OOPs In Kotlin

Friday, 5 June 2020 3:01 AM

class Student → min valid class

Properties of a class

```
// var <property name>[:<property Type>] = [<property_initializer>]
// [<getter>]
// [<setter>]

class Student {
    val school: String = "unknown"
    var age: Int = 5
    val isTeenager: Boolean
        get() = age >12

    val name: String? = null
        get() = field ?: "Unknown"
        set(value) {
            if (value != null) field = value
        }
}
```

Backing field - if we will write name here then it will cause recursion

Constructor

constructor → primary constructor, optional if there is no modifier or annotation

(firstName → if we write val/var here we don't need it here)

```
class Student constructor (firstName : String ="Mark") {
    val firstName :String
    var lastName : String? = null
    init {
        println ("Init block called")
        this.firstName = firstName
    }
    constructor (firstName: String, lastName: String): this (firstName){
        println ("secondary constructor called")
        this.lastName = lastName
    }
}
```

init → Called with primary constructor

constructor → secondary constructor

this (firstName) → calling primary constructor

Visibility Modifiers

		class level	top level
1	Public (default)	visible everywhere	visible everywhere
2	internal	visible in a module	visible in a module
3	protected	Visible inside the class/subclass	N/A
4	private	Visible inside the class	Visible inside the file

Inheritance

open → specifies class is inheritable because default value is final

```
open class person (val name: String) {
    init {
        println ("This is a person")
    }

    var age : Int = 1
open fun doWork (){
        println ("person is doing somework")
    }
}

class Student (name: String, val school :String) : Person(name){
    init {
        println ("This is a student")
    }

    override fun doWork() {
        println ("Student is studying")
        super.doWork()     //calls parent's dowork
    }
}
```

Person(name) → parent class constructor is called first; can inherit only one class

* Any class is parent of all classes & it contains hashCode and toString() func^n

abstract classes

```
abstract class person (val name: String) {
    abstract fun doWork()
    fun doTalk() {
        // a function to talk)
    }
}
class Student (name:String) : Person (name) {
    override fun doWork(){
        println ("student is studying");
    }
}
```

abstract fun doWork() → no body as abstract

Interface → A class can implement multiple interfaces. Interfaces are made using keyword interface. They are diff from abstract class as they don't possess state

Nested Inner Class →

```
class Student(val name: String) {
    inner class schoolBag () {
        fun printBagOwner() {
            println (" This school bag belongs to $name")
        }
    }
}
fun Main() {
    Student("Mark").SchoolBag().printBagOwner()
}
```

inner → if we don't use inner then it'll be just an nested class

Anonymous Class -

```
open class Person {
    fun walk() {
        println (" Person is walking");
    }
    fun sleep() {
        println ("Person is sleeping")
    }
    open fun talk() {
        println ("Person is talking")
    }
}
fun main() {
    startTalking (object: Person() {
        override fun talk() {
            println ("I am talking")
        }
    })
}
fun startTalking ( person: Person) {
    person. talk()
}
```

Data Class

```
//properties can only have properties in constructor
data class Student (val name: String, var section: String){
    var age: Int = 8
}

// data class considers only properties while equating objects
```

ENUM class

```
enum class Metal (val symbol: String) {
    IRON ("Fe")
    GOLD ("Ag")
}
fun main () {
    for (metal in Metal.values()){
        println ("${metal. symbol} ${metal.name} ${metal.ordinal}")
    }
    printMetal (Metal.IRON)
}
fun printMetal (metal : Metal) {
    println (metal.name);
}
```

Sealed class

```
sealed class NetworkState {
    data class Error (val errorText :String): NetworkState()
    object loading : Network State()
    data class loaded (val content : String) : NetworkState()
}
fun main () {
    val data: NetworkState.loaded ("Data loaded")
    process (data)
}
fun process (state : Network State) {
    when (state) {
        is NetworkState. Error → {print ("${state. errorText}")
        is Networkstate . loading → {//}
        is NetworkState . loaded → {print ("${state. content}")
    }
}
```

→ sealed class is diff from enum because it can have it's subclass to contain properties in them.

Extension function → add methods to class during runTime()

```
fun Int. getEven () : Int {
    if( this %2 ==0) return this
    else return this +1
```

Singleton

↳ no constructor

```
object GameScore {
    var score:Int =0
        private set
    fun addScore( point :Int) {
        if(point >0)
            this. score += point
        }
    }
}

class Student (val name: String) {
    companion object {
        const val schoolName = "Dp School"
    }
}

fun main() {
    Game Score. add Score (10)
    println("${Student . schoolName"})
}
```

↳ if we use companion object otherwise we need to mention inner class name.