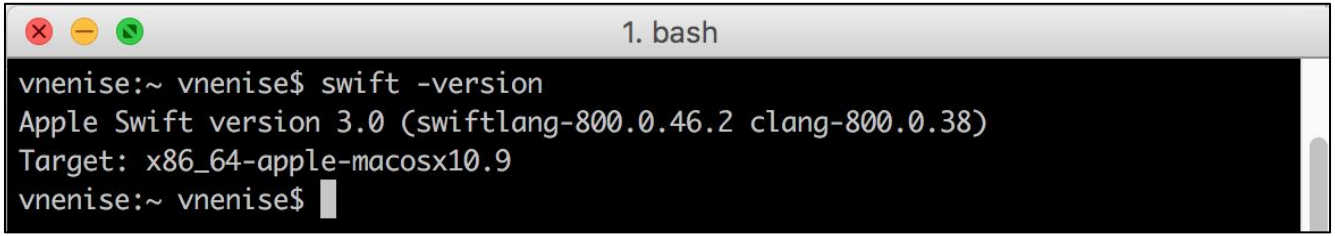
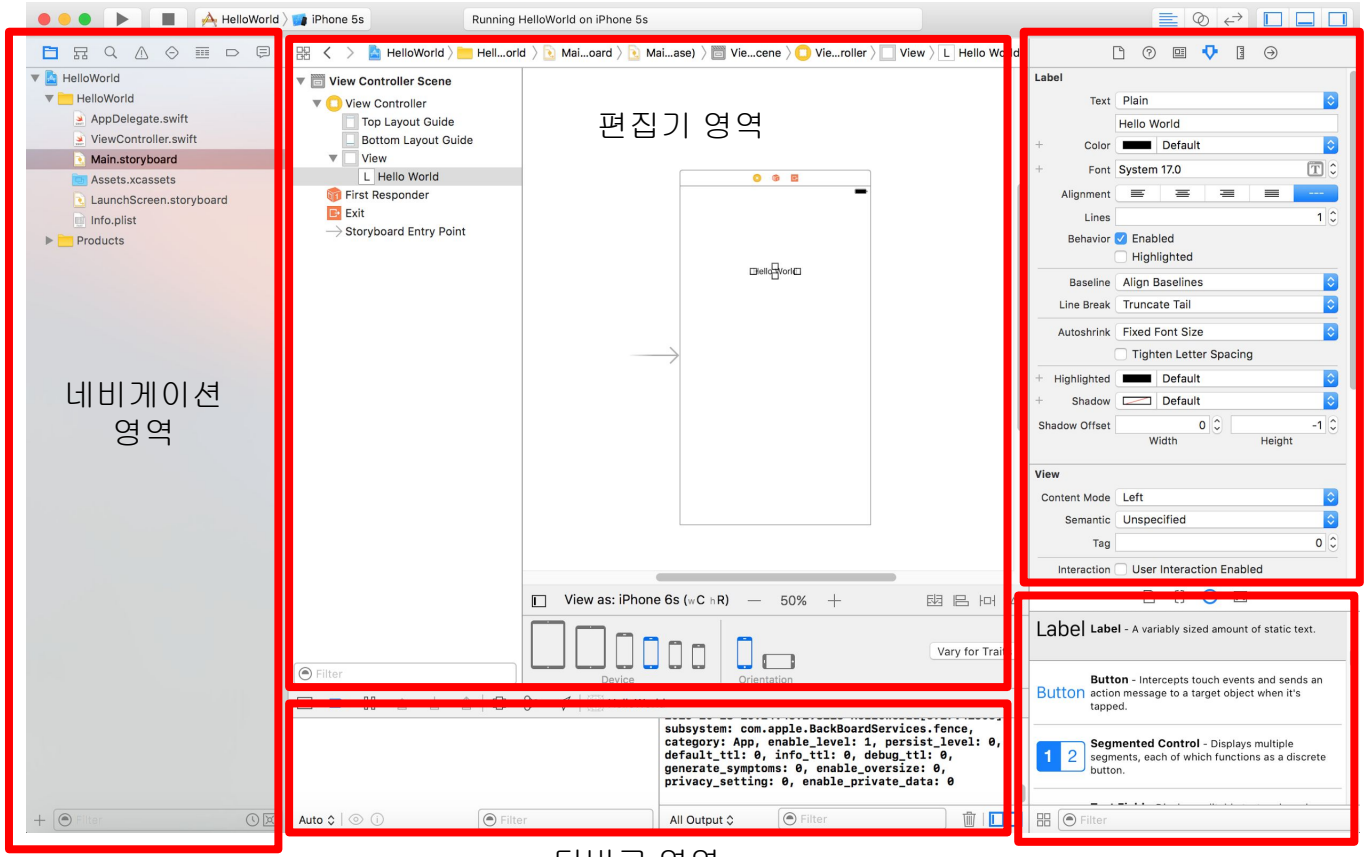
Swift Study 02

2016. 11.5

Swift 3.0 & Xcode 8.0



Xcode



Swift 문법

- 기본 자료형 (데이터 타입)
- 변수 및 선언방식
- 옵셔널 (optional)
- 조건문

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”)
}
```

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.

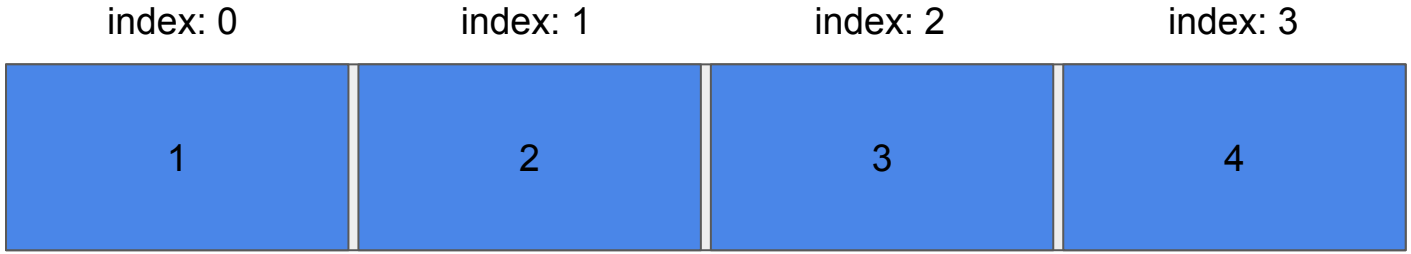
extension / final 키워드

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
		}

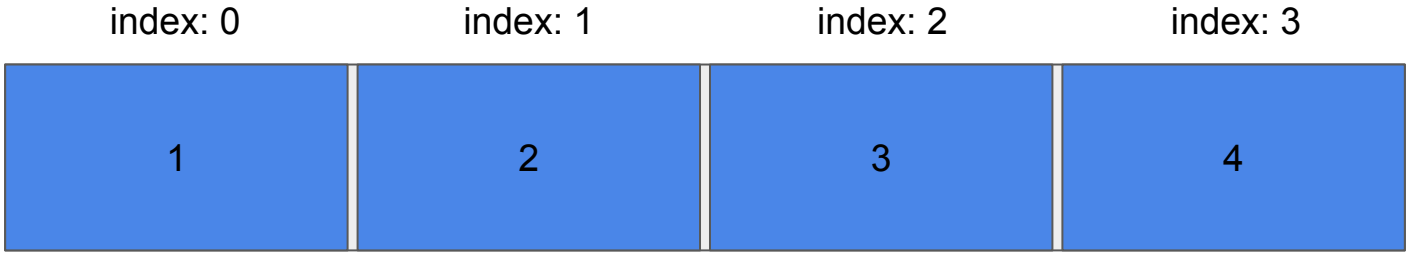
컬렉션(Collection) - 배열(Array)

- 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
```