2019

Compte rendu de TP : Rappel de notions de programmation objet

Java avancé – TP n°2

Jonathan CRÉTÉ – Filière Informatique

Exercice 1 - Path, Stream et try-with-resources

1) Avant de commencer, rappeler pourquoi, depuis la version 7 de Java, on représente un chemin en Java en utilisant la classe java.nio.file.Path et pas la classe java.io.File comme précédemment.

Tout d'abord, le package *java.nio.file.Path* reprend les trois-quarts des fonctionnalités de *java.io.File*. Par exemple les méthodes du package ne levaient pas d'exception en cas de mauvais comportement, ou encore on ne pouvait changer les permissions des fichiers. Il vaut mieux utiliser le package *java.nio.file.Path* que *java.io.File* à cause de l'encodage. En plus la classe java.nio.file.Path facilite pour le développeur le fonctionnement des entrées-sorties. La classe *java.io.File.java.io.File* est destinée à ne plus être utilisée (**deprecated**).

2) Créer un Path en utilisant la factory method Path.of() sur le fichier movies.txt.

On crée le Path.

```
var path = Path.of("data/movies.txt");
```

3) En utilisant la méthode Files.lines qui permet d'extraire toutes les lignes d'un fichier, afficher celles-ci à l'aide de la méthode forEach. Dans un premier temps, nous allons essayer de gérer l'exception éventuelle à l'aide d'un try-catch.

```
var path = Path.of("data/movies.txt");

try {
	Stream<String> st = Files.lines(path);
	st.forEach(System.out::println);
} catch(IOException e) {
	System.out.println(e);
}
```

4) En fait, le code que vous avez écrit est (très probablement) faux, car vous avez ouvert un fichier, vu celui-ci comme un Stream mais vous avez oublié de fermer le Stream pour fermer le fichier sous-jacent avec la méthode close.

Modifiez votre code.

On appel la méthode close() sur notre stream :

```
var path = Path.of("data/movies.txt");

try {
	Stream<String> st = Files.lines(path);
	st.forEach(System.out::println);
	st.close();
} catch(IOException e) {
	System.out.println(e);
}
```

5) Le code est encore (très probablement) faux, car si vous avez une exception lors du forEach, la méthode close ne sera jamais appelée.

Modifiez votre code en utilisant un finally pour résoudre le problème.

On développe le code avec le bloc finally

```
Stream<String> st = null;
	try {
		st = Files.lines(path);
		st.forEach((line) -> System.out.println(st));
	} catch (IOException e) {
		System.err.println(e.getMessage());
	} finally {
		st.close();
	}
```

6) Si vous vous êtes laissés guider par Eclipse, vous vous êtes probablement donné beaucoup de mal pour réussir à fermer un stream qui n'a même pas été ouvert...

Modifiez votre code pour ne pas avoir à initialiser le stream avec null..

```
Stream<String> st = Files.lines(path);

try {
	// Créer un stream des lignes du fichier
	st = Files.lines(path);

} catch (IOException e) {
	return;
}
try {
	st.forEach(System.out::println);
}
finally{
	st.close();
}
```

7) Quelle est la différence entre l'utilisation de la construction try-catch et un throws.

Pourquoi vaudrait-il mieux utiliser un throws ici?

Modifier le code en conséquence.

On utilise un try-catch pour reprendre sur l'erreur, gérer l'exception. Ceci permet de donner une exception plus précise et obtenir **Unchecked exception**

On utilise le mot clé *throws* pour ne pas avoir à la traiter.

8) En fait, il est plus pratique d'utiliser la construction try-with-resources, le try(...) car dans ce cas l'appel à close est fait automatiquement à la fin du bloc try.

Modifiez une nouvelle fois votre code.

La construction try-with-resources permet de déclarer une ou plusieurs ressources. Une ressource est un objet qui a besoin d'être fermé lorsqu'il n'est plus utilisé. De fait, On utilise un **try-with-resources** pour l'appel automatique à close comme demandé :

```
try(Stream<String> st = Files.lines(path)){
                    st.forEach(System.out::println);
}
```

9) A la maison, faites une recherche pour savoir pourquoi le try(..) est mieux que le try/finally.

Notez que les deux premiers morceaux de code que vous avez écrits semblaient bons et marchaient alors qu'ils étaient farcis de bugs. C'est le gros problème des tests : ce n'est pas parce qu'un test affiche ce qu'il faut que le code est correct.

Tout compte fait, l'instruction « *try-with-ressources* » est mieux que le *try/finally*, car il garantit que chaque ressource sera fermée lorsqu'elle n'est plus utilisée. Une ressource et un objet qui implémentent l'interface *java.lang.AutoCloseable*. Cette interface ne définit qu'une seule méthode abstraite : **la méthode close**. Ainsi, tout type de données concret implémentant cette interface proposera une méthode close. C'est ce dont nous avons besoin dans l'exercice, de manière à garantir que l'instruction « *try-with-resources* » qu'elle pourra forcer la fermeture de la ressource. De fait, si on utilise l'instruction sur un type n'implémentant pas l'interface, Le compilateur nous renverra une erreur de compilation.

Exercice 2 – Movies Stars

1) On cherche dans un premier temps à écrire une méthode actorsByMovie qui prend en paramètre un fichier (un Path) et renvoie une Map qui associe à un nom de film une List des noms des acteurs.

Pour cela, écrivez le code qui lit le fichier ligne à ligne avec la méthode Files.lines et qui stocke les infos dans la Map.

Pour séparer une chaîne de caractères en plusieurs parties, il existe la méthode String.split(). Il existe de plus une méthode Stream.skip() qui permet de ne pas prendre en compte certaines valeurs dans un stream.

Si ce n'est pas déjà fait, allez regarder la documentation de la méthode Collectors.toUnmodifiableMap et utilisez-la dans votre code.

```
public Map<String, List<String>> actorsByMovie(Path path) throws
IOException {
      try(Stream<String> lines = Files.lines(path)){
            var moviesMap = new HashMap<String, List<String>>();
            lines.forEach(line  -> {
                  // tableau de stream
                  var tab = line.split(";");
                  moviesMap.put(tab[0],
Arrays.stream(tab).skip(1).collect(Collectors.toList()));
            });
            return moviesMap;
      }
}
```

« Collectors.toUnmodifiableMap » retourne un Collector qui accumule les éléments d'entrée dans une Map non modifiable, dont les clés et les valeurs sont le résultat de l'application des fonctions de mappage fournis aux éléments d'entrée.

```
public static Map<String, List<String>> actorsByMovie(Path path) throws
IOException {
	try(Stream<String> lines = Files.lines(path)){

		return lines.map(line -> line.split(";"))
				.collect(
					Collectors.toUnmodifiableMap(
						tab -> tab[0], tab ->
Arrays.stream(tab).skip(1).collect(Collectors.toList())));
	}

}
```

2) On cherche à afficher le nombre total d'acteurs ayant joué dans au moins un film.

L'idée est de créer un Stream d'acteurs à partir de la Map précédente, puis de les compter.

À quoi sert la méthode Stream.flatMap() ?

Comment peut-on l'utiliser dans notre cas ?

Pour tester, affichez les 50 premiers acteurs à partir du Stream de tous les acteurs. Utilisez limit pour les 50 premiers et forEach pour l'affichage.

On développe la méthode :

```
public static void displayFirst50Actors(Path path) throws IOException {
		try(Stream<String> lines = Files.lines(path)){

			lines.flatMap(line ->
Arrays.stream(line.split(";")).skip(1)).limit(50).forEach(System.out::pri
ntln);
		}
	}
```

- La méthode de classe *Files.lines(path)* nous retourne une stream de String (chaque String représente une ligne du fichier)
- La méthode flatMap(...) attend une fonction en argument. La méthode permet de ramener à une dimension et d'pplatit l'ensemble des Streams de chaque élement. On passe en argument une lambda qui nous permet de d'exprimer que l'on veut créer un stream des éléments qui sont dans les éléments du stream que nous avons.
- La lambda prend en entrée chaque élément du stream (Objet String = ligne du fichier) et renvoi une nouvelle stream qui contient des objets String séparés par des « ; »
- Ensuite on crée une stream à partir d'un tableau de chaînes avec la méthode *Arrays.stream(...)*
- Il faut toujours qu'on enlève le premier élément (*titre du film*) avec la méthode *skip(...)*
- On limite au 50 premiers et on affiche avec la méthode *foreach(...)* qui prend en argument une méthode référence : *System.out.println();*

3) Au lieu d'afficher les 50 premiers, comptez le nombre d'acteurs et affichez le résultat.

```
public static Long countTotalActors(Path path) throws IOException {
	try(Stream<String> lines = Files.lines(path)){

		return lines.flatMap(line ->
Arrays.stream(line.split(";")).skip(1)).count();

	}
}
```

On a 382545 acteurs

4) En fait, le calcul précédent est faux car nous pouvons compter le même acteur plusieurs fois, il faut éviter les doublons !

Dans un premier temps, nous allons éviter les doublons en stockant les acteurs dans une structure de données qui a la propriété de ne pas enregistrer les doublons.

Quelle est l'interface Java qui correspond à cette structure de données ?

Quelle implantation de l'interface allons-nous choisir ?

Écrire le code de la méthode numberOfUniqueActors qui prend en paramètre une Map qui associe à un film la liste de ses acteurs et renvoie le nombre total d'acteurs différents ayant joué dans les films de la Map, sous forme d'un entier long.

On va utiliser l'interface JAVA **Set<E> qui** modélise un ensemble d'objets dans lequel on ne peut pas trouver de doublons. Contrairement à l'interface List, l'ajout d'un élément dans un Set peut échouer, si cet élément s'y trouve déjà (C'est pourquoi la méthode *add(...)* retourne un booléen). On va choisir HashSet

```
public  static long numberOfUniqueActors(Map<String, List<String>> map) {

	return map
	.values()
	.stream()
	.flatMap(Collection::stream)
	.distinct()
	.collect(Collectors.toList())
	.size();

}
```

La méthode values() retourne une vue de la collection des valeurs contenues dans la Map. On crée un stream. On appel la méthode size de Map de manière à retourner le renvoyer le nombre d'éléments dans notre liste

5) En fait, il existe une méthode Stream.distinct(). Comment peut-on l'utiliser pour trouver le nombre total d'acteurs?

Écrire le code correspondant

Voir **ci-dessus.**

La méthode permet d'éliminer les doublons. On l'appelle juste après flatMap(...).

6) On souhaite maintenant écrire une méthode numberOfMoviesByActor qui prend paramètre une Map qui associe à un film la liste de ses acteurs et calcule pour chaque acteur le nombre de films auxquels il a participé.

Quelle est le type de retrour de la méthode numberOfMoviesByActor ?

Pour l'implantation, nous allons tricher, on va repartir de notre Stream d'acteurs (non uniques) et compter le nombre d'apparitions de chaque acteur.

Nous allons pour cela utiliser un Collector particulier appelé Collectors.groupingBy(Function).

Rappeler comment marche la méthode collect et les Collector.

Comment peut-on utiliser le collecteur ci-dessus pour grouper les acteurs en fonction d'eux-mêmes ?

Quelle sera le type de retour de l'appel à collect ?

Écrire le code de la méthode numberOfMoviesByActor puis dans le main afficher le nombre de films auxquels a participé Brad Pitt (ou un autre acteur de votre choix), histoire de voir quelque chose !

La méthode de classe numberOfMovies retour un Objet de type **Map<String, Long>**

La méthode *collect(...)* permet de transformer un Stream qui contiendra le résultat des traitements de réduction dans un conteneur mutable

Collector est une Interface pour une opération de réduction qui accumule les éléments dans un conteneur mutable avec éventuellement une transformation du résultat une fois tous les éléments traités. Collectors (classe) est une implémentation de **Collector.**

```
public static Map<String, Long> numberOfMoviesByActor(Map<String,
List<String>> map) {

	return map.values()
	.stream()
	.flatMap(Collection::stream)
	.collect(Collectors.groupingBy(element-> element ,
Collectors.counting()));
}
```

7) Il existe une méthode Function.identity(). Comment peut-on l'utiliser dans notre cas ?

En remplaçant notre lambda par la méthode. En effet, *Function.identity()* va retourner la même instance que chaque occurrence de la lambda «*element -> element* ». La méthode va non seulement créer sa propre instance, mais en plus une classe d'implémentation distincte.

```
public static Map<String, Long> numberOfMoviesByActor(Map<String,
List<String>> map) {

	return map.values()
	.stream()
	.flatMap(Collection::stream)
```

```
        .collect(Collectors.groupingBy(Function.identity() ,
Collectors.counting()));
}
```

8) On cherche enfin à écrire une méthode actorInMostMovies qui prend en paramètre la structure de donnée qui associe à un acteur le nombre de film dans lequel il a joué (le résultat de la fonction de la question précédente), et renvoie une paire contenant l'acteur ayant joué dans le plus de film ainsi que le nombre de films dans lequel il a joué.

Sachant que la structure de donnée passée en paramètre pourrait être vide, quel doit être le type de retour de cette méthode?

En Java, il n'existe pas de classe Pair, car il existe l'interface Map.Entry et sa factory method Map.entry(first, second).

En fait, il s'agit de trouver le maximum parmi tous les couples (acteur, nombre de films), et l'on peut utiliser pour cela le collecteur Collectors.maxBy(). sur le stream des couples (acteur, nombre de films) de la structure de donnée passée en paramètre.

Attention, le comparateur attendu par maxBy ne doit pas être capable de comparer des nombre de films mais des couples (acteur, nombres de film).

En sachant que la structure de donnée passée en paramètre peut être vide, le type de retour de la méthode *actorInMostMovies(...)* doit être un objet de type Optional de manière à eviter null et donc de lever une *NullPointerException(...)*

```
public static Optional<Entry<String, Long>> actorInMostMovies(Map<String,
Long> map) {

      return map.entrySet()
            .stream()
            .collect(Collectors.maxBy((x1, x2) -> {
                  return x1.getValue().compareTo(x2.getValue());
            }));
}
```

Conclusion

Durant ce second TP de JAVA avancé, j'ai pu bien revoir les lambda tout en implémentant l'API des streams avec une liste et table de hachage. J'ai aussi pu comprendre dans quel cas on utilise un try/catch et dans quel cas le mot clé throws

J'ai compris que l'interface Stream permettait d'effectuer des opérations ensemblistes tout comme avec le langage déclaratif SQL.

J'ai rencontré des difficultés, dans l'exercice n°2, dés qu'il s'agissait d'utiliser la méthode de classe *flatMap*(son fonctionnement), mais aussi dés qu'on nous demande de choisir une structure de donnée approprié(un recherche a été nécessaire). Toutefois, je trouve que l'écriture de lambda trop verbeuses (trop longues), peut nuire à la lisibilité du code.

Je dois continuer à m'entrainer sur les lambda avec l'API des streams de manière à être plus efficace, et revoir les interfaces fonctionnelles de java.util.function.