Formation Java 17

API Stream

Sommaire

API Stream

Classe Collectors

Comparators

Un Stream ?

Un **stream** est une **interface** du package **java.util.stream**

La classe **ReferencePipeline** est l'implémentation par défaut.

Un **stream** permet de manipuler une **collection** sans la modifier.

Peut être infinie.

```
List<Ville> villes = new ArrayList<>();
Stream<Ville> stream = villes.stream();
```

Que propose le Stream ?

De nombreuses méthodes dont les principales sont :

- **filter** : permet de filtrer une collection sur la base d'un **prédicat** (une condition)

- **map** : permet de créer un nouveau stream d'objets de nature différente

```
int populationTot = villes.stream()
                          .filter(v -> v.getPopulation() > 100000)
                          .map(v -> v.getPopulation())
                          .reduce((t1, t2) -> t1+t2);
```

Qu'est-ce qu'un prédicat ?

La méthode **filter** permet de filtrer une collection sur la base d'un **prédicat** :

- Interface fonctionnelle java.util.functions.Predicate<T>

- Prend en paramètre un objet T et retourne un booléen

```
public interface Predicate<T> {

        boolean test(T t);

}
```

```
Predicate<Person> filtreSurPersonne = p -> p.getAge()>=100;
Predicate<Account> filtreSurCompte = a -> a.getBalance() >=1000.0;
```

Qu'est-ce qu'un mapper ?

La méthode **map** permet de créer une nouvelle collection sur la base d'un **mapper** :

- Interface fonctionnelle **java.util.functions.Function<T, R>**

- Prend en paramètre un objet T et retourne un objet R

```
public interface Function<T, R> {

    R apply(T t);

}
```

```
Function<Person, Account> mapper = p -> new Account(p, 100.0);
```

Les règles

Deux types d’opérations:

- opérations **intermédiaires**
- opérations **terminales**

Un stream ne peut être **traité qu’une seule fois**

Une seule opération terminale est autorisée

Opération intermédiaire

Ne déclenchent pas de traitement

Exemple : map, filter

```
somme = list.stream()
        .map(t -> t.getSolde())
        .filter(t -> t > 0)
        .reduce((t1, t2) -> t1+t2)
```

Opération terminale

Déclenche un traitement

Exemple : reduce, collect

```
somme = list.stream()
        .map(t -> t.getSolde())
        .filter(t -> t > 0)
        .reduce((t1, t2) -> t1+t2)
```

Quelques opérations terminales

reduce(BinaryOperator)
count(), min(Comparator), max(Comparator)
anyMatch(Predicate), allMatch(Predicate), noneMatch(Predicate)
findFirst(), findAny()
toArray()
forEach(Consumer)

La classe Collectors

Classe utilitaire fournissant les réductions usuelles (30+ méthodes)

count(), minBy(Comparator), maxBy(Comparator)

summing, averaging, summarizing

joining(), joining(CharSequence)

toList, toSet

mapping, groupingBy, partionningBy

Collectors en action

```
// Transformer un Stream en List
List<Person> liste1 = persons.stream().(…).toList();

// Transformer un Stream en List avec la méthode générique collect
List<Person> liste1 = persons.stream().(…).collect(Collectors.toList());

// Transformer un Stream en Set
Set<String> liste2 = persons.stream().(…).toSet();

// Transformer un Stream en Set avec la méthode générique collect
Set<String> liste2 = persons.stream().(…).collect(Collectors.toSet());
```

Collectors en action

```
// Concaténer les noms d'une liste de personnes
String names1 = persons.stream().map(p-> p.getName()).collect(Collectors.joining());

// Concaténer les noms séparés par une virgule d'une liste de personnes
String names2 = persons.stream().map(p -> p.getName()).collect(Collectors.joining(","));
```

Collectors en action

```
// Compter le nombre de personnes
int nbPersons = persons.stream().collect(Collectors.counting());

// Moyenne des ages des personnes
double moyenneAge = persons.stream().collect(Collectors.averagingDouble(p -> p.getAge()));

// Regroupement des personnes par age
Map<Integer, List<Person>> map = persons.stream().collect(Collectors.groupingBy(p -> p.getAge()));

// Regroupement des personnes par age en utilisant un Set
Map<Integer, Set<Person>> map = persons.stream().collect(Collectors.groupingBy(p-> p.getAge(),Collectors.toSet()));

// Répartir les données en 2 ensembles : true -> liste des personnes age > 20 et false -> le reste
Map<Boolean, List<Person>> map = persons.stream().collect(Collectors.partitionningBy(p -> p.getAge() > 20));
```

Un Stream Parallèle ?

Distribution de traitements
à travers plusieurs Threads
(par défaut 1 Thread / coeur).

```
popTot = villes.parallelStream()
                .map(t -> t.getPopulation())
                .filter(t -> t > 100000)
                .reduce((t1, t2) -> t1+t2)
```

```
popTot = villes.stream()
                .map(t -> t.getPopulation())
                .filter(t -> t > 100000)
                .parallel()
                .reduce((t1, t2) -> t1+t2)
```

Travaux Pratiques