Swift Study 14

2016. 03.04

Swift 문법

- 클로저(closure)
- 클래스(class) / 구조체(struct)
- 타입 캐스팅
- 열거형(enumeration)
- 접근제한자 / extension / final

일급 객체 / 함수

일급 객체(First class object)

일급/이급 객체 개념은 영국 컴퓨터 과학자 크리스토퍼 스트레이치에 의해 1960년대에 소개가 되었고, Algol언어의 Real number와 프로시저를 비교함으로 일급 객체의 개념에 대해 처음으로 언급 했다.
90년대 들어와 미국 컴퓨터 과학자 라파엘 핀켈이 이급/삼급 정의를 제안 했으나, 받아들여지진 않았다.

조건

- 변수나 데이터 구조안에 담을 수 있다.
- 파라미터로 전달 할 수 있다.
- 반환 값으로 사용할 수 있다.
- 할당에 사용된 이름과 관계없이 고유하게 구별이 가능하다.
- 동적으로 Property 할당이 가능하다.

Scala type의 Integer, Floating point number의 경우 거의 모든 언어에서 항상 일급 객체에 해당한다.

개념	기술	지원언어
일급함수	클로저	Scheme,ML,Haskell,F#,Scala,Swift
일급 클래스	메타 클래스	Smalltalk, Object-c,Ruby,Python

익명함수 / 클로저 (closure)

- 이름없는 함수 또는 익명함수, 람다함수라고 칭함.
- 코드 내부에 일시적으로 생성하여 사용함수
- 매개변수의 와일드카드 이용하여 매개변수 사용가능
- 시작 index 0부터 : $0 / $1 / $2

```
{ (매개변수:타입) -> 리턴타입 in
        //실행구문 또는  return 정보
}
```

case.1 함수정의 후 적용할 경우

```
func backward(_ s1: String, _ s2: String) -> Bool {
   return s1 > s2
}
 var reverseNames.sorted(by:backward)
//reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

case.2 클로저 사용할 경우

```
var reverseNames.sorted(by:{(s1:String, s2:String) -> Bool in  return s1 > s2 })
//reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

익명함수 / 클로저 (closure)

ex) func sorted(by: (String, String)-> Bool))

1) 기본형

```
sorted(by:{(s1:String, s2:String) -> Bool in
        return s1 > s2
})
```

2) 매개변수타입 생략

```
sorted(by:{(s1, s2) -> Bool in
        return s1 > s2
})
```

익명함수 / 클로저 (closure)

ex) func sorted(by: (String, String)-> Bool))

3) 리턴타입 생략

```
sorted(by:{(s1, s2) in
    return s1 > s2
})
```

4) 암묵적 리턴타입

```
sorted(by:{(s1, s2) in s1 > s2})
```

익명함수 / 클로저 (closure)

ex) func sorted(by: (String, String)-> Bool))

5) 매개변수의 와일드카드 사용
sorted(by:{ $0 > $1 })

6) 꼬리형태의 클로저**
sorted(){ $0 > $1 } //함수 뒤에 클로저 정의

sorted{ $0 > $1 } //함수() 생략

클래스 - class

- 데이터변수/함수(메소드)를 담는 틀 / 객체
- 클래스는 변수(프로퍼티)와 함수(메소드)를 가지며, 상속기능과 인스턴스를 만들 수 있다.
- 객체: 틀, 인스턴스: 실질적인 값을 갖는 존재 (예, 붕어빵틀 - 클래스/객체, 붕어빵 - 인스턴스)

<형식>

```
class [클래스명] {
  변수 선언

  func 함수정의 {

  }
}
```

예)

```
class Test {
  var a:String          //프로퍼티

  func testFunc(){   //메소드
    print("test")
  }
}

var test = Test()     //객체생성 (인스턴스), 생성자
test.a                //프로퍼티 접근
test.testFunc()       //메소드 호출
```

구조체 - struct

- 구조체는 변수의 값을 복사 형태
- 구조체는 클래스와 같이 기본적인 프로퍼티와 메소드, 초기화 구문 사용가능
- 구조체는 클래스와 달리 상속기능이나 소멸자 기능 없음, 타입 캐스팅 불가

<형식>

```
struct [구조체명] {
  변수 선언

  func 함수정의 {

  }
}
```

예)

```
struct Test {
  var a:String          //프로퍼티

  func testFunc(){   //메소드
    print("test")
  }
}

var test = Test()     //객체생성 (인스턴스), 생성자
test.a                //프로퍼티 접근
test.testFunc()       //메소드 호출
```

call of value / call of reference

- call of value : 값 자체를 복사 형태 (구조체 인스턴스)
- call of reference: 값의 메모리 주소 참조 (클래스 인스턴스)

```
//call of value
struct Test {
    var name = ""
    func print() -> String {
        return "name: \(name)"
    }
}
var test1 = Test()
test1.name = "test1"
test1.print()         //test1

var test2 = test1
test2.name = "test2"
test2.print()         //test2
test1.print()         //test1
```

```
//call of reference
class ClassTest {
    var name = ""
    func print() -> String {
        return "name: \(name)"
    }
}
var cTest1 = ClassTest()
cTest1.name = "clasTest1"
cTest1.print()      //clasTest1

var cTest2 = cTest1
cTest2.name = "clasTest2"
cTest2.print()      //clasTest2
cTest1.print()      //clasTest1
```

프로퍼티 / 메소드

- 프로퍼티 : 클래스/구조체/enum 내에 정의된 변수
- 메소드: 클래스/구조체/enum 내에 정의된 함수

예)
```
struct PersonInfo{
  //프로퍼티
  var name = ""
  var age = 0

  //메소드
  func getInfo() -> String {
    return "name: \(name) age: \(age)"
  }
}
```

클래스)
```
class Person{
  //프로퍼티
  var userId = 0
  var name = ""

  //메소드
  func getPersonInfo() -> String {
        return "userId: \(userId) name: \(name)"
  }
}
```

인스턴스 생성 및 참조

- **클래스/구조체/enum의 메모리에 할당된 실체**
- **다른언어와 달리 swift에서는 각 클래스/구조체/enum명만 선언하여 인스턴스 생성**
- **인스턴스의 프로퍼티/메소드 참조시 점(.)으로 접근**

1) 인스턴스 생성

```
let personInfo = PersonInfo()  //구조체 PersonInfo를 personInfo라는 인스턴스로 생성

let person1 = Person()  //클래스 Person를 person1라는 인스턴스로 생성
```

2) 인스턴스 프로퍼티 참조

```
let personInfo = PersonInfo()  //구조체
person.name         //프로퍼티 접근
person.getPersonInfo()   //메소드 접근

let person1 = Person()  //클래스
person1.userId      //프로터피 접근
person1.getInfo()  //메소드 접
```

프로퍼티 - 저장 프로퍼티

- **클래스/구조체/enum의 일반적인 변수를 지칭.**
- **그 외의 lazy키워드를 이용한 지연 저장 프로퍼티, 클로저로 프로퍼티를 초기화시킬 수 있음.**
- **lazy키워드 경우 호출하는 시점에서 값 초기화**
  (느슨한 연산 초기화, 시스템 부하 줄이거나 초기화시점 늦춤)

1) 지연 저장 프로퍼티

```
class LazyTest {
        lazy var late = Test()    //lazy 프로퍼티
}
var test = LazyTest()
test.late      //late호출시점에 Test() 인스턴스 생성
```

2) 클로저를 이용한 초기화

```
let/var 프로퍼티명:타입 = {
        정의내용
    return 반환값
}()

ex)
class Test(){
        var value:Int! = {
                return 1+1;
        }()
}
```

프로퍼티 - 연산 프로퍼티

- 클래스/구조체/enum의 변수를 간접적으로 접근 (setter/getter 용도)
- 연산 프로퍼티 자기 자신은 저장/변경할 수 없음.

형태)

```
class/struct/enum 객체명 {
        var 프로퍼티명:타입 {
                get {
                        //연산내용
                        return 반환값
                }
                set(매개변수명){
                        //필요연산
                }
        }
}
```

ex)

```
struct UserInfo {
        var a = 0
        var b = 0

        var data: Int {
                get {
                        return a + b
                }
                set(value) {
                        self.a = value / 2
                        self.b = value / 2
                }
        }
}
```

```
var test = UserInfo()
test.a              // 0
test.b              // 0
test.data = 10      // a=5, b=5
test.a              // 5
test.b              // 5
test.a = 6
test.b = 10
test.data           // 16
```

프로퍼티 - 옵저버 프로퍼티

- 클래스/구조체/enum의 프로퍼티의 변경시점의 상태제어를 할 수 있는 문법장치
- willSet(변경전) / didSet(변경후) 두가지 정의하여 사용
- 옵저버 프로퍼티 사용시 setter/getter를 사용불가.

1) willSet

```
var <프로퍼티명> : <타입> [= <초기값>] {
    willSet(매개변수명){
        실행내용
    }
}
```

2) didSet

```
var <프로퍼티명> : <타입> [= <초기값>] {
        didSet(매개변수명){
            실행내용
        }
}
```

프로퍼티 - 옵저버 프로퍼티

```
struct Job {
  var oldValue = 10
  var income:Int = 0{
        willSet(newIncome){
          self.income = newIncome
          print(self.income)
        }
        didSet{
           if income > self.oldValue {
                print("월급이 \(income-self.oldValue) 증가")
           }else{
                print("월급이 늘지 않았습니다.")
           }
        }
  }
}
```

```
var job = Job()

job.income = 9
//willSet:9, didSet:월급이 늘지 않았습니다.

job.income = 20
//willSet:20, didSet:월급이 10 증가
```

프로퍼티 - 정적 프로퍼티

- static 키워드 선언하여 인스턴스 생성없이 프로퍼티 사용

형태)

```
static let/var 프로퍼티명 = 값

static let/var 프로퍼티명 : 타입 {
        get{
                return 반환값
        }
        set{
        }
}
```

ex)

```
class Test {
        static let value = 10
        static var TestValue:Int {
            get{
                return 20
            }
        }
}

Test.value
Test.TestValue
```

메소드 - 인스턴스/타입 메소드

- 생성된 인스턴스의 함수 = 인스턴스 메소드
- static 키워드가 붙은 정적 메소드 = 타입 메소드 (인스턴스 생성 없이 메소드 사용)

```
class TestMethod {
        func test(){
            print("test")
        }
}

let testmethod = TestMethod()
testmethod.test()   //인스턴스 메소드 호출
```

```
class TestMethod {
        static func returnValue() -> Int {
            return 10
        }
}

TestMethod.returnValue()    //정적(타입) 메소드 호출
```

클래스 - 상속

- 상속은 부모와 자식의 관계로 부모 클래스에 대한 정보를 가질 수 있다.
- “ : “ (콜론)으로 상속을 표현하며, 상속할 부모클래스를 지정.
- 만약 부모의 함수를 다시 정의하면 **override**라는 키워드가 붙게 된다**(overriding)**

<형식>

```
class [클래스명] : [클래스명] {

}
```

예)

```
class A {
  var a:String =”a”
  func test(){}
}

class B : A {
  var b:String = “b”
  override func test(){ print(“test”) }
}

var b = B()
b.a             // A클래스의 a접근 => “a”
b.b             // B클래스의 b접근 => ”b”
b.test()        // “test” 출력
```

초기화(생성자)

- 클래스/구조체의 인스턴스 생성시 초기값을 설정하는 있는 문법적 장치
- init이라는 특별한 메소드명으로 생성
- self 키워드는 자기 자신 (클래스)를 가르킨다.

형태)

```
init(매개변수:타입, 매개변수:타입 ...){

}
```

```
class TestClass {
        var val1:Int? = 1
        var val2:Int? = 2
        init(){}
        init(val1:Int, val2:Int){
            self.val1 = val1
            self.val2 = val2
        }
}

let testclass1 = TestClass()
let testclass2 = TestClass(val1: 3, val2: 3)
```

소멸자

- ARC(Automatic reference counting)에 의해 인스턴스가 메모리에 할당해제될 때 실행됨.

형태)

```
deinit {
    // 소멸자에 대한 실행구문
}
```

Ex)

```
deinit {
    close(file)              //파일닫기를 해야할 경우
}
```

서브스크립트(subscript)

- 클래스/구조체에 첨자를 정의하기 위한 키워드
- subscript라는 특정이름으로 구현
- 첨자: num[index]와 같이 []사이에 인덱스를 넣어서 인스턴스 정보를 접근

형태)

```
subscript(매개변수:타입) -> 리턴타입 {
        get {
            return 반환값
        }
        set(매개변수) {
            //수행코드
        }
}
```

서브스크립트(subscript)

Ex)

```
struct PersonArray {
        var person = ["홍길동","김철수","박영희"]
        subscript(index: Int) -> String {
                get {
                     if index < person.count {
                       return person[index]
                     }else{
                       throw Error
                     }
                }
                set(value) {
                        person[index] = value
                }
        }
}
```

```
let personArray = PersonArray()

personArray[0]
//"홍길동 출력" (get)

personArray[1] = "김수흥"
//"index 1의 김철수를 김수흥으로 변경" (set)
```

Mutating 메소드에서의 self 할당(Assigning to self Within a Mutating Method)

Mutating 메소드는 암시적인 `self` 프로퍼티에 완전히 새로운 인스턴스를 할당할수 있다. 위에서 보여준 `Point` 예제는 다음과 같은 방법으로 작성 할 수 있다.

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

이 버전의 mutating `moveBy(x:y:)` 메소드는 `x` 와 `y` 값을 대상 위치로 설정하는 새로운 구조체를 생성한다. 이번 버전의 메소드는 이전 버전과 똑같은 결과가 될것이다.

Type Casting - type check

- is 라는 키워드로 type 확인
- 연산에 대한 return은 Bool (true / false)

형식)

[비교대상] is [비교타입]

=> 비교결과 (true / false)

```
let text = 123

if text is String {
    print("String형입니다.")
} else if text is Int {
    print("Int형입니다.")
} else{
    print("다른 타입입니다.")
}


// 결과는 "Int형입니다."
```

Type Casting - as 키워드

- as라는 키워드로 형변환 (상속 관련 부모-자식 관계타입)
- as뒤에 !, ?으로 옵셔널을 사용가능

형식)

[타입변환 대상] as [변환타입]

as! : 옵셔널 force unwrapping
as?: 옵셔널 타입

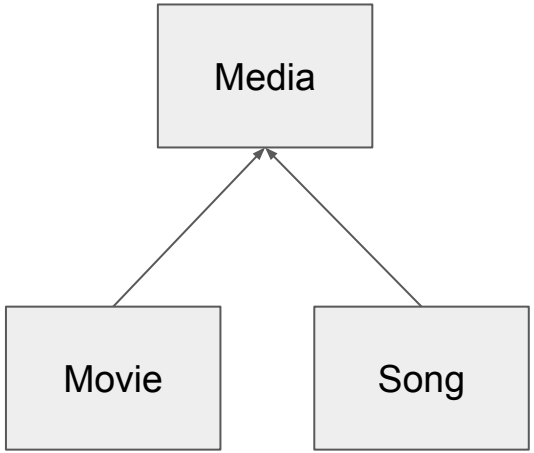
```
let text = 123 as Double
// text => Int -> Double

// super: Media, sub: Movie

let movie = media as Movie
// media => Media -> Movie
```

Type Casting - upcasting/downcasting

- as라는 키워드로 형변환 (상속 관련 부모-자식 관계타입)
- upcasting / downcasting 으로 구분



upcasting

```
let media = movie as Media
```

downcasting

```
let song = media as Song
```

Enum

- 열거형, 특정 주제에 관련 데이터를 멤버로 구성하기 위한 자료형 객체
- enum이라는 키워드로 선언하여 case별 값 카테고리 구분
- 스위프트 3버전부터 열겨형의 멤버를 소문자로 쓰는 정책변화

<형태>

```
enum 열거형 이름 {
   case 멤버값
   case 멤버값
   case 멤버값
   case 멤버값
}
```

ex)

```
enum NATION {
   case korea
   case america
   case japan
   case china
}
```

Enum

<특정값 적용>

```
enum NATION : String {
   case korea = “KR”
   case america = “EN”
   case japan = “JP”
   case china = “CN”
}
```

<사용>

```
NATION.korea
let nation:NATION = NATION.korea
let nation:NATION = .korea
```

<enum 특정값 사용>

```
NATION.korea.rawValue   // => return “KR”
```

<활용>

```
switch nation{
  case .korea:  print(nation.rawValue)
  case .america: print(nation.rawValue)
  case .japan: print(nation.rawValue)
  case .china: print(nation.rawValue)
}
```

Enum - 연관값 (associated values)

<연관값 타입 지정>

```
enum ImageFormat {
   case jpeg
   case png(Bool)
   case gif(Int, Bool)
}
```

<연관값 초기화>

```
var newImage = ImageFormat.png(true)
newImage = .gif(256,false)
```

<연관값 사용>

```
switch newImage {
    case .jpeg:
        print("jpeg")
    case .png(let bool):
        print("jpeg \(bool)")
    case .gif(let num, let bool):
        print("jpeg :\(num), \(bool)")
}
```

접근 제한자 (access cotrol)

- 클래스(class), 구조체(struct), 열거형(enum), 변수, 함수 등에 대한 접근제한
- swift 3.0 5가지 open, public, private, internal, fileprivate 제공

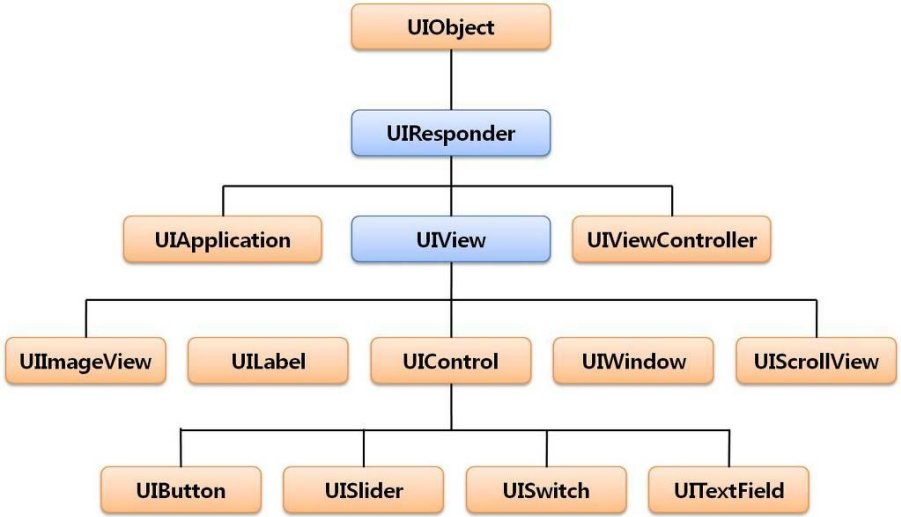
접근 제한자 종류	특징
open	외부모듈에 접근제한이 없으며, 오버라이딩이 가능
public	외부모듈에 접근제한이 없으나 외부모듈에 대한 오버라이딩은 할 수 없다
internal	해당 프로젝트에서만 접근가능 (멀티 프로젝트가 아닌 경우 **public** 동급)
private	자기 자신이나 상속받은 자식 내부에서만 접근가능
fileprivate	동일 파일 내에서는 액세스가 가능하게, 그 외에는 접근 할 수 없게 만들어준다.

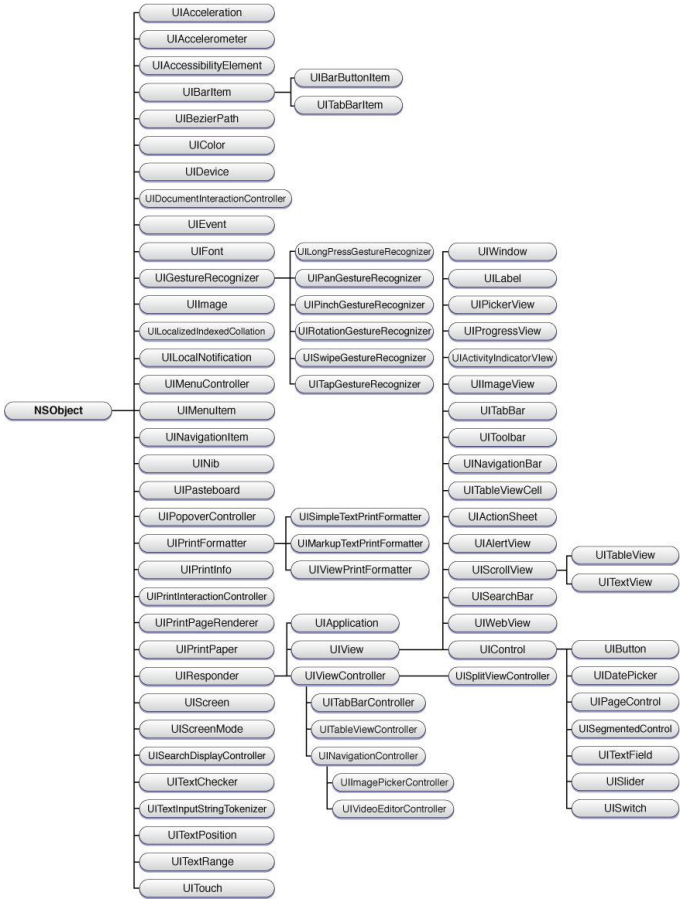
extension / final 키워드

extension	**생성된 클래스/구조체/enum에 대한 기능을 추가할 때 사용**	class A {
		  func test1(){}
		}
		
		extension A {
		  func test2(){}
		  var num:Int { return 1 }
		}
		
		A().test1() //기존 정의한 함수
		A().test2() //새로 추가된 함수
		A().num //새로 추가된 연산프로퍼티

final	**클래스의 프로퍼티의 overriding을 막음**	class A {
		  final func test1(){}
		}
		
		class B : A{
		  override func test1(){} //불가능
		}

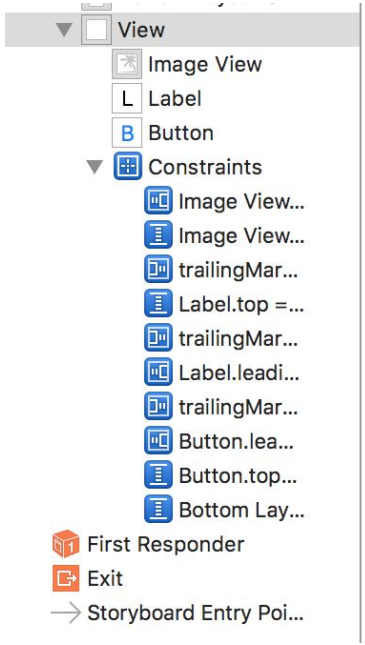
뷰 계층구조

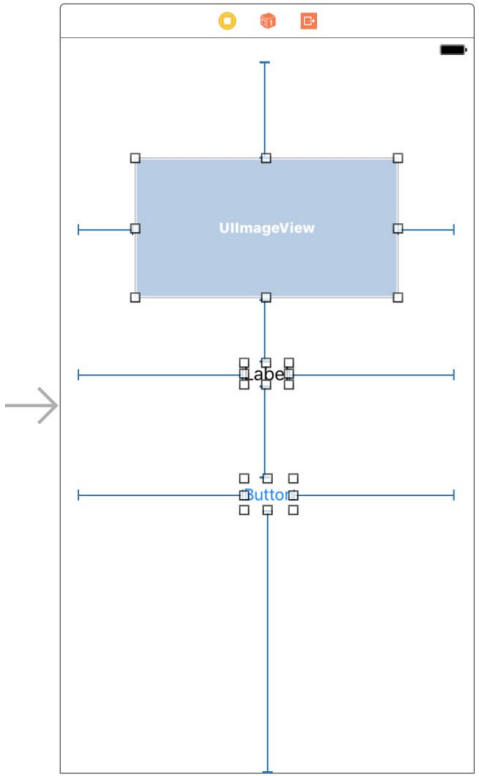

UIObject
UIResponder
UIApplication
UIView
UIViewController
UIImageView
UILabel
UIControl
UIWindow
UIScrollView
UIButton
UISlider
UISwitch
UITextField


NSObject
UIAcceleration
UIAccelerometer
UIAccessibilityElement
UIBarItem
UIBarButtonItem
UITabBarItem
UIBezierPath
UIColor
UIDevice
UIDocumentInteractionController
UIEvent
UIFont
UIGestureRecognizer
UILongPressGestureRecognizer
UIPanGestureRecognizer
UIPinchGestureRecognizer
UIRotationGestureRecognizer
UISwipeGestureRecognizer
UITapGestureRecognizer
UIImage
UILocalizedIndexedCollation
UILocalNotification
UIMenuController
UIMenuItem
UINavigationItem
UINib
UIPasteboard
UIPopoverController
UIPrintFormatter
UISimpleTextPrintFormatter
UIMarkupTextPrintFormatter
UIViewPrintFormatter
UIPrintInfo
UIPrintInteractionController
UIPrintPageRenderer
UIPrintPaper
UIResponder
UIScreen
UIScreenMode
UISearchDisplayController
UITextChecker
UITextInputStringTokenizer
UITextPosition
UITextRange
UITouch
UIApplication
UIView
UIViewController
UITabBarController
UITableViewController
UINavigationController
UIImagePickerController
UIVideoEditorController
UIWindow
UILabel
UIPickerView
UIProgressView
UIActivityIndicatorView
UIImageView
UITabBar
UIToolbar
UINavigationBar
UITableViewCell
UIActionSheet
UIAlertView
UIScrollView
UITableView
UITextView
UISearchBar
UIWebView
UIControl
UIButton
UIDatePicker
UIPageControl
UISegmentedControl
UITextField
UISlider
UISwitch
UISplitViewController

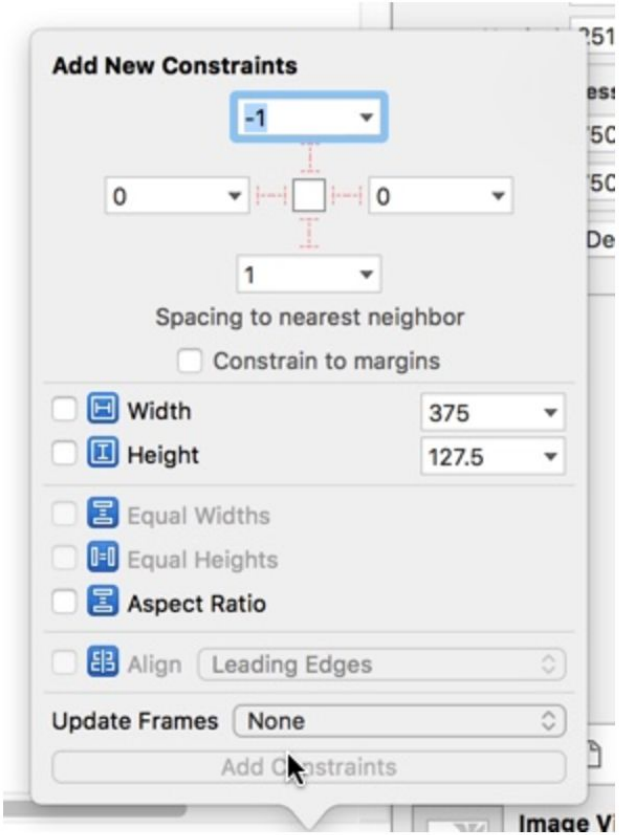
AutoLayout(오토 레이아웃)

- 뷰와 뷰 간의 위치/여백으로 크기를 유동적으로 맞춰주는 기능
- constraints이라는 제한조건 값에 의해 위치/너비 설정



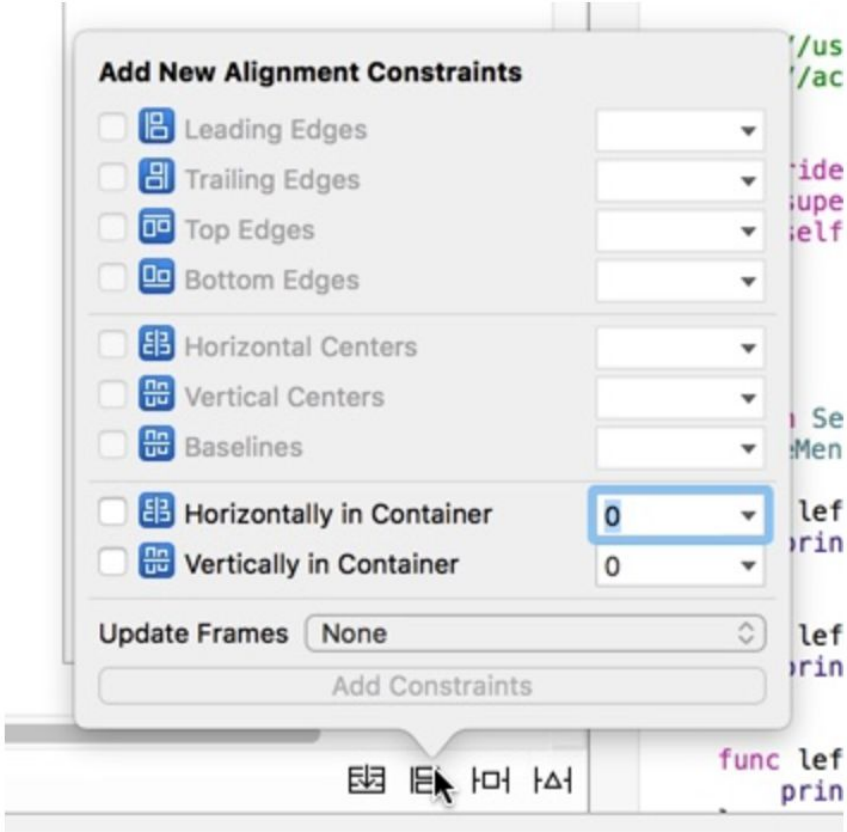


AutoLayout(오토 레이아웃)



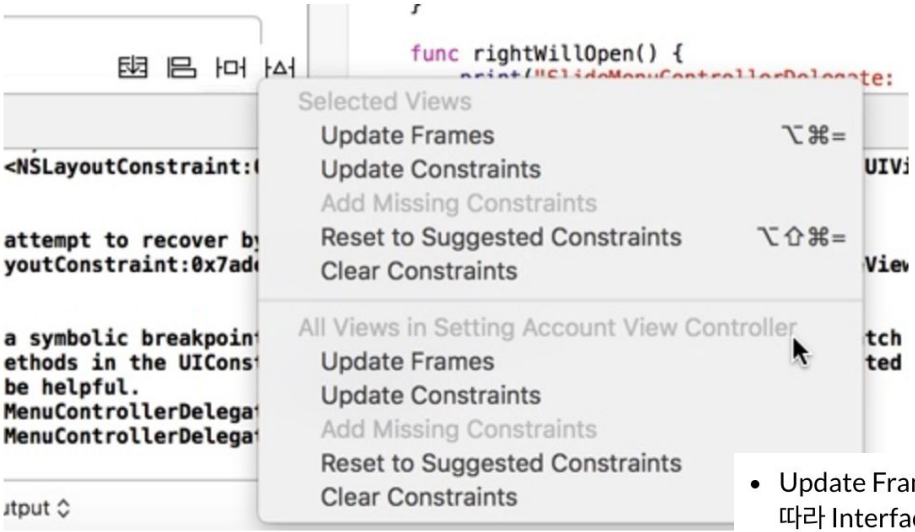
- Leading: 요소왼쪽(첫글자 방향)
- Trailing: 요소 오른쪽 (문자 끝 방향)
- Top: 요소 상단
- Bottom: 요소 하단
- Width: 요소 너비
- Height: 요소 높이
- centerX: 요소 가로방향 중심위치
- cetnerY: 요소 세로방향 중심위치
- Baseline: 문자 baseline
- Horizontal: 수평
- Verical: 수직
- Aspect Ratio: 요소 가로세로 비율

AutoLayout(오토 레이아웃)



- Leading Edges: 여러 요소의 왼쪽에 맞춤
- Trailing Edges: 여러 요소의 오른쪽 가장자리에 맞춤
- Top Edges: 여러 요소 상단에 맞춤
- Bottoms Edges: 여러 요소의 하단에 맞춤
- Holizontal Centers: 여러 요소의 수평방향 중심위치에 맞춤
- Vertical Centers: 여러 요소의 수직 방향의 중심위치에 맞품
- Baselines: 여러 레티블등의 텍스트 아래를 맞춤
- Horizontal Center in Contrainer: 표시 영역의 수평방향 중심에 맞춤[
- Vertical Center is Container: 표시 영역의 수직방향 중심에 맞춤

AutoLayout(오토 레이아웃)



- Update Frames: Constraints 및 Interface Builder에서 위치가 어긋나 있는 경우 제약에 따라 Interface Builder에서 위치를 이동한다.
- Update Constraints: Constraints 및 Interface Builder에서 위치가 어긋나 있을 때 Interface Builder에서의 위치에 따라 제약을 변경한다.
- Add Missing Constraints: Constraints가 부족한 경우 보족한 제약을 자동으로 추가한다.
- Reset to Suggested Constrains: 설정되어 있는 Constraints를 삭제하고 알맞다고 판단되는 제약을 자동 추가한다.
- Clear Constraints: 지정된 제약조건을 삭제한다.