

Framework de calcul distribué

Chapitre1 Introduction Scala

Mr DIATTARA Ibrahima



Sommaire

1. Introduction
2. Ouverture compte databricks
3. Déclaration
4. Interpolation
5. Fonction & méthode
6. Structure de contrôle /boucles
7. List & Array
8. Pattern matching
9. Optimisation des initis
10. Programmation fonctionnelle Vs impérative
11. La récursivité simple / Terminale(accumulateurs)
12. Fold left & fold right
13. Lambda Expression/Fonction anonyme
14. Fonction implicit
15. Gestion des exceptions (try catch final)

Introduction

Scala intègre les paradigmes de la programmation orientée objet et ceux de la programmation fonctionnelle

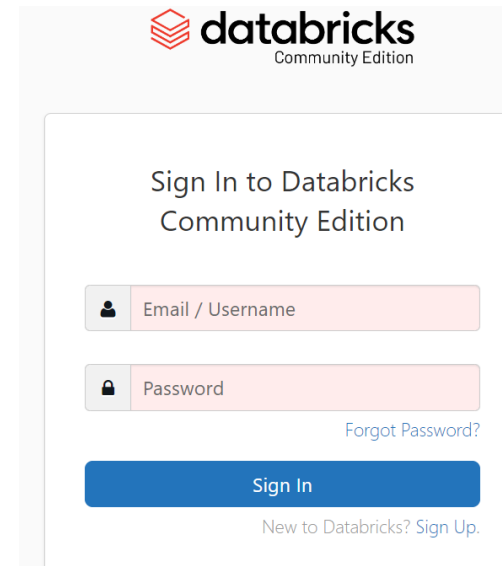
Le Framework **Spark** et le système de messagerie **Kafka** sont développés en Scala

Ouverture Compte Databricks

Databricks vous offre la possibilité de créer:

- ❖ Machine
- ❖ Notebook (Spark Scala , pySpark, R, ..)
- ❖ Un système de fichier DBFS
- ❖

Lien pour ouvrir un compte : <https://community.cloud.databricks.com/login.html>



The screenshot shows the login interface for Databricks Community Edition. At the top, the Databricks logo (a red cube icon) and the text "databricks Community Edition" are displayed. Below this, the heading "Sign In to Databricks Community Edition" is centered. There are two input fields: the first is labeled "Email / Username" and the second is labeled "Password". To the right of the password field is a link that says "Forgot Password?". Below the input fields is a blue "Sign In" button. At the bottom, there is a link that says "New to Databricks? Sign Up."

Déclaration

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

- ❑ **var** valeur peut changer (**variable**)
- ❑ **val** valeur immutable(**constante**)
- ❑ **final** éviter les dérivés (**override**)

```
class Gouv {  
  final val mandat=2  
  val nb_ministre=30  
}  
  
class President extends Gouv {  
  override val nb_ministre=35  
  override val mandat= 3  
}
```

⊕
command-4004476459509344:8: error: type mismatch;
found : Int(3)
required: Int(2)
override val mandat= 3

- ❑ **Lazy** (lazy est toujours accompagné de val): pour faire de l'évaluation paresseuse , maintenir l'évaluation d'une expression jusqu'à ce que sa valeur soit nécessaire=> **optimisation**

Variable Type defined	Variable Type Inference
val a: String = "diattara"	val a = "Foo"

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

	Array	List
Access the ith element	$\theta(1)$	$\theta(i)$
Delete the ith element	$\theta(n)$	$\theta(i)$
Insert an element at i	$\theta(n)$	$\theta(i)$
Reverse	$\theta(n)$	$\theta(n)$
Concatenate (length m,n)	$\theta(n+m)$	$\theta(n)$
Count the elements	$\theta(1)$	$\theta(n)$

Application

- 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 singificationCouleur(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:singification introuvable")  
    }  
  }  
}
```

7 Option[type]

Option[Type] permet d'inviter l'initialisation des variables, afin d'optimiser la mémoire

Elle permet aussi de gérer les exceptions

❑ Optimisation des affectations

```
def prixAchat(prix:Int , bonnus:Option[Int]=None)=prix - bonnus.getOrElse(0)
```

```
prixAchat(3)
```

```
prixAchat(4, Some(1))
```

❑ Gérer les exceptions

```
def divide(x:Double, y:Double) : Option[Double]={  
    y match {  
        case 0 => None  
        case _ => Some(x/y)  
    }  
}
```

```
val a= divide(1, 0).getOrElse("error check your input parametres!!!! ")
```

Récurtivité Simple

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

Programmation Impérative	Récurtivité
<pre>def factorial(n: Int): Int = { var result = 1 for (i <- 1 to n) { result *= i } result }</pre>	<pre>def factorial(n: Int): Int = { if (n <= 1) 1 else n * factorial(n - 1) }</pre>

Récurtivité simple

Dans les architectures modernes, une zone de la mémoire est allouée pour le programme et utilisée de façon particulière : . C'est dans cette zone que sont allouées les variables locales à la fonction

La pile grandit à chaque appel récursif. Si on fait trop d'appels (en particulier mauvais cas de base, on ne s'arrête jamais), la pile dépasse sa taille maximale autorisée \Rightarrow **Erreur Stack Overflow** Par défaut sous linux, la pile fait 8192 octets. Elle peut être agrandie par le système ou l'utilisateur (command `ulimit -s`)

Récurtivité Terminale / Accumulateurs

Dans la récursivité simple, chaque appel récursif s'accumule dans la pile d'appels jusqu'à ce que la condition de base soit atteinte. Cela peut entraîner un débordement de pile .

En revanche, la récursivité terminale est optimisée par le compilateur pour être équivalente à une boucle, éliminant ainsi le besoin d'accumuler des appels sur la pile. Cela permet d'économiser de l'espace mémoire

@tailrec

```
def fact(n: Int, acc: Int = 1): Int = {  
  if (n <= 1)  
    acc  
  else  
    fact(n - 1, n * acc)  
}
```

L'ajout de l'annotation **@tailrec** à une fonction en Scala permet au compilateur de vérifier si la fonction est récursive terminale. Si la fonction satisfait les conditions de récursivité terminale, le compilateur effectuera une optimisation de la pile d'appels pour éviter tout débordement de pile.

Si la fonction n'est pas réellement récursive terminale, le compilateur générera une erreur, indiquant que l'optimisation de récursivité terminale ne peut pas être appliquée.

Fold Left et Foldright

Les fonctions `foldLeft` et `foldRight` sont des opérations de pliage (ou réduction) sur des collections en Scala.

Elles permettent de combiner les éléments d'une collection en appliquant une opération associée à chaque élément.

Ces fonctions sont couramment utilisées dans la programmation fonctionnelle pour effectuer des traitements sur des collections de manière concise et lisible

Fold Left et Foldright

```
1 val nums = List(1, 2, 3, 4)
2 val sum = nums.foldLeft(0){
3   (acc, num) => acc + num
4 }
```

Le déroulé de l'opération foldLeft est le suivant :

```
1 (((((0 + 1) + 2) + 3) + 4))
2 (((((1) + 2) + 3) + 4))
3 (((3) + 3) + 4)
4 ((6) + 4)
5 (10)
```

```
1 scala> val nums = List(1, 2, 3, 4)
2 nums : List[Int] = List(1, 2, 3, 4)
3 scala> val sum = nums.foldRight(0) {(num, acc) =>
4   num + acc
5 }
6 sum : Int = 10
```

En image cela donne:

Acc { 1, 2, 3, 4 }

0 + 1 = 1
{ 2, 3, 4 }

1 + 2 = 3
{ 3, 4 }

3 + 3 = 6
{ 4 }

6 + 4 = 10
10

```
1 (1 + (2 + (3 + (4 + 0))))
2 (1 + (2 + (3 + (4))))
3 (1 + (2 + (7)))
4 (1 + (9))
5 (10)
```

Fold Left et Foldright

```
def nboc(list :List[Int], elt:Int)=list.foldLeft(0){(acc,currentelement)=>if (currentelement==elt) acc + 1 else acc }
```

```
def nboc(list :List[Int], elt:Int)=list.foldRight(0){(acc,currentelement)=>if (currentelement==elt) acc + 1 else acc }
```

```
nboc (List(1,2,1,3,5,1) , 1)
```

```
res1: Int = 3
```

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: Manipulation des structures à une dimension

Les fonctions doivent être codées de 3 façons **itérative**, **réursive** 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)).....

Exo2: Manipulation des Structures à deux dimensions

Les fonctions doivent être codées de deux façons **itérative** et **fonctionnelle(lambda expression)**

- ❖ sommeMatrice retourne la somme de tous les éléments de la matrice

sommeMat(mat:List[List[Int]]).....

- ❖ sommeLigneMat retourne une liste qui contient la somme de chaque ligne d'une matrice

sommeligne(mat:List[List[Int]]).....

!!!!!! Pour les lambda expression limitez vous sur les fonctions filter, map, flatMap, reduce et length

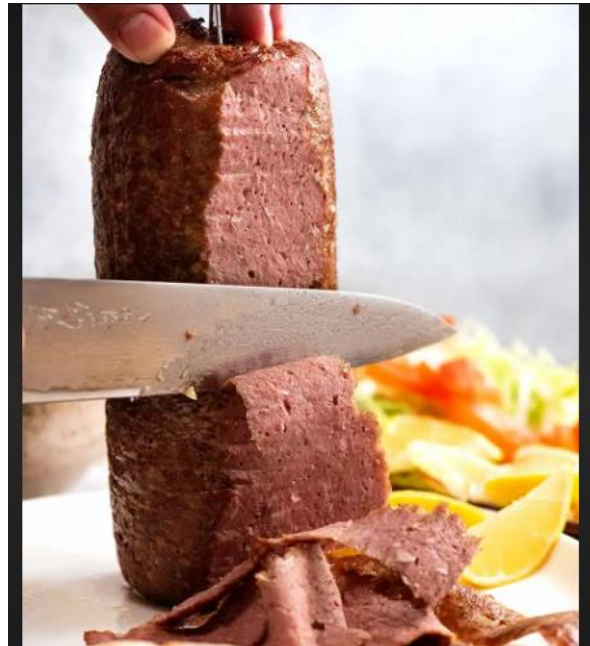
Mise en Exergue impérative vs fonctionnelle

```
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))  
  }  
}
```

- ❑ Le code fonctionnel reflète bien les deux étapes de calcul : **filtrer** l'élément puis **compter**, alors que le code impératif **fait tout en même temps**
- ❑ La programmation fonctionnelle s'affranchit de façon radicale **des effets secondaires** (effets de bord) en **interdisant toute opération d'affectation**, on constate aussi qu'elle est moins **verbeuse**

10 Implicite

L'utilisation des implicits permet aux développeurs de se faciliter la tâche en laissant le compilateur le soin d'aller chercher dans le scope ce qui manque



Devinez ce que Fatou va préparer si tu es intelligent(e) et donne ce qui manque

10 Implicite[variable/constante]

```
implicit val myname= "Diop"
```

```
implicit val myage = 20
```

```
def salam(salutation:String)(implicit name:String, age:Int ) = s"$salutation $name, you are $age"
```

❑ Passage explicite

```
salam("Helo")("diop",28)
```

❑ Passage implicite

```
salam("Hello")
```

Il est impossible d'avoir deux variables ou constantes implicites de même type dans un scope

=> Impossible d'avoir deux variables ou constantes implicites de même type dans une fonction ou méthode

Implicite: [Fonction]

```
case class Personne(nom: String , age:Int)
```

```
implicit def toPersonne(  personne: String) = Personne(personne.split(",")(0),  personne.split(",")(1).toInt )
```

```
def getAgeDans( n:Int, p:Personne ): Int = p.age+n
```

```
val s:String= "ndiaye,28"
```

```
getAgeDans(10, S)
```

```
implicit def toPersonneDasn10(  personne: String ) = Personne( personne.split(",")(0),  personne.split(",")(1).toInt +10 )
```

```
// error
```

Il est impossible d'avoir deux méthodes/fonctions implicites de même **signature** dans un même scope

Gestion des erreurs

```
object DBExample {  
  
  def main(args: Array[String]): Unit = {  
  
    var connection: Connection = null  
    var statement: Statement = null  
    var resultSet: ResultSet = null  
    try {  
      connection = DriverManager.getConnection(url, user, password)  
      statement = connection.createStatement()  
      val query = "SELECT * FROM votre_table"  
      resultSet = statement.executeQuery(query)  
    } catch {  
      case e: Exception =>  
        e.printStackTrace()  
    } finally {  
      if (connection != null) connection.close()  
    }  
  }  
}
```