KOTLIN

Dominique BESSON

SOMMAIRE

PRÉ-REQUIS

1 ➔ Un IDE (IntelliJ par exemple)

2 ➔ JDK 1.6+
➔ Configurer les variables d'environnement JAVA

RAPPELS KOTLIN

Langage

- ➔ Orienté Object et Fonctionnel
- ➔ Typage statique
- ➔ Compile pour JVM, Interpréteur JS,, Natif...

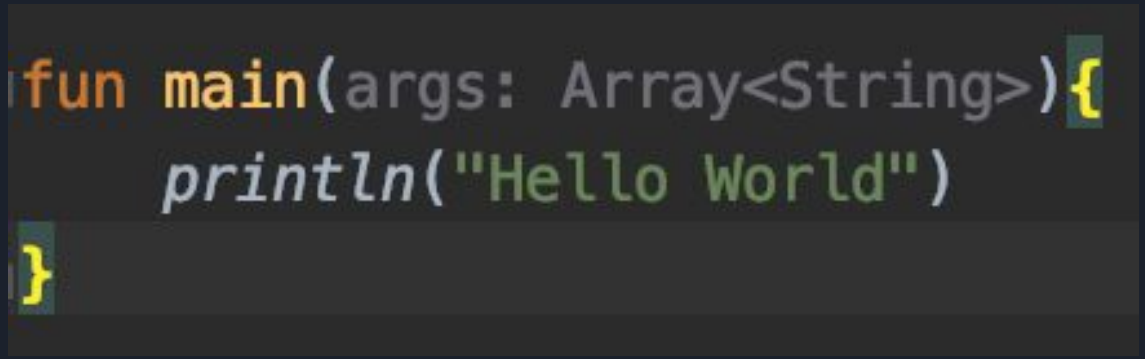
Kotlin VS Java

- ➔ Code moins verbeux comparé à Java
- ➔ Pas obligation d'utilisation de la JVM
- ➔ Moins de NullPointerException grâce à la syntaxe et au check du compilateur

Résultat

- ➔ Langage intuitif
- ➔ Est interopérable avec Java
- ➔ Productivité augmentée

Hello World



Variables

```
var var2 : String = "hehe"
var var3 = "hihi"
var2 = "hihi"

val var1 = "varvar"
val var4 : Int = 4

lateinit var late : String

var var5 : Int?
val var6 : String? = null

println(var6?.toLowerCase())

var obj = StringBuilder()
```

- Var: Déclare une variable qui peut changer de valeur
- Val: Initialise une variable qui ne peut changer de valeur
- lateinit var: peut être initialisée après, le compilateur fermera les yeux
- <Type>?: permet à une variable d'être null
- <variable>?.methode(): exécute une méthode lorsqu'un variable est potentiellement null
- const val: pareil que val mais valeur connue à la compilation
- Plus de New du Java

Fonctions

```
fun main(args: Array<String>){
    var func1 = fun(){
        // do something
    }
    val fun2 = fun(x: Int, y: Int): Any{
        println("$x + $y = "+ (x+y))
        return true
    }

    fonctionInutile(fun2)
}

fun maFonction(arg1: Int, arg2: String?): String{
    return arg1.toString()+arg2?.trim()
}

fun fonctionInutile(callback: (Int, Int) -> Any){
    callback(1, 2)
}

fun sansCorps() = "hello"
```

- Une fonction peut être affectée à une variable
- Si pas de type de retour -> Unit (comme void en Java)
- Une fonction peut prendre en argument une autre fonction
- Any -> n'importe quel type
- On peut faire des fonction sans corps -> la valeur doit être une expression

Les structures de contrôle

```
fun fonctionInutile2(a: Int): Int{
    return if(a < 18){
        println("Bah mineur")
        a+1
    }else{
        println("Enfin un vrai!")
        a
    }
}
```

if:

- Change le flow d'exécution par rapport à une condition
- Est une expression
  - Une expression retourne toujours une valeur

When:

- Comme le switch en Java
- Est une expression
- peut s'utiliser sans passer de valeur à évaluer

```
var code = 200

when(code){
    in 200..204 -> {
        println("Tout s'est bien passé")
    }
    400, 404 -> println("Il y a un problème de votre côté")
    else -> println("Je sais pas quoi dire")
}

when{
    true -> println("true")
    false -> println("false")
}
```

Les structures de contrôle

```
for(i in 1..50){
  // do something
}

val tab = listOf<Int>(4, 5, 0)
for(value in tab){
    println(value)
}
```

Les boucles:

- Du while, do while classique comme java
- For change comparé aux autres
  - itération sur range
  - itération sur valeur dans une collection
  - itération avec index et valeur
  - pleins d'autres

Les listes

```
var arr = arrayOf(5,6)
arr[0] = 6
arr[3] = 7

val list = listOf(0.5, 2.3)
// comme du Java
val arrayList = arrayListOf<String>()

val set = setOf<String>("string", "string2lol")
println(set.random())
```

Différents types génériques:

- arrayOf
  - tableau d'éléments ordonnés et muable
- listOf
  - liste d'éléments ordonnés et immuable
- mutableListOf
  - liste d'éléments ordonnés mais muable
- setOf
  - ensemble d'éléments immuable
- mutableSetOf
  - ensemble d'éléments muable

et pleins d'autres types de liste.

EXO: plus ou moins

Consigne

Faire un programme qui applique le jeu du plus ou moins.

Une première personne entre un nombre et ensuite la seconde doit trouver le nombre entré.
Tant qu'elle n'a pas trouvé, le programme continue.
Quand elle a trouvé, le programme s'arrête.
Quand le nombre entré par la seconde personne est inférieur, printer plus et vice versa.

Bonus

- un compteur d'essaie
- un chaud/froid like
- le premier chiffre entré ne doit pas se voir
- une option en pleine partie pour abandonner

Cast

“Caster c’est convertir une variable d’un type en un autre type.”

- On ne convertit pas tout en n’importe quoi ! il doit y avoir un sens
- Si A est considéré comme un type B, je peux caster et manipuler en tant que B, sinon je peux pas
- is: teste le type et convertit directement si le type est bon
- as: convertit le type sans tester
- as?: convertit le type, si c’est pas bon, retourne null

```
val typeAny: Any = "mon text"
val uneString:  String = typeAny as String
val uneString2 = typeAny as? String
```

```
fun main(args: Array<String>){
    quiSuisJe( variable: "texte")
    quiSuisJe( variable: 2.3)
    quiSuisJe( variable: false)
}

fun quiSuisJe(variable: Any){
    if(variable is Int)
        println("Je suis un entier")
    else if (variable is String)
        println("Je suis une string: "+variable.toUpperCase())

    // autre manière de faire
    when(variable){
        is Boolean -> println("Je suis un booléen")
        is Double -> println("Je suis un double")
    }
}
```

Les exceptions

Une exception c'est quand votre programme fait une opération illégale qui peut le faire crash (division par 0, mauvais cast, index supérieur à la taille du tableau/liste…).

- il faut englober le code dangereux dans un try {}
- ensuite capturer l'exception avec un catch(<nom>: <Type>){}
- dans le catch, on exécute le code en cas d'exception

```
try{
    val text = "je suis un texte"
    val entier = text as Int
    println(entier)
}catch (exception: ClassCastException){
    println("C'était pas un entier :/...")
}
```

À savoir:

- on peut nous même lancer une exception
- il existe d'autres choses à savoir sur les exceptions, je vous laisse vous renseigner vous même ;)

Programmation orientée objet

Permet de résoudre un problème (faire son programme) en manipulant des objets qui:

- possèdent des données (des **propriétés**)
- peuvent réaliser des actions (des **méthodes**).

- les propriétés
  - sont des variables
  - peuvent être de n'importe quel type
- les méthodes
  - sont des fonctions
  - on les définit comme on veut
  - leur but est de modifier les propriétés de leur objet

Un (vrai) programme en POO, est un ensemble d'objets qui appellent leurs méthodes, modifie leurs propriétés pour apporter une solution à un problème.

```
fun main(args: Array<String>){
    val voiture = Voiture()
    voiture.alume()
    do{
        voiture.avancer()
    }while(voiture.distance != 80)
    voiture.stop()
    voiture.eteindre()
}
```

Classes

Un classe est le modèle des objets.

- pour créer une classe, j'utilise le mot Class
  - première lettre du nom en majuscule
- les méthodes de la classe peuvent prendre des paramètres et peuvent modifier les propriétés de la classe
- pour créer un objet (instancier une classe), regardez mon diapo précédent.

```
class Voiture {
    var distance: Int = 0
    var estAllume : Boolean = false
    var vitesse = 0

    fun alume() { estAllume = true }
    fun avancer() { vitesse += 10; distance += 10 }
    fun stop() { vitesse = 0; estAllume = false; vitesse}
    fun eteindre() { estAllume = false  }
}
```

Interface

Une interface est une classe abstraite.

- contient que des méthodes (vides ou non). méthode vide = méthodes **Abstraites**
- peut contenir d'autres choses, je vous laisse checker la doc

l'intérêt est qu'on peut manipuler des objets en utilisant juste les méthodes de l'interface, sans se soucier de ce qu'est réellement l'objet.

Héritage

Une classe peut hériter d'une classe ou d'une interface.

Une interface peut hériter d'une autre interface.

- l'héritage permet de partager les méthodes de la classe/interface parente aux enfants
- Les méthodes vides d'une interface sont à redéfinir si elles sont héritées dans une classe
  - marquer override devant

```
var moyenDeDeplacement : MoyenDeDeplacement = Voiture()
moyenDeDeplacement.avancer()
moyenDeDeplacement.stop()
// je change moyen de déplacement
moyenDeDeplacement = Velo()
moyenDeDeplacement.avancer()
moyenDeDeplacement.stop()
```

```
class Voiture : MoyenDeDeplacement{
    var estAllume = false

    fun checkAllume(){
        if(estAllume)
            return
        estAllume = true
    }
    override fun avancer() {
        checkAllume()
        println("je met les mains sur le volant")
        println("j'appuie sur la pédale")
    }
    override fun stop() {
        println("j'appuie sur le frein")
        println("")
    }
}

class Velo : MoyenDeDeplacement{
    override fun avancer() { println("je pédale") }
    override fun stop() { println("j'appui sur mes freins") }
}
```

EXO: Animalerie

Consigne

Faire un programme qui gère une animalerie.

Votre animalerie contiendra des animaux. Il y aura possibilité d'ajouter autant d'animaux qu'on veut. Elle pourra contenir des chiens, des chats et des perroquets. Chaque animal doit pouvoir parler en faisant le son qui lui correspond et doit pouvoir dire ce qu'il est.

Techniquement, il faut

- Un interface Animal
  - une méthode parler
  - une méthode quiSuisJe
- les classes d'animaux qu'il faut
- Une classe Animalerie
  - une méthode add pour lui ajouter un animal passé en paramètre
  - un méthode affiche qui appelle quiSuisJe pour tous les animaux de l'animalerie

Dans votre fonction main, vous créerez un animalerie, ajouterez des animaux, afficherez tous vos animaux, bref faites vous plaisir pour montrer ce que sait faire votre programme.

Constructeur

Un constructeur permet d'initialiser la valeur des propriétés et de réaliser des actions au moment de la création de l'objet.

- **var:** déclare une valeur attribuée à la construction et peut être changée après dans les méthodes ou via l'objet
- **val:** déclare une valeur attribuée à la construction et ne peut être changée
- **Init:** block de code appelé à chaque objet créé
- **constructor**: permet de définir d'autres constructeur . Un constructeur peut en appeler un autre

```
class BasseDeDonnee(var adresseIp: String,
                    var nomUtilisateur: String){
    var mdp : Int = readLine()!!.toInt()

    constructor(): this( ip: "127.0.0.1")
    constructor(ip: String): this(ip,  nomUtilisateur: "defaultUser")
```

```
class BasseDeDonnee(var adresseIp: String,
                    var nomUtilisateur: String){
    var mdp : Int = readLine()!!.toInt()
    init {
        if(checkAddressIp() == true)
            connect()
    }
    fun checkAddressIp() = true
    fun connect(){
        /* j'utilise adresseIp et
            nomUtilisateur avec le mdp pour me connecter
         */
    }

    fun quiSuisJe(){
        println("je suis une basse de donnée connectée à l'ip $adresseIp")
    }
}

fun main(){
    val bdd = BasseDeDonnee( adresseIp: "10.24.56.90",  nomUtilisateur: "Jean")
    bdd.quiSuisJe()
}
```

```
val bdd = BasseDeDonnee( adresseIp: "10.24.56.90",  nomUtilisateur: "Jean")
val bdd2 = BasseDeDonnee()
bdd.quiSuisJe()
bdd2.quiSuisJe()
```

Visibilité

Les propriétés et les méthodes ont une visibilité qui varie selon des mots clés:

- private:
  - n'est accessible que dans sa classe
- public
  - est accessible partout où on manipule un objet de sa classe
- protected
  - est accessible dans sa classe et les classes qui en héritent
- internal
  - est accessible uniquement dans le module (une librairie qui contient notre code).

par défaut, sans rien marquer, la visibilité est public.

```
//val car ces données ne vont pas changer
class BasseDeDonnee(val adresseIp: String,
                    private val nomUtilisateur: String){
    private val mdp : Int = readLine()!!.toInt()

    fun quiSuisJe(){
        println("je suis une basse de donnée connectée à l'ip $adresseIp")
    }

    public fun visible(){}

    internal fun visibleMaisQueDansCeProjet(){}
}

fun main(){
    val bdd = BasseDeDonnee( adresseIp: "10.24.56.90",  nomUtilisateur: "Jean")
    bdd.quiSuisJe()
    bdd.visible()
    bdd.visibleMaisQueDansCeProjet()
    bdd.adresseIp
    bdd.mdp
}
```

Héritage ++

Une classe peut hériter d'une autre classe, qu'elle soit abstraite ou non, et bénéficier des même avantages de l'héritage que ceux déjà vus.

- **open:** permet de spécifier
  - qu'une class peut hériter de ma class concrète
  - qu'une méthode/propriété peut être redéfinie
- **final:** c'est l'inverse d'open (rien ne peut hériter de ma classe/on ne peut pas redéfinir)
- **abstract:** rend une class/méthode/propriété abstraite. Quand une classe concrète héritera de ma class asbtraite, il faudra redéfinir les méthodes/propriétés abstraites (s'il y en a)

```
open class Dog(private val name: String){
    open fun aboyer(){ println("j'aboie")}
    fun manger(){/*code pour manger*/}
}

class SuperDog(x: String): Dog(x){
    fun envoler(){/*code pour s'envoler*/}
    override fun aboyer() {
        super.aboyer()
        println("maintenant c'est aussi un super aboiement")
    }
}

fun main(){
    var dog = Dog( name: "albert")
    dog.aboyer()

    dog = SuperDog( x: "super albert")
    dog.aboyer()
    dog = dog as SuperDog
    dog.envoler()
}
```

Héritage ++

Une classe peut hériter d'une autre classe, qu'elle soit abstraite ou non, et bénéficier des même avantages de l'héritage que ceux déjà vus.

- **super:** permet d'appeler la méthode/propriété de la classe parente (si elle est définie). Me permet par exemple de faire le comportement parent puis de faire le mien (comme aboyer)
- quand ont hérite d'une class avec un constructeur, il faut absolument appeler le constructeur de cette class dans le notre en lui passant des paramètres si besoin
- **this:** correspond à la classe courante.

```
open class Dog(private val name: String){
    open fun aboyer(){ println("j'aboie")}
    fun manger(){/*code pour manger*/}
}

class SuperDog(x: String): Dog(x){
    fun envoler(){/*code pour s'envoler*/}
    override fun aboyer() {
        super.aboyer()
        println("maintenant c'est aussi un super aboiement")
    }
}

fun main(){
    var dog = Dog( name: "albert")
    dog.aboyer()

    dog = SuperDog( x: "super albert")
    dog.aboyer()
    dog = dog as SuperDog
    dog.envoler()
}
```

Les fonctions ++

quelques trucs sympa à savoir sur les fonctions/méthodes:

- on peut donner une valeur par défaut à leur paramètre (du dernier au premier)
- on peut utiliser le nom des paramètres quand on appelle la fonction. Ça s'appelle la surcharge d'opérateur.

@JvmOverload permet de produire le code compatible en java pour la surcharge d'opérateur.

```
@JvmOverloads
fun enregistrerDansFichier(message: String,
                           nomFichier: String = "defaut.txt"){

}

fun main(){
    enregistrerDansFichier( message: "ma ligne",
        nomFichier: "fichier.txt")
    enregistrerDansFichier(
        message: "ma ligne dans le fichier par défaut")
    enregistrerDansFichier(nomFichier = "désordre.txt",
        message = "rien")
}
```

Lecture d’un fichier

On le fait ensemble

Android et Kotlin: Démonstration

On le fait ensemble