Kotlin 기초 02

- http://developer.android.com

목차

- 람다함수
- 고차함수
- Kotlin 클래스
- 유용한 클래스
- 실습

람다함수 (Lambda function)

in Kotlin

- 이름 없는 익명함수로 표현되는 코드 조각
- 변수 취급 가능 (변수에 대입, 매개변수로의 전달 등)
- 기본 형식

{ 매개변수 ... -> 함수 Body }

람다함수의 사용

- 람다함수는 함수명이 없으므로 필요 시 변수에 대입한 후 변수명으로 함수 사용
- 람다함수의 마지막 값은 return 값 취급

• 매개변수가 없을 경우 -> 생략 가능

```
val lambdaNoParam = { -> println("lambda!")}  // == { println("lambda!") }
lambdaNoParam()

val lambdaWithParam = { num1: Int -> num1 + 10 }
// val lambdaWithParam : (Int) -> Int = { num1: Int -> num1 + 10 }  // 변수에 할당 시 람다 함 타입 선언
val result = lambdaWithParam(10)
println (result)
```

• 코드의 마지막이 값일 경우 return 값(표현식) 취급

람다함수의 매개변수

매개변수가 있는 람다함수의 예

```
val lambdaFunc : (Int, String) -> Unit = { num: Int, str: String ->
    println (num * num)
    println (str)
}
```

- 람다함수 Type: 생략 가능
- 람다함수 Type이 있을 경우 생략 가능

it 키워드의 사용

- 람다함수의 매개변수가 하나일 경우 생략 후 키워드 it 로 사용 가능

```
val lambdaPower : (Int) -> Int = { num: Int ->
    num * num
}
println (lambdaPower(2))
val lambdaWithIt : (Int) -> Int = { it: Int
    it * it
}
println (lambdaWithIt(3))
```

- 매개변수가 하나일 경우 해당 매개변수 표현

고차함수 (Higher-order function) 01

in Kotlin

- 다른 함수를 매개변수로 전달받거나 반환할 수 있는 함수
  → 함수형 프로그래밍(Functional programming)의 기법
- 예:

```
val nameFunc : () -> Unit = {
    println ("SomSom!")
}

val subjectFunc : () -> Unit = {
    val subjectName = "Mobile software"
    println (subjectName)
}

fun higherOrderFunc (argFunc: () -> Unit) {
    println("Dept: Computer")
    argFunc()
}

higherOrderFunc(nameFunc)
higherOrderFunc(subjectFunc)
```

• 함수의 Type 동일
- 매개변수/반환타입이 없는 함수

• 함수를 매개변수로 전달받음

```
Dept: Computer
SomSom!
Dept: Computer
Mobile software
```

고차함수 02

반환타입이 함수인 고차함수

- 매개변수로 String을 받음
- 반환 타입은 없음

```
fun higherOrderFunc () : (String) -> Unit {
    return { grade ->
        println("Dept: Computer")
        println("Subject: Mobile Software")
        println("Grade: $grade")
    }
}

val returnedFunc = higherOrderFunc()
returnedFunc("A")
```

- 문자열 type인 grade 를 전달 받아 화면 출력을 수행하는 람다함수

Null 안정성(Null-safety) 연산자 01

개요

- 객체가 null인 상태에서 사용할 때 발생하는 NullPointerException 을 예방하기 위해 사용

```
val str: String = null
println (str.length)
```

- null 인 객체를 사용하므로 실행할 경우 NullPointerException 발생
- 일반적인 변수는 null 값을 가질 수 없도록 제한

- [타입?]연산자 : 변수 선언 시 null을 할당가능한 변수로 선언하고자 할 때 사용
- [객체?.멤버]연산자: 객체가 null 상태일 경우 null 이라는 상수 값으로 대체

```
val str: String? = null
println (str?.length)
// null 값 출력
```

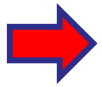
- ? 연산자를 사용하여만 null 할당 가능
- ?. 사용하여 객체가 null 상태일 경우 null 이라는 값으로 대체

Null 안정성(Null-safety) 연산자 02

[값 ?: 대체값] 연산자

- 엘비스 연산자
- [값] 부분이 NullPointerException이 발생하지 않는 경우에는 [값] 사용, 발생할 경우 [대체값] 사용

```
var str: String? = "Mobile"
println ( str?.length ?: 0 )
str = null
println ( str?.length ?: 0 )
```



```
6
0
```

[값!!] 연산자

- [값] 부분이 null 일 경우 NullPointerException 발생
- 개발자가 예외 상태를 확인할 필요

```
var str: String? = null
println ( str!!.length ?: 0 )   // NullPointerException 발생
```

Kotlin Class 01

class 선언

- Body가 없는 클래스도 선언 가능

```
class MyClass {
    var dept : String

    constructor(dept: String){
        this.dept = dept
    }

    fun getDept() {
        println (dept)
    }
}

fun main() {
    val myObject : MyClass = MyClass("computer")
    myObject.getDept()
```

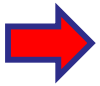
- 멤버변수
- 생성자
- 멤버함수
- 객체 생성

Kotlin Class 02

생성자

- 주생성자 : class 선언부에 선언
- constructor 키워드 생략 가능
- 생성자 선언이 없을 경우 매개변수가 없는 주생성자 생성
- 주생성자 body 는 init { } 에 기술 : 생략 가능

```
class MyClass constructor(dept: String) {
    var dept : String

    init {
        this.dept = dept
    }
}
```



```
class MyClass (dept: String) {
    var dept : String

    init {
        this.dept = dept
    }
}
```

- constructor 생략
- init {} : 주생성자 body

- 주생성자 생략 → 매개변수가 없는 주생성자 자동 생성

```
class MyClass {
    var dept : String

    init {
        dept = "computer"
    }
}
```

- 멤버변수는 init { } 가 있을 경우 초기화 생략 가능

주생성자에서의 멤버변수 선언

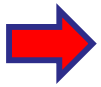
주생성자에서 변수 선언 키워드 사용: val 또는 var

```
class MyClass (dept : String) {
    init {
        println ("$dept")
    }

    fun printDept() {
        println("dept: $dept")
    }
}
```

- dept: 주생성자의 매개변수 → 지역변수 취급
- dept는 소멸한 상태이므로 오류 발생

- val 을 사용하였으므로 dept는 멤버변수

```
class MyClass (val dept : String) {
    init {
        println ("$dept")
    }

    fun printDept() {
        println("dept: $dept")
    }
}
```



```
computer
dept: computer
```

Kotlin Class 03

보조 생성자

- 주생성자 이외의 생성자
- class body 부분에 constructor 키워드로 선언

```
class MyClass {
    constructor(dept: String){
        println("${dept}")
    }
}
```

- 매개변수가 없는 주생성자와 하나의 매개변수를 갖는 보조 생성자

- 주생성자를 명시적으로 구현할 경우 반드시 보조생성자에서 주생성자 (최종적으로) 호출
  - 보조생성자의 개수와 상관 없음

```
class MyClass() {
    constructor(dept: String) : this() {
        println("${dept}")
    }
}
```

- 매개변수가 없는 주생성자 선언 포함
- 보조생성자 호출 후 주생성자를 호출하도록 구현

클래스 상속(Inheritance) 01

기존 클래스의 멤버를 상속

- open 클래스만 상속 가능

```
open class SuperClass {
    init {
        println("Super class")
    }
}

class SubClass : SuperClass() { // 상위클래스의 생성자 호출 필요
    init {
        println("Sub class")
    }
}
```

```
Super class
Sub class
```

클래스 상속(Inheritance) 02

멤버의 재정의(Overriding)

- open으로 선언한 멤버만 재정의 가능
- 재정의하는 항목은 override 키워드 표시
- 접근제어지정
  - public/protected/private
  - Internal : 같은 모듈 허용

```
open class SuperClass {
    var memberVar01 = 1
    open var memberVar02 = 2
    fun memberFunc01() {
        println ("super01!")
    }
    open fun memberFunc02() {
        println ("super02!")
    }
}

class SubClass : SuperClass() { // 상위클래스의 생성자 호출 필요
    override var memberVar01 = 2
    override var memberVar02 = 3
    override fun memberFunc01() {
        println("sub01!")
    }
    override fun memberFunc02() {
        println("sub02!")
    }
}
```

• 상위 클래스의 open 키워드를 기록한 멤버만 재정의 가능

유용한 클래스 01

data 클래스

- 상수값을 보관하는 용도로 사용하는 클래스
- VO(Value Object) 또는 DTO(Data Transfer Object)
- toString(), equals() 등이 자동으로 구현됨

```
data class Subject(val title: String, var credit: Int)

fun main() {
    val subject1 = Subject("mobile", 3)
    val subject2 = Subject("mobile", 3)

    println("${subject1}")
    println("${subject1.equals(subject2)}")
}
```

객체가 만들어진 위치 비교

```
Subject(title=mobile, credit=3)
true
```

- Java의 경우 서로 다른 객체이므로 false 반환
- Kotlin의 data 클래스는 멤버변수의 값이 같으면 equals() 가 true 반환

유용한 클래스 02

익명 클래스의 선언

- class 또는 interface 에서 상속받는 클래스의 선언 없이 일시적으로 객체를 만들고자 할 때 사용
- object 키워드 사용

```
open class SuperClass {
    init {
        println ("Super Class")
    }
}

interface SomeInterface {
    val dept : String
    fun getDept()
}
```

• SuperClass를 상속하는 하위클래스 없이 필요한 객체 생성

```
val obj1 = object : SuperClass() {
    var memberValue = 10
    fun memberFunc () {
        println ("Object class!")
    }
}

val obj2 = object : SomeInterface {
    override val dept: String = "computer"
    override fun getDept() {
        println("$dept")
    }
}
```

• interface 에서 필요한 객체 생성

유용한 클래스 03

Companion object

- 클래스의 모든 객체들이 함께 사용하는 멤버 정의
- static 과 유사

```
class MyClass {
    companion object {
        var coData = 10
        fun coFunc () {
            println("Companion!")
        }
    }
}

fun main() {
    println("${MyClass.coData}")
    println("${MyClass.coFunc()}")
```

```
10
Companion!
```

- MyClass로 생성한 모든 객체들이 공용
- static 과 유사한 방식으로 사용

실습 01

- data 클래스 작성

- 가입일을 표현하는 RegDate를 작성
  - year/month/day
  - equals 를 재정의 하여 연도가 같으면 true 로 구현

- 주소를 표현하는 Address 를 작성
  - city/dong/addrNo

실습 02

회원 정보를 표현하는 Member 클래스 작성

- 멤버 변수
  - 성명/가입일(RegDate)/주소(Address)
- 멤버 함수 구현
  - getShippingCost() : seoul 은 3000, 그 외의 지역은 4000 반환
  - getTerm(year : Int, month : Int) : 가입한지 24개월이 넘을 경우 long-term, 아닐 경우 short-term 반환

함수 구현

- Member 를 매개변수로 전달받아 가입한지 몇 일이 지났는지 계산하여 반환 함수 : getPeriod
- 구현 후 람다식으로 표현

main() 함수 구현

- Member 객체 생성
- 가입일과 주소는 자유롭게 지정 후 멤버함수를 호출하여 print
- getPeriod 호출