Swift Study 16

2016. 03.18

Swift 문법

- 접근제한자 / final
- 프로토콜(protocol)
- 델리게이트 패턴 (delegate pattern)
- Error Handle (에러처리/예외처리)
- 제네릭(Generic)

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, 외부모듈에 대한 상속, 메소드 오버라이딩 가능
public	외부모듈에 접근제한이 없으며, 외부모듈에 대한 상속, 메소드 오버라이딩 불가능
internal	해당 모듈만 접근가능 (멀티 모듈이 아닌 경우 **public** 동급)
fileprivate	동일 파일 내에서만 접근가능
private	블록({ }) 안에서도 제한적으로 접근가능

***final** : 특정 프로퍼티 / 메소드의 **overriding**을 막는 키워드

프로토콜 (protocol)

- 구현체 없는 함수/메소드 (함수이름과 매개변수/반환형만 정의된 형태)
- 메소드에 대한 설계/명세의 목적으로 사용되는 문법 (프로퍼티/초기화 정의가능)
- type으로 취급되기 때문에 변수 대입, return값, 매개변수 등 사용가능
- 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)

```
let test:TestProtocol = TestProtocol()

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
    }
}
```

프로토콜 - 프로퍼티 정의

- 프로토콜에서 프로퍼티 정의는 get set 키워드로 읽기전용 / 읽기,쓰기 전용 구분
- get set 경우는 상수 프로퍼티나 읽기전용 연산 프로퍼티를 정의할 수 없음
- get 경우 모든 경우의 프로퍼티 정의 가능

<형식>

```
protocol 프로토콜명 {
    var 프로퍼티명:타입 { get set }
    var 프로퍼티명:타입 { get }
    static var 프로퍼티명:타입 {get set}
}

class 객체명 : 프로토콜명 {
    var 프로퍼티:타입

    var 프로퍼티: 타입 {
        get {   return 값   }
        set(매개변수) {  //실행구문 }
    }
}
```

예제)

```
protocol TestProtocol {
    var name1:String {get set}
    var description:String {get}
    static var num:Int {get set}
}

class Test : TestProtocol {
    var name1:String
    var description:String {
        get { return "test" }
        set(value) { self.name1 }
    }
    static var num: Int = 1
}
```

프로토콜 - 메소드 정의

- 메소드는 구현체 없는 몸체만 정의
- 내부프로퍼티 참조하는 mutating 키워드 / 인스턴스 없이 사용가능한 정적(타입) 메소드 정의가능
- 클래스 경우에는 mutating 생략가능 / static키워드 경우 class 키워드로 대체가능

```
protocol Test {
    func testfunc(str:String) -> String                  //general method
    mutating func testfunc2(str:String) -> String   //inner property modify - mutating
    static func testfunc3(str:String) -> String        //static method
}

class Test1 : Test {
        func testfunc(str: String) -> String {  return str }
        func testfunc2(str: String) -> String { self.name1 = "test1234"; return str; }
        static func testfunc3(str: String) -> String {  return str+" static" }
        //class func testfunc3(str: String) -> String {  return str+" static" }
}
```

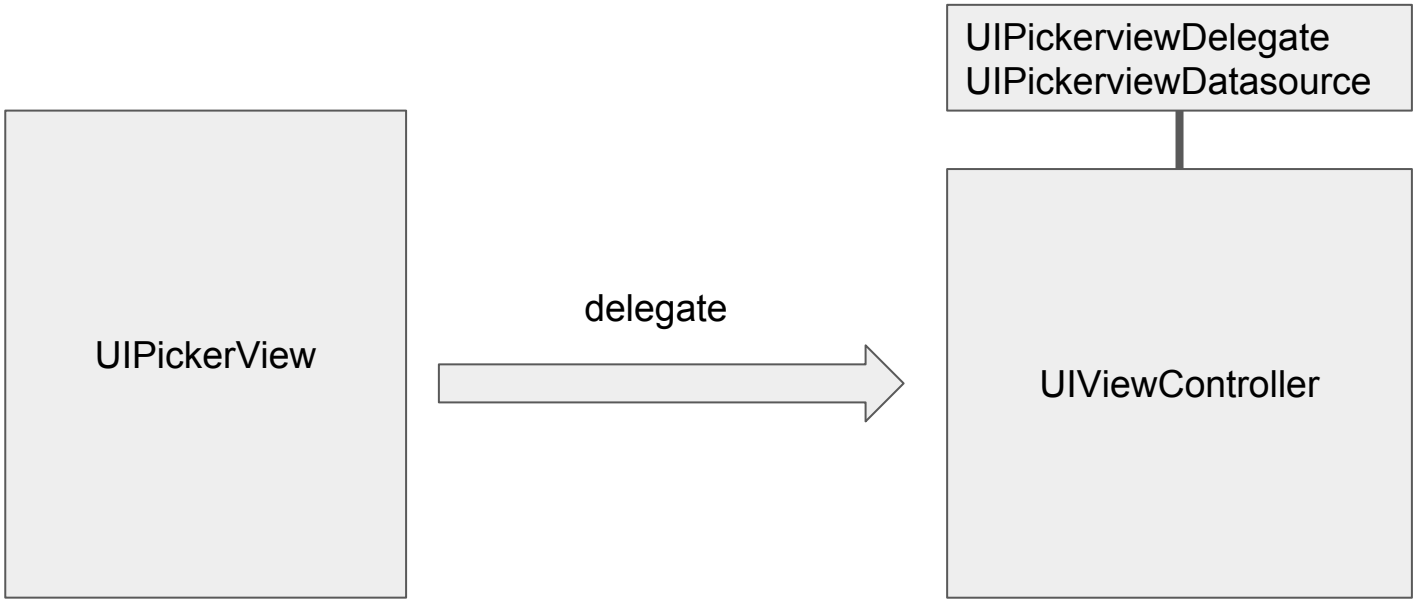
프로토콜 - 초기화 정의

- class / struct / enum에 사용하는 init() 초기화 사용가능
- class 경우 클래스 내부의 init()과 구분하기 위해 프로토콜의 init() 정의시 require 키워드 붙음
- init?() failable initializer 정의가능 (초기화 조건 만족 안할시 옵셔널 특정 초기화 설정)

```
protocol Test {
    init(str:String)                        //initializer
    init?(str:String?, str2:String?)    //failable initializer
}

class Test1 : Test {
        required init(str: String) { self.name1 = str }
        required init?(str: String?, str2: String?) {
                if let s1 = str, let s2 = str2 {
                        self.name1 = s1+s2
                }else{
                        return nil
                }
        }
}
```

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....
    }
}
```

델리게이트 패턴

```
import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource  {

    override func viewDidLoad() {
        super.viewDidLoad()

        pickerView.delegate = self
        pickerView.dataSource = self
    }

    //열의 갯수
    func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return PICKER_VIEW_COLNUM
    }

    //행의 갯수(실제 보일 컨텐츠 갯수)
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return imageFileName.count
    }

    //피커뷰의 높이
    func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) -> CGFloat {
        return PICKER_VIEW_HEIGHT
    }

    //피커뷰 이름 타이틀 세팅
    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return imageFileName[row]
    }
```

델리게이트 패턴

```
public protocol UIPickerViewDelegate : NSObjectProtocol {


    // returns width of column and height of row for each component.
    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, widthForComponent component: Int) -> CGFloat

    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) -> CGFloat


    // these methods return either a plain NSString, a NSAttributedString, or a view (e.g UILabel) to display the row for the component.
    // for the view versions, we cache any hidden and thus unused views and pass them back for reuse.
    // If you return back a different object, the old one will be released. the view will be centered in the row rect
    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?

    @available(iOS 6.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, attributedTitleForRow row: Int, forComponent component: Int) -> NSAttributedString? // attributed
        title is favored if both methods are implemented

    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component: Int, reusing view: UIView?) -> UIView


    @available(iOS 2.0, *)
    optional public func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int)
}
```

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 NumCheckError:Error {
        case notNum
        case characterNotParsing(char:Character)
}
```

Error handle - throws / throw

- throws 키워드를 통해 에러 예외처리 호출정의
- throw로 작성한 에러를 호출 (에러를 던진다는 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: value)
                print(num)
        } catch {
                print(error.localizedDescription)
        }
}
```

제네릭 (Generic)

- 특정 인자타입에 대한 유동적으로 적용하기 위한 문법장치 (유형매개변수)
- 특정타입에 대한 이름을 지어 작성. (와일드 카드 표기)
- 제네릭 타입에 제한을 주기 위해 “ : ” 상속/채택을 사용하며, 클래스/프로토콜 타입으로만 제한구현 가능

```
ex)
struct Stack<Element> {
        var items = [Element]()
        mutating func push(_ item:Element){
                items.append(item)
        }

        mutating func pop() -> Element {
                return items.removeLast()
        }
}
```

```
ex)
var doubleStack = Stack<Double>();
doubleStack.append(3.1)

var intStack = Stack<Int>();
instatck.append(23)

var stringStack = Stack<String>();
stringStack.append(“test”)
```

제네릭 (Generic) - 타입 제약

- 제네릭 타입에 제약을 주기 위해 " : " 상속/채택을 사용하며, 클래스/프로토콜 타입으로만 제약가능

```
ex)
struct Stack<Element:Integer> {
    var items = [Element]()
    mutating func push(_ item:Element){
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

```
ex)
var intStack = Stack<Int>()      // (o)
var doubleStack = Stack<Double>() // (x)
var stringStack = Stack<String>()  // (x)
```

```
/// not use it directly.
public protocol _Integer : ExpressibleByIntegerLiteral, CustomStringConvertible,
    Hashable, IntegerArithmetic, BitwiseOperations, _Incrementable {
}

/// A set of common requirements for Swift's integer types.
public protocol Integer : _Integer, Strideable {
}
```

```
/// on 64-bit platforms, `Int` is the same size as `Int64`.
public struct Int : SignedInteger, Comparable, Equatable {
```

```
/// A set of common requirements for Swift's signed integer t
public protocol SignedInteger : _SignedInteger, Integer {
```

제네릭 (Generic) - 관련타입

- associatedtype 키워드로 프로토콜 재정의시 제네릭타입 유동적으로 적용가능

ex) associatedtype

```
protocol Container {
        associatedtype ItemType:Integer

        var count:Int {get}
        mutating func append(_ item:ItemType)
        subscript(i:Int) -> ItemType {get}
}
```

ex) 적용

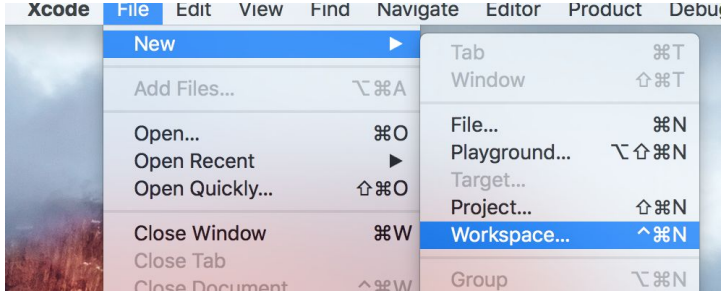
```
class MyContainer : Container {
        var items:Array<Int> = Array<Int>()
        var count: Int {
                return items.count
        }
        func append(_ item: Int) { // type associated
                items.append(item)
        }

        subscript(i:Int) -> Int {  // type associated
                return items[i]
        }
}
```

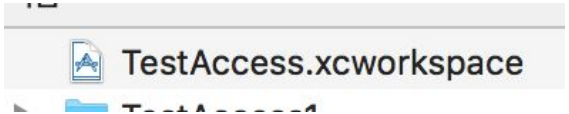
ios

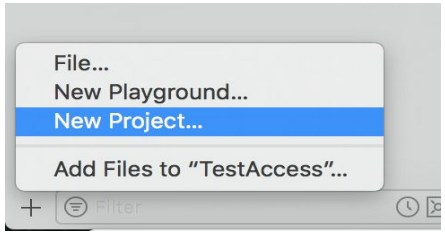
- 워크스페이스 생성 / 프로젝트명 수정방법
- 화면 전환 (프로그래밍 방식 / segue 방식) / unwind
- 뷰 간 데이터 전달(passing data)
- UIAlertViewController

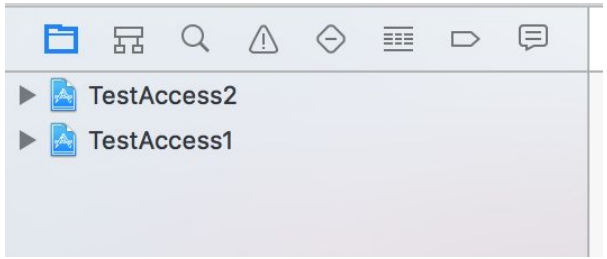
워크스페이스 생성

- 여러 프로젝트(모듈)를 묶는 큰 그룹 개념
- 각 프로젝트는 모듈로 취급 (import시 프로젝트명)
- 네이게이션 영역 왼쪽 하단의 + 눌러서 필요한 프로젝트 생성 및 추가 가능



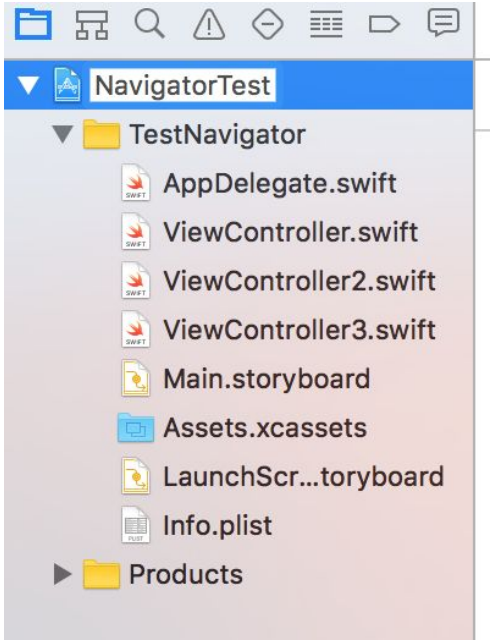


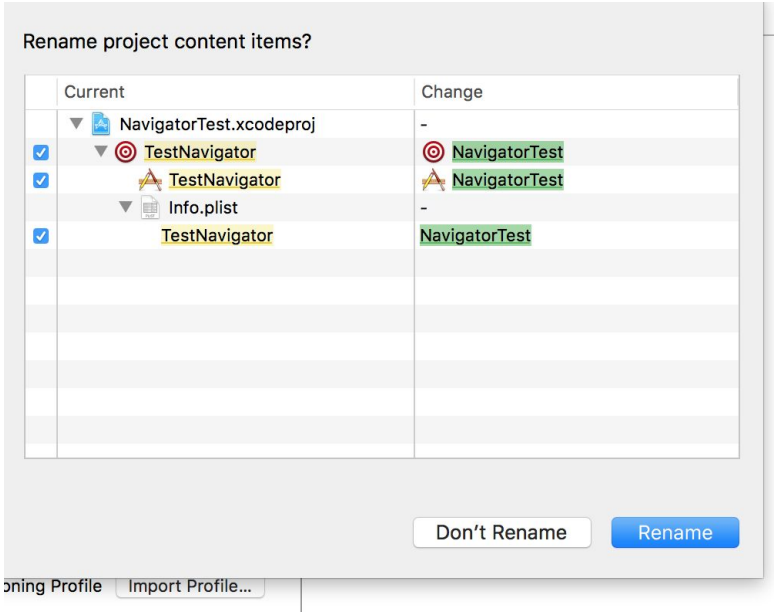




프로젝트명 수정

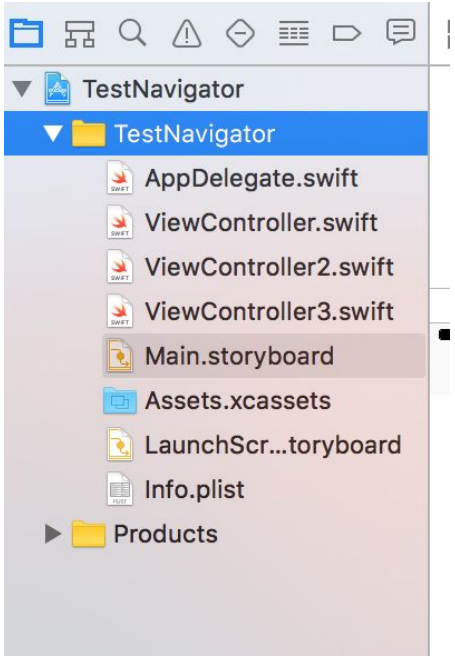
- 네이게이터 영역의 프로젝트명 폴더를 클릭하여 이름 수정

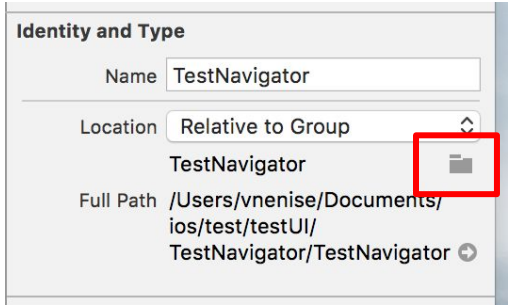


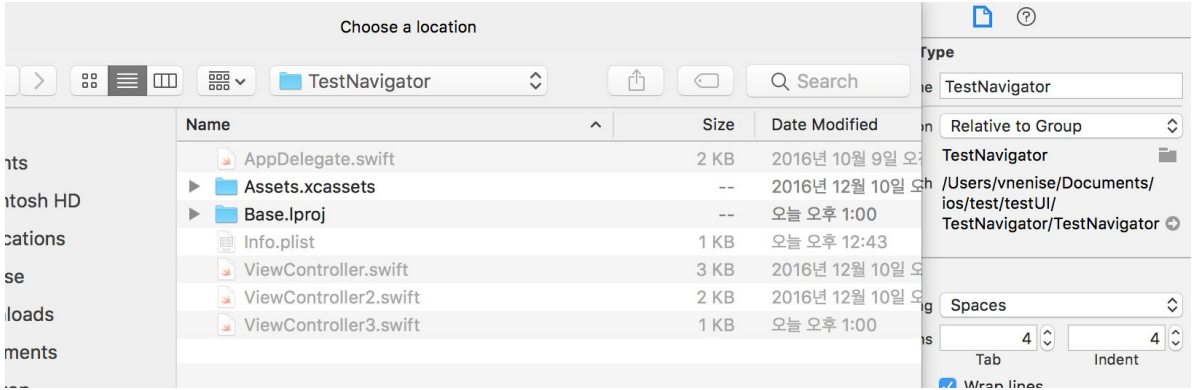


프로젝트명 수정

- 프로젝트명 수정 후에는 실제 참조하는 프로젝트 폴더의 위치 설정 필요.

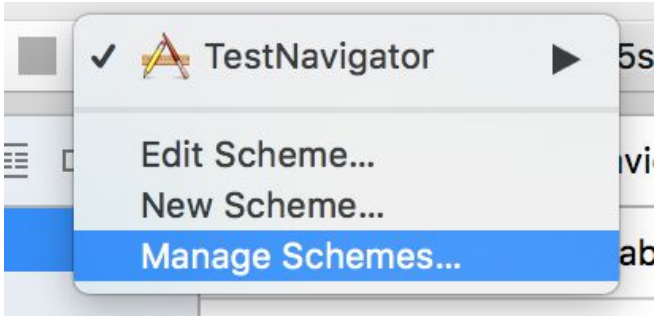


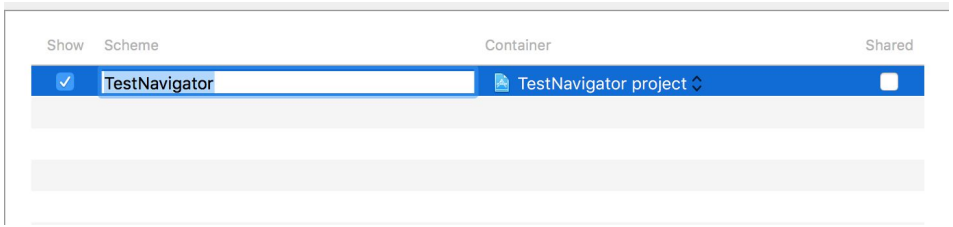




프로젝트명 수정

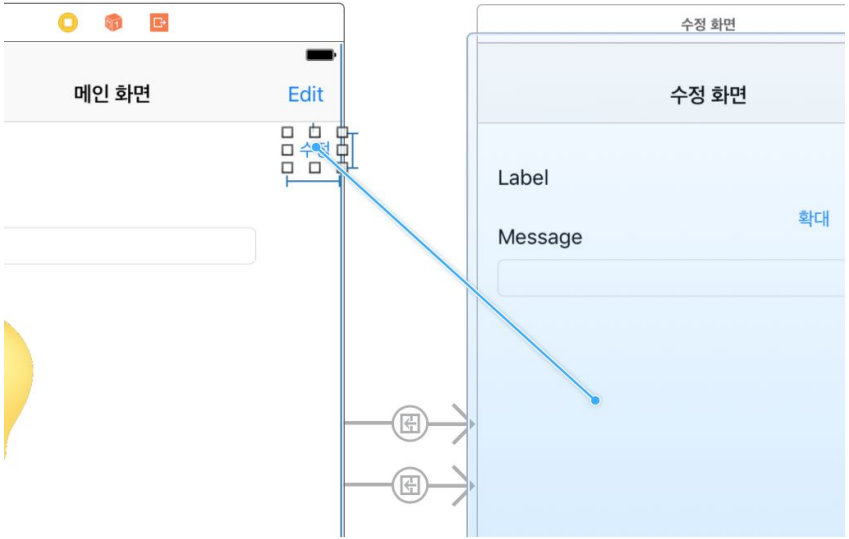
- 실행앱 이름 변경하고 싶을시 manage Schemes에 Scheme명 변경

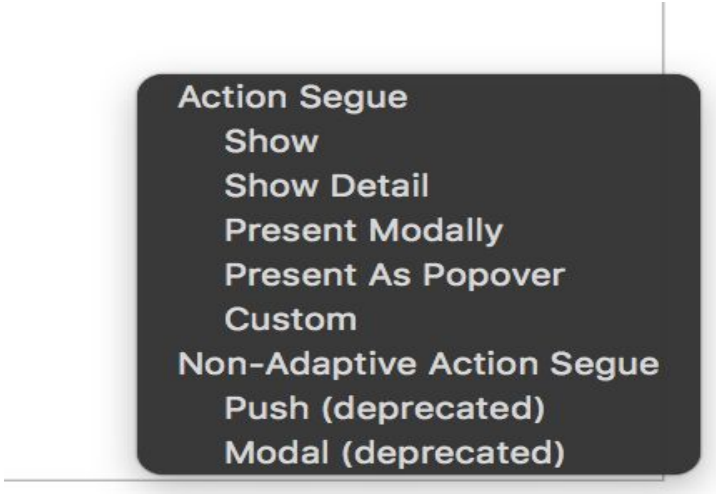




화면전환

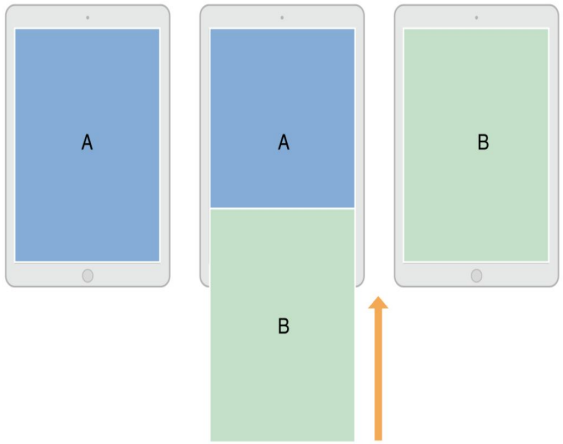
- 화면전환에 대한 식별자 Segue 사용 (안드로이드: intent와 유사)
- 화면전환에는 직접 UI에 대한 presentController를 호출하거나 Segue 식별자로 전환방식
- IB(interface builder)에는 segue에 대한 화면전환 action 제공
- UIViewController (manual segue) 또는 UIControl(action segue) 형태의 segue 나눔

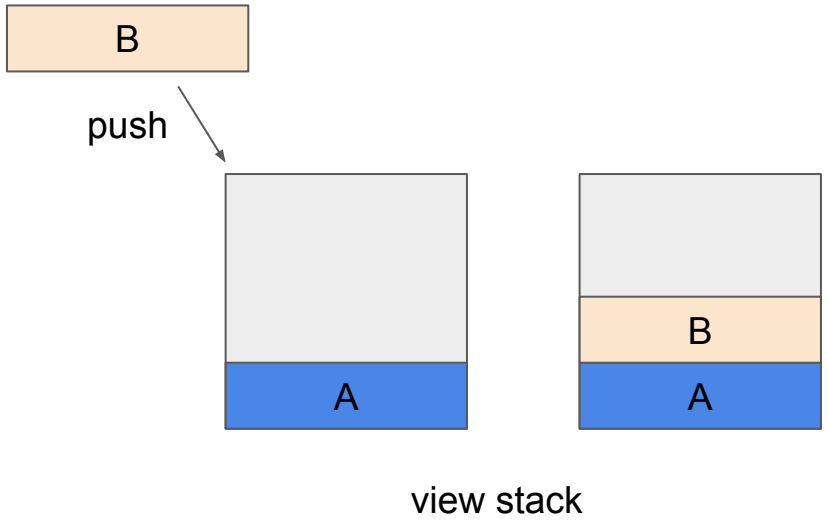




Segue - Show (push)

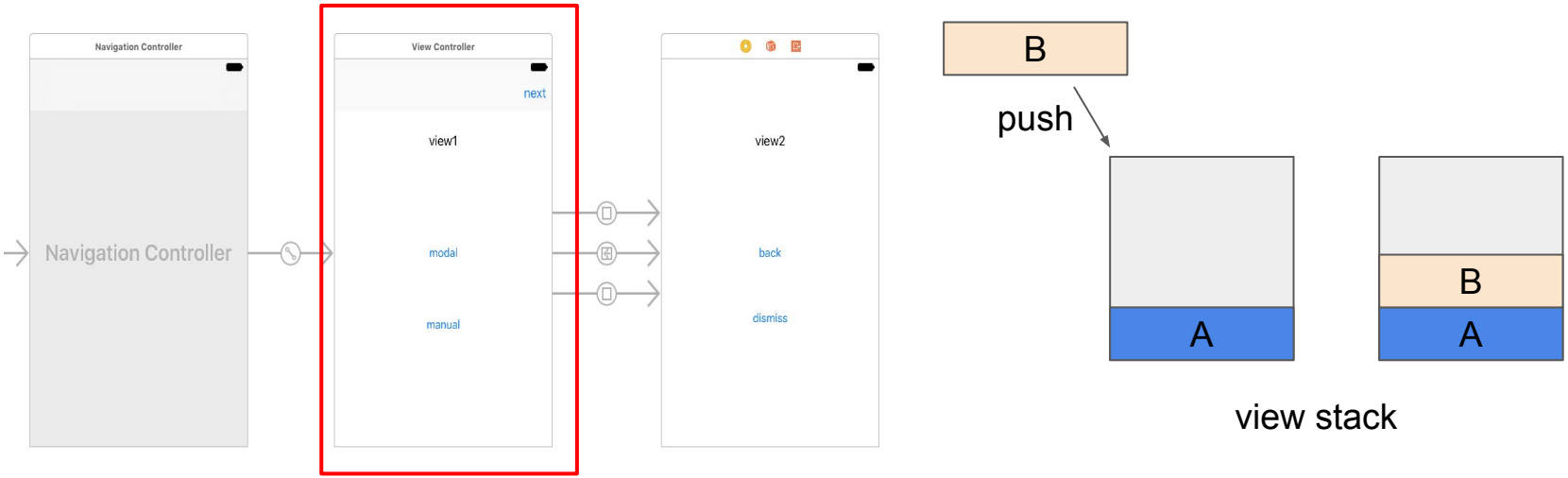
- NavigationViewController에 대한 뷰스택에 쌓으면서 보여주는 화면형태 (show)
- NavigationViewController는 push/pop 구조로 화면전환 관리
  (pushViewController / popViewController)





NavigationViewController

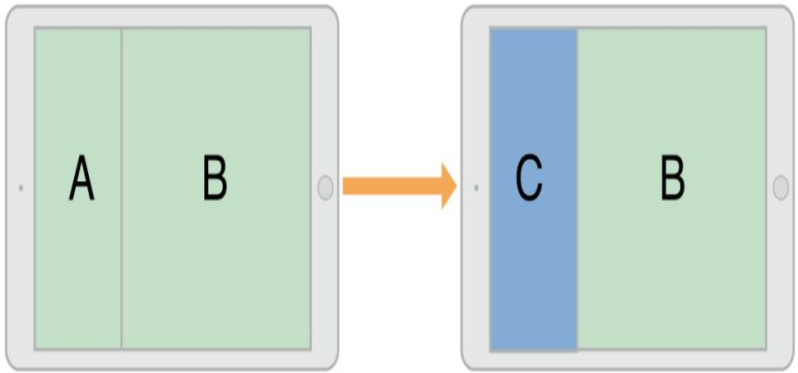
- 최초 실행 ViewController를 rootViewController 지칭
- 스택구조로 뷰를 관리되어, 화면전환시 push/pop 메소드로 전환 필요
- NavigationViewController 연결되어 상단의 navigation 메뉴가 자동 할당됨



rootViewController

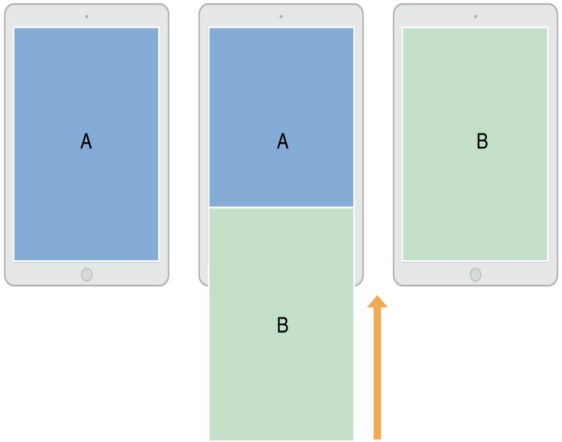
Segue - Show Detail (replace)

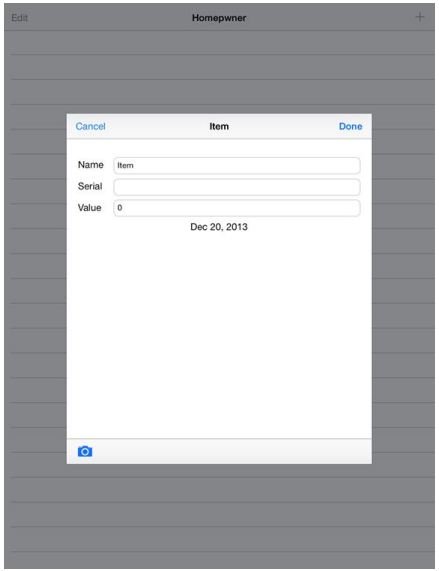
- master와 detail로 나눠지는 화면구성에서 detail 영역을 대치(replace)해서 화면형태
- 태블릿이 지원되는 Universal 앱 경우 show-detail 화면 많이 활용
- view스택에서는 영향이 없음



Segue - Present Modally

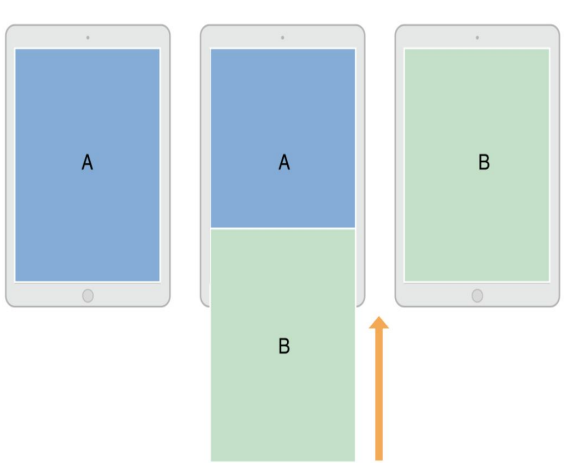
- 기존 화면을 덮으면서 위로 뜨는 화면형태 (ios에서 가장 일반적인 뷰 전환)
- ios 8부터 modal 대신 present modally 사용 (modal deprecated)

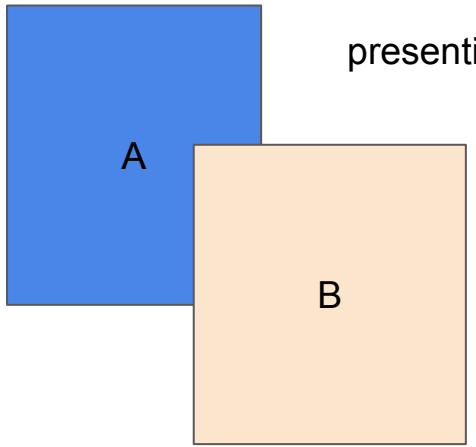




presenting / presentedViewController

- present() 화면전환 경우 두 뷰 간의 관계를 presented / presenting으로 구별
- 보여진 뷰를 presentedViewController에 참조 포인터 저장
- 보여지고 있는 뷰는 presentingViewController에 참조 포인터 저장



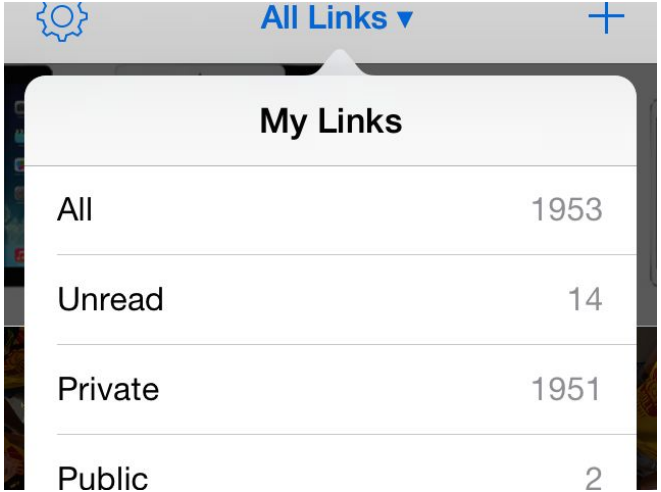


presentingViewController (A)

presentedViewController (B)

Segue - Present As Popover

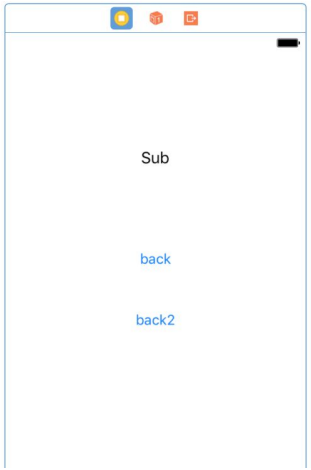
- 작은 팝업형태의 뷰 띄우는 화면형태
- 새로 띄운 뷰의 바깥영역을 터치하면 뷰 사라짐

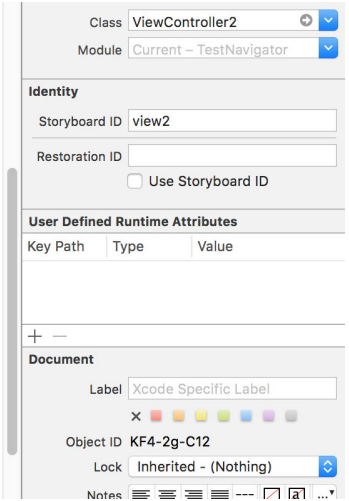


화면전환 - 프로그래밍 방식

- 특정 ViewController 인스턴스 생성하여 화면전환
- ViewController에 스토리보드 특정id 부여

1) present()를 통한 present modally 방식
2) NavigationViewController를 통한 pushViewController 방식






```
let storyboard = UIStoryboard(name: "Main", bundle: nil)
let uv = storyboard.instantiateViewController(withIdentifier: "view2")

self.present(uv, animated: true)       // modally 방식
self.navigationController?.pushViewController(uv, animated: true) //nvaigation 방식
```

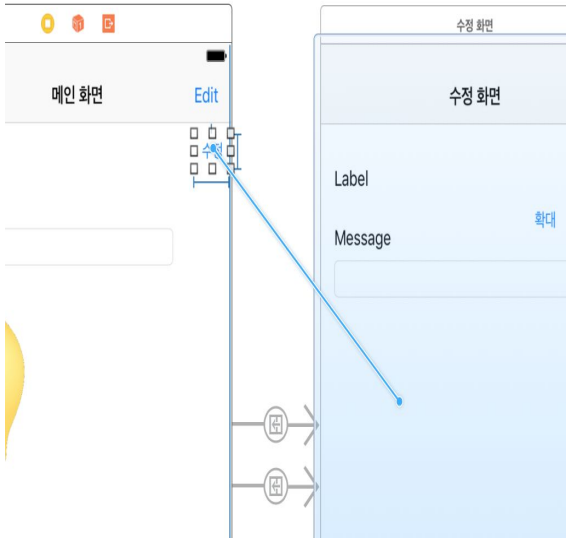
화면전환 - 프로그래밍 방식

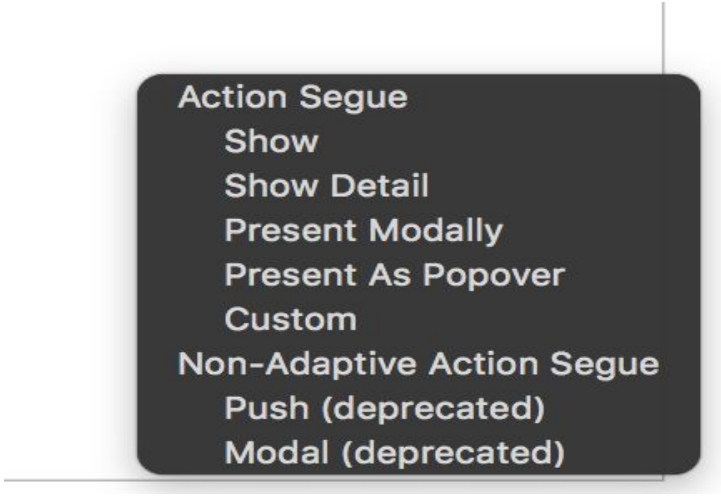
- 이벤트 함수에 직접 불러온 ViewController 지정 후 presentController 함수호출

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

화면전환 - segueway 이용

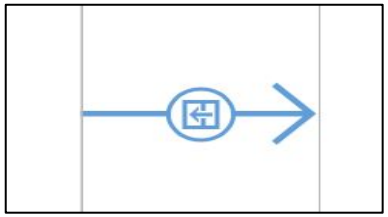
- Segue라는 화면전환 객체를 이용한 방식
- UIControl 상속받고 있는 특정대상에 화면전환 이벤트 거는 방식 (action segue)
- UIViewController 자신에게 화면전환 이벤트 거는 방식 (manual segue)

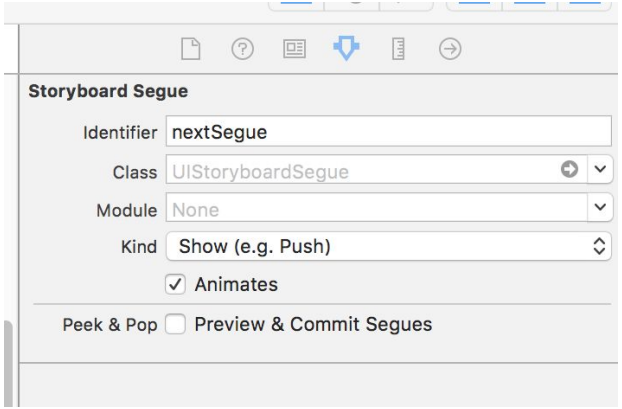




Segue - 제어방법

- segue에 identifier 설정 후 ViewController에서 segue 관련 code정의





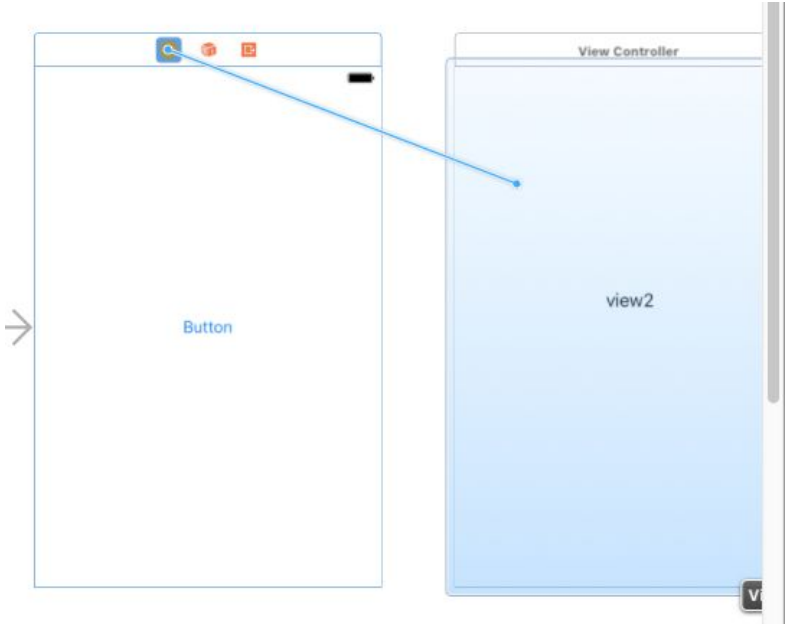
func **prepare**(**for segue**: UIStoryboardSegue, sender: Any?) **//segue**가 실행전에 초기화 또는 특정기능 설정

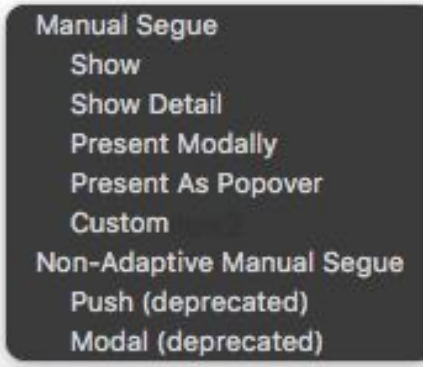
func **performSegue**(**withIdentifier** identifier: String, sender: Any?) **//performSegue()**에 대한 특정 **segue** 호출처리

UIViewController 자신.**performSegue**(withIdentifier: "세그 identifier id", sender: 이벤트대상) **//특정segue** 호출설정

화면전환 - manual Segue

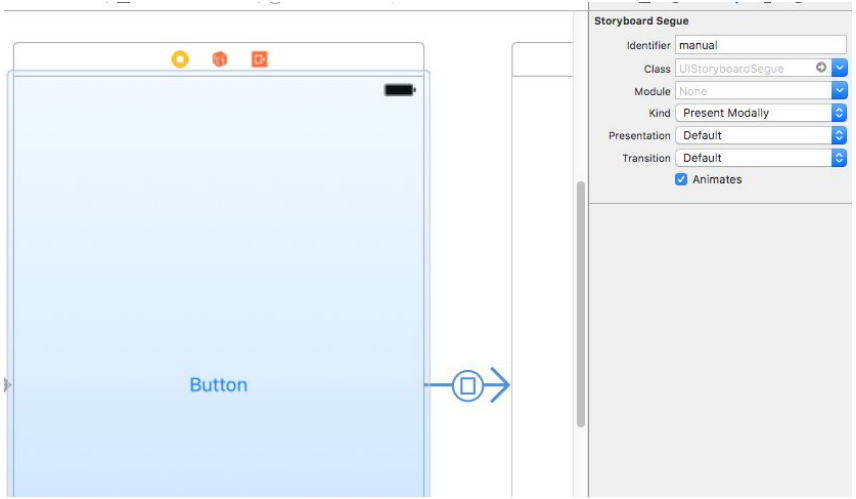
- UIViewController 자체에 전환할 뷰 지정하는 segue
- performSegue()를 통한 화면전달 또는 구현





화면전환 - manual Segue

performSegue(withIdentifier identifier: String, sender: Any?)

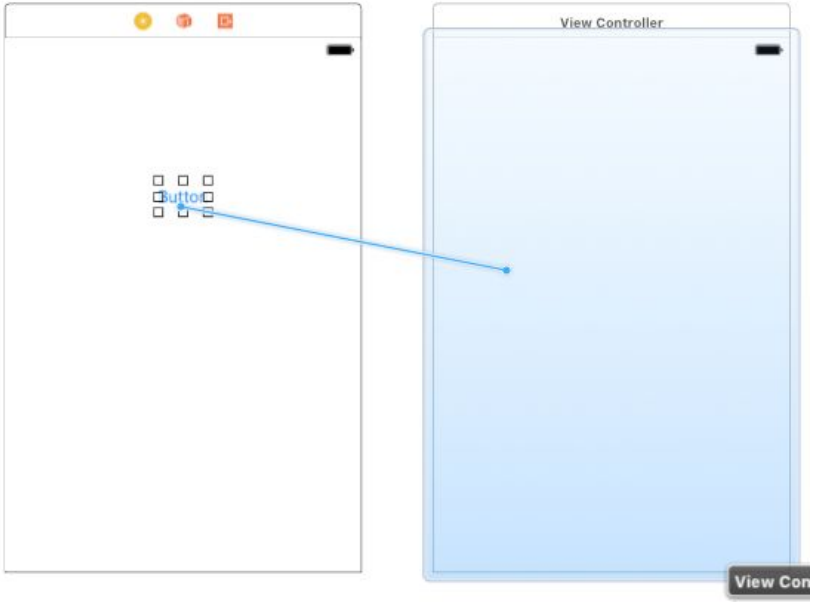


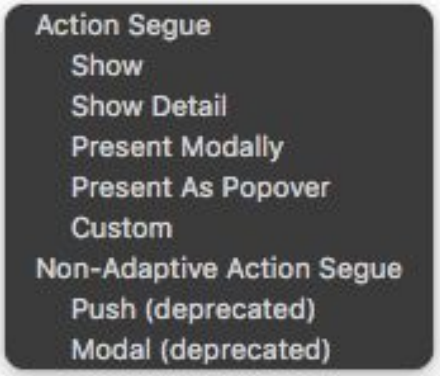
ViewController.class

```
@IBAction func next(_ sender: UIButton) {
        self.performSegue(withIdentifier: "manual", sender: self)
}
```

화면전환 - Action Segue

- 이벤트 설정이 가능한 UI요소에 대한 segue
- 별도의 설정없이 이벤트 발생시 화면전환
- 특정조건만 전환 원할시 shouldPerformSegue() 특정조건 구현하여 return Boolean처리





화면전환 - Action Segue

func **shouldPerformSegue**(withIdentifier identifier: String, sender: Any?) -> **Bool**

ViewController.class

```
override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?) -> Bool {
        if identifier == "action" {
            return false  //화면전환 안함
        }
        return true       //화면전환
}
```

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"
        }
}
```

이전화면 전환 - unwind

- 뷰간의 연결고리가 되어 있어 dismiss()나 unwind기능으로 이전화면 전환
- UIViewController 에 dismiss()함수 내장
- present() 전환할 경우 presentingViewController에서 dismiss 호출해야함
- navigationView로 연결된 뷰는 popViewController 사용

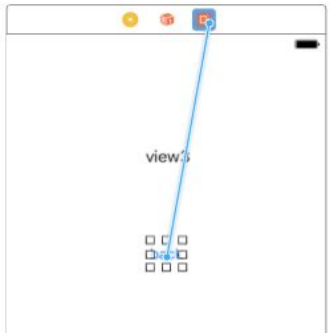
```
self.dismiss(animated: true, completion: nil)

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

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

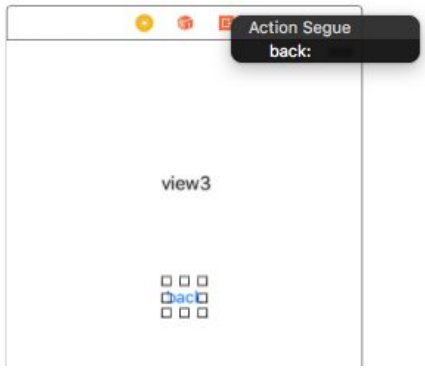
이전화면 전환 - unwind Segue

- ViewController에 UIStoryboardSegue타입의 매개변수 있는 @IBAction함수를 갖는 뷰 지정하여 이전화면 전환

ViewController1 class

```
    @IBAction func back(_ segue:UIStoryboardSegue){

    }
}
```





뷰간 데이터 전달(passing data)

1) viewController 인스턴스 생성하여 프로퍼티 참조방식

```
@IBAction func sendData(_ sender: UIButton) {
        let story = UIStoryboard(name: "Main", bundle: nil)
        let vc2 = story.instantiateViewController(withIdentifier: "view2") as!
        ViewController2
        vc2.str = "test data" //인스턴스의 프로퍼티 참조하여 값 전달
        present(vc2, animated: true)
}
```

뷰간 데이터 전달(passing data)

2) prepare()함수를 이용하여 viewController 프로퍼티 참조방식

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        let vc2 = segue.destination as! ViewController2
        vc2.str = "test data"          //전환할 viewcontroller에 프로퍼티 참조
}
```

이전화면 데이터 전달

1) viewController 인스턴스 프로퍼티 참조방식 - present() 경우

```
@IBAction func back(_ sender: UIButton) {
    let vc1 = self.presentingViewController as! ViewController
    vc1.str = "return data!!"
    vc1.dismiss(animated: true)
}
```

이전화면 데이터 전달

2) viewController 인스턴스 프로퍼티 참조방식 - navigationView 경우

```
@IBAction func back(_ sender: UIButton) {
    let list = self.navigationController?.viewControllers
    let vc = list?[(list?.count)!-2] as! ViewController
    vc.str = "test nav!"
    _ = self.navigationController?.popViewController(animated: true)
}
```

이전화면 데이터 전달

3) delegate 패턴을 이용한 값 전달방식

```
class ViewController: UIViewController, ViewController2Delegate {

    var str:String?

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        let vc2 = segue.destination as! ViewController2
        vc2.delegate = self
    }

    func sendData(data: String) {
        self.str = data
    }
```

ViewController class

- 리턴받아야 할 뷰의 delegate함수 대신 구현처리
- 화면전환시 전환화면의 protocol를 self로 등록

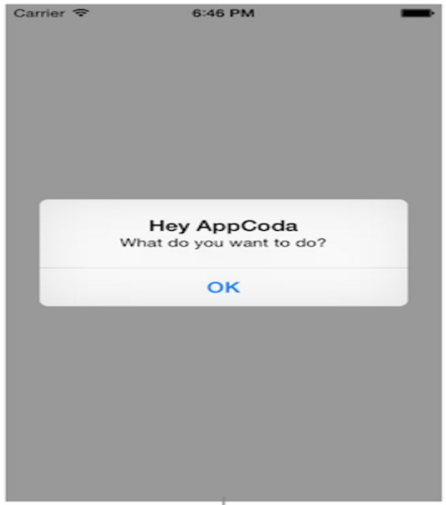
```
protocol ViewController2Delegate {
    func sendData(data:String)
}

class ViewController2: UIViewController {

    var delegate:ViewController2Delegate?

    @IBAction func back(_ sender: UIButton) {
        if let delegate = delegate {
            delegate.sendData(data: "teet return data!!")
        }
        _ = self.navigationController?.popViewController(animated: true)
    }
```

ViewController2 class

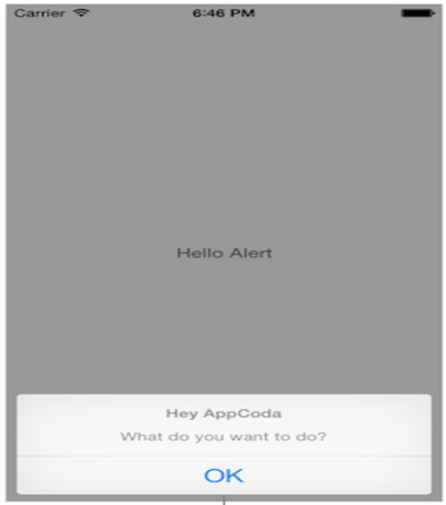
- 리턴값 전달시 ViewController전환시 등록된 delegate 함수를 이용하여 값 전달

UIAlertController

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



Alert



ActionSheet

UIAlertController

클래스	설명	기타항목
UIAlertController	Alert창에 대한 UIViewController	- addAction메소드로 특정액션 추가가능 - AlertAction의 종류 - actionSheet / alert (UIAlertControllerStyle enum값 제공)
UIAlertAction	Alert창에 대한 각 분기 메뉴	- handler 부분에 동작내용 정의 - 동작내용은 클로저(closure)로 구현

<예제>

```
let alertController = UIAlertController(title: "경고", message: "경고창", preferredStyle:
UIAlertControllerStyle.alert)
let closeAction = UIAlertAction(title: "닫기", style: UIAlertActionStyle.default, handler: nil)

alertController.addAction(closeAction)
present(alertController, animated: true, completion: nil)
// 화면출력에 대한 UIViewController 메소드
```