Kotlin

Eléments du langage

Variables Types

- Types prédéfinis: Int,Short, Long, Float, Double, Byte, Char, Boolean, String, Array.
- Une variable doit toujours avoir un type de données soit explicitment, soit implicitement lors de son initialisation

```
var v1:Float = 3.4f   //par défaut 3.4 est de type Double
var v2 = 3.4 // de type Double
var v3:Int
```

- Une variable peut être déclarée soit par le mot clé « `var` » ou bien « `val` », dans ce dernier cas on ne peut plus lui réssigner une autre valeur.
- L'opérateur « is » pour tester le type d'une variable, exemple: v2 is Double

Nulabilité

- Approche cliassique Pour tester la nulabité

```
var nombre = 6
        if (nombre != null) {
            nombre = nombre.dec()
            print(nombre)
        }
```

- Approche Kotlin

en kotlin une ne peut être initialisée par null que si on déclare le type de la variable explicitement suivi par l'opérateur ?

```
var nombre: Int? = null
```

Utilisation d'une variable nulable

1. L' opérateur ?.

```
nombre = nombre?.dec() // dec() sera exécutée uniquement si nombre est non null
print(nombre)    // « null » sera affiché
```

2. ?.let

it contient la valeur assignée à nombre et le bloc let{ … } sera exécuté uniquement si nombre est non null

```
nombre?.let{
    it?.dec()
    print(it)
}
```

3. l'opérateur Elvis (?:))

```
nombre=nombre?.dec() ?:0  // Si nombre est null alors il recevra la valeur 0
```

4. Déclencher une exception dans le cas d'une valeur nule avec l'opérateur !!

```
var nombre:Int? = null
nombre=nombre!!.dec()
//java.lang.NullPointerException
print(nombre)
```

Tableaux et listes

- Tableaux

```
val entiers:Array<Int> = arrayOf(44,12,13)
//on pourrait écrire aussi val val entiers = arrayOf(44,12,13)
print(entiers[0])
Pour avoir un tableau contenant des éléments de différents types de données, on crée un tableau de type « Any »
   val valeurs: Array <Any> = arrayOf("AA",2,true)
// ou bien val valeurs = arrayOf("AA",2,true)
    println(valeurs[0])
    print ("Elément 2: ${valeurs[1]}")
```

Structures de contrôle de flux

- Kotlin supporte les structures de contrôle suivantes:
  - if
  - when
  - while,do while
  - for
- if et when sont utilisées comme des instructions (comme en java, when est l'équivalent de l'instruction switch) ou bien comme des expressions dont la valeur peut être affectée à une variable.

if

```
val a=1100
if (a>1000)
 print ("sup à 1000")
else if (a in 100..999)
 print("entre 100 et 999")
else if (a>0)
 print("strictement positif")
else
 print ("un nombre négatif ou nul")
```

```
val a = 1100
val resultat = if (a > 1000)
    "sup à 1000"
else if (a in 100..999)
    "entre 100 et 999"
else if (a > 0)
    "strictement positif"
else
"un nombre négatif ou nul"
print(resultat)
```

when expression

```
val jourSemaine=3
val jour = when(jourSemaine){
    1-> "Lundi"
    2-> "Mardi"
    3-
> {"Mercredi"} // les {} sont obligatoires dans
 le cas de plusieurs lignes
    4-> "Jeudi"
    5 -> "Vendredi"
    6,7 -> "Weekend"
    else -> "Numéro invalide"
}
print (jour)
```

for

```
val fruits = listOf("Pomme", "Orange")
println(fruits)
//
for (element in fruits)
    println(element + " ")


for ((i, e) in fruits.withIndex()) {
    println("position:  $i ,elt: $e")
}

for (i in 1..10) print(i)


for (i in 10 downTo 1) print(i)


for (i in 3..12 step 2) print(i)


for (i in 'a'..'m') print (i)
```

Pour répéter un ensemble d'instructions un certain nombre de fois

```
repeat(10) {
    println("Hello World")
}
```

while

```
var n = 0
while (n < 50) {
    n++
}


do {
    n--
} while (n > 50)
```

Fonctions

```
fun fonction(p1:Int,p2:String="",p3:Int=2): Int
 {
    // instructions
}
```

- Une fonction peut avoir 0 ou plusieurs paramètres
- Un paramètre doit avoir un nom et un type et peut avoir une valeur par défaut.
- Une fonction peut avoir un type de retour
- Une fonction qui ne contient q'une seule intstruction peut être définie comme suit:

```
fun somme (v1:Int,v2:Int)= v1+v2
```

Au lieu d'écrire :

```
fun somme (v1:Int,v2:Int):Int{
    return v1 + v2
}
```

Classes et objets

Définition

```
class Boite {
    var largeur: Int = 10
    var hauteur: Int = 30
    var longueur: Int = 20

   fun volume()= largeur*hauteur *longueur
    }

}
```

Instanciation

```
val paquet = Boite()
print(paquet.volume())
```

Une classe peut avoir un ou plusieurs constructeurs secondaires déclarés dans le corps de la classe

```
class Boite( val largeur : Int,  val hauteur : Int)
{ var longueur:Int=1
// Le mot clé this permet d'appeler et d'initialiser le constructeur primaire
constructor(largeur:Int,hauteur:Int,longueur:Int):this(largeur,hauteur){
        this.longueur=longueur
    }
    fun volume()= largeur * hauteur * longueur
}
```

Une autre définition en utilisant le constructeur primaire

```
class Boite(val largeur : Int, val hauteur : Int, val longueur : Int)
{

    fun volume()= largeur * hauteur * longueur

}
```

Propriétés et méthodes

- visibilité

Par défaut les propriétés et méthodes sont publiques, pour les rendre privées il faut ajouter le modificateur « private » , kotlin supporte aussi les modificateurs « protected » et « internal » (visibilité à l'intérieur d'un même module)

héritage

- Par défaut les classes sont finales « final » et ne peuvent pas être héritées, pour permettre l'héritage à partir d'une classe, le modificateur « abstract » ou bien « open » doit être utilisé.

Classes imbriquées et classes internes

Classe imbriquée

```
class C1 {
    class C2(){
        fun c2_message(){
            print ("Message à partir de la classe C2")
        }
    }
}
```

Appel de la méthode c2_message

```
    C1.C2().c2_message()
}
```

Une classe imbriquée n'a pas un accès aux membres de la classe externe, la classe imbriquée est considérée comme une classe statique

Classe interne

```
class C1 {
    var a=44
  inner  class C2(){
        fun c2_message(){
            print ("valeur de a à partir de C2:$a")
        }
    }
}

   C1().C2().c2_message()
}
```

Une classe interne peut avoir accès aux membres de la classe externe, pour utiliser la classe interne il fau créer une instance de la classe parente.

Gestion des exceptions

La gestion des exceptions dans kotlin est similaire à celle du langage Java

```
try{
    20/0
}
catch(e:Exception){
    println(e.message)
}
finally{
    println ("toujours exécuté")
}
```