

Un tutoriel de Scala

pour les programmeurs Java

Version 1.3
23 mars 2011

Michel Schinz
Philipp Haller

Version française :
Pierre-André Mudry

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Introduction

Ce document donne une brève introduction au langage de programmation Scala et à son compilateur. Il est destiné à des personnes ayant une expérience en programmation et qui désirent une vue d'ensemble de ce qu'il est possible de faire avec Scala. Une connaissance de base de la programmation orientée-objet, particulièrement en Java, est assumée.

2 Un premier exemple

En guise de premier exemple nous allons utiliser le classique programme *Hello world*. Il n'est pas très fascinant mais permet de démontrer facilement l'utilisation des outils Scala sans trop de connaissances du langage. Voici à quoi cela ressemble :

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

La structure de ce programme est certainement familière pour les programmeurs Java : elle consiste en une méthode nommée `main` prenant comme argument depuis la ligne de commande un tableau de chaînes de caractères ; le corps de cette méthode consiste en un appel à la méthode prédéfinie `println` ayant comme argument la sympathique salutation. La méthode `main` ne retourne pas de valeur (c'est une procédure). Pour cette raison, il n'est pas nécessaire de déclarer un type de retour.

La partie moins familière pour les programmeurs Java est sans doute la déclaration **object** contenant la méthode `main`. Une telle déclaration introduit ce qui est traditionnellement désigné sous le terme d'*objet singleton*, c'est-à-dire une classe possédant une seule instance. La déclaration ci-dessus déclare en même temps une classe nommée `HelloWorld` ainsi qu'une instance de cette classe, également nommée `HelloWorld`. Cette instance est créée à la demande lorsqu'elle sera utilisée pour la première fois.

Le lecteur attentif aura peut-être remarqué que la méthode `main` n'est pas déclarée comme `static` ici. Cela s'explique par le fait que les membres statiques (méthodes ou champs) n'existent pas en Scala. Plutôt que de définir des membres statiques, le programmeur Scala déclare ces membres dans un objet singleton.

2.1 Compiler l'exemple

Pour compiler l'exemple ci-dessus, nous utilisons `scalac`, le compilateur Scala. `scalac` fonctionne comme la plupart des compilateurs : il prend un fichier source comme

argument, éventuellement quelques options et produit alors un ou plusieurs fichiers objet. Ces fichiers objet sont des classes Java standard.

Si nous sauvegardons le fichier ci-dessus dans un fichier nommé `HelloWorld.scala`, nous pouvons le compiler grâce aux commandes suivantes (le signe plus grand que `'>'` représente l'invite de commande¹ et ne doit pas être tapé) :

```
> scalac HelloWorld.scala
```

Cela va générer quelques fichiers de classe dans le répertoire courant. L'un de ces fichiers sera nommé `HelloWorld.class` et contiendra une classe pouvant être exécutée directement en utilisant la commande `scala`, comme montré dans la section suivante.

2.2 Exécuter l'exemple

Une fois compilé, un programme Scala peut être lancé en utilisant la commande `scala`. Son usage est très similaire à celui de la commande `java` utilisée pour lancer les programmes Java et accepte les mêmes options. L'exemple ci-dessus peut être exécuté en utilisant la commande suivante, qui produit le résultat attendu :

```
> scala -classpath . HelloWorld
```

```
Hello, world!
```

3 Interaction avec Java

L'une des forces de Scala est qu'il permet facilement d'interagir avec du code Java. Toutes les classes du paquetage `java.lang` sont importées par défaut alors que les autres doivent être importés explicitement.

Examinons maintenant un exemple démontrant cela. Nous désirons obtenir et formater la date actuelle en fonction des conventions utilisées dans un pays spécifique, disons la France².

Les bibliothèques de Java définissent de puissantes classes utilitaires, comme `Date` ou `DateFormat`. Étant donné que Scala interopère sans heurts avec Java, il n'est pas nécessaire d'implémenter des classes équivalents dans les bibliothèques Scala – nous pouvons simplement importer les paquetages Java correspondants :

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._
```

1. *Ndt* : traduction de *shell prompt* en français.

2. D'autres régions comme la suisse romande utilisent les mêmes conventions.

```
object FrenchDate {  
  def main(args: Array[String]) {  
    val now = new Date  
    val df = getDateInstance(LONG, Locale.FRANCE)  
    println(df format now)  
  }  
}
```

La manière d'importer les paquetages en Scala est très similaire à son équivalent Java, bien que plus puissant. En effet, de multiples classes peuvent être importées depuis le même paquetage en les entourant dans des accolades comme sur la première ligne. Une autre différence est, lorsque l'on importe tous les noms d'un paquetage ou d'une classe, il faut utiliser le caractère d'espace souligné (`_`) à la place de l'astérisque (`*`). Cela est provient du fait que l'astérisque est un identificateur valide en Scala (par exemple pour un nom de méthode) comme nous allons le voir plus loin.

L'instruction d'importation à la troisième ligne importe ainsi tous les membres de la classe `DateFormat`. Cela rend la méthode statique `getDateInstance` ainsi que le champ statique `LONG` directement visibles.

À l'intérieur de la méthode `main` nous créons premièrement une instance de la classe Java `Date` qui, par défaut, contient la date actuelle. Ensuite, nous définissons un format de date utilisant la méthode statique `getDateInstance` qui a été importée précédemment. Finalement, nous imprimons la date actuelle en fonction de l'instance localisée `DateFormat`. Cette dernière ligne montre une propriété intéressante de la syntaxe de Scala : les méthodes prenant un seul argument peuvent être utilisées avec une syntaxe infixée. Ainsi, l'expression

```
df format now
```

est simplement une autre version un peu moins prolix de l'expression

```
df.format(now)
```

Cela peut sembler un détail syntaxique mineur, mais cela a d'importantes conséquences, dont une qui sera explorée dans la section suivante.

Pour conclure cette section au sujet de l'intégration avec Java, il doit être noté qu'il est également possible d'hériter de classes Java et implémenter des interfaces Java directement en Scala.

4 Tout est un objet

Scala est un langage orienté-objet pur dans le sens que *tout* est un objet, nombres et fonctions y compris. Il diffère en cela de Java, car Java fait la distinction entre les types primitifs (comme les `boolean` et les `int`) et les types référencés et ne permet pas la manipulation de fonctions comme des valeurs.

4.1 Les nombres sont des objets

Comme les nombres sont des objets, ils possèdent également des méthodes. En fait, une expression arithmétique comme :

$$1 + 2 * 3 / x$$

consiste exclusivement dans des appels de méthodes car elle est équivalente à l'expression suivante :

$$(1).+(((2).*(3))./(x))$$

Cela signifie également que `+`, `*`, ... sont des identificateurs valides en Scala.

Les parenthèses autour des nombres dans la seconde version sont nécessaires car le lecteur de Scala utilise la règle de la plus grande correspondance pour les jetons³. Ainsi, l'expression suivante :

$$1.+(2)$$

est découpée dans les jetons `1.`, `+` et `2`. La raison pour laquelle ce découpage a été choisi est que `1.` est une correspondance valide plus longue que `1`. Le jeton `1.` est interprété comme le littéral `1.0`, faisant de celui-ci un `Double` plutôt qu'un `Int`. Écrire l'expression comme :

$$(1).+(2)$$

empêche `1` d'être interprété comme un `Double`.

4.2 Les fonctions sont des objets

Fait peut-être un peu plus surprenant pour le programmeur Java, les fonctions sont également des objets en Scala. Il devient ainsi possible de passer des fonctions en argument, de les stocker sous formes de variables et de les retourner depuis d'autres fonctions. Cette capacité à manipuler les fonctions comme des valeurs est l'une des pierres d'achoppement d'un paradigme de programmation très intéressant nommé *programmation fonctionnelle*.

Comme exemple simple expliquant pourquoi il peut être utile d'utiliser des fonc-

3. *Ndt* : traduction de *token*

tions comme valeurs, considérons une fonction de minuteur (*timer*) dont le but est de réaliser une certaine action chaque seconde. Comment lui passons-nous cette action? De manière assez logique en tant que fonction. Ce type simple de passage de fonction est vraisemblablement familier à beaucoup de programmeurs : il est souvent utilisé dans les interfaces utilisateurs afin d'enregistrer les fonctions de *call-back* qui sont appelées lorsqu'un événement survient.

Dans le programme suivant, la fonction minuteur est appelée `oncePerSecond` et elle reçoit une fonction de *call-back* comme argument. Le type de cette fonction est écrit `() => Unit` et correspond au type de toutes les fonctions ne prenant aucun argument et ne retournant rien (le type `Unit` est similaire à `void` en C/C++). La fonction principale de ce programme appelle simplement ce minuteur avec un *call-back* qui imprime une phrase sur le terminal. En d'autres termes, ce programme affiche sans fin la phrase "time flies like an arrow" chaque seconde.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread.sleep(1000) }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Notez qu'afin d'imprimer la chaîne de caractères, nous avons utilisé la méthode prédéfinie `println` à la place de celle provenant de `System.out`.

4.2.1 Fonctions anonymes

Bien que ce programme soit très simple à comprendre, il peut être affiné un peu. Premièrement, notez que la fonction `timeFlies` n'est définie qu'afin de pouvoir être passée ensuite à la fonction `oncePerSecond`. Devoir nommer cette fonction, qui n'est utilisée qu'une seule fois, peut sembler superflu et il serait intéressant de pouvoir construire cette fonction uniquement afin de la passer à `oncePerSecond`. Cela est possible en Scala par le truchement des *fonctions anonymes* qui sont exactement ce que leur nom indique : des fonctions sans nom. La version révisée de notre programme minuteur utilisant une fonction anonyme à la place de `timeFlies` est ainsi :

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread.sleep(1000) }
  }
}
```

```
def main(args: Array[String]) {  
  oncePerSecond(() =>  
    println("time flies like an arrow..."))  
}  
}
```

La présence d'une fonction anonyme dans cet exemple est révélée par la flèche '=>' séparant les arguments de la fonction de son corps. Dans cet exemple, la liste d'arguments est vide, comme on peut le voir sur la paire de parenthèses vide à gauche de la flèche. Le corps de la fonction est le même que dans le code de `timeFlies` ci-dessus.

5 Classes

Comme on peut le voir ci-dessus, Scala est un langage orienté-objet et possède en tant que tel le concept de classe⁴. Les classes en Scala sont déclarées en utilisant une syntaxe proche de celle du Java. Une différence importante est toutefois que les classes en Scala peuvent avoir des paramètres. Ceci est illustré sur la définition suivante de nombres complexes :

```
class Complex(real: Double, imaginary: Double) {  
  def re() = real  
  def im() = imaginary  
}
```

Cette classe pour les complexes prend deux arguments qui sont la partie réelle et la partie imaginaire du complexe. Ces arguments doivent être passés lors de la création de l'instance de la classe `Complex` comme suit : `new Complex(1.5, 2.3)`. Cette classe contient deux méthodes, nommées `re` et `im`, donnant accès à ces deux parties.

Il faut noter que le type de retour de ces deux méthodes n'est pas donné explicitement. Il sera inféré automatiquement par le compilateur, qui va examiner la partie droite de ces méthodes et déduire qu'elles retournent les deux une valeur de type `Double`.

Le compilateur n'est pas toujours à même d'inférer les types comme il fait ici et il n'y a malheureusement pas de règle simple permettant de savoir quand il pourra le faire ou non. Dans la pratique, cela ne constitue pas un problème car le compilateur indiquera lorsqu'il ne sera pas à même de trouver un type non donné explicitement. Comme règle simple, les programmeurs débutant en Scala devraient essayer d'omettre les déclarations de type semblant simple à déduire du contexte et voir si le compilateur les accepte. Après un certain temps, il devient alors possible d'avoir

4. Afin d'être complet, il convient de noter que certains langages orienté-objet ne possèdent pas le concept de classe. Scala ne fait pas partie de ces langages.

un bon sentiment de quand il devient possible d'omettre les types et de quand il est nécessaire de les spécifier explicitement.

5.1 Méthodes sans arguments

Un petit problème lié aux méthodes `re` et `im` est qu'afin de les appeler il est nécessaire de mettre une paire de parenthèses après leur nom, comme le montre l'exemple suivant :

```
object ComplexNumbers {  
  def main(args: Array[String]) {  
    val c = new Complex(1.2, 3.4)  
    println("imaginary part: " + c.im())  
  }  
}
```

Il serait plus élégant de pouvoir accéder les parties réelle et imaginaire comme s'il s'agissait de champs, c'est-à-dire sans mettre de parenthèses. Cela est parfaitement faisable en Scala, simplement en les définissant comme *méthodes sans arguments*. De telles méthodes diffèrent des méthodes avec zéro arguments en ceci qu'elles n'ont pas de parenthèses après leur nom, ni dans leur définition, ni dans leur usage. Notre classe `Complex` peut ainsi être réécrite comme suit :

```
class Complex(real: Double, imaginary: Double) {  
  def re = real  
  def im = imaginary  
}
```

5.2 Héritage et redéfinition

Toutes les classes en Scala héritent d'une super-classe. Lorsque aucune super-classe n'est spécifiée, comme dans le code de `Complex` de la section précédente `scala.AnyRef` est utilisé de manière implicite.

Il est possible de redéfinir les méthodes héritées depuis une super-classe en Scala. Toutefois il est nécessaire de spécifier explicitement qu'une méthode en redéfinit une autre en utilisant le modificateur **`override`**, ceci afin d'éviter les redéfinitions accidentelles. En guise d'exemple, notre classe `Complex` peut être augmentée avec une redéfinition de la méthode `toString` héritée de la classe `Object` :

```
class Complex(real: Double, imaginary: Double) {  
  def re = real  
  def im = imaginary  
  override def toString() =  
    "" + re + (if (im < 0) "" else "+") + im + "i"  
}
```


6 Les case classes et le pattern matching ⁵

Un type de structure de données apparaissant souvent dans des programmes est l'arbre. Par exemple, les interpréteurs et les compilateurs représentent usuellement les programmes de manière interne comme des arbres ; les documents XML sont des arbres et plusieurs types de conteneurs sont basés sur des arbres, comme les arbres rouge-noir.

Nous allons maintenant examiner comment de tels arbres sont représentés et manipulés en Scala à l'aide un petit programme de calculatrice. Le but de ce programme est de manipuler des expressions arithmétiques très simples composées de sommes, de constantes entières et de variables. Deux exemples de telles expressions sont par exemple $1 + 2$ et $(x + x) + (7 + y)$.

Nous devons tout d'abord décider d'une représentation pour de telles expressions. La plus naturelle est l'arbre avec pour nœuds les opérations (ici, l'addition) et pour feuilles les valeurs (ici constantes ou variables).

En Java, un tel arbre serait représenté à l'aide d'une super-classe abstraite pour les arbres et une sous-classe concrète par nœud ou feuille. Dans un langage fonctionnel, on utiliserait un type de donnée algébrique pour la même chose. Scala donne accès au concept de *case classes* qui se trouve quelque part entre les deux. Voici comment il peut être utilisé pour définir le type des arbres de notre exemple :

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Le fait que les classes Sum, Var et Const sont déclarées comme des *case classes* signifie qu'elles diffèrent des classes standard par plusieurs aspects :

- le mot-clé **new** n'est pas obligatoire pour créer des instances de ces classes (càd on peut écrire `Const(5)` au lieu de `new Const(5)`),
- les getters sont automatiquement créés pour les paramètres du constructeur (càd qu'il est possible d'obtenir une valeur d'un paramètre de constructeur `v` d'une instance `c` d'une classe `Const` en écrivant `c.v`),
- des définitions par défaut des méthodes `equals` et `hashCode` sont fournies, fonctionnant sur la *structure* des instances et non sur leur identité,
- une définition par défaut de la méthode `toString` est fournie, qui donne la valeur sous forme de "source" (càd l'arbre de l'expression $x + 1$ retourne la chaîne de caractère `Sum(Var(x), Const(1))`),
- les instances de ces classes peuvent être décomposées à l'aide de *pattern matching* comme nous allons le voir ci-dessous.

Maintenant que nous avons défini un type de données pour représenter nos expres-

5. *Ndt* : si la traduction de *pattern matching* par filtrage de motifs est raisonnable, les *case classes* ne peuvent décemment pas être traduites par "classes de cas".

sions arithmétiques, nous pouvons commencer à définir des opérations afin de les manipuler. Nous commencerons par une fonction évaluant une expression dans un certain *environnement*. Le but de cet environnement est de donner des valeurs à des variables. Par exemple, l'expression $x + 1$ évaluée dans un environnement associant la valeur 5 à la variable x , écrite $\{x \rightarrow 5\}$, donne 6 pour résultat.

Nous avons ainsi à trouver un moyen de représenter des environnements. Nous pourrions bien entendu utiliser des structures de données associatives comme des tables de hachage mais nous pouvons également utiliser directement des fonctions ! Un environnement n'est en fait rien d'autre qu'une fonction associant une valeur à un nom (de variable). L'environnement $\{x \rightarrow 5\}$ donné ci-dessus peut ainsi être écrit en Scala simplement comme :

```
{ case "x" => 5 }
```

Cette notation définit une fonction qui, lorsque l'on lui donne la chaîne de caractère "x" en argument, retourne l'entier 5 et échoue avec une exception autrement.

Avant d'écrire la fonction d'évaluation, donnons un nom au type des environnements. Nous pourrions bien entendu toujours utiliser le type `String => Int` pour les environnement mais l'introduction d'un nom simplifie le programme et rend les changements futurs plus faciles. Ceci est accompli en Scala avec la notation suivante :

```
type Environment = String => Int
```

A partir de là, le type `Environment` peut être utilisé comme un alias d'un type de fonction d'un `String` vers un `Int`.

Nous pouvons désormais donner la définition d'une fonction d'évaluation. Conceptuellement, elle est vraiment simple : la valeur d'une somme de deux expressions est simplement la somme de la valeur de ces expressions ; la valeur d'une variable est obtenue directement depuis l'environnement et la valeur d'une constante est la constante elle-même. Exprimer cela en Scala n'est pas plus compliqué que cela :

```
def eval(t: Tree, env: Environment): Int = t match {  
  case Sum(l, r) => eval(l, env) + eval(r, env)  
  case Var(n)    => env(n)  
  case Const(v)  => v  
}
```

Cette fonction d'évaluation fonctionne en réalisant du *filtrage de motif* sur l'arbre `t`. Intuitivement, la signification de la définition ci-dessus devrait être claire :

1. elle contrôle premièrement si l'arbre `t` est un `Sum` et si tel est le cas, elle lie le sous-arbre gauche à une nouvelle variable nommée `l` et procède ensuite à l'évaluation suivant la flèche ; cette expression peut (et le fait) faire usage de variables limitées par le motif apparaissant à la gauche de la flèche, càd `l` et `r`
2. si le premier test ne réussit pas, c'est-à-dire si l'arbre n'est pas un `Sum`, l'éva-

- évaluation continue et contrôle si t est un `Var` ; si c'est le cas, le nom contenu dans le nœud est alors lié à une variable n et on procède avec la partie droite de l'expression
- 3. si le deuxième test échoue également, c'est que le t n'est ni un `Sum` ni un `Var`. Est donc contrôlé si le nœud est un `Const` et, si tel est le cas, la valeur contenue dans le nœud est liée à une variable v et l'évaluation continue avec la partie droite
- 4. finalement, si tous les tests échouent, une exception est levée afin de signaler l'échec de l'expression du filtrage de motif ; cela ne pourrait se produire ici uniquement si des sous-classes supplémentaires de `Tree` étaient déclarées.

Nous voyons que l'idée de base du filtrage de motif est d'essayer de faire correspondre un valeur à une série de motifs et, dès qu'un motif correspond, extraire et nommer différentes parties de la valeur pour finalement évaluer du code utilisant typiquement ces parties nommées.

Un programmeur aguerri dans le domaine de l'orienté-objet peut se demander pourquoi n'avons pas défini `eval` comme une *méthode* de la classe `Tree` et de ses sous-classes. Nous aurions en fait pu le faire étant donné que Scala permet la définition de *case classes* comme des classes normales. Décider ou non d'utiliser le filtrage de motif ou des méthodes est ainsi une question de goût mais de là découle également d'importantes implications sur l'extensibilité :

- lorsque on utilise des méthodes, il est facile d'ajouter de nouveaux types de nœuds car il suffit pour cela de définir de nouvelles sous-classes à `Tree` pour cela ; d'un autre côté, ajouter de nouvelles opérations afin de manipuler l'arbre est plus complexe car cela requiert de modifier toutes les sous-classes de `Tree`,
- lorsque on utilise le filtrage de motifs, la situation est inverse : ajouter un nouveau type de nœud requiert la modification de toutes les fonctions qui font du filtrage de motif sur l'arbre afin de prendre en compte ce nouveau nœud en compte ; d'un autre côté, ajouter une nouvelle opération est simple car il suffit de la définir comme une fonction indépendante.

Afin d'explorer plus avant le filtrage de motifs, définissons une nouvelle opération sur les expressions arithmétiques : la dérivation symbolique. Le lecteur se rappellera les règles suivantes concernant cette opération :

1. la dérivée d'une somme est la somme des dérivées,
2. la dérivée d'une variable v est un si v est la variable par rapport à laquelle la dérivation est effectuée et zéro autrement,
3. la dérivée d'une constante est zéro.

Ces règles peuvent être pour ainsi dire littéralement traduites dans du code Scala afin d'obtenir les définitions suivantes :

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Cette fonction introduit deux nouveaux concepts liés au filtrage de motif. Premièrement, une expression **case** possède une *garde*, une expression suivant le mot-clé **if**. Cette garde permet d'empêcher le succès du filtrage sauf si cette garde est vraie. Ici elle est utilisée afin d'être certain que nous retournons la constante 1 uniquement si le nom de la variable est le même que la variable de dérivation *v*. La deuxième nouveauté du filtrage de motif présent ici est l'utilisation d'un *joker*, écrit `_`, qui permet de faire correspondre le filtre à n'importe quel valeur sans lui donner de nom.

Nous n'avons pas encore exploré complètement la puissance du filtrage de motif mais allons brièvement nous arrêter ici afin de garder ce document court. Nous désirons toutefois montrer comment les deux fonctions ci-dessus se comportent sur un exemple réel. Pour cela, écrivons une simple fonction `main` qui réalise plusieurs opérations sur l'expression $(x + x) + (y + 7)$: elle calcule dans un premier temps sa valeur dans l'environnement $\{x \rightarrow 5, y \rightarrow 7\}$ puis calcule sa dérivée par rapport à x puis y

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"), Var("x")), Sum(Const(7), Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

En exécutant ce programme, nous obtenons le résultat attendu :

```
Expression: Sum(Sum(Var(x), Var(x)), Sum(Const(7), Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
  Sum(Sum(Const(1), Const(1)), Sum(Const(0), Const(0)))
Derivative relative to y:
  Sum(Sum(Const(0), Const(0)), Sum(Const(0), Const(1)))
```

En examinant la sortie, nous voyons que le résultat de la dérivée doit être simplifié avant d'être présenté à l'utilisateur. Définir une simplification de base des fonctions utilisant le filtrage de motif est un problème intéressant (mais étonnamment difficile), laissé en exercice au lecteur.

7 Traits

À part pouvoir hériter du code depuis une super-classe, une classe Scala peut également importer du code depuis un ou plusieurs *traits*.

Peut-être la manière la plus simple pour un programmeur Java de comprendre ce que sont les traits est de les voir comme des interfaces qui contiennent également

du code. En Scala, lorsqu'une classe hérite d'un trait, elle implémente l'interface du trait et hérite de tout le code contenu dans le trait.

Afin de voir l'utilité des traits, considérons un exemple classique : les objets ordrés. Il est souvent utile de pouvoir comparer entre eux des objets d'une certaine classe, par exemple afin de pouvoir les trier. En Java, les objets pouvant être comparés implémentent l'interface Comparable. En Scala, nous pouvons faire un peu mieux qu'en Java en définissant un équivalent à Comparable comme un trait, que nous nommerons Ord.

Lorsque l'on compare des objets, six différents prédicats peuvent être utiles : plus petit, plus petit ou égal, égal, différent de, plus grand ou égal et plus grand. Toutefois, les définir tous est fastidieux, particulièrement car quatre de ces prédicats parmi six peuvent être exprimés en utilisant les deux autres. Ainsi, en partant des prédicats égal et plus petit que (par exemple), il est possible d'exprimer les autres. En Scala, toutes ces observations peuvent être capturées dans la déclaration de trait suivante :

```
trait Ord {  
  def < (that: Any): Boolean  
  def <=(that: Any): Boolean = (this < that) || (this == that)  
  def > (that: Any): Boolean = !(this <= that)  
  def >=(that: Any): Boolean = !(this < that)  
}
```

Cette définition crée à la fois un nouveau type nommé Ord, jouant le même rôle que l'interface Comparable en Java ainsi qu'une implémentation par défaut de trois prédicats en termes d'un quatrième qui est abstrait. Les prédicats pour l'égalité et l'inégalité n'apparaissent pas ici car ils sont présents par défaut dans tous les objets.

Le type Any qui est utilisé ci-dessus est le type qui est le super-type de tous les autres types en Scala. Il peut être vu comme une version plus générale du type Objet de Java car il est également le super-type des types de basiques comme Int, Float, etc.

Pour rendre les objets d'une classes comparables, il est dès lors suffisant de définir les prédicats testant l'égalité et l'infériorité et de mélanger⁶ la classe Ord. En guise d'exemple, définissons une classe Date représentant les dates dans le calendrier grégorien. De telles dates sont composées d'un jour, d'un mois et d'une année que nous représenterons tous comme des entiers. Nous commençons ainsi la définition de la classe Date comme suit :

```
class Date(y: Int, m: Int, d: Int) extends Ord {  
  def year = y  
  def month = m  
  def day = d  
  
  override def toString(): String = year + "-" + month + "-" + day
```

6. *Ndt* : traduction de *mix in*.

La partie importante ici est la déclaration **extends** `Ord` qui suit le nom de la classe et les paramètres. Elle déclare que la classe `Date` hérite du trait `Ord`.

Ensuite, nous redéfinissons la méthode `equals`, héritée d'`Object`, afin qu'elle puisse comparer les dates en comparant leurs champs individuels. L'implémentation par défaut d'`equals` n'est pas utilisable car, comme en Java, elle compare les objets physiquement. Nous arrivons à la définition suivante :

```
override def equals(that: Any): Boolean =  
  that.isInstanceOf[Date] && {  
    val o = that.asInstanceOf[Date]  
    o.day == day && o.month == month && o.year == year  
  }
```

Cette méthode utilise les méthodes prédéfinies `isInstanceOf` et `asInstanceOf`. La première, `isInstanceOf`, correspond à l'opérateur Java `instanceof` et retourne vrai si et seulement si l'objet sur lequel elle est appliquée est une instance d'un certain type. La seconde, `asInstanceOf`, correspond à l'opérateur de *cast* en Java : si un objet est une instance du type donné, il est vu comme tel et autrement une `ClassCastException` est levée.

Finalement, la dernière méthode à définir est un prédicat qui testant l'infériorité. Ceci est réalisé en utilisant une autre méthode prédéfinie, `error`, qui lève une exception avec le message d'erreur donné en argument.

```
def <(that: Any): Boolean = {  
  if (!that.isInstanceOf[Date])  
    error("cannot compare " + that + " and a Date")  
  
  val o = that.asInstanceOf[Date]  
  (year < o.year) ||  
  (year == o.year && (month < o.month ||  
                     (month == o.month && day < o.day)))  
}
```

Ceci complète la définition de la classe `Date`. Les instances de cette classe peuvent être vues ou comme des dates ou comme des objets comparables. De plus, elles définissent toutes les six prédicats de comparaison mentionnés ci-dessus : `equals` et `<` car ils apparaissent directement dans la définition de la classe `Date` et les autres car ils sont hérités du trait `Ord`.

Les traits sont utiles dans d'autres situations que celle montrée ici, bien entendu, mais discuter leur application plus avant sort de la portée de ce document.

8 Généricité

La dernière caractéristique du Scala que nous allons explorer dans ce tutoriel est la généricité. Les programmeurs Java ont bien conscience des problèmes posés par le manque de généricité dans leur langage, un manquement qui a été résolu à partir de Java 1.5.

La généricité est la capacité d'écrire du code paramétré par des types. Par exemple, un programmeur écrivant une librairie pour les listes chaînées doit faire face au problème de décider quel type donner aux éléments de la liste. Vu qu'une liste est censée être utilisée dans un grand nombre de contextes différents, il n'est pas possible de décider que le type des éléments doit être, disons, `Int`. Cela serait complètement arbitraire et par trop restrictif.

Les programmeurs Java ont résolu cela en utilisant `Object` qui est le super-type de tous les objets. Cette solution n'est toutefois pas idéale, car elle ne fonctionne pas pour les types de base (`int`, `long`, `float`, etc.) et implique un grand nombre de conversions dynamiques de type insérées par le programmeur.

Scala rend possible de définir des classes génériques (ainsi que des méthodes) afin de résoudre ce problème. Examinons cela avec un exemple de la classe conteneur la plus simple possible : une référence qui peut être soit vide soit pointer vers un objet d'un type particulier.

```
class Reference[T] {  
  private var contents: T = _  
  
  def set(value: T) { contents = value }  
  def get: T = contents  
}
```

La classe `Reference` est paramétrée par un type, nommé `T`, qui est le type de ces éléments. Ce type est utilisé dans le corps de la classe comme le type de la variable `contents`, le type de l'argument de la méthode `set` et le type de retour de la méthode `get`.

L'exemple ci-dessus introduit les variables en Scala, ce qui ne requiert pas plus d'explications. Il est toutefois intéressant de noter que la valeur initiale donnée à cette variable est `_` qui représente une valeur par défaut. Cette valeur est `0` pour les types numériques, `false` pour les type `Boolean`, `()` pour le type `Unit` et `null` pour les types objets.

Pour utiliser cette classe `Reference` il est nécessaire de spécifier quel type utiliser pour le paramètre de type `T`, c'est-à-dire le type de l'élément contenu dans la cellule. Par exemple, pour créer et utiliser une cellule contenant un entier, on peut écrire :

```
object IntegerReference {  
  def main(args: Array[String]) {
```

```
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Comme on peut le voir sur cet exemple, il n'est pas nécessaire de convertir la valeur retournée par la méthode `get` avant de l'utiliser comme un entier. Il n'est également pas possible de stocker autre chose qu'un entier dans cette cellule particulière car elle a été déclarée comme contenant un entier.

9 Conclusion

Ce document a donné un bref survol du langage Scala et présenté quelques exemples basiques. L'utilisateur intéressé peut poursuivre son apprentissage en lisant le document compagnon *Scala By Example*, qui contient des exemples bien plus avancés et consulter le *Scala Language Specification* si nécessaire.