Streams

L'API *streams* a été introduite avec Java 8 pour permettre la programmation fonctionnelle. Un *stream* (flux) est une représentation d'une séquence sur laquelle il est possible d'appliquer des opérations. Cette API a deux principales intérêts :

1. Elle permet d'effectuer les opérations sur une séquence sans utiliser de structure de boucle. Cela permet de réaliser des traitements complexes tout en maintenant une bonne lisibilité du code.
2. Les opérations sur les *streams* sont réalisées en flux (d'où leur nom) ce qui limite l'empreinte mémoire nécessaire. Il est même possible de réaliser très simplement des traitements en parallèle pour tirer partie des possibilités d'une processeur multi-cœurs ou d'une machine multi-processeurs.

Création d'un stream

Un *stream* est représenté par une instance de l'interface générique Stream. On peut créer un Stream en utilisant un objet de type builder

```
Stream<String> stream = Stream.<String>builder().add("Hello").add("World").build();
```

Il existe également des interfaces filles de Stream pour certains types primitifs : IntStream, LongStream et DoubleStream. On peut créer des *streams* de ces types soit à partir d'une liste de valeurs soit en donnant les limites d'un intervalle.

```
IntStream intStream = IntStream.of(1, 20, 30, 579);


IntStream rangeIntStream = IntStream.range(0, 1_000_000_000);
```

> Comme mentionné à la section précédente, un des intérêts des *streams* vient de leur nature même de flux. Ainsi dans l'exemple précédent, la création d'un *stream* à partir d'un intervalle ne crée pas une valeur pour chaque élément. Ainsi la création d'un *stream* sur un intervalle d'un milliard est instantanée et ne prend presque aucune place en mémoire.

Il est même possible de créer un *stream* « infini » dont les valeurs sont calculées par une lambda.

```
// Un stream commençant à la valeur 1 et qui est représenté par la suite n = n + 1
LongStream longStream = LongStream.iterate(1, n -> n + 1);
```

Il est également possible de créer un *stream* à partir d'un tableau grâce aux méthodes Arrays.stream :

```
int[] tableau = { 1, 2, 3, 4 };
IntStream tableauStream = Arrays.stream(tableau);
```

Les collections peuvent également être utilisées sous la forme d'un *stream* car l'interface Collection définit la méthode Collection.stream.

```
List<String> liste = new ArrayList<>();
liste.add("Hello");
liste.add("World");

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

Le contenu d'un fichier texte peut aussi être parcouru sous la forme d'un *stream* de chacune de ses lignes grâce à la méthode Files.lines :

```
Path fichier = Paths.get("fichier.txt");
Stream<String> linesStream = Files.lines(fichier);
```

Ainsi, toutes les opérations qui impliquent une séquence d'éléments peuvent être traitées sous la forme d'un *stream*.

Il est possible de réaliser un traitement sur chaque élément du *stream* grâce à la méthode Stream.forEach.

```
// Affiche les chiffres de 10 jusqu'à 0
IntStream.iterate(10, n -> n - 1).limit(11).forEach(System.out::println);
```

Un *stream* est également utilisé pour produire un résultat unique ou une collection. Dans le premier cas, on dit que l'on réduit, tandis que dans le second cas, on dit que l'on collecte.

La réduction

La réduction consiste à obtenir un résultat unique à partir d'un *stream*. On peut par exemple compter le nombre d'éléments. Si le *stream* est composé de nombres, on peut réaliser une réduction mathématique en calculant la somme, la moyenne ou en demandant la valeur minimale ou maximale...

```
long resultat = LongStream.range(0, 50).sum();
System.out.println(resultat);

OptionalDouble moyenne = LongStream.range(0, 50).average();
if (moyenne.isPresent()) {
  System.out.println(moyenne.getAsDouble());
}
```

L'API *streams* introduit la notion de *Optional*. Certaines opérations de réduction peuvent ne pas être possibles. Par exemple, le calcul de la moyenne n'est pas possible si le *stream* ne contient aucun élément. La méthode average qui permet de calculer la moyenne d'un *stream* numérique retourne donc un OptionalDouble qui permet de représenter soit le résultat, soit le fait qu'il n'y a pas de résultat. On peut appeler la méthode OptionalDouble.isPresent pour s'assurer qu'il existe un résultat pour cette réduction.

Pour les streams de tout type, il est possible de réaliser une réduction à partir d'une lambda grâce à la méthode Stream.reduce.

```
List<String> liste = Arrays.asList("une chaine", "une autre chaine", "encore une chaine");
Optional<String> chaineLaPlusLongue = liste.stream().reduce((s1, s2) -> s1.length() >
s2.length() ? s1 : s2);

System.out.println(chaineLaPlusLongue.get()); // "encore une chaine"
```

La collecte

La collecte permet de créer un nouvelle collection à partir d'un stream. Pour cela, il faut fournir une implémentation de l'interface Collector. Cette interface est assez complexe, heureusement la classe outil Collectors fournit des méthodes pour générer une instance de Collector. Pour réaliser la collecte, il faut appeler la méthode Stream.collect.

On peut ainsi collecter les éléments d'un stream sous la forme d'une List, d'un Set ou de tout type de Collection.

```
List<String> liste = Arrays.asList("une chaine", "une autre chaine", "encore une chaine");
List<String> autreListe = liste.stream().collect(Collectors.toList());
```

L'exemple précédent peut sembler trivial puisqu'au final, ce code crée un copie de la liste d'origine. Son intérêt deviendra évident lorsque nous appliquerons des opérations de filtre ou de mapping sur un *stream*.

Un Collector peut également réaliser un opération de regroupement pour créer des Map. Si on dispose de la classe *Voiture* :

```
package com.cgi.udev;

public class Voiture {

  private String marque;

  public Voiture(String marque) {
    this.marque = marque;
  }

  public String getMarque() {
    return marque;
  }
}
```

Alors il devient facile de grouper des instances d'une liste de *Voiture* selon leur marque.

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                    new Voiture("renault"),
                                    new Voiture("audi"),
                                    new Voiture("citroen"));

Map<String, List<Voiture>> map =
liste.stream().collect(Collectors.groupingBy(Voiture::getMarque));

System.out.println(map.get("citroen").size()); // 2
System.out.println(map.get("renault").size()); // 1
System.out.println(map.get("audi").size());    // 1
```

On peut également créer une chaîne de caractères en joignant les éléments d'un *stream* :

```
List<String> list = Arrays.asList("un", "deux", "trois", "quatre", "cinq");
String resultat = list.stream().collect(Collectors.joining(", "));

System.out.println(resultat); // "un, deux, trois, quatre, cinq"
```

Le filtrage

Une opération courante sur un *stream* consiste à appliquer un filtre pour éliminer une partie de ses éléments. Pour, cela on peut utiliser la méthode Stream.filter.

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                    new Voiture("audi"),
                                    new Voiture("citroen"));

// on construit la liste des voitures qui ne sont pas de marque "citroen"
List<Voiture> sansCitroen = liste.stream()
                                 .filter(v -> !v.getMarque().equals("citroen"))
                                 .collect(Collectors.toList());

System.out.println(sansCitroen.size()); // 1
```

```
// On affiche les 500 premiers nombres qui ne sont pas divisibles par 7
IntStream.iterate(1, n -> n + 1)
         .filter(n -> n % 7 != 0)
         .limit(500)
         .forEach(System.out::println);
```

La méthode Stream.filter peut accepter une lambda qui reçoit en paramètre un élément du *stream* et qui retourne un **boolean** (**true** signifie que l'élément doit être conservé dans le *stream*). On peut bien évidemment chaîner les appels à la méthode Stream.filter :

```
// On affiche les 500 premiers nombres qui ne sont pas divisibles par 7
// et qui sont impairs
IntStream.iterate(1, n -> n + 1)
         .filter(n -> n % 7 != 0)
         .filter(n -> n % 2 != 0)
         .limit(500)
```

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

Le mapping

Le mapping est une opération qui permet de transformer la nature du *stream* afin de passer d'un type à un autre.

Par exemple, si nous voulons récupérer l'ensemble des marques distinctes d'une liste de *Voiture*, nous pouvons utiliser un mapping pour passer d'un *stream* de *Voiture* à un *stream* de String (représentant les marques des voitures).

```
List<Voiture> liste = Arrays.asList(new Voiture("citroen"),
                                    new Voiture("audi"),
                                    new Voiture("renault"),
                                    new Voiture("volkswagen"),
                                    new Voiture("citroen"));

// mapping du stream de voiture en stream de String
Set<String> marques = liste.stream()
                           .map(Voiture::getMarque)
                           .collect(Collectors.toSet());

System.out.println(marques); // ["audi", "citroen", "renault", "volkswagen"]
```

Pour réaliser un mapping vers un type primitif, il faut utiliser les méthodes Stream.mapToInt, Stream.mapToLong ou Stream.mapToDouble. On peut également utiliser ces méthodes pour convertir un *stream* contenant un type primitif vers un *stream* contenant un autre type primitif.

```
// Affichage de la racine carré des 100 premiers entiers
IntStream.range(1, 101)
         .mapToDouble(Math::sqrt)
         .forEach(System.out::println);
```

Pour la méthode Stream.map, le type de retour de la lambda ou de la référence de méthode indique le nouveau type du *stream*.

Le parallélisme

Afin de tirer profit des processeurs multi-cœurs et des machines multi-processeurs, les opérations sur les *streams* peuvent être exécutées en parallèle. À partir d'une Collection, il suffit d'appeler la méthode Collection.parallelStream ou à partir d'un Stream, il suffit d'appeler la méthode BaseStream.parallel.

Un *stream* en parallèle découpe le flux pour assigner l'exécution à différents processeurs et recombine ensuite le résultat à la fin. Cela signifie que les traitements sur le *stream* ne doivent pas être dépendant de l'ordre d'exécution.

Par exemple, si vous utilisez un *stream* parallèle pour afficher les 100 premiers entiers, vous constaterez que la sortie du programme est imprédictible.

```
// affiche les 100 premiers entiers sur la console en utilisant un stream parallèle.
// Ceci n'est pas une bonne idée car l'opération d'affichage implique
// que le stream est parcouru séquentiellement. Or un stream parallèle
// est réparti sur plusieurs processeurs et donc l'ordre d'exécution
// n'est pas prédictible
IntStream.range(1, 101).parallel().forEach(System.out::println);
```

Par contre, les streams parallèles peuvent être utiles pour des réductions de type somme puisque le calcul peut être réparti en sommes intermédiaires avant de réaliser la somme totale.

Created 2025-02-10 23:26:15 UTC by Nicolas
Updated 2025-02-10 23:27:25 UTC by Nicolas