05

04-b. 확장

다양한 기법

확장의 원리

확장이란?

- 클래스 내부에 선언된 함수와 프로퍼티 이외에 다른 함수나 프로퍼티를 추가
- 일반적으로 클래스 확장은 상속 기법이용

```
open class Super {
    val superData: Int = 10
    fun superFun() {
        println("superFun....")
    }
}

class Sub: Super() {
    val subData: Int = 20
    fun subFun() {
        println("subFun....")
    }
}
```

확장의 원리

확장이란?

- 상속을 받지 않고 이미 선언된 클래스에 함수나 프로퍼티를 추가한후 이용하는 방법

```
class Super {
    val superData: Int = 10
    fun superFun() {
        println("superFun....")
    }
}

val Super.subData: Int
    get() = 20

fun Super.subFun() {
    println("subFun.....")
}

fun main(args: Array<String>) {
    val obj: Super = Super()
    println("superData : ${obj.superData}, subData : ${obj.subData}")
    obj.superFun()
    obj.subFun()
}
```

확장의 원리

정적 등록에 의한 실행

- 확장 함수는 기존 클래스내에 정적으로 추가되지는 않는다.
- 실행 시점에 확장 클래스(Receiver Type) 만 보고 실행

```
class Sub: Super() {
    var data: Int = 20
    override fun superFun() {
        println("Sub .. superFun.... ${data}")
    }
    fun some1(data: Int) {
        this.data=data
        superFun()
        super.superFun()
    }
}
//실행시에 판단하는 것임으로
fun Sub.some2(data: Int){
    this.data = data
    superFun()
    super.superFun()//error
}
fun main(args: Array<String>) {
    val obj: Sub = Sub()
    obj.some1(10)
    obj.some2(100)
}
```

확장의 원리

정적 등록에 의한 실행

- 정적 등록임으로 OOP의 다형성도 불가
- 상속을 통한 다형성을 구현

```
open class Super {
    open fun sayHello(){
        println("Super1.. sayHello()")
    }
}

class Sub : Super() {
    override fun sayHello(){
        println("Sub1.. sayHello()")
    }
}

fun some(obj: Super){
    obj.sayHello()
}
fun main(args: Array<String>) {
    some(Sub())
}
```

실행결과

```
Sub.. sayHello()
```

확장의 원리

정적 등록에 의한 실행

- 확장에 의한 다형성을 구현

```
open class Super

class Sub : Super()

fun Super.sayHello(){
    println("Super..sayHello()")
}
fun Sub.sayHello(){
    println("Sub..sayHello()")
}

fun some(obj: Super){
    obj.sayHello()
}

fun main(args: Array<String>) {
    some(Sub())
}
```

실행결과

```
Super..sayHello()
```

확장의 원리

확장 클래스에 동일 이름의 함수가 있는 경우

```
class Test {
    fun sayHello(){
        println("Test.. sayHello()")
    }
}

fun Test.sayHello(){
    println("Test extension.. sayHello()")
}


fun main(args: Array<String>) {
    val test=Test()
    test.sayHello()
}
```

실행결과

```
Test.. sayHello()
```

프로퍼티와 컴패니언 오브젝트 확장

프로퍼티 확장

- 프로퍼티 확장이 가능
- getter에 의해서만 초기화 가능

```
class Test {
    val classData: Int = 0
}

//val Test.extensionData1: Int = 0//error

val Test.extensionData2: Int
    get() = 10

fun main(args: Array<String>) {
    val obj=Test()
    println("classData ${obj.classData} ... extensionData2 : ${obj.extensionData2}")
}
```

프로퍼티와 컴패니언 오브젝트 확장

컴패니언 오브젝트의 확장

- companion 예약어가 추가되면 Nested 클래스의 맴버를 companion 클래스를 포함하는 클래스명으로 접근이 가능
- Companion Object도 확장이 가능

```
class Test {
    companion object {
        val data1: Int = 10
        fun myFun1(){
            println("companion object myFun1()....")
        }
    }
}

val Test.Companion.data2: Int
    get() = 20

fun Test.Companion.myFun2() {
    println("extension myFun2()....")
}

fun main(args: Array<String>) {
    println("data1 : ${Test.data1} .. data2 : ${Test.data2}")
    Test.myFun1()
    Test.myFun2()
}
```

확장 구문의 위치에 따른 이용

최상위 레벨에 작성

- Top-Level 로 선언한 확장 함수나 프로퍼티를 동일 파일에서 사용하는 것은 문제가 없지만 외부 파일에서 이용할때는 별도로 import를 받아서 사용

```
package sixteen_three_one_one

class Test {
    val data1: Int = 10
}

val Test.data2
    get() = 20

fun main(args: Array<String>) {
    val obj: Test = Test()
    println("data2 : ${obj.data2}")
}
```

```
package sixteen_three_one_two

import sixteen_three_one_one.Test
import sixteen_three_one_one.data2

fun main(args: Array<String>) {
    val obj: Test = Test()
    println("data1 :  ${obj.data1}")
    println("data2 :  ${obj.data2}")
}
```

확장 구문의 위치에 따른 이용

다른 클래스 내에 작성

- 확장 대상이 되는 클래스를 extension receiver 라고 부르고 확장구문이 작성된 클래스를 dispatch receiver 라고 부른다.
- dispatch receiver 내에 선언된 extension receiver의 확장함수는 dispatch receiver와 extension receiver 내의 함수에 모두 접근이 가능
- dispatch receiver 내부에 정의된 확장함수는 dispatch receiver 내부에서만 사용가능

확장 구문의 위치에 따른 이용

다른 클래스 내에 작성

```
class ExtensionClass {
    fun some1() {
        println("ExtensionClass some1()")
    }
}
class DispatchClass {
    fun dispatchFun() {
        println("DispatchClass dispatchFun()")
    }
    fun ExtensionClass.some2() {
        some1()
        dispatchFun()
    }
    fun test() {
        val obj: ExtensionClass = ExtensionClass()
        obj.some1()
        obj.some2()
    }
}
fun main(args: Array<String>) {
    val obj: ExtensionClass = ExtensionClass()
    obj.some1()
    obj.some2()//error
}
```

확장 구문의 위치에 따른 이용

익스텐션 리시버와 디스패치 리시버의 함수명 중복

- extension receiver 와 dispatch receive 에 동일 이름의 함수가 모두 선언된 경우

```
class ExtensionClass {
    fun myFun() {
        println("ExtensionClass myFun()")
    }
}
class DispatchClass {
    fun myFun() {
        println("DispatchClass myFun()")
    }
    fun ExtensionClass.some() {
        myFun()
        this.myFun()
        this@DispatchClass.myFun()
    }
    fun test() {
        val obj: ExtensionClass = ExtensionClass()
        obj.some()
    }
}
fun main(args: Array<String>) {
    val obj: DispatchClass = DispatchClass()
    obj.test()
}
```

실행결과

```
ExtensionClass myFun()
ExtensionClass myFun()
DispatchClass myFun()
```

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare