POCA — TP noté n°2
Java programmation fonctionnelle et streams

Jean-Baptiste Yunès
Jean-Baptiste.Yunes@u-paris.fr

13 novembre 2025

Rendu final

Le rendu se fera par un clonage du dépôt gitlab du TP. Il faudra donc créer un projet pour cela et inviter l'enseignant au projet.

Généralités

Dans ce TP, on vise à utiliser le modèle de calcul en flot de données ainsi qu'à utiliser systématiquement le style fonctionnel de Java. On doit s'obliger à écrire un code concis et lisible.

Préliminaire (à lire attentivement)

Sont données quelques classes :

- `Identity` qui possède deux attributs : un nom (`lastName`) et un prénom (`firstName`),
- `Salary` qui possède deux attributs : un date (`date`) et un montant (`value`),
- `Person` qui possède comme attributs : une identité (`Id`), une catégorie d'emploi (`position`), une date de naissance (`birthDay`) et une liste de salaires (`salaries`).
- la classe `Data` qui contient essentiellement des chaînes de caractères qui serviront à effectuer des tests.
- la classe `Verifiers` qui contient deux fonctions utilitaires réalisant certains types de tests.

Il est strictement interdit de modifier ces classes en quoi que ce soit. Il ne faut pas ajouter quoique ce soit au package `fr.yunes.tp2`, merci.

D'autre part, il est donné un fichier `person.ser` qui contient la sérialisation d'objets `Person` qui devront servir à réaliser des tests (il doit y avoir 1031 enregistrements). Attention, selon votre IDE et sa configuration ce fichier doit être placé à un endroit particulier, à vérifier où. Mais normalement, il est automatiquement généré s'il n'existe pas.

Attention : il faut activer les exécutions avec les assertions afin que les tests soient réalisés. C'est l'option `-ea` de la JVM ; il faut vérifier avec votre IDE comment activer cette option.

On trouvera aussi un code source à compléter `MainTemplate.java`.

Vous ne devez rien changer aux tests proposés. Votre code doit fonctionner avec ces tests exactement. Vous ne devriez pas avoir besoin non plus de modifier le code de `main` sauf le contenu des fonctions nommées `questionX`.

Pour chaque question (sauf précision) il s'agit d'écrire une fonction comme :

```
public static type questionX(Stream<Person> s) {
    return ...;
}
```

Normalement, il ne devrait y avoir qu'une seule instruction... Tout doit se passer dans un seul `Stream`.

Pour vous aider à comprendre ce qui est attendu, on fournit le fichier `exec.log` qui contient l'exécution du programme tel que réalisé par l'enseignant (faillible).

1 Question n°1

Écrire les classes `FileSupplier` et `FileConsumer` qui permettent d'obtenir/écrire les `Person` (sérialisées) dans le fichier `person.ser`. Ceci devrait permettre de construire la liste des `Person` à l'aide du code situé entre les commentaires `BEGIN lecture` et `END lecture`.

Vous devriez obtenir une liste de taille 1031.

Normalement, si vous n'avez pas le fichier `person.ser`, il est automatiquement généré avec les bonnes données (à vérifier tout de même).

2 Question n°2

Avec un `Stream`, construire une liste des premiers caractères de tous les noms des `Person` de la liste. Liste sans doublons et ordonnée alphabétiquement.

3 Question n°3

Il est demandé de construire une liste ordonnée des noms de famille des `Person` qui commencent par la lettre `S`.

4 Question n°4

On demande la liste des `Identity` des `Person` triée par nom de famille puis en cas d'égalité par prénom.

5 Question n°5

Il est demandé d'obtenir la liste des `Identity` des `Person` qui sont `Position.MANAGER`. La liste doit être trié par nom de famille puis prénom.

6 Question n°6

On doit construire une structure de donnée `Map` où les clés sont des `Position` et les valeurs le nombre de personnes qui ont cette position dans l'entreprise.

7 Question n°7

On doit construire une `Map` dont les clés sont des années de naissance et les valeurs une collection de `Identity` des `Person` nées cette année là. Les clés de la `Map` doivent être triées de façon croissante, et les valeurs constituées d'identités triées par nom/prénom.

8 Question n°8

Ici on souhaite obtenir la liste triée des identités des managers qui gagnent au moins 4000$.

9 Question n°9

On doit extraire la liste ordonnée par date des `Salary` d'au moins 4000$.

10 Question n°10

Calculer la somme des salaires versés en janvier 2022.

11 Question n°11

Obtenir la liste ordonnée (nom/prénom) des paires identité, total des salaires versés.

12 Question n°12

Obtenir la liste ordonnée (nom/prénom) des paires identité, moyenne des salaires versés.

Aide : `IntSummaryStatistics`

13 Question n°13

Obtenir la liste ordonnée (nom/prénom) des paires identité, taux de progression entre le premier et le dernier salaire exprimé en pourcentage arrondi en entier ($100 * \frac{ds}{ps}$).

Question subsidiaire, obtenir à partir de ce résultat la personne qui a la plus grosse progression salariale.

14 Question n°14

Obtenir pour chaque tranche de 500$ de salaire, la liste ordonnée (nom/prénom) des identités.

15 Question n°15

Obtenir pour chaque tranche de 500$ de salaire, pour chaque première lettre du nom de famille, la liste ordonnée des identités.

16 Question n°16 [difficile] (indépendante)

Essayer de recoder des streams de votre cru. La classe s'appellerait `MyStream<T>` et permettrait de construire un « stream » à l'aide d'itérateurs sur des `Collections`. Sur ces streams on pourrait filtrer (`MyStream<T> filter(Predicate<T>)`) et transformer (`MyStream<U> map(Function<T,U>)`), avec une opération finale de type `void forEach(Consumer<T>)`.

Ajouter d'autres fonctionnalités au gré de vos envies...

17 Question n°17 [difficile] (indépendante)

Essayer de coder une liste fainéante. Elle pourrait devra être construite à l'aide d'un itérateur sur une séquence (infinie ou non). Quand on accède à une donnée (via un indice) si la donnée a déjà été lue on ira la chercher dans un cache (une liste java standard ou un tableau), si elle n'a jamais été lue on fait fonctionner l'itérateur pour la retrouver. L'affichage d'une liste fainéante ne fera qu'afficher les éléments déjà lus, les autres apparaîtront sous la forme `...`, par exemple `[1, 2, 3, 4, ...]` qui indique que les 4 premiers éléments ont déjà été lus (d'une manière ou d'une autre) et qu'il existe une suite d'éléments non encore lus.

On pourra aussi concaténer deux listes fainéantes entre elles pour obtenir quelque chose comme `[1, 2, 3, ..., 100, 101, ...]` ?

On pourra aussi faire qu'une liste fainéante soit un foncteur (`map`) ?