Sprawozdanie o obiektowości w języku Kotlin

Norbert Janas

Klasy i dziedziczenie

Klasy w języku Kotlin tworzymy podając słowo kluczowe **class** *nazwa_klasy*, po czym ciało klasy zawieramy w **{ }**. Jeśli obok nazwy klasy postawimy **( )** to używamy wtedy podstawowego konstruktora do którego możemy podać słowa kluczowe **val** lub **var**, dzięki czemu dla podanego w taki sposób argumentu zostaną utworzone własności .

Konstruktor podstawowy nie może zawierać żadnego kodu, dlatego możemy użyć bloku **init { ... }** do zawarcia wybranych instrukcji, ponieważ blok ten wykonuje się zaraz podczas inicjalizacji instancji klasy.

Pozostałe konstruktory deklarowane są za pomocą słowa kluczowego **constructor**, a jeśli klasa posiada również konstruktor podstawowy to każdy następny konstruktor musi odnosić się do podstawowego używając **this**.

Domyślnie każda klasa w *kotlinie* jest zamknięta na dziedziczenie (**final**) i trzeba ją ‘otworzyć’ w celu uzyskania funkcjonalności dziedziczenia za pomocą słowa **open**.

```
// (argument, var/val properties)
open class Class3 (arg: String, var prop: String){ // Body of class
    val prop2 : Int = 5
    val prop3 : String
    var param1 : Int = 0

    constructor(arg: String, prop: String, param :Int) : this(arg, prop){
        param1 = param
    }
    init {      // Execute after initialization
        prop3 = arg
        println("Third property $prop3")
    }
}


fun main(){
    // Creating Object
    val ob1  = Class3( arg: "Argument",  prop: "Property")
    println(ob1.prop)
    // no property println(ob1.arg)
    println(ob1.prop2)
    println(ob1.prop3)

    val ob2 = Class3( arg: "a",  prop: "b",  param: 13)
    println(ob2.param1)
}
```

Aby nadpisać metodę lub pole klasy musimy jawnie to określić w klasie bazowej słowem **open**, a dopiero wtedy możemy przesłaniać w klasie pochodnej za pomocą **override**.

```
open class BaseClass (param: String, val id: Int){
    open val _isDerived: Boolean = false
    private val baseParam = param
    fun printState() = "Class is ${if(_isDerived) "not " else ""}base"
}

class DerivedClass(param : String, id : Int, var doubNum: Double) : BaseClass(param, id){
    override val _isDerived: Boolean = true
}

fun main(){
    val ob1 = BaseClass( param: "base",  id: 0)
    println(ob1.printState())

    val ob2 = DerivedClass( param: "p2",  id: 1,  doubNum: 2.5)
    println(ob2.printState())
    // Error ob2.baseParam

}
```

```
Class is base
Class is not base

Process finished with exit code 0
```

Klasy abstrakcyjne

Jeśli chodzi o klasy abstrakcyjne to są one domyślnie otwarte i nie trzeba jawnie określać ich słowem **open**. Klasę, pola oraz metody poprzedzamy słowem kluczowym **abstract**. Jeśli natomiast tego nie zrobimy są one domyślnie *nie abstrakcyjne* i musimy określić je **open** aby je przesłaniać.

```
abstract class Vehicle(var fuel: Double, var color: String){
    abstract val wheels: Int

    open fun refilFuel (){
        println("Tank is now full")
    }
    abstract fun drive()
}

class Car(fuel: Double, color: String, override val wheels: Int) : Vehicle(fuel, color){
    override fun drive() {
        println("Car is moving")
    }
}

class Bike(fuel: Double, color: String, override val wheels: Int) : Vehicle(fuel, color){
    override fun drive() {
        println("Bike is moving")
    }

    override fun refilFuel() {
        println("There is no tank")
    }
}
```

```kotlin
fun main(){
    val car = Car( fuel: 40.5,  color: "black",  wheels: 4)
    val bike = Bike( fuel: 0.0,  color: "red",  wheels: 2)
    car.drive()
    bike.drive()
    car.refilFuel()
    bike.refilFuel()
}
```

Bike › drive()

TestowankoKt

```
"C:\Program Files\Android\Android Studio\jre\bin\java.exe" ...
Car is moving
Bike is moving
Tank is now full
There is no tank
```

Interfejsy

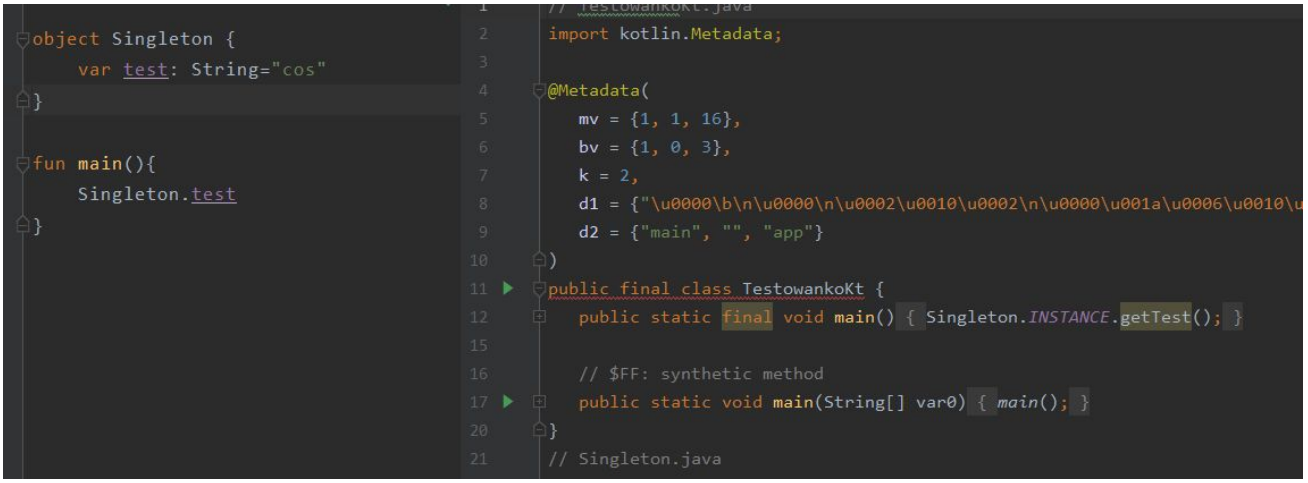
Interfejsy nie mogą przechowywać stanu, jednak w języku Kotlin można utworzyć tam **var/val**, ponieważ w kotlinie domyślnie są to *własności* (abstrakcyjne lub implementując akcesor), dla których nie tworzone są pola zapasowe. Dodatkowo zostało umożliwione aby funkcje w interfejsach posiadały ciało dopóki nie są one **final** (blokada przesłonięcia) (w klasie abstrakcyjnej metody domyślnie są **final**). (można implementować wiele interfejsów, a dziedziczyć tylko jedną klasę)

```kotlin
interface Animal {
    var name: String // abstract property
    fun walk() {
        println("walk")
    }
    fun makeSound()
}

class Dog(override var name: String): Animal {
    // No walk override

    override fun makeSound() {
        println("BARK")
    }
}

fun main(){
    val dog = Dog( name: "Husky")
    dog.makeSound()
    dog.walk()
    println(dog.name)
}
```

Singleton

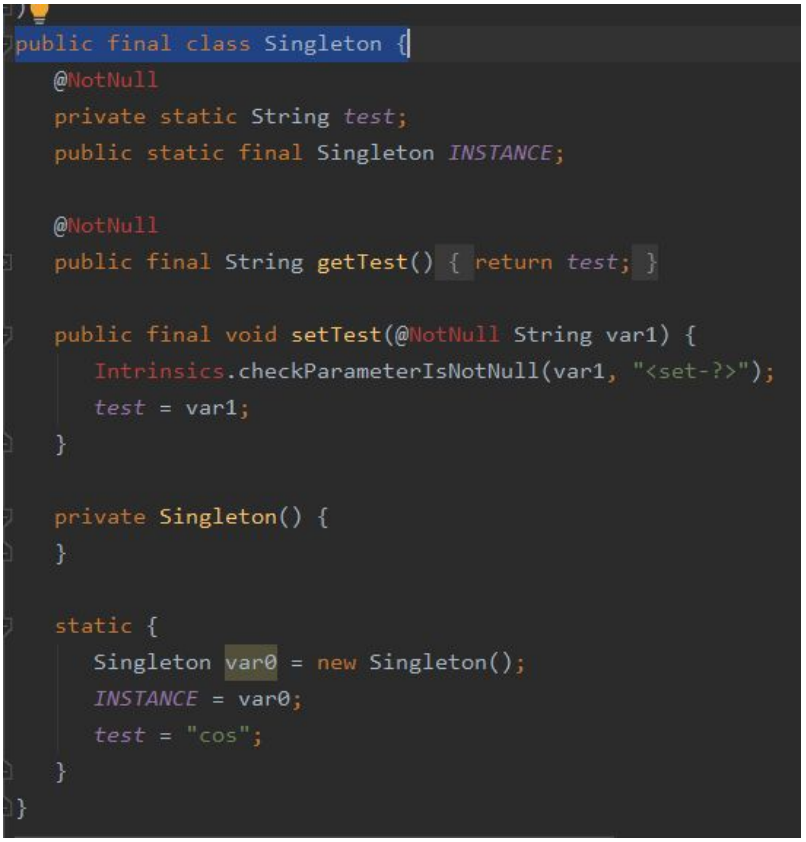
Chcąc utworzyć singleton używamy słowa kluczowego **object** po czym podajemy nazwę singletona. Używając **object** dosłownie tworzymy klasę oraz jej jedyną instancję w jednym miejscu (jest tworzona przy pierwszym użyciu). Nie piszemy konstruktora ponieważ to kompilator tworzy jedyną instancję za nas. Po deklaracji **object** *nazwa* {... możemy projektować jak w normalnej klasie. Używamy odwołując się do *nazwa*.

Po wykonaniu *nazwa.pole/metoda* i dekompozycji kotlina do kodu javy faktycznie możemy zauważyć, że pod przykryciem java odwołuje się do instancji utworzonego przy **object** Singletona w celu pobrania zmiennej.



```
object Singleton {
    var test: String="cos"
}

fun main(){
    Singleton.test
}
```

```
// TestowankoKt.java
import kotlin.Metadata;

@Metadata(
   mv = {1, 1, 16},
   bv = {1, 0, 3},
   k = 2,
   d1 = {"\u0000\b\n\u0000\n\u0002\u0010\u0002\n\u0000\u001a\u0006\u0010\u
   d2 = {"main", "", "app"}
)
public final class TestowankoKt {
   public static final void main() { Singleton.INSTANCE.getTest(); }

   // $FF: synthetic method
   public static void main(String[] var0) { main(); }
}
// Singleton.java
```



```
public final class Singleton {
   @NotNull
   private static String test;
   public static final Singleton INSTANCE;

   @NotNull
   public final String getTest() { return test; }

   public final void setTest(@NotNull String var1) {
      Intrinsics.checkParameterIsNotNull(var1, "<set-?>");
      test = var1;
   }

   private Singleton() {
   }

   static {
      Singleton var0 = new Singleton();
      INSTANCE = var0;
      test = "cos";
   }
}
```

‘Statyczność’ (companion object)

W języku Kotlin nie słowa kluczowego static, więc jeśli potrzebujemy utworzyć elementy należące do klasy, a nie instancji możemy użyć **companion object** (*opcjonalnie nazwa*). Instrukcja ta powoduje utworzenie się zagnieżdżonej klasy (singleton), która przetrzymuje elementy powiązane z klasą zewnętrzną (**companion object** może implementować interfejsy). Do elementów takiego obiektu możemy odwoływać się jakby były obiektami klasy (w której utworzony został **companion object)**.