

Programmation Orientée Objets avec Java

Prof. Yassin Rekik et Prof. Stéphane Malandain - 2022-2023

(Basé sur le cours de M. El Kharoubi et K. Heirich)



Chapitre 12

Les streams

Les streams

Introduction

- Les streams représentent un flux d'éléments.
- Ce flux supporte des opérations, séquentielles comme parallèles.
- Les streams sont particulièrement adaptés pour opérer sur des collections et constituent une approche de programmation déclarative.

Principe général

4

Les streams permettent d'enchaîner de nombreuses opérations de manière compacte et lisible

Par exemple, pour enchaîner des opérations dans une `Collection<Devices>`, nous pouvons écrire :

```
int sum = widgets.stream()  
    .filter(w -> w.getColor() == RED)  
    .mapToInt(w -> w.getWeight())  
    .sum();
```

Streams : composition

5

Un stream peut être divisé en trois grandes parties :

- L'instanciation (`Stream.of()`, ...)
- Les opérations intermédiaires (`filter()`, `map()`, ...)
- L'opération terminale (`count()`, `collect()`, ...)

Instanciación

Streams : Instanciation

7

- Avec la méthode 'of' :

```
// Stream
Stream<String> stream = Stream.of("a", "b", "c", "d");
// Operation
stream.forEach(System.out::println);
```

- Avec un StreamBuilder :

```
// Builder
Stream.Builder<String> builder = Stream.builder();

// Stream
Stream<String> stream = builder.add("Test").build();

// Operation
stream.forEach(System.out::println);
```

Streams : Instanciación

- A partir d'une collection :

```
// Collection
```

```
List<String> items = new ArrayList<String>();
```

```
items.add("one");
```

```
items.add("two");
```

```
items.add("three");
```

```
// Stream
```

```
Stream<String> stream = items.stream();
```

```
// Operation
```

```
stream.forEach(System.out::println);
```

Opérations intermédiaires

Opérations intermédiaires

10

- Il existe de très nombreuses opérations intermédiaires
- Nous allons voir le `filter()`, `map()`, `sorted()` qui sont des opérations de base
- Bien comprendre comment lire la signature d'une méthode permet ensuite de rapidement comprendre n'importe quelle autre dans la `javadoc`

Opérations intermédiaires

11

```
Stream<T> filter(Predicate<? super T> predicate)  
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

```
Stream<Integer> s = Stream.of(1,2,3,4,5);  
s.filter(i -> i % 2 == 0).forEach(System.out::println);  
// Affiche : 2, 4  
s.map(i -> i * 2).forEach(System.out::println);  
// Affiche : 2, 4, 6, 8, 10
```

Opérations intermédiaires

12

```
Stream<T> sorted()
```

Sorted() permet de trier un stream qui contient des objets implémentant l'interface Comparable

```
Stream<Integer> s = Stream.of(-9, -18, 0, 25, 4);
```

```
s.sorted().forEach(System.out::println);  
// Affiche -18, -9, 0, 4, 25
```

```
Stream<T> sorted(Comparator<? super T> comparator)
```

Sorted() accepte également un Comparator pour déterminer l'ordre du tri

```
Stream<Integer> s = Stream.of(-9, -18, 0, 25, 4);
```

```
s.sorted(Comparator.reverseOrder()).forEach(System.out::println);
```

Opérations intermédiaires

13

Il est possible de chaîner les opérations de comparaisons pour utiliser plusieurs critères.

```
Comparator.comparing(People::getName).thenComparing(People::getAge)
```

Il est également possible de définir vos propres comparateurs en implémentant une classe qui étend l'interface `Comparator`

```
// Comparateur en classe anonyme
new Comparator<People>() {
    @Override
    public int compare(People p1, People p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

Autres Opérations intermédiaires

14

- `limit()` : permet de limiter le nombre d'éléments évalués
- `skip()` : permet de ne pas évaluer un certain nombre d'éléments
- `distinct()` : permet de filtrer les éléments considérés comme égaux (avec `equals()`)

Il est possible d'obtenir des streams typés avec des méthodes telles que `mapToDouble()` ou `mapToInt()`.

Le stream standard est générique, alors que ces streams sont typés. Nous n'avons donc pas un `Stream<Double>` mais un `DoubleStream`.

Leur utilisation n'est pas obligatoire mais certaines opérations comme la moyenne peuvent être rendue moins verbeuse grâce à des méthodes de ces streams typés tel que `average()`.

Nous vous invitons à consulter la `javadoc` un maximum.

Autre opération intermédiaire

15

```
Stream.iterate(0, n -> n + 1)
    .filter(x -> x % 2 != 0) //odd
    .limit(10)
    .forEach(x -> System.out.println(x));
```

// suite de fibonacci

```
Stream.iterate(new int[]{0, 1}, n -> new int[]{n[1], n[0] + n[1]})
    .limit(20)
    .map(n -> n[0])
    .forEach(x -> System.out.println(x));
```

Opérations terminales

Opérations terminales: forEach

17

Le `forEach` applique un `Consumer` sur les éléments d'un stream, c'est une opération terminale.

```
s.sorted().forEach(System.out::println);
```

Opérations terminales: reduce 1/2

18

Nous avons déjà vu le `reduce`, qui permet d'obtenir un résultat en appliquant une opération répétée sur plusieurs éléments.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

int result = numbers
    .stream()
    .reduce(0, (subtotal, element) -> subtotal + element);

// 0 est notre élément neutre et valeur de départ
// subtotal est notre accumulateur
// element est l'élément courant lors du reduce
// Le résultat donne 21 (somme des éléments)
```

Opérations terminales: reduce 2/2

19

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);  
// .reduce(0, (subtotal, element) -> subtotal + element);  
  
int result = numbers.stream().reduce(0, Integer::sum);  
  
// Le résultat donne toujours 21
```

Opérations terminales: collect

20

L'opération `collect` nous permet de récupérer le résultat de nos opérations de différentes manières.

C'est la manière la plus simple est de renvoyer le résultat de nos opérations vers une collection donnée en utilisant la classe `Collectors` sans ajouter de traitement.

```
Stream<String> s = ...  
List<String> result = s.collect(Collectors.toList());  
Set<String> result = s.collect(Collectors.toSet());  
Collection<String> result = s.collect(Collectors.toCollection(TreeSet::new))
```

Opérations terminales: groupingBy

21

`Collectors.groupingBy()` nous permet d'agréger nos valeurs selon un critère donné pour notre résultat.

```
List<Student> students = Arrays.asList(  
    // Student -> subject, name, age  
    new Student("Math", "Michael", 18),  
    new Student("Math", "Joel", 24),  
    new Student("Programming", "Kevin", 24),  
    new Student("Math", "Quentin", 27)  
);  
  
Map<String, List<Student>> studentsBySubject = students  
    .stream()  
    .collect(  
        Collectors.groupingBy(Student::getSubject)  
    );  
studentsBySubject.get("Math").forEach(p -> System.out.println(p.getName()));  
// Michael, Joel, Quentin
```

Opérations terminales: groupingBy

22

`groupingBy()` accepte un second paramètre qui est un `Collector`. Ce dernier est alors appliqué sur toutes les données après l'agrégation.

```
Map<City, Set<String>> namesByCity =  
    people.stream().collect(  
        groupingBy(  
            // 1 : Grouper par ville  
            Person::getCity,  
            // 2 : Mapper les noms dans un set  
            // Collector mapping(Function, Collector)  
            mapping(Person::getLastName,  
                toSet())));
```

Opérations terminales: findFirst

23

`findFirst()` permet d'obtenir le premier élément du stream

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);

Optional<Integer> firstResult = list.stream()
    .filter(i -> i % 2 == 0)
    .findFirst();
// firstResult sera 2
```

Opérations terminales: findAny

24

`findAny()` permet d'en obtenir un arbitrairement

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);

Optional<Integer> anyResult = list.stream()
    .filter(i -> i % 2 == 0)
    .findAny();
// anyResult peut être 2 ou 4
```

Opérations terminales: collectingAndThen

25

`Collect(Collectors.collectingAndThen(...))` permet de collecter les valeurs une première fois avant d'appliquer une opération finale.

```
Collection<Person> townHabitants = ...  
townHabitants.stream().collect(Collectors.collectingAndThen(  
    // Map<Gender, Collection<Person>>  
    Collectors.groupingBy(Person::GetGender),  
    PersonUtility::ComputeMedianAgeForEachKey));
```

- `count()` : retourne le nombre total d'éléments.
- `anyMatch()` : permet de vérifier une condition via `Predicate` sur les éléments du stream. Il existe des variantes telles que `noneMatch()`.
- `Collect(Collectors.maxBy(...))` permet d'obtenir le maximum en fournissant un comparateur

Autre opération : concat

27

La méthode `concat` nous permet de concaténer les valeurs de deux streams. Elle retourne un stream contenant le résultat.

```
Stream<String> networkTeachers = Stream.of("Hoerdt");  
Stream<String> softwareTeachers = Stream.of("Malandain", "Rekik");  
  
Stream.concat(networkTeachers, softwareTeachers)  
    .forEach(System.out::println);  
// Hoerdt, Malandain, Rekik
```

Lazy evaluation

- Les streams sont lazy, c'est à dire que chaque opération intermédiaire retourne un nouveau stream
- Ces opérations intermédiaires (ces streams intermédiaires) ne sont pas évaluées avant que l'opération terminale ne soit invoquée
- Quand l'opération terminale est invoquée, les streams intermédiaires sont traversés et leurs fonctions associées sont exécutées l'une après l'autre. Quand une optimisation entre les opérations est possible, elle est faite.
- Exemple :

```
Stream<Integer> s = Stream.of(1,2,3,4,5);  
int result = s.map(i -> i * 2).map(i -> i + 2).reduce(0, Integer::sum);
```

- Cette exécution ne donne pas :

```
Stream<Integer> is = ...  
int result = 0;  
for(Integer i: is) {  
    result += i * 2;  
}  
for(Integer i: is) {  
    result += i + 2;  
}
```

- Mais donne en fait quelque chose comme :

```
Stream<Integer> is = ...  
int result = 0;  
for(Integer i: is) {  
    result += i * 2 + 2;  
}
```