TD Kotlin

Découverte du langage Kotlin

Déclaration de variables

Kotlin propose deux mots clés pour déclarer une variable selon qu'elle peut changer de valeur ou pas. Dans le premier cas, on utilisera **var**, tandis que ce sera **val** dans le second. On trouve ensuite le nom de la variable, puis son type séparé par un deux points :

Java	Kotlin

`String nom;`	`val nom : String`
	
	`ou`
	
	`var nom : String`

En Kotlin il est inutile de déclarer le type si on initialise la variable en même temps que sa déclaration. En effet le type est déduit de la valeur qu'on lui affecte.

Java	Kotlin
`String nom = "nicolas";`	`val nom = "nicolas"`

Déclaration de fonctions

Une fonction se déclare avec le mot clé **fun** suivi du nom de la fonction. Comme en Java on trouve les paramètres entre parenthèses, mais à la différence de Java le type de ce que retourne la fonction vient à la fin :

Java	Kotlin

`int somme(int a, int b) {`
`  return a + b;`
`}`
	`fun somme(a: Int, b: Int) : Int {`
	`  return a + b`
	`}`

On remarquera au passage que le type entier s'écrit **Int** en Kotlin. En effet dans ce langage même les types simples (entiers, booléens, flottants) sont des objets et ont donc droit à leur majuscule.

Déclaration de classes

Les classes Kotlin ressemblent beaucoup aux classes Java mais sont en général plus simples car Kotlin génère tout seul le constructeur par défaut et les getters/setters de variables de classe.

Java	Kotlin

`public class Personne {`
`  String nom;`
`  String prenom;`
 	
`  public Test(String nom, String prenom) {`
`  this.nom = nom;`
`  this.prenom = prenom;`
`  }`
 	
`  public String getNom() {`
`  return nom;`
`  }`
 	
`  public void setNom(String nom) {`
`  this.nom = nom;`
`  }`
 	
`  public String getPrenom() {`
`  return prenom;`
`  }`
 	
`  public void setPrenom(String prenom) {`
`  this.prenom = prenom;`
`  }`
 	
`}`
	`class Personne(var nom: String, var prenom: String) {`
	`}`

Pour créer un nouvel objet, Kotlin se contente de nommer le type de l'objet (et les éventuels paramètres du constructeur) sans utiliser le mot clé **new**.

Java	Kotlin
`Personne moi = new Personne("nicolas", "singer");`	`val moi = Personne("nicolas","singer")`

Collections de type listes

Kotlin et Java partagent la même plateforme d'exécution et on retrouve donc en Kotlin les mêmes objets qu'en java pour gérer les collections. Kotlin offre néanmoins une syntaxe plus simple pour déclarer et parcourir ces collections.

Par exemple voici comment on déclare une liste de deux personnes (avec la classe vue ci-dessus)

```
val moi = Personne("nicolas","singer")
val lui = Personne("adrien","defossez")

val liste = listOf(moi, lui)
```

ou plus simplement une liste de bonbons :

```
val bonbons = listOf("guimauve", "caramel","berlingot")
```

Pour parcourir les listes, Kotlin propose une forme spécifique de la boucle for. Voici par exemple comment on affiche tous les noms de bonbons de la liste ci-dessus :

```
for (bonbon in bonbons) {
    println(bonbon)
}
```

On peut également utiliser les méthodes `forEach`, `map` et `filter` que l'on connait dans tous les langages à dominante fonctionnelle (comme par exemple java 8, mais aussi javascript). Voici par exemple comment utiliser `forEach` pour faire la même chose que la boucle `for` précédente :

```
bonbons.forEach {  bonbon -> println(bonbon) }
```

ou plus simplement (quand il y a un seul argument comme ici, on peut l'omettre et il s'appelle par défaut **it**) :

```
bonbons.forEach {  println(it) }
```

Autre exemple, voici comment on peut utiliser map pour transformer la liste de personnes en une liste de chaines de caractères composées du nom et du prénom, puis afficher cette liste :

```
liste.map { it.prenom + it.nom }.forEach {  println(it) }
```

ou encore en affichant uniquement les personnes ayant pour prénom n*icolas* :

```
liste.filter { it.prenom == "nicolas" }.map { it.prenom + it.nom }.forEach {
println(it) }
```

Collections de type Map

Sans surprise on trouve aussi l'équivalent des `Map` Java. Revisitons par exemple notre liste de contacts en en faisant une `Map` composée de deux clés « nom » et « prenom » ayant chacune une valeur :

```
val moi = mapOf( "prenom" to "nicolas", "nom" to "singer") // { prenom: nicolas,
nom: singer }
```

et pour afficher le nom et prénom de cette personne :

```
moi.forEach { k,v  -> print(k +  ":" + v) } // affiche prenom: nicolas, puis nom:
singer
```

Gestion des *null* en Kotlin

Kotlin propose un type spécial pour les variables pouvant être *null*. Ce type se note en utilisant le signe ? après le type de base.

Par exemple voici comment on déclare une chaine de caractère contenant la valeur *null* :

```
val nom : String? = null
```

Autre exemple plus parlant, voici une méthode qui cherche dans une liste de personnes, la personne ayant pour prénom « fabien », et qui retourne *null* si elle ne la trouve pas. Remarquez bien le symbole ? à la suite du type de retour :

```
fun cherchefabien(personnes : List<Personne>) : Personne? {
    for (personne in personnes) {
        if (personne.prenom == "nicolas") return  personne
    }
    return null
}
```

L'intérêt de cette gestion du type null est de détecter rapidement les erreurs de type *null pointer exception* qui risquent de survenir lors de l'exécution du programme.

Par exemple le programme suivant est incorrect et Kotlin nous le dira avant même la compilation. Etes-vous capable de dire pourquoi ?

```
val moi = Personne("nicolas","singer")
val lui = Personne("adrien","defossez")

val liste = listOf(moi, lui)
val personne = cherchefabien(liste)
println(personne.nom)
```

La raison pour laquelle ce programme ne compilera pas, c'est que la fonction « cherchefabien » peut retourner une valeur nulle (c'est ce que nous dit son type de retour `Personne ?`) et que donc il est dangereux d'utiliser l'expression « `personne.nom` » car `personne` est susceptible d'être *null.* Si cela arrive, votre programme plante avec la fameuse *« null pointer exception »*.

Le programmeur doit donc prendre ses précautions et par exemple reformatter son code pour qu'il devienne :

```
val personne = cherchefabien(liste)
if (personne != null) println(personne.nom)
```

Ce type de test est tellement fréquent, que kotlin nous propose l'opérateur **`?.`** appelé « *type safe* » pour le réaliser plus simplement. L'opérateur **`?.`** s'utilise à la place du « **`.`** » et se situe donc entre une instance d'objet et un appel à une de ses méthodes ou propriétés. Il signifie « n'exécute cette méthode ou ne retourne la valeur de cette propriété que si l'objet n'est pas *null*, sinon ne fait rien et renvoi *null* ».

Voici donc une façon plus simple d'écrire le code précédent :

```kotlin
val personne = cherchefabien(liste)
println(personne?.nom) // évite l'éventuelle null pointer exception
```

Interfaces

Soit une interface dotée d'une seule méthode comme l'exemple ci-dessous :

Java	Kotlin

`public interface Cliquable {`
 	
`    void onclick() ;`
`}`
	`interface Cliquable {`
	`    fun onclick()`
	`}`

Quand, comme c'est le cas dans cette exemple, l'interface ne propose qu'une seule méthode, il est extrêmement simple en Kotlin de créer un objet de ce type tout en implémentant la méthode en question. Regardez la différence en Kotlin et Java :

Java	Kotlin

`Cliquable un = new Cliquable() {`
`    @Override`
`    public void onclick() {`
`        // implémentation de la méthode`
`        System.out.println("je suis cliqué ");`
`    }`
`};`
	`val un = Cliquable { println("je suis cliqué ") }`

Quand l'interface possède plusieurs méthodes c'est un peu plus compliqué. Il faut utiliser le mot clé **object** de Kotlin pour construire un nouvel objet doté des deux méthodes. Par exemple :

Java	Kotlin

`public interface Cliquable {`
 	
`    void onclick() ;`
 	
`    void onLeftClick();`
`}`
	
	`interface Cliquable {`
	`    fun onclick()`
	`    fun onleftclick()`
	`}`

Java	Kotlin

`Cliquable un = new Cliquable() {`
`    @Override`
`    public void onclick() {`
`        // implémentation de la méthode`
`        System.out.println("je suis cliqué ");`
	`val un = object : Cliquable{`
	`    override fun onclick() {`
	`        println("je suis cliqué")`
	`    }`
	
	`    override fun onleftclick() {`

```
    }

    @Override
    public void onLeftClick() {
        System.out.println("je suis
cliqué par le bouton gauche");

    }
};
```

```
        println("je suis cliqué par le
bouton gauche")
    }
}
```

Exercices

1. Définissez une classe Ville possédant deux propriétés publiques et non modifiables : *nom* de type chaine de caractère et *rating* de type entier.
2. Définissez une liste de trois villes que vous initialiserez avec les valeurs que vous voulez.
3. Affichez uniquement le nom des villes qui possèdent un *rating* supérieur à 2.
4. Soit trois villes déclarées littéralement comme suit :

   ```
   val villes = listOf(mapOf("nom" to "Marseille" ,  "rating" to 2) ,
   mapOf("nom" to "Bordeaux" , "rating" to  4), mapOf("nom" to "Toulouse" ,
   "rating" to 5))
   ```

   Ecrivez le code qui, à partir de cette liste, retourne un tableau de trois objets de type `Ville`.
5. Ecrivez le code qui, toujours à partir de la liste `villes`, retourne un tableau d'objet de type Ville ne contenant que les villes de rating supérieur à 2.
6. Ajoutez à la classe `Ville` une méthode appelée `affStars()` qui retourne une chaîne de caractère composée d'autant d'étoiles que la valeur *rating* de la ville.
7. Ecrivez le code qui à partir du tableau `villesTab`, affiche le nom des villes suivi du nombre d'étoiles de toutes les villes dont le rating est supérieur à 2.