Java Avancé

Fiche 3-4 : Programmation fonctionnelle, les Streams

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 Concepts de base

2.1 Introduction à la programmation fonctionnelle

« En programmation fonctionnelle, on décrit le résultat souhaité mais pas comment on obtient le résultat. Ce sont les fonctionnalités sous-jacentes qui se chargent de réaliser les traitements requis en tentant de les exécuter de manière optimisée.
Ce mode de fonctionnement est similaire à SQL : le langage SQL permet d'exprimer une requête mais c'est le moteur de la base de données qui choisit la meilleure manière d'obtenir le résultat décrit. Comme avec SQL, la manière dont on exprime le résultat peut influencer la manière dont le résultat va être obtenu notamment en termes de performance.

L'API Stream de Java 8 propose une approche fonctionnelle dans les développements avec Java. Elle permet de décrire de manière concise et expressive un ensemble d'opérations dont le but est de réaliser des traitements sur les éléments d'une source de données. Cette façon de faire est complètement différente de l'approche itérative utilisée dans les traitements d'un ensemble de données avant Java 8.»[1]

Pour avoir une idée de la puissance de l'approche, regardons le code écrit en style fonctionnel ci-dessous qui calcule la taille moyenne des employés masculins à partir d'une collection. Nous allons détailler ces points au long de la fiche, nous ne nous attarderons pas sur les détails dans un premier temps.

```
double tailleMoyenneDesHommes = employes
        .stream()
        .filter(e -> e.getGenre() == Genre.HOMME)
        .mapToInt(e -> e.getTaille())
        .average()
        .orElse(0);
```

Nous voyons qu'il enchaîne une série d'opérations. Les trois premières opérations produisent un nouveau stream sur lequel on applique un nouveau traitement

```
        .stream()
```

➔Transforme la collection en stream

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

➔ Filtre le stream en ne gardant que les objets dont le genre est HOMME

```
        .mapToInt(e -> e.getTaille())
```

[1] Extrait de https://www.jmdoudoux.fr/java/dej/chap-streams.htm

➔ Converti (map) le stream d'Employe en stream d'Integer

```
        .average()
```
➔ Calcule la moyenne (ce n'est plus un stream)

```
        .orElse(0);
```
➔ renvoie 0 si le stream après filtrage est vide (dans le cas où il n'y avait aucun homme parmi les employés).

En programmation « normale » chacune de ces opérations aurait nécessité l'écriture d'une boucle explicite. Ici implicitement l'ensemble du stream est traité pour produire un nouveau résultat.

On peut résumer les streams par les points suivants :

- Stream prend une collection, un tableau ou des E/S.
- Les Stream ne modifient pas la structure de données d'origine. Ils fournissent uniquement le résultat selon les méthodes enchaînées.
- Chaque opération intermédiaire est exécutée et renvoie un stream en conséquence, de sorte que diverses opérations intermédiaires peuvent être enchaînées.
- Les opérations terminales marquent la fin du flux(stream) et renvoient le résultat.[2]

OBSERVATIONS & QUESTIONS

- Imaginez l'écriture du traitement des employés en programmation « classique ». Laquelle est la plus compact/lisible selon vous ?

2.2 Lambda expression et Filtrage

La ligne

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

Passe en paramètre de la méthode filter( ) une condition de filtrage booléenne.
Nous pourrions écrire la méthode suivante représentant cette condition :

```
boolean isHomme(Employe e){
    return e.getGenre() == Genre.HOMME;
}
```

Lorsqu'on utilise des streams, plutôt que de définir une méthode pour chaque traitement, on utilise ce que l'on appelle des **Lambda Expressions**. Une Lambda expression est ce qu'on pourrait appeler une méthode anonyme.

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

est une lambda expression qui prend un paramètre *e* qui sera de type *Employe* et renvoie une valeur booléenne. Une Lambda Expression qui renvoie un booléen est aussi appelé **prédicat**. Lors du filtrage par filter, cette Lambda Expression est appliquée à chaque élément contenu dans le stream.

[2] Issu de https://waytolearnx.com/2020/04/les-stream-en-java.html

Le type du paramètre e est déduit selon le contexte. Ici le stream contient des Employe ; e sera donc de type Employe.

Nous verrons lors de la seconde fiche de cet atelier que les lambdas expression peuvent être utilisées dans d'autres contextes que les streams ; comme par exemple pour définir des comparateur[3].

2.3 Collecter les résultats

L'exemple donné en introduction nous renvoie un entier après traitement. Souvent on voudrait retrouver une collection telle une liste.
La méthode *collect()* nous permet de transformer notre stream en autre chose.
Pour reprendre notre exemple si nous voulons uniquement une liste des employés homme, nous pourrions faire ceci :

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

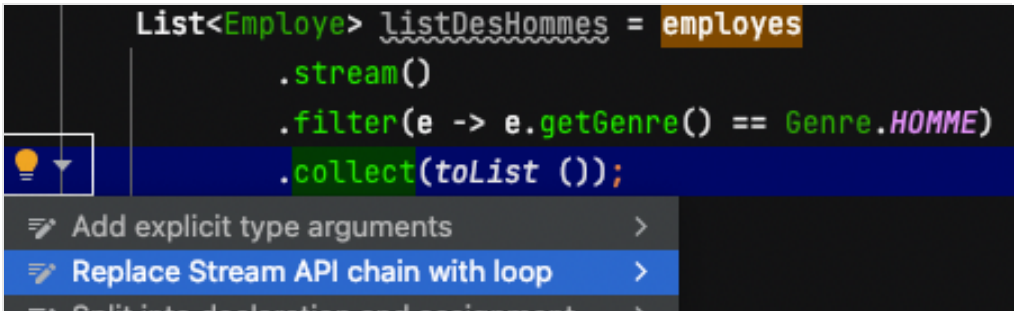
On passe en paramètre à *collect()*[4] un **Collector** qui indique vers quoi on veut aller ; ici une liste.

2.4 Comprendre la boucle implicite des Stream

Exercice :
Téléchargez le projet *AJ_Programmation_Fonctionnelle_Etudiants* qui se trouve sur Moodle et ouvrez le dans IntelliJ. Ouvrez la classe *ExercicesEmployes*. Vous y trouverez le code suivant :

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

Ensuite cherchez la petite ampoule de suggestion et sélectionnez « replace Stream API chain with loop » comme dans la figure ci-dessous



[3] https://mkyong.com/java8/java-8-lambda-comparator-example/

Regardez le code généré et cherchez le lien avec le code en Stream.

2.5 Écrivons quelques prédicats

Exercices :
Ouvrez la classe *ExercicesDeBase*.
Le projet contient les classes *Trader* et *Transaction* décrivant des transactions effectuées par des traders. Le *main* initialise une liste de transactions et lance les tests pour les exercices de base.
On vous demande en **utilisant les streams** de filtrer ces transactions de différentes façons et d'afficher le résultat. Ecrivez votre code dans les méthodes *predicats1*, *predicats2* et *predicat3*. Les résultats attendus sont dans le fichier *resultats.txt*.

1. Construire la liste de toutes les transactions de 2011
2. Construire la liste de toutes les transactions dont la valeur est > 600
3. Construire la liste de toutes les transactions de Raoul

Si vous vous demandez ce que veut dire le mot clef *var*, lisez la section suivante.

2.6 Inférence de type à la compilation (var)

Dans la méthode *predicat1*, nous avons déclaré notre Stream résultat de la façon suivante :

```
Stream<Transaction> s = transactions.stream();
```

Dans *predicat2*, nous la déclarons en utilisant le mot clef *var*

```
var s = transactions.stream();
```

Ce mot clef permet de ne pas devoir spécifier le type de la variable s. Il est déduit (inféré) lors de la compilation. Puisque la variable s prend lors de son initialisation *transaction.stream()*, le compilateur est capable de déduire que son type *Stream<Transaction>*.

Ceci fonctionne pour la plupart des initialisations comme par exemple :

```
var nom = "Gregory";
```

Attention, l'inférence de type à la compilation n'est pas la même chose que le typage dynamique comme en JavaScript ou Python. Ici le type doit est déterminable à l'initialisation. Une fois déduit, celui-ci ne pourra plus changer.

2.7 Itérer

Regardez la façon dont le résultats du filtrage est affiché.
Nous tentons d'abord d'afficher simplement le Stream comme nous le ferions avec une collection :

```
System.out.println("sout du Stream brut" + s);
```

Ceci nous affiche simplement le type et la référence du stream.
Du coup, on sera tenté d'écrire une boucle *for* pour parcourir tous les éléments. Pour pousser la logique fonctionnelle jusqu'au bout, nous utilisons plutôt la méthode *foreach* qui va parcourir le stream et appliquer une méthode passée en paramètre à chaque élément de celui-ci.

```
s.forEach(System.out::println);
```

System.out::println est un raccourci pour la Lambda expression suivante :

e -> System.out.println(e)

C'est ce qu'on appelle une **référence de méthode**. Elle peut être utilisée quand la lambda expression se contente d'appeler une seule méthode.

2.8 Transformer (map)

Attention : l'opération *map* des streams n'a rien à voir avec la structure de données *Map* !

En programmation, il est courant de devoir sélectionner des informations de certains objets comme en sql lorsque l'on sélectionne une colonne d'une table. Les méthodes `map` et `flatMap` de l'API Stream offrent ces possibilités.

La méthode `map()` prend en argument une fonction. Cette fonction est alors appliquée à chaque élément pour créer un nouvel élément. `map` renvoie un autre stream dont le type des éléments correspond à celui du renvoi de la fonction en paramètre :

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

Ici nous aurons donc un Stream de String.

Exercices :
Reprenez les classes sur les transactions. Construisez et afficher les listes suivantes respectivement dans les méthodes map1, map2 et map3 et vérifiez vos résultats avec le fichier resultats.txt.

1. Construire la liste des villes où travaillent les courtiers (traders).
2. Construire la liste de tous les courtiers de Cambridge.
3. Construire une **`String`** contenant tous les noms des traders séparés par une virugle. Comme il s'agit d'une String et non d'une liste, utilisez la méthode *joining()* dans votre *collect()*.

Observation : Vous avez des doublons ? Utilisez la méthode ***distinct***() après votre *map*() !

2.9 Trier

La méthode sorted() permet de trier les éléments d'un flux. Si l'on veut utiliser l'ordre « naturel » c'est-à-dire celui défini par le compareTo, on l'utilise sans paramètre comme dans cet exemple :

```
List<Integer> list = Arrays.asList(6, 2, 9, 1, 7);
list.stream().sorted().forEach(System.out::println);
```

Si l'on veut trier selon un autre ordre, il va falloir fournir un *Comparator* à cette méthode. On peut facilement créer un comparateur à l'aide de la méthode *comparing()* comme dans l'exemple suivant :

```
var employeTries =
        employes.stream()
                .sorted(Comparator.comparingInt(Employe::getTaille);
```

La méthode comparing() va créer un comparateur sur base de la valeur retournée par la méthode passée en paramètre.
On peut aussi inverser l'ordre de tri comme dans l'exemple suivant :

```
var employeTries =
        employes.stream()
                .sorted(Comparator.comparingInt(Employe::getTaille)
                .reversed());
```

Exercices :
Reprenez les transactions.

1. Construire la liste de toute les transactions triée par ordre décroissant de valeurs.
2. Construire une **`String`** contenant tous les noms de traders triés par ordre alphabétique.

2.10 Des lambdas expressions plus complexes

2.10.1 Corps plus complexe

Dans les exercices précédents, nous avions toujours utilisé une forme simple des lambas expressions avec un seul argument et dont le corps était constitué d'une expression simple comme par exemple

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

Le corps de cette lambda est une expression booléenne.
Nous pourrions écrire celle-ci de façon moins compacte mais tout aussi correcte :

```
e -> {
    if (e.getGenre() == Genre.HOMME)
        return true;
    else
        return false;
}
```

Nous voyons que nous pouvons écrire le corps de l'expression Lambda comme le corps d'une méthode « normale » pour peu qu'elle renvoie une valeur.

2.10.2 Lambda avec plusieurs paramètre

De même tout comme les méthodes, il est possible d'avoir une lambda qui prend plusieurs paramètres :

```
(a, b) -> a + b
```

Ici ce code est équivalent à une méthode prenant du paramètre et renvoyant leur somme. Les lambdas à plusieurs paramètres sont utilisés dans l'opération de réduction (reduce) que nous verrons à la section suivante.

2.11 Réduire (reduce)

Réduire un stream consiste à le … réduire à une seule valeur. Pour cela, les éléments contenus dans le stream vont être combinés deux à deux à l'aide d'un opérateur binaire ; c'est-à-dire une expression lambda prenant deux paramètres.
Par exemple si l'on veut faire la somme des éléments contenus dans notre stream, on peut utiliser la Lambda expression `(a, b) -> a + b` comme dans le code suivant :

```
List<Integer> numbers = Arrays.asList(3,4,5,1,2,-5);
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

Ce code correspond à la boucle suivante en *vieux* Java :

```
int sum = 0 ;
for(int i : numbers){
    sum+=i ;
}
```

Nous voyons ici que la méthode *reduce()* prend deux paramètres : la lambda expression et une valeur neutre (ici 0). Cette valeur neutre permet d'éviter les erreurs si le stream est vide !

Un autre exemple classique est la prise du maximum :

```
Integer max2 = Stream.of(1, 2, 3, 4, 5)
        .reduce(Integer.MIN_VALUE, Integer::max);

System.out.println(max2);
```

Exercices :
Toujours sur les transactions

1. Afficher la **valeur** max des transactions
2. Afficher la **transaction** dont la valeur est la plus petite. Attention : on demande bien d'afficher la transaction et non sa valeur ! Vous **ne pouvez pas** utiliser la méthode *min*. Au besoin, créer une transaction « neutre » avec une valeur de *Integer.MAX_VALUE*.

2.12 Petit résumé

Un stream est une séquence d'éléments provenant d'une source qui supportent des traitements de données :

- il ne possède pas les données qu'il traite ;
- il conserve l'ordre des données[5] ;
- il s'interdit de modifier les données qu'il traite ;
- il traite les données en une passe ;

Il y a deux grands types d'opérations :

- Les **opérations** dites **intermédiaires** ; elles renvoient un autre stream en résultat, ce qui permet de les connecter pour former une requête. Il s'agit des opérations `filter`, `map`, `sorted`, `distinct` ou `limit`.

[5] Sauf dans le cas des stream parallèles (que nous ne verrons pas dans cet atelier).

- Les **opérations terminales**, elles, produisent un résultat à partir d'un pipeline. Ce résultat n'est pas un stream, il peut s'agit d'un nombre, d'une liste ou même `void`. Il s'agit des opérations comme `collect, forEach` ou `count`

3 Concepts (un peu) plus avancés

3.1 Introduction

Il est possible de faire tout ce que l'on faisait en programmation classique avec l'API Stream. Dans le chapitre précédent vous avez vu les principes de bases. Autour de ceux-ci vont se greffer une multitude de méthodes utilitaires et types. Nous n'allons lever qu'un tout petit coin du voile. Pour aller plus loin, voir la théorie dans Moodle ou encore ce lien : https://www.jmdoudoux.fr/java/dej/chap-streams.htm.

3.2 Gestion du vide (type Optional<>)

Supposons par exemple que nous ayons besoin pour changer une ampoule d'un employé mesurant au moins 2m10. Nous n'avons pas besoin d'une liste ici mais d'un seul objet. L'API Stream fourni les méthodes *findFirst* et *findAny* permettant respectivement de renvoyer le premier élément d'un stream ou n'importe quel élément.
On peut donc écrire ceci pour notre recherche de géant :

```
Employe geant = employes.stream()
        .filter(e -> e.getTaille() > 210)
        .findAny();
```

Malheureusement, il se pourrait qu'en entrée ou après filtrage on se retrouve avec un Stream vide. Heureusement, l'API Stream, pour éviter les bugs dus à null, a introduit la classe *Optional* qui permet de représenter l'existence ou l'absence de résultat. Il s'agit en fait d'un conteneur de réponse qui indique s'il y a une réponse (isPresent) ou encore fournit la réponse (get).

Notre code précédant ne compilait pas en fait car le type de retour de *findAny* est *Optional<Employe>*. Pour corriger cela, on peut donc écrire maintenant ceci :

```
Optional<Employe> geantEnOption = employes.stream()
        .filter(e -> e.getTaille() > 210)
        .findAny();

if (geantEnOption.isPresent()){
    System.out.println("Le géant: " + geantEnOption.get());
}
```

 Notez que *get()* lève une exception s'il n'y a rien dans l'Optional.

La méthode *reduce()* renvoie également un *Optional* si elle est utilisée sans fournir de valeur neutre.

Si l'on veut rester en fonctionnel pur, on peut utiliser la méthode *orElse* qui renverra la valeur passée en paramètre s'il *l'Optional* est vide :

```
Employe geante = employes.stream()
        .filter(e -> e.getTaille() > 210)
        .findAny()
        .orElse(new Employe(Genre.FEMME, 220, "La géante"));
```

Exercices :

Ouvrez la classe ExercicesOptional. Dans les exercices sur les reduce, on utilisait la version de *reduce()* avec une valeur neutre. Dans les méthodes optional1 et optional2 copiez coller vos solutions pour le reduce1 et reduce2. Modifiez les pour utiliser maintenant la version à un seul paramètre. Testez aussi avec un stream vide.

1. Pour l'exercice trouvant la valeur max, utilisez *orElse()*
2. Pour l'exercice dans lequel on demandait la transaction de valeur minimale, affichez un message lorsque l'optional est vide après réduction. Vous pouvez faire cela en deux étapes.

3.3 Grouper

Le regroupement (`group by`) est une opération fréquente en base de données mais elle devient vite complexe en programmation traditionnelle. Par exemple, pour regrouper les plats selon leur type :

```
Map<Type, List<Dish>> platsGroupes = new HashMap<>();
for (Dish d : menu) {
      Type t = d.getType();
      if (platsGroupes.get(t) == null)
            platsGroupes.put(t, new ArrayList<Dish>());
      platsGroupes.get(t).add(d);
}
```

En Java 8, la ligne de code suivante suffit :

```
Map<Type, List<Dish>> pGroup2 = menu.stream().collect(groupingBy(Dish::getType));
```

`groupingBy` est une fonction de classification. Elle peut reposer sur une propriété de l'élément mais aussi sur des critères plus complexes.

Considérant l' `enum CaloricLevel { DIET, NORMAL, FAT }`

```
menu.stream().collect(
            groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            } ))
```

permet de construire une `Map<CaloricLevel, List<Dish>>`.

On peut également transmettre en second argument de la méthode `groupingBy` n'importe quel autre collector. Par exemple

```
Map<Type, Long> pg = menu.stream().collect(groupingBy(Dish::getType, counting()));
```

compte le nombre de plats de chaque type.

Ouvrez la classe `ExerciceGroupingby` et complétez les méthode groupBy1, groupBy2 et groupBy3 en utilisant les streams afin de construire d'afficher les dictionnaires suivants à partir de la liste de transactions présente dans la classe.

1. `Map<Trader, List<Transaction>>` Transactions du trader
2. `Map<Trader,Long>` (nombre de transactions de ce trader). Pour ceci il faudra utiliser la méthode *counting*() dans le collector.
3. `Map<TransactionsLevel,List<Transaction>>`

Considérant l'**`enum`** `TransactionsLevel {`***`VERY_HI, HI, LO, ME`***`},` les transactions sont réparties selon les critères suivants :

- si valeur >= 1000, elle est `VERY_HI` ;
- si 800 <= valeur <1000, elle est `HI` ;
- si 600 <= valeur < 800, elle est `ME` ;
- sinon elle est `LO`.

La map ne doit pas être triée.