

# Swift Study 13



2016. 02.25

# Swift 란?



- wwdc(The Apple Worldwide Developers Conference) 2014에 발표한 프로그래밍 언어로 애플의 osx / ios 등을 개발목적으로 만든 언어
- 기본의 object-c보다 성능적으로나 문법적으로나 간편 유연하며 학습하기 편하다는 의견

# Swift 란?



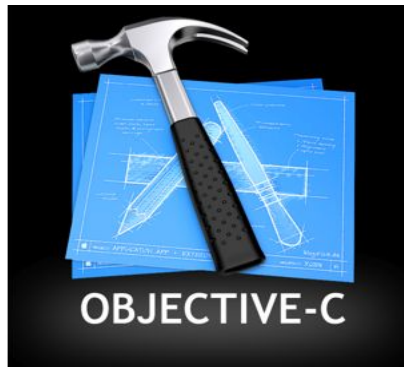
- **wwdc 2015에서 swift를 오픈소스(GPL v3)로 전환하여 osx와 linux 환경에서 사용가능하게 되었다.**
- 더불어 기존의 **2.2에서 3.0으로 버전업하면서 일부 문법 및 함수 네이밍, 넘버링 등의 전체적인** 틀 일부가 변경되었다.
- 현재 **object-c와 혼용으로 사용가능하며 아직 osx, ios의 많은 라이브러리가 object-c기반이라 점점 swift 코드로 대체 중이다. (cocoa framwork / cocoa touch framework 기반 언어가 object-c)**

# Swift 란?



- **object-c**와 **swift**의 컴파일러(백그라운드 컴파일러)인 **LLVM**의 메인개발자 크리스 래트너(**Chris Lattner**)에 의해 최초 개발되었다. 현재는 애플을 떠난 상태
- 현재 오픈소스 프로젝트로 **github**(<https://github.com/apple/swift>)에 애플이 공개한 상태이다.
- **2017.02** 기준으로 **3.0**버전이 **release**상태이며, **3.0**버전부터는 문법적인 큰 수정이 없다고 공식 발표하였다.

# Cocoa Framework 란?



- 스티브잡스가 애플을 떠나 있을 당시 Next사의 메인언어인 **Object-C**를 기반으로 한 **UI** 및 전체적인 기능 프레임워크.
- 라이브러리 네이밍의 **NS**가 붙은 것은 당시 넥스트사의 운영체제인 넥스트스텝(**NeXTSTEP**)에 따온 약어
- **osx** 나 **ios** 개발시 **cocoa framework**의 라이브러리를 사용하여 개발하게 됨
- **osx**는 **cocoa framework**, **ios/tablet**은 **cocoa touch framework** 사용  
(대표적인 **Foundation Kit**, **UI Kit** 등이 있다.)

# Swift 3.0 & Xcode 8.0

```
1. bash
vnenise:~ vnenise$ swift -version
Apple Swift version 3.0.2 (swiftlang-800.0.63 clang-800.0.42.1)
Target: x86_64-apple-macosx10.9
vnenise:~ vnenise$
```



# Xcode



## Welcome to Xcode

Version 8.0 (8A218a)



### Get started with a playground

Explore new ideas quickly and easily.



### Create a new Xcode project

Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.



### Check out an existing project

Start working on something from an SCM repository.



### ImageView

~/Documents/ios/SwiftStudy/week01



### Week01

~/Documents/ios/SwiftStudy



### ImageView

~/Documents/ios/doitEasypub



### MyPlayground

~/Documents/ios/playground



### Audio

~/Documents/ios/doitEasypub



### DatePicker

~/Documents/ios/doitEasypub



### PickerView

~/Documents/ios/doitEasypub



### Test1

~/Documents/ios/test

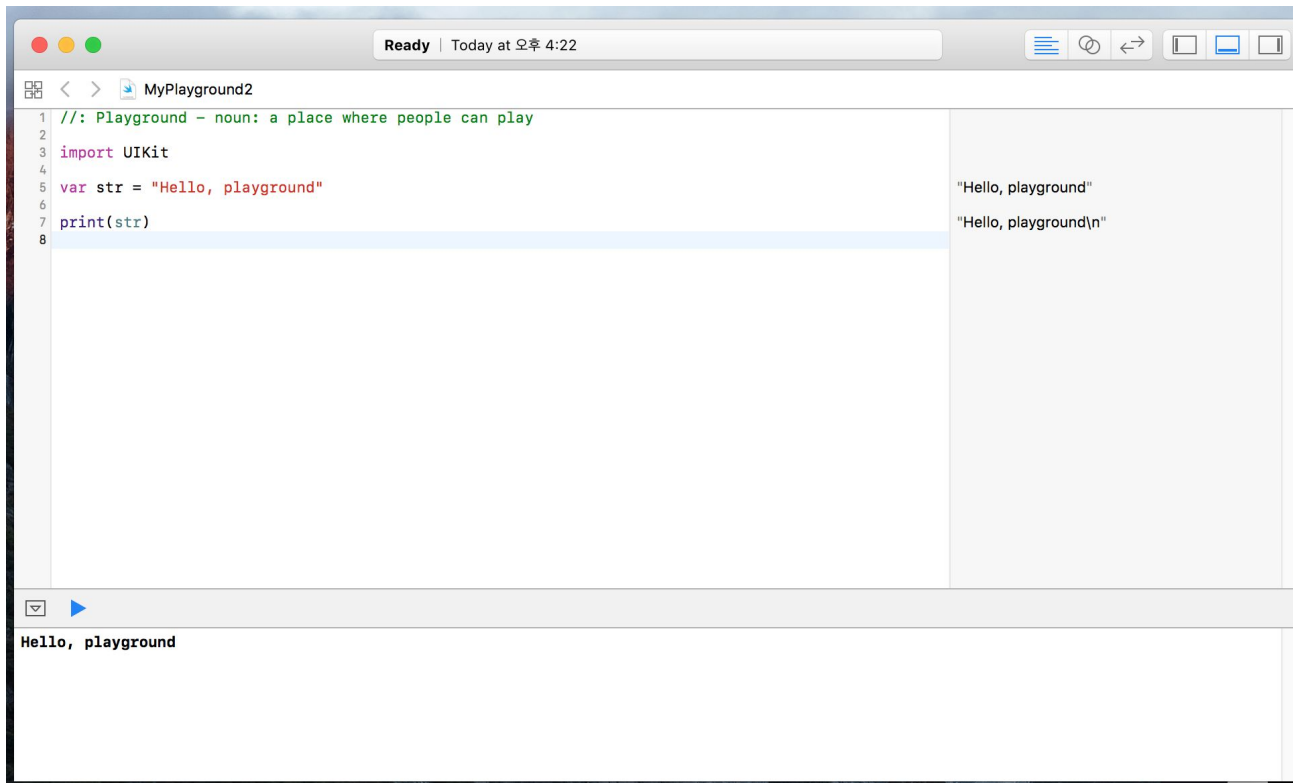


### MeetingRooms

~/Documents/ios/helloworld

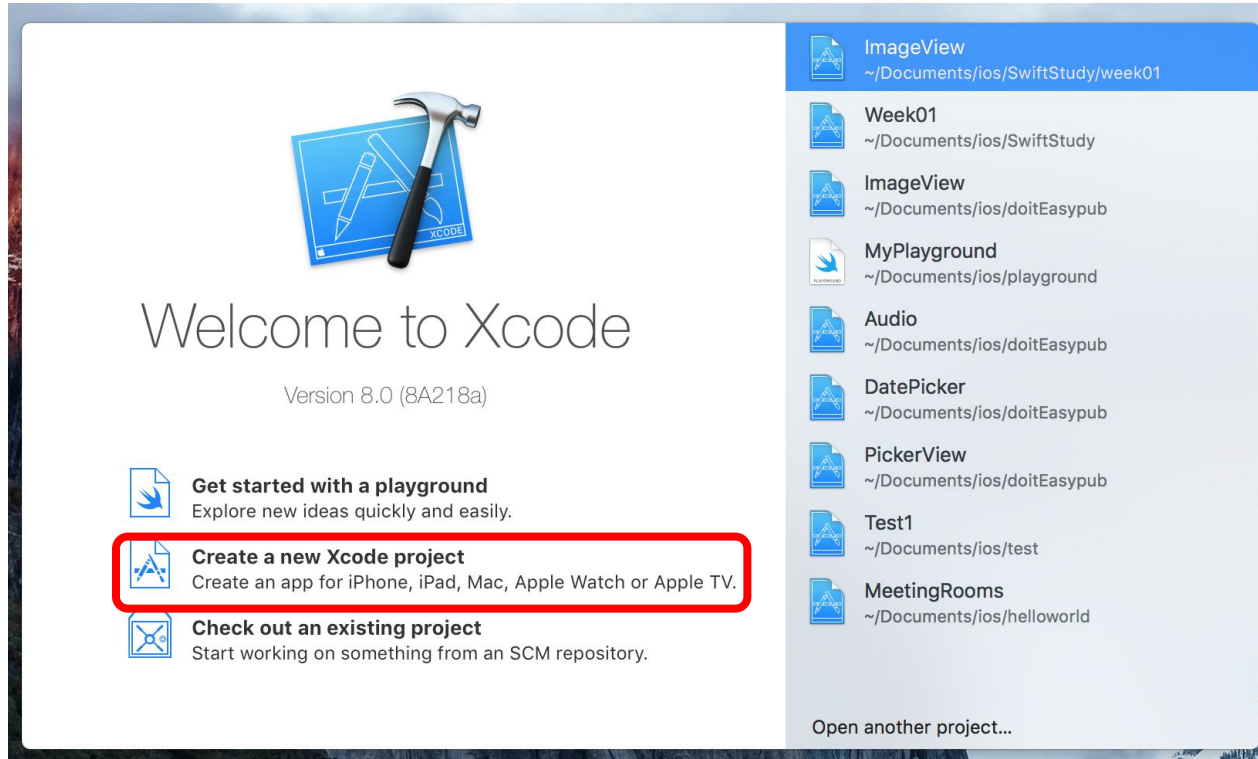
Open another project...

# Xcode

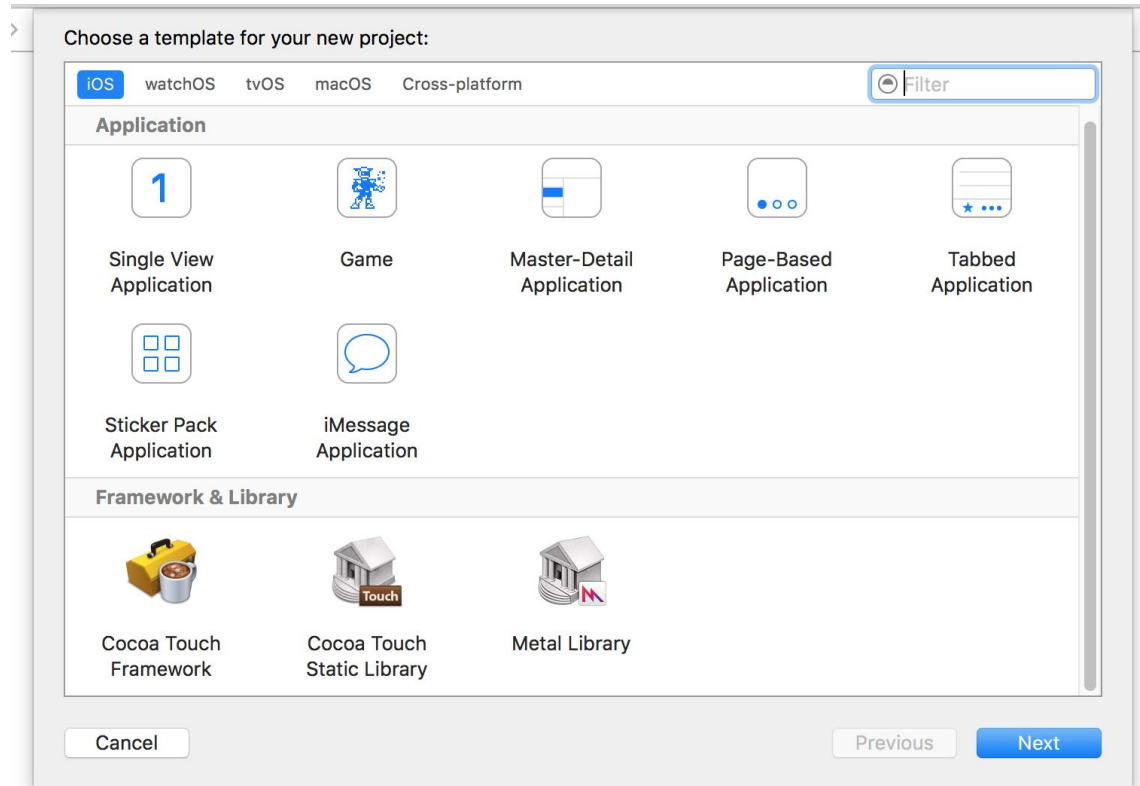




# Xcode



# Xcode



# Xcode

Choose options for your new project:

Product Name: HelloWorld

Team: han-jin Ryu (Personal Team)

Organization Name: vnenise

Organization Identifier: com.vnenise

Bundle Identifier: com.vnenise.HelloWorld

Language: Swift

Devices: iPhone

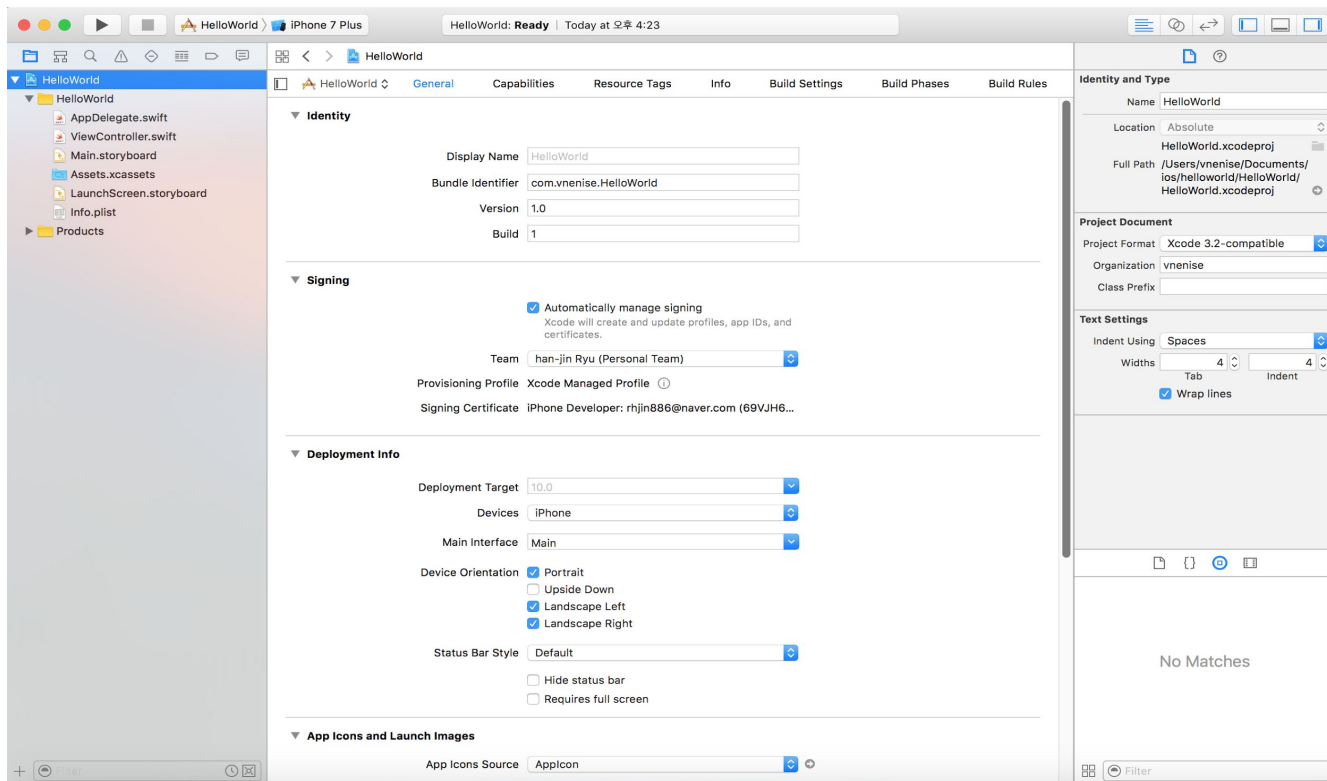
☐ Use Core Data

☐ Include Unit Tests

☐ Include UI Tests

Cancel Previous Next

# Xcode



# Xcode



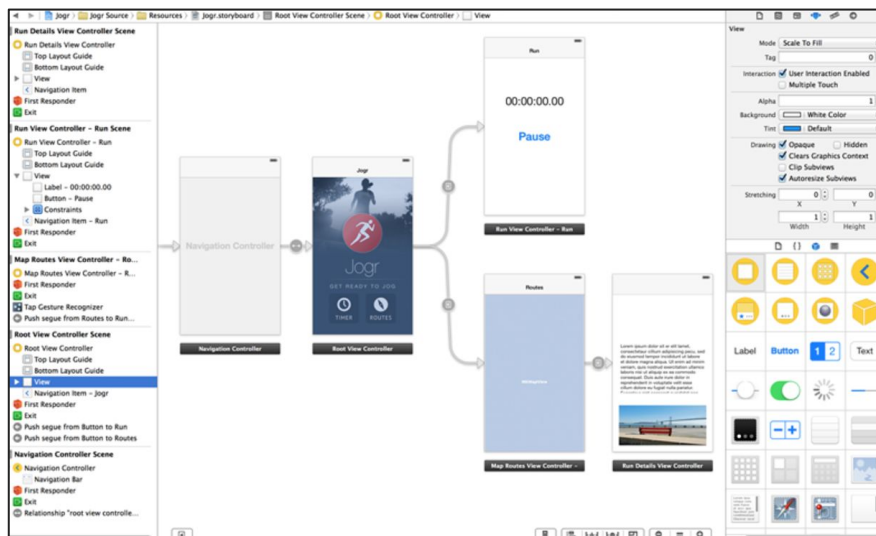
인스펙터  
영역

라이브러리  
영역

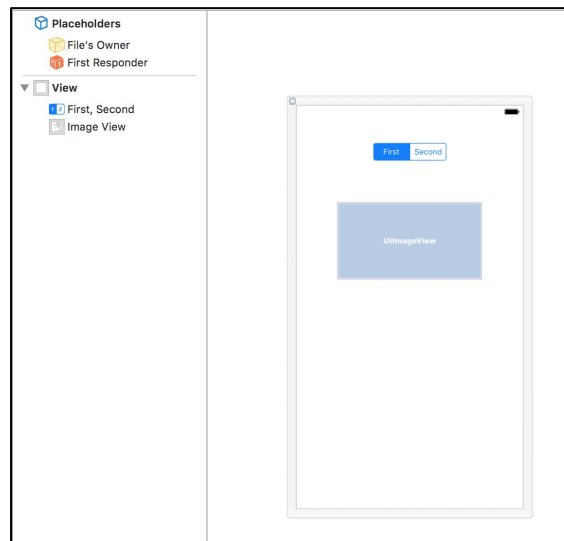
디버그 영역

# Xcode - storyboard

- UI기반의 화면레이아웃 (xcode4, ios5부터 지원)
- 개별적인 뷰 레이아웃을 담당하는 xib파일 방식도 있음.
- 이런 UI기반의 화면빌더를 Interface Builder라고 한다면 뭐라나...



storyboard



xib

# Xcode - Swift

- **import** - 외부라이브러리, 프레임워크 참조용도, 그 외의 프로젝트 내에 존재하는 파일의 객체를 참조할 수 있다.

(클래스 외부 선언시 어느파일에서든 접근가능, 전역변수, **current module**)

- 문자열, 문자 - 쌍따옴표(“”) => 문자선언시 **Character** 명시

- **main** 존재 안함 - **@UIApplicationMain**

- 주석: **//**, **/\* \*/**

- 한글, 영어, 한자, 알라비아숫자 조합 변수, 자음, 바이너리 이모티콘 (연산자조합, 공백, 상수명, 첫숫자는 사용불가)

# Swift 문법

- 기본 자료형 (데이터 타입), 연산자
- 변수 및 선언방식, 문자열 포맷
- 조건문
- 반복문



# Swift - 기본 자료형

타입	특징	예
<b>Int</b> <b>Int8</b> (127 ~ -128) <b>Int16</b> (32,767 ~ -32,768) <b>Int32</b> (2,147,483,647 ~ 2,147,483,648) <b>Int64</b> (9,223,372,036,854,775,807 ~ -9,223,372,036,854,775,808)	작은 수 또는 큰 수의 음수, 양수값	<b>4, 523, -45565, 5342, -28, 54, 234</b>
<b>UInt</b> <b>UInt8</b> (0 ~ 255) <b>UInt16</b> (0 ~ 65,535) <b>UInt32</b> (0 ~ 4,294,967,295) <b>UInt64</b> (0 ~ 18,446,744,073,709,551,615)	작은 수 또는 큰 수의 양수값	<b>5, 123, 3432432, 52, 34, 5, 123</b>

- 뒷수자는 bit 자릿수 의미 (8bit - 1byte)
- 자료형의 숫자는 각 cpu운영체제별 자동부여 (Int형 선언시 32bit운영체제 - Int32, 64bit운영체제 - Int64)

# Swift - 기본 자료형

타입	특징	예
<b>Float (Float32)</b> - 소수점 7~8자리 <b>Double (Float64)</b> - 소수점 15~16자리	부동 소수점, 분수의 음수, 양수값 <b>Double</b> 은 <b>Float64</b> 의 <b>typealias</b>	<b>11.453, 234.23, -123.34, 2.231231123, 0.012345</b>
<b>Character</b>	단일 문자 (큰따옴표로 묶어서 표현)	<b>"T", "한", "H", "*", "3"</b>
<b>String / NSString(object-c객체)</b>	문자열 데이터 (큰따옴표로 묶어서 표현) <b>String/NSString</b> 서로 호환변환 가능 <b>NSString</b> 사용시 <b>import Foundation</b> 필요	<b>"Filasa", "문장입니다.", "New York"</b>
<b>Bool</b>	참/거짓을 표현하는 논리데이터 표현	<b>true, false</b>
<b>nil</b>	값이 없다는 표현 (자료형이라고 하기 애매하네요)	<b>nil</b>

- 이 밖에도 **Collection** 타입: **Array, Tuple, Dictionary** / 구조체 **Struct** / 열거형 **enum** / **Class** 등 참조타입이 있음.

# 튜플 - Tuple

- 여러 자료형을 묶는 집단 자료형
- () 안에 타입 또는 값을 정의 / 쉼표(,)로 분류
- 점(.)으로 내부적인 인덱스(index) 및 라벨(label) 참조접근

```
//형태  
(1, "test", 2.3, true) //(Int, String, Double, Boolean)
```

```
//초기화  
let test:(Int,String) = (123, "test")
```

```
//선언 및 대입  
var test:(Int, String)  
test.0 = 123  
test.1 = "test"
```

```
//타입 생략  
let test = (123, "test")
```

```
//인덱스  
let test:(Int,String) = (123, "test")  
test.0 => 123  
test.1 => "test"
```

```
//레이블  
let test = (num:123, str:"test")  
test.num => 123  
test.str => "test"
```

```
//변수 할당  
let (num, str) = (123,"test")  
num => 123  
test => "test"
```

# Any / AnyObject

- 모든 데이터 자료형을 포함하는 Any
- Class타입에 한정된 AnyObject
- 모든 자료형은 Any / Class 참조형 AnyObject

```
//AnyObject
func anyType(any:AnyObject){
    //code
}
```

```
class Test{
var test = Test()
anyType(any: test)    //허용o
```

```
let num:Int = 123
anyType(any: num)    // 허용x
```

```
//Any type
func anyType(any:Any){
    //code
}
```

```
anyType(any: test)    //허용o
anyType(any: num)    //허용o
```

## 연산자 - 산술연산자

구분	연산자	예
단항연산자	-	-9
이항연산자	+, *, /, %	1+2 1*2 2/1 6%4

# 연산자 - 비교연산자

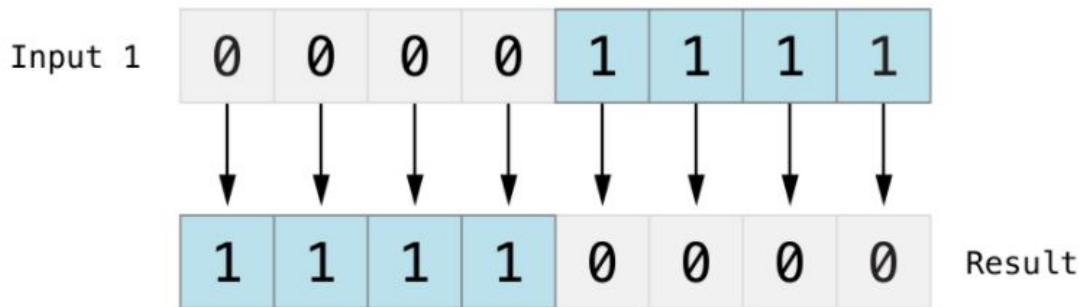
연산자	예	의미
<	$a < b$	a가 b보다 작으면 true, 그 반대는 false
>	$a > b$	a가 b보다 크면 true, 그 반대는 false
>=	$a \geq b$	a가 b보다 크거나 같으면 true, 그반대는 false
=<	$a \leq b$	a가 b보다 작거나 같으면 true, 그반대는 false
==	$a == b$	a가 b와 같으면 true, 같지 않으면 false
!=	$a != b$	a가 b와 같지 않으면 true, 같으면 false

# 연산자 - 논리연산자

연산자	예	의미
! (NOT)	!a	a가 true면 false, a가 false면 true
&& (AND)	a && b	a와 b 모두 true면 true, 둘 중 하나가 false면 false
(OR)	a    b	a 또는 b 둘중 하나라도 만족하면 true, 둘 다 false면 false

# 연산자 - 이진연산자 (binary operator)

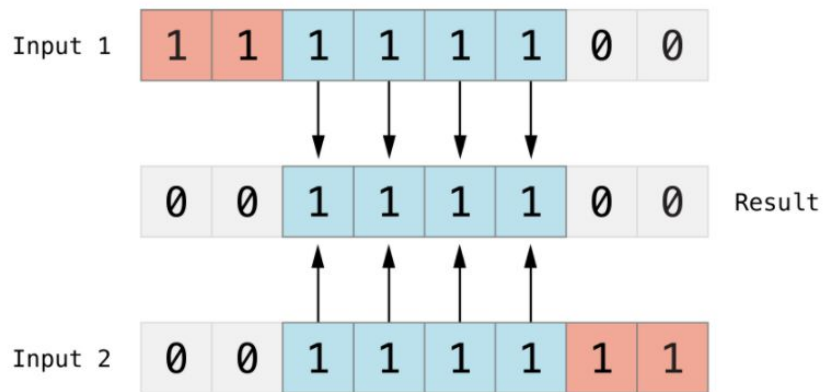
연산자	예
~ (NOT)	<pre>let initialBits: UInt8 = 0b00001111 let invertedBits = ~initialBits // equals 11110000</pre>





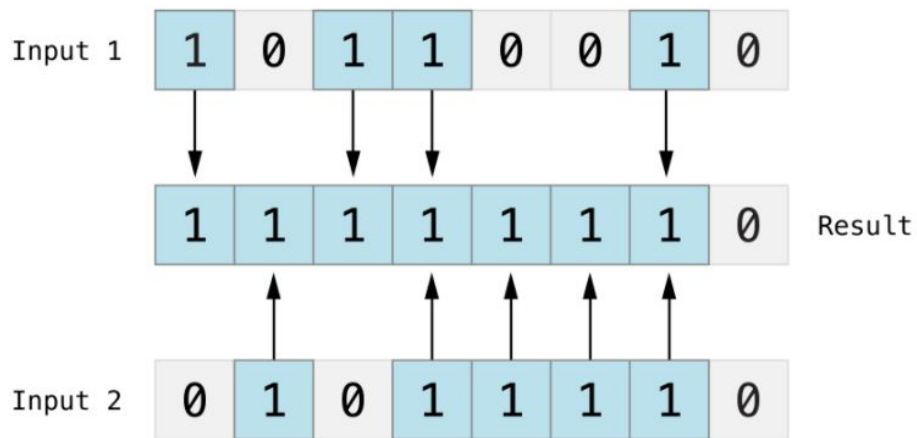
# 연산자 - 이진연산자 (binary operator)

연산자	예
& (AND)	<pre>let firstSixBits: UInt8 = 0b11111100 let lastSixBits: UInt8 = 0b00111111 let middleFourBits = firstSixBits &amp; lastSixBits // equals 00111100</pre>



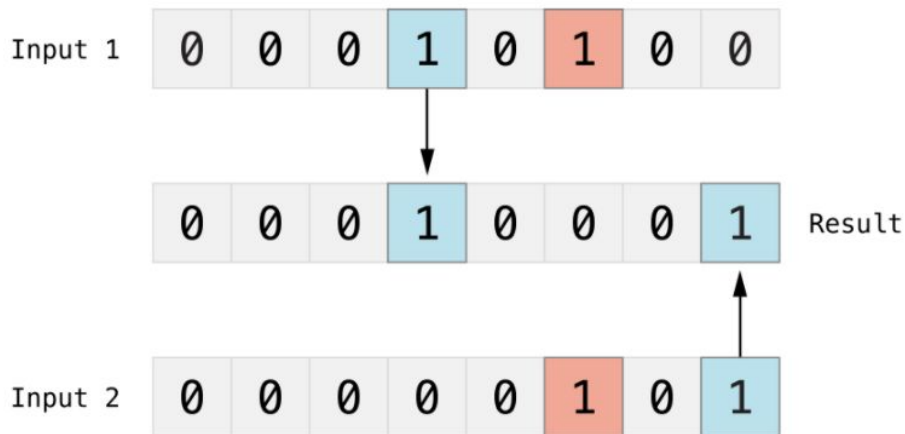
# 연산자 - 이진연산자 (binary operator)

연산자	예
(OR)	<pre>1 let someBits: UInt8 = 0b10110010 2 let moreBits: UInt8 = 0b01011110 3 let combinedbits = someBits   moreBits // equals 11111110</pre>



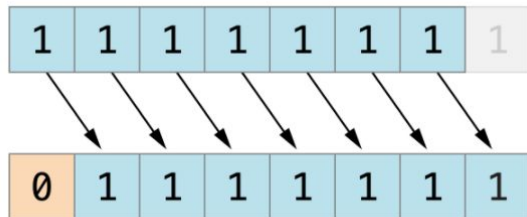
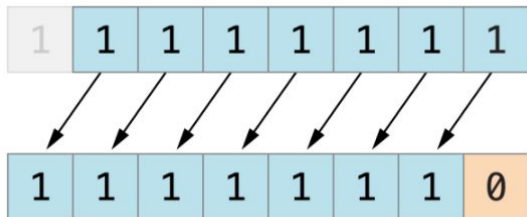
# 연산자 - 이진연산자 (binary operator)

연산자	예
$\wedge$ (XOR)	<pre>1 let firstBits: UInt8 = 0b00010100 2 let otherBits: UInt8 = 0b00000101 3 let outputBits = firstBits ^ otherBits // equals 00010001</pre>



# 연산자 - shift 연산자 (비트 이동연산자)

연산자	예
<<  >>	<pre>1 let shiftBits: UInt8 = 4 // 00000100 in binary 2 shiftBits &lt;&lt; 1           // 00001000 3 shiftBits &lt;&lt; 2           // 00010000 4 shiftBits &lt;&lt; 5           // 10000000 5 shiftBits &lt;&lt; 6           // 00000000 6 shiftBits &gt;&gt; 2           // 00000001</pre>



# 연산자 - 범위연산자

연산자	예	의미
$\dots$ (닫힌 범위 연산자)	1...5	1에서 5까지 (1,2,3,4,5)
$..<$ (반닫힌 범위연산자)	1.. $<5$	1에서 4까지 (1,2,3,4)

**5>..1 (x) 의 범위연산자 표기 불가 (반닫힌 연산자는 오른쪽만 가능)**

# 연산자 - 대입연산자

=	a = 1	변수 a에 1대입
+=	a += 1	a = a + 1
-=	a -= 1	a = a - 1
*=	a *= 2	a = a * 2
%=	a %= 3	a = a % 3
<<=	a <<= 3	a = a << 3
>>=	a >>= 3	a = a >> 3
&=	a &= b	a = a & b
^=	a ^= b	a = a ^ b
=	a  = b	a = a   b

\* swift 3.0부터 ++, -- (증감/가감연산자) 삭제됨 => +=1 / -=1로 사용권장

# Swift - 변수

\*변수: 프로그래밍 언어에서 일반적으로 어느 메모리주소에 데이터 값을 담고 사용하는 논리적 공간개념

**var 키워드:** 언제든지 값을 변경할 수 있다

<형식>

**var** [변수명]

**var** [변수명]:[데이터 자료형]

**var** [변수명]:[데이터 자료형]

**var** [변수명]:[데이터 자료형] = 데이터값

(예)

var a

var a:Int

var a = 123

a = 567 (a의 값이 567로 변경됨)

**let 키워드:** 값을 넣으면 변경이 불가능한 값

<형식>

**let** [변수명]

**let** [변수명] = 데이터값

**let** [변수명]:[데이터 자료형]

**let** [변수명]:[데이터 자료형] = 데이터값

(예)

let a

let a:Int

let a = 123

a = 567 (x) (값을 변경할 수 없다)

(\*var는 타입 추론을 하여 알아서 데이터의 자료형을 판단합니다.)

# Swift - 문자열 포맷

## 1) 문자열 조합

```
var str1 = "test1"
```

```
var str2 = "test2"
```

```
str1+str2 => "test1test2"
```

```
var str = "hello"+"world" => "helloworld"
```

## 2) 문자열 템플릿

```
let num = 123
```

```
let str = "Integer"
```

```
print("\(num) is \(str)") => "123 is Integer"
```

swift에서는 print함수가 기본 newline제공

=> print("Hello", terminator:"") 이런식으로 line break 사용.



# 조건문 - if / else

\* 조건의 참/거짓에 따라 해당 구문을 실행하겠금 분기구문

**1) 단독 if문:** if문에 조건식이 맞으면 실행

```
if 조건식 {  
    실행할 내용  
}
```

**2) if / else :** if문 조건식이 맞지 않으면  
else문을 실행

```
if 조건식 {  
    실행할 내용  
} else {  
    실행할 내용  
}
```

예제)

```
if 5 == 5 {  
    print("5와 5는 같습니다.")  
}
```

```
if 3 == 5 {  
    print("if문 출력")  
} else {  
    print("else문 출력") => 3 == 5이 같지 않기에 출력  
}
```

# 조건문 - if / else if / else

**3) if / else if :** if문 조건식이 맞지 않으면 else if 조건식을 판별하여 실행

```
if 조건식 {  
    실행할 내용  
} else if 조건 {  
    실행할 내용  
}
```

**4) if /else if / else**

```
if 조건식 {  
    실행할 내용  
} else if 조건 {  
    실행할 내용  
} else {  
    실행할 내용  
}
```

예제)

```
if 3== 5 {  
    print("if문 출력")  
} else if 3 == 3 {  
    print("else if문 출력")  
}
```

```
if 3== 5 {  
    print("if문 출력")  
} else if 3 == 1 {  
    print("else if문 출력")  
} else {  
    print("else문 출력")  
}
```

# 조건문 - switch-case

- **switch**의 비교대상에 따라 **case**별 실행문을 분기하는 조건문

<형식>

```
switch [비교대상]{  
  case [비교패턴1] : [실행문]  
  case [비교패턴2] : [실행문]  
  default: [실행문]  
}
```

예)

```
var char:Character = "B"  
switch char {  
  case "A": print("A")  
  case "B": print("B")  
  default: print("A나 B가 아니네")  
}
```

예)

```
var a:Int = 1  
switch a {  
  case 1: print("1")  
  case 2: print("2")  
  default: print("1이나 2가 아니네")  
}  
  
var b:Int = 7  
switch b {  
  case 0...3: print("0~3 범위")  
  case 4...6: print("4~6 범위")  
  default: print("7이상 범위")  
}
```

- \* 다른언어에는 **break**라는 제어전달문을 사용하지만 **swift**에서는 제공하지 않으며, 단일 **case**를 실행
- \* **default** 조건은 의무적이지 않음 (조건식에 따라 넣어야 하고, 안 넣을 수도 있다.)

# 조건문 - guard

- **guard** 키워드를 쓰면, 조건이 거짓(**false**)일 경우 **else**이 실행되는 구조
- 후속조치에 대한 조건문으로, 보통 **nil** 체크나 종료에 대한 예외처리에 많이 쓰임
- **else**문에는 **return** 제어전달문이 필수이며, 함수 안에서만 조건식이 가능

<형식>

```
guard [조건식] else {  
    실행문  
}
```

예)

```
var val1:String?  
func test2222(val1:String?){  
    guard val1 != nil else {  
        print("nil이네")  
        return  
    }  
}  
test2222(val1: val1)    //"nil이네" 출력
```

예제)

```
func foo(m:Int){  
    guard m > 2 else {  
        print("2보다 작습니다.")  
        return  
    }  
}  
  
foo(m:1) // “2보다 작습니다.” 출력
```

# 반복문 - for문

- 반복적인 작업을 위한 문법적 장치로 특정범위를 지정하여 반복
- 범위를 “...” 또는 “..<” 으로 표현
- 범위에 **collection**이나 연속적인 데이터도 대입 가능

<형식>

```
for [변수] in [시작]...[끝] { //시작부터 끝  
    실행문  
}
```

```
for [변수] in [시작]..<[끝] { //시작부터 끝미만  
    실행문  
}
```

예)

```
for num in 0...3 {  
    print("숫자: \num)") //0부터 3까지 반복  
    출력  
}
```

```
var str:String = "string123"
```

```
for char in str.characters {  
    print("문자: \char)") //문자 하나씩 출력  
}
```

```
var array:Array = [1,2,3,4]
```

```
for val in array {  
    print("\val)") //배열 각 요소값  
}
```

```
for (index, val) in array.enumerated() {  
    print("index: \index), value: \val)") //배열 index와 값  
}
```

# 반복문 - while / repeat-while

<형식>

```
while [조건식] {  
    실행문  
}
```

예)

```
var i=0  
while i < 3 {  
    print(i) //3번 반복 출력  
    i+=1  
}
```

```
var n=0  
while true {  
    if(n >= 3){ break }  
    print(n) //3번 반복 출력  
    n+=1  
}
```

- for문과 다르게 조건식이 만족(true)할 동안 계속 반복
- break 제어전달문으로 반복문을 임의로 종료가능
- repeat-while문 경우는 최초 repeat를 실행하고 조건식 판별

<형식>

```
repeat {  
    실행문  
} while [조건식]
```

예)

```
var j=0  
repeat {  
    print(j)  
    j+=1  
} while i < 3
```

# Swift - 옵셔널

- \* 데이터의 유무를 판별하기 위한 문법적 장치, ! 와 ?를 데이터자료형과 변수 뒤에 명시
- \* 옵셔널 값은 nil 이거나 nil이 아닌 경우로 나뉨
- \* 스위프트의 데이터는 초기화 값이 없는 경우 항상 옵셔널 타입 값으로 존재.
- \* ?를 붙이면 실제데이터는 Optional(데이터) 형식으로 데이터형을 감싸게 표현됨 (초기값 없을시 nil로 초기화)
  - 값의 optional 낙인을 찍으므로, 이 데이터가 존재하는지의 여부에 대한 문법적인 경고를 준다.
  - optional 붙은 자료형은 !로 값이 존재함을 알림으로 변수/데이터를 사용하게 됩니다.
- \* 프로그래밍의 null point exception에 대한 문법적 안전장치로 swift에 도입되었습니다.

(유형 3가지)

```
var str:String? //optional wrapping
```

```
var str:Int! //force unwrapping
```

```
//implicity unwrapping
```

```
if let str2 = str {  
    print(str2)  
}
```

(예제)

```
var str:String?
```

```
str = "String입니다." // Optional("String입니다.") 형태
```

```
var str2:String = str! // !로 str의 Optional 낙인을 지워 값을  
대입
```

```
// str의 값은 "String입니다." 형태
```

# Double Question - ??

- 옵셔널에 대한 null 처리에 대한 축약 연산자 (3항 연산자의 응용)
- nil 비교 후의 결과값을 (force unwrapping) 하거나 (??뒤의 값)을 적용

```
let str:String? = "test"  
let test = str ?? "empty"  
  
=> str != nil ? str! : "empty"
```

```
let str1:String? = "test"  
let test = str1 ?? "empty"  
=> "test"  
  
let str1:String? = nil  
let test2 = str1 ?? "empty"  
=> "empty"
```



# 함수 - func (function)

- 특정 기능을 하는 코드를 특정 방법으로 묶어낸 것, 다른 의미에서는 반환형이 있는 프로시저
- 매개변수, 반환형이 있고 **return** 제어전달문이 있는 일반적인 형태
- 매개변수나 반환형이 없는 경우도 있다.

<형식>

```
func [함수명](매개변수:타입) -> (반환형) {  
    실행내용  
    return 반환형값  
}
```

예)

```
func add1(val1:Int) -> (Int){  
    return val1+1  
}
```

add(val1:1) //2를 반환

```
func showString(){  
    print("string")  
}  
showString() // "string" 출력
```

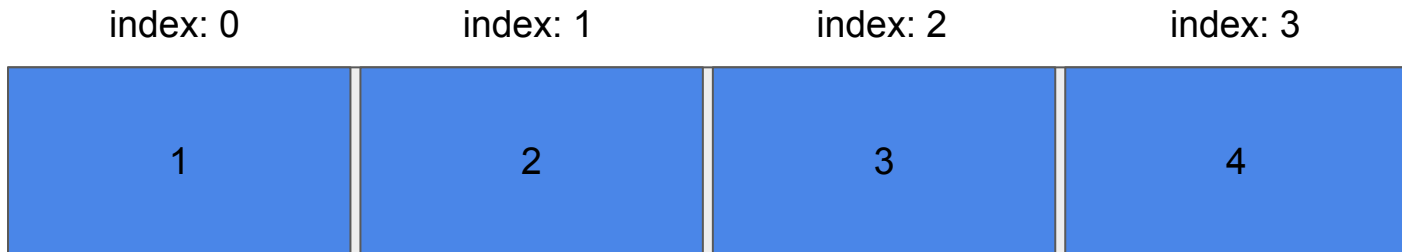
예)

```
func printStr(str:String) { //반환형 없는 경우  
    print("\{str}")  
}  
printStr(str:"test11")    //"test11" 출력
```

```
func printStr(_ str:String) { // _ 불일경우  
    print("\{str}")  
}  
printStr("test11") //변수명 명시 안해도 됨
```

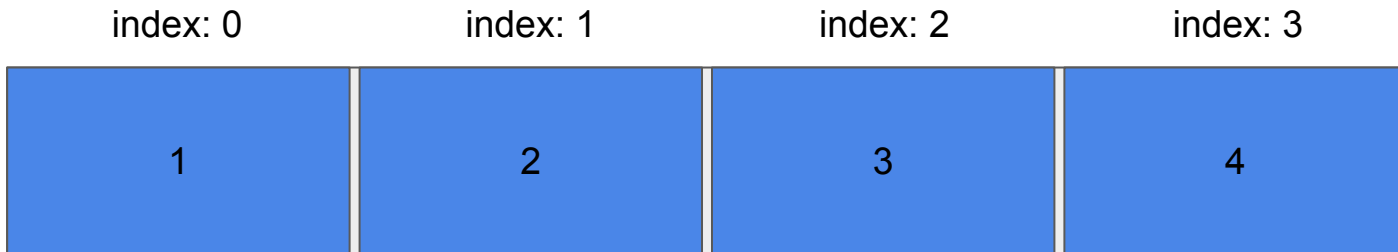
# 컬렉션(Collection) - 배열(Array)

- 컬렉션(Collection)은 컴퓨터 이론에서 쓰는 자료구조 형태 지정하는 용어
- Swift에서 지원되는 컬렉션 중 순차적으로 데이터를 저장/탐색 위한 자료구조 - 배열(Array)
- 각 위치별로 index라는 위치정보를 갖으며, 순차적인 데이터를 저장/검색할 때 주로 사용
- index 시작번호는 0 으로 시작되며, 한가지 종류의 자료형만 저장한다.



```
var array = [1,2,3,4]
array[0] // 1
array[1] // 2
```

# 배열(Array) - 선언 및 초기화



<배열 선언>

```
var array:Array<Int>  
var array = Array<Int>()  
var array:Array<Int> = Array<Int>()  
var array = [Int]()
```

<초기화>

```
var array = [1,2,3,4]  
var array:[Int] = [1,2,3,4]  
var array:Array<Int> = [1,2,3,4]
```

<추가>

```
array.append(값)
```

<삭제>

```
array.remove(at: index번호)
```

<수정>

```
array[index번호] = 값
```

```
array.count //배열의 전체크기, 길이
```

# Dictionary

- swift에 제공하는 mutable한 collection type 종류
- **key:value**라는 쌍데이터를 이루어져있으며, 적용할 타입제한이 없다.(단, 일치된 타입 사용)
- **key**의 타입 경우 해시연산 가능한 타입이어야 함 (식별가능한 키)

<형태>

[ key : value, key : value, ...]

ex)

[ 1 : "aaa", 2 : "bbb", 3 : "ccc"]

<타입 선언>

Dictionary<key type : value type>

ex)

Dictionary<Int, String>

# Dictionary

<선언>

```
Dictionary<Int, String>()
```

```
[Int, String]()
```

<초기화>

```
let dic:Dictionary<Int, String> = [1 : "aaa", 2 : "bbb", 3 : "ccc"]
```

```
let dic = [ 1 : "aaa", 2 : "bbb", 3 : "ccc"]
```

<추가/수정/삭제>

```
var dic = [String:String]()
```

```
dic["test1"] = "abc"    //test1라는 키이름으로 abc값 추가.
```

```
dic.updateValue("test2",forKey:"bbb") //test2키의 값을 bbb로 수정(키값 존재하지 않을시  
추가)
```

```
dic.updateValue("test1",forKey:"bbb") //test1키의 값을 bbb로 수정
```

```
dic.removeValue(forKey: "test1")    //test1키의 키와 값을 삭제
```

# Dictionary

- array와 달리 순차적인 저장을 하지 않음 (key값을 기준으로 정렬, 순차적 저장순서X)

<순회탐색>

```
let dictionary = ["test2":"bbb", "test1":"aaa"]           // [String:String]()
```

```
for (key,value) in dictionary {  
    print(key, value)  
}
```

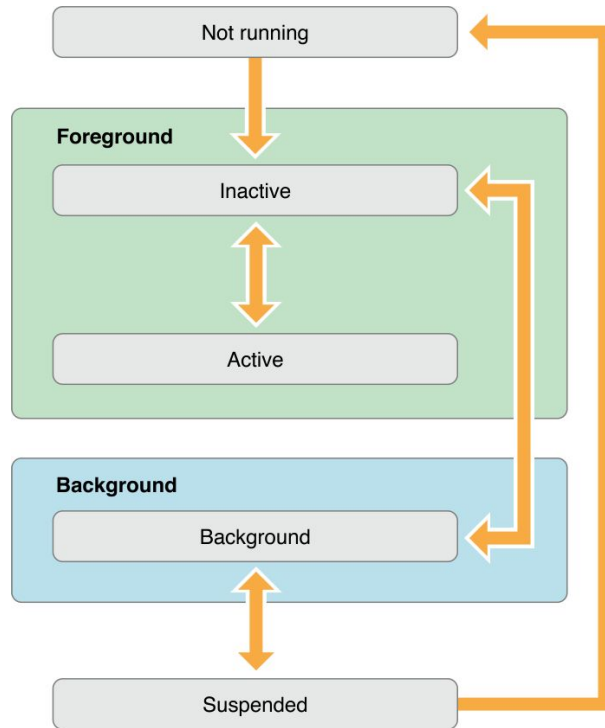
⇒ 출력

test1 bbb

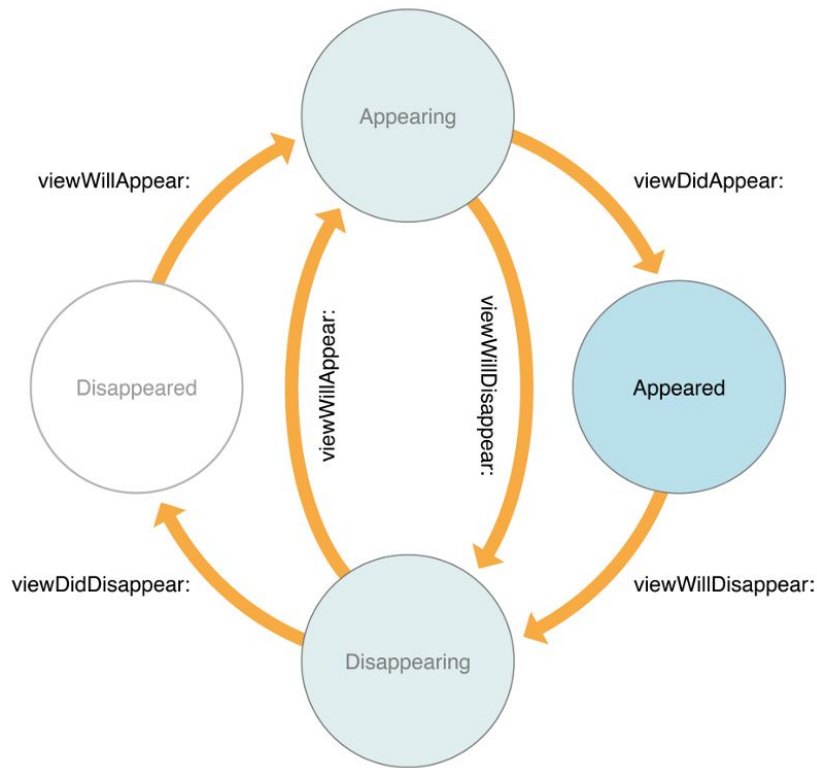
test2 bbb

# ios 앱 생명주기

- ios의 앱이 실행되고, 중지되는 생명주기 존재
- foreground**: 화면으로 보여지는 상태
- background**: 화면에 보여지지 않는 상태
- 기본적으로 **UIApplicationDelegate** 프로토콜을 구현한 형태인 **AppDelegate.swift**가 제공.



# ios 뷰 생명주기



`didReceiveMemoryWarning` : 메모리 부족시 호출

`viewDidLoad` : 최초 뷰를 호출

`viewWillAppear` : 메인뷰가 보이기 전

`viewWillLayoutSubviews` : 메인뷰의 자식뷰들이 보이기전

`viewDidLayoutSubviews` : 메인뷰의 자식뷰들이 보여진 후

`viewDidAppear` : 메인뷰 보여진 후

`viewWillDisappear` : 메인뷰가 사라지기 전

`viewDidDisappear` : 메인뷰가 사라진 후