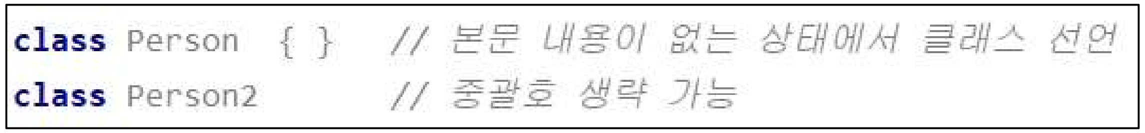
Kotlin : 클래스

Mobile Software
2019 Fall
인천대학교 컴퓨터공학부
홍 윤식 교수

Class

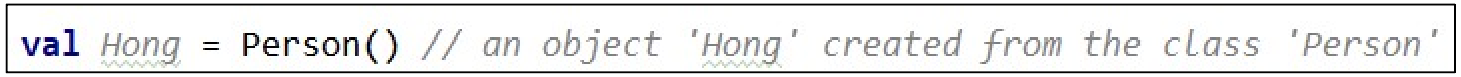
- **클래스 선언**
  - 클래스는 본체 없이도 이름만으로 선언할 수 있음



```
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**

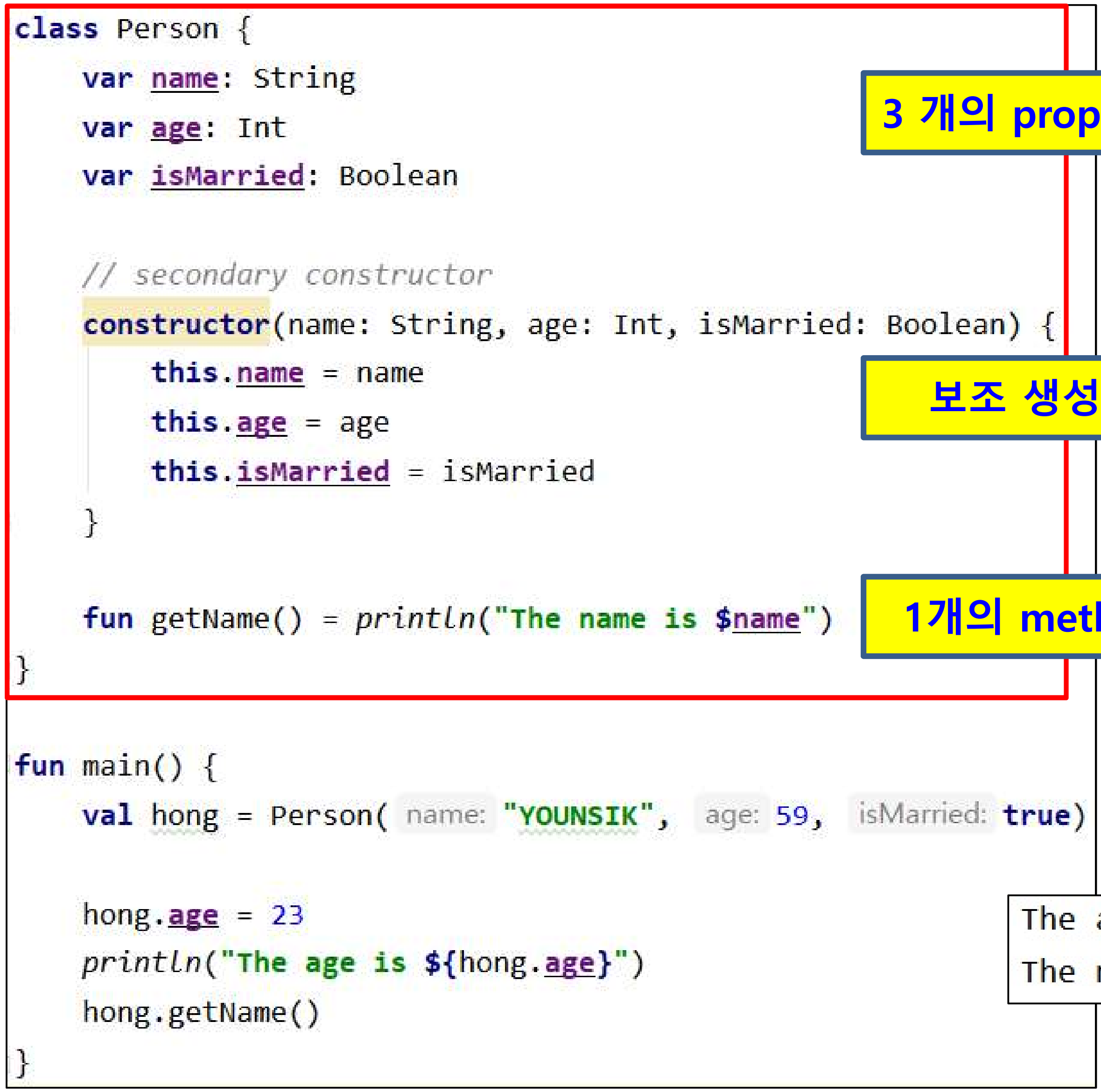


```
val Hong = Person() // an object 'Hong' created from the class 'Person'
```

    - 인터페이스도 클래스와 동일한 규칙이 적용됨

Class Person with secondary constructor

클래스 정의



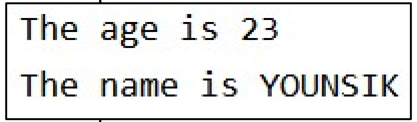
```kotlin
class Person {
    var name: String
    var age: Int
    var isMarried: Boolean

    // secondary constructor
    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( name: "YOUNSIK",  age: 59,  isMarried: true)

    hong.age = 23
    println("The age is ${hong.age}")
    hong.getName()
}
```

3 개의 property

보조 생성자

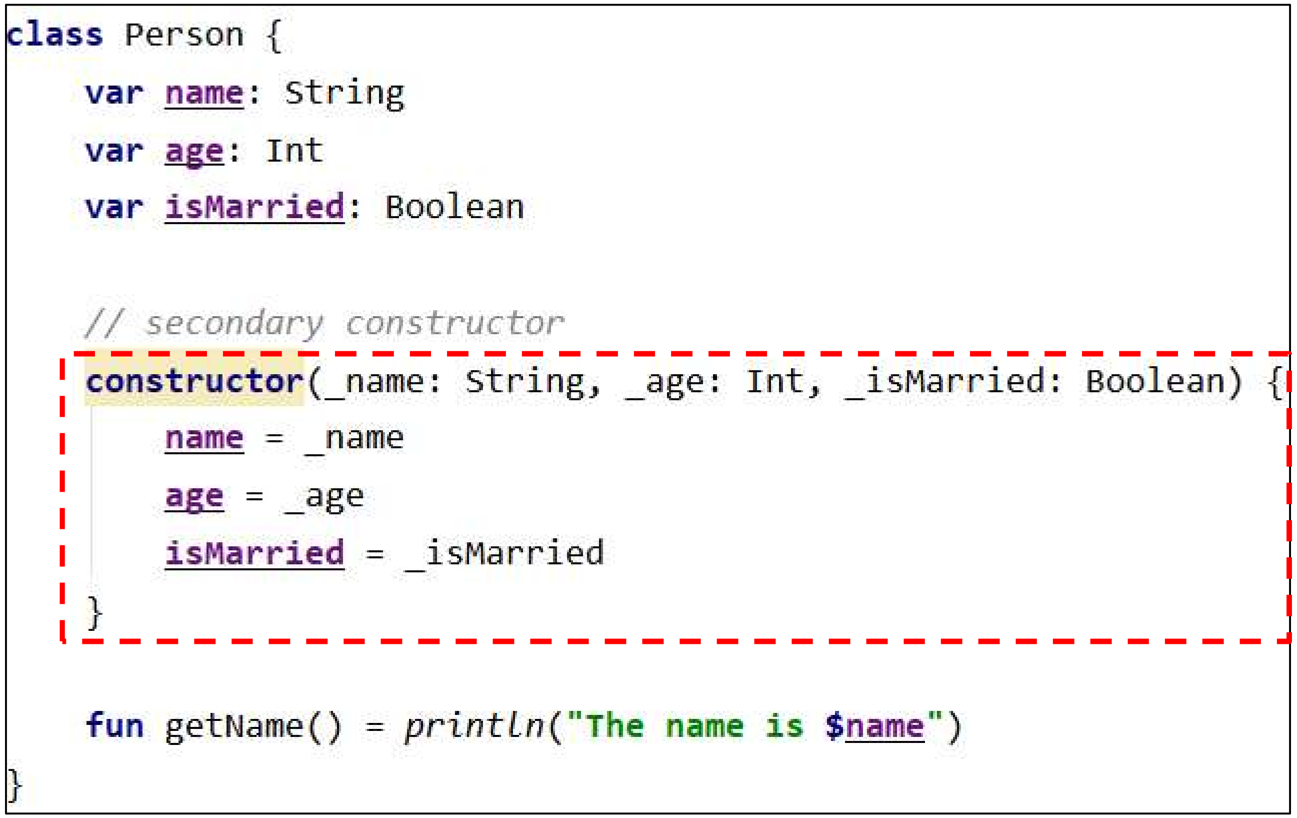
1개의 method

객체 생성



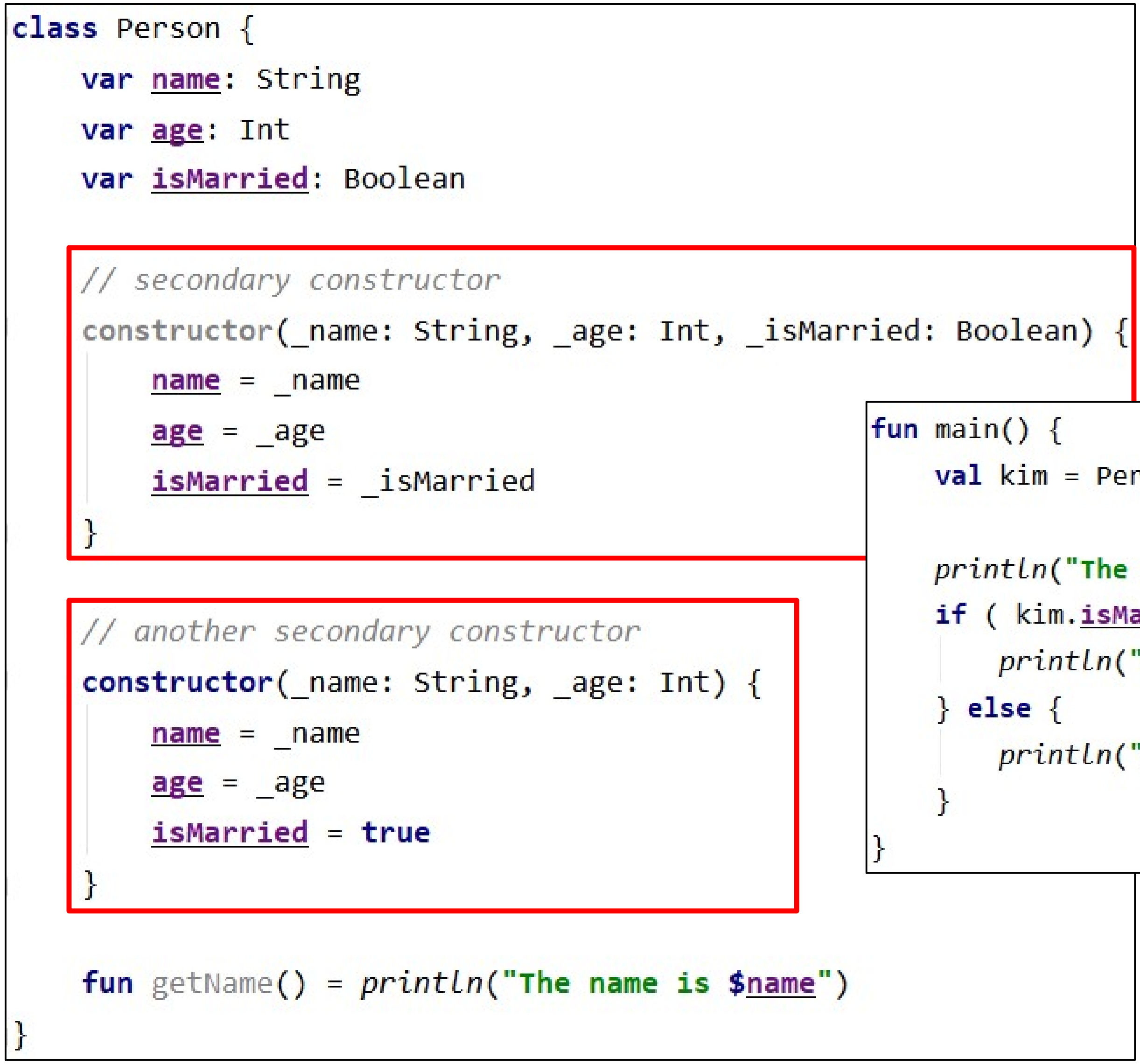
```
The age is 23
The name is YOUNSIK
```

What's the difference?

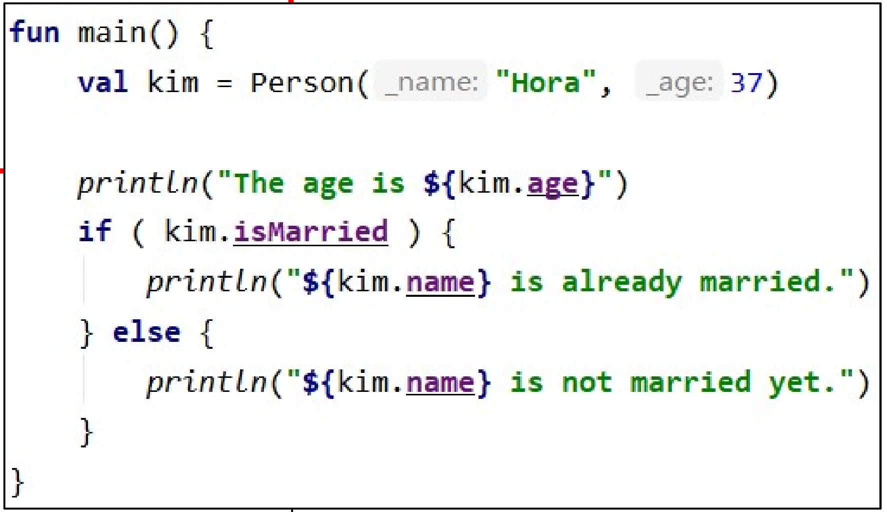


```
class Person {
    var name: String
    var age: Int
    var isMarried: Boolean

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

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

Class Person with two secondary constructors

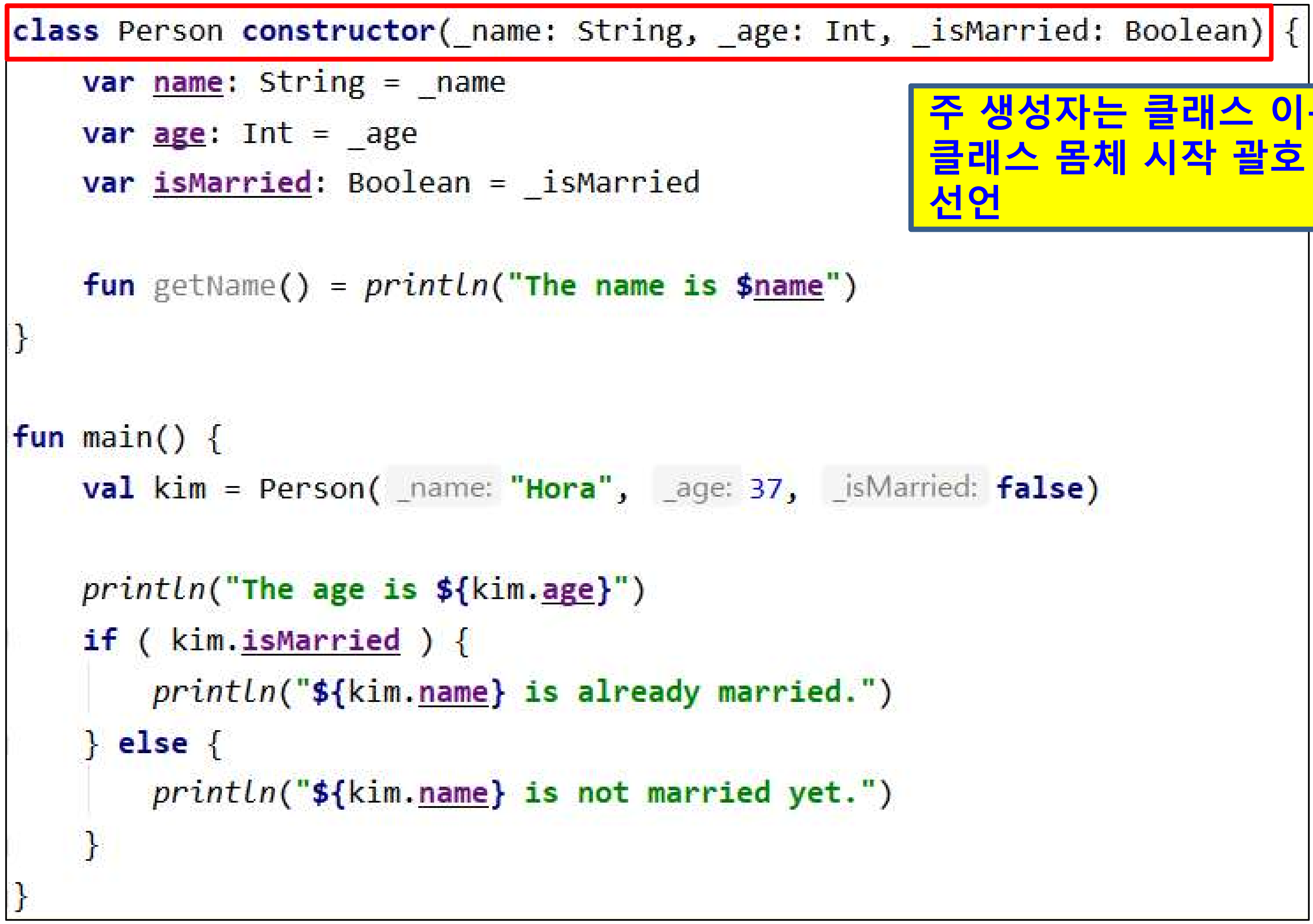


```kotlin
class Person {
    var name: String
    var age: Int
    var isMarried: Boolean

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

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

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



```kotlin
fun main() {
    val kim = Person( _name: "Hora", _age: 37)

    println("The age is ${kim.age}")
    if ( kim.isMarried ) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

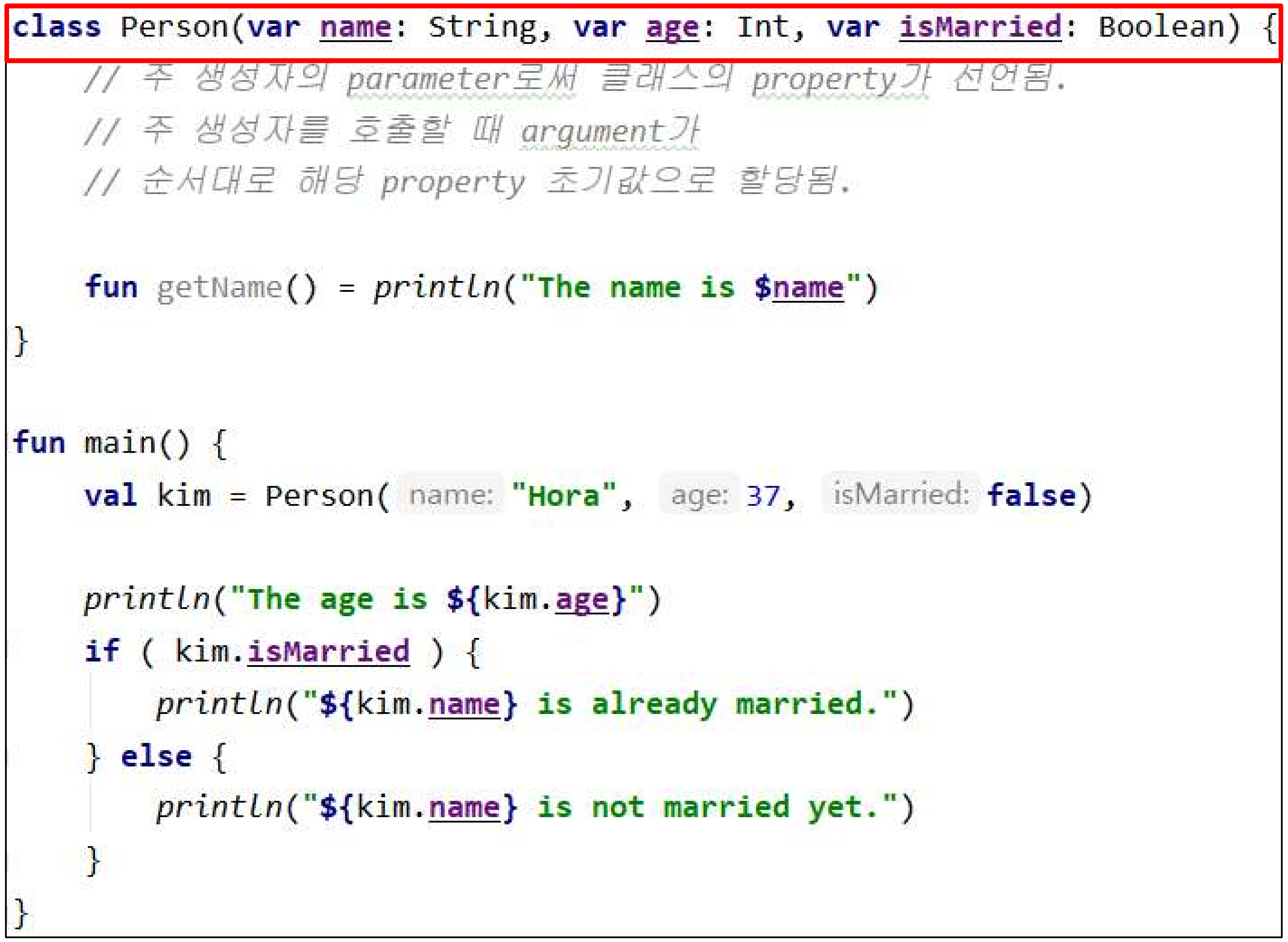
Class Person with primary constructor



```
class Person constructor(_name: String, _age: Int, _isMarried: Boolean) {
    var name: String = _name
    var age: Int = _age
    var isMarried: Boolean = _isMarried

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

fun main() {
    val kim = Person( _name: "Hora", _age: 37, _isMarried: false)

    println("The age is ${kim.age}")
    if ( kim.isMarried ) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

주 생성자는 클래스 이름과 클래스 몸체 시작 괄호 사이에 선언

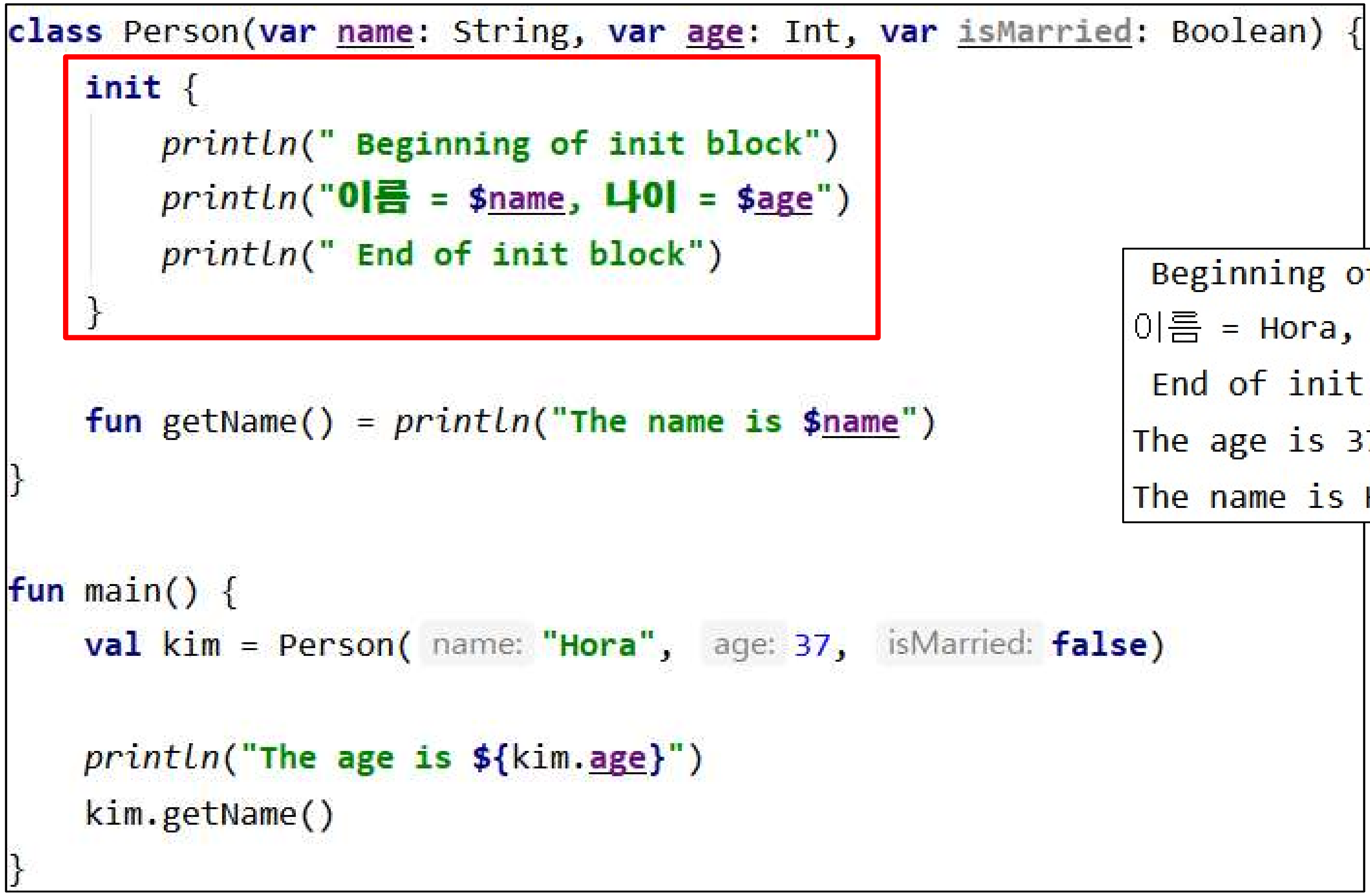
What’s the difference?



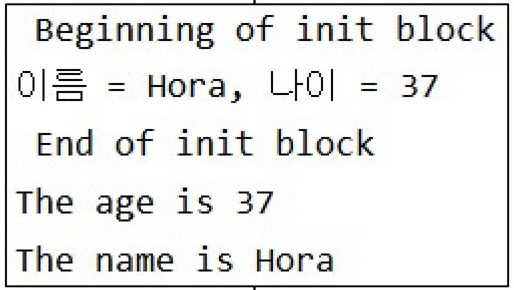
```
class Person(var name: String, var age: Int, var isMarried: Boolean) {
    // 주 생성자의 parameter로써 클래스의 property가 선언됨.
    // 주 생성자를 호출할 때 argument가
    // 순서대로 해당 property 초기값으로 할당됨.

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

fun main() {
    val kim = Person( name: "Hora",  age: 37,  isMarried: false)

    println("The age is ${kim.age}")
    if ( kim.isMarried ) {
        println("${kim.name} is already married.")
    } else {
        println("${kim.name} is not married yet.")
    }
}
```

Primary constructor with init block

주 생성자는 property를 초기화하는 역할.
Property 초기화가 아닌 **다른 작업을 위한 코드를 추가하려면 init block이 필요!**

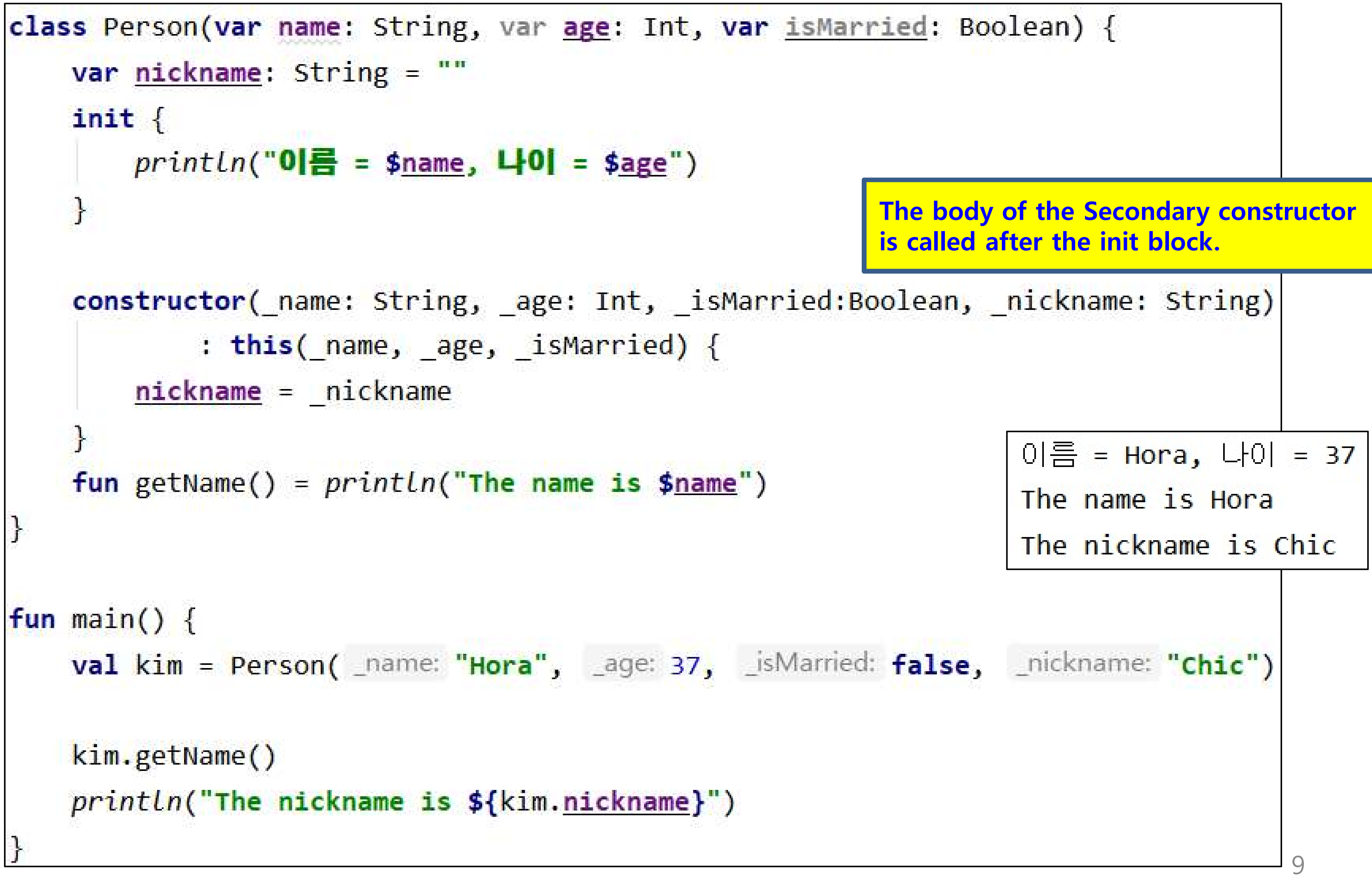


```kotlin
class Person(var name: String, var age: Int, var isMarried: Boolean) {
    init {
        println(" Beginning of init block")
        println("이름 = $name, 나이 = $age")
        println(" End of init block")
    }

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

fun main() {
    val kim = Person( name: "Hora",  age: 37,  isMarried: false)

    println("The age is ${kim.age}")
    kim.getName()
}
```



```
 Beginning of init block
이름 = Hora, 나이 = 37
 End of init block
The age is 37
The name is Hora
```

Primary constructor와 secondary constructor를 함께 사용



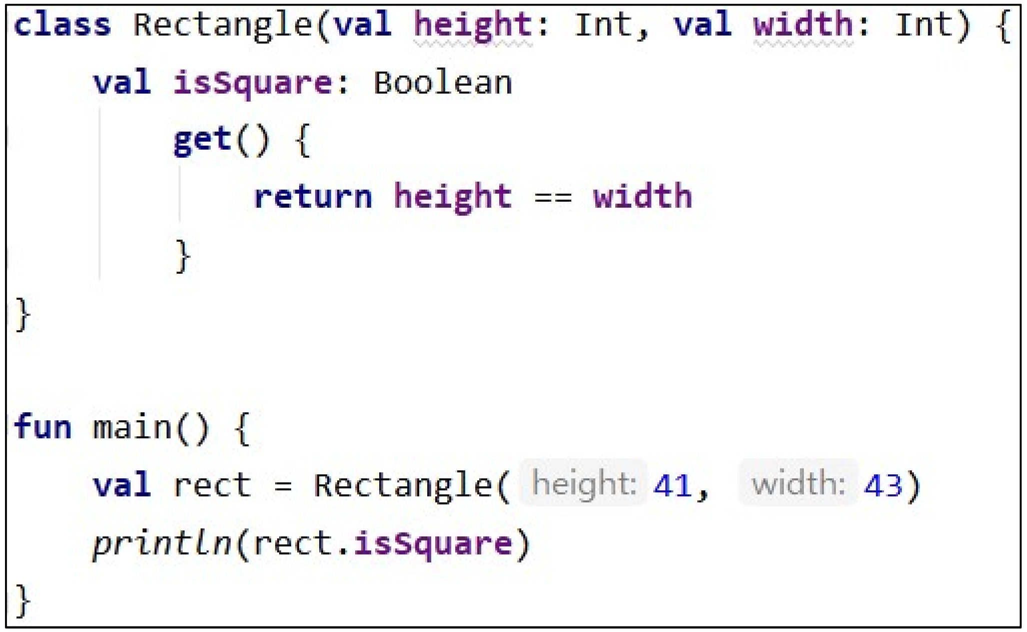
```kotlin
class Person(var name: String, var age: Int, var isMarried: Boolean) {
    var nickname: String = ""
    init {
        println("이름 = $name, 나이 = $age")
    }

    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( _name: "Hora", _age: 37, _isMarried: false, _nickname: "Chic")

    kim.getName()
    println("The nickname is ${kim.nickname}")
}
```

```
이름 = Hora, 나이 = 37
The name is Hora
The nickname is Chic
```

The body of the Secondary constructor is called after the init block.

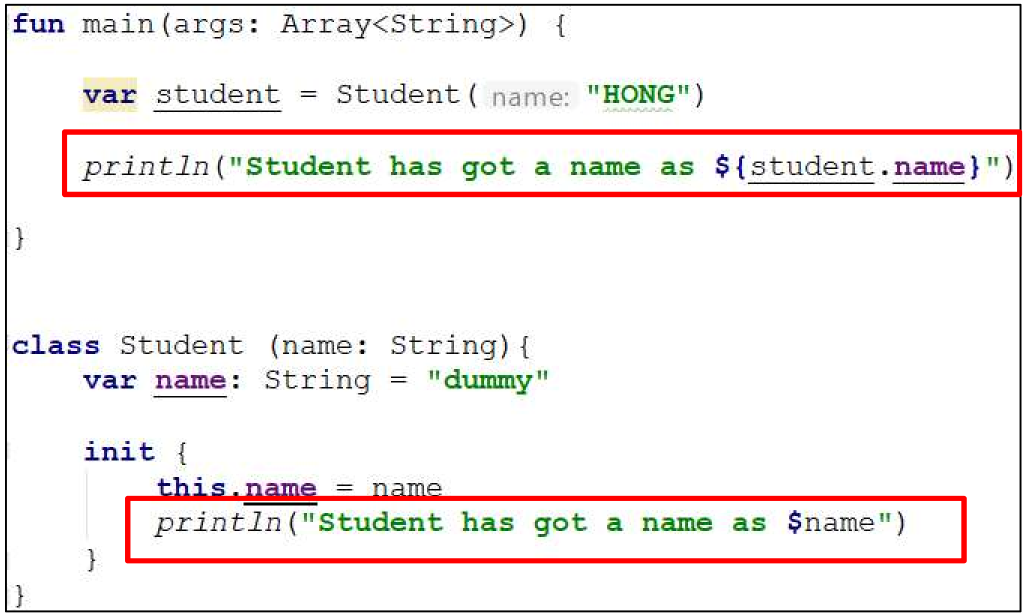
Class : Property

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



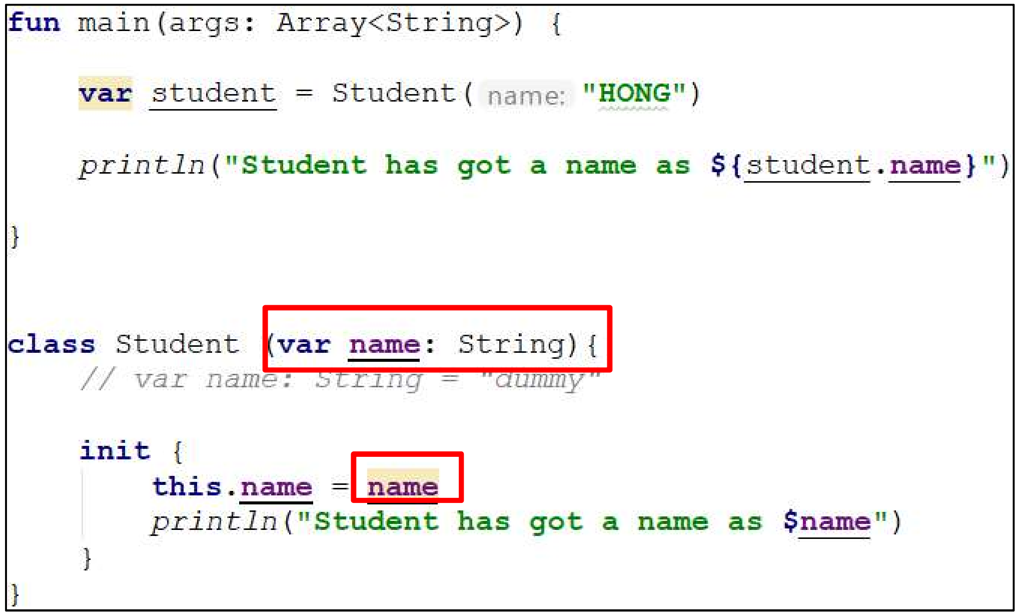
```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
        }
}

fun main() {
    val rect = Rectangle( height: 41,  width: 43)
    println(rect.isSquare)
}
```

Quiz #1 : 어떤 값이 출력될까?



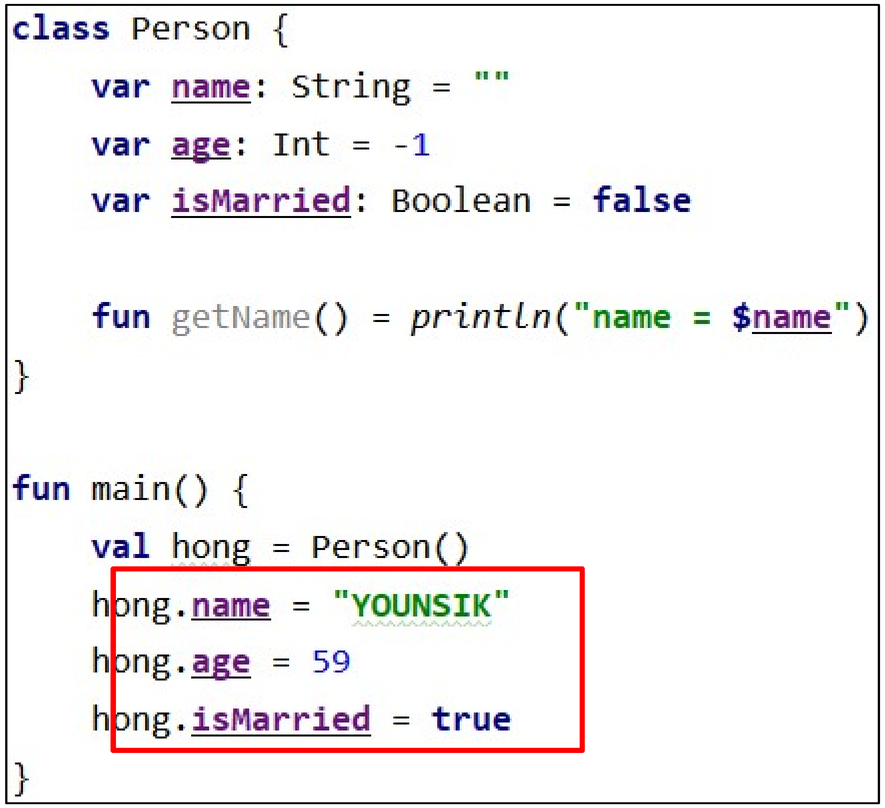
```
fun main(args: Array<String>) {

    var student = Student( name: "HONG")

    println("Student has got a name as ${student.name}")

}


class Student (name: String){
    var name: String = "dummy"

    init {
        this.name = name
        println("Student has got a name as $name")
    }
}
```

Quiz #2 : 어떤 값이 출력될까?

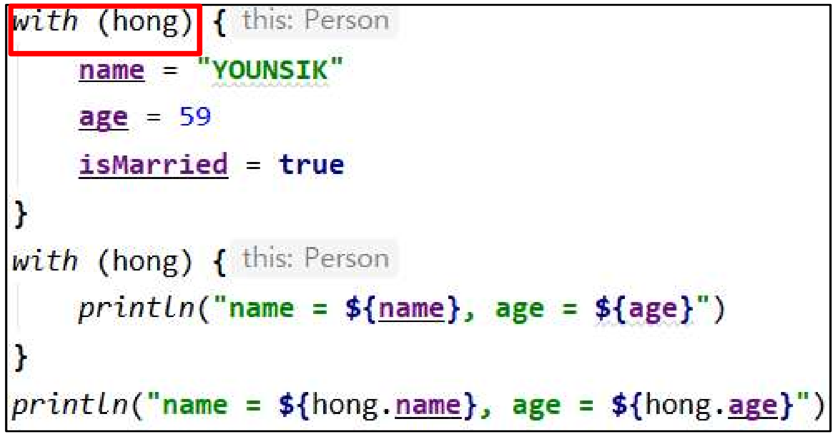


```
fun main(args: Array<String>) {

    var student = Student( name: "HONG")

    println("Student has got a name as ${student.name}")

}


class Student (var name: String){
    // var name: String = "dummy"

    init {
        this.name = name
        println("Student has got a name as $name")
    }
}
```

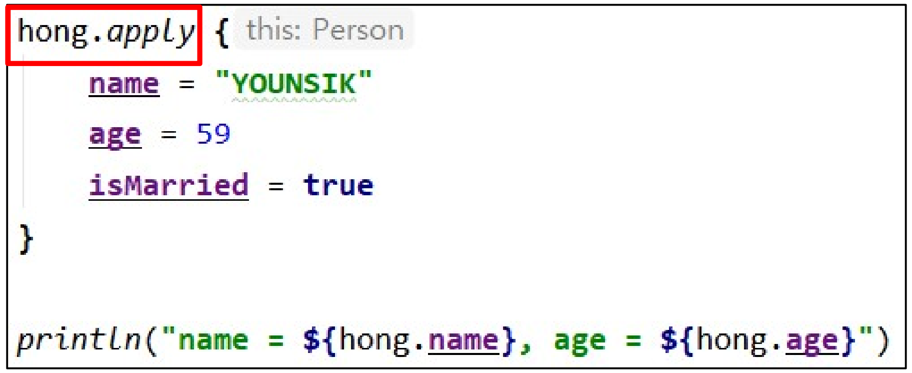
Use with statement



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

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

fun main() {
    val hong = Person()
    hong.name = "YOUNSIK"
    hong.age = 59
    hong.isMarried = true
}
```



```kotlin
with (hong) { this: Person
    name = "YOUNSIK"
    age = 59
    isMarried = true
}
with (hong) { this: Person
    println("name = ${name}, age = ${age}")
}
println("name = ${hong.name}, age = ${hong.age}")
```



```kotlin
hong.apply { this: Person
    name = "YOUNSIK"
    age = 59
    isMarried = true
}

println("name = ${hong.name}, age = ${hong.age}")
```

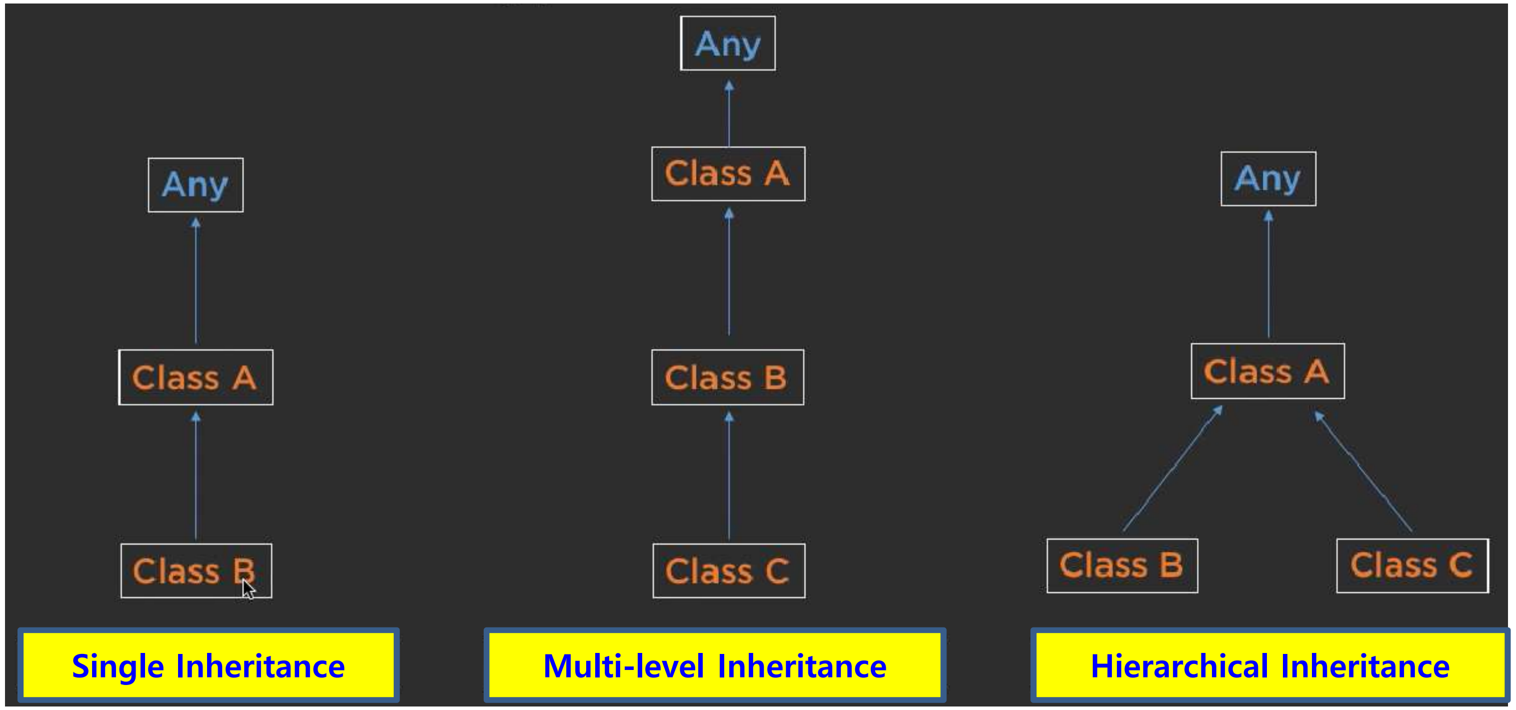
Inheritance (상속)

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

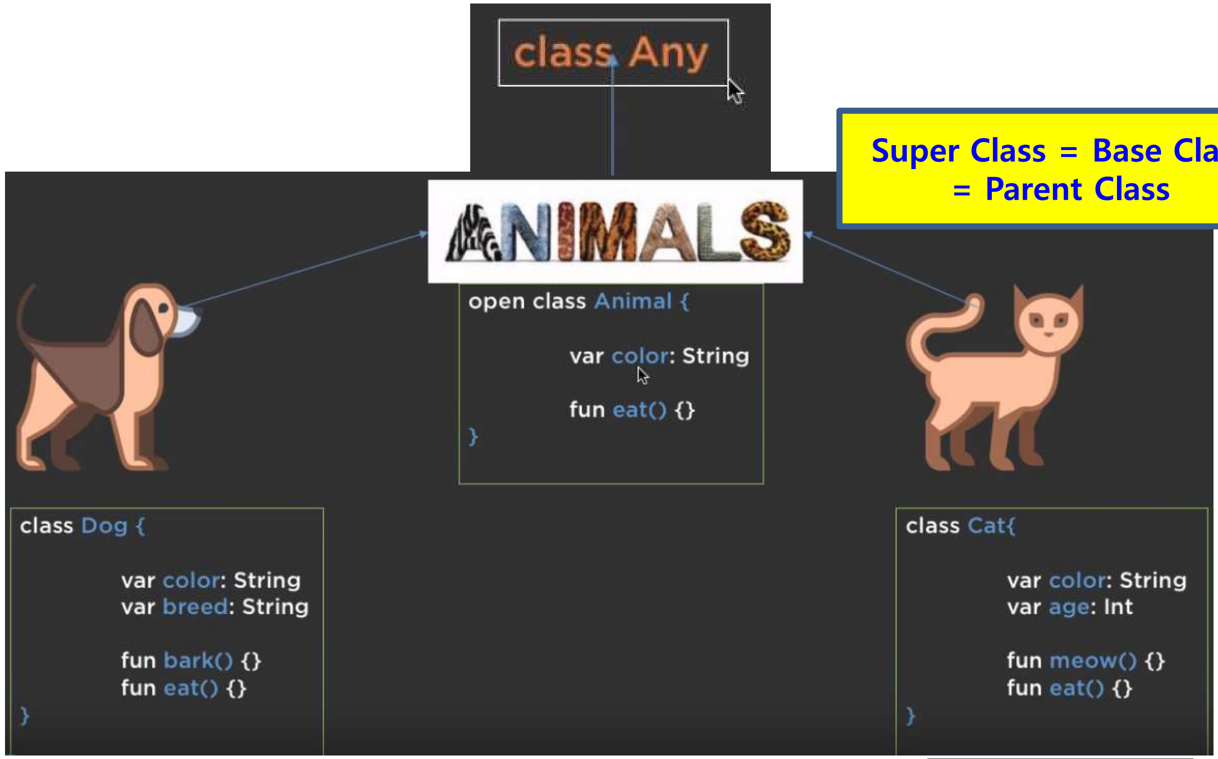
- For inheritance
  - You need to make a class '**open**'
    - A Child object acquires all the properties from its Parent class object.

- Advantages
  - For code reusability
  - For method overriding

Types of Inheritance



Inheritance 예(1/2)

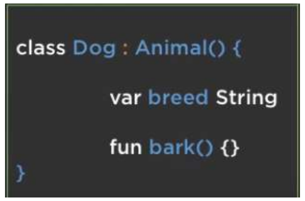


```
open class Animal {
        var color: String
        fun eat() {}
}
```

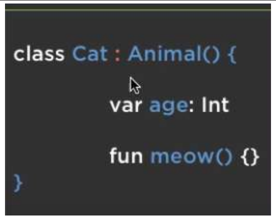
```
class Dog {
        var color: String
        var breed: String

        fun bark() {}
        fun eat() {}
}
```

```
class Cat{
        var color: String
        var age: Int

        fun meow() {}
        fun eat() {}
}
```

Super Class = Base Class = Parent Class



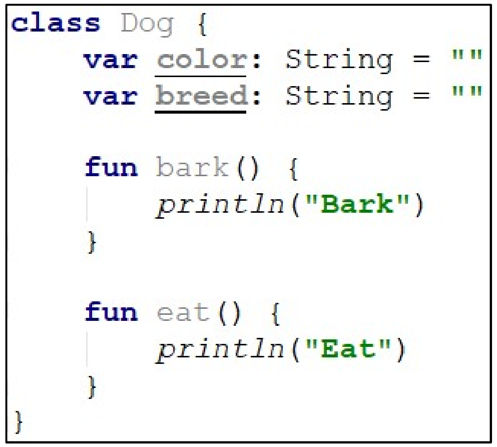
```
class Dog : Animal() {
        var breed String
        fun bark() {}
}
```



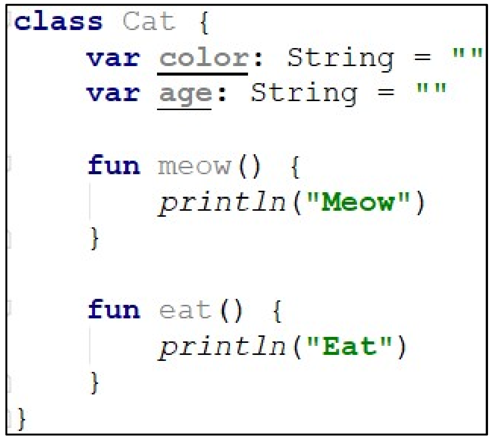
```
class Cat : Animal() {
        var age: Int
        fun meow() {}
}
```

Sub Class = Derived Class = Child Class

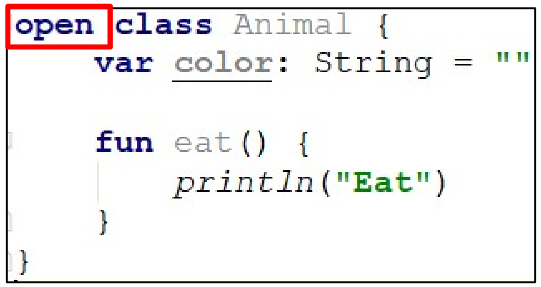
Inheritance 예(2/2)



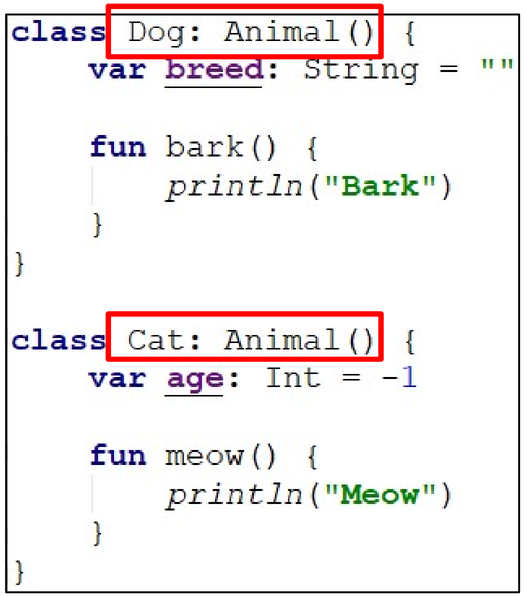
```
class Dog {
    var color: String = ""
    var breed: String = ""

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

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



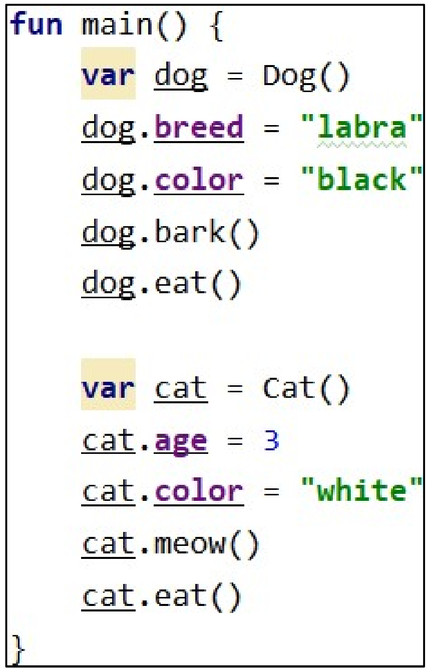
```
class Cat {
    var color: String = ""
    var age: String = ""

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

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



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

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

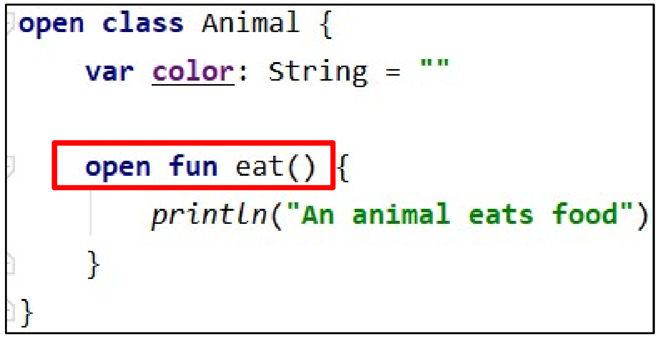


```
class Dog: Animal() {
    var breed: String = ""

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

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

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

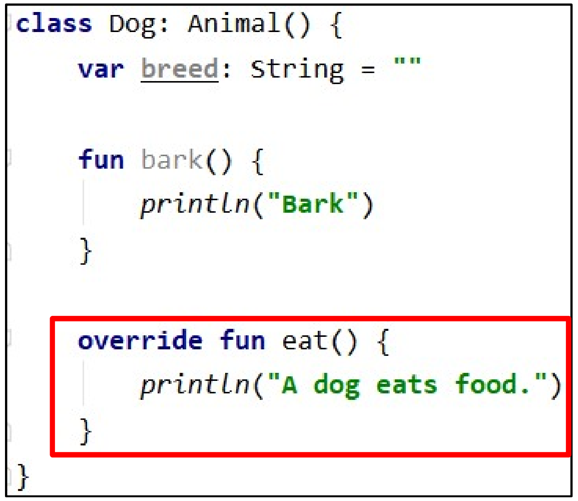


```
fun main() {
    var dog = Dog()
    dog.breed = "labra"
    dog.color = "black"
    dog.bark()
    dog.eat()

    var cat = Cat()
    cat.age = 3
    cat.color = "white"
    cat.meow()
    cat.eat()
}
```

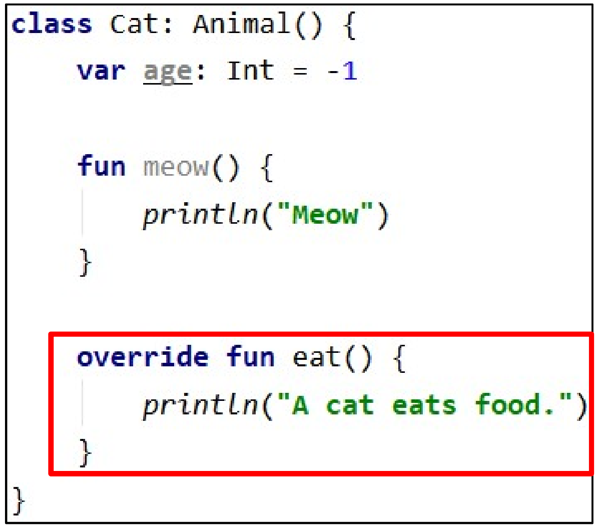
Overriding Properties and Methods (1/2)



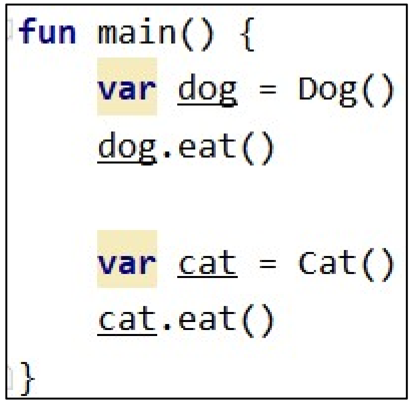
```kotlin
open class Animal {
    var color: String = ""

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



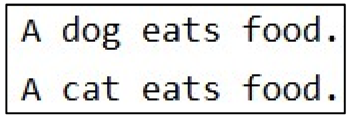
```kotlin
class Dog: Animal() {
    var breed: String = ""

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

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



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

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

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

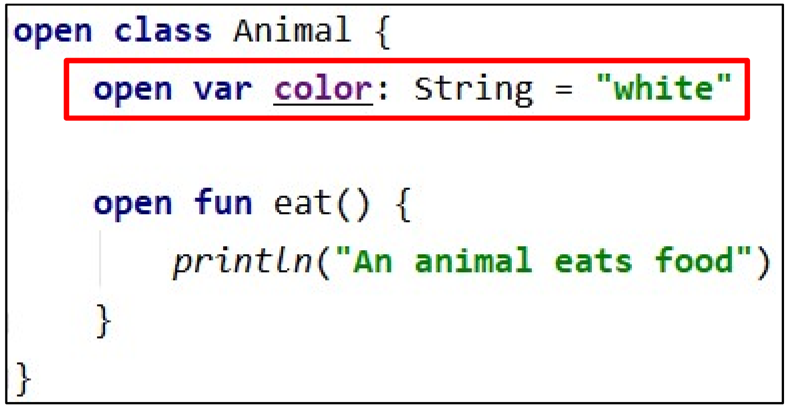


```kotlin
fun main() {
    var dog = Dog()
    dog.eat()

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

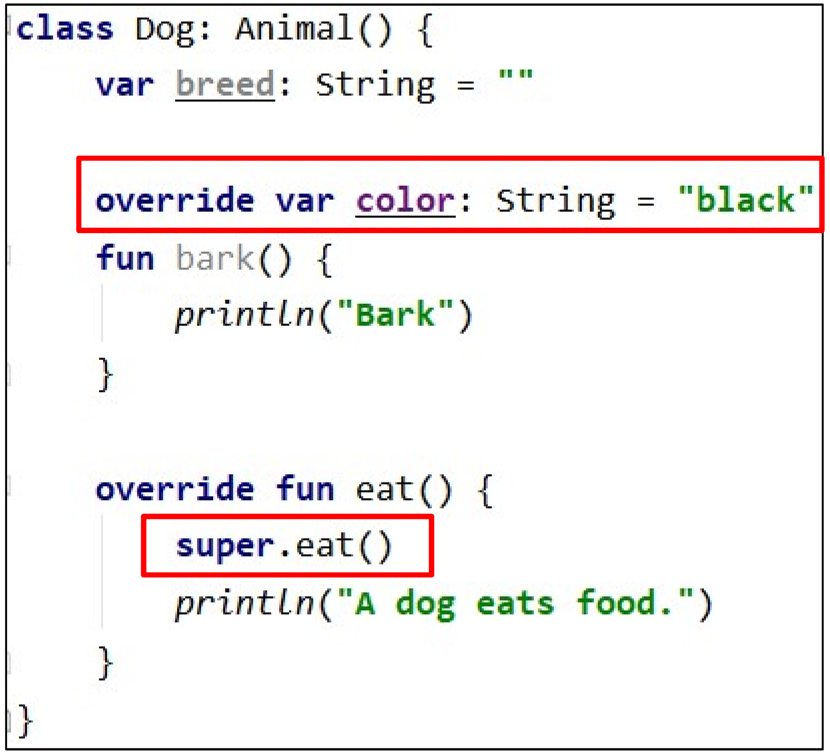


```
A dog eats food.
A cat eats food.
```

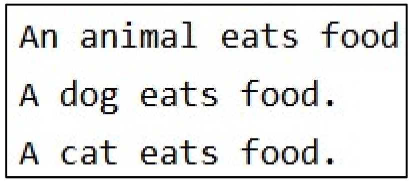
Overriding Properties and Methods (2/2)



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

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

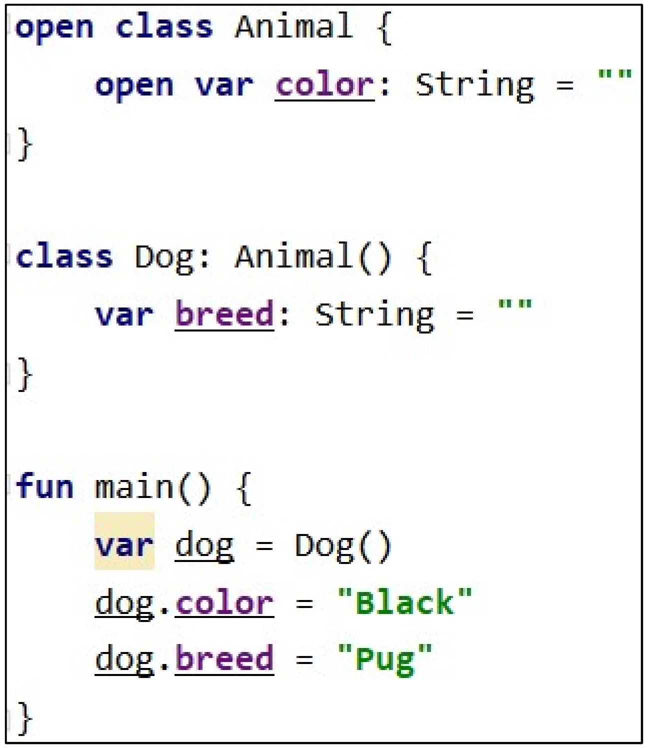


```kotlin
class Dog: Animal() {
    var breed: String = ""

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

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

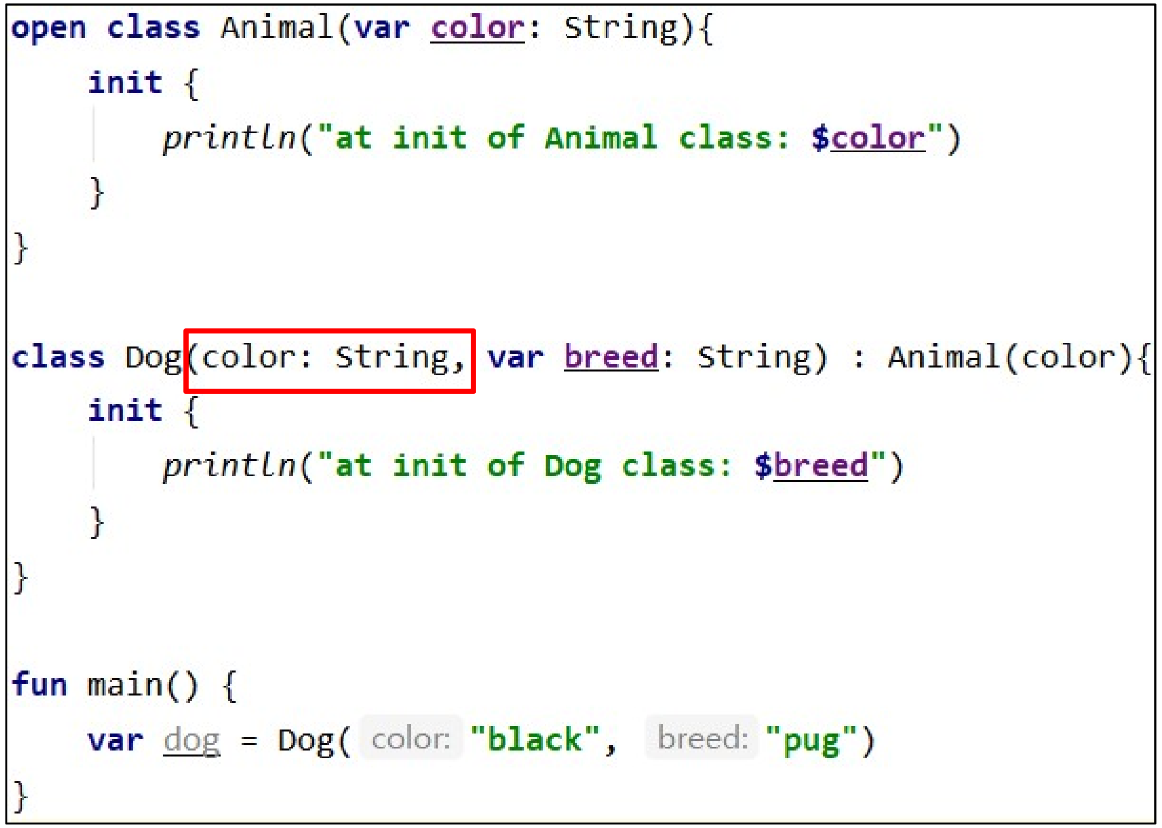


```
An animal eats food
A dog eats food.
A cat eats food.
```

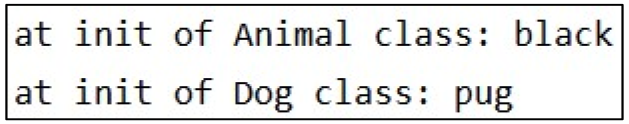
Primary and Secondary Constructors (1/2)



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

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

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

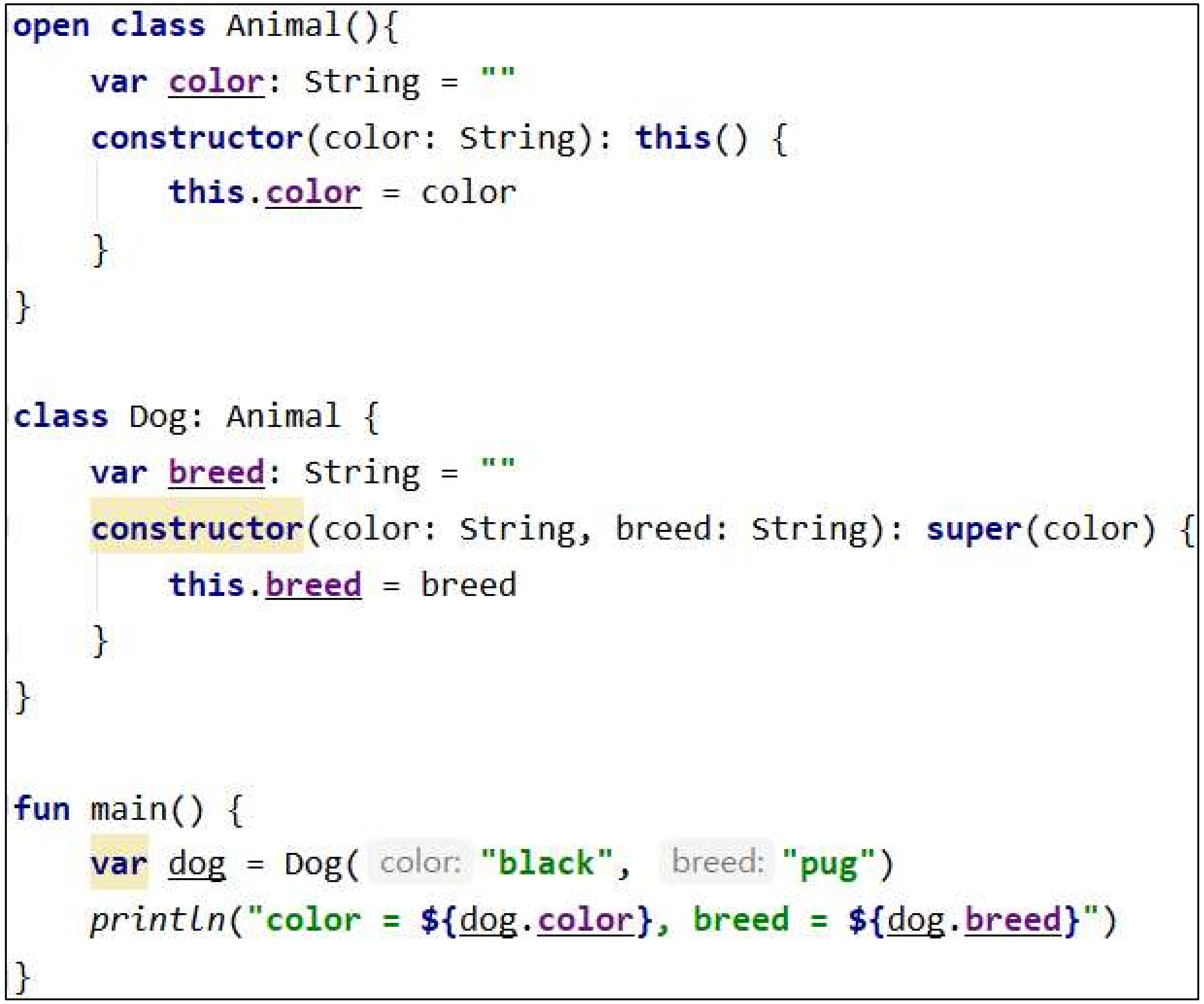


```kotlin
open class Animal(var color: String){
    init {
        println("at init of Animal class: $color")
    }
}

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

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



```
at init of Animal class: black
at init of Dog class: pug
```

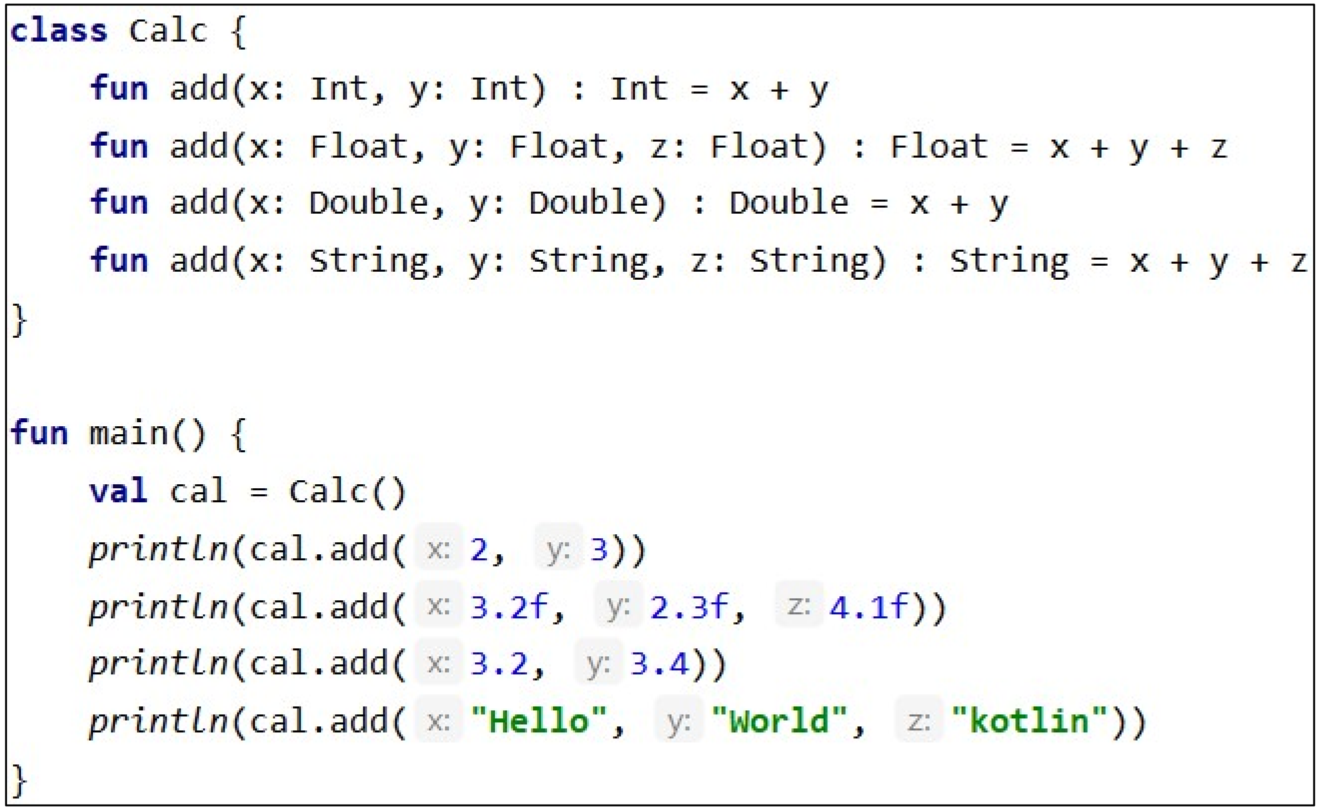
Primary and Secondary Constructors (2/2)



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

Overriding 과 Overloading

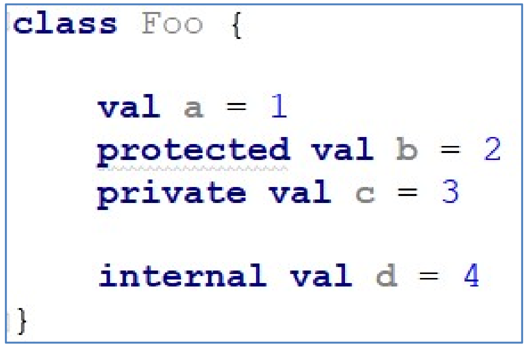
Overriding : method나 property의 이름은 같지만 동작이나 값을 재정의
Overloading : 동작은 같지만 parameter의 type이 다름.



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

fun main() {
    val cal = Calc()
    println(cal.add( x: 2,  y: 3))
    println(cal.add( x: 3.2f,  y: 2.3f,  z: 4.1f))
    println(cal.add( x: 3.2,  y: 3.4))
    println(cal.add( x: "Hello",  y: "World",  z: "kotlin"))
}
```

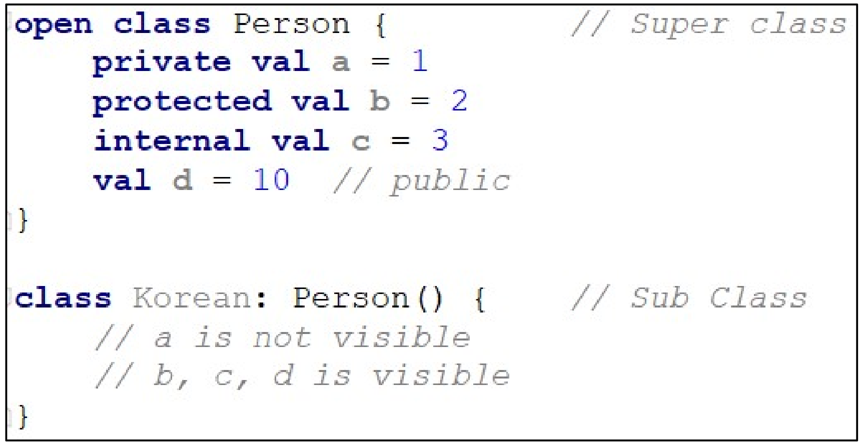
Visibility Modifiers

- **Visibility** (가시성, 접근 제한)
  - 클래스의 method나 property의 접근 권한을 지정
    - 정보 은닉
- **Visibility modifier**
  - **public (+)**: 모두에게 공개(default)
  - **protected (#)**: 상속받은 클래스에서는 접근 가능
  - **internal** : 같은 모듈 내에서는 접근 가능
  - **private (-)**: 접근 불가
    - **괄호 안 기호는 UML에서 사용**

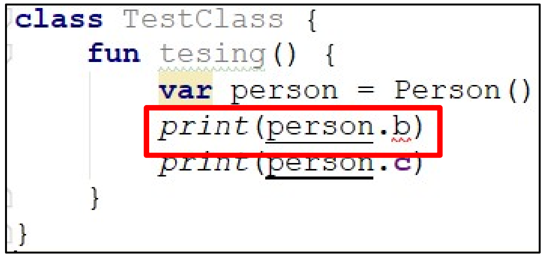


```
class Foo {

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

    internal val d = 4
}
```

Visibility Modifiers 예



```
open class Person {                    // Super class
    private val a = 1
    protected val b = 2
    internal val c = 3
    val d = 10   // public
}

class Korean: Person() {               // Sub Class
    // a is not visible
    // b, c, d is visible
}
```



```
class TestClass {
    fun tesing() {
        var person = Person()
        print(person.b)
        print(person.c)
    }
}
```

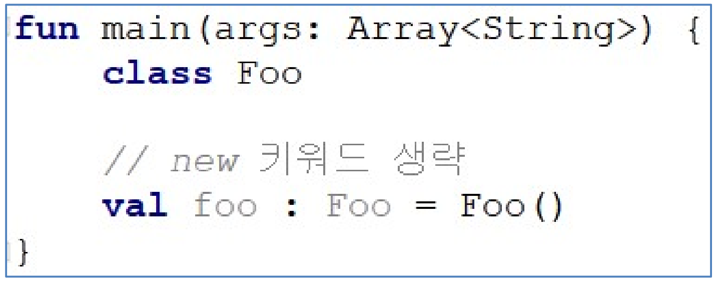
person . a, person . b are not visible
person . c, person . d are visible

Abstract Class (1/2)

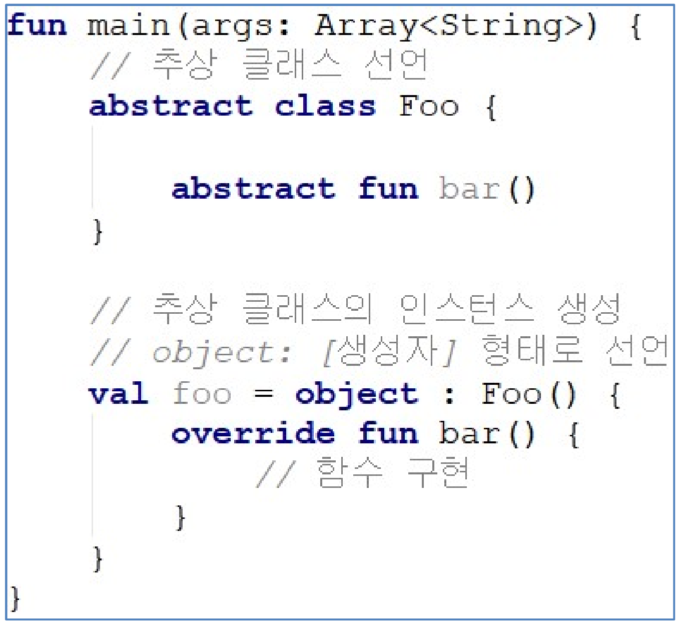
- Classes can be abstract in nature.
  - The role of abstract class is to just provide a set of method and properties.
- Abstract class is **a partially defined class**.
  - Abstract methods **have no body** when declared.

Abstract Class (2/2)

- **추상 클래스**는 java와 동일한 방법으로 선언하지만
  - 인스턴스를 생성하는 형태는 다름

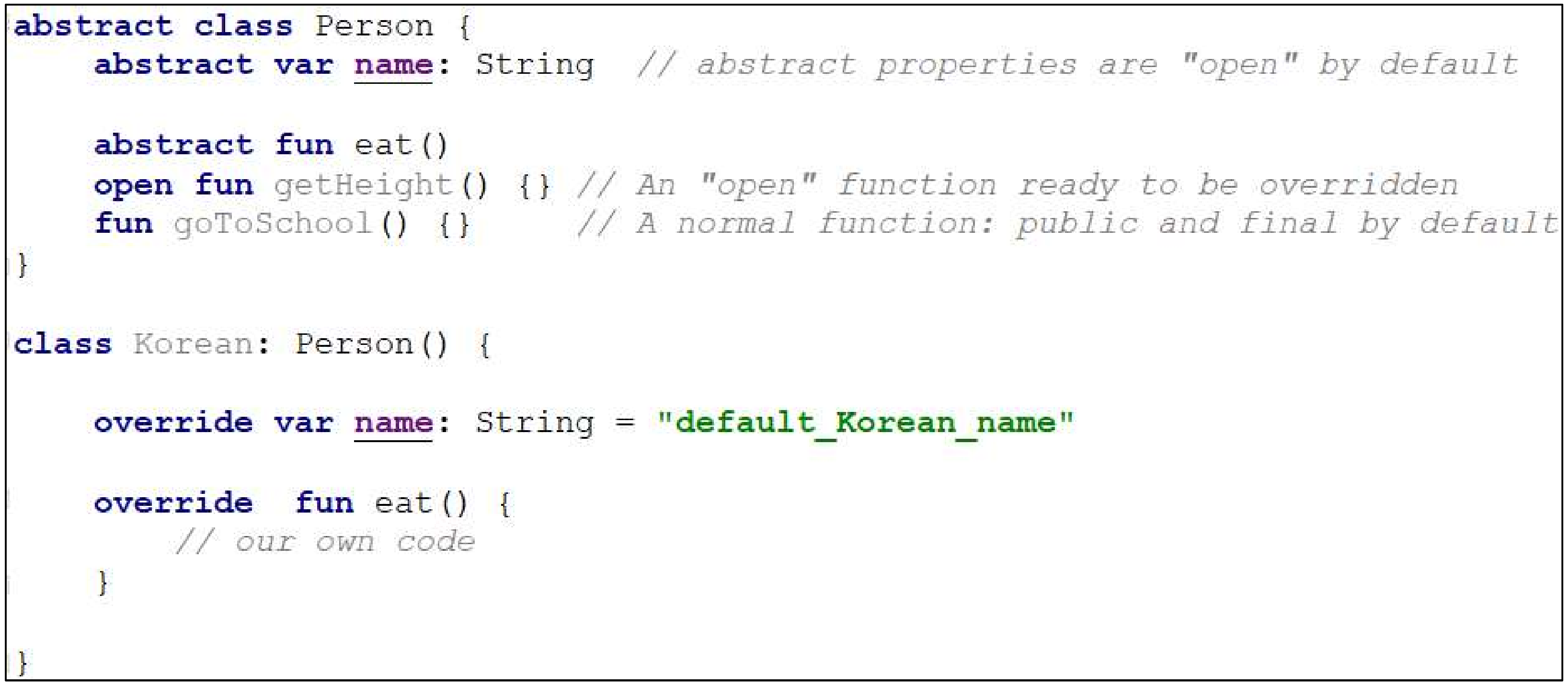


```
fun main(args: Array<String>) {
    class Foo

    // new 키워드 생략
    val foo : Foo = Foo()
}
```



```
fun main(args: Array<String>) {
    // 추상 클래스 선언
    abstract class Foo {

        abstract fun bar()
    }

    // 추상 클래스의 인스턴스 생성
    // object: [생성자] 형태로 선언
    val foo = object : Foo() {
        override fun bar() {
            // 함수 구현
        }
    }
}
```

Abstract Class 상속

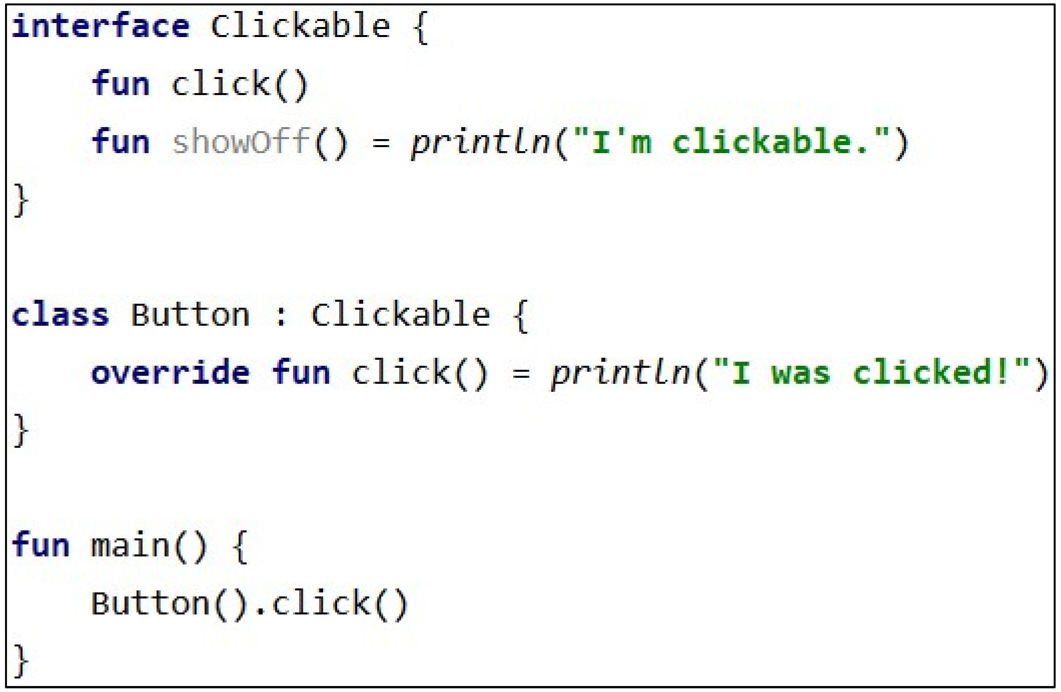


```
abstract class Person {
    abstract var name: String  // abstract properties are "open" by default

    abstract fun eat()
    open fun getHeight() {} // An "open" function ready to be overridden
    fun goToSchool() {}     // A normal function: public and final by default
}

class Korean: Person() {

    override var name: String = "default_Korean_name"

    override  fun eat() {
        // our own code
    }

}
```

You cannot create instance of abstract class.

Interface (1/2)

- Interface can contains both **NORMAL Methods and ABSTRACT Methods**.
  - But they contains only **ABSTRACT PROPERTY**.



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

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

fun main() {
    Button().click()
}
```

Interface (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()
    }
}
```

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

```
I'm clickable.
I'm focusable.
I got focus.
I was clicked!
```