

# Formation Scala

Olivier GIRARDOT

Lateral-Thoughts

18 Janvier 2018

# Jour 2

## Ce que vous allez apprendre

- Injection de dépendances en Scala
- La concurrence en Scala

## Section 2

# Les Implicits

# Qu'est ce qu'un implicit

```
1 implicit def displayHello(x:Int):(Int, String) = (x, "Hello")
2
3 def treat(aList:List[Int])(implicit f: Int => (Int, String)) = {
4   aList.map(f(_))
5 }
6
7 treat(List(1,2,3))
8 treat(List(1,2,3))(x => (x, "coucou"))
```

## Commentaires

- On peut déclarer une fonction ou un objet comme implicit
- Et on peut déclarer ensuite des arguments d'une fonction comme implicit
- Lorsque la précédente fonction est appelée, si on ne passe rien pour l'argument défini comme implicit, le compilateur va chercher une fonction ou un objet déclaré comme implicit ayant le même type que la fonction ou l'objet passé en argument.
- C'est notamment comme cela que toutes les conversions de types (comme Int vers Long par exemple) sont déterminées

# Qu'est ce qui se passe s'il n'y a pas d'implicit

```
1 def treat2(aList:List[Int])(implicit f: Int => (String, Int)) = {  
2   aList.map(f(_))  
3 }  
4  
5 treat2(List(1,2,3))
```

```
scala> def treat2(aList:List[Int])(implicit f: Int => (String, Int)) = aList.map(  
    f(_))  
treat2: (aList: List[Int])(implicit f: Int => (String, Int))List[(String, Int)]  
  
scala> treat2(List(1,2,3))  
<console>:14: error: No implicit view available from Int => (String, Int).  
    treat2(List(1,2,3))
```

## Commentaires

- Si on ne définit pas d'implicit, la recherche d'implicit va être infructueuse
- Résultat, le compilateur lève une erreur

# Dans quel ordre les implicits sont résolus ?

## Ordre de résolution

- Le scope actuel (comme dans les exemples précédents)
- Les imports explicites : `import training.displayHello`
- Les imports généraux : `import training._`
- Dans le companion object du type demandé
- Puis les imports explicites, généraux et companion objects des arguments du type demandé de manière récursive

## Commentaires

- C'est pourquoi, à l'usage de certaines bibliothèques, on va vous demander d'importer des choses que vous pensez ne pas utiliser
- Par exemple en Spark, `import sqlContext.implicits._` pour des conversions de type
- Et c'est aussi l'une des raisons de la lenteur de compilation de Scala : à chaque fois que le compilateur rencontre un implicit, il faut qu'il aille chercher où l'implicit est défini

# Bonnes pratiques avec les implicits

## Règle n°1 des implicits

- On n'utilise pas les implicits

## Pourquoi

- Les implicits sont un outil très puissant, un implicit défini à un endroit A peut agir sur du code à un endroit B
- Surtout que la méthode de résolution des implicits fait qu'on peut très vite se fourvoyer sur l'implicit chargé
- Avec pour conséquence des résultats incohérents au runtime si le mauvais implicit est chargé
- Et plus il y a d'implicits définis, plus le risque augmente
- Donc on évite au maximum les implicits s'il y a une autre manière de faire

# Bilan

## Ce que vous avez vu

- Comment définir un implicite
- Comment utiliser un implicite dans les arguments d'une fonction
- L'ordre de résolution des implicits
- Pourquoi il faut éviter d'utiliser les implicits



## Section 3

# La Concurrency en Scala

# Calculs parallèles sur les séquences

```
1 val aList = (0 to 100).toList
2
3 aList.par
4 aList.par.map(_*2)
```

---

## Commentaires

- On peut transformer une séquence en une séquence dont les calculs vont se faire en parallèle en appelant la méthode `par`

# Attention avec les calculs parallèles

```
1 val aList = (0 to 100).toList
2
3 aList.reduce((x,y) => x + y)
4 aList.par.reduce((x,y) => x + y)
5
6 aList.reduce((x,y) => x - y)
7 aList.par.reduce((x,y) => x - y)
```

---

## Commentaires

- Si la fonction passée en argument d'un `reduce` (ou d'un `fold`) est commutative (c'est à dire que l'on peut inverser l'ordre des arguments sans changer le résultat), le calcul parallèle et le calcul classique donne le même résultat
- Si la fonction passée en argument d'un `reduce` (ou d'un `fold`) n'est pas commutative, le calcul parallèle va être complètement imprévisible

# Le contexte d'exécution global

```
1 implicit val executionContext = ExecutionContext.global
2 import ExecutionContext.Implicits.global
```

## Qu'est ce que c'est le contexte d'exécution global

- Un contexte d'exécution, c'est ce qui va déterminer dans quel contexte un calcul parallèle en Scala va s'exécuter : nombre de threads, comment ces threads vont être gérés
- C'est l'équivalent d'un `Executor` en Java
- Scala définit un contexte d'exécution global par défaut, on peut soit le définir explicitement (première ligne de l'exemple), soit l'importer de manière implicite (deuxième ligne). Ces deux lignes sont donc équivalentes
- Le contexte d'exécution global par défaut est un pool de threads ayant autant de threads que de processeurs sur la machine

# Utiliser des contextes d'exécution Java

```
1 implicit val executionContext = ExecutionContext.fromExecutor(new  
    ThreadPoolExecutor( /* your configuration */ ))
```

## Commentaires

- On peut importer un Executor Java directement à l'aide de la fonction `ExecutionContext.fromExecutor`
- Il y a différents types d'exécuteurs
  - **FixedThreadPool** : un nombre fixé de threads sont créés. C'est un pool adapté pour les opérations lourdes
  - **CachedThreadPool** : on crée autant de threads dont on a besoin, et on réutilise les threads qui ne sont plus utilisés. C'est un pool adapté pour les opérations courtes et nombreuses (par exemple des accès aux bases de données)
  - **ForkJoinPool** : chaque thread va essayer d'exécuter des sous-tâches d'autres thread. C'est un pool adapté pour une opération lourde qui peut se diviser en pleins de petites tâches

# Les Futures

```
1 import scala.concurrent._
2 import ExecutionContext.Implicits.global
3 import scala.util.{Success, Failure}
4
5 val future:Future[Int] = Future { Thread.sleep(10000); 1}
6
7 future onComplete {
8   case Success(value) => println(value)
9   case Failure(throwable) => println(throwable.getMessage())
10 }
```

## Commentaires

- Les Futures permettent d'exécuter un calcul non bloquant. On définit un résultat que l'on obtiendra "dans le future"
- On définit des callback en utilisant la méthode `onComplete`
- La méthode `onComplete` prend en argument une fonction ayant pour argument un objet `Try`
- Si vous jouez les commandes suivantes dans le REPL, vous pouvez voir que le résultat ne s'affiche pas tout de suite, mais que le REPL vous rend la main : les Futures ne sont pas bloquantes

# Changer le context d'exécution d'une Future

```
1 import scala.concurrent._
2 import scala.util.{Success, Failure}
3
4 val customExecutionContext = ExecutionContext.global
5 val future:Future[Int] = Future { Thread.sleep(10000); 1}(customExecutionContext)
6
7 future.onComplete {
8   case Success(value) => println(value)
9   case Failure(throwable) => println(throwable.getMessage())
10 }
```

## Commentaires

- Vous pouvez passer un contexte d'exécution personnalisé lorsque vous exécutez une Future
- Cela peut vous permettre de créer plusieurs contextes d'exécution et d'utiliser l'un ou l'autre en fonction de vos besoins

# Quelques méthodes utiles sur les Futures

```
1 import scala.concurrent._
2 import ExecutionContext.Implicits.global
3
4 val future = Future { Thread.sleep(10000); 1 }
5
6 future.isCompleted
7 future onSuccess { case value => println(value) }
8 future onFailure { case throwable => println(throwable.getMessage()) }
```

## Commentaires

- `isCompleted` permet de savoir si une future est terminée ou pas
- `onSuccess` et `onFailure` permet d'appliquer qu'une partie du `onComplete`
  - `onSuccess` ne fera quelque chose que si l'exécution de la Future est un succès
  - `onFailure` ne fera quelque chose que si l'exécution de la Future est un échec



# exécuter des Futures en parallèle

```
1 import scala.concurrent._
2 import ExecutionContext.Implicits.global
3
4 val future1 = Future { Thread.sleep(10000); 1}
5 val future2 = Future { Thread.sleep(10000); "Finished"}
6
7 val future3 = future1.flatMap(x => future2.map(y => println((x,y))))
```

## Commentaires

- Une Future est une Monade.
- Cela permet de facilement les composer
- On peut même utiliser les For Comprehension sur les Futures

# Attendre la fin d'une Future

```
1 import ExecutionContext.Implicits.global
2 import scala.concurrent._
3 import scala.concurrent.duration._
4
5 object BlockingProgram extends App {
6   val future = Future { Thread.sleep(10000); 1 }
7   Await.result(future, Duration.Inf)
8 }
```

## Commentaires

- On peut utiliser l'objet `Await` pour attendre la fin de l'exécution d'une Future
- Pour mettre une durée infinie, il faut utiliser `Duration.Inf`
- On peut aussi mettre une durée d'attente, par exemple `Await.result(aFuture, 10 seconds)`
  - Vous avez accès à `nanos`, `micros`, `seconds`, `minutes`, `hours`, `days`
  - Ne pas oublier d'importer `scala.concurrent.duration._` pour avoir accès à ces unités

# Le modèle acteur

## Qu'est ce qu'un Acteur

- Un Acteur est une classe qui implémente une méthode `receive`
- Cette méthode va permettre de recevoir des messages venant d'autres acteurs ou du monde extérieur
- Chaque Acteur a une queue de messages à traiter, la mailbox
- Une fois un message envoyé, l'objet qui a envoyé le message ne s'en occupe plus (pas de callback)
- Un système de gestion d'acteurs va attribuer les threads en fonction des besoins des différents acteurs

## Le modèle acteur

- Le modèle acteur c'est donc des unités de calcul (les acteurs) qui s'échangent des messages tout cela en non bloquant
- En Scala, ce modèle est implémenté dans la bibliothèque Akka : <http://akka.io/>

# Un Acteur dans Akka

```
1 import akka.actor.Actor
2
3 class anActor extends Actor {
4   def receive = {
5     case "Hello" => {
6       doSomething()
7       sender ! "How are you ?"
8     }
9   }
10 }
```

## Commentaires

- Lorsqu'un acteur reçoit un message, il fait du pattern matching sur le message reçu
- S'il doit renvoyer quelque chose à un autre acteur, il utilise le bout de code suivant  
`otherActor ! message`
- `sender` est l'acteur qui a envoyé le message

# Système d'acteurs dans Akka

```
1 import akka.actor.{Actor, ActorSystem, Props}
2
3 val system = ActorSystem("mySystem")
4
5 val myActor = system.actorOf(Props(new AnActor), "myActor")
6 val sameActor = system.actorSelection(system / "myActor")
7
8 myActor.tell("Hello", sameActor)
```

## Commentaires

- Pour créer un système d'acteur, il suffit d'appeler `ActorSystem` avec un nom
- `myActor` et `sameActor` désignent le même acteur
- Donc sur la dernière ligne, j'envoie le message "Hello" depuis l'acteur `sameActor` vers l'acteur `myActor`
- On remarquera que la bibliothèque s'appuie beaucoup sur les noms des objets plutôt que sur leur type, il faut donc faire très attention quand on travaille sur les noms des acteurs

# Bilan

## Ce que vous avez vu

- Comment paralléliser des traitements sur des collections Scala
- Comment définir un contexte d'exécution
- Comment effectuer des traitements asynchrones et en parallèle avec des Futures
- Le modèle d'acteur et son implémentation dans Akka

## Section 4

# Scala et Java 8

# Java 8

## Généralités

- Sorti le 18 mars 2014
- Version actuelle : 8u60

## Les nouveautés

- Fonctions et Lambdas
- Nouvelle API Stream



# Les fonctions

```
1 Function<Integer, String> toString = new Function<Integer, String>() {  
2  
3     @Override  
4     public String apply(Integer integer) {  
5         return integer.toString();  
6     }  
7 };
```

---

## Commentaires

- Une fonction en java 8, c'est une classe étendant l'Interface Function
- L'Interface Function est paramétrée par deux types, l'un étant l'argument de la fonction et l'autre le type de retour de la fonction
- Une fonction implémente une méthode apply
- Cela ressemble beaucoup aux fonctions Guava

# Les prédicats

```
1 Predicate<Integer> isNotZero = new Predicate<Integer>() {  
2     @Override  
3     public boolean test(Integer integer) {  
4         return integer != 0;  
5     }  
6 };
```

---

## Commentaires

- Un prédicat, en Scala, c'est simplement une fonction qui retourne un booléen
- En Java 8, les prédicats et les fonctions n'ont aucun lien
- Un prédicat en Java 8 est une classe étendant l'Interface `Predicate`
- Un prédicat implémente la méthode `test`
- Encore une fois, cela ressemble énormément à Guava

# Les lambdas

```
1 Interface Operation {
2     int operation(int x, int y);
3 }
4
5 Operation additionWithLambda = (int x, int y) -> x + y
6
7 Operation additionWithoutLambda = new Operation() {
8     @Override
9     public int operation(int x, int y) {
10         return x + y
11     }
12 }
```

## Commentaires

- Pour alléger un peu la création de fonctions, les lambdas ont été intégrés dans Java 8
- Les lambdas permettent de créer une instance d'une classe qui implémente une Interface contenant qu'une seule méthode
- En Scala, le lambda similaire aurait été défini comme ceci :  
(x:Int,y:Int) => x + y

# L'API Streams

```
1 List<Integer> aList = Lists.newArrayList(1,2,3);  
2 aList.stream();  
3 aList.parallelStream();
```

---

## Commentaires

- Pour ne pas tout casser, une nouvelle API a été créée en Java 8 en plus de l'API Collection
- Pour pouvoir utiliser l'API Stream, il nous faut d'abord transformer la collection en stream
- Pour cela, on appelle simplement sur une collection `stream()` ou `parallelStream()`
- La différence entre `stream()` et `parallelStream()`, c'est que `parallelStream()` va traiter les transformations sur le stream en parallèle

# Les méthodes disponibles

```
1 List<Integer> aList = Lists.newArrayList(1,2,3);
2 aList.stream();
3 aList.parallelStream();
4 aList
5   .map(x -> x + 1)
6   .filter(x -> x > 2)
7   .flatMap(x -> Lists.newArrayList(x, x+1, x+2))
8   .distinct()
9   .reduce((x,y) -> x+y)
```

## Commentaires

- Nous avons la plupart des fonctions que l'on a vues avec Scala : `map`, `flatMap`, `filter`, `reduce`, `distinct`
- Par contre, pas de `foldLeft`, du coup on est toujours obligé de renvoyer le même type que celui contenu dans la collection quand on fait un `reduce`
- De plus, les streams ne peuvent être consommés qu'une seule fois

# Récupérer les données après les traitements

```
1 Stream<Integer> aStream;  
2 aStream.collect(Collectors.toList())  
3 aStream.reduce((x,y) -> x + y).get
```

---

## Commentaires

- Pour récupérer une liste après avoir fait des traitements sur un stream, il faut faire appel à `collect(Collectors.toList())`
- Un `reduce` retourne un `Optional`, c'est l'équivalent du type `Option` en Scala
- Du coup, pour récupérer la valeur dans l'optional, il faut appeler la méthode `get`

# Java 8 par rapport à Scala

## Des progrès notables

- Les streams qui permettent de faire des traitements de flux sur des collections Java
- Les lambdas qui allègent ces traitements de flux par rapport à ce qu'on pouvait avoir sur les versions précédentes de Java avec Guava

## Mais encore loin derrière Scala

- Api Stream incomplète : pas de `fold` par exemple
- Des limitations lorsqu'on veut réutiliser le résultat d'un stream plusieurs fois : comme un stream ne peut être consommé qu'une fois, il faut ressortir de l'API pour y réentrer avec un `.collect(Collectors.toList()).stream()`
- Des lourdeurs syntaxiques pour les conversions `Collection` vers/depuis `Stream`

# Bilan

## Ce que vous avez vu

- La nouvelle API Stream de Java 8
- Les lambdas dans Java 8
- Pourquoi Java 8 est encore loin derrière Scala pour ce qui est traitements sur les collections