Java 8 et les Streams (2)

Marianne Simonot

.

exercice 1

Aide-mémoire

Les opérations court-circuit (1)

Les méthodes vues jusqu'ici (`map` et `filter`) s'utilisent lorsqu'on on a besoin de traiter l'ensemble des données du stream.
Dans certains cas, il n'est pas nécessaire de traiter tous les éléments. C'est le cas par exemple quand on cherche si tous les éléments vérifient une propriété : dès qu'on rencontre un élément qui ne vérifie pas la propriété, nous pouvons stopper le parcours et répondre `false`.
L'api `Stream` nous fournit des méthodes pour répondre à ces cas de figure. Ces méthodes, appelées court-circuit, seront implémentées de façon à ce que le parcours de la structure s'achève dès que cela est possible.

méthode `AnyMatch` Cette méthode s'applique un stream et retourne un booléen. C'est donc une méthode terminale. Elle possède un prédicat en paramètre et retourne `true` dès que au moins un des éléments du Stream vérifie le prédicat. Elle vérifie donc qu'il existe un élément du stream vérifiant le prédicat. Cette méthode stoppera donc le parcours du stream dès qu'un tel élément est trouvé.

méthode `AllMatch` Cette méthode s'applique un stream et retourne un booléen. C'est donc une méthode terminale. Elle possède un prédicat en paramètre et retourne `true` ssi tous les éléments du Stream vérifient le prédicat. Cette méthode stoppera donc le parcours du stream dès qu'un élément ne vérifiant pas le prédicat est trouvé.

méthode `NoneMatch` Cette méthode s'applique un stream et retourne un booléen. C'est donc une méthode terminale. Elle possède un prédicat en paramètre et retourne `true` ssi aucun élément du Stream ne vérifie le prédicat. Cette méthode stoppera donc le parcours du stream dès qu'un élément vérifiant le prédicat est trouvé.

1. Utilisez les streams pour tester s'il existe quelqu'un dans `coll` qui a moins de 20 ans. La réponse doit être `true`.
2. Utilisez les streams pour tester s'il existe quelqu'un dans `coll` qui a moins de 15 ans. La réponse doit être `false`.
3. Utilisez les streams pour tester si tous les gens de `coll` sont majeurs. Vous donnerez une version avec `AllMatch` et une autre avec `NoneMatch`

exercice 2

Aide-mémoire

Les opérations court-circuit (2) et `Optional`

2 autres méthodes court-circuit sont fournies : `findAny` et `findFirst`. Comme les précédentes, elles sont terminales.

méthode `findAny` Cette méthode s'applique un stream et retourne l'un quelconque des éléments du stream.

méthode `findFirst` Lorsqu'un stream est obtenu à partir d'une structure ou les éléments ont une place (les listes, les sortedSet) il conserve l'ordre des éléments. En ce cas, la méthode `findFirst` permet de retourner le premier élément du stream. Cette méthode est à utiliser avec parcimonie car elle difficile à paralléliser.

`Optional<T>` Lorsque le stream ne contient pas d'élément, `findAny` et `findFirst` ne sont pas définies. Plutôt que de retourner `null` ou de lancer une exception, Java 8 introduit un nouveau type `Optional<T>` pour gérer les cas des valeurs manquantes. Un `Optional<T>` est une "surcouche" qui entoure soit un objet de type T, soit rien du tout.
Prenons un exemple :

colVide est vide
colPasVide a des éléments
sur `colVide` findAny ne retourne rien. `v` est donc l'optional `empty`
sur `colPasVide` findAny retourne un optional qui contient 2.

```
Collection<Integer> colVide = new ArrayList<Integer>();
Collection<Integer> colPasVide = Arrays.asList(2, 3, 4, 5);

Optional v = colVide.stream().findAny();
System.out.println(v);   => Optional.empty


Optional pv = colPasVide.stream().findAny();
System.out.println(pv); => Optional[2]
```

`isPresent` permet de tester si on a renvoyé un "vrai" qqechose
`get()` permet d'accéder à la valeur quand elle existe.
`orElse` permet de retourner soit la valeur de l'optionnel soit, lorsqu'il est vide, une autre valeur.

```
System.out.println(v.isPresent ()); => false
System.out.println(pv.isPresent ()); => true
// System.out.println(v.get ()); => NoSuchElementException
System.out.println(pv.get ()); => 2

System.out.println(v.orElse(-1)); => -1
System.out.println(pv.orElse(-1)); => 2
```

1. Utilisez les streams pour afficher le premier entier divisible par 3 de la liste `(2, 5, 7, 4, 12, 3, 8, 3)`.
2. Utilisez les streams pour ecrire une méthode qui prend une Collection de Gens et retourne un `Optional` contenant l'un quelconque des éléments de cette liste si elle n'est pas vide. `Optional.empty` sinon.
3. Utilisez les streams pour écrire une méthode qui prend une Collection de Gens et retourne l'un quelconque des éléments de cette liste si elle n'est pas vide. La méthode doit propager l'exception NoSuchElementException sinon.
4. Utilisez les streams pour écrire une méthode qui prend une Collection de Gens et affiche l'un quelconque des éléments de cette liste si elle n'est pas vide. La méthode doit afficher "impossible" si la liste est vide.
5. Utilisez les streams pour définir une méthode qui prend une Collection de Gens et retourne l'un quelconque des éléments de cette liste dont le nom commence par 't'. Propage l'exception NoSuchElementException si la liste est vide.
6. Utilisez les streams pour définir une méthode qui prend une Collection de Gens et retourne le nom de l'un quelconque des gens majeur de cette liste. Propage l'exception NoSuchElementException si la liste est vide.

exercice 3 : L'opération terminale `reduce`

Aide-mémoire

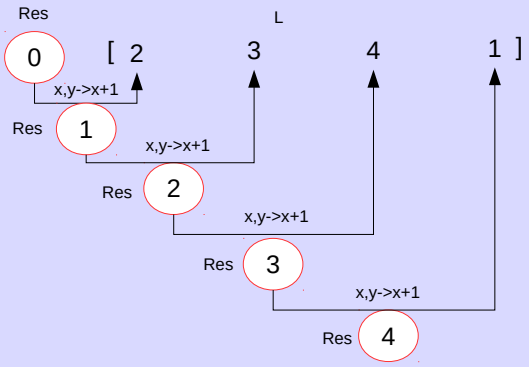
Nous connaissons 3 types d'opérations terminales :

1. Les collecteurs qui permettent de collecter les éléments d'un stream dans une collection : `collect(Collectors.toList())` et `collect(Collectors.toSet())`
2. les opérations court -circuit qui retournent des booléens ou des Optionnal.
3. `.count()` qui retourne le nombre d'éléments du stream.

L'opération `.count()` fait partie de la catégorie des opérations de **réduction**. elle réduit le stream en une valeur à partir d'une valeur de départ (0) et d'une opération appliquant itérativement une opération (x+1) sur les éléments du Stream. Pour compter les éléments d'une liste l avec une boucle nous écririons :

```
int res=0;
for(Integer n:l){
    res=res+1;
}
return res;
```

La variable `res` est initialisée à 0, puis à l'étape 1 vaut 0+1, à l'étape 2 vaut (0+1)+1. Autrement dit à chaque étape de la boucle, on applique la fonction $(x, y) -> x + 1$ avec res pour x et le ième élément de la liste pour y.



la méthode `T reduce(T identity, BinaryOperator<T> accumulator)` Ce mécanisme, tout comme ceux décrits par `filter` et `map` est très courant et est capturé par l'opération sur les stream : `T reduce(T identity, BinaryOperator<T> accumulator)`. Pour compter le nombre des éléments d'un Stream, nous pourrions ignorer `.count()` et utiliser la méthode plus générale `reduce` :

```
List<Integer> l = Arrays.asList(2, 3, 4, 1);
Integer nbre = l.stream().reduce(0, (x, y) -> x + 1);
```

Le premier paramètre de `reduce` est l'initialisation de la variable permettant de construire le résultat et le deuxième paramètre représente la fonction (ayant 2 paramètres) que l'on appliquera successivement à la variable résultat et au ième élément du stream.

la méthode `Optional<T> reduce(BinaryOperator<T> accumulator)` Java fournit aussi une méthode `Optional<T> reduce(BinaryOperator<T> accumulator)` avec comme seul paramètre la fonction à appliquer. En ce cas, l' initialisation est faite avec le premier élément du stream et la méthode retourne un Optional.

Associativité du paramètre `accumulator` L' opérateur binaire utilisé dans le reduce doit être associatif. Rappelons que cela signifie que $a \oplus (b \oplus c)) = (a \oplus b) \oplus c$ où $\oplus$ désigne l'opérateur. Ceci est nécéssaire car l'ordre dans lequel les éléments du Stream sont parcourus n'est pas garanti spécialement lorsque l'on utilise des streams parrallèles.

1. Utilisez `T reduce(T identity, BinaryOperator<T> accumulator)` pour calculer la somme des entiers de la liste `(2, 5, 7, 4, 3, 8, 3)`.
2. Utilisez `Optional<T> reduce( BinaryOperator<T> accumulator)` pour calculer la somme des entiers de la liste `(2, 5, 7, 4, 3, 8, 3)`.
3. Pourquoi est ce 10 qui s'affiche ?

```
List<Integer> l = Arrays.asList(4, 5, 7, 4, 3, 8, 3);
Optional<Integer> nbre1 = l.stream().reduce((x, y) -> x + 1);
System.out.println(nbre1.get ());
```

4. Utilisez `T reduce(T identity, BinaryOperator<T> accumulator)` pour calculer le produit des entiers de la liste `(2, 5, 7, 4, 3, 8, 3)`.
5. Utilisez un Stream (et nécessairement reduce) pour définir une méthode qui prend une collection de gens et retourne une String contenant les prénoms des gens de 20 ans séparés par une virgule.