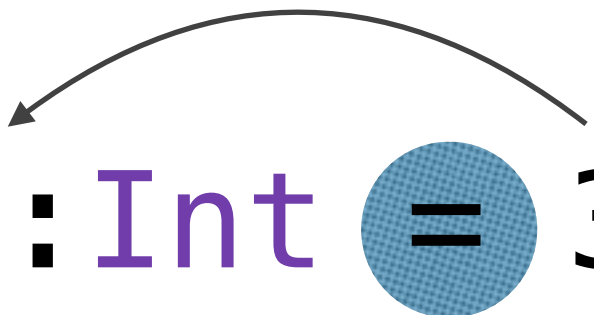
```
var stringNum:String  
var doubleNum:Double  
let intNum:Int = 3
```

```
stringNum = String(intNum) ← int to string  
doubleNum = Double(intNum) ← int to double
```


# 변수 값 지정

---

`var number: Int = 3`



`var age = 31` (타입 추론)



대입연산자	예제	설명
=	number = 4	number변수에 숫자 4를 넣는다.

# 다양한 형태의 변수

---

//일반 변수 선언

```
var name:String = "joo"
```

//변수 값 재정의

```
var number:Int = 50
```

```
number = 100
```

//상수 선언

```
let PI = 3.14
```

//옵셔널 변수 선언(나중에 배울꺼예요)

```
var address:String?
```

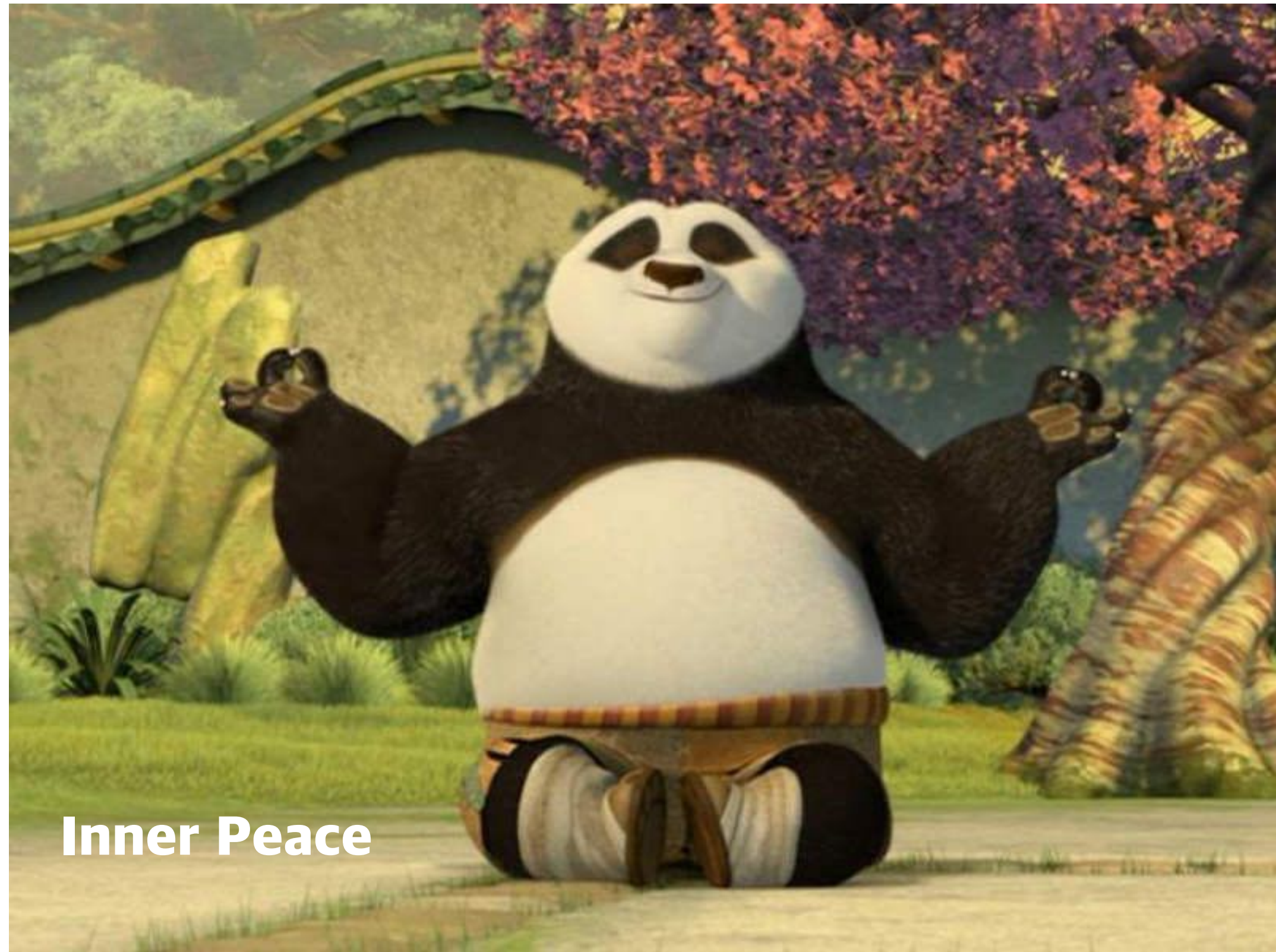
```
address = "서울시 신사동"
```

# 놀이터에서 문법 익히기

---

다양한 변수를 만들어 봅시다.

이름, 나이, 성별, 학교, 직업, 연봉 등  
다른 타입으로 30개의 변수(상수) 작성하기.



**Inner Peace**



# Swift 문법 - 함수

---

```
func fName(agumentName paramName:Int) -> Int
{
    return paramName + 3
}
```

# Swift 문법 - 함수

---

키워드                      인수명                      매개변수명                      반환타입

함수 이름                      매개변수타입

```
func fName(argumentName paramName: Int) -> Int  
{  
    return paramName + 3  
}
```

함수 내용

The diagram illustrates the Swift function syntax with the following components and annotations:

- func**: Keyword (키워드), circled in red.
- fName**: Function name (함수 이름), circled in red.
- argumentName**: Parameter name (인수명), circled in red.
- paramName**: Parameter name (매개변수명), circled in red.
- Int**: Parameter type (매개변수타입), circled in red.
- >**: Return type arrow, circled in red.
- Int**: Return type (반환타입), circled in red.
- { ... }**: Function body (함수 내용), enclosed in a blue box.
- return paramName + 3**: Return statement inside the function body.

# Argument Labels and Parameter Names

---

인수레이블 명

매개변수명

매개변수타입

```
func fName(agumentName paramName: Int) -> Int  
{
```

```
    return paramName + 3
```

```
}fName(agumentName: 10) ← 함수호출
```

- 인수레이블은 함수 호출시 사용 되는 이름표.
- 매개변수는 함수 내부에서 사용 되는 변수명
- 인수레이블은 생략가능하며 없을때는 매개변수명이 인수레이블로 사용된다.

# Default Parameter Values

---

```
func number(num1:Int, num2:Int = 10) -> Int {  
    return num1 + num2  
}
```

```
number(num1: 10)           ← 20  
number(num1: 10, num2: 5) ← 15
```

- 매개변수에는 기본값을 설정할수 있다.
- 기본값은 인자로 값이 들어오지 않을때 사용된다.

# In-Out Parameter Keyword

---

inout Keyword

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

- 매개변수는 기본 상수값이다.
- 만약 매개변수의 값을 변경해야 한다면 inout 키워드를 사용하여 inout 변수로 지정해야만 한다.
- inout 변수 지정은 타입 앞에 inout keyword를 작성해준다.
- inout 변수가 지정된 함수의 인수앞에서 & 가 붙어야 한다.

# In-Out Parameter Keyword

---

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
var someInt = 3  
var anotherInt = 107  
swapTwoInts(&someInt, &anotherInt)
```



---

```
swapTwoInts(3, 107)  
swapTwoInts(&3, &107)
```



# 여러가지 함수 - 매개변수

---

```
func getNumber(firstNum num1:Int) -> Int {  
    return num1  
}
```

```
func getNumber(num1:Int) -> Int {  
    return num1  
}
```

```
func getNumber() -> Int {  
    var num1:Int = 22  
    return num1  
}
```

```
func getNumber(firstNum num1:Int, secondNum num2:Int) -> Int {  
    return num1 + num2  
}
```

```
func sumNumber(num1:Int, num2:Int = 5) -> Int {  
    return num1 + num2  
}
```





**Inner Peace**



# 반환타입

---

반환타입

```
func fName(agumentName paramName: Int) -> Int
{
    return paramName + 3
}
```

- 함수 실행 결과의 타입을 명시 해준다. (Return Type)
- **return** 키워드를 사용하여 함수 결과 반환.  
반환 타입과 같은 타입의 데이터를 반환 해야 한다.
- 한개의 값만 반환 할수 있다.
- 반환값이 없는 경우는 Return Type을 작성하지 않고( -> 제거)  
**return** 키워드를 사용할 필요가 없다.(반환값이 없기때문)

# 반환타입 예제

---

```
func printName() -> String{  
    return "my name is youngmin"  
}
```

```
func printName(){  
    print("my name is youngmin")  
}
```

```
func printName(name:String = "youngmin"){  
    print("my name is \(name)")  
}
```

```
func printName(explain str:String, name str2:String) -> String{  
    return str + str2  
}
```

```
func printName(explain str: inout String) -> String{  
    str += "joo"  
    return str  
}
```

## Step 2. Class



# 함수 꾸미기

---

- 단순 명령어의 순서가 아닌 복잡한 명령을 내리기 위해선 프로그램 명령어 컨트롤 방법(Control Flow)이 필요하다.
- 조건문(while, for-in) & 선택문(if, guard, switch)을 통해 함수를 컨트롤 할수 있다.

---

# 조건문

---

강사 주영민

# 조건문 개론

---

- 함수 내부에서 실행되는 선택문
- 특정 조건에 따라 선택적으로 코드를 실행시킨다.
- 대표적인 조건문으로 if-else문과 switch-case문이 있다.

# if-else문

---

조건이 참일경우 if문 대괄호 안의 코드가 실행된다.

만약 조건이 거짓인 경우 else문 대괄호 안의 코드가 실행된다.

```
if 조건 {  
    //조건이 만족되면 실행  
}  
else {  
    //조건이 만족되지 않을때 실행  
}
```

**\*조건값은 참,거짓의 나타나는 Bool값으로 표현.**

# else if문

---

추가 조건 방법으로 반복해서 추가 할수 있다.

```
if 조건1 {  
    //조건1이 만족되면 실행  
}else if 조건2{  
    //조건1이 만족되지 않을때 실행  
}else{  
    //조건들 모두 만족되지 않을때 실행  
}
```

**\*조건2는 조건1이 거짓일때 실행된다.**



# 조건 만들기

---

- 비교연산자를 통해 조건의 결과가 bool값으로 나와야 한다.
- 논리 연산자로 다양한 조건의 조합이 가능하다.

---

# 연산자

---

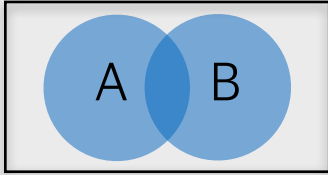
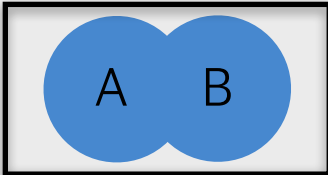
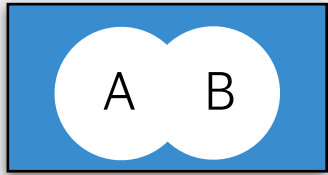
# 산술 연산자

기호	예제	설명
+	$1 + 2 = 3$	더하기
-	$2 - 1 = 1$	빼기
*	$2 * 1 = 2$	곱하기
/	$10 / 3 = 3$	나누기
%	$10 \% 3 = 1$	나머지

# 비교 연산자

기호	예제	설명
<code>==</code>	<code>A == B</code>	A와 B가 <b>같다</b> .
<code>&gt;=</code>	<code>A &gt;= B</code>	A가 B보다 <b>크거나 같다</b>
<code>&lt;=</code>	<code>A &lt;= B</code>	A가 B보다 <b>작거나 같다</b> .
<code>&gt;</code>	<code>A &gt; B</code>	A가 B보다 <b>크다</b>
<code>&lt;</code>	<code>A &lt; B</code>	A가 B보다 <b>작다</b>

# 논리 연산자

기호	예제	집합	설명
&&	A조건 && B조건		A조건이 참이고, B조건이 참이면 참이다.
	A조건    B조건		A조건이나, B조건 둘중에 하나가 참이면 참이다.
!	!(A조건    B조건)		A    B조건의 반대

# 추가연산자

복합연산자	예제	설명
<code>+=</code>	<code>a += 1</code>	a에 값을 더하기
<code>-=</code>	<code>b -= 2</code>	b에 값을 빼기

범위 연산자	예제	설명
<code>a...b</code>	<code>3...10</code>	a~b까지의 숫자
<code>a..&lt;b</code>	<code>0..&lt;10</code>	a~b까지 숫자중 b는 포함 안함

Identity 연산자	예제	설명
<code>===</code>	<code>person2 === person1</code>	person1과 person2는 같은 인스턴스를 참조하고 있다.
<code>!==</code>	<code>person2 !== person1</code>	person1과 person2는 다른 인스턴스를 참조하고 있다.

# 삼항연산자

삼항연산자	설명
question ? answer1 : answer2.	question이 참이면 answer1값을 거짓이면 answer2값을 지정한다.

```
let age = 20  
var result:String = age > 19 ? "성년" : "미성년"
```

# 조건의 예

---

```
func printGeneration(age:Int)
{
    if (age >= 30) {
        print("30대 이상")
    } else if (age >= 20)
    {
        print("20대")
    } else
    {
        print("미성년자")
    }
}
```

다음 실행 결과는?

1. printGeneration(10);  
    >> 미성년자
2. printGeneration(40);  
    >> 30대 이상
3. printGeneration(20);  
    >> 20대



# 조건문 예시

---

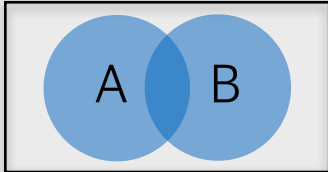
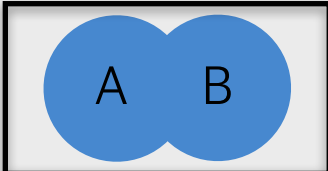
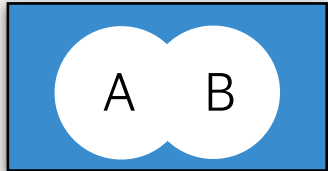
```
func check(name compareName:String )
{
    if compareName == "주영민"
    {
        print("주영민 입니다.")
    }else
    {
        print("주영민이 아닙니다")
    }
}
```

# 조건문 예시

---

```
func isEven(number num:Int) -> Bool
{
    if(num%2 == 0)
    {
        //짝수
        return true
    }else
    {
        //홀수
        return false
    }
}
```

# 논리연산자

기호	예제	집합	설명
&&	A조건 && B조건		(A조건이 참이고, B조건이 참이면) 참이다.
	A조건    B조건		(A조건이나, B조건 둘중에 하나가 참이면) 참이다.
!	!(A조건    B조건)		조건을 반대(참 -> 거짓, 거짓 -> 참)

# 조건의 조합

---

1. or 조합( || ) : 조건들 중 한개만 참이어도 전체 조건이 참이다.
2. and 조합( && ) : 조건들 모두 참이어만 전체 조건이 참이다.
3. not ( ! ) : 조건 결과의 반대값

# 조건의 조합

---

<시험 점수를 구하는 함수>

시험점수 95점 이상 100점 이하는 A+,  
90점 이상 95점 미만은 A,  
85점 이상 90점 미만은 B+  
80점 이상 85점 미만은 B...

\* 다음과 조건들은 어떻게 처리해야 될까요?

# 조건의 조합

---

시험점수 95점 이상 100점 이하는 A+

```
if( 100 >= score && 95 <= score)
```

90점 이상 95점 미만은 A

조건을 적어보세요

85점 이상 90점 미만은 B+

조건을 적어보세요

80점 이상 85점 미만은 B...

조건을 적어보세요

# 문제

---

1. 시험 점수를 받아서 해당 점수의 등급(**Grade**)을 반환해주는 함수

\***Grade** = A+, A, B+, B, C+...

2. **Grade** 받아서 **Point**로 변환해주는 함수

**Point** = 4.5, 4.0, 3.5, 3.0...

# Switch문

---

- 패턴 비교문
- 가장 첫번째 매칭되는 패턴의 구문이 실행된다.



# switch 문법

---

```
switch some value to consider {  
  case value 1:  
    respond to value 1  
  case value 2,  
    value 3:  
    respond to value 2 or 3  
  default:  
    otherwise, do something else  
}
```

# switch 문법

---

- 각의 상태는 case 키워드를 통해 나타낼수 있다.
- 각 case 상태 끝에는 콜론(:)을 작성해서 매칭 패턴을 종료한다.
- 여러개의 case가 필요하면 콤마(,)를 통해 상태를 추가 할수 있다.
- 각 case는 분기로 실행되며 매칭된 해당 case문이 종료되면 switch 문을 종료시킨다.
- 각 case의 상태는 switch 문의 value값에 매칭된 지점을 결정하며, 모든 value에 대해 매칭 되어야 한다. 만약 매칭되는 값이 없다면 default 키워드를 통해 기본 매칭값을 설정하며, default키워드는 마지막에 배치된다.

# switch 예제

---

```
func sampleSwitch(someCharacter:Character)
{
    switch someCharacter {
    case "a":
        print("The first letter of the alphabet")
    case "z":
        print("The last letter of the alphabet")
    default:
        print("Some other character")
    }
}
```

# 문제

---

1. 시험 점수를 받아서 해당 점수의 등급(**Grade**)을 반환해주는 함수

\***Grade** = A+, A, B+, B, C+...

2. **Grade** 받아서 **Point**로 변환해주는 함수

**Point** = 4.5, 4.0, 3.5, 3.0...

# Interval Matching

---

- Switch 문의 상태는 단순 value매칭을 넘어 좀더 다양한 패턴을 통해 매칭이 가능하다.
- interval matching은 범위 연산자를 통해 해당 범위에 해당하는 value를 매칭 시킬수 있다.

# Interval Matching 예제

---

```
func interSwitch(count:Int)
{
    let countedThings = "moons orbiting Saturn"
    let naturalCount: String
    switch count {
    case 0:
        naturalCount = "no"
    case 1..<5:
        naturalCount = "a few"
    case 5..<12:
        naturalCount = "several"
    case 12..<100:
        naturalCount = "dozens of"
    case 100..<1000:
        naturalCount = "hundreds of"
    default:
        naturalCount = "many"
    }
    print("There are \(naturalCount) \(countedThings).")
}
```

# 튜플매칭

---

- 튜플을 사용해서 여러개의 value를 동시에 확인 할수 있습니다.
- 사용 가능한 모든 값에 대한 매칭은 와일드 카드 ( \_ )를 통해서 매칭 가능합니다.

# 튜플 예제

---

```
func getPoint(somePoint:(Int,Int))
{
    switch somePoint {
    case (0, 0):
        print("\(somePoint) is at the origin")
    case (_, 0):
        print("\(somePoint) is on the x-axis")
    case (0, _):
        print("\(somePoint) is on the y-axis")
    case (-2...2, -2...2):
        print("\(somePoint) is inside the box")
    default:
        print("\(somePoint) is outside of the box")
    }
}
```



# 값 바인딩

---

- case 내부에서 사용되는 임시 값으로 매칭 시킬수 있다.

# 값 바인딩 예제

---

```
func getPoint(somePoint:(Int,Int))
{
    switch somePoint {
    case (0, 0):
        print("\(somePoint) is at the origin")
    case (let x, 0):
        print("on the x-axis with an x value of \(x)")
    case (0, let y):
        print("on the y-axis with an y value of \(y)")
    case (-2...2, -2...2):
        print("\(somePoint) is inside the box")
    default:
        print("\(somePoint) is outside of the box")
    }
}
```

# where문

---

- where 문의 추가로 추가 조건을 넣을수 있다.

# where문 예제

---

```
func wherePoint(point:(Int,Int))
{
    switch point {
    case let (x, y) where x == y:
        print("\(x), \(y)) is on the line x == y")
    case let (x, y) where x == -y:
        print("\(x), \(y)) is on the line x == -y")
    case let (x, y):
        print("\(x), \(y)) is just some arbitrary point")
    }
}
```

## Step 3. 버튼 액션



## Step 4. 계산기 완성

