

03

03-4. 접근제한과 오버라이드

Object Oriented Programming

오버라이드

함수 오버라이드

- 함수를 선언하면 기본으로 final
- final 클래스 : 이 클래스를 상속받아 하위 클래스 작성 금지
- final 함수 : 이 함수를 하위 클래스에서 오버라이드 금지
- final 프로퍼티 : 프로퍼티 오버라이드 금지
- 함수의 오버라이드를 허용하려면 open 예약어로 명시
- override 예약어를 이용하여 이 함수는 상위 함수를 재정의 한것임을 명시적으로 선언

오버라이드

함수 오버라이드

```
open class Shape {  
    //.....  
    open fun print() {  
        println("$name : location : $x, $y")  
    }  
}
```

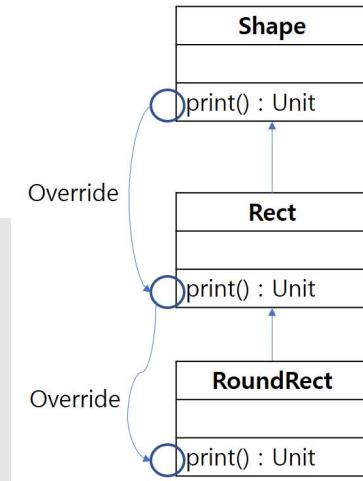
```
class Circle: Shape() {  
    //.....  
    override fun print() {  
        println("$name : location : $x, $y ... radius : $r ")  
    }  
}
```

오버라이드

override 예약어

- override 예약어가 추가되면 해당 함수는 자동으로 open 상태

```
open class Shape {  
    open fun print() {  
        //.....  
    }  
}  
open class Rect: Shape() {  
    override fun print() {  
        //.....  
    }  
}  
class RoundRect: Rect() {  
    override fun print() {  
        //.....  
    }  
}
```



오버라이드

프로퍼티 오버라이드

- 프로퍼티의 오버라이드도 먼저 상위 클래스의 프로퍼티에 open 예약어로 명시해 재정의를 허용해야 합니다.
- 그리고 하위 클래스에서는 override 예약어로 이 프로퍼티는 상위 클래스의 프로퍼티를 재정의한 것이라고 명시해야 합니다.

```
open class Super {  
    open val name: String = "kkang"  
}  
open class Sub: Super() {  
    final override var name: String = "kim"  
}
```

오버라이드

프로퍼티 오버라이드

- 상위 클래스의 프로퍼티와 이름 및 타입이 동일
- 상위에 val로 선언된 프로퍼티는 하위에서 val, var로 재정의 가능
- 상위에서 var로 선언된 프로퍼티는 하위에서 var로 재정의 가능, val은 불가
- 상위에서 Nullable로 선언된 경우 하위에서 Non-Null로 선언 가능
- 상위에서 Non-Null로 선언된 경우 하위에서는 Nullable로 재정의 불가

```
open class Super {  
    open val name: String = "kkang"  
    open var age: Int = 10  
    open val email: String?=null  
    open val address: String="seoul"  
}  
class Sub: Super() {  
    override var name: String = "kim" //ok~~  
    override val age: Int = 20 //error  
    override val email: String = "a@a.com" //ok~~~  
    override val address: String? = null //error  
}
```

오버라이드

상위 클래스 멤버 접근

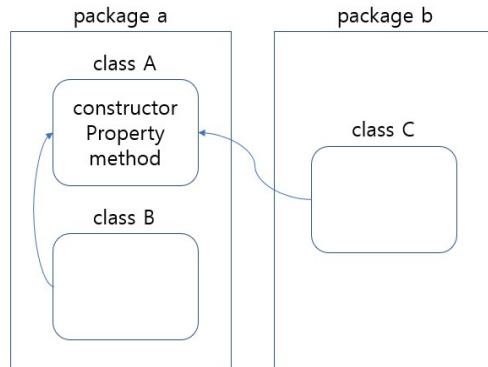
- 재정의한 멤버가 있어도 때로는 상위 클래스에 정의한 멤버도 함께 이용해야 할 때가 있습니다.
- 이럴 때 super를 사용합니다.

```
open class Super {  
    open var x: Int = 10  
    open fun someFun(){  
        println("Super... someFun()")  
    }  
}  
class Sub : Super() {  
    override var x: Int = 20  
    override fun someFun() {  
        super.someFun()  
        println("Sub... ${super.x} .... $x")  
    }  
}  
  
fun main(args: Array<String>) {  
    var sub=Sub()  
    sub.someFun()  
}
```

접근 제한자

접근 제한자란?

- 외부에서 클래스, 생성자, 프로퍼티, 함수 등을 이용할 때 접근의 범위를 지정
- public, internal, protected, private



접근 제한자

Top-Level 구성요소의 접근범위

- public : (Default) 만약 접근제한자가 명시적으로 선언되지 않는다면 자동으로 public이 적용. public은 접근 제한이 없다는 의미. 어느 곳에서나 접근이 가능.
- private : 동일 file 내에서만 접근이 가능.
- internal : 같은 module내에 어디서나 접근이 가능.
- protected : top-level 에서는 사용 불가능.

접근 제한자

Top-Level 구성요소의 접근범위

```
package nine_five_two
val myData1: Int = 10
private val myData2: String = "hello"
class MyClass1()
private class MyClass2()
fun myFun1() {
    println("myFun() call..")
}
private fun myFun2(){
    println("myFun() call..")
}
fun main(args: Array<String>) {
    println("$myData1 .. ")
    println("$myData2 .. ")
    val obj1=MyClass1()
    val obj2=MyClass2()
    myFun1()
    myFun2()
}
```

```
package nine_five_two
fun main(args: Array<String>) {
    println("$myData1 .. ")
    println("$myData2 .. ")//error
    val obj1=MyClass1()
    val obj2=MyClass2()//error
    myFun1()
    myFun2()//error
}
```

접근 제한자

클래스 멤버의 접근범위

- public : public : (Default) 만약 접근제한자가 명시적으로 선언되지 않는다면 자동으로 public이 적용. public은 접근제한이 없다는 의미. 어느 곳에서나 접근이 가능.
- private : 동일 클래스내에서만 접근가능.
- protected : private + 서브 클래스에서 사용가능
- internal : 같은 모듈에 선언된 클래스에서 사용가능

```
open class Super {  
    val publicData: Int = 10  
    protected val protectedData: Int = 10  
    private val privateData: Int = 10  
}  
class Sub: Super() {  
    fun visibilityTest() {  
        println("$publicData .. ")  
        println("$protectedData .. ")  
        println("$privateData .. ")//error  
    }  
}  
  
class SomeClass {  
    fun visibilityTest() {  
        val obj=Super()  
        println("${obj.publicData} .. ")  
        println("${obj.protectedData} .. ")//error  
        println("${obj.privateData} .. ")//error  
    }  
}
```

접근 제한자

프로퍼티와 접근 제한자

- 프로퍼티란 변수처럼 이용되지만 실제로는 getter/setter를 포함하는 변수라는 의미입니다.
- 어떤 프로퍼티가 있는데 때로는 외부에서 데이터를 사용할 수는 있어도, 변경하지는 못하게 만들어야 할 때가 있습니다

```
class PropertyVisibilityTest {  
    private var data: Int = 10  
  
    fun getData(): Int {  
        return data  
    }  
}
```

```
class PropertyVisibilityTest2 {  
    var data: Int = 10  
    private set(value) {  
        field=value  
    }  
  
    fun main(args: Array<String>) {  
        val obj2=PropertyVisibilityTest2()  
        println("${obj2.data}")  
        obj2.data=20//error  
    }  
}
```

접근 제한자

프로퍼티와 접근 제한자

- `get()` 의 경우 프로퍼티의 접근제한자와 항상 동일한 접근제한자가 적용된다.
- `set()` 의 경우 프로퍼티의 접근제한자와 다른 접근제한자 설정이 가능하지만 범위를 넓혀서 설정할수는 없다.
- `public` 프로퍼티 → `set(): public, protected, internal, private`
- `protected` 프로퍼티 → `set(): protected, private`
- `internal` 프로퍼티 → `set(): internal, private`
- `private` 프로퍼티 → `set(): private`

접근 제한자

생성자와 접근제한

- 코틀린의 생성자는 주 생성자와 보조 생성자로 구분되며 주 생성자와 보조 생성자 모두 접근 제한자를 지정 할 수 있습니다.
- 주 생성자는 constructor를 생략할 수 있지만, 접근 제한자를 지정하려면 constructor 키워드를 함께 써줘야 합니다.

```
class ConstructorVisibilityTest private constructor(name: String) {  
    public constructor(name: String, no: Int): this(name){ }  
}
```

접근 제한자

상속 관계와 접근제한자

- open 과 private은 같이 사용할수 없다.
- 하위 클래스에서 상위 맴버를 오버라이드 받을 때 접근 범위를 줄일수는 없다.

```
open class Super1 {  
    open private fun myFun1() { //error  
  
    }  
    open fun myFun2() {  
  
    }  
    open protected fun myFun3() {  
  
    }  
  
}  
  
class Sub1: Super1() {  
    override private fun myFun2() { //error  
  
    }  
  
    override fun myFun3() {  
  
    }  
}
```



감사합니다

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



윌리엄 셰익스피어

William Shakespeare