Framework de calcul distribué

Chapitre1 Introduction Scala

Mr DIATTARA Ibrahima

Sommaire

Déclaration

Le langage Scala propose différentes façons de déclarer une variable ou une constante

- var valeur peut changer (**variable**)
- val valeur immutable(**constante**))

Interpolation

L' interpolation consiste d'incorporer des références de variables et de les évaluer directement dans une chaîne de caractère

- **Interpoler une référence dans une chaîne**

```
val name="fall"
println("ça va  mister " + name+ " ?" )    //dirty code

println(s"ça va  mister $name ?")
```

- **Interpoler une évaluation dans une chaîne**

```
val a=10
val b=5
println(" la valeur de a+b est "+a+b+5) ????

println(s" la valeur de a+b est  ${a+b+5} ") ?????
```

Fonction & Méthode

- **Fonction**

```
def nom_fonction(param1:type,….paramN:type):[typeRetour]={
        Intruction1
        ……..
        intructionN
        return [typeRetour]
}
def somme(a : Int, b:Int) : Int = {
        return a+b
}
```

Quand on a une seule instruction on peut simplifier une fonction comme suit

```
def nom_fonction(param1:type, ….paramN:type)=...
def somme(a : Int, b:Int) = a+b
```

- **Méthode**

```
def affiche(value : Int) : Unit = {
        print(s"la valeur est $value")
}
```

Boucles/Structures de contrôles

```
for (i <-1 to 5) {
  val carre = i * i
  if (carre % 2 == 0) {
    println(s"Le carré de $i est $carre (pair)")
  } else {
    println(s"Le carré de $i est $carre (impair)")
  }
}
```

```
while (j <= 5) {
  val carre = j * j
  if (carre % 3 == 0) {
    println(s"Le carré de $j est $carre (multiple de 3)")
  } else if (carre % 2 == 0) {
    println(s"Le carré de $j est $carre (pair)")
  } else {
    println(s"Le carré de $j est $carre (impair)")
  }
  j += 1
}
```

Array & List

val l = List(1,2,3)

val tab = Array(1,2,3)

val mat = List(List(1,2), List(1,3)) ou val mat = Array(Array(1,2), Array(1,3))

Pour voir les opérations CRUD des listes ou tab voici le lien

https://www.tutorialspoint.com/scala/scala_lists.htm

Application

En utilisant une boucle for :

- Écrire une fonction pour calculer la somme des éléments d'une liste
- Écrire une fonction pour afficher tous les éléments pairs d'une liste
- Écrire une fonction pour calculer la moyenne d'une liste

Pattern matching

Le pattern matching de Scala possède un cas d'utilisation qui est similaire aux switch-case de Java et de C

```
def  significationCouleur(couleurs:List[String]):Unit={

    for (couleur <- couleurs){
            couleur match {
                case "rouge" | "orange" => println(s"$couleur:danger")
                case "jaune" => println(s"$couleur: c est  la richesse")
                case "blanc" => println(s"$couleur: paix")
                case _=>println (s"$couleur:signification introuvable")
            }
     }
}
```

Récursivité Simple

La récursivité en programmation permet à une fonction de se résoudre en appelant elle-même

Programmation Impérative	Récursivité

def factorial(n: Int): Int = {
  var result = 1
  for (i <- 1 to n) {
  result *= i
  }
  result
}
	def factorial(n: Int): Int = {
	  if (n <= 1) 1
	  else n * factorial(n - 1)
	}

Lambda expression

Expression Lambda fait référence à une expression qui utilise une **fonction anonyme** au lieu d'une variable

```
var l=List(1, 3,2,0)
var mat=List(List(4,6), List(1,3,4))
```

- **Map**

```
l.map(x => e * e )
mat.map(e=>e.map(a=>a*a))
```

- **Reduce**

```
l.reduce((a, b) => a + b)
```

```
mat.map(e=>e.reduce((a, b)=>a+b)).reduce((a,b)=>a+b)
```

```
mat.flatMap(e=>e).reduce((a,b)=>a+b)
```

- **MapReduce & filter**

```
l.map(e=>e*2).filter(e=>e+1>=3).reduce((a,b)=>a+b)
l.map(e=>e*2).filter(_>3).reduce((a,b)=>a+b)
l.filter(_==1).length ou l.filter(e=>e==1).length
```

Exo:

Exo1:Manupilation des structures à une dimension

Les fonctions doivent être codées de 3 **façons itérative , récursive et fonctionnelle(lambda expression)**

- Recherche un élément dans une liste => true si trouvé false sino

  recherche(list :List[Any], elt:Any).....

- Retourne le nombre d'occurrence d'un élément dans un tableau

  nboc(list:List[Int],elt:Int)).....

Mise en Exergue impérative vs fonctionnelle

```scala
object MyListe {

  def nbocIt(list:List[Int],elt:Int):Int={
    var compteur=0
    for(e<-list){
      if(e==elt)
        compteur=compteur+1
    }
    return compteur
  }
  def nbocRec(list:List[Int],elt:Int):Int={
    list match {
      case Nil =>0
      case head::Nil => if(head==elt)  return 1 else  return 0
      case head::tail=> if(head==elt) return 1 + nbocRec(tail, elt)
                        else return 0 + nbocRec(tail, elt)
    }
  }
  def nbocLamEx(list:List[Any], elt:Int) :Int =list.filter(e=>e==elt).length

  def main(args: Array[String]): Unit = {
    val list=List(1,4,3,2,1,1,4)
    println(nbocIt(list, elt = 4))
    println(nbocRec(list, elt = 0))
    println(nbocLamEx(list, elt = 1))
  }
}
```