Scala, la puissance de la POO et de la PF

11 mars 2019

Introduction

Langage de programmation Scala

- Créé à l'EPFL par Martin Odersky.
- Inspiré de Java, Smalltalk, Erlang, etc.
- Utilisé par Twitter, Netflix, etc.

Caractéristiques principales

Un langage aux fonctionnalités multiples

- Objet
- Fonctionnel
- Dynamique
- Typage statique

Pourquoi l'utiliser ?

Les avantages de Scala

- Lisible.
- Concis.
- Léger.
- Profond.

Par des profesionnels, pour des profesionnels.

Le langage Scala

Le classique Hello Word

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

Les variables et les constantes

```
var i: Int = 2 /* Variable */
i = 3

val e: Double = 2.718 /* Constante */
/* e = 2.7 => erreur */

val s: String = "Hello Word"
```

Inférence de type

```
var i: Int = 2 /* On donne le type */
i = 3
var i = 3 /* type inféré */
var sum: Int = i + 2
val s: = "Hello Word"
```

Déclaration de méthodes

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

val x: Int = sum(x, y)
```

Attention

- Types des paramètres obligatoires.
- Type des arguments facultatifs.

Surcharge de méthodes

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

val x: Int = sum(1, 2) /* => 3*/
val s: String = sum("ab", "cd") /* => "abcd" */
```

Attention

Une seule méthode par signature

Boucles et conditions

Condition

```
def f(x: Int): Int = {
  if(x < 0) {
    return 42
  }
  else if(x > 10){
    return 22
  }
  else {
    return 52
  }
}
```

Pattern-matching

```scala
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("> 10") /* default case */
}
```

Boucle while

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

Attention

La boucle while est déconseillée

Boucle for

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

Note

Équivalente à la boucle while précédente, mais avec i locale au for.

Avec les objets énumérables

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

Une petite fermeture ?

Ici, on a une fermeture, on en reparlera plus tard.

Classes et objets

Les classes

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

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

Le mot-clé `this`

Fait référence à l'instance courante de la classe.

Constructeur

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

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

La méthode `this`

La méthode `this` est le constructeur de la classe.

Objet singleton

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

Héritage et redéfinition

```
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")
}
```

Classes abstraites

- Avec `abstract class` plutôt que `abstract`.
- Ne peut être instancié.
- Utilisée pour l'héritage.

Traits et héritage multiple

Scala et l'héritage multiple

- Pas d'héritage multiple de classes en Scala.
- Simulation grâce aux traits.
- Les traits sont en quelque sorte des classes abstraites.
- Pas de champs dans les traits.
- Comme les mixins de Ruby.

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

```
class A

class B extends A
with Trait1
with Trait2

class C extends Trait1
```

La méthode `this`

Le mot-clé `extends` pour la première extension, puis le mot-clé `with`.

En particulier, pas de **`with`** sans **`extends`** !

Restreindre un trait à une classe

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

`Carnivore` est restreint à `Animal` !

Seules les sous-classes de `Animal` peuvent étendre `Carnivore`.

Un exemple

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

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

class Prof extends Peronne with Mechant
```

Le prof est bien évidemment un méchant !

Principes de POO

Pour du code propre et maintenable

- Principe du DRY.
- Loi de Démeter.
- Principes SOLID
- Design Patterns.

Voir le lien suivant pour une explication humouristique des principes SOLID.

`http://www.arolla.fr/blog/2017/02/principes-solid-vie-de-jours/`

La loi de Démeter

Donnez à vos collaborateurs exactement ce qu'ils demandent et ne leur fournissez pas quelque chose qu'ils vont devoir inspecter à la recherche de ce dont ils ont besoin.

En particulier, une classe devrait être vue comme un fournisseur de service plutôt que comme un agrégat d'objet.

Le principe de responsabilité unique (SRP)

Ne faire qu'une seule chose, mais bien la faire ; n'avoir qu'une seule raison de changer.

Permet d'avoir du code plus robuste, plus maintenable et plus testable.

Le principe ouvert/fermé (OCP)

Une classe doit être ouvert à l'extension, mais fermée à la modification.

Pour rajouter une fonctionnalité, le code source doit pouvoir être étendu sans modification du code précédent.

Une des idées est qu'une fois le foncionnement du code validé et testé, on ne modifie plus ce code.

Le principe de substitution de Liskov (LSP)

Si V est une sous-classe de T, alors on doit pouvoir remplacer n'importe quel objet du type T par un objet du type V, sans causer de problèmes.

En particulier, toutes les méthodes implémentées par T le sont par V et elles sont aussi valides.

Le principe de ségrégation des interfaces (ISP)

Aucun client ne devrait dépendre de méthodes qu'il n'utilise pas.

Il faut donc diviser les interfaces en interfaces plus petites (un peu associée à la loi de Démeter et au principe de responsabilité unique).

Le principe d'inversion des dépendances (DIP)

Les modules de haut-niveau ne doivent pas dépendre des modules de bas-niveau, les deux doivent dépendre d'abstraction.

Les abstractions ne doivent pas dépendre des détails, mais l'inverse. Le code doit dépendre de l'abstraction et pas de l'implémentation (un peu avec la Loi de Démeter).

Par exemple, une `Voiture` ne doit pas être dépendant d'un `Moteur`, mais d'une abstraction entre la voiture et le moteur. Ce n'est pas la voiture qui décide du moteur qu'elle doit embarquer, c'est l'objet qui construit la voiture qui le fait.

Et du fonctionnel...

La factorielle récursive

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

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

Fog(fac, fac, 3)
```

Fonctions en paramètre

Ici, on crée une fonction qui compose deux fonctions.

Fermetures et fonctions anonymes

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

Capture de l'environment

Changer la valeur de `factor` ne modifie pas `multiplier`.

Fonction d'ordre supérieur

```
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)
```

Fonction de composition

On crée la fonction de composition qui prend en paramètre deux Fonctions et en renvoie une troisième.

Concurrence en Scala avec Akka

Pourquoi un Framework ?

- Sémantique de haut-niveau pour la concurrence.
- Pas de manipulation directe de threads.

Et donc ?

- Du code plus robuste.
- Du code plus lisible.
- Du code plus fiable.

Le modèle d'acteur

- Modèle défini en 1973, popularisé par Erlang.
- Travail réparti entre des entités (acteurs) s'exécutant en parallèle.
- Acteurs isolés, information circule sous forme de message.
- Acteurs modifient données des messages, envoient des message et créent des acteurs.

Métaphore de la vie réelle : personnel d'une entreprise travaillent en parallèle et discutent (envoi de message) pour échanger de l'information.

- Adapté aux systèmes distribués.
- Les acteurs sont supervisés par leurs parents et en cas de mort prématurée, propagation aux parents.
- Permet de relancer facilement en cas de crash (système auto-réparant ?)

Et concrètement

- Un `ActorSystem` pour gérer les acteurs (un contexte).
- Des acteurs (des classes étendant le trait `Actor`).
- Doivent définir la méthode `receive`.
- Création d'acteurs avec la méthode `actorOf`.

Premier exemple

```
import akka.actor.{Actor, ActorSystem, Props}

class FirstActor extends Actor {
  def receive = {
    case "hello" => println("hello word.")
    case _       => println("bad message.")
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[HelloActor], "helloactor")
  helloActor ! "hello"
  helloActor ! "boo"
}
```

Et donc ?

- Dans le code précédent, un seul acteur, et les messages sont des chaînes de caractères.
- Les messages peuvent être n'importe quoi ; on utilise la puissance du pattern-matching.
- Les acteurs ont accès à l'émetteur du message à travers sender et au contexte à travers context.
- Gestion d'erreur => quand une exception a lieu, par défaut, le superviseur stoppe l'acteur et le relance.
- Acteur stoppés avec la méthode `stop`.

Un exemple un peu plus complexe

```
case object PingMessage
case object PongMessage
case object StartMessage
case object StopMessage
```

Deux acteurs vont s'envoyer des messages (un petit ping-pong), les messages étant ici des `case object`.

La classe `Ping`

```
class Ping(pong: ActorRef) extends Actor {
  var count = 0
  def incrementAndPrint { count += 1; println("ping") }

  def receive = {
    case StartMessage =>
      incrementAndPrint
      pong ! PingMessage
    case PongMessage =>
      incrementAndPrint
      if (count > 99) {
        sender ! StopMessage
        println("ping stopped")
        context.stop(self)
      }
      else
        sender ! PingMessage
    case _ => println("Ping got something unexpected.")
  }
}
```

La classe Pong

```
class Pong extends Actor {
  def receive = {
    case PingMessage =>
      println(" pong")
      sender ! PongMessage
    case StopMessage =>
      println("pong stopped")
      context.stop(self)
    case _ => println("Pong got something unexpected.")
  }
}
```

Un poil plus simple que la classe `Ping`. Se contente d'envoyer un message de Pong à la classe qui lui a envoyé le message de Ping, et de s'arrêter si demandé (message Stop).

Le programme

```
object PingPongTest extends App {
  val system = ActorSystem("PingPongSystem")
  val pong = system.actorOf(Props[Pong], name = "pong")
  val ping = system.actorOf(Props(new Ping(pong)), name = "ping")

  ping ! StartMessage
}
```

Résultat ?

On obtient un code simple et facile à comprendre. Essayez d'écrire le même code avec des threads et des mutex !

Conclusion

Scala est une Porsche... Ou une Lamborghini !

- Langage léger, concis, lisible.
- Langage puissant, permettant diverses approches de la programmation.
- Pas vraiment adapté à l'apprentissage de la programmation ?
- Langage riche ; permet d'être productif dès le premier jour, tout en continuant à apprendre tous les jours.

Un petit TP pour la route ?

- Gestion d'un aquarium.
- Poisson avec une énergie (initialement à 10), meurt si 0.
- Poissons carnivores, herbivores ou omnivores.
- Poissons mâles, femelles ou hermaphrodites.
- Poissons mangent et cherchent à se reproduire.
- Trois types de poisson : mérou, sole et carpe.

Où donc le trouver !

Pour plus d'informations sur le TP, consulter le dossier `triplek/cours` du Github du cours. Le TP au complet est à la fin du fichier `cours.pdf`.