Tour rapide Scala

11 mars 2019

Table des matières

Introduction

Scala est un langage de prorammation moderne créé à l'EPFL par Martin Odersky (le père de `javac`. Il s'inspire de langages comme Java, Ruby, Smalltalk ou encore Erlang. Le nom Scala vient du mot ń évolutif ż et il est utilisé par de nombreux sites Web dont Twitter, Netflix LunkedIn, et bien d'autres. Ce langage concilie programmation orienté objet d'un côté, et programmation fonctionnelle de l'autre. Ainsi,

— toutes les variables sont des objets et tous les opérateurs sont des méthodes ;
— les fonctions peuvent être transmises en argument à d'autres fonctions.

Il s'agit donc d'un langage très puissant qui concilie ces deux types de programmation. De plus, Scala est un langage léger, concis et lisible (comme Ruby). Néanmoins, il est compilé dans des fichiers de classe empaquété sous forme de fichier 'textttjar' ensuite exécuté par la machine virtuelle Java. Sa concision et sa lisibilité sont des atouts indéniables sachant qu'un développeur passe beaucoup plus de temps à lire du code qu'à l'écrire.

La puissance de Scala fait que c'est un langage très profond, adapté au programmaeur profesionnel. Il est possible d'être productif dès le premier jour, mais tous les jours de nouvelles façons d'écrire du code concis et efficace peuvent être trouvées. De plus, Scala présente une nouvelle manière de voir la programmation ce qui ne peut être qu'une bonne chose.

Notons que Sala est un langage dynamique, mais typé statiquement.

1. Premiers pas

Commençons par un classique *Hello Word*.

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!") /* Hello Word*/
  }
}
```

Nous créons un objet `HelloWord` avec une méthode `main` dans laquelle nous affichons un message. Nous en profitons pour présenter la syntaxe des commentaires (bien sûr, il faudra éviter les commentaires inutiles et évidents).

Scala possède des variables, déclarées avec le mot-clé `var` et des constantes déclarées avec le mot-clé `val`. Les règles à suivre pour nommer les variables sont classiques ; le nom

- peut contenir des lettres, des chiffres et des symboles mais pas un caractère blanc (espace, retour chariot) ;
- ne peut pas commencer par un chiffre ;
- ne doit pas contenir des parenthèses, accolades, crochets, points, virgules, points-virgules, apostrophes ou guillemets ;
- doit être différent des mots-clés du langage.

```
var i: Int = 2
val e: Double = 2.718
val s: String = "Hello Word"
i = 3
```

Scala peut inférer les types, nous ne sommes donc pas obligé de les donner, mais il est plutôt conseillé de les donner.

Nous avons déjà vu un exemple de méthode avec `main`. Nous les déclarons avec le mot-cléé `def`. Nous devons donner les types des arguments et nous pouvons également donner le type de retour de la méthode (`Unit` étant en gros un type `void`). De plus, les méthodes peuvent être surchargées, Scala se chargera de trouver la bonne méthode.

```
def sum(x: Int, y: Int) = x + y
def sum(x: String, y: String) = x + y
def sum(x: Double, y: Double, z: Double) = x + y

val x: Int = sum(1, 2)
val s: String = sum("ab", "cd")
```

2. Boucles, conditions et fermetures

Nous disposons du `if-else` comme structure de contrôle et de *pattern matching*.

```
def f(x: Int): Int = {
  if(x < 0) {
    println("Tu as perdu !")
    return 42
  }
  else if(x > 10){
    println("10")
    return 22
  }
  else {
    println("Tu as encore perdu !")
    return 42
  }
}
```

Et le *pattern-matching* se fait à l'aide des mots-clés `match` et `case`.

```
val i: Int = 5

i match {
  case 0 => println("0 received")
  case 1 => println("1 is good, too")
}
```

```
i match {
  case 1 | 3 | 5 | 7 | 9 => println("odd")
  case 2 | 4 | 6 | 8 | 10 => println("even")
  case _ => println("greater than 10") /* default case **/
}
```

Au niveau des boucles, nous avons les classiques boucles `while` et `for`. En voici deux exemples affichante les entiers de 1 à 10.

```
val i: Int = 1
while(i < 11) {
  println(i)
  i += 1
}
```

```
for(i <- 1 to 10) {
  println(i)
}
```

En Scala, les boucles `while` sont déconseillées. Cependant, elles seront parfois utiles. Il est cependant à noter que la plupart des objets énumérables (listes, tableaux, etc.) possèdent des méthodes permettant d'itérer sur leurs éléments. Ce seront alors leurs méthodes que nous utiliserons la plupart du cas pour itérer sur le contenu d'un de ces objets.

```
val a: List[Int] = List(1, 2, 3)
a.foreach { i =>
  println(i)
}
```

Ce code, en plus d'illustrer cela, contient une fermeture (*closure*), concept dont nous reparlerons plus tard.

3. Classes et objets

La déclaration d'une classe se fait avec le mot-clé `class` suivi du nom de la classe. Ensite, nous créeons des instances de la classe avec le mot-clé `new`. La classe sert alors de modèle pour la création d'objets.

```
class Example(var x: Int) {
  var pseudo: String = "default"
  def exampleMethod: Unit = println(s"${pseudo}, ${x}.")
  def exampleMethod2: Unit = println(s"${this.pseudo}, ${this.x}.")
}

val e1 = new Example(3)
val e2 = new Example(5)
e1.exampleMethod
e2.exampleMethod
```

Le mot-clé `this` permet de faire référence à l'instance courante de la classe. Ainsi, `exampleMethod2` est strictement équivalent à `exampleMethod`. Néanmoins, il est plus simple (et conseillé) de ne pas l'utiliser. Il sera utile lorsque l'on voudra passer l'instance courante en argument.

La méthode `this` correspond d'ailleurs au constructeur de la classe et nous pouvons le surcharger pour en créer d'autres.

```
class Example(var x: Int, var pseudo: String) {
  def this(x: Int) = this(x, "default")
  def exampleMethod: Unit = println(s"${pseudo}, ${x}.")
  def exampleMethod2: Unit = println(s"${this.pseudo}, ${this.x}.")
}

val e1 = new Example(3, "pseudo")
val e2 = new Example(5)
e1.exampleMethod
e2.exampleMethod
```

Il est possible de créeer un objet singleton avec le mot-clé `Object`.

```
object Example {
  def meth(x: Int): Unit = println(x + 42)
}
```

4. Héritage

L'héritage se fait avec le mot-clé `extends` et la redéfinition de méthode avec le mot-clé `extends`.

```
class A(var x: Int) {
  println("Constructeur de A")

  def m1 = println("m1 Classe A")
  def m2 = println("m2 Classe A")
}

class B(x: Int) extends A(x) {
  println("Constructeur de B")

  override def m1 = println("m1 Classe B")
  def m3 = println("m3 Classe B")
}

val a = new A(1)
val b = new B(2)

a.m1
b.m1
a.m2
b.m2
b.m3
/* a.m3 => erreur */
```

Il est également possible de créer des classes abstraites en utilisant le mot-clé `abstract` avant le mot-clé `class`. Une classe abstraite ne pourra pas être instanciée, mais d'autres classes pourront en hériter.

4.1. Traits et héritage multiple

Scala ne gère pas l'héritage multiple. Pour le simuler, nous allon utiliser un outil du langage appelé ń trait ż. Un trait eut être vue comme une classe abstraite particulière. Les habitués de Ruby pourront les voir comme des *mixins*.

Un trait se déclare avec le mot-clé `trait` et contient des méthodes. La différence avec les classes abstraites est qu'il ne peut pas contenir de champs. Les traits nous permettent de programmer des comportements.

```
trait Multipliable {
  def *(x: Double)
}
```

Une classe peut alors étendre un trait. Cette opération se fait à l'aide des mot-clés `extends` et `with`. Le mot-clé `extends` est utilisé si la classe n'hérite de rien et qu'il s'agit du premier trait qu'elle étend, et le mot-clé `with` dans les autres cas.

```
class A

class B extends A
        with Trait1
        with Trait2

class C extends Trait1
```

De plus, nous pouvons restreindre un trait aux classes filles d'une autre classe. De cette manière, seules les sous-classes de cette classe pourront étendre ce trait.

```
trait Carnivore <: Animal{
  def manger(a: NourritureCarnivore)
}
```

Ici, seuls les classes filles de `Animal` (dont `Animal`) peuvent étendre `Carnivore`.

On pourrait par exemple écrire eci.

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

trait Mechant <: Personne {
  def direBonjour = println("Je vous déteste les gars ! ")
}

class Prof extends Peronne with Mechant
```

5. Principes de POO

Écrire du code c'est bien, écrire du code propre et maintenable c'est mieux. C'est pour cela que nous allons essayer d'écrire du code en respectant quelques principes.

5.1. Le principe du DRY

Le principe du DRY (*Don't Repeat Yourself*) ne s'applique pas seulement en POO, mais est assez important. En particulier, un copier-coller de dix lignes de code est très probablement un copier-coller de trop et signifie que quelque chose doit être fait pour extraire ce bout de code et le rendre réutilisable.

5.2. La loi de Démeter et les principes SOLID

À côté de ça, la loi de Démeter et les principes de Démeter peuvent être un peu plus compliqué à comprendre. Le site suivant donne des exemples assez humouristiques et permet de mieux comprendre les tenants et aboutissements de ces principes.

Scala est un langage puissant et aux fonctionnalités multiples ; elles permettent d'écrire du code respectant ces principes plutôt facilement.

6. Et du fonctionnel

Avec Scala, nous pouvons bien sûr faire de la programmation fonctionnelle. Un exemple simple de code serait la factorielle récursive.

```
def fac(x: Int): Int = x match {
  case 0 => return 1
  case _ => return x * fac(x - 1)
}
```

De plus, les fonctions sont des objets de premier ordre, et donc nous pouvons passer une fonction en paramètre à une autre fonction. Nous pouvons par exemple écrire une fonction qui renvoie le résultat de la composée de deux fonctions appliqués à un argument `x`.

```
def Fog(f: Int => Int,
        g: Int => Int,
        x: Int): Int = f(g(x))

val facOfFacOfThree = Fog(fac, fac, 3)
```

Plus encore, nous pouvons avoir des fonctions anonymes grâce aux Fermetures (nous avions bien dit que nous en reparlerions). En effet, les fermetures peuvent être récupérées dans des variables, ce qui nous permet d'écrire ce code.

```
var factor: Int = 10
val multiplier: Int => Int = (i: Int) => i * factor
val x: Int = multiplier(5)
factor = 6
val x: Int = multiplier(5)
```

Ici, nous avons dans la variable `multiplier` une fonction qui prend en paramètre un entier et le multiplie par `factor`. Nous remarquons bien que la fermeture a capturé son environment lexical ; changer la valeur de `factor` ne modifie pas `multiplier` car lors de sa création, la variable `factor` a été copiée.

Finalement, nous pouvons écrire des fonctions d'ordre supérieur. Par exemple, nous pouvons écrire la fonction qui prend en paramètre deux fonctions et renvoie la composée de ces deux fonctions.

```
def Fog(f: Int => Int,
        g: Int => Int): Int => Int =
  {x: Int => f(g(x))}

val fac2: Int => Int = Fog(fac, fac)
val facOfFacOfThree = fac2(3)
```

7. Petit (gros ?) TP

Nous allons simuler le développement d'un aquarium simplifié. Un aquarium contient une liste de poissons.

- Tout poisson possède un attribut `energie` (un entier) qui est initialisé à 10 par défaut (tout poisson dont l'attribut `energie` atteint 0 meurt.
- Un poisson peut être carnivore, herbivore, ou omnivore :
  - un carnivore a besoin de manger un autre poisson pour survivre, si un poisson mange un autre, cet autre disparaît et son énergie est ajoutéée à celle de son mangeur ;
  - un herbivore se contente de manger de l'algue supposée infini dans l'aquarium et augmente son énergie de 2 ;
  - un omnivore peut faire les 2.
- Un poisson peut être mâle, femelle, ou hermaphrodite :
  - un mâle a besoin d'une femelle de la même espèce pour se reproduire ;
  - nne femelle a besoin d'un mâle de la même espèce pour se reproduire ;
  - un hermaphrodite a juste besoin d'un poisson de la même espèce pour se reproduire.

À chaque étape, un poisson va chercher à se reproduire 3 fois avec un poisson au hasard.

Il existe 3 types de poissons.

- Le mérou qui est mono-sexué (mâle ou femelle) et carnivore. À chaque étape, il a une chance sur deux de manger un autre poisson au hasard dans l'aquarium
- La sole qui est hermaphrodite et herbivore. À chaque étape elle a une chance sur deux de manger de l'algue.
- La carpe qui est mono-sexuée et omnivore. À chaque étape elle a une chance sur trois de manger de l'algue et une chance sur six de manger un autre poisson.

Nous pourrons faire :

- Une classe `Aquarium` ;
- Des classes `Merou`, `Sole`, `Carpe` ;
- Des traits `Carnivore`, `Herbivore`, `Omnivore` ;
- Des traits `Male`, `Femelle`, `Hermaphrodite`.

A la fin, notre main devrait se terminer par le code suivant.

```
while(true) { aquarium.update }
```