Java Avancé

Fiche 3-4 : Programmation fonctionnelle, les Streams

Partie 2

Table des matières

1 Objectifs

ID	Objectifs
AJ01	Comprendre les intentions de l'API Stream.
AJ02	Écrire des codes permettant d'extraire les informations souhaitées grâce à l'API Stream.
AJ03	Être capable de créer un stream à partir d'un fichier.

2 Derrière les fagots

2.1 Et le typage dans tout ça ?

Les Streams viennent de la programmation fonctionnelle. Or Java est à la base un langage orienté objets. Dans cette section nous allons voir un peu plus en profondeur les extensions apportées à Java pour permettre aux Stream de s'intégrer harmonieusement dans Java malgré cette différence de paradigme.

2.2 Les méthodes default

Les collections Java se transforment en stream comme par magie. Notre liste de choses devient en un seul appel de méthode un fameux stream …
Reprenons justement cet exemple, avant Java 8, `List<T>` ne contenait pas la méthode `stream()`.

Ajouter une méthode abstraite `stream()` dans `Collection` et une méthode concrète dans `ArrayList` aurait pu convenir pour notre cas mais quel cauchemar pour toutes les autres classes qui étendent `Collection` qu'il aurait alors fallu modifier. Il devenait impossible de faire évoluer les interfaces sans interrompre le fonctionnement des implémentations de celle-ci, quel dilemme ! C'est pour cela que Java 8 permet désormais d'avoir des **méthodes default** dans les interfaces ; les classes ne doivent pas fournir d'implémentation pour celles-ci. Il n'est donc plus obligé d'ajouter des implémentations dans toutes les classes qui implémentent des interfaces évoluant. Abracadabra !

Autrement dit, les interfaces fonctionnelles permettent de faire comme si on ajoutait l'implémentation d'une méthode à une classe sans modifier celle-ci.

Prenons un exemple concret. Il est possible d'invoquer la méthode *sort()* sur une List, pourtant, cette dernière n'en possède aucune implémentation directe. L'interface collection propose une implémentation par défaut de celle-ci.

Exercice :
Aller voir le code de l'interface Collections et voyez qu'il s'agit bien d'une interface et que celle-ci propose une implémentation par défaut pour la méthode sort :
https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Collections.java

Pour vraiment comprendre cette méthode, lisez la section suivante.

2.3 Petit détour par les Generics

En allant voir dans l'interface la méthode `sort`, vu avez dû remarquer que sa signature comportait des éléments relatifs aux types génériques assez complexe :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Pour l'instant nous n'avons fait qu'utiliser les génériques, nous n'avons jamais vu comment déclarer des méthodes ou classes génériques.

Prenons les choses une à une en commençant par le type du paramètre list :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ce type indique simplement que le paramètre list est une List de n'importe quel type symbolisé ici par T. On peut lui passer une List<String> aussi bien qu'une List<Integer>. Dans le cas de la List<String>, pour pourrait imaginer de remplacer tous les T par String pour comprendre ce que fait le code.

Etant donné que la méthode sort va trier les éléments, ces éléments doivent être comparables (ici on n'utilise pas de comparator, mais l'ordre « naturel » défini par le compareTo). Il faut donc que ce type implémente l'interface Comparable.
C'est ce que veut dire la partie en gras ci-dessous :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ce principe vaut aussi pour la déclaration des interfaces et classes. Prenons par exemple l'interface List : [jdk8/jdk8/jdk: 687fd7c7986d src/share/classes/java/util/List.java]

Nous voyons à la ligne 111 le code suivant :

```
public interface List<E> extends Collection<E> {
```

Nous voyons ici que List est déclaré générique par rapport au type E, sur lequel il n'y a aucune contrainte. Nous pouvons donc avoir une List de n'importe quoi.
Si nous avons une List<String>, E représente le type String quand il est utilisé dans le code de l'interface.
Ici List<String> extends donc Collection<String>

Ligne 238 nous voyons la méthode add :

```
boolean add(E e);
```

Encore une fois si nous avons une List<String> la signature de la méthode add doit être comprise comme

```
boolean add(String e)
```

Pour une explication encore plus détaillée, voir
https://docs.oracle.com/javase/tutorial/extra/generics/methods.html

2.4 Les interfaces fonctionnelles et lambda expression

Une expression lambda est en fait une fonction ; ce n'est pas une classe mais plutôt une méthode que l'on passe en paramètre. Une lambda expression est donc une fonction manipulée comme une valeur !
Or Java est un langage fortement typé et les lambdas doivent respecter cela.
Essayons de comprendre ce qui se cache derrière la magie. Regardons le code suivant :

```
List<Employe> listDesHommes = employes
        .stream()
        .filter(e -> e.getGenre() == Genre.HOMME)
```

La méthode filter reçoit en paramètre une lambda. Regardez avec IntelliJ la signature de cette méthode (click-droit -> Goto -> declaration or usage) :

```
Stream<T> filter(Predicate<? super T> predicate)
```

Nous voyons qu'elle prend en paramètre un *Predicate*. La lambda passée en paramètre est donc considérée comme étant de type *Predicate*.
Regardez maintenant la définition de Predicate avec IntelliJ :

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);
```

Nous voyons:

- Qu'il s'agit d'une interface
- Qu'elle possède une méthode abstraite *test* qui prend un paramètre d'un type quelconque et renvoie un booléen.
- Qu'elle est annotée @FunctionalInterface, ce qui l'oblige à n'avoir qu'une seule méthode abstraite (donc non implémentée par défaut).

Notre lambda

```
e -> e.getGenre() == Genre.HOMME
```

est en fait l'implémentation implicite de la méthode test contenue dans l'interface Predicate.

On pourrait écrire une classe implémentant l'interface Predicate :

```
public class PredicatGenreHomme implements Predicate<Employe> {
    @Override
    public boolean test(Employe e) {
        return e.getGenre() == Genre.HOMME;
    }
}
```

et l'utiliser ensuite dans notre filter :

```
List<Employe> listDesHommesBis = employes
        .stream()
        .filter(new PredicatGenreHomme())
```

La lambda expression dans ce contexte n'est autre qu'un raccourci pour l'implémentation et l'instanciation de l'interface Predicate ! La méthode filter va appeler test() sur chacun des employés contenus dans le stream.

L'interface Predicate est dite interface fonctionnelle car elle ne possède qu'une seule méthode abstraite. Elle représente donc la définition d'une fonction.

2.5 Exercices d'analyse

2.5.1 Paramètre de la méthode map

Ouvrez la classe ExerciceFunctionalInterface dans lequel se trouve la méthode ExMap qui contient le code suivant :

```
Stream<String> listeNom = employes.stream()
        .filter(e -> e.getGenre() == Genre.HOMME)
        .sorted(Comparator.comparingInt(Employe::getTaille)
                .reversed())
        .map( e -> e.getNom());
```

Trouvez le type du paramètre de la méthode map.
Faites le même raisonnement que nous avons fait au point précédent et transformez le code en implémentant cette interface plutôt que de passer un lambda.

2.5.2 Paramètre de la méthode foreach

Prenez la méthode ExForEach. Trouvez le type du paramètre de la méthode forEach.
Faites le même raisonnement que nous avons fait au point précédent et transformez le code en implémentant cette interface plutôt que de passer un lambda.

2.6 Des lambdas dans un autre contexte

Les lambda ne sont pas seulement utiles dans le contexte de l'API Stream. Les comparator que nous avons écrit lors du travail sur les collections peuvent aussi s'écrire avec des lambdas.

Exercice :
Dans la classe ExerciceFunctionalInterface prenez la méthode exComparator. Celle-ci utilise un Comparator en créant une classe et en l'instanciant. Modifiez le code pour remplacer le comparator par une lambda expression.

2.7 Quelques Predicate et Function

Dans cet exercice, vous utiliserez des `Predicate` et des `Function`. N'utilisez pas de Stream ! Vous devrez utiliser la méthode `test` des `Predicate` et la méthode `apply` des `Function`.

Remarque : ne passez pas par l'API Streams pour faire les points 1 à 4.

1. Dans la classe `Lambda` du package lambda, implémentez les méthodes `allMatches` et `transformAll` en respectant la javadoc fournie.
2. Modifiez la classe `TestLambda` en fournissant des expressions lambda qui correspondent aux indications en commentaires. Utilisez-la ensuite pour tester vos méthodes `allMatches` et `transformAll`.
3. Modifiez vos méthodes pour qu'elles puissent maintenant accepter des `List` de n'importe quel type (au lieu de uniquement des `Integers`).
4. Dans la classe `TestLambda`, décommentez la deuxième partie et complétez les expressions lambda.
5. Ajoutez deux méthodes à votre classe Lambda, appelées `filter` et `map`. Ces deux méthodes doivent faire exactement la même chose que `allMatches` et `transformAll`, mais **doivent** utiliser l'API Stream. Dans la classe TestLambda effectuez les mêmes tests que pour `allMatches` et `transform`.

2.8 Supplier appliqués au traitement de fichiers

Une autre interface fonctionnelle très utile est l'interface *Supplier*. Celle-ci possède une méthode get() qui ne prend pas de paramètre et renvoie un élément d'un type donné. Comme son nom l'indique, l'interface Supplier sert à fournir des données comme, par exemple, lorsqu'on lit un fichier. Voici une application :

La boîte StreamingVF vous fournit un fichier contenant les informations de tous ses employés afin de procéder à des opérations et des vérifications. Le fichier est un csv : **streamingvf.csv**.

Ce fichier représente les employés par 5 attributs : un id de 5 chiffres, un nom en majuscules, un prénom, une adresse mail et une indication de si la personne travaille à plein temps (TP) ou à mi-temps (MT).

Afin d'utiliser correctement un fichier de ressources comme celui-ci avec IntelliJ, il faut l'importer dans un dossier spécifiquement destiné aux ressources (ce n'est pas obligatoire mais vivement recommandé). Sur la racine de votre projet, clic droit >> new >> directory. Appelez ce dossier *resources*. Effectuez un clic droit sur ce nouveau dossier et sélectionnez Mark Directory as >> Resources Root Déposez ensuite le csv dans le dossier. De cette manière, le chemin d'accès à votre fichier sera toujours "*./resources/streamingvf.csv*".

Attention, si vous utilisez les modules d'IntelliJ, vous devrez préciser le nom du dossier contenant le module au début du chemin : "*./<module>/resources/streamingvf.csv*".

Dans le package `employee`, deux classes sont à votre disposition : `Employee`, qui représente un employé (attention aux attributs de la classe `Employee`, TP devient `true` et MT `false` dans l'attribut `fullTime`), et `EmployeeManagement`.

Complétez la classe `EmployeeManagement` avec les streams et le `try-with-resource` (https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) pour effectuer les opérations suivantes :

1. Afficher la première ligne du csv. Vous devrez au préalable compléter le `Supplier` en début de classe.

Pour la suite, vous devrez construire des Stream d'employés. Le constructeur d'`Employee` prend une String en paramètre et la splitte là où se trouvent des ";" (un fichier csv sépare ses colonnes par des ";"), afin d'initialiser ses attributs correctement. Vous devrez opérer de cette manière pour les exercices 2 à 7.

2. Construire une liste avec les noms de famille de plus de 8 caractères qui contiennent la lettre 'O' ou 'K'. Il faut compléter le prédicat et n'appeler qu'une seule fois la fonction `filter`.
3. Contruire une BiFunction (une Function avec deux paramètres) qui compte les occurrences d'un character passé en paramètre dans le prénom de l'employé passé en paramètre. Utiliser cette BiFunction pour créer une liste du compte d'occurrences du caractère 'e' dans les prénoms de chaque employé.
4. Indiquer si tous les employés ont un email se terminant par @streamingvf.be.
5. Indiquer le prénom d'un employé dont le nom de famille fait plus de 14 caractères ou « None » s'il n'en existe pas.
6. Retourner le nombre d'employé à mi-temps du fichier csv.
7. Construire une Map<Boolean,List<Integer>> regroupant les id des employés selon qu'ils travaillent à plein temps (`true`) ou à mi-temps (`false`).
8. Complétez la fonction `withLines(Consumer<Stream<String>> consumer)` qui s'occupe d'encapsuler le try-with-resource et son catch et fournira le stream de lignes au consumer qui lui est fourni. Complétez ensuite la fonction `printLongestName()` qui utilise `withLines` pour imprimer le plus long nom de famille du fichier.

3 Pour aller plus loin

3.1 Les Stream parallèles

Si l'on réfléchit bien, la plupart des traitements effectués par les stream pourraient être exécutés indépendamment sur différents cœurs de processeur. En effet, vu que les opérations sur un stream ne modifient pas le stream d'origine, il n'y a pas de risque de modifications concurrentes. Une version parallèle des stream est présente dans l'API. Pour en savoir plus aller voir https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html ou lisez la théorie se trouvant sur Moodle.

3.2 Exercices sur les Streams parallèles

Un stream parallèle va traiter en parallèle les différents éléments de la collection qu'il représente. L'interface DelayedOperations contient des opérations qui prennent un certain temps à s'exécuter :

- `fastMult2(Integer value)` qui retourne `2*value` après avoir attendu 1 ms.
- `slowMult2(Integer value)` qui retourne `2*value` après avoir attendu 10 ms.
- `ultraSlowMult2(Integer value)` qui retourne `2*value` après avoir attendu 100 ms.

- `randomlySlowMult2(Integer value)` qui retourne `2*value` après avoir attendu 500 ms si `value` est un multiple de 10, et après 1 ms sinon.

Elle contient également la fonction `runAndRecordTime(Runnabe process)` qui prend un processus en paramètre, l'exécute et retourne son temps d'exécution (référez-vous aux commentaires et à l'exemple qui se trouve dans la classe `ParallelStreams` pour voir comment l'utiliser).

Complétez la classe ParallelStreams :

1. Initialisez la liste `numbers` à l'aide d'un `IntStream` pour obtenir une liste d'entier de 1 à 100.
2. Complétez la fonction `serialMap()` qui retourne le temps nécessaire à appliquer chacune des 3 transformations, l'une après l'autre, à un stream issu de `numbers`.
3. Complétez la fonction `parallelMap()` qui retourne le temps nécessaire à appliquer chacune des 3 transformations en parallèle à un stream issu de `numbers`. Comparez le résultat avec celui de `serialMap`. À quoi correspond le gain de temps ?
4. Complétez les fonctions `serialFilteredBefore()`, `serialFilteredAfter()`, `parallelFilteredBefore()` et `parallelFilteredAfter()`, qui vont filtrer un stream issu de `numbers` en ne gardant que les nombres pairs avant (resp. après) d'y appliquer `ultraSlowMult2()` et retourner le temps d'exécution. Comparez les résultats et repérez l'influence de la mise en parallèle.
5. Complétez la fonction `randomMap()` qui retourne le temps nécessaire à appliquer `randomlySlowMult2` à un stream issu de `numbers`. Combien de temps devrait prendre cette méthode en théorie ? Quelle est la réalité ?