Swift Study 03

2016. 11.12

Swift 문법

- 연산자 / 조건(switch-case, guard)
- 반복문(for문, while문)
- 함수(function) / underscore (_)
- 클래스(class) / property / method
- 컬렉션(collection) - 배열(array)

연산자 - 산술 / 비교 / 논리 / 범위 / 대입

산술연산자	+, -, *, /, %
비교연산자	<, >, <=, >=, ==, !=
논리연산자	&&(and), ||(or), !(not)
범위연산자	1...5 (1~5), 1..<5 (1~4)
대입연산자	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

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

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.

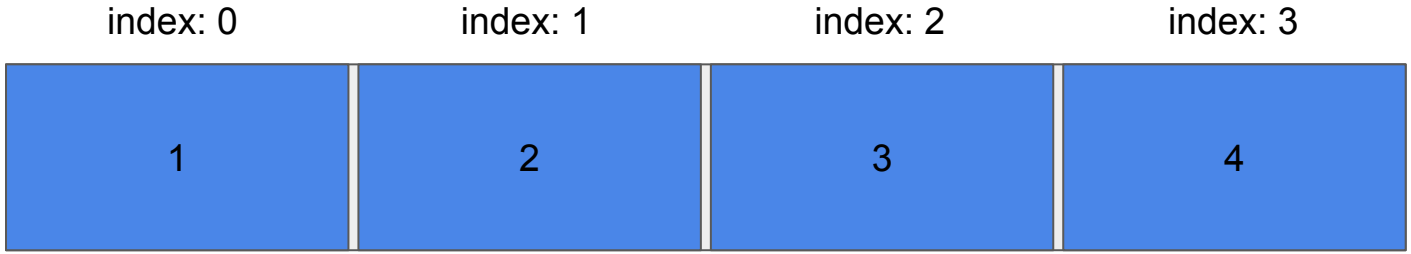
extension / final 키워드

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
		}

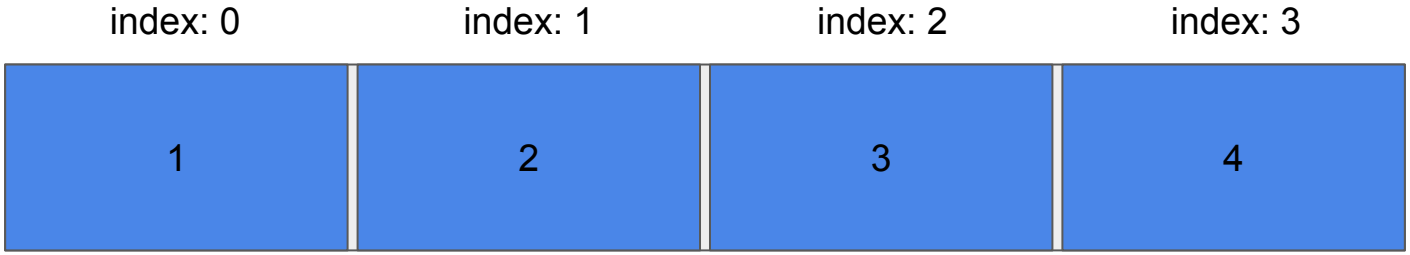
컬렉션(Collection) - 배열(Array)

- 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
```

Swift 문법

- 프로토콜 (protocol)
- 익명함수 (클로저 closure)
- 델리게이트 패턴 (delegate pattern)

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
    }
}
```

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)

- 이름이 없는 함수 (매개변수와 반환형, 실제 구현체만 정의)

<형식>

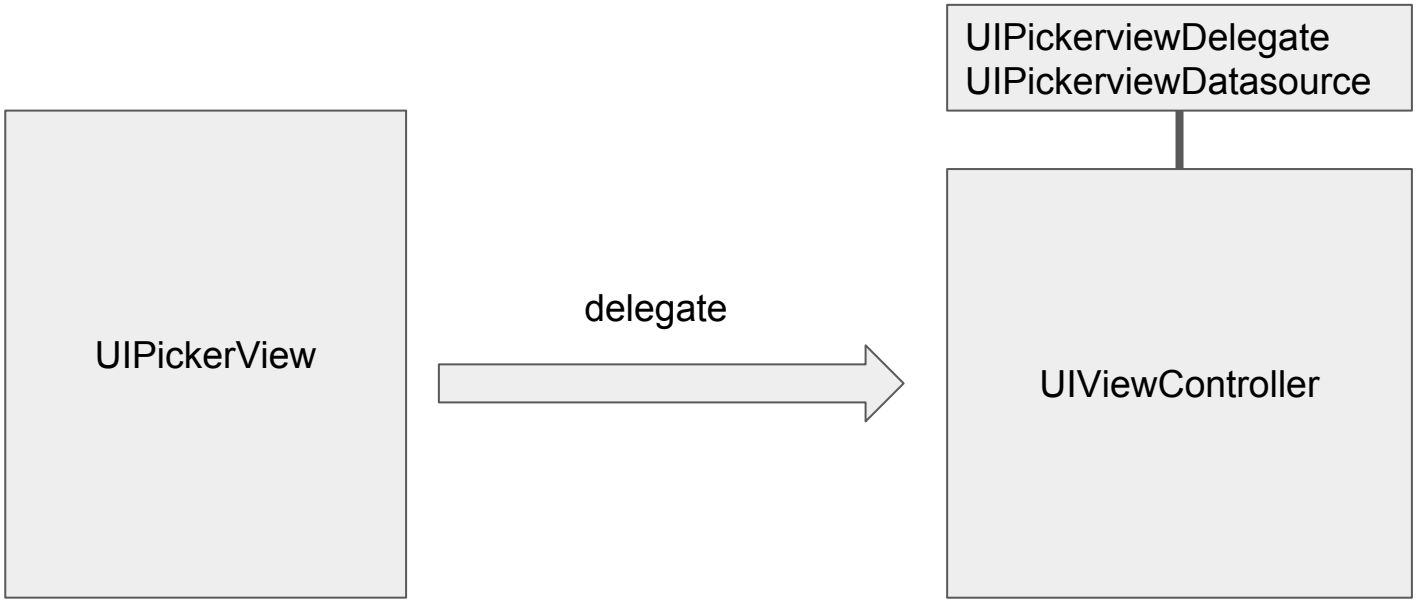
```
// 기본형
{
   (매개변수:타입) -> (반환형) in
      작성할 내용
}

// 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
}
```

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

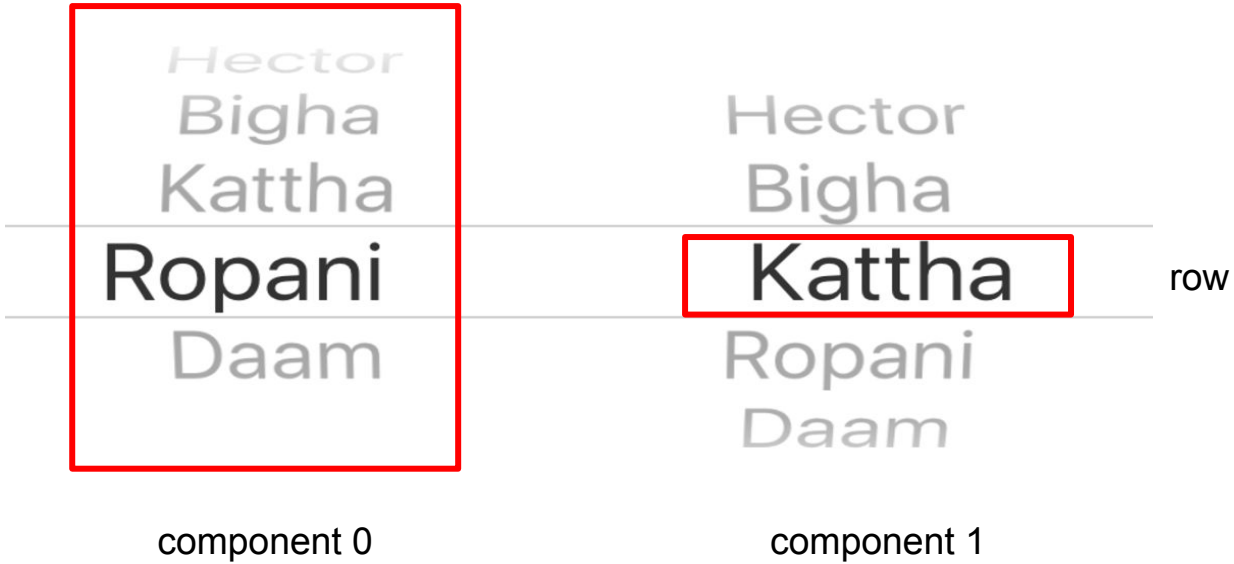
```
import UIKit

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)
}
```

PickerView

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



PickerViewDatasource

메소드	매개변수 / 반환형	기능
numberOfComponents	(in pickerView: UIPickerView) -> Int	피커뷰의 전체 열 갯수 설정
pickerView	(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int	행의 갯수(실제 보일 컨텐츠 갯수)

PickerViewDelegate

// component 너비
func pickerView(_ pickerView: UIPickerView, widthForComponent component: Int) -> CGFloat

// 행의 높이
func pickerView(_ pickerView: UIPickerView, rowHeightForComponent component: Int) -> CGFloat

// 행의 타이틀 세팅
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String?

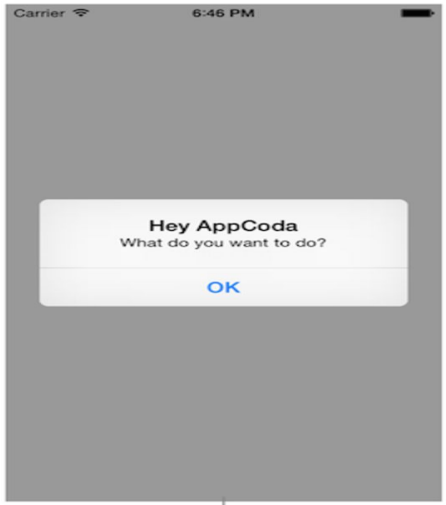
// title 속성 설정
func pickerView(_ pickerView: UIPickerView, attributedTitleForRow row: Int, forComponent component: Int) -> NSAttributedString?

// 행에 대한 뷰 설정
func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component: Int, reusing view: UIView?) -> UIView

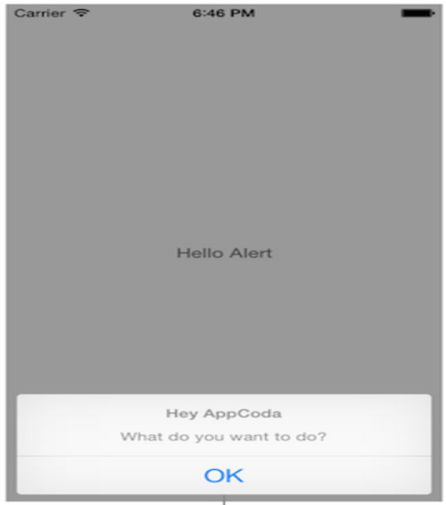
// 피커뷰를 선택시 설정
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int)

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 lampOnAlert = UIAlertController(title: "경고", message: "경고창", preferredStyle: UIAlertControllerStyle.alert)
let onAction= UIAlertAction(title: "네, 알겠습니다.", style: UIAlertActionStyle.default, handler: nil)

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