Kotlin : 클래스

Mobile Software

2021 Fall

What to do next?

- 클래스
- 상속
- 추상 클래스와 인터페이스

- Kotlin 강의 노트에서는
  - 소스 코드를 별도로 제공하지 않음
  - 실행 결과를 포함하지 않음.
  - 직접 코드를 입력하고, 강의 노트 설명을 확인해야 함.
  - Required assignments
    - 강의 노트에 해당하는 강의가 종료되면, 해당 주 일요일 23:59:59까지 kotlin 실습 파일 제출 ➔ 평소 점수에 반영

Class

- **클래스 선언**
  - 클래스는 본체 없이도 이름만으로 선언할 수 있음

```
class Person  { }   // 본문 내용이 없는 상태에서 클래스 선언
class Person2       // 중괄호 생략 가능
```

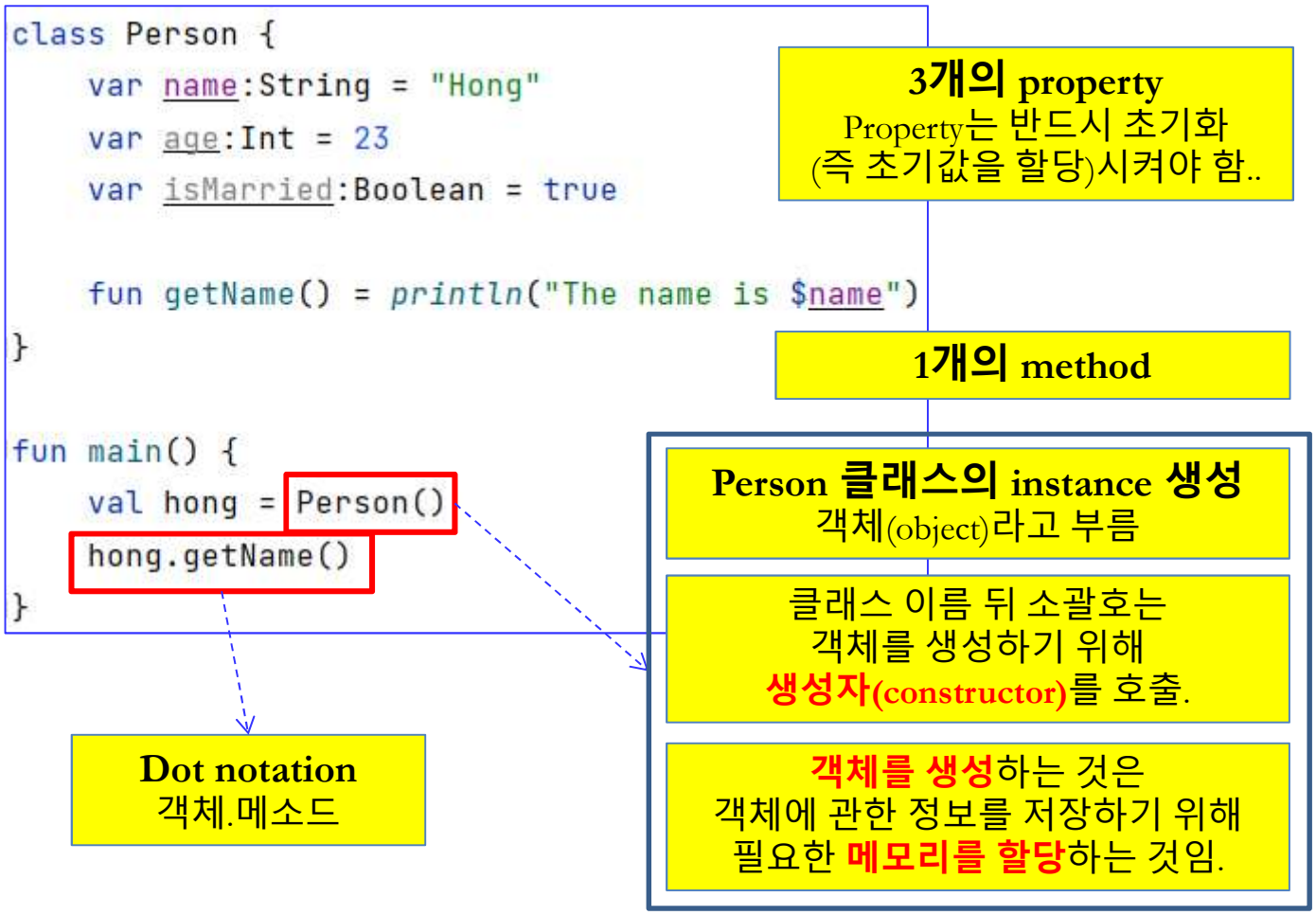
    - Visibility modifier를 생략하면 **public**
    - visibility modifier: **public, private, protected, internal**

- **객체(object) = 클래스의 instance**
  - 클래스로부터 객체를 생성 ➔ 메모리가 할당됨
    - 객체를 생성할 때 **키워드 new를 사용하지 않음**

```
val Hong = Person() // an object 'Hong' created from the class 'Person'
```

    - 인터페이스도 클래스와 동일한 규칙이 적용됨

class Person



class Person with secondary constructor

클래스 정의

```
class Person {
    var name:String
    var age:Int
    var isMarried:Boolean

    constructor(name:String, age:Int, isMarried:Boolean) {
        this.name = name
        this.age = age
        this.isMarried = isMarried
    }

    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true)
    println("The age is ${hong.age}")
    hong.getName()
}
```

3개의 property
선언만 하고 초기화하지 않음

this
class Person을 가리킴

Secondary constructor
보조 생성자 – property 초기화

1개의 method

객체 생성

What's the difference?

```
class Person {
    var name:String
    var age:Int
    var isMarried:Boolean

    constructor(_name:String, _age:Int, _isMarried:Boolean) {
        name = _name
        age = _age
        isMarried = _isMarried
    }

    fun getName() = println("The name is $name")
}

fun main() {
    val hong = Person("Hong", 23, true)
    println("The age is ${hong.age}")
    hong.getName()
}
```

생성자의 parameter 앞에
underscore(_)를 붙여
class Person의 property와 구분.
➔ this를 사용하지 않아도 됨.

class Person with two secondary constructors

```
class Person {
    var name:String
    var age:Int
    var isMarried:Boolean

    constructor(_name:String, _age:Int, _isMarried:Boolean) {
        name = _name
        age = _age
        isMarried = _isMarried
    }

    constructor(_name:String, _age:Int) {
        name = _name
        age = _age
        isMarried = true // default
    }

    fun getName() = println("The name is $name")
}
```

```
fun main() {
    val kim = Person("Hora", 37)
    println("The age is ${kim.age}")
    if (kim.isMarried) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

Another secondary constructor는
2개의 parameter만을 전달받아
2개 property를 초기화.
남은 1개의 property는 default 값을 할당.

class Person with primary constructor

```
class Person constructor(var name:String,
                         var age:Int,
                         var isMarried:Boolean) {
    fun getName() = println("the name is $name")
}

fun main() {
    val kim = Person("Hora", 37, false)
    kim.getName()
    println("${kim.age}")
    if (kim.isMarried) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

Primary constructor
(주 생성자)

Primary constructor의 parameter가
클래스의 property.

객체가 생성될 때 Primary constructor 를 호출
➔parameter 순서대로
클래스의 property 초기값이 할당됨.

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean)
```

Primary constructor의 경우
키워드 constructor 생략할 수 있음.

Primary constructor with init block

Primary constructor는 property를 초기화하는 역할.
Property 초기화가 아닌 다른 작업을 위한 코드를 추가하려면 **init** 블록이 필요!

```
class Person (var name:String,
              var age:Int,
              var isMarried:Boolean){
    init {
        println("Beginning of init block")
        println("이름=$name, 나이=$age")
        println("End of init block")
    }
    fun getName() = println("The name is $name")
}

fun main() {
    val kim = Person("Hora", 37, false)
    println("The name is ${kim.age}")
    kim.getName()
}
```

Primary constructor와 secondary constructor를 함께 사용

```
class Person (var name:String, var age:Int, var isMarried:Boolean){
    var nickname:String = ""
    init {
        println("이름=$name, 나이=$age")
    }
    constructor(_name:String, _age:Int, _isMarried:Boolean, _nickname:String)
        :this(_name, _age, _isMarried) {
        nickname = _nickname
    }
    fun getName() = println("The name is $name")
}

fun main() {
    val kim = Person("Hora", 37, false, "Chic")
    println("The nickname is ${kim.nickname}")
    kim.getName()
}
```

The body of the secondary constructor is called after the **init** block.

Primary constructor에서 초기화한 클래스의 property.

Class의 Property (1/2)

- **Property** : 클래스의 멤버 변수
  - 값 또는 상태를 저장할 수 있는 필드(field)
  - Getter와 Setter 메서드를 자동 생성
    - **val**로 선언한 property ➔ Getter (읽어올 수 있음)
    - **var**로 선언한 property ➔ Getter와 Setter (읽어오거나 변경할 수 있음)
  - 자신이 원하는 getter 또는 setter를 정의.

```
class Rectangle(val shape:String, var height:Int, var width:Int)

fun main() {
    val rect = Rectangle("Rectangle", 30, 30)
    rect.height = 40
    rect.width = 40
    println("${rect.shape}, ${rect.width}, ${rect.height}")
}
```

Setter를 사용한 property 값 변경

Getter를 사용한 property 값 읽어 오기

Class의 Property (2/2)

- **Property** : 클래스의 멤버 변수
  - 자신이 원하는 getter 또는 setter를 정의.

```
class Rectangle(_shape:String, _height:Int, _width:Int) {
    val shape: String = _shape
        get() = field
    var height: Int = _height
        get() = field
        set(value) {
            field = value
        }
    var width: Int = _width
        get() = field
        set(value) {
            field = value
        }
}
```

Getter 정의

field : property를 참조하는 변수

value : Setter의 parameter
(인자로 전달되는 값)

Getter와 Setter 정의

Use with statement

```
class Person {
    var name:String = ""
    var age:Int = -1
    var isMarried:Boolean = false
}

fun main() {
    val hong = Person()
    hong.name = "Hong"
    hong.age = 23
    hong.isMarried = true
}
```

```
val hong = Person()
with (hong) {
    name = "Hong"
    age = 23
    isMarried = true
}
with (hong) {
    println("name = $name, age = $age")
}
println("name = ${hong.name}, age = ${hong.age}")
```

```
hong.apply {
    name = "Hong"
    age = 23
    isMarried = true
    println("name = $name, age = $age")
}
```

What to do next?

- 클래스
- **상속**
- 추상 클래스와 인터페이스

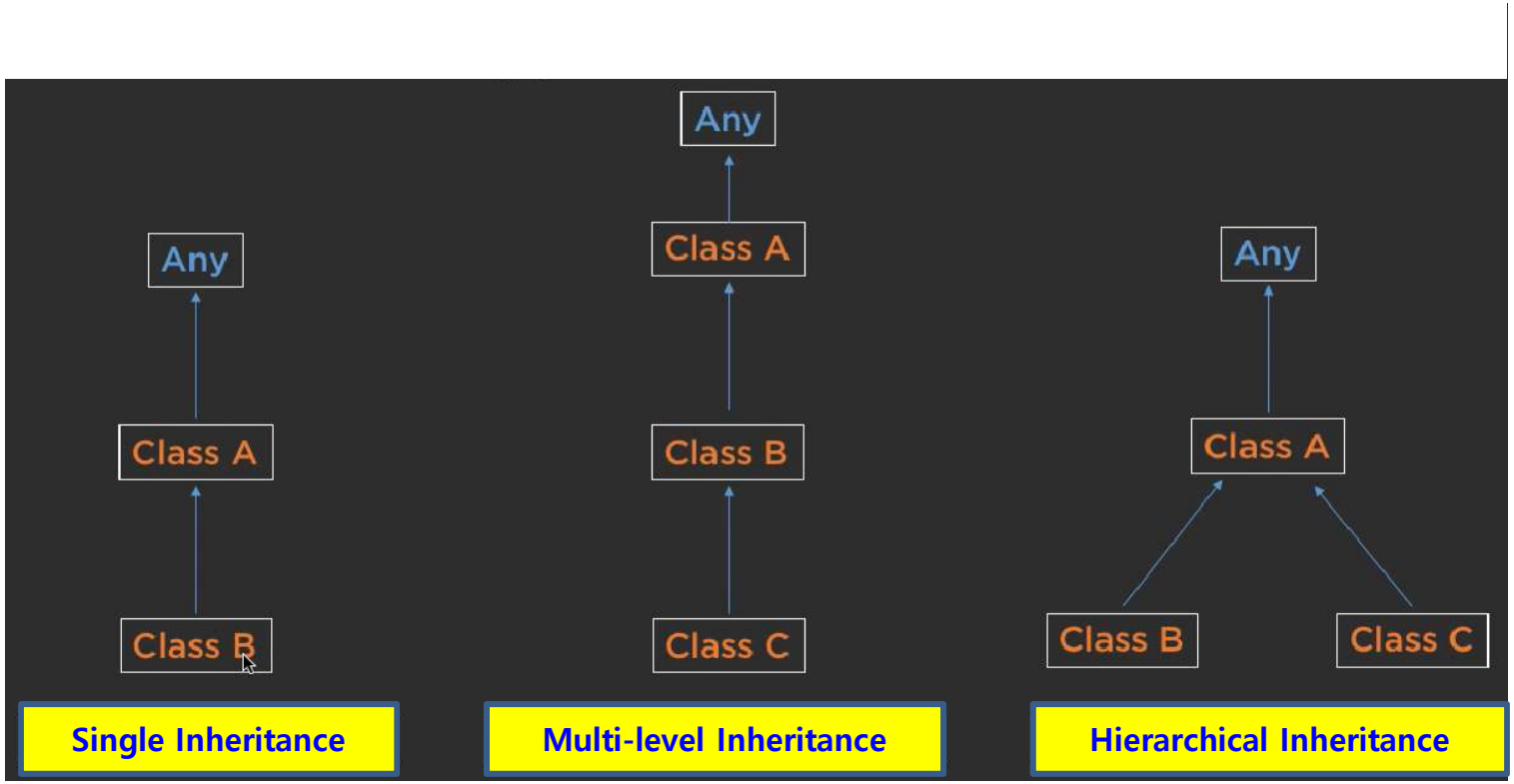
Inheritance (상속)

- By default Classes are:
  - **public**
  - **final**

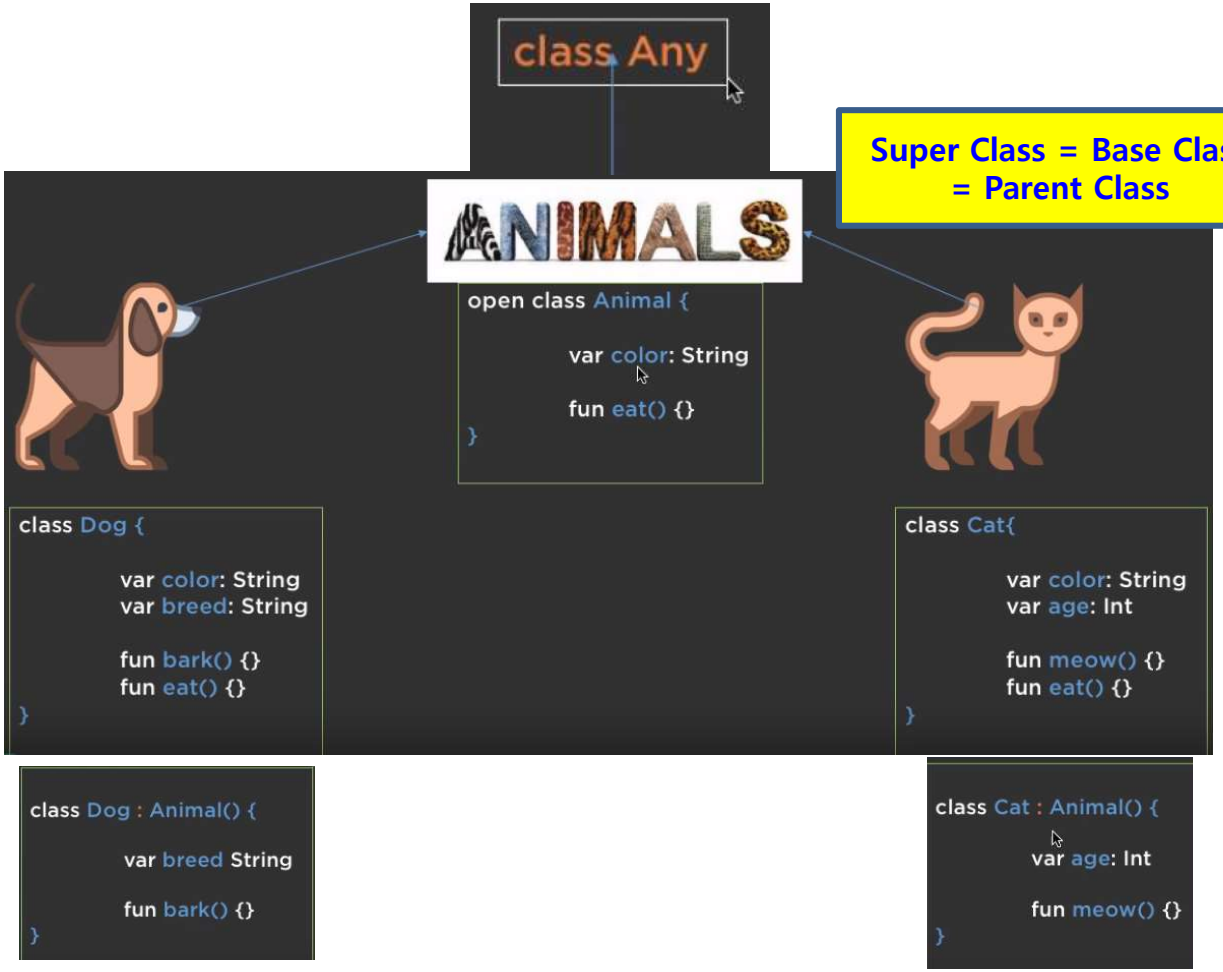
- For inheritance
  - You need to make a class '**open**'
    - A Child object acquires all the properties from its Parent class object.

- Advantages
  - For code reusability
  - For method overriding

Types of Inheritance



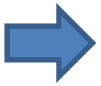
Inheritance 예(1/2)



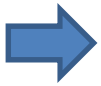
Inheritance 예(2/2)

```
class Dog {
    var color: String = ""
    var breed: String = ""

    fun bark() {
        println("Bark")
    }

    fun eat() {
        println("Eat")
    }
}
```

```
class Cat {
    var color:String = ""
    var age:Int = -1
    fun meow() {
        println("Meow")
    }
    fun eat() {
        println("Eat")
    }
}
```



```
open class Animal {
    var color: String = ""

    fun eat() {
        println("Eat")
    }
}

class Dog: Animal() {
    var breed: String = ""

    fun bark() {
        println("Bark")
    }
}

class Cat: Animal() {
    var age: Int = -1

    fun meow() {
        println("Meow")
    }
}
```



```
fun main() {
    var dog = Dog()
    with (dog) {
        breed = "labra"
        color = "black"
        bark()
        eat()
    }
    var cat = Cat()
    with (cat) {
        age = 3
        color = "white"
        meow()
        eat()
    }
}
```

Overriding Properties and Methods (1/2)

```
open class Animal {
    var color: String = ""

    open fun eat() {
        println("An animal eats food")
    }
}
```

```
class Dog: Animal() {
    var breed: String = ""

    fun bark() {
        println("Bark")
    }

    override fun eat() {
        println("A dog eats food.")
    }
}
```

```
class Cat: Animal() {
    var age: Int = -1

    fun meow() {
        println("Meow")
    }

    override fun eat() {
        println("A cat eats food.")
    }
}
```

```
fun main() {
    var dog = Dog()
    dog.eat()

    var cat = Cat()
    cat.eat()
}
```

Overriding Properties and Methods (2/2)

```
open class Animal {
    open var color: String = "white"

    open fun eat() {
        println("An animal eats food")
    }
}
```

```
class Dog: Animal() {
    var breed: String = ""

    override var color: String = "black"
    fun bark() {
        println("Bark")
    }

    override fun eat() {
        super.eat()
        println("A dog eats food.")
    }
}
```

```
fun main() {
    var dog = Dog()
    println("color = ${dog.color}")
    dog.eat()

    var cat = Cat()
    println("color = ${cat.color}")
}
```

super : 상위 클래스(super class)
super ( ): 상위 클래스 생성자
super . eat( ): 상위 클래스의 eat( ) 메소드

With Primary Constructors

```
open class Animal {
    open var color: String = ""
}

class Dog: Animal() {
    var breed: String = ""
}

fun main() {
    var dog = Dog()
    dog.color = "Black"
    dog.breed = "Pug"
}
```

```
open class Animal(var color:String) {
    init {
        println("at init of Animal class: $color")
    }
}

class Dog(color:String, var breed:String)
    : Animal(color) {
    init {
        println("at init of Dog class: $breed")
    }
}

fun main() {
    var dog = Dog("black", "pug")
}
```

super class의
primary constructor

Quiz : 어떤 값이 출력될까요?

```
open class Animal (open var color:String) {
    init {
        println("at init of the Animal class: $color")
    }
}

class Dog(override var color:String, var breed:String) : Animal(color) {
    init {
        println("at init of the Dog class: $breed")
    }
}

fun main() {
    val animal = Animal("white")
    val dog = Dog("black", "pug")
    println("${animal.color}")
    println("${dog.color}, ${dog.breed}")
}
```

With Secondary Constructors

```
open class Animal() {
    var color:String = ""
    constructor(color: String): this() {
        this.color = color
    }
}

class Dog : Animal {
    var breed:String = ""
    constructor(color: String, breed:String): super(color) {
        this.breed = breed
    }
}

fun main() {
    var anim = Animal()
    println("color = ${anim.color}")

    var dog = Dog("black", "pug")
    println("color = ${dog.color}, breed = ${dog.breed}")
}
```

default constructor.
= property의 초기값 설정

super class의
secondary constructor

Polymorphism(다형성) : Overriding 과 Overloading

Overriding : method나 property의 이름은 같지만 동작이나 값을 재정의.
Overloading : 동작은 같지만 parameter의 type이나 개수가 다름.

```
class Calc {
    fun add(x:Int, y:Int):Int = x + y
    fun add(x:Float, y:Float):Float = x + y
    fun add(x:Float, y:Float, z:Float):Float = x + y + z
    fun add(x:Double, y:Double):Double = x + y
    fun add(x:String, y:String):String = x + y
}

fun main() {
    val calc = Calc()
    println(calc.add(2, 3))
    println(calc.add(4.1f, 3.5f))
    println(calc.add(4.1f, 3.5f, 6.9f))
    println(calc.add(4.0, 5.0))
    println(calc.add("Hello, ", "Kotlin"))
}
```

가시성 수식어 (visibility modifiers)

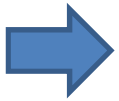
- **Visibility** (가시성, 접근 제한)
  - 클래스의 method나 property의 접근 권한을 지정
    - 정보 은닉 (information hiding)
      - 클래스 외부에서 필요한 부분은 개방(public).
      - 클래스 내부에서만 접근 가능하고 외부 접근을 차단(private).
- **Visibility modifier**
  - **public (+)**: 모두에게 공개(**default**)
  - **protected (#)**: 상속받은 클래스에서는 접근 가능(accessible)
  - **internal** : 같은 모듈 내에서 접근 가능.
  - **private (-)**: 외부 접근 불가.
    - **괄호 안 기호는 UML에서 사용**

```
class Foo {

    val a = 1
    protected val b = 2
    private val c = 3

    internal val d = 4
}
```

UML : Unified Modeling Language

Visibility Modifiers 예

```
open class Base {
    private var a = 1
    protected fun baseFunc() {
        a += 1
        println("a is $a")
    }
}

class Derived : Base() {
    fun deriveFunc() {
        super.baseFunc()
    }
}
```



```
fun main() {
    val derived = Derived()
    derived.deriveFunc()
}
```

상속받은 클래스에서는
protected로 선언된
method를 access할 수 있음.

super 클래스의
private로 선언된
property의 값을 변경

What to do next?

- 클래스
- 상속
- 추상 클래스와 인터페이스

추상 클래스 (Abstract Class)

- **Classes can be abstract in nature**.
  - The role of abstract class is to just provide a set of method and properties.

- **Abstract class is a partially defined class**.
  - Abstract methods **have no body** when declared.

abstract class : abstract method (1/2)

- **추상 클래스**는 상속 관계에 있는 하위 클래스에서
  - 추상 메소드를 overriding 해서 구현

```
class Foo

fun main() {
    val foo:Foo = Foo()
}
```

You cannot create an instance of an abstract class without implementing the abstract methods of the abstract class.

```
abstract class Foo {
    abstract fun bar()
}

class Anonymous : Foo() {
    override fun bar() {
        println("bar() is implemented.")
    }
}

fun main() {
    val foo = Anonymous()
    foo.bar()
}
```

추상 클래스 Foo 정의

추상 메소드 bar( ) 선언

추상 클래스를 상속받은 클래스 정의

Super class의 추상 메소드를 override해서 구현

abstract class : abstract method (2/2)

- **추상 클래스**를 상속받은 하위 클래스의 instance 를 여러 번 사용하지 않고 한 번만 사용할 거라면
  - **object** 를 사용하여 무명 클래스의 객체를 생성하는 방식이 효과적.

```
abstract class Foo {
    abstract fun bar()
}
```

```
class Anonymous : Foo() {
    override fun bar() {
        println("bar() is implemented.")
    }
}

fun main() {
    val foo = Anonymous()
    foo.bar()
}
```

```
fun main() {
    val foo = object : Foo() {
        override fun bar() {
            println("bar() is implemented.")
        }
    }
    foo.bar()
}
```

추상 클래스를 상속받은 무명 클래스의 객체 foo 생성

Super class의 추상 메소드를 override해서 구현

object : 무명(anonymous) 클래스

abstract class: abstract property

```
abstract class Foo {
    abstract var name:String
    abstract fun bar()
    open fun openFunction() {}
    fun publicFunction() {}
}

class Anonymous : Foo() {
    override var name:String = "new name"
    override fun bar() {
        println("bar() is implemented.")
    }
}

fun main() {
    val foo = Anonymous()
    foo.bar()
    print(foo.name)
}
```

추상 클래스에서 abstract property는 초기값을 할당하지 않은 상태로 선언

상속을 허용한 method.

상속을 허용하지 않은 method.

상속받은 하위 클래스에서 abstract property의 초기값 할당

Interface

- **Interface 와 abstract class**
  - **같은 점**
    - Interface can contain both normal methods and abstract methods.
  - **다른 점**
    - Interface contain **only abstract method(s)**.

- **Interface를 사용하는 목적**
  - **구현 상속**
    - 인터페이스가 바뀌어도 이를 구현하는 하위 클래스는 클래스 상속과 달리 영향을 거의 받지 않음
  - **다중 상속**
    - 하위 클래스가 여러 개의 인터페이스를 상속받을 수 있음.
    - 클래스는 단일 상속만 가능. 클래스 상속은 1:1 관계.

Interface 예 (1/2)

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable")
}

class Button : Clickable {
    override fun click() = println("I was clicked!")
}

fun main() {
    val button = Button()
    button.click()
    button.showOff()
}
```

abstract method
abstract 키워드 없이
Method 만 선언

Interface는 클래스가 아님
따라서 구현 상속받을 때
생성자를 호출하는 괄호가 없음.

Interface 예 (2/2)

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable")
}

interface Focusable {
    fun setFocus(b: Boolean) =
        println("I ${if (b) "got" else "lost"} focus.")
    fun showOff() = println("I'm focusable")
}

class Button : Clickable, Focusable {
    override fun click() = println("I was clicked!")

    override fun showOff() {
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

다중 상속

2개의 interface가 공통으로 **showOff()** 메소드를 갖고 있기 때문에 구분이 필요

```
fun main() {
    val button = Button()

    button.showOff()
    button.setFocus(true)
    button.click()
}
```