

# Fiche-Triche pour Scala

Julien Grange  
<julien.grange@lac1.fr>



## Généralités

Scala a été développé en 2004 ; il s'agit du petit frère sous stéroïdes de Java.

## Caractéristiques

- Compilation vers du bytecode Java
- Typage statique
- Orienté objet
- Fonctionnel

## Exécution

Compilation en bytecode via  
`scalac Source.scala`

puis exécution via  
`scala Source`

Il faut pour cela avoir déclaré une méthode `main` dans un objet :

```
object Main {  
  def main(args:Array[String]) = {  
    ...  
  }  
}
```

## Syntaxe élémentaire : différences avec Java

Déclaration : `var a:Int = 42` mutable  
`val b:String = "foo"` non mutable

On préférera toujours `val` quand c'est possible.

L'égalité `==` est un alias de la méthode `equals()`. Pour l'égalité de référence, on utilisera `eq()`.

## Méthodes et fonctions

### Méthodes

Les méthodes sont définies via le mot-clef `def` :

```
def cat(s1:String, s2:String) : String = {  
  s1+s2  
}
```

Différences avec Java :

- symbole = après le `def`
- pas besoin de `return` : la dernière valeur calculée est renvoyée
- les méthodes imbriquées sont autorisées

La méthode `cat` n'est pas une valeur. L'expression `cat` n'a donc pas de sens en elle-même, et est transformée en l'appel `cat()` !

### Fonctions

Contrairement aux méthodes (morceaux de code), les fonctions sont des objets.

On peut définir une fonction `funcat` en utilisant une lambda-expression :

```
val funcat : (String,String) => String =  
  (s1,s2) => s1+s2
```

Contrairement à `cat`, `funcat` est un objet de type

```
(String,String)=>String,  
qui est un synonyme de  
Function2[String,String,String]
```

Il existe un type `FunctionX[...]`, paramétré par `X+1` types (type des arguments puis type de retour), pour chaque `X` entre 0 et 22.

On aurait pu transformer la méthode `cat` en fonction, en écrivant

```
val funcat = cat _ ou  
val funcat = (x:String,y:String) => cat(x,y)
```

## Curryfication

On peut vouloir appliquer partiellement la fonction `funcat`.

Dans ce cas, on peut définir par exemple

```
val currycat = (s1:String) => (s2:String) => s1+s2  
(les => sont associées à droite), de type String=>(String=>String),  
i.e. Function1[String,Function1[String,String]]
```

On peut alors écrire

```
val prefixe = currycat("pref")  
prefixe("ixe") évalué à "prefixe"
```

On aurait pu définir directement

```
val currycat = funcat.curried ou  
val prefixe = cat("pref", _ )
```

## apply()

Si `f` est une fonction (et pas une méthode), un appel `f(x)` est traduit en `f.apply(x)`, où `apply` est une méthode de l'objet `f`. Réciproquement, tout objet possédant une méthode `apply` peut être appelé sur un argument, de la même manière qu'une fonction.

## Classes

On doit définir un constructeur principal, intégré à la déclaration de la classe. On peut exiger des conditions sur les arguments du constructeur.

Pour créer une classe représentant des points non-mutables dans le quadrant du plan où l'abscisse et l'ordonnée sont positives :

```
class Point(val x:Double, val y:Double) {  
  require(x>=0 && y>=0)  
  private val dist = Math.sqrt(x*x + y*y)  
  def this(x:Double) = this(x,x)  
  def plusLoinQue(that:Point) = this.dist >= that.dist  
}
```

Ici, on a déclaré

- deux attributs publics `x` et `y`
- un attribut privé `dist`
- une méthode `plusLoinQue()`
- un deuxième constructeur attendant un seul `Double`

Pour créer deux points et comparer leurs distances à l'origine :

```
val p1 = new Point(1)  
val p2 = new Point(0,2)  
p1.plusLoinQue(p2) évalué à false
```

Si l'on essaie de créer un `Point` hors du quadrant (avec un `x` ou un `y` négatif), une `IllegalArgumentException` est levée.

## Objet compagnon

L'objet compagnon d'une classe est un objet (singleton) ayant le même nom. On peut y définir les méthodes statiques de la classe. Par exemple, pour écrire une méthode-usine créatrice de points :

```
object Point {  
  def apply(x:Double,y:Double) = new Point(x,y)  
}
```

On peut alors utiliser l'objet `Point` comme une fonction, et écrire

```
val p3 = Point(2,3)
```

## Héritage

Comme en Java, l'héritage se fait via le mot-clef `extends` :

```
class ColoredPoint(x:Double, y:Double, val col:String)  
  extends Point(x,y)
```

Il faut déclarer la réimplémentation des méthodes en ajoutant le mot-clef `override` devant `def`.

## Case classes et pattern matching

```
sealed abstract class EntierOuCaractere  
case class E(n:Int) extends EntierOuCaractere  
case class C(c:Char) extends EntierOuCaractere  
def which(x:EntierOuCaractere) = x match {  
  case E(x) if x%3==0 => "un entier divisible par 3"  
  case E(_) => "un autre entier"  
  case C(c) => "le caractère " + c  
  case _ => throw new Exception("Impossible")  
}
```

`sealed` empêche la création de nouveaux `case` : on sait donc que notre matching est exhaustif.

Pour le type `Option[T]`, deux `case` existent : `Some[T]` et `None`

## Tuple

Les tuples sont non-mutables.

Un tuple `var t = (42,"foo")` est de type `(Int,String)`.

On peut lire ses coordonnées via `t._1` et `t._2`.

## List, HashMap et classes génériques

### List

`List[T]` est le type des listes non-mutables d'objets de type `T`. `List` est donc une classe générique, paramétrée par le type `T`.

On peut faire du pattern matching sur les listes (ici, la fonction est paramétrée par le type `T`) :

```
def somme[T](l: List[T], eval: T=>Int):Int = {  
  1 match {  
    case Nil => 0  
    case tete::queue => eval(tete)+somme(queue,eval)  
  }  
}
```

Opérations et méthodes sur les `List[T]` :

<code>x::l</code>	ajout en tête
<code>l1:::l2</code>	concaténation
<code>length(): Int</code>	longueur
<code>isEmpty(): Boolean</code>	true si la liste est vide
<code>contains(x:T): Boolean</code>	appartenance
<code>exists(p:T=&gt;Boolean): Boolean</code>	existence d'un témoin
<code>forall(p:T=&gt;Boolean): Boolean</code>	chaque élément de la liste vérifie le prédicat
<code>filter(p:T=&gt;Boolean): List[T]</code>	éléments vérifiant p
<code>map[U](f:T=&gt;U): List[U]</code>	liste des images par f
<code>foldLeft[U](x:U)(f:(U,T)=&gt;U): U</code>	application récursive de f à partir de x, gauche→droite

## HashMap

```
import scala.collection.immutable.HashMap
```

`HashMap[K,V]` représente les tableaux associatifs dont les clefs sont de type K, et le valeurs de type V.

Quelques méthodes de `HashMap[K,V]` :

<code>get(k:K): Option[V]</code>	<code>Some(valeur pour k)</code> ou <code>None</code> si pas de valeur
<code>apply(k:K): V</code>	valeur associée à k, ou <code>NoSuchElementException</code>
<code>contains(k:K): Boolean</code>	appartenance
<code>updated(k:V,v:V): HashMap[K,V]</code>	ajout/update pour la clef k
<code>removed(k:V): HashMap[K,V]</code>	retrait de la clef k
<code>transform[W](f:(K,V)=&gt;W) : HashMap[K,W]</code>	calcule des nouvelles valeurs via f

Puisque la classe est non mutable, les méthodes d'ajout et de retrait renvoient une nouvelle `HashMap`.

## Égalité et hachage

En Scala, `==` appelle `equals()`. Pour l'égalité d'adresse mémoire, il faut utiliser la méthode `eq`.

Pour réimplémenter `equals(that:Any)`, il est pratique de faire du pattern matching sur le type de l'argument :

```
override def equals(that:Any): Boolean = {
  that match {
    case that:MaClasse => attr == that.attr && ...
    case _ => false
  }
}
```

Lorsqu'on réimplémente `equals()`, il faut réimplémenter `hashCode()` au passage, pour que deux objets égaux aient le même `hashCode()`. Il faut que `hashCode()` disperse au maximum les valeurs, pour garantir des performances raisonnables quand on utilise des `HashMap`, `HashSet`, etc.

```
override def hashCode(): Int = {
  (this.n1 + 23) * 23 + n2
}
```

Prudence : on ne peut pas se baser sur les attributs mutables des objets, au risque de ne pas les retrouver dans les structures de données utilisant le hachage...

## Traits

C'est l'équivalent des interfaces de Java. Avantage : ils peuvent contenir des méthodes concrètes.

Cela permet par exemple de factoriser du code une bonne fois pour toutes :

```
trait Comparable[T] {
  def <(that: T): Boolean // méthode abstraite
  def <=(that: T) = this < that || this == that
  def >=(that: T) = !(this < that)
  def >(that: T) = !(this <= that)
}
```

En implémentant une classe C, on peut ensuite se contenter de la déclarer en

```
class C extends Comparable[C]
```

et d'implémenter la méthode `<` : les trois autres méthodes sont alors immédiatement héritées du trait `Comparable[C]`.

## Traits mutliples

On peut hériter de plusieurs traits :

```
class Personne(val nom: String) {
  ...
}
trait Salarie {
  def revenus(): Double
}
class Smicard(nom:String)
  extends Personne(nom) with Salarie {
  override def revenus() = 1300
}
trait ToucheAPL extends Salarie {
  abstract override def revenus() = super.revenus()+200
}
trait TouchePrime extends Salarie {
  abstract override def revenus() = super.revenus()*1.1
}
class Doctorant(nom:String) extends Smicard(nom)
  with ToucheAPL with TouchePrime
```

Les `revenus()` d'un `Doctorant` sont alors de 1650 : les méthodes `revenus()` sont ordonnées de droite à gauche parmi les traits et classes dont `Doctorant` hérite. Autrement dit, un appel à `revenus()` sur un `Doctorant` entraîne, dans l'ordre, la pile d'appels

```
Doctorant::revenus()
TouchePrime::revenus()
ToucheAPL::revenus()
Smicard::revenus()
```

On obtient donc  $(1300+200)*1.1$ . En revanche, si l'on avait déclaré

```
class Doctorant(nom:String) extends Smicard(nom)
  with TouchePrime with ToucheAPL
```

les appels auraient été

```
Doctorant::revenus()
ToucheAPL::revenus()
TouchePrime::revenus()
Smicard::revenus()
```

et l'appel à `revenus()` aurait renvoyé  $(1300*1.1)+200$  soit 1630.