클래스와 구조체

Classes & Structures

"*Classes* and *structures* are general-purpose, flexible constructs that become the building blocks of your program's code.
You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions."

Classes & Structures

- 프로그램 코드 블럭의 기본 구조이다.
- 변수, 상수, 함수를 추가 할수 있다. (두 구조의 문법 같음)
- 단일 파일에 정의 되며 다른 코드에서 자동으로 사용 할수 있습니다.(접근 제한자에 따라 접근성은 차이가 있다. internal 기본 접근제한자)
- 초기 상태를 설정하기 위해 initializer가 만들어 지고, 사용자가 추가로 정의할 수 있다.
- 사용 시 인스턴스(instance)라고 불린다.
- 기본 구현된 내용에 기능을 더 추가해서 확장 할수 있다. (Extensions)
- 프로토콜을 상속받아 사용할수 있다. (Protocols)

기본 구조

```
class SomeClass {
    // class definition goes here
}
```

```
struct SomeStructure {
    // structure definition goes here
}
```

기본 구조

```
struct Resolution {
    var width = 0
    var height = 0
}
```

```
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

인스턴스

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Properties 접근

```
print("Width of Resolution is \(someResolution.width)”)
print("Width of VideoMode is \(someVideoMode.resolution.width)”)

someVideoMode.resolution.width = 1280
```

- 닷(.) 문법을 통해 접근

Initialization

Initialization is the process of preparing an instance of a class, structure, or enumeration for use.

초기화

- 인스턴스에 설정된 속성의 초기값을 설정과 초기화하는데 목적이 있다.
- 클래스 및 구조체는 인스턴스로 만들어 질때 프로퍼티는 적절한 초기값으로 모두 초기화 해야 한다.
- 모든 구조체는 자동으로 Memberwise Initializers가 만들어 진다.

base Initializers

```
struct Subject {
    var name:String
}

class Student {

    var subjects:[Subject] = []

    func addSubject(name:String) {

        let subject = Subject(name: name)
        subjects.append(subject)
    }
}

var wingMan:Person = Person()
```

Memberwise Initializers

Initializers

Memberwise Initializers

```
struct Subject {
    var name:String
}

class Student {

    var subjects:[Subject] = []

    func addSubject(name:String) {

        let subject = Subject(name: name)
        subjects.append(subject)
    }
}

var wingMan:Person = Person()
```

Struct의 경우 모든 프로퍼티가 초기화 할수 있게 모든 멤버에 대해 initializer가 생긴다.
`init(name:String)`

Memberwise Initializers

Custom Initializers

```
class Student {
    var subjects:[Subject] = []

    func addSubject(name:String) {
        var sub1:Subject = Subject(gender:true)
        sub1.name = "joo"
        sub1.age = 30
        subjects.append(sub1)
    }
}

struct Subject {
    var name:String?
    var age:Int?
    var gender:Bool

    init(gender:Bool) {
        self.gender = gender
    }
}
```

상속과 Initializers

- 부모 클래스로부터 상속받은 모든 저장 속성은 초기화할 때 초기 값을 할당받아야 함.
- Swift는 클래스 타입에 모든 저장 속성에 초기 값을 받도록 도와주는 두가지 이니셜라이저를 정의함. 이를 지정 이니셜라이저(designated initializers)와 편의 이니셜라이저(convenience initializers)라고 함.

Designated initializers

```
init(parameters) {

   statements

}
```

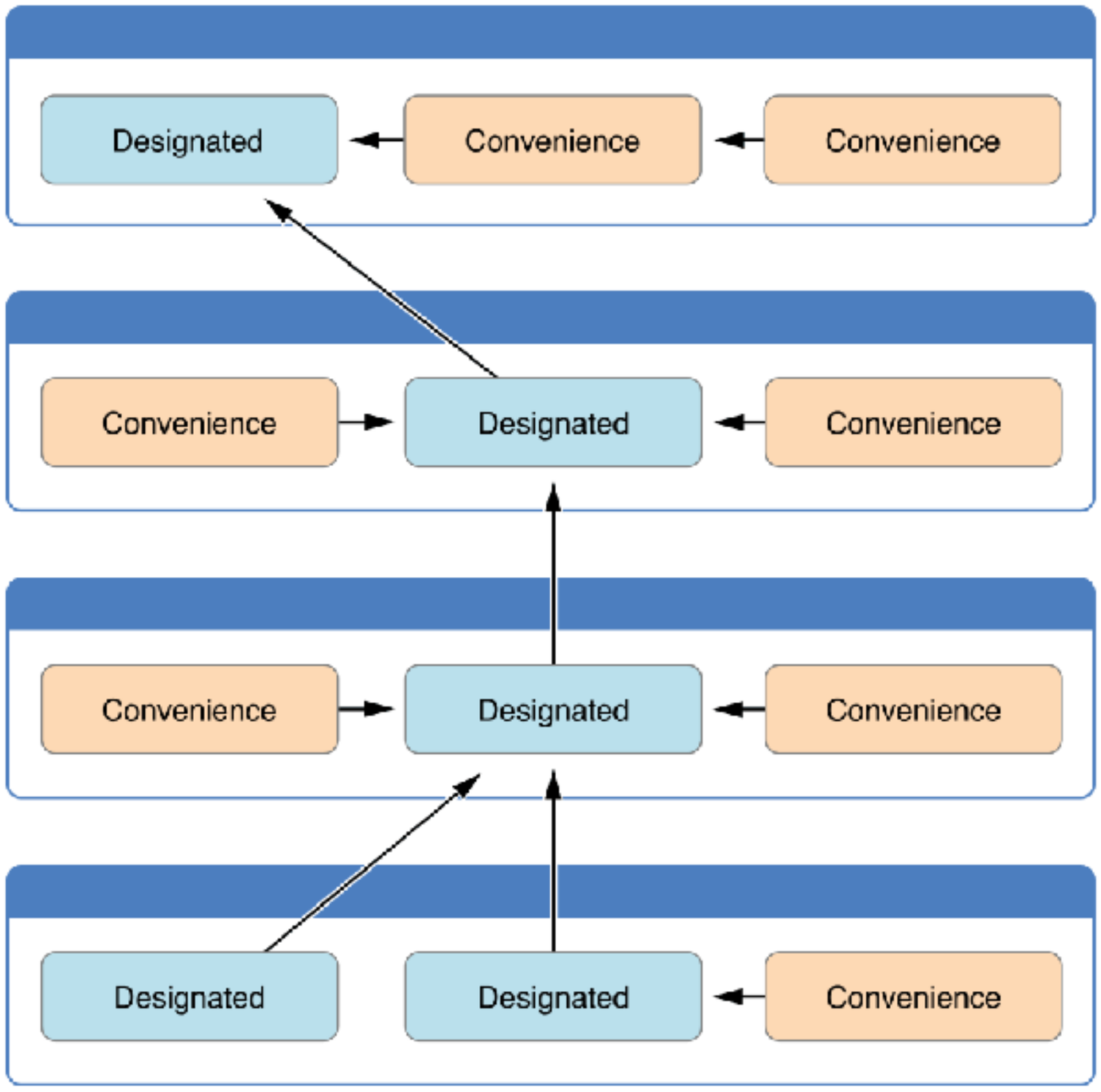
- 모든 프로퍼티가 초기화 되어야 한다.
- 상속을 받았다면 부모 클래스의 Designated initializers를 호출 해야 한다.

Convenience initializers

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

- 다른 convenience initializer을 호출할수 있다.
- 하지만 궁극적으론 designated initializer을 호출해야만 한다.

Designated & Convenience 모식도



Required Initializers

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

- 해당 initializer는 필수적으로 구현해야만 한다.
- 상속받은 모든 클래스는 필수로 구현해야 한다.
- required initializer를 구현할땐 override 수식어를 사용할 필요 없다.

Setting a Default Property Value with a Closure or Function

```
class SomeClass {

    let someProperty: SomeType = {
        // 해당 클로져 안에 프로퍼티의 기본값을 지정한다.
        // someValue는 반드시 SomeType과 타입이 같아야 한다.
        return someValue
    }()

}
```

- 클래스의 init시 해당 프로퍼티의 값이 할당되며, 값대신 클로져나 전역 함수를 사용할수 있다.
- 클로져 사용시 마지막에 **()**를 붙여 클로져를 바로 실행 시킨다.

Classes VS Structures

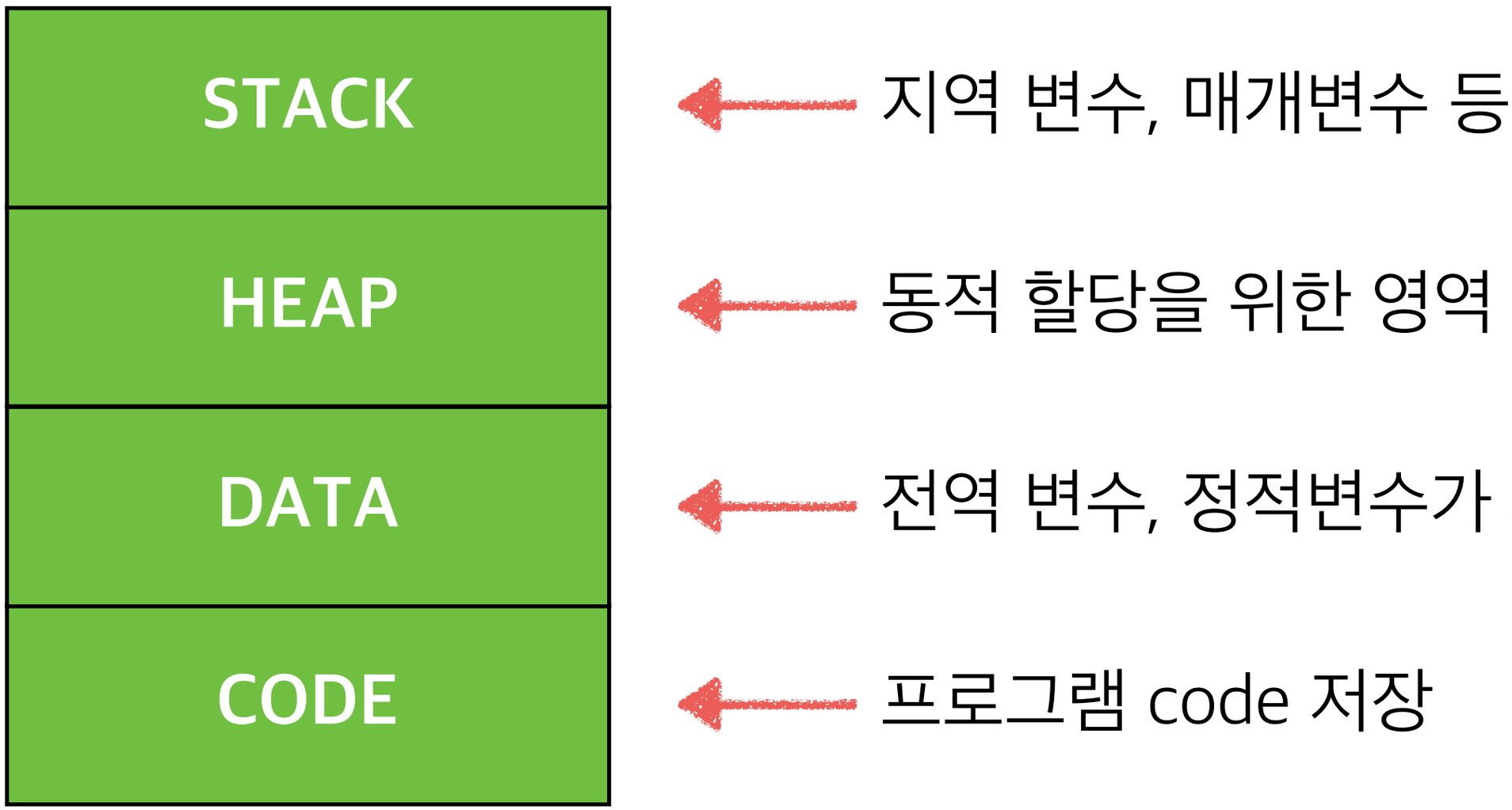
- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Class는 deinitializer를 사용할수 있습니다.

Classes VS Structures

- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Class는 deinitializer를 사용할수 있습니다.

값타입 vs 참조타입

Memory구조

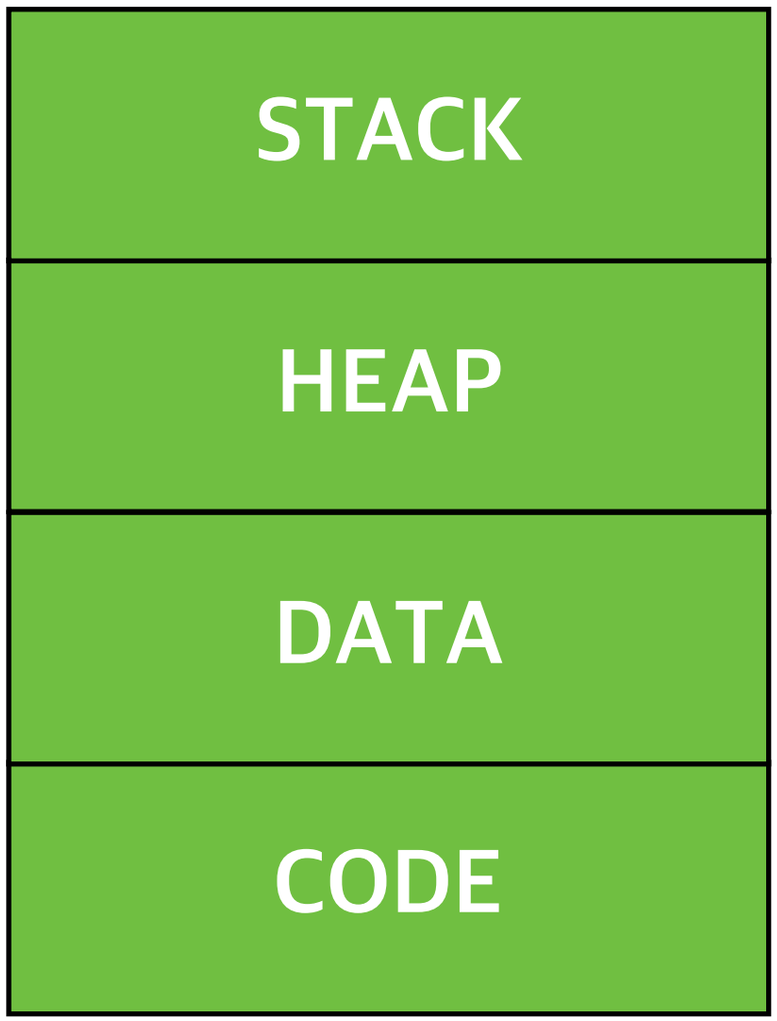


Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
var num:Int = 4
var num2:Int = 5
```

Memory구조 파악하기



num = 4, num2 = 5

프로그램 code 저장

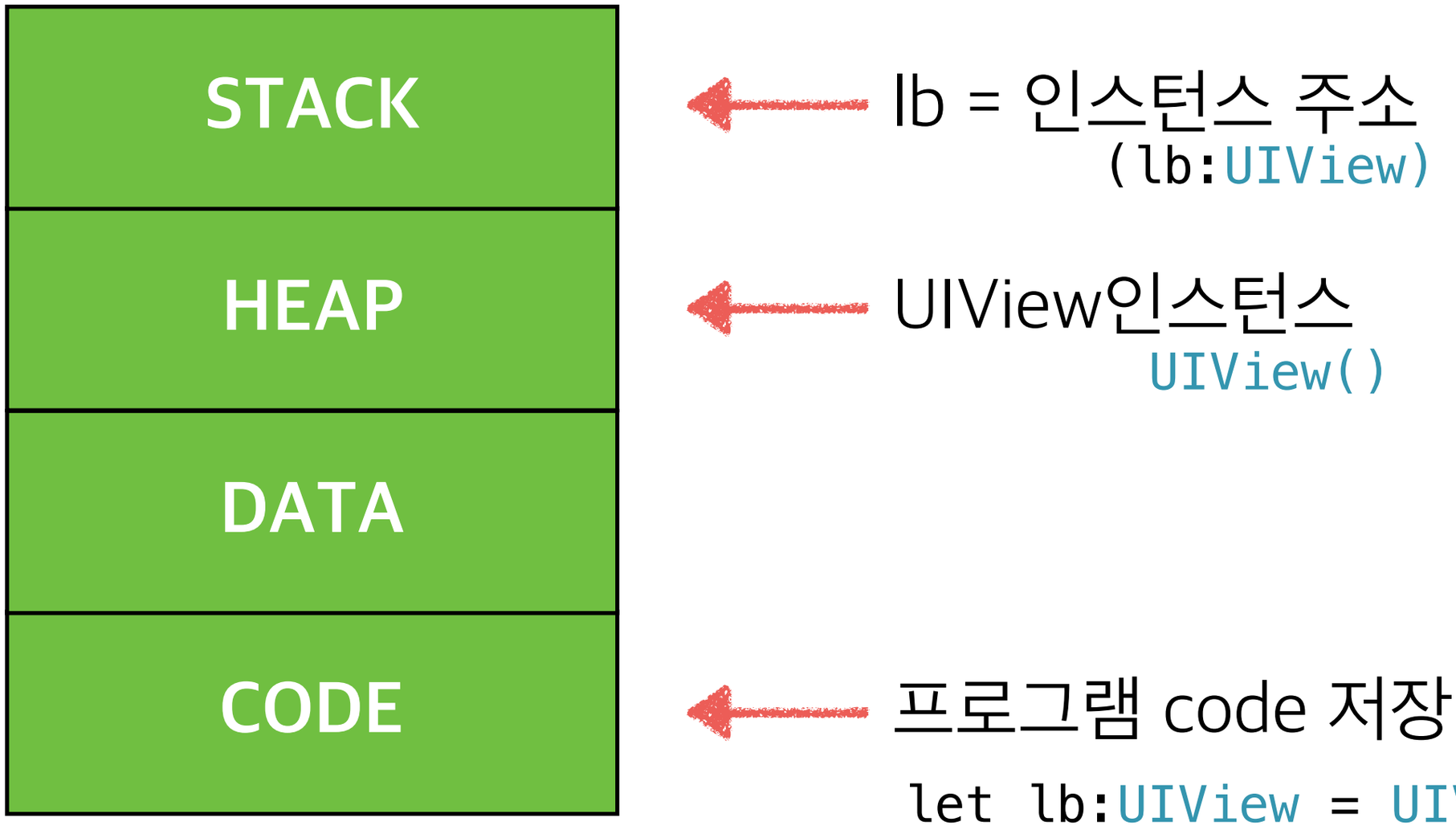
```
var num:Int = 4;
var num2:Int = 5;
```

Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
let lb:UIView = UIView()
```

Memory구조 파악하기

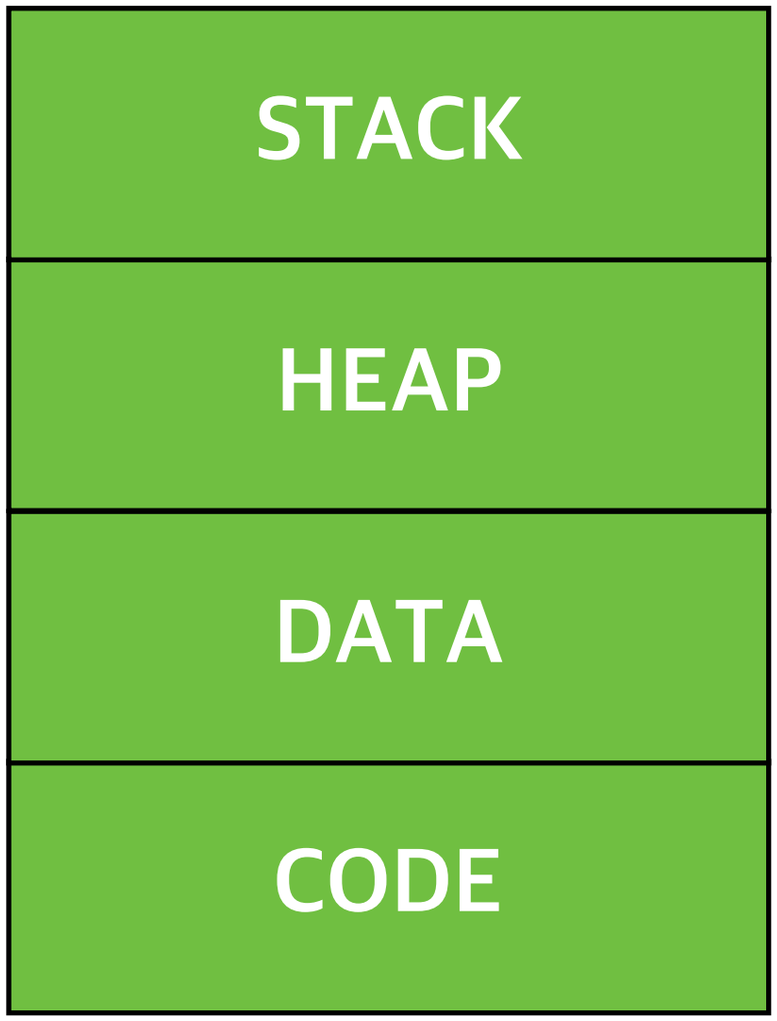


Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
static let number:Int = 5
```

Memory구조 파악하기



number = 5

프로그램 code 저장

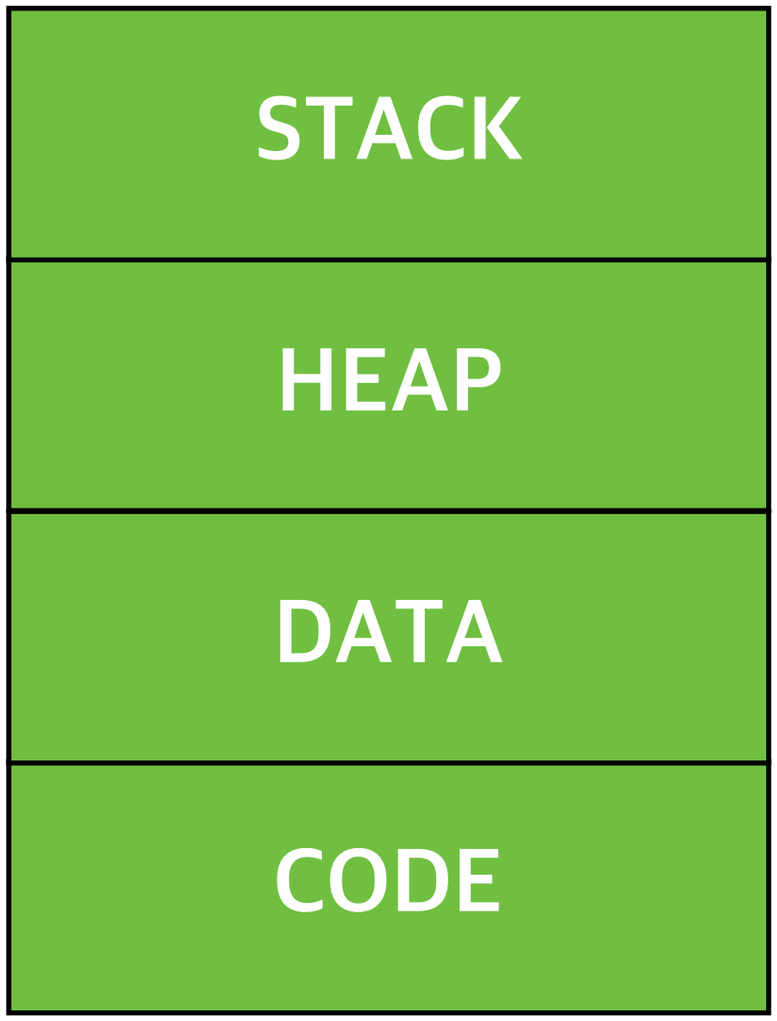
```
static let number:Int = 5
```

Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
func sumTwoNumber(num1:Int, num2:Int) -> Int
{
    return num1 + num2
}
```

Memory구조 파악하기



num1 = 3, num2 = 4

```
sumTooNumber(num1:3, num2:4)
```

프로그램 code 저장

```
func sumTwoNumber(num1:Int, num2:Int)
-> Int {
      return num1 + num2
}
```

Fast campus

두 코드의 차이점

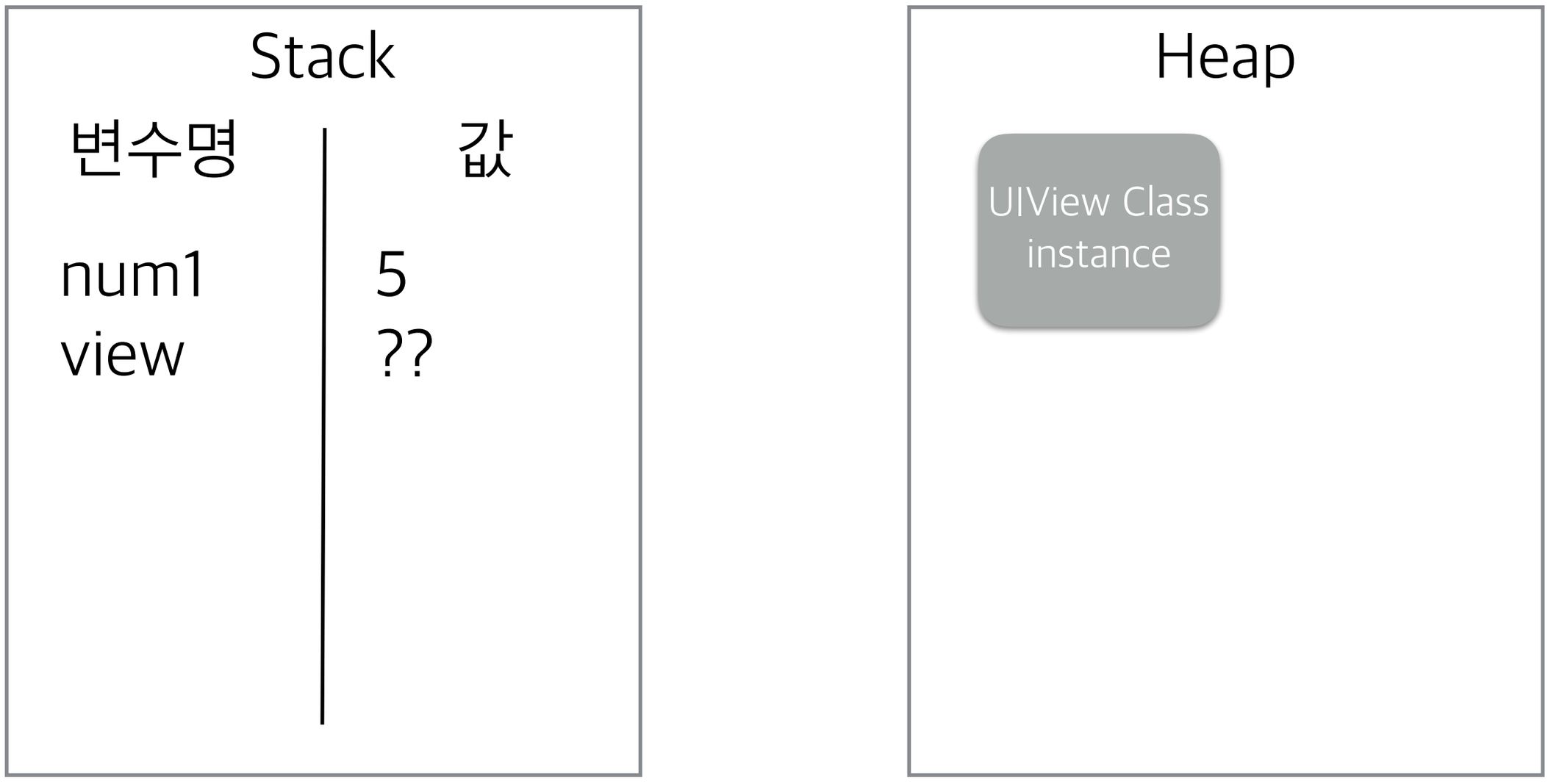
```
let num2:Int = 5
```

Struct VS Class

```
let lb:UIView = UIView()
```

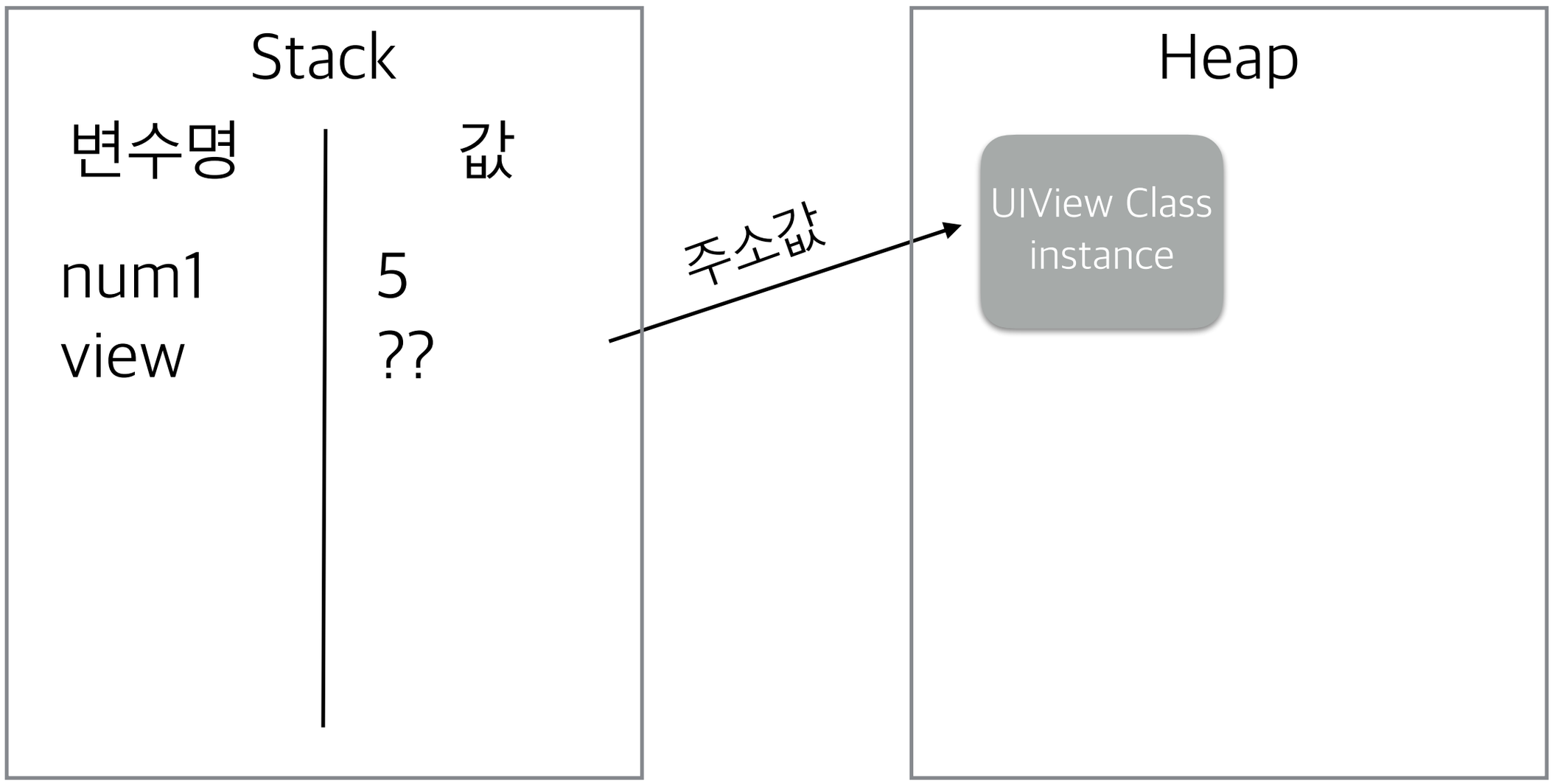
두 코드의 차이점

```
let num1:Int = 5
let view:UIView = UIView()
```



두 코드의 차이점

```
let num1:Int = 5
let view:UIView = UIView()
```



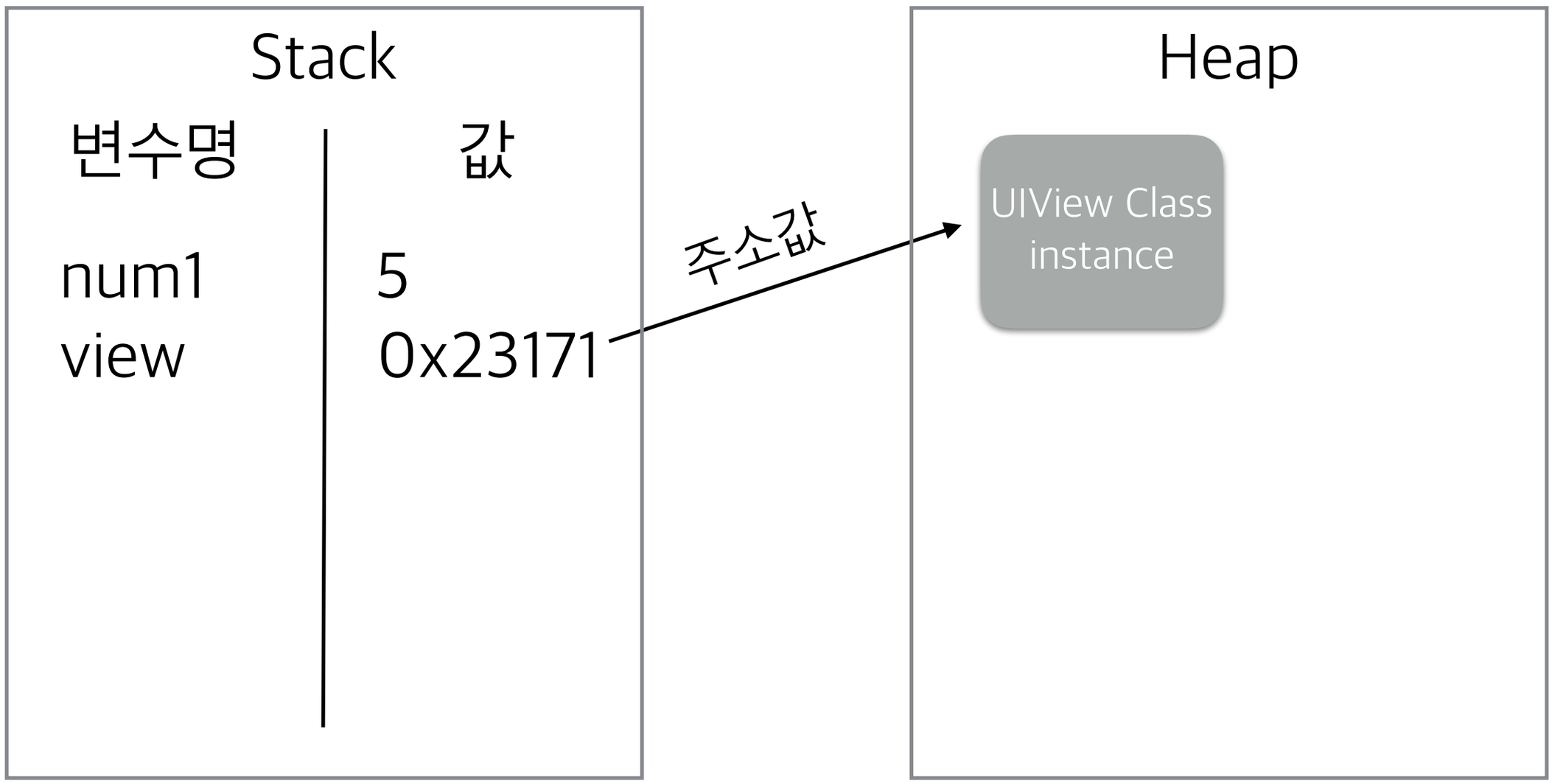
Pointer

- 포인터(pointer)는 프로그래밍 언어에서 다른 변수, 혹은 그 변수의 메모리 공간주소를 가리키는 변수를 말한다.

-Wikipedia-

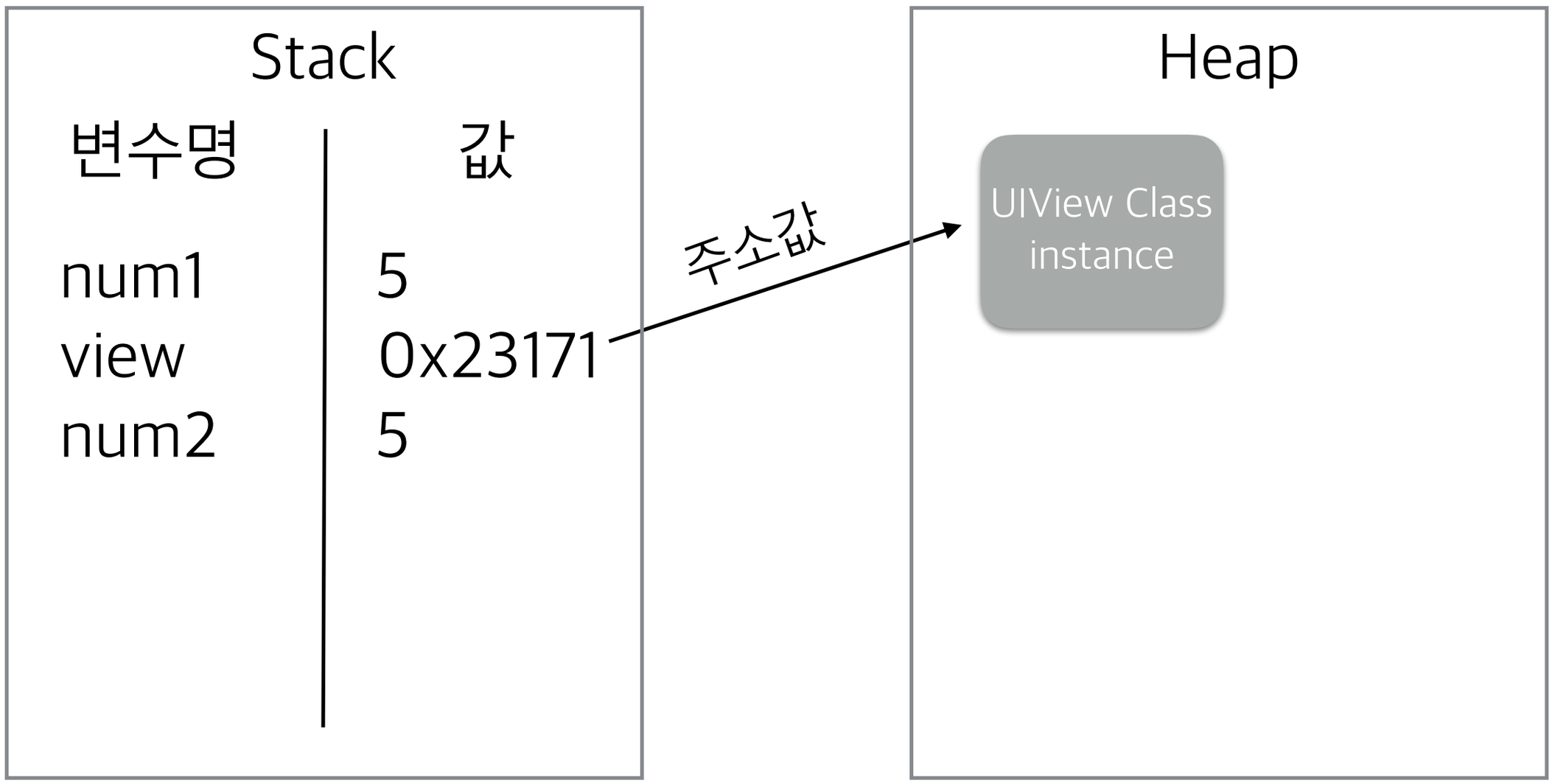
Pointer

```
let num1:Int = 5
let view:UIView = UIView()
```



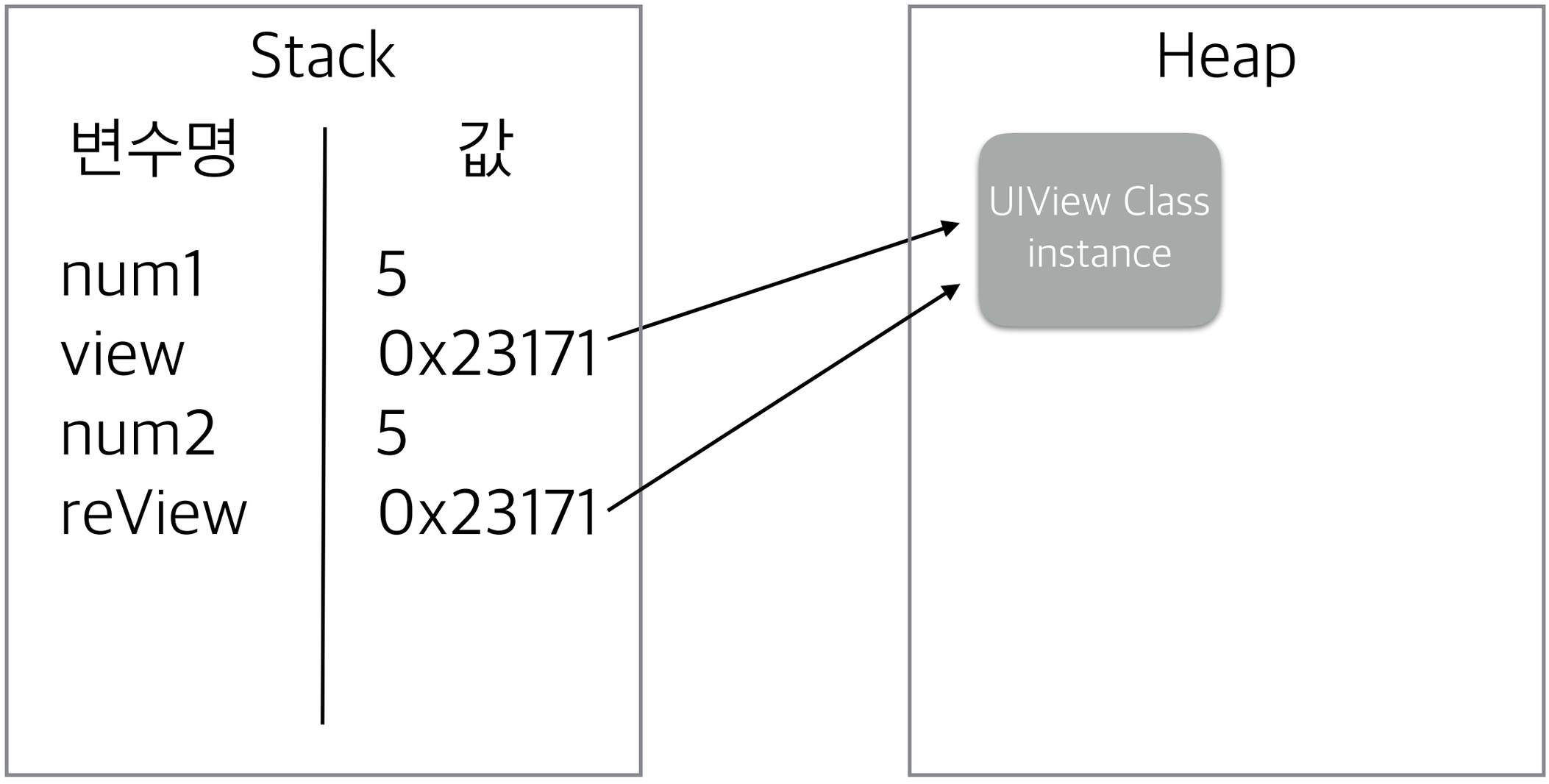
Pointer

```
let num1:Int = 5
let view:UIView = UIView()
let num2 = num1
```



Pointer

```
let view:UIView = UIView()
let num2 = num1
let reView:UIView = view
```



결과값은?

```
let view:UIView = UIView()
let reView:UIView = view
view.backgroundColor = .red
//현재 view의 배경색은? reView의 배경색은?

reView.backgroundColor = .gray
//현재 view의 배경색은? reView의 배경색은?
```

Classes VS Structures

- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Calss는 deinitializer를 사용할수 있습니다.

Value Type 프로퍼티 수정

- 기본적으로 구조체와 열거형의 값타입 프로퍼티는 인스턴스 메소드 내에서 수정이 불가능 하다.

- 그러나 특정 메소드에서 수정을 해야 할경우에는 mutating 키워드를 통해 변경 가능하다.

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
```

Deinit

```
class Student {

    init() {
        //인스턴스 생성시 필요한 내용 구현
    }

    deinit {
        //종료직전 필요한 내용 구현
    }


}
```

Classes VS Structures

- 어떤걸 선택해서 써야할까요?
- 기본 SDK에 클래스와 구조체의 예제를 찾아봅시다.

애플 가이드라인

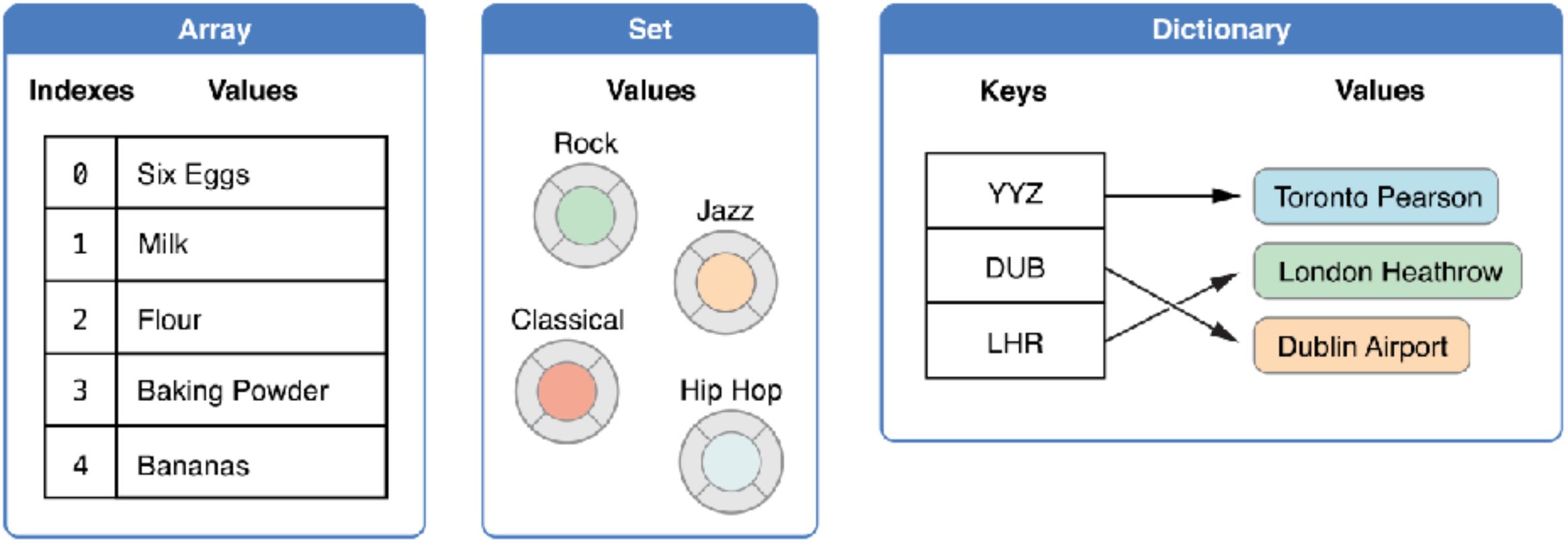
- 연관된 간단한 값의 집합을 캡슐화하는 것만이 목적일 때
- 캡슐화된 값이 참조되는 것보다 복사되는 것이 합당할 때
- 구조체에 저장된 프로퍼티가 값타입이며 참조되는 것보다는 복사되는 것이 합당할때
- 다른 타입으로부터 상속받거나 자신이 상속될 필요가 없을 때

다시보기 Collection Type

Fast campus

Collection Type

- Swift는 값의 모음을 저장하기위한 배열, 집합 및 사전이라는 세 가지 기본 형식을 제공 합니다. 배열은 정렬 된 값 모음입니다. 집합은 고유 한 값의 정렬되지 않은 모음입니다. 사전은 키 - 값 연관의 정렬되지 않은 모음입니다.



Mutability of Collections

- 변수(var) 에 할당하면 Collection를 변경가능하다.
- 즉 Collection에 추가, 제거, 수정할수 있다.
- 하지만 상수(let)에 할당하면 Collection를 변경 불가능 하다.

Array

- 배열(영어: array)은 번호(인덱스)와 번호에 대응하는 데이터들로 이루어진 자료 구조를 나타낸다. 일반적으로 **배열에는 같은 종류의 데이터들이 순차적으로 저장**되어, 값의 번호가 곧 배열의 시작점으로부터 값이 저장되어 있는 상대적인 위치가 된다.

Array 문법

- 기본 표현은 Array<Element>로 Array Type을 나타낸다.
- 여기에서 Element는 배열에 저장할수 있는 타입이다.
- 또 다른 축약 문법으로 [Element] 로 표현할 수 있다.

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

배열 리터럴

- 배열 리터럴 문법은 대괄호 [ ] 를 사용한다.

```
[ 값 1 , 값 2 , 값 3 ]
```

```
var someInts:[Int] = [1,2,3,4]
someInts = []
```

배열 Element 가져오기

- index를 통해 배열의 값을 가져올수 있다.
- index는 0부터 시작된다.

```
var someInts:[Int] = [1,2,3,4]
print("\(someInts[0])")
print("\(someInts[3])")
```

Dictionary

- Dictionary는 순서가 정해져 있지 않은 데이터에 키값을 통해 구분할수 있는 자료구조. 항목의 순서가 중요치 않고 key값을 통해서 데이터를 접근할때 사용합니다.

Dictionary 문법

- 기본 표현은 Dictionary<key, value>로 Dictionary Type을 나타낸다.

- Key값 은 Dictionary에서 value를 가져오는데 사용되는 값이다.

- 또 다른 축약 문법으로 [key:value] 로 표현할 수 있다.

```
var someInts:[String:Int] = [String:Int]()
var someInts:Dictionary<String,Int> = [:]
```

딕셔너리 리터럴

- 딕셔너리의 리터럴 문법은 [:] 를 사용한다.

```
[ 키 1 : 값 1 , 키 2 : 값 2 , 키 3 : 값 3 ]
```

```
var airports: [String:String] = ["ICH": "인천공항", "CJU": "제주공항"]
```

딕셔너리 Value 가져오기

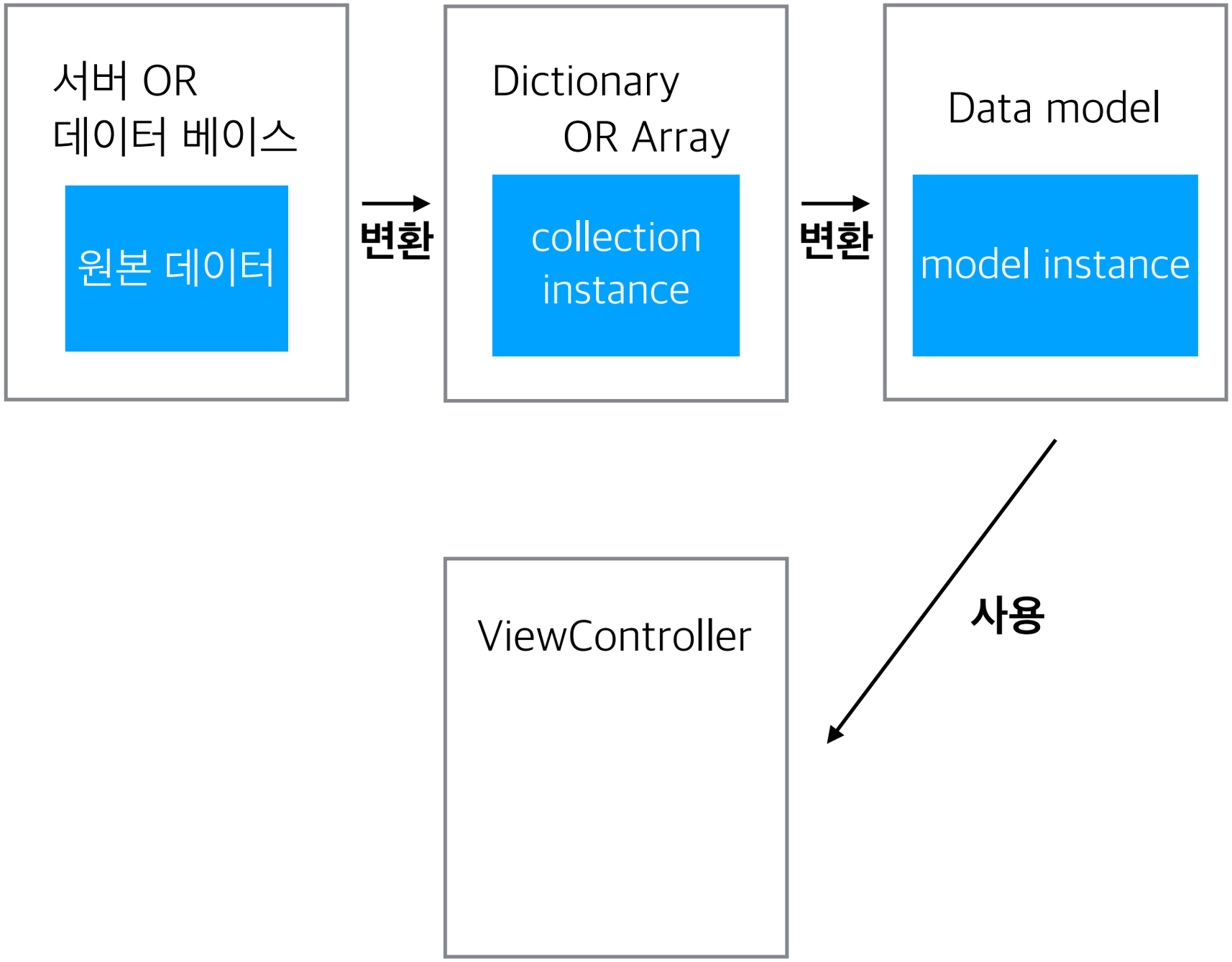
- key값을 통해 Value값을 가져올수 있다.

```
var airports: [String:String] = ["ICH": "인천공항", "CJU": "제주공항"]
print("\(airports["ICH"])")
print("\(airports["CJU"])")
```

Struct로 데이터 모델 만들기

- 서버나 데이터베이스에서 가져온 데이터를 바로 사용하지 않고 데이터 모델을 만들어서 사용
- Struct로 데이터 모델을 만들어 사용한다.(데이터의 경우 참조형 보다는 값 복사형 타입이 좋기때문)

데이터 컨트롤 흐름



왜 데이터 모델을 사용하나?

- 복잡한 구조의 Collection Type을 주로 사용하게 되며, 매번 사용할때마다 데이터를 끄집어 내기위해 Array & Dictionary instance를 만들어야 한다. (사용 편의성)

- Dictionary의 데이터는 key값을 통해 데이터에 접근 한다. String type인 key값은 다양한 곳에서 사용하게 되면 오타의 위험성이 커진다. (안정성)

- 데이터중 Dictionary의 key값 변경시 코드내 사용된 모든곳을 찾아서 직접 바꿔줘야 한다. (수정 용의)

데이터 모델 만들기

- 데이터 하나당 Property 하나로 매칭
- 필수 데이터와 기타 데이터를 접근 제한자로 구분
- 데이터 구조 단순화 작업
- 가공이 필요한 데이터 만들기

예제 : 로그인 정보

- 로그인 데이터

1. 사용자 ID
2. 비밀번호
3. 이메일
4. 추가 정보 : 생년월일, 성별,

- 실제 더미 데이터

```
let userDic:Dictionary<String,Any>
  = ["userID":"joo",
     “userPW":"12345!@",
     "email":"knightjym@gmail.com",
     "birthDay":"2017/10/15",
     "gender":1
    ]
```

예제 : 로그인 정보

- 실제 더미 데이터

```
let userDic:Dictionary<String,Any>
  = ["userID":"joo",
     “userPW":"12345!@",
     "email":"knightjym@gmail.com",
     "birthDay":"2017/10/15",
     "gender":1
]
```

- Model

key	valueType
userID	String
userPW	String
email	String
birthday	String
gender	Int

예제 : 로그인 정보

- Model

key	valueType
userID	String
userPW	String
email	String
birthday	String
gender	Int

- DataModel Struct

```
struct UserModel
{
    var userID:String
    var pw:String
    var email:String
    var birthday:String?
    var gender:Gender?

    init?(dataDic:[String:Any])
    {
        //다음 페이지

    }
}
```

로그인 정보 초기화 메소드 예제

```
init?(dataDic:[String:Any])
{
    //필수 항목 모델화
    guard let userID = dataDic["UserID"] as? String else {return nil}
    self.userID = userID

    guard let pw = dataDic["userPw"] as? String else {return nil}
    self.pw = pw

    guard let email = dataDic["email"] as? String else {return nil}
    self.email = email
   //옵셔널 항목 모델화
    self.birthday = dataDic["birthDay"] as? String

    if let rawData = dataDic["gender"] as? Int,
           (rawData == 1 || rawData == 2)
    {
        self.gender = Gender(rawValue:rawData)
    }
}
```

복잡한 데이터 모델

- 음악앨범 정보

1. 앨범 정보
   - 앨범 타이틀
   - 가수 이름
   - 장르

2. [노래 정보 리스트]
   - 노래 정보
     - 노래제목
     - 가수
     - 작곡가, 작사가
     - 총 플레이 시간
     - 노래 URL

- 실제 더미 데이터

```
let album:Dictionary<String,Any> =
    ["albumInfo":["albumTitle":"2집 Oh!",
                  "artist":"소녀시대",
                  "genre":"댄스"],
     "songList":[["songTitle":"Oh!",
                  "trackNum":1,
                  "artist":"소녀시대",
                  "writer":"김정배",
                  "playTime":12340,
                  "playURL":"http://music.naver.com/123"],
                 ["songTitle":"Show! Show! Show!",
                  "trackNum":2,
                  "artist":"소녀시대",
                  "writer":"김부민",
                  "playTime":10130,
                  "playURL":"http://music.naver.com/124"],
                 ["songTitle":"웃자 (Be Happy)",
                  "trackNum":4,
                  "artist":"소녀시대",
                  "writer":"이트라이브",
                  "playTime":12134,
                  "playURL":"http://music.naver.com/126"]
          ]
]
```

- Model

key	valueType		
albumInfo	Dictionary		
	key	valueType	
	albumTitle	String	
	artist	String	
	genre	String	
songList	Array		
	[	Dictionary	
		key	valueType
		songTitle	String
		trackNum	Int
		artist	String
		writer	String
		playTime	Int
		playURL	String
	]		

DataModel Struct

- 복잡한 구조의 데이터를 구분하여 만들기 위해선?!
- 키-Value가 있는 Dictionary구조마다 데이터 모델 만들기
- AlbumModel, AlbumInfo, SongData

DataModel Struct

```
struct AlbumInfo
{
    var albumTitle:String
    var artist:String
    var genre:String

    init?(dataDic:[String:Any])
    {
        //이전방법과 동일
    }
}
```

```
struct SongData
{
    var songTitle:String
    var trackNum:Int
    var artist:String
    var writer:String
    var playTime:Int
    var playURL:String

    init?(dataDic:[String:Any])
    {
        //이전방법과 동일
    }
}
```

DataModel Struct

```
struct AlbumModel
{
    var albumInfo:AlbumInfo
    var songList:[SongData] = []

    init?(dataDic:[String:Any])
    {
        guard let infoDic = dataDic["albumInfo"] as?
                            Dictionary<String,Any> else {return nil}
        //데이터 인스턴스 만들기
        albumInfo = AlbumInfo(dataDic: infoDic)!

        guard let list = dataDic["songList"] as?
                         [Dictionary<String,Any>] else {return nil}
        //for문을 통해 각 데이터를 모델로 만든후 Array에 추가
        for songDic in list
        {
            songList.append(SongData(dataDic: songDic)!)
        }
    }
}
```

Data 저장

강사 주영민

Property list

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)
Item 0	String	armv7
▶ Supported interface orientati...	Array	(3 items)

Property list - plist

- 심플한 “파일” 저장 방법 중 하나.
- Key,Value구조로 데이터 저장
- File 형태로 저장되다 보니 외부에서 Access가능(보안 취약)

파일 위치

- 파일이 저장되는곳 Bundle & Documents 폴더
- Bundle은 프로젝트에 추가된 Resorce가 모인 곳
- 프로그램이 실행되며 저장하는 파일은 Documents폴더에 저장된다.
- **즉! plist파일의 데이터만 불러오는 역할은 Bundle을 통해서, plist파일에 데이터를 쓰고 불러오는 역할은 Documents폴더에 저장된 파일로!**

Plist File In Bundle

1. bundle에 있는 파일 Path 가져오기
2. Path를 통해 객체로 변환, 데이터 불러오기
3. 사용

Bundle

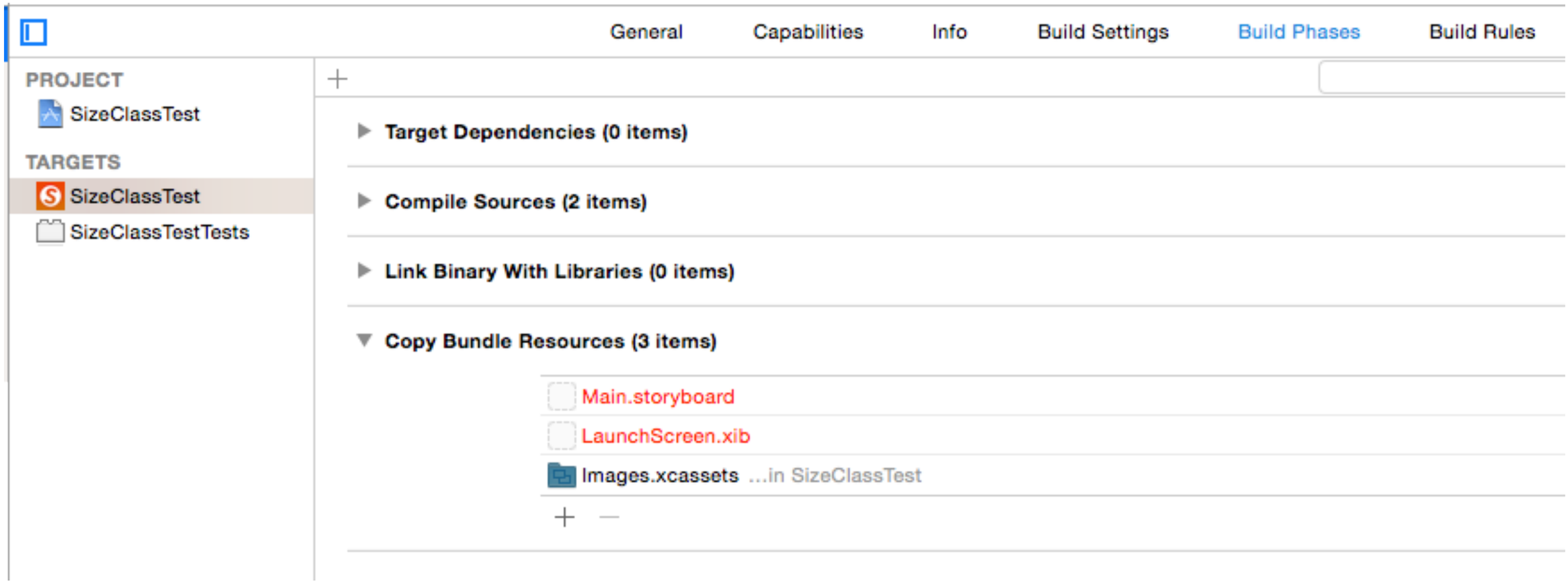
강사 주영민

Fast campus

Bundle

- 실행코드, 이미지, 사운드, nib 파일, 프레임 워크,설정파일 등 코드와 리소스가 모여있는 file system내의 Directory

Bundle 리소스 확인



Main Bundle 가져오기

```swift
// Get the main bundle for the app.
let mainBundle = Bundle.main
```

리소스 파일 주소 가져오기

```
// Get the main bundle for the app.
let mainBundle = Bundle.main
let filePaht = mainBundle.path(forResource: "rName", ofType:
"rType")
```

데이터 불러오기

```
if let path = filePaht {

    let image = UIImage(contentsOfFile: filePaht!)

}
```

Plist File In Bundle

```swift
if let filePaht = mainBundle.path(forResource: "rName", ofType: "rType"),
   let dict = NSDictionary(contentsOfFile: filePaht) as? [String: Any]
{
   // use swift dictionary as normal
}
```

Plist파일 불러오기

- 앱내 저장되어 있는 plist 리소스 파일의 데이터는 번들을 통해 가져올 수 있다.

Plist File In Document

1. Document folder Path 찾기

2. Document folder에 plist 파일이 있는지 확인
   - 만약 없다면 : bundle에 있는 파일 Document에 복사

3. Path를 통해 객체로 변환, 데이터 불러오기

4. writeToFile 메소드로 파일 저장

1. 파일 불러오기 (NSSearchPathForDirectoriesInDomains)

```
let path:[String] =
NSSearchPathForDirectoriesInDomains(.documentDirectory, .userDomainMask,
true)
let basePath = path[0] + "/fileName.plist"
```

- doucument 폴더에 Path구하기

2. Document folder에 파일 있는지 확인

```
if !FileManager.default.fileExists(atPath: basePath)
{

}
```

- doucument 폴더에 plist파일이 존재 하지는지 확인

3. bundle에 있는 파일 Document에 복사

```
 if let fileUrl = Bundle.main.path(forResource: "fileName", ofType: "plist")
{
    do {
        try FileManager.default.copyItem(atPath: fileUrl, toPath: basePath)
    } catch  {
        print("fail")
    }
}
```

- 만약 doucument에 해당 plist파일이 존재 하지 않을때, bundle에 있는 파일을 document폴더로 복사

4. Dictionary 인스턴스 불러오기

```swift
if let dict = NSDictionary(contentsOfFile: basePath) as? [String: Any]
{
   // use swift dictionary as normal
}
```

- document 폴더에 있는 파일을 NSDictionary을 통해서 Dictionary인스턴스로 불러오기

5. write(toFile)메소드를 통해 파일 저장

```
if let dict = NSDictionary(contentsOfFile: basePath) as? [String: Any]
{
    var loadData = dict
    loadData.updateValue("addData", forKey: "key")

    let nsDic:NSDictionary =  NSDictionary(dictionary: loadData)
    nsDic.write(toFile: basePath, atomically: true)
}
```

- dictionary를 수정
- NSDictionary로 변경후 writeTofile 메소드를 통해 파일에 저장

실습

- 한번 같이 만들어 볼까요?

Singletone Pattern

강사 주영민

Singleton Pattern

1. 싱글톤 이란? 어플리케이션 전 영역의 걸쳐 하나의 클래스의 단 하나의 인스턴스만(객체)을 생성하는 것을 싱글톤 패턴이라고 한다.

2. 사용이유 : 어플리케이션 내부에서 유일하게 하나만 필요한 객체에서 사용(셋팅, 데이터 등)

3. 클래스 메소드로 객체를 만들며 static을 이용해서 단 1개의 인스턴스만 만든다.

4. 앱 내에서 공유하는 객체를 만들 수 있다.

Singleton Pattern 인스턴스 만들기

```
class SingletonClass {
    // MARK: Shared Instance
    static var sharedInstance:SingletonClass = SingletonClass()

    // Can't init is singleton
    private init()
    {
        //초기화가 필요하면 private로 생성
    }

}
```

Singleton Pattern 예제

```
//스크린 정보를 가지고 있는 객체
let screen = UIScreen.main

//사용자 정보를 저장하는 객체
let data = UserDefaults.standard

//어플리케이션 객체
let app = UIApplication.shared

//파일 시스템 정보를 가지고 있는 객에
let fileManager = FileManager.default
```

“싱글턴 패턴은 왜 사용하는 것일까?”