

Formation Scala

Olivier GIRARDOT

Lateral-Thoughts

16 Janvier 2018

Jour 2

Ce que vous allez apprendre

- Construire son projet en Scala, SBT
- Tester son projet en Scala, ScalaTest et ScalaCheck
- Créer une API en Scala, Scalatra
- Traitement de flux en Scala : les opérations sur les collections et For-Comprehension
- Les types algébriques et les monades
- La gestion des erreurs en Scala

Développement d'une API en Scala

Thème de l'exercice

- Nous sommes une société de livraison de colis
- Nous avons une base de donnée contenant la liste des colis et des destinataires
- Nous développons actuellement une interface d'administration pour superviser nos colis

Stack Technique

- L'API de cette interface sera développé en Scala, avec Scalatra
- Cette API renverra du JSON
- Le système de build sera SBT, et nous testerons l'application avec ScalaTest

Section 2

SBT

Qu'est ce que c'est SBT

Simple Build Tool

- C'est l'outil de build de Scala (comme Maven)
- L'outil de build sert à construire le projet. Pour cela, à un moment il va appeler le compilateur pour compiler le projet.
- Contrairement à Maven qui est en XML, SBT est en Scala : on peut faire du code Scala dans un fichier de build SBT
- L'outil est actuellement en version 1.0.3 et est développé activement, il vient de sortir dans sa première version stable

Définir un build

```
project/  
  build.properties  
  build.scala  
  plugins.sbt  
build.sbt
```

Commentaires

- On peut définir un build soit dans `project/build.scala`, ou dans `build.sbt`
- Il faut avoir au moins un de ces fichiers pour pouvoir faire un build avec sbt
- le fichier `plugins.sbt` sert à déclarer des plugins, un peu comme les plugins maven
- le fichier `build.properties` sert à déclarer les propriétés externes du build, par exemple la version de SBT à utiliser

Fichier build.sbt le plus simple

```
1 name := "FormationScala"
2
3 organization := "com.myCompany.training"
4
5 version := "1.0"
6
7 scalaVersion := "2.12.4"
```

Commentaires

- On définit les informations de build comme des valeurs dans une table clé-valeur : c'est le sens de l'opérateur `:=`, la partie à gauche c'est la clé, à droite c'est la valeur
- Sur des versions précédentes de SBT, on est obligé de sauter une ligne entre chaque valeur, ce n'est plus le cas aujourd'hui
- `name`, `organization` et `version` sont équivalents à `artifactId`, `groupId` et `version` de maven
- `scalaVersion` détermine la version de Scala avec laquelle le projet est compilé. Attention, les binaires Scala 2.10 et 2.11 ne sont pas compatibles

le fichier project/build.scala équivalent

```
1 import sbt._
2 import Keys._
3
4 object Build extends Build {
5   lazy val root = Project(id = "root", base = file(".")).settings(
6     name := "FormationScala",
7     organization := "com.myCompany.training",
8     version := "1.0",
9     scalaVersion := "2.12.4"
10  )
11 }
```

Commentaires

- C'est un peu plus verbeux
- Là on voit vraiment que le build est défini en Scala
- Pour les projets simples, c'est conseillé de rester avec la version build.sbt

Structure d'un projet SBT

```
src/  
  main/  
    resources/  
      <files to include in main jar here>  
    scala/  
      <main Scala sources>  
    java/  
      <main Java sources>  
  test/  
    resources/  
      <files to include in test jar here>  
    scala/  
      <test Scala sources>  
    java/  
      <test Java sources>
```

La commande interactive

```
10:30:44 vincent: ~/repository % sbt
[info] Set current project to FormationScala (in build file:/home/vincent/
repository/)
> compile
[info] Updating {file:/home/vincent/repository/}beginners...
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[success] Total time: 1 s, completed 16 sept. 2015 10:54:18
```

les commandes utiles

- **run** : lance le programme, s'il y a plusieurs programmes possibles à lancer, vous pouvez sélectionner le bon programme
- **compile** : compile le projet
- **test** : lance tous les tests
- **package** : crée un paquet jar
- **publish** : publie le paquet jar
- **clean** : supprime le dossier target
- **exit** : quitte la console

Le mode bash

```
11:36:48 vincent: ~/repository % sbt clean compile
[info] Set current project to FormationScala (in build file:/home/vincent/
repository/)
[success] Total time: 0 s, completed 16 sept. 2015 11:40:08
[info] Updating {file:/home/vincent/repository/}beginners...
[info] Resolving jline#jline;2.12.1 ...
[info] Done updating.
[info] Compiling 5 Scala sources and 1 Java source to /home/vincent/repository/
target/scala-2.11/classes...
[success] Total time: 6 s, completed 16 sept. 2015 11:40:14
sbt clean compile 26,85s user 0,83s system 286% cpu 9,666 total
```

Commentaires

- On peut lancer sbt dans le terminal comme Maven
- Dans ce cas, on peut même chaîner les opérations (ici clean et compile)

La gestion des dépendances

```
1 libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" % "test"
2
3 libraryDependencies += Seq(
4   "junit" % "junit" % "4.8" % "test"
5   "net.databinder" %% "dispatch-meetup" % "0.7.8"
6 )
```

Commentaires

- On définit les dépendances dans une séquence `libraryDependencies`
- On sépare les différentes informations de la dépendance avec `%`
- Les informations de la dépendances sont dans cet ordre : `groupId`, `artifactId`, `version`, `scope`
- Le séparateur `%%` permet d'ajouter la version majeur de Scala à la bibliothèque, par exemple ici on téléchargera l'artifact `scalatest_2.11`
- On peut adjoindre plusieurs dépendances en créant une séquence que l'on concatène à la séquence `libraryDependencies`

Bilan

Ce que vous avez vu

- Le système de build de Scala
- Les commandes SBT de base
- La structure d'un projet Scala
- Ajouter une dépendance dans un build SBT

Section 3

ScalaTest, ScalaCheck

Qu'est ce que c'est ScalaTest/ScalaCheck

ScalaTest

- C'est une bibliothèque de test (un peu comme Junit en Java)
- Elle est actuellement en version 2.2.4

ScalaCheck

- Bibliothèque de test par propriété qui s'interface bien avec ScalaTest
- Elle est actuellement en version 1.12.5

Listing 1: Ajouter la bibliothèque scalaTest et scalaCheck

```
1 libraryDependencies += Seq(  
2   "org.scalatest" %% "scalatest" % "2.2.4" % "test",  
3   "org.scalacheck" %% "scalacheck" % "1.12.5" % "test"  
4 )
```

Un test unitaire de Base

```
1 import org.scalatest._
2
3 class ReverseStringTest extends FlatSpec with Matchers {
4
5   "A reverser" should "reverse string" in {
6     Reverser.reverse("ABCD") should be ("DCBA")
7   }
8
9 }
```

Commentaires

- On étend FlatSpec pour récupérer les structures des tests
- On étend Matchers pour récupérer les assert (ici le should be)
- Des exemples de matchers peuvent être trouvés sur la page suivante :
http://www.scalatest.org/user_guide/matchers_quick_reference

Les tests par propriété

Principe

- On ne teste plus sur un cas en particulier mais sur une propriété
- Par exemple, sur la fonction `reverse`, on peut tester que $\text{reverse}(\text{reverse}(x)) = x$

Avantages

- Si on choisit bien les propriétés, on peut caractériser complètement une fonction
- Par exemple, on peut définir `reverse` par deux propriétés :
 - Pour deux listes a et b , $\text{reverse}(a ++ b) = \text{reverse}(b) ++ \text{reverse}(a)$
 - Pour une liste a , $\text{reverse}(\text{reverse}(a)) = a$

Exemple de test par propriété

```
1 import org.scalatest.prop._
2 import org.scalatest.{Matchers, _}
3
4 class ReverseStringPropertyTest extends FlatSpec with PropertyChecks with
  Matchers {
5
6   "reverse of reverse" should "return same string" in {
7     forAll { (s:String) =>
8       Reverser.reverse(Reverser.reverse(s)) should be (s)
9     }
10  }
11 }
12
13 }
```

Commentaires

- On étend le trait `PropertyChecks`
- On ajoute le terme *ForAll*, et on définit une fonction dans les accolades

Bonnes pratiques

Sur les tests unitaires

- On évite au maximum les mocks : la programmation fonctionnelle permet de ne pas avoir d'effets de bord et donc de pouvoir tester une fonction qu'avec ses entrées et ses sorties
- Voir <http://engineering.monsanto.com/2015/07/28/avoiding-mocks/>

Sur les tests de propriété

- À chaque fois que l'on tombe sur cas qui échoue, on extrait ce cas dans un test unitaire

Bilan

Ce que vous avez vu

- Scalatest, la bibliothèque de test de Scala
- Les tests par propriété

Section 4

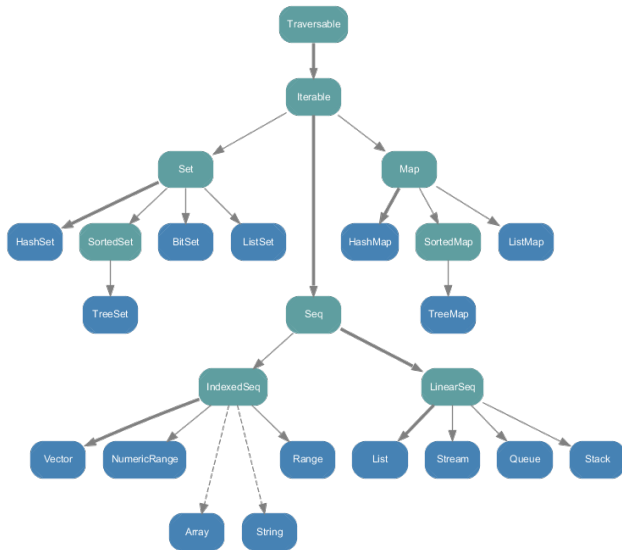
API Collection

Les collections en Scala

Les différents types disponibles

- `Map` : un ensemble de valeurs et leur clé associée
- `Set` : un ensemble sans doublon
- `Array` : un tableau
- `List` : une liste
- `Stream` : une liste paresseuse

La hiérarchie des collections en Scala



La hiérarchie des séquences (1)

Traversable

- Traversable est le trait d'origine de toutes les collections immutables
- Il définit la méthode `foreach` (qui n'est pas implémentée)

Iterable

- Iterable est un trait qui implémente la méthode `foreach`
- C'est le trait parent des `Map`, `Set` et `Seq`

La hiérarchie des séquences (2)

Seq

- Seq est un trait représentant des collections dont les éléments sont à un index fixé
- Deux traits importants héritent de Seq : IndexedSeq et LinearSeq
- Ces deux traits se distinguent surtout par la structure interne des collections qu'ils représentent

IndexedSeq

- IndexedSeq permet des accès rapides à chaque élément.
- Les séquences héritant de ce trait sont : Array, Range, Vector

LinearSeq

- LinearSeq permet l'accès et l'insertion rapide du premier élément
- Les séquences héritant de ce trait sont : List, Stream

Quelques méthodes de base sur les séquences

```
1 val aSeq = Array(1,2,3,4,5,4,3,2,1)
2 val anotherSeq = Array(1,2,3)
3 aSeq(0)
4 aSeq.indexOf(3)
5 aSeq ++ anotherSeq
6 aSeq.last
7 aSeq.head
8 aSeq.tail
9 aSeq.isEmpty
10 aSeq.length
11 aSeq.take(3)
12 aSeq.contains(10)
```

Commentaires

- On accède à un élément donné d'une séquence avec son indice entre parenthèses et non pas entre crochets
- ++ est l'opérateur de concaténation de deux séquences
- take permet de prendre les n premiers éléments d'une séquence

Les Listes

```
1 val aList = 1::2::3::Nil
2 val anotherList = List(1,2,3)
3 val emptyList = Nil
4 val anotherEmptyList = Nil
5 aList ++ anotherList
6 aList.reverse
```

Commentaires

- List est une classe qui étend LinearSeq
- L'accès et l'insertion du premier élément d'une liste est très efficace
- :: permet de créer une nouvelle liste avec un élément en plus en tête de la liste
- Nil représente la liste vide
- reverse permet d'inverser la liste, le premier élément devient le dernier

Les Tuples

```
1 val aTuple = (1, "coucou", true)
2 aTuple._1 // renvoie 1
3 aTuple._3 // renvoie true
```

Commentaires

- Un tuple represente un ensemble de données hétérogènes
- On peut mettre jusqu'à 22 éléments dans un tuple
- On appelle les éléments du tuple avec la méthode `_X`, où `X` est la position de l'élément dans le tuple.
- L'index des éléments d'un tuple commence à 1
- Un tuple est un type comme un autre qui peut être notamment utilisé en retour de fonction

Filter

```
1 val aList = 1::2::3::2::Nil
2 aList.filter(x => x == 2)
```

Commentaires

- `filter` est une méthode permettant de récupérer les éléments d'une séquence qui correspondent à un conditionnel passé en argument
- Par exemple ici le retour serait `List(2, 2)`
- Sa signature est : `filter(p: (A) => Boolean):Seq[A]`

Map

```
1 val aList = 1::2::3::Nil
2 aList.map(_*2)
```

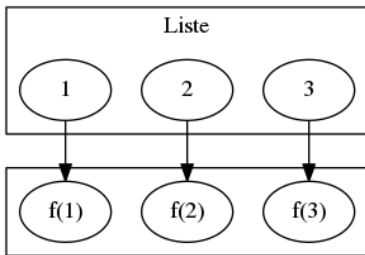
Commentaires

- map est une méthode permettant d'appliquer une fonction sur l'ensemble des éléments d'une séquence
- Par exemple ici le retour serait List(2,4,6)
- Sa signature est : `map[B](f: (A) => B):Seq[B]`

Map

```
1 val aList = 1::2::3::Nil  
2 aList.map(_*2)
```

Fonctionnement



FlatMap

```
1 val aList = 1::2::3::Nil
2
3 // retourne List(1,4,7,2,5,8,3,6,9)
4 aList.flatMap(x => List(x,x+3,x+6))
```

Commentaires

- flatMap est une méthode permettant d'appliquer une fonction qui retourne un itérable et d'ensuite d'aplatir tout cela dans une liste
- Si on avait utilisé map au lieu de flatMap, on aurait obtenu `List(List(1,4,7),List(2,5,8),List(3,6,9))`
- flatMap permet donc d'exécuter une fonction sur une séquence (map) puis d'aplatir le résultat (flat)
- Sa signature est :
`flatMap[B](f: (A) => GenTraversableOnce[B]):Seq[B]`

Reduce

```
1 val aList = 2::3::4::5::Nil
2 aList.reduce((x,y) => x * y)
```

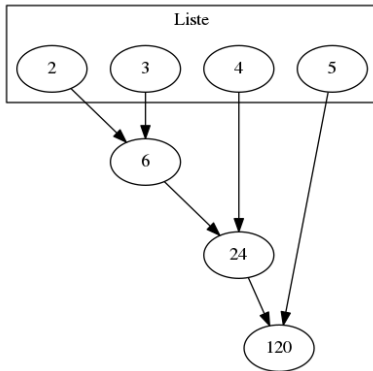
Commentaires

- reduce est une méthode permettant, en appliquant une fonction qui merge deux éléments de la séquence, de réduire la séquence à un élément
- Sa signature est : `reduce(op: (A,A) => A):A`

Reduce

```
1 val aList = 2::3::4::5::Nil  
2 aList.reduce((x,y) => x * y)
```

Fonctionnement



FoldLeft

```
1 val aList = 1::2::3::Nil
2
3 // Retourne "Liste : 1 2 3"
4 aList.foldLeft("Liste :")((acc,elem) => acc + " " + elem)
```

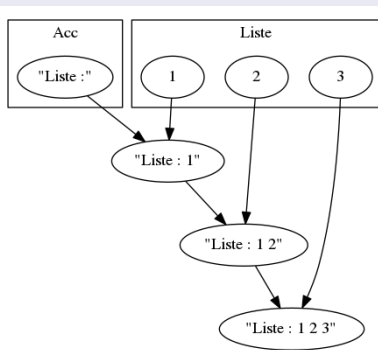
Commentaires

- FoldLeft est une méthode similaire à reduce, mais en plus on a un élément de départ et on peut choisir le type de retour de la fonction
- Cet élément de départ est l'élément qui est retourné si on appelle foldLeft sur une séquence vide
- Sa signature est : `foldLeft[B](z: B)(op: (B, A) => B): B`

FoldLeft

```
1 val aList = 1::2::3::Nil
2
3 // Retourne "Liste : 1 2 3"
4 aList.foldLeft("Liste :")((acc,elem) => acc + " " + elem)
```

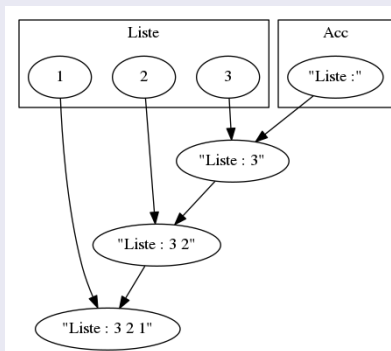
Fonctionnement



FoldRight

```
1 val aList = 1::2::3::Nil
2
3 // Retourne "Liste : 3 2 1"
4 aList.foldRight("Liste :")((elem,acc) => acc + " " + elem)
```

Fonctionnement



Différence entre FoldLeft et FoldRight

```
1 val aList = 1::2::3::Nil
2
3 // retourne List(3,2,1)
4 aList.foldLeft(List[Int]())((xs,x) => x::xs)
5
6 // retourne List(1,2,3)
7 aList.foldRight(List[Int]())((x,xs) => x::xs)
```

Commentaires

- Alors que foldLeft lit la séquence en commençant par la gauche, foldRight la lit en commençant par la droite
- C'est pour cela que foldLeft retourne la liste inversée, et foldRight retourne la liste sans la changer
- Si la fonction d'aggrégation est commutative, il vaut mieux utiliser foldLeft qui est plus efficace sur des linearSet et qui est récursive terminale

Les autres fonctions sur les séquences

```
1 val aList = 1::2::3::Nil
2 val anotherList = 3::2::1::Nil
3 aList.zip(anotherList)
4 aList.takeWhile(x => x != 2)
5 aList.slice(1,2)
6 aList.span(x => x != 2)
```

Commentaires

- `zip` permet d'associer deux séquences dans une unique séquence composée de tuples dont le premier élément est de la première séquence et le deuxième élément de la seconde
- `takeWhile` prend les premiers éléments de la séquence tant qu'ils respectent le prédicat passé en argument
- `slice` retourne un sous-ensemble de la séquence
- `span` sépare la séquence en deux avec d'un côté les premiers éléments qui respectent le prédicat, et de l'autre le reste des éléments à partir du premier élément qui ne respecte pas le prédicat

Tri d'une séquence

```
1 val aList = 3::1::2::Nil
2
3 // retourne List(1, 2, 3)
4 aList.sortBy(x => x)
5
6 // retourne List(3, 2, 1)
7 aList.sortBy(x => -x)
```

Commentaires

- sortBy permet de trier une séquence
- Elle prend en argument une fonction qui va lui déterminer sur quel objet elle doit se baser pour le tri
- C'est l'ordre canonique qui est défini (alphabétique pour les String, croissant pour les entiers)
- On peut jouer sur l'ordre de tri en jouant sur l'objet qui va être trié (voir deuxième exemple)

Chainer les fonctions

```
1 val aRange = 1 to 100
2 aRange.filter(_ % 2 == 0).map(x => x*x).reduce((x,y) => x + y)
```

Commentaires

- Comme la plupart des méthodes présentées précédemment retournent une séquence, on peut chaîner les opérations
- Ici par exemple, je fais la somme de tous les carrés des nombres pairs de 1 à 100
- Les fonctions que l'on ne peut pas chaîner sont les opérations qui réduisent la séquence à un élément, comme les `fold` et `reduce`

À vous de jouer

Liste de courses

- On a une liste de courses dans le fichier `ShoppingList.scala` (Exercise03)
- Et on veut faire quelques calculs de comptabilité sur cette liste dans `Accountancy.scala`
- Faites ces calculs en utilisant les méthodes que vous venez de voir

Bilan

Ce que vous avez vu

- La structure de l'API collection
 - Seq, LinearSeq, IndexedSeq
- Quelques méthodes sur les séquences
 - head, tail, isEmpty, take
 - filter, map, flatMap, reduce, foldLeft, sortBy
- Les tuples

Section 5

For Comprehension

Qu'est-ce que c'est ?

```
1 val aList = List(1,2,3)
2 for {
3   x <- aList
4 } yield x+1
```

Commentaires

- La For Comprehension est un sucre syntaxique (facilité du langage) permettant d'éviter les longues compositions de `map`, `flatMap`, `filter`
- Par exemple, ici, l'expression est l'équivalente de `aList.map(x => x + 1)`
- Dans le bloc `for`, on détermine les collections sur lesquelles on va itérer
- `yield` permet de renvoyer le résultat

Quand on a plusieurs séquences

```
1 val anotherList = List("a","b","c")
2 for {
3   x <- 1 to 3
4   y <- anotherList
5 } yield (x,y)
```

Équivalent avec map et flatMap

```
1 val anotherList = List("a","b","c")
2 (1 to 3).flatMap(x => anotherList.map(y => (x,y)))
```

Et encore plus de séquences

```
1 for {  
2   x <- 1 to 3  
3   y <- 4 to 6  
4   z <- 7 to 9  
5 } yield x*y*z
```

Équivalent avec map et flatMap

```
1 (1 to 3).flatMap(x => (4 to 6).flatMap(y => (7 to 9).map(z => x*y*z)))
```

Commentaires

- On constate que pour les premières séquences on applique flatMap puis on applique map pour la dernière

Les filtres

```
1 for {  
2   x <- 1 to 10  
3   if x % 2 == 0  
4 } yield x  
5  
6 def isEven(x:Int) = x % 2 == 0  
7 for {  
8   x <- 1 to 10  
9   if isEven(x)  
10 } yield x
```

Commentaires

- On peut utiliser des filtres dans les For Comprehension, en utilisant un conditionnel if
- Comme dans tous les conditionels, on peut utiliser un prédicat
- ici, la traduction en map, flatMap, filter du premier cas serait :
(1 to 10).filter(_ % 2 == 0).map(x => x)

Les ranges

```
1 val aRange = 1 to 10
2 val anotherRange = 1 until 10
3 val evenRange = 0 to 100 by 2
4 val reverseRange = 10 to 1 by -1
5 evenRange.start
6 evenRange.end
7 evenRange.step
```

Commentaires

- Range est une IndexedSeq
- to renvoie un Range contenant l'élément passé en argument
- until renvoie un Range s'arrêtant à l'élément avant l'élément passé en argument
- On peut définir le pas avec le mot clé by
- On peut récupérer le début, la fin et le pas du range avec les méthodes start, end, step

À vous de jouer

Convertissez les expressions suivantes en For Comprehension

- `(1 to 10).map(x => x*2)`
- `(1 to 10).flatMap(x => (10 to 1 by -1).map(y => x + y))`
- `(1 to 10).filter(x => x % 3 == 0).map(x => x*x)`

Convertissez les For Comprehension suivantes en suite de map, flatMap, filter

- `for {x <- 1 to 100} yield x/2`
- `for {x <- 1 to 100; if (x < 50)} yield x+50`
- `for {x <- 1 to 50; y <- x to 50; if Math.sqrt(x*x + y*y).isValidInt} yield (x,y, Math.sqrt(x*x + y*y).toInt)`

Bilan

Ce que vous avez vu

- Itérer sur différentes collections à l'aide des For Comprehension
- Utiliser les filtres dans les For Comprehension
- Comment convertir les For Comprehension en suite de `map`, `flatMap`, `filter`
- La classe `Range`

Section 6

Types Avancés

Type algébrique de données

```
1 sealed trait Boolean
2 case object Vrai extends Boolean
3 case object Faux extends Boolean
```

Qu'est ce qu'un type algébrique de données

- C'est un type qui est la composition d'autres types
- Ici, par exemple, le type Boolean est la conjonction des types Vrai et Faux
- Ces types n'ont qu'une instance, c'est pour cela qu'ils sont déclarés en tant qu'object
- Ces types permettent le pattern matching

Liste comme Type algébrique de données

```
1 sealed trait Liste[+A]
2 case object Vide extends Liste[Null]
3 case class Cons[+A](tete:A, queue:Liste[A]) extends Liste[A]
```

Commentaires

- Une liste est un type algébrique de données : en effet, elle est définie par la conjonction de la liste vide, et d'un type contenant à la fois un élément (la tête) et une autre liste (la queue)
- C'est pour cette raison que l'on peut faire du pattern matching sur les listes
- Dans la bibliothèque standard Scala, le type Cons est appelé `::`, c'est pourquoi on peut écrire `1::2::3::Nil` pour créer la liste `List(1,2,3)`

Qu'est ce que la Variance

La Variance

- Cette notion permet de résoudre les problèmes d'héritage entre des types paramétrés
- Par exemple, si un type A hérite d'un type B, est-ce que Liste[A] hérite de Liste[B] ?

Les différents cas : soit un type A qui hérite d'un type B

- Si un type paramétré UnType est **invariant**, alors UnType[A] est complètement indépendant de UnType[B]
- Si un type paramétré UnType est **covariant**, alors UnType[A] hérite de UnType[B]
- Si un type paramétré UnType est **contravariant**, alors UnType[A] est parent de UnType[B]

La Variance en Java et Scala

La variance en Java

```
1 public class Invariant<A> {}  
2 public class Contravariant<? super A> {}  
3 public class Covariant<? extends A> {}
```

La variance en Scala

```
1 class Invariant[A]  
2 class Contravariant[-A]  
3 class Covariant[+A]
```

- On en déduit donc que Liste, qui est déclaré Liste[+A] est covariant
- Ce qui veut dire que si A hérite de B, alors Liste[A] hérite de Liste[B]
- On pourra donc utiliser Liste[A] dans des fonctions qui demandent Liste[B] en argument

Qu'est ce qu'une monade

- Une monade `Monad` est un type qui implémente deux fonctions :
 - une fonction `return` qui est de signature `T => Monad[T]`
 - une fonction `bind` qui est de signature `Monad[T] => (T => Monad[U]) => Monad[U]`

Plus concrètement en Scala

- La fonction `return` c'est le constructeur
- La fonction `bind` c'est `flatMap`
- Donc en fait, une monade, c'est un type paramétré qui implémente `flatMap` !

```
1 class Monad[A](inner:A) {  
2   def flatMap[B](f:A => Monad[B]):Monad[B] = f(inner)  
3 }
```

Intérêt des monades

Encapsuler l'aléatoire du monde extérieur

- L'un des principes de la programmation fonctionnelle, c'est d'avoir des fonctions transparentes référentiellement.
- Le problème, c'est que le monde entier autour du programme n'est pas référentiellement transparent
- Par exemple, une base de données peut renvoyer un enregistrement ou rien
- Les monades permettent d'encapsuler ce genre de comportements dans un type

Chainer des actions facilement

- Il est très facile de définir `map` à partir de `flatMap` :

```
def map[B] (f:A => B):Monade[B] = Monade(f(inner))
```
- Ainsi, il va être très facile de chainer les transformations avec une `Monade`

Option : Cas d'usage

Le problème

- Soit une fonction qui fait un appel à une base de données qui renvoie un enregistrement ou rien
- Qu'est ce qu'on retourne lorsqu'il n'y a rien ?

```
1 def retrieveRecord(id:Int):Record = {  
2   callDatabase(id:Int)  
3 }
```

Pourquoi ce n'est pas idéal

- S'il n'y a pas d'enregistrement, on retourne un type Null
- Or Null ne contient aucune information, c'est un type qui hérite de tous les autres, Null peut être n'importe quoi
- De plus, ensuite, à chaque fois que l'on va traiter cette réponse, on va devoir faire un test pour vérifier que l'instance n'est pas Null

Option : La Solution

La Solution

- On va utiliser un type `Option`
- `Option` est un type algébrique de données, contenant deux types `None` et `Some[T]`
- `Option` est une monade, donc ayant une méthode `flatMap` (et aussi une méthode `map`)

```
1 def retrieveRecord(id:Int):Option[Record] = {  
2   val record = callDatabase(id:Int)  
3   if (record == null) None else Some(record)  
4 }
```

- Nous renvoyons maintenant toujours une `Option`, et non plus soit un `Record` soit un type `Null`
- Plus besoin également de faire des tests de nullité, on appelle les fonctions sur ce type avec `map`

Méthodes sur le type Option

```
1 val anInt = 23
2 val anOption = Some(anInt)
3 val anotherOption = None
4 anOption.get
5 anotherOption.getOrElse(1)
```

Commentaires

- Une Option peut être soit vide (None) soit contenir quelque chose (Some(something))
- Pour récupérer la valeur de l'Option, on peut soit faire appel à get, soit faire appel à getOrElse
- getOrElse permet de définir une valeur par défaut à retourner si l'Option est vide
- Le type Option peut être utilisé dans une For Compréhension

Bilan

Ce que vous avez vu

- Les types algébriques de données
- Les notions d'invariant, contravariant, covariant pour les types paramétrés
- Qu'est ce qu'une monade et à quoi cela sert
- Le type Option

Section 7

Gestion des Erreurs

Lever une exception

En Java

```
1 public void throwException() throw Exception {  
2     throw new Exception("it fails");  
3 }
```

En Scala

```
1 def throwException {  
2     throw new Exception("it fails")  
3 }
```

Commentaire

- Pour lever une exception, c'est la même chose en Java et en Scala
- Par contre, Scala n'a pas de système de checked exception : on ne sait pas si la méthode `throwException` lève une exception en Scala.

Rattraper une exception

En Java

```
1 try {  
2     // Do something  
3 } catch (Exception1 e) {  
4     // Treat Error for Exception1  
5 } catch (Exception2 e) {  
6     // Treat Error for Exception2  
7 } finally {  
8     // Do another thing  
9 }
```

En Scala

```
1 try {  
2     // Do something  
3 } catch {  
4     case Exception1 => // Treat Error for Exception1  
5     case Exception2 => // Treat Error for Exception2  
6 } finally {  
7     // Do another thing  
8 }
```

Le problème avec les exceptions

Pourquoi lever et rattraper les exceptions est une mauvaise chose

- Une fois une exception levée, on ne sait jamais où elle sera rattrapée : la lecture du programme est moins facile
- Lever une exception casse la transparence référentielle des fonctions : en effet, une exception est rattrapée ou pas en fonction du contexte (si on est à l'intérieur ou à l'extérieur d'un try)
- On ne sait pas si une exception est levée ou non dans une méthode donnée, et on ne peut s'en rendre compte qu'au moment de l'exécution du programme

Le type Either

```
1 sealed trait Either[+E,+A]
2 case class Left[+E](value:E) extends Either[E, Nothing]
3 case class Right[+A](value:A) extends Either[Nothing, A]
```

Le Principe

- Le type Either est un type algébrique contenant deux types Gauche et Droite
- Si une erreur a été levée, c'est Gauche qui va être renvoyé avec un message d'erreur, une exception...
- Si tout s'est bien passé, c'est Droite qui va être renvoyé avec l'objet que l'on veut retourner

Le type Try

Le Problème

- J'appelle un service qui renvoie une exception, comment je fais pour récupérer l'exception ?
- Je peux le faire avec `Either`, mais je dois faire le lien moi-même avec un `try/catch`

```
1 sealed trait Try[+T]
2 case class Failure[+T](exception:Throwable) extends Try[T]
3 case class Success[+T](value:T) extends Try[T]
```

La Solution

- Le type `Try` est un type algébrique contenant deux types `Failure` et `Success`, on va donc pouvoir faire du pattern matching dessus
- C'est aussi une `Monade`, on va donc pouvoir chaîner les traitements

Usage de Try

```
1 import scala.util.{Try, Success, Failure}
2
3 val result = Try { /* code that may throw exceptions */ }
4
5 result match {
6   case Success(value) => println(value)
7   case Failure(e:NullPointerException) => println(e.getMessage)
8   case Failure(_) => println("Unknown Exception")
9 }
```

Commentaires

- On exécute le traitement qui lève des exceptions dans un bloc Try (avec une majuscule)
- Ensuite, on obtient une instance de Try sur laquelle on peut faire du pattern matching
- On peut même faire du pattern matching sur le type d'exception

Bilan

Ce que vous avez vu

- Lever et rattraper une exception en Scala
- Pourquoi les exceptions posent problème
- Le type Either, une manière élégante de gérer les erreurs
- Le type Try, pour encapsuler du code qui renvoie des exceptions