## Design patterns, Java

## Exercice streams.

## Préparation

Reprendre le programme qui avait servi aux exercices sur les lambda expressions.

On pourra présenter les réponses aux questions d, e, f, g du premier exercice sous forme d’instructions successives dans la méthode main

### 1- Utilisation directe

Pour les questions c à f, il faut enrichir la fonction main et utiliser systématiquement une des méthodes d’affichage des questions a et b, au choix.

Ajouter les différents traitements les uns après les autres.

Donc à la fin on devrait avoir à peu près ça :

public static void main(String[] args) {

List<String> maListe = Arrays.of("Un", "Deux", "Trois", "Quatre", "Cinq", "Six", "Sept", "Huit", "Neuf");

Stream<String> leStream;

// c

leStream = maListe.stream();

affiche(leStream.filter(…)...);

// d

leStream = Stream.of("Un", "Deux", "Trois", "Quatre", "Cinq", "Six", "Sept", "Huit", "Neuf");

affiche(leStream.filter(…)...);

// e

leStream = Stream.of("Un", "Deux", "Trois", "Quatre", "Cinq", "Six", "Sept", "Huit", "Neuf");

affiche(leStream.filter(…)...);

// f

leStream = Stream.of("Un", "Deux", "Trois", "Quatre", "Cinq", "Six", "Sept", "Huit", "Neuf");

affiche(leStream.filter(…)...);

}

public static void affiche(Stream<String> inStream) {

...

}

public static void affiche0(Stream<String> inStream) {

}

1. Modifier la méthode affiche(List laListe) pour qu’elle attende un « stream » de chaînes et qu’elle utilise la méthode forEach de ce stream pour en afficher le contenu. Chaque élément de la liste doit être précédé de deux espaces.
2. Créer la méthode affiche0(Stream<String> inStream) qui affiche aussi le contenu du stream reçu en paramètre mais sans espace devant les mots affichés. Il faut utiliser une référence à une méthode plutôt qu’une lambda explicite
3. Dans la fonction main, filtrer la liste pour ne garder que les chaînes de caractères contenant un « e » et afficher le résultat à l’aide d’une des deux méthodes d’affichage, au choix
4. Dans la fonction main, filtrer la liste pour ne garder que les chaînes de caractères contenant au plus 4 caractères et afficher le résultat à l’aide d’une des...
5. Dans la fonction main, filtrer la liste pour ne garder que les chaînes de caractères contenant un « e » et ayant au plus 4 caractères ; afficher le résultat à l’aide...
6. Dans la fonction main, effectuer le même filtre mais afficher les chaînes de caractères en majuscules à l’aide...

### 2- Function, Predicate

1. Reprendre les questions e et f de l’exercice précédent en définissant deux variables locales aUnE et aAuPlus4Caracteres contenant les prédicats correspondants. Filtrer en utilisant ces variables : filter(aUnE) par exemple.
2. Reprendre la question 1-f en n’utilisant qu’une fois la méthode filter. Cette méthode doit invoquer un prédicat composé de aUnE ET aAuPlus4Caracteres (consulter la documentation de Predicate ou utiliser l’autocomplétion sur aUnE. et un peu d’intuition).

### 3- Reduce, Collector

Dans cet exercice, nous allons utiliser les classes Tache et Agenda, ainsi qu’un programme principal (projet Maven fourni dans le dossier partagé)

On trouvera de nombreuses informations utiles sur le site <http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

Cette page est aussi fournie dans le dossier partagé.

1. Ajouter à la classe Agenda une méthode filter qui prend en paramètre un prédicat et qui renvoie la liste des tâches pour lequel ce prédicat est true

public Stream<Tache> filter(Predicate<Tache> inPredicate) {

return stream().filter(inPredicate);

}

1. Dans le programme principal, demander la liste des tâches qui ne sont pas effectuées, récupérer ces tâches sous forme d’un « stream »

agenda.filter(t -> !t.isEffectuee());

1. Demander ensuite le coût total de toutes les tâches

agenda.

agenda.stream()

.mapToDouble(t -> t.getCout())

.sum();

1. Demander ensuite les tâches ayant une date prévue en les regroupant par cette date. On doit obtenir une instance de « Map<Date, List<Tache>> »

Map<Date, List<Tache>> collect = agenda.stream()

.filter(t->t.getDatePrevue() != null)

.collect(Collectors.groupingBy(t -> t.getDatePrevue()));

1. Calculer les coûts totaux par jours (on doit obtenir une « Map<Date, Double> »). On pourra utiliser la méthode Stream.groupingBy avec deux arguments dont le deuxième est un Collector. Voir la méthode Collectors.summingDouble.

Map<Date, Double> coutParJour = agenda.stream()

.filter(t->t.getDatePrevue() != null)

.collect(Collectors.groupingBy(

t→t.getDatePrevue(), Collectors.summingDouble(Tache::getCout)));

1. Reprendre la question précédente en utilisant la méthode Stream.collect(Supplier, BiConsumer, BiConsumer).

Supplier<Map<Date, Double>> supplier = HashMap::new;

BiConsumer<Map<Date, Double>, Tache> accumulator

= (Map<Date, Double> a, Tache t)

-> {

Double valeur = a.get(t.getDatePrevue());

if (valeur == null) valeur = 0.0;

valeur += t.getCout();

a.put(t.getDatePrevue(), valeur);

};

BiConsumer<Map<Date, Double>, Map<Date, Double>> combiner

= (Map<Date, Double> a1, Map<Date, Double> a2) -> {

for (Date d : a1.keySet()) {

if (a2.containsKey(d)) {

a1.put(d, a1.get(d) + a2.get(d));

}

}

for (Date d : a2.keySet()) {

if (! a1.containsKey(d)) {

a1.put(d, a2.get(d));

}

}

};

Map<Date, Double> collect = agenda.stream()

.filter(t->t.getDatePrevue() != null)

.collect(supplier, accumulator, combiner);