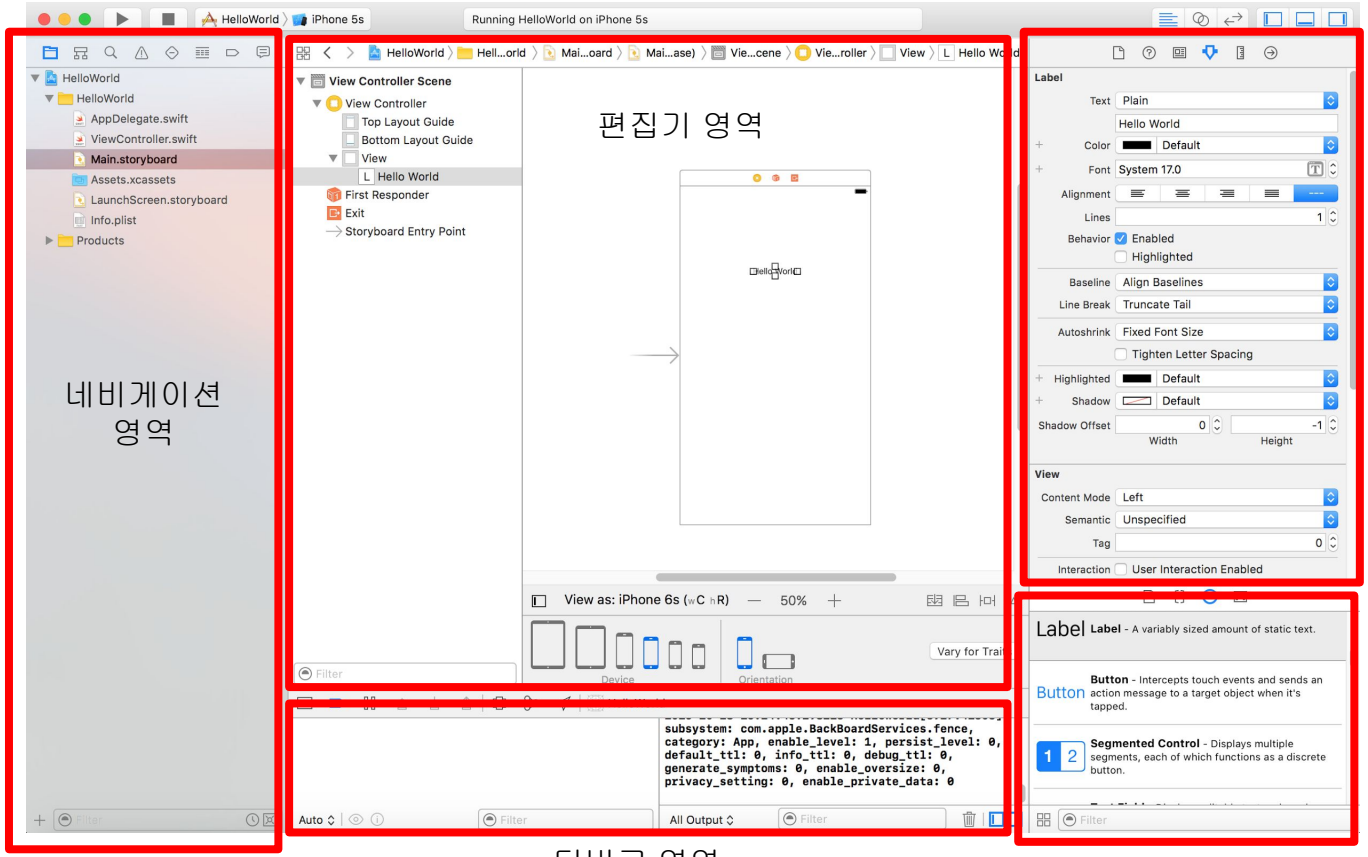
Swift Study Review

2017. 02.11

Xcode



Swift 문법

- 자료형, 연산자, 변수, 옵셔널, 조건문, 반복문
- 클래스, 함수, 접근제한자, 프로토콜, enum, 클로저
- 컬렉션 (array, dictionary), 델리게이트 패턴
- type check/casting, selector
- error handle(exception)

ios - cocoa touch framework

- UIPickerView, UIAlertController, UIWebView
- MKMapView, pageControl, Segue
- UIGestureRecognizer, Core Graphics
- AVFoundation (AVAudio, AVPlayer)
- UIImagePickerController

swift 문법

Swift - 기본 자료형

타입	특징	예
Int, Int8, Int16, Int32, Int64	작은 수 또는 큰 수의 음수, 양수값	**4, 523, -45565, 5342, -28, 54, 234**
UInt, UInt8 ,UInt16, UInt32, UInt64	작은 수 또는 큰 수의 양수값	**5, 123, 3432432, 52, 34, 5, 123**
Float, Double	부동 소수점, 분수의 음수, 양수값	**11.453, 234.23, -123.34, 2.231231123, 0.012345**
Character	단일 문자 (큰따옴표로 묶어서 표현)	**“T”, “한”, “H”, “*”, “3”**
String	문자열 데이터 (큰따옴표로 묶어서 표현)	**“Filasa”, “문장입니다.”, “New York”**
Bool	참/거짓을 표현하는 논리데이터 표현	**true, false**

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

***값이 없다는 표현으로 swift에서는 nil 이라고 표현함.**

Swift - 변수

***변수: 프로그래밍 언어에서 일반적으로 어떤 공간에 데이터 값을 담고 사용하는 공간 또는 그릇**

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 영원히 그 값 유지

<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 - 옵셔널

*데이터의 유무를 판별하기 위한 문법적 장치, ! 와 ?를 데이터자료형과 변수 뒤에 명시

* ?를 붙이면 실제데이터는 **Optional(데이터)** 형식으로 데이터형을 감싸게 표현됨
  - 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 exception**에 대한 문법적 안전장치로 **swift**에 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
```

Swift - 조건문 if / else 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
}
```

Swift - 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

- switch의 비교대상에 따라 case별 실행문을 분기하는 조건문

<형식>

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

예)

```
var char:Character = “B”
swift char {
 case “A”: print(“A”)
 case “B”: prunt(“B”)
 default: print(“A나 B가 아니네”)
}
```

예)

```
var a:Int = 1
swift a {
 case 1: print(“1”)
 case 2: prunt(“2”)
 default: print(“1이나 2가 아니네”)
}

var b:Int = 7
swift b {
 case 0...3: print(“0~3 범위”)
 case 4...6: prunt(“4~6범위”)
 default: print(“7이상 범위")
}
```

*다른언어에는 **break**라는 제어전달문을 사용하지만 **swift**에서는 제공하지 않으며, 단일 **case**를 실행

Swift - 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
```

함수 - func (function)

- 특정 기능을 하는 코드를 특정 방법으로 묶어낸 것, 다른 의미에서는 반환형이 있는 프로시져
- 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) {  // _ 붙일경우
  print(“\(str)”)
}
printStr(“test111”)  //변수명 명시 안해도 됨
```

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

  init(){        //생성자

  }

  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
```

* 프로퍼티는 부가적으로 **@lazy, getter/setter** 등 기타옵션 제공
* 메소드를 통해 프로퍼티 변경이 필요한 경우 메소드 앞에 **mutating** 키워드를 사용 **(권장하지 않음)**

클래스 - 상속

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

<형식>

```
class [클래스명] : [클래스명] {

}

class [클래스명] : [클래스명1], [클래스명2] {

}
```

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
```

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 **class**에 대한 기능을 추가할 때 사용	class A {
		  func test1(){}
		}
		
		extension A {
		  func test2(){}
		}
		
		A().test1() //기존 정의한 함수
		A().test2() //새로 추가된 함수

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

프로토콜 (protocol)

- 구현체 없는 함수/메소드 (함수이름과 매개변수/반환형만 정의된 형태)
- 메소드에 대한 설계/명세의 목적으로 사용되는 문법
- 다른 언어에서는 종종 인터페이스(interface)라는 용어 많이 사용
- protocol이라는 키워드를 쓰며, 클래스 / 구조체 / 열거형 뒤에 " : " 으로 프로토콜 채택 (adopt)

<형식>

```
protocol 프로토콜명 {
    구현해야 할 메소드 정의1
    구현해야 할 메소드 정의2
}

class/struct/enum 객체명 : 프로토콜명 {

}
```

예제)

```
protocol TestProtocol {
    func stringTest() -> String
    func addOne(num: Int) -> Int
}

class/struct/enum Test : TestProtocol {
    func printTest() -> String {
        return "test"
    }

    func addOne(num: Int) -> Int {
        return num+1
    }
}
```

Enum

- 열거형, 특정 주제에 관련 데이터를 멤버로 구성하기 위한 자료형 객체
- 남/여, 국가, 지역 등 구분 지어지는 데이터를 분류할 용도
- enum이라는 키워드로 선언하여 case별 값 카테고리 구분

<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())
}
```

익명함수 / 클로저 (closure)

- 이름이 없는 함수 (매개변수와 반환형, 실제 구현체만 정의)

<형식>

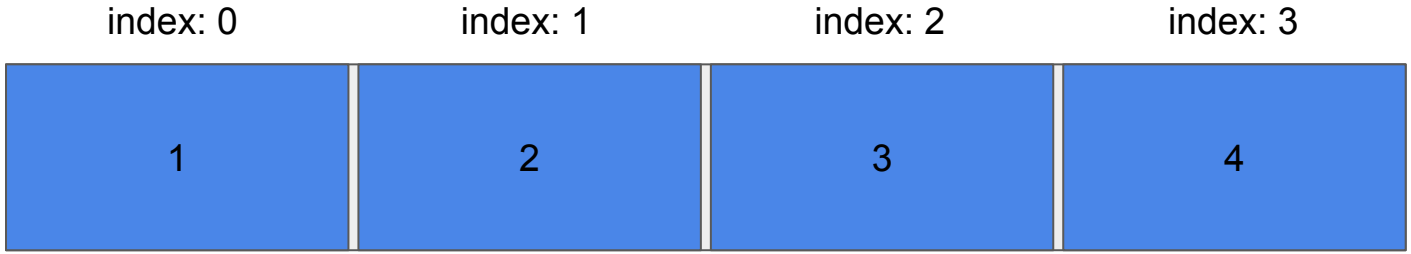
```
// 기본형
{
   (매개변수:타입) -> (반환형) in
      작성할 내용
}

// 반환형 없을시 생략
{
  (매개변수:타입) in
      작성할 내용
}

// 매개변수 타입 생략
{
   매개변수 in
      작성할 내용
}
```

예제)

```
{
   (str:String) -> (String) in
        return “test”
}

{
   (str:String) in
        print(“test”)
}

{
   action in
        self.lampImg.image = self.img
        self.isLampOn=false
}
```

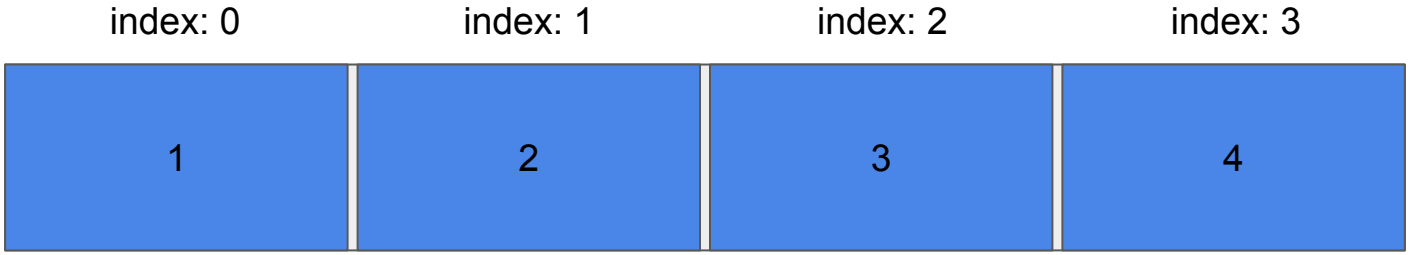
컬렉션(Collection) - 배열(Array)

- 컬렉션(Collection)은 컴퓨터 이론에서 쓰는 자료구조 형태 지정하는 용어
- Swfit에서 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:value라는 쌍데이터를 이루어져있으며, 적용할 타입제한이 없다.(단, 일치된 타입 사용)
- key의 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>

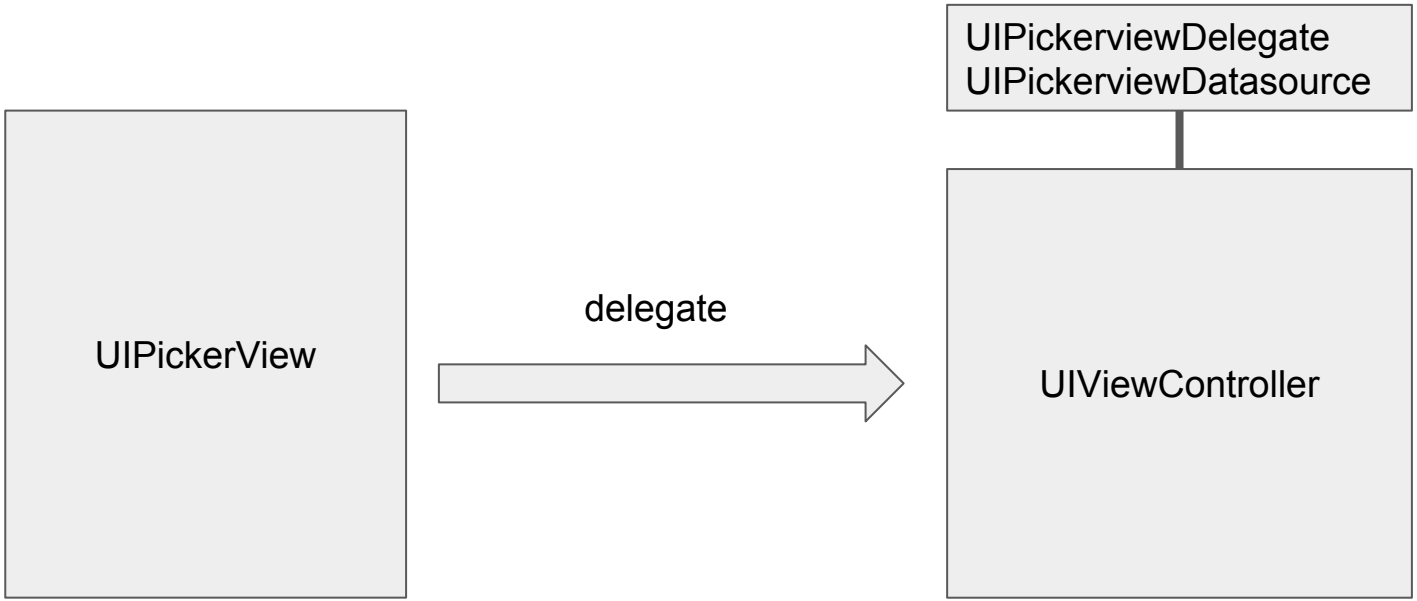
```
let dictionary = [“test2”:”bbb”, ”test1”:”aaa”]            // [String:String]()

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

⇒ 출력

```
test1 bbb
test2 bbb
```

델리게이트 패턴 (delegate pattern)

- 특정 대상에게 자신의 임무/기능을 대신 맡기겠금 하는 방식 (대리자 위임)
- cocoa framework의 근간이 되는 이벤트 및 다양한 기능들이 델리게이트 패턴으로 구현되어 있음
- protocol 타입의 delegate / datasource 을 구현하여 대리자가 대신 기능을 구현



델리게이트 패턴 (delegate pattern)

기능을 갖는 클래스

```
protocol UIActionDelegate {
    func click(uiAction: UIAction)
}

class UIAction {
    var delegate: UIActionDelegate?

    func clickEvent() {
        self.delegate?.click(self)
    }
}
```

대신 기능을 구현하는 대리자 클래스

```
class Test: UIActionDelegate {
    let uiAction = UIAction()

    init() {
        self.uiAction.delegate = self
        self.uiAction.clickEvent()
    }

    func click(uiAction: UIAction) {
        //해당 메소드에 대한 기능 구현....
    }
}
```

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)

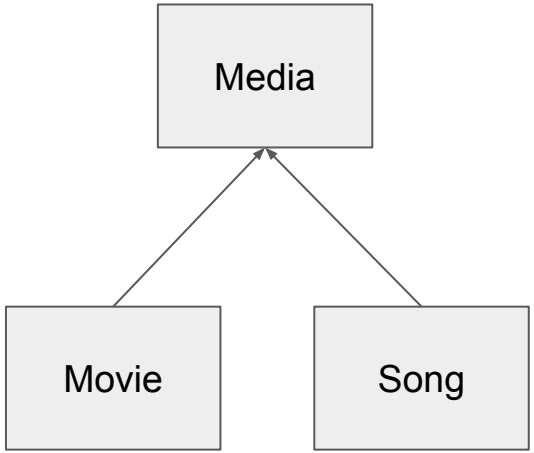
[타입변환 대상] as [변환타입]

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
```

Selector

- Objective-C는 C의 함수 포인터와 유사한 개념으로 'SEL' 이라는 데이터 타입을 지원
- @selector 지시어와 임의의 메소드 이름을 사용하여 값을 설정하여 특정함수를 가르키게 함.
- Swift 3.0부터는 #selector(함수) 형태로 선언하여 해당함수를 가르킴

```
//생성할 클래스 Test
class Test {
    public init(target: Any?, action: Selector?)
}
```

```
//특정 클래스에 정의된 함수
func add(_ number:int){
    //덧셈연산
}

//Test 클래스에게 동작에 필요한 add()함수 지정
Test(target: self, action: #selector(self.add(_:)))
```

Error handle - Error protocol

- 특정 조건에 대한 에러 제어 흐름을 제어하기 위한 문법적 장치
- 언어에서는 보통 exception (예외처리) 용어로 많이 사용
- Error라는 protocol타입을 구현한 enum타입에 에러를 정의
- Error protocol은 의미없는 빈 프로토콜로 표시의 의미가 강함.

```
public protocol Error {
}
```

<형태>

```
enum 에러명 : Error {
   case 에러함수명
   case 에러함수명(매개변수)
}
```

<형태>

```
enum IntegerParseError : Error{
   case nilNotParsing
   case characterNotParsing(char:Character)
}
```

Error handle - throws / throw

- throws 키워드를 통해 에러 예외처리 호출정의
- throw로 작성한 에러를 호출 (에러를 던지다는 표현함)

```
<형태>
func 함수명(매개변수) throws -> 리턴형 {

    if 조건문 {
        //조건 실행
    } else {
        throw 에러명.에러함수
    }
    ...
}
```

```
ex)
func numCheck(value:Any?) throws ->
Int {
    if let num = value as? Int {
        return num
    } else {
        throw NumCheckError.notNum
    }
  }
```

Error handle - do / try ~ catch

- throw로 던져지는 error를 호출받아 분기시키는 구문
- try로 throws에 대한 결과를 처리하여 실패시 catch로 에러처리

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

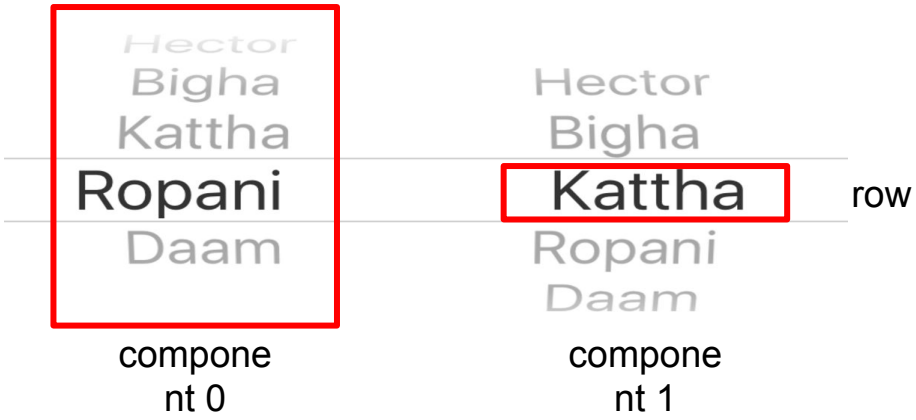
ex)

```
func testNum(_ value:Any){
        do {
            let num = try numCheck(value)
            print(num)
         } catch {
            print((error as! NumCheckError).description)
         }
    }
```

ios - cocoa touch framework

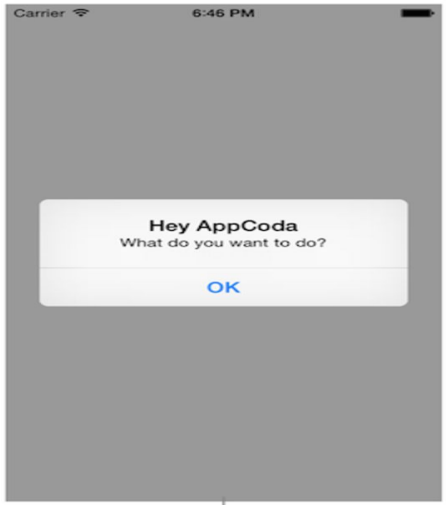
UIPickerView / UIDatePicker

- UIView 하위 자식인 UIPickerView 클래스 (UIDatePicker는 별도의 기능 클래스)
- 피커뷰에 대한 설정을 delegate protocol로 PickerViewDatasource / PickerViewDelegate 제공
- 피커뷰의 열에 대한 정보를 Component 지칭, index번호는 0부터 시작

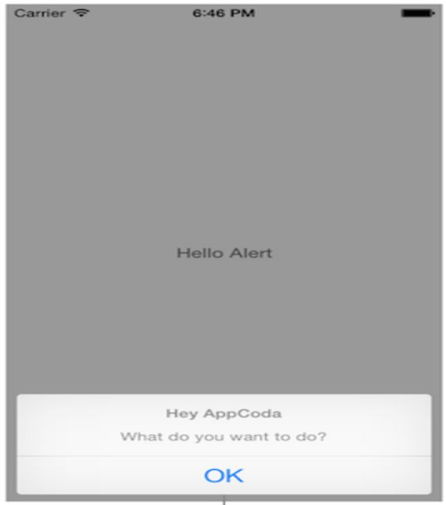


UIAlertController

- 팝업 화면창을 띄우기 위한 UI 라이브러리
- 기존 ios 8이전 버전까지는 UIActionSheet와 UIAlertView 구분되다 UIAlertController 통합됨



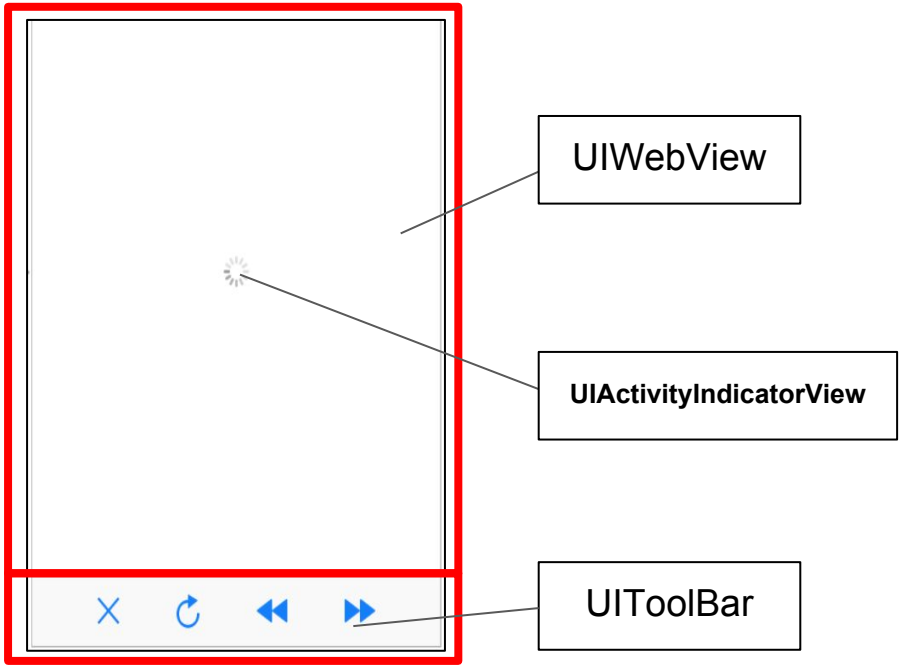
Alert



ActionSheet

UIWebView

- **html web브라우저를 지원하는 뷰**
- **UIView 하위 클래스**



Mapkit - MKMapView

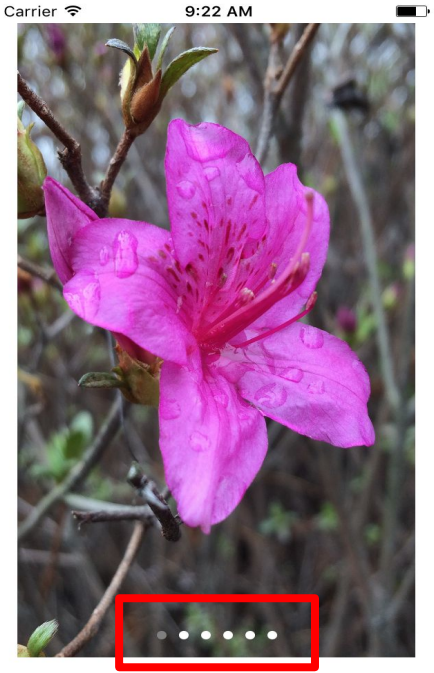
- 지도정보를 위한 **framework**
- 다양한 클래스 중 **MKMapView**를 이용하여 **MapView** 사용
- 기본클래스에서는 **UIKit**만 **import**되어있기 때문에 별로도 **import MapKit** 선언 필요

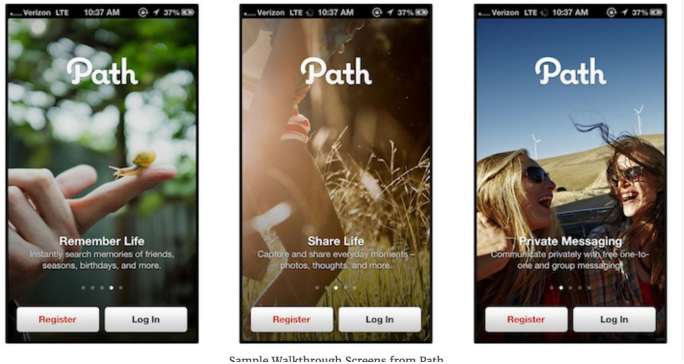
```
import Foundation
import MapKit.MKAnnotation
import MapKit.MKAnnotationView
import MapKit.MKCircle
import MapKit.MKCircleRenderer
import MapKit.MKCircleView
import MapKit.MKDirections
import MapKit.MKDirectionsRequest
import MapKit.MKDirectionsResponse
import MapKit.MKDirectionsTypes
import MapKit.MKDistanceFormatter
import MapKit.MKFoundation
import MapKit.MKGeodesicPolyline
import MapKit.MKGeometry
import MapKit.MKLocalSearch
import MapKit.MKLocalSearchCompleter
import MapKit.MKLocalSearchRequest
import MapKit.MKLocalSearchResponse
import MapKit.MKMapCamera
import MapKit.MKMapItem
import MapKit.MKMapSnapshot
import MapKit.MKMapSnapshotOptions
import MapKit.MKMapSnapshotter
import MapKit.MKMapView
import MapKit.MKMultiPoint
import MapKit.MKOverlay
import MapKit.MKOverlayPathRenderer
import MapKit.MKOverlayPathView
import MapKit.MKOverlayRenderer
import MapKit.MKOverlayView
import MapKit.MKPinAnnotationView
import MapKit.MKPlacemark
import MapKit.MKPointAnnotation
import MapKit.MKPolygon
import MapKit.MKPolygonRenderer
import MapKit.MKPolygonView
import MapKit.MKPolyline
import MapKit.MKPolylineRenderer
import MapKit.MKPolylineView
import MapKit.MKReverseGeocoder
import MapKit.MKTileOverlay
import MapKit.MKTileOverlayRenderer
import MapKit.MKTypes
import MapKit.MKUserLocation
import MapKit.MKUserTrackingBarButtonItem
import MapKit.NSUserActivity_MKMapItem
import MapKit
```

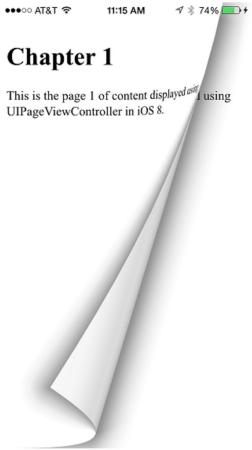
MKMapView

UIPageControl / UIPageViewController

- 화면내 **page** 표시/전환기능 제공하는 **UIPageControl**와 **UIPageViewController (ios5)**
- 변화에 대한 **pageControl** 지정



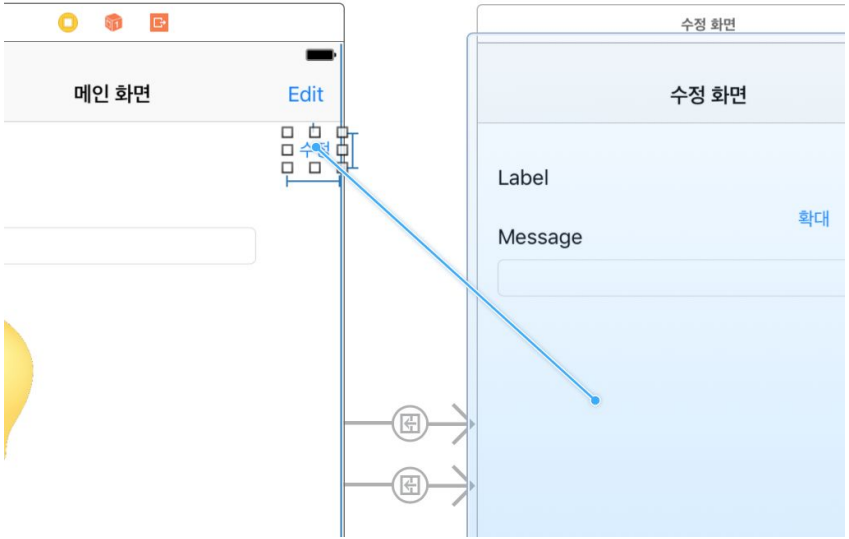


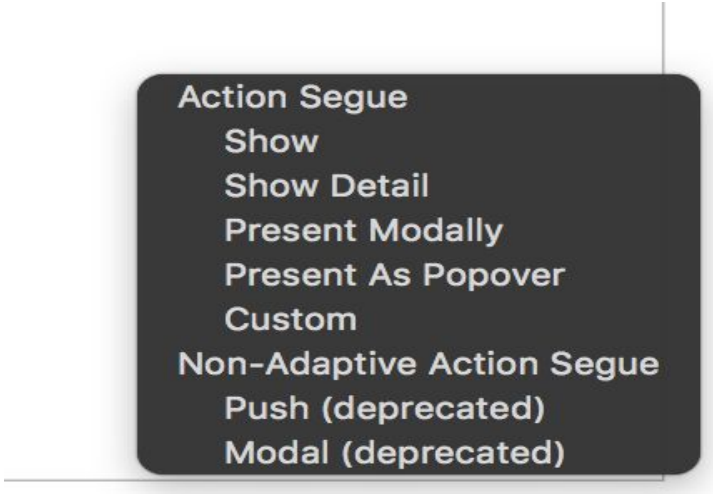


UIPageViewController

Segue

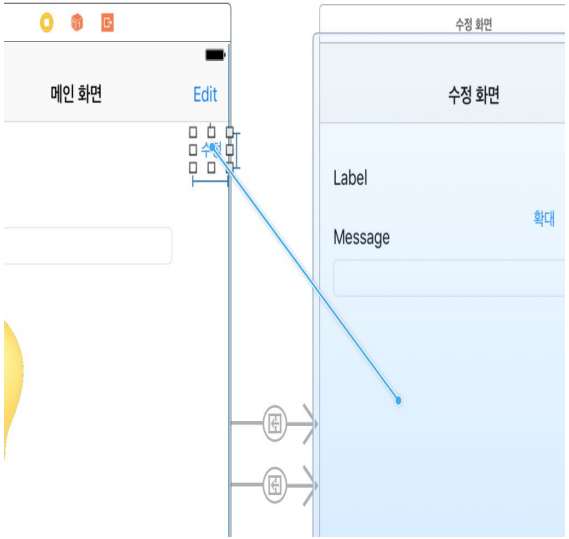
- 화면전환에 대한 식별자 (안드로이드: intent와 유사)
- 화면전환에는 직접 UI에 대한 presentController를 호출하거나 Segue 식별자로 전환방식
- IB(interface builder)에는 segue에 대한 화면전환 action 제공
- UIController 또는 UIControl를 상속받는 대상만 사용가능

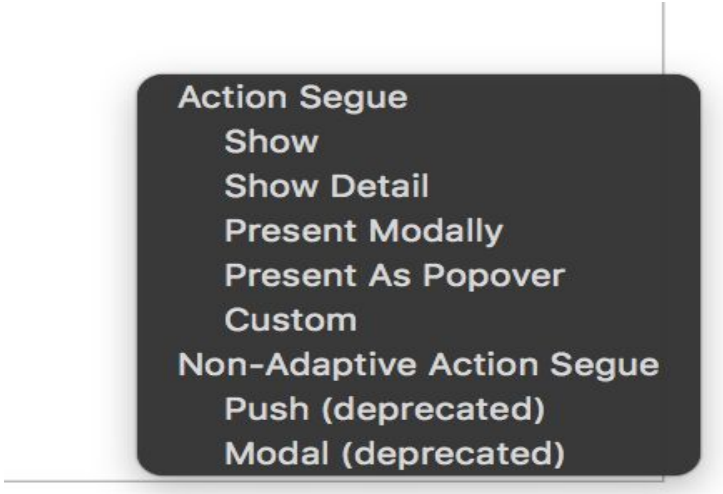




Segue - 단순 방식

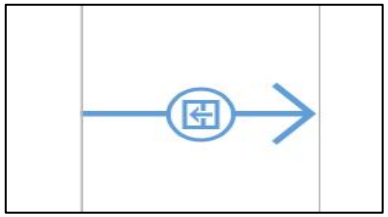
- UIControl를 상속받는 UI의 segue action 추가
- UIViewController 또는 UIControl를 상속받는 UI요소만 segue 설정가능

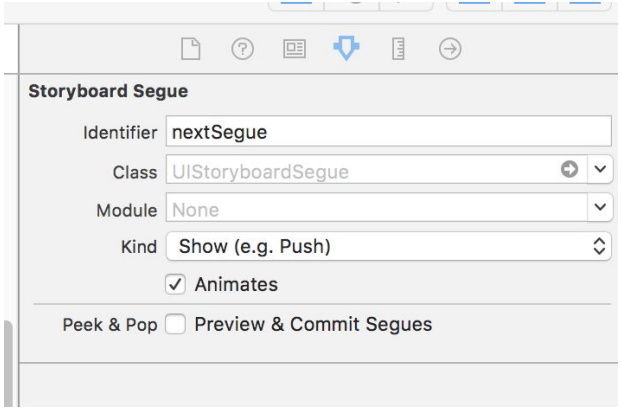




Segue - 제어방법

- segue에 identifier 설정 후 ViewController에서 segue 관련 code정의





func **performSegue**(**withIdentifier** identifier: String, sender: Any?) **//생성된 특정 segue**

func **prepare**(**for segue**: UIStoryboardSegue, sender: Any?) **//segue가 실행전에 초기화 또는 특정기능 설정**

UIViewController 자신.**performSegue**(withIdentifier: "세그 identifier id", sender: 이벤트대상) **//특정이벤트에서 segue 설정**

Segue - 제어방법

```
func prepare(for segue: UIStoryboardSegue, sender: Any?)    //segue가 실행전에 초기화 또는 특정기능 설정
```

ViewController.class

```
func prepare(for segue: UIStoryboardSegue, sender: Any?) {

        let editViewController = segue.destination as! EditViewController

        if segue.identifier == "nextSegue" {
            editViewController.textWayValue = "segue : use button"
        }
}
```

Segue - 프로그래밍 방식

- 이벤트 함수에 직접 불러온 ViewController 지정 후 presentController 함수호출

```
@IBAction func segueSend(_ sender: UIButton) {
        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        let view2 = storyboard.instantiateViewController(withIdentifier: "ViewController2") as! ViewController2
        present(view2, animated: true, completion: nil)
}
```

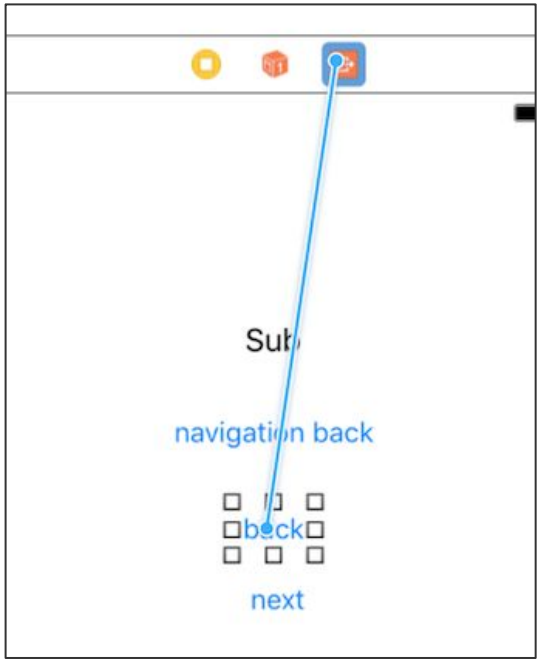
Segue - dismiss / unwind

- segue간의 연결고리가 되어 있어 dismiss()나 unwind기능으로 이전화면 전환
- UIViewController 에 dismiss()함수 내장
- navigationView를 상속받는 화면은 popViewController 사용

```
self.dismiss(animated: true, completion: nil)

self.presentingViewController?.dismiss(animated: true, completion: nil)

self.navigationController?.popViewController(animated: true)
```

Segue - dismiss / unwind

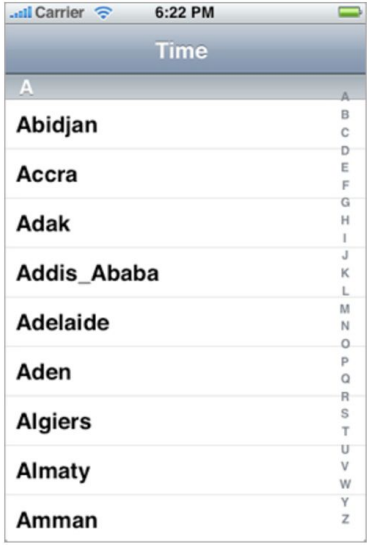
- 특정함수를 정의 후 exit 상단버튼에 unwind 바인딩
- 함수정의시 인자타입이 UIStoryboardSegue형 (segue == UIStoryboardSegue 타입)



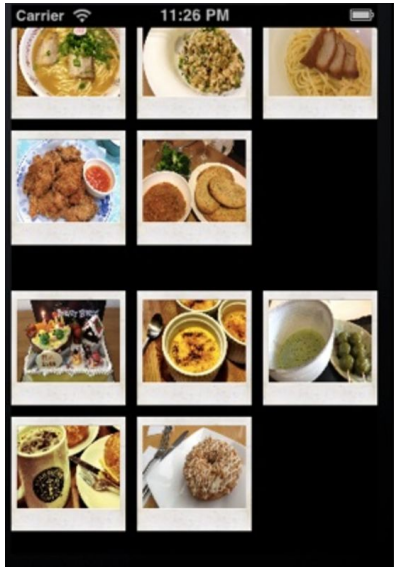

```
*/
@IBAction func backAction(_ sender: UIStoryboardSegue) {
}
```

UITableView

- 하나의 열을 갖는 목록 View (각 열은 UITableViewCell로 구성)
- 유동적인 열/셀을 목록 - UICollectionView



UITableView



UICollectionView

UIGestureRecognizer

- UIView의 제스처(동작행위)에 대해 이벤트리스너 클래스 (recognizer)
- 제스처의 swipe(방향지시) / pan(drag) / tab / rotate 등 다양한 서브클래스 존재

Figure 1-1 A gesture recognizer attached to a view

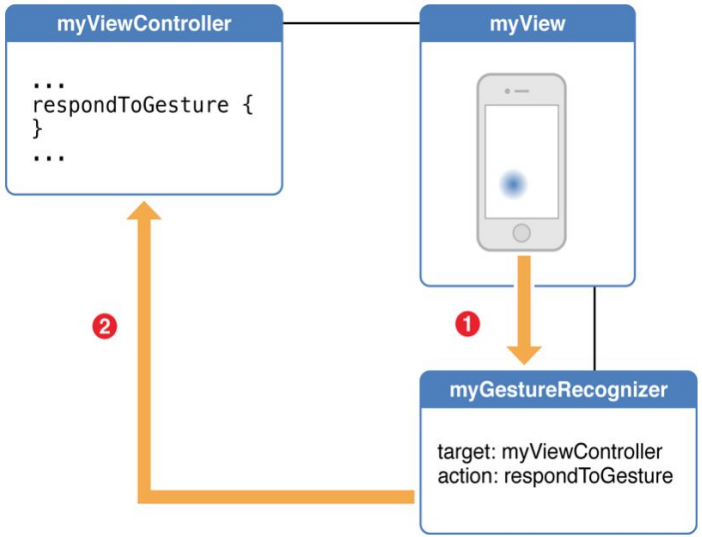


Table 1-1 Gesture recognizer classes of the UIKit framework

Gesture	UIKit class
Tapping (any number of taps)	`UITapGestureRecognizer`
Pinching in and out (for zooming a view)	`UIPinchGestureRecognizer`
Panning or dragging	`UIPanGestureRecognizer`
Swiping (in any direction)	`UISwipeGestureRecognizer`
Rotating (fingers moving in opposite directions)	`UIRotationGestureRecognizer`
Long press (also known as “touch and hold”)	`UILongPressGestureRecognizer`

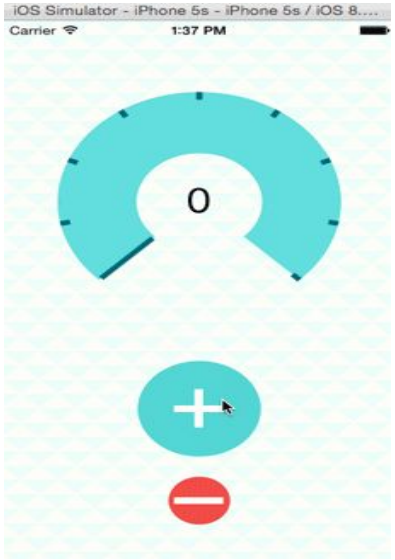
UIGestureRecognizer

The concrete subclasses of `UIGestureRecognizer` are the following:

- `UITapGestureRecognizer` 터치에 대한 제스처 감지기
- `UIPinchGestureRecognizer` 확대/축소에 대한 제스처 감지기(주로 두손가락 오므리고/펴기 제스처)
- `UIRotationGestureRecognizer` 회전(각도)에 대한 제스처 감지기(주로 두손가락으로 돌리는 제스처)
- `UISwipeGestureRecognizer` 방향에 대한 제스처 감지기(슬라이식 손가락 제스처)
- `UIPanGestureRecognizer` 특정 이동좌표에 대한 제스처 감지기
- `UIScreenEdgePanGestureRecognizer` 화면 가장자리에 대한 제스처 감지기
- `UILongPressGestureRecognizer` 긴 터치에 대한 제스처 감지기

Core Graphics

- 쿼츠 (Quartz) 기술의 힘을 이용하여 고해상도 출력으로 가벼운 2D 렌더링
- 경로 기반 드로잉, 앤티 앨리어싱 렌더링, 그라디언트, 이미지, 색상 관리, PDF 문서 등 이용



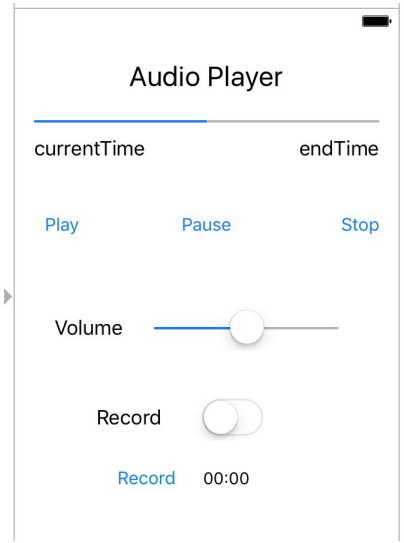
CGContext

- 코어 그래픽스(Quartz 2D)에서 그림을 그리는 영역을 가르키는 객체
- 윈도우, 비트 맵 이미지, PDF 문서 등 페이지의 대상을 대상에 렌더링하는 데 필요한 그리기 매개 변수와 모든 장치 관련 정보가 들어 있음.

```
UIGraphicsBeginImageContext(imageView.frame.size)  //CGContext 시작
let context = UIGraphicsGetCurrentContext()    //CGContext 얻기
context?.setLineWidth(2.0)      //선의 굵기 설정
context?.setStrokeColor(UIColor.red.cgColor)        //그려지는 색
context?.move(to: CGPoint(x: 50, y: 50))        //기준점 좌표이동 (기본 0,0)
context?.addLine(to: CGPoint(x: 250, y: 250))        //선 추가
context?.strokePath()      //설정한 선 그리기
imageView.image = UIGraphicsGetImageFromCurrentImageContext()  //Context 적용
UIGraphicsEndImageContext()        //CGContext 끝
```

AVFoundation

- 영상 및 미디어에 대한 기능을 제공하는 라이브러리
- 음성(AVAudioPlayer), 녹음(AVAudioRecorder), 동영상(AVPlayer)



AVAudioPlayer/AVAudioRecorder

AVPlayer

UIImagePickerController

- 사진/동영상 기능을 제공하는 사용자 인터페이스
- UIImagePickerControllerSourceType로 모드전용

`case photoLibrary`

장치의 사진 라이브러리를 이미지 선택 컨트롤러의 원본으로 지정합니다.

`case camera`

장치의 내장 카메라를 이미지 선택 컨트롤러의 소스로 지정합니다. 사용하여 (사용 가능한 등의 전면 또는 후면) 당신이 원하는 특정 카메라 나타내는 `cameraDevice`속성입니다.

`case savedPhotosAlbum`

장치의 카메라 롤 앨범을 이미지 선택 컨트롤러의 소스로 지정합니다. 장치에 카메라가 없으면 저장된 사진 앨범을 소스로 지정합니다.