Projet programmation 2
Introduction à Scala

Emile Contal & Nathan Grosshans & Mathieu Hilaire
& Juraj Kolčák & Amélie Ledein & Louis Lemonnier & Stefan Schwoon

21 janvier 2022

Table des matières

1 Le langage Scala et SBT

Scala est un langage de programmation objet construit par dessus Java, tout ce qui est disponible en Java l'est aussi en Scala. Contrairement à Java il est également possible d'écrire dans un style fonctionnel, ce qui permet d'améliorer grandement la lisibilité du code.

1.1 Introduction et syntaxe

Pour commencer, lancez l'interpréteur Scala en tapant `scala` dans une console, et recopiez ligne par ligne les commandes ci-dessous en observant leur résultat.

```
1+1
```

Pour déclarer une nouvelle variable, on utilise le mot clé `var` :

```
var x = 0
x = x+1
x += 1
```

On peut déclarer des valeurs immuables avec `val` :

```
val y = 0
y = y+1
```

On définie une fonction avec `def` :

```
def plusOne(n:Int):Int = n+1
plusOne(1)
```

On peut également omettre le type de sortie, et Scala l'inférera :

```
def plusOne(n:Int) = n+1
plusOne(1)
```

Pour des fonctions qui nécessitent plusieurs lignes, on utilisera les accolades et éventuellement des points-virgules :

```
var x = 0
def count() = { x += 1; println("Counter: "+x) }
def count() = {
  x += 1
  println("Counter: "+x)
}
```

On remarque que lorsque la fonction ne prend pas d'argument, on peut omettre les parenthèses lors de son appel et de sa définition, ce qui pourrait rendre le lecteur du code perplexe :

```
var x = 0
def count = { x += 1; println("Counter: "+x) }
count
count
```

En Scala, il est possible de définir des fonctions à la volée avec `=>` :

```
(n: Int) => n+1
```

Et comme les valeurs et variables peuvent être fonctionnelles le code suivant est valide, nous reviendrons dessus en Section 4 sur la programmation fonctionnelle.

```
val plusOne = (n: Int) => n+1
```

Une syntaxe des boucles `for` est la suivante :

```
for( x <- 0 to 10 ) { println(x) }
```

▶ Écrivez une fonction calculant la factorielle d'un entier de deux manières différentes : avec une boucle et récursive.

1.2 Écrire et compiler des fichiers

Quittez l'interpréteur et écrivez un fichier `hello.scala` contenant :

```
object MyProgram {
  def main(args: Array[String]): Unit = {
    println("Hello!")
  }
}
```

Le sens du mot `object` sera défini plus tard en Section 3.1. On compile ce fichier en tapant `scalac hello.scala`, qui crée des fichiers *.class, puis on lance le programme avec `scala MyProgram`. La compilation devient terriblement longue lorsqu'il y a beaucoup de fichiers, pour se simplifier la tâche nous allons utiliser l'outil `sbt`.

1.3 SBT (Simple Build Tool)

SBT est un outil pour compiler et lancer efficacement du code Scala (ou Java). Pour l'utiliser pleinement, veuillez respecter la structure de dossiers ci-dessous :

```
(build.sbt)
src/
    main/
        scala/
            yourfiles.scala
        resources/
            (yourimages.png)
```

Vous lancerez la commande `sbt` à la racine et lorsque vous taperez `run` ou `compile`, SBT compilera vos fichiers dans un dossier "target" dont vous n'aurez pas à vous préoccuper. SBT ne recompile que les sources qui ont été modifiées, ce qui fait gagner beaucoup de temps. Vous mettrez éventuellement vos fichiers auxiliaires (ex : images) dans le dossier `resources`. Le fichier `build.sbt` contient les options de compilation ; par exemple, pour utiliser la librairie graphique Swing dont on aura besoin en Section 6, il contiendra :

```
val swing = "org.scala-lang.modules" %% "scala-swing" % "2.0.1"

lazy val root = (project in file(".")).
  settings(
    name := "My Project",
    libraryDependencies += swing
  )
```

2 Programmation Orientée Objet

L'idée de base est d'identifier les « objets » manipulés par un programme et de structurer la programmation autour de ceux-ci. Un objet peut représenter un objet naturel qui interagit avec d'autres objets ou bien une structure de données avec ses opérations.

Quelques exemples pour des objets :

— l'objet `MyProgram` (voir le premier exemple) ;
— dans une base de données, les tableaux, les requêtes ;
— dans une interface graphique, les fenêtres, les boutons, etc ;
— dans un jeu graphique, les différents acteurs.

Un objet dispose des données (des `val` et `var`) qui définissent l'état interne de l'objet et des *méthodes* (des `def`). Les méthodes permettent d'interagir avec l'objet, elles peuvent modifier les données internes, renvoyer de l'information et interagir avec d'autres objets. L'exécution d'un programme démarre avec un certain nombre d'objets, dont un qui contient la méthode `main`. Pendant l'exécution, d'autres objets peuvent être créés.

2.1 Classes

On regroupe des objets similaires dans une *classe*. Par exemple, dans une interface graphique, les fenêtres formeraient une classe. On écrit donc du code en décrivant le comportement des classes. Pendant l'exécution d'un programme, une classe `C` peut être instanciée avec le mot-clé `new` ; ceci crée un objet de la classe `C`.

Comme exemple, on regarde un vélo : on considère qu'il dispose d'un compteur kilométrique et qu'il permet de bouger et freiner.

```
class Bicycle {
  var counter:Double = 0
  def move {
    counter += 1
    println("mon compteur est à"+counter)
  }
  def brake { println("j'arrête") }
}
```

Dans `main`, on place le code suivant qui sert à pédaler 10 km.

```
val b = new Bicycle
while (b.counter < 10) b.move
b.brake
```

La première ligne crée une instance de `Bicycle` ce qui va exécuter le code de la classe en dehors des `def`. À remarquer que chaque instance possède son propre compteur. Du coup, on peut créer deux vélos en même temps, chacun faisant son voyage :

```
val b = new Bicycle
val c = new Bicycle
while (b.counter < 10) b.move
while (c.counter < 5) c.move
```

Les objets peuvent prendre des paramètres lors de leur création :

```
class Bicycle (name:String) { ... }
val b = new Bicycle("Moulinette")
```

▶ Faites afficher le nom du vélo dans `move`.

2.2 Héritage

Un aspect intéressant de la programmation orientée objet est le partage de code. Si certains objets d'une classe `C` ont des comportements différents ou supplémentaires par rapport aux autres membres, il convient de les regrouper dans une sous-classe. Les classes forment donc une hiérachie.

Dans une sous-classe, on ne décrit que les différences à la super-classe. Par exemple, considérons un vélo de route (qui est plus rapide que les autres) ou un vélo rouillé (qui fait du bruit lors du freinage) :

```
class RoadBicycle(name:String) extends Bicycle(name) {
  override def move {
    counter += 1.5
    println(name+": *roule*")
  }
}

class RustyBicycle(name:String) extends Bicycle(name) {
  override def brake { println("eek") }
}
```

Le mot-clé `override` spécifie que la méthode remplace celle de la super-classe. On peut toutefois réutiliser une fonction remplacée, p.ex. `move` dans `RoadBicycle` est équivalent à :

```
override def move { counter += 0.5 ; super.move }
```

Une méthode qui accepte comme argument un objet d'une classe `C` peut travailler sur n'importe quelle sous-classe de `C`, tout en utilisant les méthodes remplacées de la sous-classe.

▶ Regroupez les lignes concernant `move` et `brake` dans `main` dans une méthode `travel` qui prend comme paramètre un `Bicycle` et la distance à parcourir. Utilisez-la avec deux vélos différents.

Toute classe possède automatiquement une méthode `toString`, mais à priori celle-ci n'est pas d'une grande utilité :

```
val b = new Bicycle("Moulinette")
println(b)
```

Pour debugger ses programmes, il peut être très utile d'y inclure plus d'information.

▶ Redéfinissez `toString` pour afficher le nom et le compteur d'une bicyclette. Servez-vous des «chaînes interpolées», p.ex. `s"Bonjour $nom"` donne `Bonjour Jean` si `nom.toString` donne `Jean`.

2.3 Classes Abstraites

Les classes *abstraites* ne peuvent pas être instanciées, mais possèdent un intérêt pour le partage du code. Par exemple, tout véhicule peut être classé comme vélo, train ou bien d'autres, il est donc inutile d'instancier un 'véhicule'. Néanmoins, les différents types de véhicules ont des choses en commun : p.ex. un voyage se fait en bougeant jusqu'à ce qu'on ait parcouru une certaine distance. Considérons donc la déclaration suivante :

```
abstract class Vehicle {
  var counter:Double = 0
  def move : Unit
  def brake { println("j'arrête") }
}
```

Puisque la classe est abstraite, il est interdit d'en instancier des objets. En plus, toute sous-classe de `Vehicle` doit spécifier le comportement concret de `move` dont on ne connaît que le type.

▶ Faites de `Bicycle` une sous-classe de `Vehicle` et ajoutez une autre sous-classe `Scooter`. Modifiez `travel` pour accepter tout véhicule.

2.4 Traits et Héritage Multiple

Un `trait` est similaire à une classe abstraite, mais avec une restriction : il ne peut avoir aucun *constructeur*, c'est-à-dire du code qui serait exécuté lorsqu'on crée une instance. Il n'y a donc pas de code hors méthodes, ni initialisation de variables, ni même paramètres. Exemple de `Runnable`, trait qu'on va rencontrer dans la Section 7 :

```
trait Runnable {
  def run(): Unit
}
```

Une classe qui hérite de `Runnable` contient donc une méthode `run` (dont le comportement reste à concrétiser). Avantage des traits : une même classe peut hériter d'une seule super-classe, mais de multiples traits. Par exemple, les déclarations suivantes sont possibles si `B` est une classe (peut-être abstraite) et `C`, `D` sont des traits :

```
class A extends B { ... }
class A extends C { ... }
class A extends B with C { ... }
class A extends B with C with D { ... }
```

3 Objects, Case Class et Pattern Matching

3.1 Objets singletons

Lorsqu'on souhaite définir une classe qui n'aura qu'une seule instance on utilise `object`. On remarque que ce mot clé a déjà été utilisé auparavant pour définir l'objet `MyProgram`. Pour ceux qui auraient déjà des notions de programmation orientée objet, les classes « statiques » n'existent pas en Scala et on se sert des `object` pour remplir cette fonction.

```
object A {
  var n=0
  def incr () { n = n+1 }
  def print () { println(n) }
}
A.print()
A.incr()
A.print()
```

Une autre façon très pratique de créer un objet d'une classe unique est de créer une sous-classe à la volée au moment de la création de la variable :

```
class A {
  var n=0
  def incr () { n=n+1 }
  def print () { println(n) }
}
val b = new A {
  override def print () { println("value: "+n) }
}
b.incr()
b.print()
```

3.2 Case class

Les *case class* sont des classes avec des propriétés particulières. Il convient de penser de leurs instances comme des n-uplets immuables. Regardons l'exemple suivant :

```
case class Complex (re:Double, im:Double) {
  def size = scala.math.sqrt(re*re + im*im)
  def + (c:Complex) = Complex(re+c.re,im+c.im)
}

val c = Complex(1,2)
val d = c + Complex(-2,3)
println(d)
val e = Complex(1,2)
println(c == e)
```

Un nombre complexe est défini entièrement par ses parties réélle et imaginaire. Ces données sont connues lors de la création d'une instance de `Complex`. Les opérations sur les nombres complexes ne changent pas leur état interne (contrairement à l'exemple de la bicyclette qui possède son compteur). Du coup, les case class sont équipées des opérations suivantes :

— On peut fabriquer leurs instances sans `new`.
— L'opérateur `==` compare les paramètres de deux instances et non leur identité référentielle.
— Une méthode `toString` est défini automatiquement (voir le résultat de `println`).

Du coup, les paramètres d'un case class sont des `val` immuables, et la pratique de déclarer des `var` à l'intérieur est découragée (ceux-ci ne seraient pas pris en compte par `==`).

▶ Comparer avec le comportement de `==` et `println` quand `Complex` est une classe ordinaire.

▶ Programmer une class abstraite `Tree` avec une méthode `sum`. Déclarer des sous-classes `Node` et `Leaf` de façon que p.ex. `Node(Node(Leaf(2),Leaf(3)),Leaf(5)).sum` donne 10.

3.3 Pattern matching

Tout comme en OCaml, il est possible d'écrire des pattern matching. Le matching est particulièrement aisé pour les case class. Avec la classe `Tree` de Section 3.2, ceci permet d'écrire :

```
val t = Node(Node(Leaf(1),Leaf(2)),Leaf(3))

def sum(t:Tree):Int = { t match {
    case Leaf(v) => v
    case Node(l,r) => sum(l)+sum(r)
  }
}

sum(t)
```

Lorsque le `match` est le seul élément de la fonction, on peut écrire :

```
def sum:Tree=>Int = {
  case Leaf(v) => v
  case Node(l,r) => sum(l)+sum(r)
}
```

Dans chaque `case` d'un pattern matching, on peut ajouter des tests sur les valeurs avec les syntaxes :

```
def f:Tree=>Int = {
  case Leaf(0) => 0
  case Leaf(v) if v < 0 => -1
  case Leaf(v) if v > 0 => 1
  case Node(l,r) => f(l)+f(r)
}
```

Et tester l'égalité d'un objet avec des backquotes :

```
class A {}
val x = new A
val y = new A
def test:A=>Unit = {
  case `x` => println("This is x!")
  case `y` => println("This is y!")
  case _ => println("unknown")
}
```

▶ Définissez des classes représentant une expression arithmétique sur des entiers relatifs avec les opérateurs “Add”, “Mul” et “Abs” (valeur absolue). Définissez une fonction qui évalue l'expression.

▶ Ajoutez le cas “X” parmi les expressions arithmétiques puis définissez une fonction qui évalue la dérivée d'une expression par rapport à “X”.

4 Collections et Programmation Fonctionnelle

Scala offre de nombreuses classes pour stocker des collections de données. On discutera ces classes et leurs opérations.

4.1 Tableaux

Commençons avec le type le mieux connu, les `Array`. Quelques possibilités pour déclarer un tableau :

```
val a = Array(1,3,5)
val b = Array[String]("good","bad","ugly")
val c = new Array[Int](3)
```

Les deux premières lignes créent un `Array` avec du contenu. Dans le premier cas, le type est déterminé implicitement par Scala (`Int`). Dans le deuxième cas, on le spécifie explicitement (`String`). Dans le troisième cas, on ne déclare que le type et la taille, le contenu sera la valeur par défaut du type, dans ce cas 0. Attention, sans le `new`, c'est une déclaration d'un `Array` de `Int` avec un seul élément, le 3. Deux choses à remarquer.

- Premièrement, `Array` (et les autres collections) sont un exemple d'une classe *polymorphe*, paramétrée par un type (tel que `Int`, `String`, ou en fait une classe quelconque).
- Deuxièmement, dans les exemples ci-dessus, `a,b,c` sont des *références* à des objets du type `Array[Int]` etc. Le fait de déclarer `a` comme `val` veut dire que `a` designera toujours le même objet pendant sa vie, même si le contenu de cet objet peut muter. Du coup, une déclaration telle que

  ```
  var f=a
  ```

  crée simplement une deuxième référence vers l'objet pointé par `a`. À remarquer que `f` est un `var`. Du coup,

  ```
  f(1)=5; f=c; f(2)=5
  ```

  modifie d'abord `a(1)`, puis `c(2)`.

Les tableaux multidimensionnels sont moyennement supportés par Scala. On peut faire

```
var g=Array.ofDim[Int](5,3)
```

pour créer un `Array` de taille 5 dont les éléments sont des `Array` de `Int` de taille 3. Accès aux éléments : p.ex. `g(4)(2)`.

4.2 Divers types de listes

Scala offre tout un zoo de classes représentantes de différentes réalisations pour stocker des collections de données. Le contenu de ces classes peut être mutable ou immuable ; ces classes se

trouvent dans la hiérarchie `scala.collection.mutable._` ou `scala.collection.immutable._`, respectivement.

Premier exemple, `List` donne une liste `immutable`, c'est-à-dire que suite à la déclaration

```
val a = List(1,2,3,4)
```

`a(2)` donne 3, mais `a(2)=5` échoue. L'expression `3::a` donne une liste avec 3 suivi par les éléments de `a`, `a:+3` rajoute 3 à la fin, et `a++b` concatène deux listes. Attention, toutes ces opérations créent de nouveaux objets de type `List`, du coup elles auront un coût de $O(n)$, pour n éléments.

`List` est réalisée par des listes chaînées. Du coup, `a(i)` n'est pas une opération en temps constant. Pour des accès aléatoires, `Vector` (parmi d'autres) est plus efficace.

Vous trouverez une liste de différentes collections en Scala sur les pages suivantes :
`https://twitter.github.io/scala_school/collections.html`
`https://twitter.github.io/scala_school/coll2.html`.
Les différentes collections se distinguent donc par les opérations possibles sur les objets et leur efficacité. La page suivante en donne un aperçu :

`http://docs.scala-lang.org/overviews/collections/performance-characteristics.html`

4.3 Tableaux associatifs

Scala supporte aussi des tableaux associatifs qui stockent des paires (*clé*,*valeur*) ; on peut alors retrouver la valeur à partir de la clé. La classe `Map` est alors paramétrée par les types des clés et des valeurs. Il en existe des versions mutables et immuables.

Exemple : Créer un `Map` (par défaut non-mutable) des `String` aux `Int` avec deux paires :

```
val m = Map("a" -> 3, "c" -> 5)
```

Par la suite `m("a")` donne 3.

Créer un tableau vide initialement mais mutable, en spécifiant les types paramétrés explicitement :

```
val n = scala.collection.mutable.Map[String,Int]()
```

Rajouter (ou mettre à jour) des valeurs : p.ex. `n("a")=5`.

4.4 Opérations

On est souvent amené à traverser les collections de données. Une possibilité sont les boucles `for`. Quelques exemples :

```
val a = Array(5,2,8,1)
var i = 0;
for (i <- 0 to 3) println(a(i))
for (i <- 0 until a.length) println(a(i))
for (i <- a.indices) println(a(i))
for (i <- a) println(i)
```

En plus, il existe plusieurs méthodes qui acceptent des fonctions comme paramètre. P.ex., `foreach` accepte une fonction renvoyant un type `Unit` (qui donc ne renvoie rien) et l'applique à tous les éléments.

```
a.foreach(x => println(x))
a.foreach(println(_))
```

À noter que la seconde forme n'est possible que pour les fonctions anonymes ayant un paramètre qui n'apparaisse qu'une seule fois.

D'autres opérations permettent de générer de nouvelles listes :

```
a.map(2 * _)
a.filter(_ >= 5)
```

Faire des calculs sur les éléments :

```
a.exists(_ >= 7)
a.find(_ >= 7)
a.count(_ <= 7)
a.foldLeft(0)((a,b) => a+b)
```

Dans la deuxième ligne, le type de retour de `find` est un type `Option[T]`, une classe abstraite comme vue en Section 3.3, qui peut soit être `None` (un `case object`) soit `Some(x:T)` où `T` est le type des éléments de `a`. Dans la dernière ligne, 0 est une valeur initiale ; la fonction est d'abord appliquée sur la valeur initiale et le premier élement, puis sur le résultat et le deuxième élément et ainsi de suite. Avec `reduceLeft`, c'est le même principe, mais en omettant la valeur initiale. Du coup, `a.reduceLeft((a,b)=>(a+b))` donne le même résultat dans ce cas. On remarque que `a.reduceLeft(_+_)` fonctionne aussi en plus court, et qu'il existe simplement `a.sum` (ainsi que `a.min`, `a.max`).

▶ Faites la concaténation des éléments d'un `Array[String]`. Trouvez aussi la longueur de la chaîne la plus courte.

Dernière remarque, un tableau multidimensionnel peut être ramené à un tableau simple par `flatten`.

4.5 For Comprehensions

Scala offre du sucre syntactique pour une notation compacte des boucles imbriqués, les *for comprehensions*. Considérons la classe suivante qui représente une cellule avec coordonnées x, y qui contient une valeur v.

```
class Cell (val x:Int,val y:Int) {
  var v:Int = 0
  def set(n:Int) = { v = n }
}
```

La boucle suivant initialise une grille $8x8$ de cellules :

```
var g = Array.ofDim[Cell](8,8)
for { x <- 0 to 7; y <- 0 to 7 } g(x)(y) = new Cell(x,y)
```

Une *for comprehension* peut aussi filtrer des valeurs :

```
for { x <- 0 to 7; y <- 0 to 7; if x-y==3 } g(x)(y).set(1)
```

Avec le mot-clé `yield`, une *for comprehension* peut construire une liste :

```
val cells = for { x <- 0 to 7; y <- 0 to 7 } yield g(x)(y)
```

L'opération précédente peut maintenant être exprimée par l'une des lignes suivantes :

```
for { c <- cells } if (c.x-c.y==3) c.set(1)
for { c <- cells; if c.x-c.y==3 } c.set(1)
cells.filter(c => c.x-c.y==3).foreach(_.set(1))
```

▶ Mettez v à $x + y$ dans toutes les cellules de `g`. Calculez la somme de toutes les valeurs v dans `g`.

5 Programmation polymorphe

Dans la Section 4 on a vu des *classes polymorphes* telles que `Array[String]` ou `List[Int]`. Dans ces cas, `Array` et `List` sont des classes qui fournissent une fonctionnalité commune pour des objets de différentes types. Dans cette section, on étudie quelques exemples où ces classes paramétrées pourront être utiles, et le fonctionnement de base.

5.1 Polymorphie

Disons qu'on a envie de déclarer une classe de graphes. Dans nos graphes chaque sommet possède un nom, et il y a un sommet appellé racine :

```
class Node (val name:String)
class Graph (root:Node) { def getRoot = root }
val n = new Node("racine")
val g = new Graph(n)
println(g.getRoot.name)
```

Jusqu'ici, tout marche bien. Maintenant, supposons qu'on a envie d'utiliser cette classe dans de multiples contextes : p.ex. dans un cas, les sommets ne portent que des noms, dans d'autres ils sont équipés d'un poids, dans encore un autre cas il s'agit d'un arbre généalogique où les sommets portent des informations sur des personnes (nom, prénom, date de naissance, ...). À défaut d'ajouter toutes les informations réquises dans tous ces contextes à la seule classe `Node` (ce qui rendrait cette classe lourde et peu lisible), on préfère la création des sous-classes. À priori, aucun souci :

```
class ExtNode (name:String, val weight:Int) extends Node(name)
val n = new ExtNode("racine",5)
val g = new Graph(n)
println(g.getRoot.name)
```

`Graph` accepte bien `n` car `ExtNode` est une sous-classe de `Node`. Mais ne serait-il pas intéressant d'afficher le poids de la racine ? Ajoutons la ligne suivante :

```
println(g.getRoot.weight)
```

Or, cela ne passe pas. En fait, le type inféré pour la méthode `getRoot` est logiquement `Node` car `root` possède ce type. Or, pour obtenir le poids de `g.getRoot` il faut que Scala sache qu'il s'agit d'un `ExtNode`. C'est à ce moment où l'utilité des classes polymorphes s'avère. On change la déclaration ainsi :

```
class Graph[N] (root:N) { def getRoot = root }
```

Ici, `N` est un type qui paramètre la classe `Graph` qui sera spécifié lorsqu'on instancie un graphe. D'ailleurs, `getRoot` renvoie alors le type `N`. Le code suivant marche alors sans problème car le type de `g` devient `Graph[ExtNode]` et `g.getRoot` devient `ExtNode`.

```
val n = new ExtNode("racine",5)
val g = new Graph(n)
println(g.getRoot.weight)
```

En effet, dans cet exemple, Scale devine que `N` vaut `ExtNode` dans `new Graph(n)` car `n` possède ce type. Si on souhaite expliciter ce choix, on peut déclarer `new Graph[ExtNode](n)`.

Parfois, il sera aussi intéressant de créer des méthodes paramétrées :

```
def firstLast[T] (a:List[T]) = List(a.head,a.last)
firstLast(List(1,2,3))
firstLast(List("good","bad","ugly"))
```

5.2 Types bornés

Mettons que la classe `Graph` devrait fournir des fonctionnalités plus intéressantes, p.ex. afficher le nom d'un sommet.

```
class Graph[N] (root:N)
        { def getRoot = root
          def printRootName = { println(root.name) } }
```

Or, ceci ne fonctionne plus car `root` est désormais du type `N`, et Scala ne connaît rien sur ce type ; en particulier Scala est incapable d'inférer que ce type possède un `name`. La solution consiste à spécifier que `N` doit être un sous-classe de `Node`.

```
class Graph[N <: Node] (root:N)
```

Inversement, `>:` spécifie une contrainte de super-classe. Regardons le code suivant (il convient d'utiliser l'interpréteur `scala`) :

```
class A
class B extends A { def mergeWith[T](x:T) = List(x,this) }
val a = new A ; val b = new B
List(a,b)
b.mergeWith(a)
```

On constate que le type inféré de `List(a,b)` est `List[A]` mais que `b.mergeWith(a)` donne `List[Any]`, bien que ces deux listes sont composées des mêmes éléments : lorsque Scala compile la méthode `mergeWith`, la super-classe la plus profonde entre `T` et `B` n'est que `Any`. La modification suivante permet de sauver la mise :

```
class B extends A { def mergeWith[T >: B](x:T) = List(x,this) }
```

On exige alors que `T` soit une superclasse de `B` ce qui permet à Scala d'inférer le type `List[T]` pour `mergeWith`. D'ailleurs, cette contrainte n'empêche pas de passer des sous-classes de `B`, le résultat de `mergeWith` sera toutefois du type `List[B]`.

```
class C extends B
val c = new C
b.mergeWith(c)
c.mergeWith(c)
```

5.3 Bornes multiples

Revenons sur l'exemple d'un graphe. Désormais on le spécifie avec une liste de sommets et arêtes :

```
class Node (val name:String)
class Edge (val from:Node, val to:Node)
class ExtNode (name:String, val weight:Int) extends Node(name)
class Graph[N <: Node] (nodes:List[N], edges:List[Edge])
  { def outgoing (n:N) = edges.filter(_.from == n)
    def successors (n:N) = outgoing(n).map(_.to) }
val n1 = new ExtNode("n1",5)
val n2 = new ExtNode("n2",3)
val edge = new Edge(n1,n2)
val g = new Graph(List(n1,n2),List(edge))
g.successors(n1).foreach(n => println(n.name))
```

Cet exemple fonctionne, mais échoue si on remplace `name` par `weight` dans la dernière ligne ; logiquement car `successors` travaille avec une liste de `Edge` qui ne relie que des `Node`. Il convient alors de paramétrer la classe `Edge` elle aussi :

```
class Edge[N] (val from:N, val to:N)
class Graph[N <: Node] (nodes:List[N], edges:List[Edge[N]])
...
g.successors(n1).foreach(n => println(n.weight))
```

Supposons maintenant qu'on veut utiliser la classe `Graph` dans un contexte où les arêtes portent des étiquettes, et qu'on a envie de récupérer les étiquettes sur les arêtes sortantes de `n1`. On se définit alors une sous-classe d'arêtes :

```
class ExtEdge[N] (from:N, val label:String, to:N) extends Edge(from,to)
val n1 = new ExtNode("n1",5)
val n2 = new ExtNode("n2",3)
val edge = new ExtEdge(n1,"a",n2)
val g = new Graph(List(n1,n2),List(edge))
g.outgoing(n1).foreach(e => println(e.label))
```

Mais bien sûr cela échoue car `outgoing` renvoie toujours `List[Edge]`. La solution consiste alors à donner deux paramètres interdépendants à la classe `Graph` :

```
class Graph[N <: Node, E <: Edge[N]] (nodes:List[N], edges:List[E])
```

5.4 Covariance et contravariance

Dans l'exemple précédent, `Edge[A]` et `Edge[B]` sont deux classes distinctes et incomparables. Même si `B` était une sous-classe de `A`, cela ne fait pas de `Edge[B]` une sous-classe de `Edge[A]`, ou inversement. Parfois, un tel comportement est néanmoins désirable ; regardons l'exemple suivant qui réalise une matrice d'un type `T` quelconque (`f` étant une fonction qui fournit le contenu de la matrice) :

```
class Matrix[T] (height:Int, width:Int, f:(Int,Int)=>T)
  { val data = IndexedSeq.tabulate[T](width,height) { (i,j)=> f(i,j) } }
```

Disons qu'on utilise une matrice de `A` où `A` possède une donnée `n` entier, et qu'on souhaite calculer les sommes des `n` dans toutes les lignes avec `rowsums` :

```
class A (val n:Int)
def rowsums (m:Matrix[A]) = m.data.map(_.map(_.n).sum)
val m1 = new Matrix(3,3,(i,j) => new A(i*3+j))
println(rowsums(m1).mkString(","))
```

Supposons maintenant qu'on possède une sous-classe `B` de `A`. Puisque toute instance de `B` possède elle aussi un `n`, on veut naturellement utiliser `rowsums` avec une matrice de `B`.

```
class B (n:Int) extends A(n)
val m2 = new Matrix(3,3,(i,j) => new B(i*3+j))
println(rowsums(m2).mkString(","))
```

Or, cela échoue car `m2` est du type `Matrix[B]` considéré incompatible avec le `Matrix[A]` attendu par `rowsums`. Si on modifie la définition de la matrice ainsi :

```
class Matrix[+T] (...) { ... }
```

alors `Matrix[B]` sera considéré comme une sous-classe de `Matrix[A]` (car `B` est sous-classe de `A`), et `rowsums` accepte bien `m2`. Dans ce cas, on dit que `Matrix` est *covariant*.

Dans certains cas rares, on a besoin de l'effet invers (*contravariance*) : un déclaration de la forme `class Matrix[-T] (...) { ... }` ferait de `Matrix[A]` une sous-classe de `Matrix[B]` si `B` est sous-classe de `A`.

6 Interfaces Graphiques

Dans cette section nous allons voir comment créer des interfaces graphiques simples avec la librairie `Swing`.

6.1 Une première fenêtre et ses composants

Pour créer une fenêtre nous pouvons utiliser la classe `SimpleSwingApplication`, et lui attribuer un objet `MainFrame` (ici une sous-classe créée à la volée) :

```
import swing._

object MyApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "MyApplication"
    contents = new Label("Hello!")
  }
}
```

La classe `SimpleSwingApplication` inclut une fonction `main`, et la fenêtre s'ouvre directement lorsqu'on exécute ce programme. Comme son nom l'indique, le contenu de la fenêtre est à placer dans le champs `contents`, ici simplement un `Label` contenant du texte. De manière plus intéressante, il convient de mettre dans `contents` un conteneur de plusieurs composants à positionner dans l'espace, par exemple un `BorderPanel` :

```
import swing._
```

```
object MyApp extends SimpleSwingApplication {
  def top = new MainFrame {
    title = "MyApplication"
    contents = new BorderPanel {
      add(new Label("En haut"), BorderPanel.Position.North)
      add(new Label("Au milieu"), BorderPanel.Position.Center)
      add(new Label("En bas"), BorderPanel.Position.South)
    }
  }
}
```

On utilise la méthode `add` d'un tel conteneur, qui prend en argument un `Component` et des contraintes de position. Ici `Component` est une classe abstraite qui pourrait être en particulier :

- un `Label` pour afficher du texte ou une image ;
- un `Button` pour interagir avec l'utilisateur ;
- un autre conteneur comme `BorderPanel`, `GridPanel` ou tout autre sous-classe de `Panel` ;
- un `TextField` ou `TextArea` comme formulaire de texte ;
- etc.

▶ Essayez ces différents composants à l'aide de la documentation en ligne (accessible à cette adresse : `http://scala-lang.org/documentation/api.html`) pour voir ce qu'ils permettent.

6.2 Réagir aux actions de l'utilisateur

Pour réagir aux actions de l'utilisateur nous allons voir deux possibilités : l'une où l'objet `MainFrame` se charge de traiter les évènements, l'autre où les boutons définissent eux-mêmes quoi faire. Il existe d'autres méthodes que nous n'abordons pas ici, comme les `Publishers` et `Reactors` qui peuvent être utiles pour gérer les clics droits.

6.2.1 Avec `listenTo` et `reactions`

La première méthode consiste à utiliser la méthode `listenTo` de `MainFrame` qui prend en argument un ou plusieurs composants, puis traiter les différents évènements en ajoutant un pattern matching dans son champ `reactions` :

```
import swing._
import swing.event._

object MyApp extends SimpleSwingApplication {
  val myButton = new Button("Click here")

  def top = new MainFrame {
    title = "MyApplication"
    contents = myButton
    listenTo(myButton)
    reactions += {
      case ButtonClicked(source) => println("Thanks")
      case _ => {}
    }
  }
}
```

Ici le pattern matching concerne des `ComponentEvent` tels que `ButtonClicked` ou `EditDone`, qui prennent en argument le composant source de l'action.

▶ Créez à l'aide de ces outils une mini calculatrice qui affiche le produit de nombres entrés dans des `TextField` lorsqu'on appuie sur un bouton.

▶ Créez un convertisseur Celsius/Fahrenheit avec deux champs textes mais sans bouton, qui s'actualise dès que l'utilisateur change l'une des deux valeurs.

6.2.2 Avec `Action`

La deuxième méthode est d'écrire le code à exécuter directement dans le composant concerné, à l'aide d'une `Action` (que l'on peut créer simplement grâce à son objet compagnon) :

```
import swing._
import swing.event._

object MyApp extends SimpleSwingApplication {
  val myButton = new Button {
    action = Action("Click here") {
      println("Thanks")
    }
  }

  def top = new MainFrame {
    title = "MyApplication"
    contents = myButton
  }
}
```

▶ Créez une grille de 20 par 10 boutons, qui affichent leur coordonnées lorsque l'on clique dessus.

6.3 Personnalisations graphiques

Vous pouvez bien sûr personnaliser l'aspect des composants swing.

▶ Cherchez dans la documentation en ligne pour personnaliser la grille précédente de telle sorte que les boutons soient de taille 30px par 30px avec une bordure bleue de 1px d'épaisseur.

Afin d'utiliser une image comme icône d'un label ou d'un bouton, on utilisera un `ImageIcon` que l'on importe avec `import javax.swing.ImageIcon` :

```
icon = new ImageIcon(getClass.getResource("myimage.png"))
```

▶ Modifiez la grille précédente pour que les boutons affichent une image lorsqu'on clique dessus.

7 Threads

Scala fournit une interface simplifiée pour travailler avec des *threads*. Un thread est une tâche qui s'exécute en parallèle du reste du programme.

7.1 Un premier exemple

En Scala, un thread est associé avec un objet de la classe `Thread` (ou d'une sous-classe de `Thread`). Cette classe comporte, parmi d'autres, deux méthodes :

— `run`, une méthode abstraite, donc à concrétiser avant d'instancier un tel objet – le programmeur y décrit le comportement parallèle souhaité ;

— `start`, méthode qui va créer un thread au sein du système qui fera appel à `run`.

En voici un exemple :

```
object main {
  val french = new Thread {
    override def run { for (_ <- 1 to 10) println("bonjour") }
  }
  val anglais = new Thread {
    override def run { for (_ <- 1 to 10) println("hello") }
  }

  def main (argv : Array[String]) {
    french.start; anglais.start
  }
}
```

▶ Exécutez l'exemple plusieurs fois pour observer différents entrelacements entre les deux threads.

7.2 Vie d'un thread et vie de son processus

Il convient de distinguer la vie d'un objet `Thread` et la vie du thread système. Le fait d'instancier un objet `Thread` crée tout simplement cet objet, mais pas encore une tâche parallèle. En plus, inutile d'appeler la méthode `run` directement – c'est simplement une méthode normale qui n'a rien de spécial. Il faut donc utiliser `start` pour démarrer la vie d'un thread système.

▶ Dans l'exemple, remplacez `start` par `run`. Qu'observe-t-on ?

Remarque technique : rappelez-vous qu'en C, l'appel système `exit` tue le processus avec tous ses threads. Il en est de même en Scala (utiliser `System.exit`). Il y a pourtant deux différences :

— `System.exit` fonctionne correctement dans les programmes Scala compilés et exécutés dans la ligne de commande. Toutefois, `sbt` rattrape cet appel.

— En C, le fait de terminer la fonction `main` entraîne un appel implicite d'`exit`. Ceci n'est pas le cas pour Scala – le compilateur assure que le processus reste en vie tant qu'il existe au moins un thread et fait `exit` seulement après.

7.3 Commment instancier un thread

Il y a plusieurs façons de générer un objet `Thread` :

— comme dans l'exemple, instancier directement un objet `Thread` avec un `override` de `run` ;

— créer une sous-classe de `Thread` avec un tel `override` que l'on peut instancier plusieurs fois par la suite ;

— créer un objet dérivé de `Runnable`, et instancier un `Thread` avec cet objet comme argument.

`Runnable` fournit simplement une méthode (abstraite) `run`. Schéma d'usage :

```
class X extends Runnable {
  override def run { ... }
}
val x = new X
val t = new Thread(x)
```

Puisque `Runnable` est un *trait*, on peut aussi faire :

```
class X extends Y with Runnable {
  override def run { ... }
}

def main (argv : Array[String]) {
  new Thread(new X).start ; ...
}
```

▶ Créez une extension `Bavardeur` de `Runnable` qui émet 10 fois une même ligne dont le contenu est un paramètre. Refaites l'exemple précédent en utilisant cette classe.

7.4 Interaction entre threads

L'objet compagnon `Thread` fournit la méthode `sleep` qui fait s'endormir le thread pendant une période donnée. Attention, l'argument est en millisecondes, du coup `Thread.sleep(2000)` correspond à une période d'endormissement de deux secondes.

La méthode `join` d'un objet `Thread` attend la fin de l'exécution du thread en question.

▶ Modifiez l'exemple de telle sorte que `main` émette « `fin` » *après* la terminaison des deux threads.

Remarque technique : contrairement à l'appel système sous-jacent, `join` ne renvoie aucune valeur.

La méthode `interrupt` permet de terminer un thread. Techniquement, cela déclenche une exception dans le thread concerné. Cette exception peut pourtant être rattrapée avec une construction `try` / `catch`. En effet, le non-rattrapage d'une exception termine le thread, mais avec un message bizarre.

Schéma d'usage pour le rattrapage :

```
override def run {
  try {
    ...
  } catch {
      case e:InterruptedException => ...
  }
}
```

Si une interruption intervient lorsque le thread exécute le bloc `try`, l'exécution de ce bloc sera terminée et le contrôle transféré vers la partie `catch`. La partie après la flèche (`=>`) peut être vide.

▶ Modifiez la classe `Bavardeur` de telle sorte qu'elle attende une seconde entre deux lignes. Faites en sorte que la méthode `main` interrompe l'un des threads après 3 secondes ; le thread devrait réagir en disant « `ouf` » et en terminant.

7.5 Synchronisation entre threads

Une difficulté bien connue dans la programmation concurrente est de coordonner l'accès aux données partagées, notamment d'en gérer les modifications. Attention, même une instruction simple comme `i+=1` est composée de deux étapes (lecture/écriture dans la mémoire). On considère l'exemple suivant :

```
object main {
  var i = 0

  object toto extends Runnable {
    override def run { for (_ <- 1 to 100000) { i+=1 } }
  }

  val t1 = new Thread(toto)
  val t2 = new Thread(toto)

  def main (argv : Array[String]) {
    t1.start; t2.start
    t1.join; t2.join
    println("i = "+i)
  }
}
```

▶ Exécutez le programme ci-dessus. Vous observerez que la valeur atteinte par `i` est bien loin de la valeur attendue (200000).

Scala permet une façon simplifiée d'utiliser des sémaphores pour gérer les accès aux données partagées. Tout objet est implicitement associé à un sémaphore. On considère le code suivant où `o` est un objet quelconque :

```
o.synchronized { ... }
```

Ce code obtient d'abord le sémaphore associé à `o`, puis exécute le code entre accolades et ensuite relâche le sémaphore. L'obtention du sémaphore n'est possible que pour un seul thread à la fois ; si jamais deux threads essayent de l'obtenir, l'un des deux doit attendre jusqu'à ce que le premier relâche le sémaphore.

▶ Modifiez le programme ci-dessus pour que `i` atteigne toujours la valeur 200000.

7.6 En savoir plus

Scala School contient un tutoriel sur la programmation concurrente (un peu abrégé), qui aborde quelques sujets supplémentaires tels que des structures de données (p.ex. collections) conçues pour être utilisées avec des threads.

`https://twitter.github.io/scala_school/concurrency.html`

8 Documentation

Lorsque l'on collabore à plusieurs sur un projet (ou pour tout projet de grande taille, même effectué seul), il est essentiel que chaque participant *documente* les divers éléments de code qu'il

produit (classes, méthodes, etc.) afin que les autres (ou même lui-même) puissent les utiliser comme des « boîtes noires », sans devoir directement lire le code en question pour savoir comment ils fonctionnent.

Scaladoc (fourni avec Scala) est un système générant de la documentation (sous forme de pages HTML) à partir de commentaires spécifiques que le programmeur ajoute aux fichiers sources. Devant chaque élément (classe, méthode, attribut, etc.) que l'on souhaite documenter, on place un bloc de commentaire débutant par `/**`, terminant par `*/` et contenant généralement un texte descriptif court, un texte descriptif plus long (qui peut néanmoins être omis) et d'autres informations introduites par des étiquettes, comme par exemple `@param` pour ajouter un descriptif lié à un paramètre d'un constructeur ou d'une méthode.

Voici un court exemple :

```
/**
 * Descriptif court de la classe ‘C‘.
 *
 * Descriptif long de la classe ‘C‘.
 *
 * Celui-ci peut s'étaler sur plusieurs lignes et paragraphes.
 *
 * @author Informations concernant l'auteur.
 *
 * @constructor
 *
 * Descriptif court du constructeur primaire de la classe ‘C‘.
 *
 * @param p Descriptif du paramètre de ce constructeur.
 */
class C(p:Int)
{
  /**
   * Descriptif court de l'attribut ‘a1‘.
   *
   * Descriptif long de l'attribut ‘a1‘.
   */
  var a1 = 0
  /** Descriptif court de l'attribut ‘a2‘. */
  val a2 = Array(293762, 89072398, 9872723, 2389723)

  /**
   * Descriptif court de la méthode ‘m1‘.
   *
   * Descriptif long de la méthode ‘m1‘.
   *
   * @param p Descriptif du paramètre ‘p‘ de la méthode ‘m1‘.
   * @return Descriptif de ce qui est retourné par la méthode ‘m1‘.
   */
  def m1 (p:Int):Int = p

  /**
   * Descriptif court de la méthode ‘m2‘.
```

```
   */
  def m2 = {}
}
```

Pour générer la documentation pour un ensemble de fichiers sources, il suffit alors de lancer la commande `scaladoc` sur ces fichiers, ce qui aura pour effet de produire une arborescence de pages de documentation en HTML affichables à l'aide d'un simple navigateur web (la racine de l'arborescence correspondant au fichier `index.html`).

Scaladoc propose de nombreuses autres étiquettes pour ajouter une multitude d'informations de types divers et supporte également le « markup » pour formatter le texte. Plus d'informations à ce sujet et une documentation plus complète se trouvent à cette adresse : `http://docs.scala-lang.org/overviews/scaladoc/for-library-authors.html`.