Programmation par flots

Michel Schinz

2020-04-20

1 Introduction

En informatique, le terme **flot** (*stream*) désigne généralement une séquence de valeurs auxquelles on accède l'une après l'autre, de la première à la dernière. En Java, cette notion apparaît dans deux contextes :

1. les entrées/sorties, puisque les données lues ou écrites par le programme sont modélisées par des flots d'octets ou des caractères,

2. ce que l'on nomme parfois la programmation par flots, sujet de cette leçon.

La **programmation par flots** (*stream programming*) consiste à exprimer un calcul sur des données au moyen d'un enchaînement d'opérations simples appliquées au flot de ces données.

Comme nous le verrons, les lambdas permettent d'exprimer de manière concise les opérations à effectuer sur les données du flot. Pour cette raison, plusieurs classes et méthodes ont été introduites en Java 8 — en même temps que les lambdas — pour faciliter la programmation par flots.

2 Exemple : conversion de température

Pour illustrer l'intérêt de la programmation par flots, admettons que l'on désire convertir un fichier contenant des températures en degrés Fahrenheit en un fichier contenant ces mêmes températures en degrés Celsius, en ignorant les lignes vides. Par exemple, si le fichier d'entrée contient les trois lignes suivantes (la seconde étant vide) :

```
1  0
2
3  100
```

le fichier de sortie doit contenir les deux lignes :

```
1  -17.777
2  37.777
```

Cette conversion peut s'exprimer par la transformation de flots suivante :

1. obtenir le flot des lignes du fichier d'entrée,
2. filtrer ce flot pour ne garder que les lignes non vides,
3. convertir le flot de lignes — des chaînes de caractères — en un flot de températures en °F — des nombres réels,
4. convertir le flot des températures en °F en flot des températures en °C, au moyen de la formule de conversion [$°C = (°F - 32) \times 5/9$],
5. écrire chaque valeur du flot dans le fichier de sortie, une par ligne.

Cette transformation de flot peut également se visualiser facilement, comme l'illustre la figure 1.

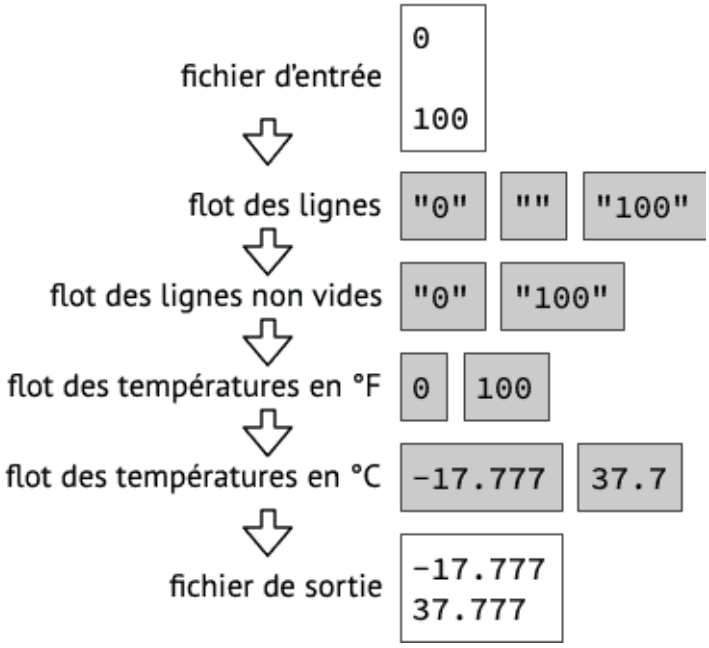

Fig. 1 : Flot de transformation de températures

En utilisant les classes et méthodes que nous allons examiner, cette conversion peut se traduire très directement en un programme Java :

```
try (BufferedReader r =
       new BufferedReader(new FileReader("f.txt"));
     PrintWriter w =
```

```
       new PrintWriter(new FileWriter("c.txt"))) {
  r.lines()                            // lignes
    .filter(l -> !l.isEmpty())         // lignes non vides
    .map(Double::parseDouble)          // températures °F
    .map(f -> (f - 32d) * (5d / 9d)) // températures °C
    .forEachOrdered(w::println);
}
```

3 Flots Java

Le paquetage `java.util.stream` — nouveauté de la version 8 de Java — définit plusieurs classes et interfaces permettant de faire de la programmation par flots.

L'interface `Stream` de ce paquetage représente un flot de valeurs d'un type donné. Elle est naturellement générique, son paramètre de type représentant le type des valeurs du flot :

```
public interface Stream<T> {
  // … méthodes
}
```

Par exemple, le flot des lignes d'un fichier a le type `Stream<String>`, celui des températures — en °F ou °C — le type `Stream<Double>`, etc.

Pour des raisons de performances, il existe également des interfaces représentant des flots de trois types de nombres, à savoir :

1. `DoubleStream` pour les flots de nombres à virgule flottante `double`,
2. `LongStream` pour les flots d'entiers `long`, et
3. `IntStream` pour les flots d'entiers `int`.

Ainsi, un flot d'entiers `int` peut être représenté de deux manières :

1. par un flot général de type `Stream<Integer>`, dans lequel chaque entier est représenté par un objet de type `Integer`,
2. par un flot spécialisé de type `IntStream`, dans lequel chaque entier est représenté par une valeur de type `int`.

La seconde représentation est plus efficace car elle évite la création d'objets, et doit autant que possible être préférée lorsque les performances sont importantes.

Les méthodes qui travaillent sur les flots — celles de l'interface `Stream` ou autres — appartiennent toutes à exactement une des trois catégories suivantes :

1. les **méthodes sources**, qui produisent un flot de valeurs à partir d'une source qui peut p.ex. être une collection, un fichier, etc.

2. les **méthodes intermédiaires**, qui transforment les valeurs du flot et produisent un nouveau flot,

3. les **méthodes terminales**, qui consomment les valeurs du flot, p.ex. en les écrivant dans un fichier, en les réduisant à une valeur unique, etc.

Ces trois types de méthodes sont utilisés pour construire ce que l'on nomme parfois des **pipelines**. Un pipeline est formé de :

- *une* méthode source, qui produit un flot de valeurs,
- *zero ou plusieurs* méthodes intermédiaires, qui transforment les valeurs,
- *une* méthode terminale, qui consomme les valeurs.

La transformation de °F en °C présentée précédemment est un tel pipeline, dans lequel la méthode `lines` est la méthode source, les méthodes `filter` et `map` sont les méthodes intermédiaires, et la méthode `forEachOrdered` la méthode terminale.

Il est important de savoir que les flots ne sont pas immuables, dans le sens où lorsque les éléments d'un flot sont consommés par une méthode terminale, ils disparaissent et le flot est ensuite vide. En cela, les flots sont similaires aux itérateurs, qui sont également modifiés lors de leur parcours.

Les principales méthodes sources, intermédiaires et terminales fournies par la bibliothèque Java sont présentées ci-dessous. Cette liste n'est toutefois pas exhaustive, et les types ont souvent été simplifiés pour des raisons pédagogiques.

3.1 Méthodes sources

Plusieurs méthodes sources sont des méthodes statiques de l'interface `Stream` ou de l'une des spécialisations susmentionnées. Il existe néanmoins des méthodes sources dans d'autres parties de la bibliothèque Java, qui permettent de faire le pont entre cette partie de la bibliothèque et les flots. La méthode `lines` utilisée plus haut en est un exemple, et fait le pont entre un lecteur et un flot.

3.1.1 Méthodes de `Stream`

L'interface `Stream` fournit, entre autres, les méthodes sources suivantes, qui sont bien entendu toutes statiques :

- `Stream<T> empty()` : retourne un flot vide,
- `Stream<T> of(T... vs)` : retourne le flot des valeurs passées en argument,

- `Stream<T> iterate(T i, UnaryOperator<T> f)` : retourne le flot infini des applications successives de l'opérateur `f` à la valeur initiale `i` ; la première valeur du flot est donc `i`, la seconde `f(i)`, la troisième `f(f(i))` et ainsi de suite.

Les différentes spécialisations de `Stream` offrent des méthodes identiques, ainsi que d'autres méthodes sources qui n'ont de sens que pour leur type d'éléments. Ainsi, `IntStream` et `LongStream` fournissent des méthodes permettant de construire un flot contenant tous les entiers d'un intervalle donné. Pour `IntStream`, ces méthodes sont :

- `IntStream range(int f, int t)` : retourne le flot des entiers compris entre `f` (inclus) et `t` (exclus), en ordre croissant,
- `IntStream rangeClosed(int f, int t)` : retourne le flot des entiers compris entre `f` (inclus) et `t` (inclus), en ordre croissant.

L'utilisation de certaines de ces méthodes est illustrée par les exemples ci-dessous :

```
Stream<Character> vowels = // voyelles latines
  Stream.of('a', 'e', 'i', 'o', 'u', 'y');
Stream<Integer> posInts =  // entiers > 0 (non spécialisé)
  Stream.iterate(1, i -> i + 1);
IntStream posIntsI =       // entiers > 0 (spécialisé)
  IntStream.iterate(1, i -> i + 1);
```

3.1.2 Bâtisseur

L'interface `Stream` offre une méthode statique nommée `builder` retournant un nouveau bâtisseur de flot de type `Stream.Builder`. Celui-ci offre deux méthodes :

- `Stream.Builder<T> add(T t)` : ajoute l'élément `t` au flot en cours de construction et retourne le bâtisseur,
- `Stream<T> build()` : construit et retourne le flot des éléments ajoutés jusqu'à présent, rendant du même coup le bâtisseur inutilisable.

Un tel bâtisseur peut p.ex. s'utiliser pour construire le flot des voyelles de l'alphabet latin :

```
Stream.Builder<Character> vowelsB = Stream.builder()
  .add('a').add('e').add('i').add('o').add('u').add('y');
Stream<Character> vowels = vowelsB.build();
```

3.1.3 Flot des éléments d'une collection

La méthode `stream` de l'interface `Collection` retourne un flot avec les éléments de la collection à laquelle on l'applique. Elle sert donc de pont entre le monde des collections et celui des flots.

Par exemple, le flot des voyelles de l'alphabet latin peut également s'obtenir ainsi :

```
List<Character> vowelsL =
  List.of('a', 'e', 'i', 'o', 'u', 'y');
Stream<Character> vowels =
  vowelsL.stream();
```

3.1.4 Flot des lignes d'un lecteur

La méthode `lines` de `BufferedReader` retourne le flot des lignes du lecteur auquel on l'applique. Elle sert donc de pont entre le monde des lecteurs et celui des flots.

3.2 Méthodes intermédiaires

Les méthodes intermédiaires sont toutes des méthodes par défaut de l'interface `Stream` ou de l'une de ses spécialisations. Parmi les méthodes intermédiaires fournies figurent, entre autres :

- `Stream<T> limit(long l)` : retourne un flot de longueur maximale égale à `l` et contenant les mêmes élément que le flot auquel on l'applique,
- `Stream<T> skip(long n)` : retourne le flot des éléments restants du flot auquel on l'applique après avoir ignoré les `n` premiers,
- `Stream<T> sorted()` : retourne un flot contenant les mêmes éléments que celui auquel on l'applique, mais triés par ordre naturel croissant,
- `Stream<T> sorted(Comparator<T> c)` : retourne un flot contenant les mêmes éléments que celui auquel on l'applique, mais triés en ordre croissant au moyen du comparateur `c`,
- `Stream<T> filter(Predicate<T> p)` : retourne le flot de toutes les valeurs du flot auquel on l'applique qui satisfont le prédicat `p`,
- `Stream<R> map(Function<T,R> f)` : retourne le flot obtenu en appliquant la fonction `f` aux éléments du flot auquel on l'applique ; des versions spécialisées de cette méthode existent (`mapToInt`, etc.) et permettent d'obtenir des flots spécialisés à partir de flots généraux.

La méthode `map` est particulièrement importante, et très fréquemment utilisée, car elle permet de transformer les valeurs d'un flot pour en obtenir un nouveau. Par exemple, pour transformer un flot de chaînes de caractères en le flot de leurs longueurs, on peut écrire :

```
Stream<String> strings =
  Stream.of(”un”, ”deux”, ”trois”, ”quatre”);
Stream<Integer> stringLengths =
  strings.map(String::length);  // 2, 4, 5, 6
```

Ce flot étant un flot d'entiers `int`, il est aussi possible d'en obtenir la version spécialisée au moyen de la méthode `mapToInt` :

```
Stream<String> strings =
  Stream.of(”un”, ”deux”, ”trois”, ”quatre”);
IntStream stringLengthsS =
  strings.mapToInt(String::length); // 2, 4, 5, 6
```

3.3 Méthodes terminales

Tout comme les méthodes intermédiaires, les méthodes terminales sont toutes des méthodes par défaut de l'interface `Stream` ou de l'une de ses spécialisations. La différence entre ces deux catégories de méthodes est que les méthodes intermédiaires retournent un flot, alors que les méthodes terminales retournent autre chose. (En termes de patrons de conception, les méthodes intermédiaires correspondent à des décorateurs, les méthodes source et terminales à des adaptateurs.)

3.3.1 Méthodes statistiques

Les méthodes terminales suivantes permettent d'obtenir des statistiques des éléments du flot auquel on les applique :

- `long count()` : retourne le nombre d'éléments du flot auquel on l'applique,
- `Optional<T> max(Comparator<T> c)` : retourne le plus grand élément du flot auquel on l'applique, selon le comparateur `c`, ou la valeur optionnelle vide si le flot est vide,
- `Optional<T> min(Comparator<T> c)` : retourne le plus petit élément du flot auquel on l'applique, selon le comparateur `c`, ou la valeur optionnelle vide si le flot est vide.

Les flots spécialisés pour les types numériques (`IntStream` et autres) fournissent d'autres méthodes statistiques, comme p.ex. `average` qui calcule la moyenne des éléments du flot auquel on l'applique.

3.3.2 Méthodes logiques

Les méthodes terminales suivantes permettent de déterminer si les éléments du flot auquel on les applique satisfont un prédicat :

- `boolean allMatch(Predicate<T> p)` : retourne vrai ssi tous les éléments du flot auquel on l'applique satisfont le prédicat `p` ; correspond à l'opérateur logique $\forall$,
- `boolean anyMatch(Predicate<T> p)` : retourne vrai ssi au moins un élément du flot auquel on l'applique satisfait le prédicat `p` ; correspond à l'opérateur logique $\exists$,
- `boolean noneMatch(Predicate<T> p)` : retourne vrai ssi aucun élément du flot auquel on l'applique ne satisfait le prédicat `p` ; correspond à l'opérateur logique $\nexists$.

3.3.3 Méthodes de consommation

La méthode terminale `forEach`, dont il existe deux variantes, permet de fournir tous les éléments du flot auquel on l'applique à un consommateur :

- `void forEachOrdered(Consumer<T> c)` : applique le consommateur `c` à tous les éléments du flot auquel on l'applique, dans l'ordre,
- `void forEach(Consumer<T> c)` : applique le consommateur `c` à tous les éléments du flot auquel on l'applique, dans un ordre quelconque.

3.3.4 Méthodes de réduction

La méthode terminale `reduce`, dont il existe plusieurs variantes, permet de réduire les éléments du flot auquel on l'applique à une seule valeur :

- `Optional<T> reduce(BinaryOperator<T> o)` : retourne la valeur résultant de l'application successive de l'opérateur binaire `o` à tous les éléments du flot auquel on l'applique ; si celui-ci est vide, retourne la valeur optionnelle vide,
- `T reduce(T z, BinaryOperator<T> o)` : retourne la valeur résultant de l'application successive de l'opérateur binaire `o` à la valeur initiale `z` puis à tous les éléments du flot auquel on l'applique ; si celui-ci est vide, retourne simplement `z`.

L'ordre dans lequel l'opérateur binaire est appliqué aux différents éléments n'est pas spécifié, et cet opérateur doit donc être associatif.

Par exemple, la fonction factorielle peut se définir au moyen d'une réduction consistant à multiplier tous les entiers compris entre 2 et l'argument, avec 1 comme valeur initiale :

```
int fact(int x) {
  return IntStream.rangeClosed(2, x)
    .reduce(1, (a, b) -> a * b);
}
```

3.3.5 Méthode de collecte

La méthode `collect` permet de collecter de manière générale les éléments d'un flot. Elle prend en argument un **collecteur** (*collector*), qui est un objet de type `Collector`.

Dans la plupart des cas, le collecteur est un collecteur prédéfini, obtenu au moyen d'une des méthodes de la classe `Collectors`, décrite ci-après, raison pour laquelle le type `Collector` n'est pas présenté en détail ici.

4 Collecteurs prédéfinis

La classe `Collectors` du paquetage `java.util.stream`, non instanciable, fournit des méthodes statiques permettant d'obtenir des collecteurs à passer à la méthode `collect` des flots.

Dans ce qui suit, le type de retour des différentes méthodes n'est jamais montré car il a tendance à être compliqué et sa compréhension n'est pas nécessaire à l'utilisation de ces collecteurs. En effet, ceux-ci sont toujours passés directement à la méthode `collect` de `Stream`. L'exemple suivant, qui trie une liste par l'intermédiaire d'un flot afin de ne pas modifier la liste originale, l'illustre :

```
List<String> l = List.of("un", "deux", "trois");
List<String> sortedL = l.stream()
  .sorted()
  .collect(Collectors.toList());
```

Seul un sous-ensemble, jugé particulièrement utile, des collecteurs existants est présenté ci-après.

4.1 Collecteurs de collections

Les collecteurs retournés par les méthodes suivantes placent les éléments du flot dans une collection :

- `toList()`, retourne un collecteur plaçant les éléments collectés dans une liste,
- `toSet()`, retourne un collecteur plaçant les éléments collectés dans un ensemble,

- `toMap(Function<T,K> k, Function<T,V> v)`, retourne un collecteur plaçant les éléments collectés dans une table associative, en utilisant la fonction `k` pour obtenir la clef correspondant à un élément, et la fonction `v` pour obtenir la valeur.

4.2 Collecteurs de chaînes

Les collecteurs retournés par les méthodes suivantes sont applicables uniquement aux flots de chaînes de caractères — pour être précis, aux flots de séquences de caractères `CharSequence`, interface que `String` implémente. Ils collectent les chaînes du flot en une chaîne finale :

- `joining()` : retourne un collecteur concaténant les chaînes collectées,
- `joining(String d)` : retourne un collecteur concaténant les chaînes collectées, en plaçant le délimiteur `d` entre chaque paire de chaînes,
- `joining(String d, String p, String s)` : retourne un collecteur concaténant les chaînes collectées, en plaçant le délimiteur `d` entre chaque paire de chaînes, le préfixe `p` au début et le suffixe `s` à la fin.

Par exemple, pour obtenir une chaîne de caractères composée de la concaténation des chaînes d'un flot, séparées par une virgule, le tout entouré d'accolades, on peut écrire :

```
String nums = Stream.of(”un”, ”deux”, ”trois”)
  .collect(Collectors.joining(”, ”, ”{ ”, ” }”));
System.out.println(nums);       // { un, deux, trois }
```

5 Références

- la documentation de l'API Java, en particulier les classes et interfaces suivantes :
  - le paquetage `java.util.stream`, en particulier sa description,
  - l'interface `java.util.stream.Stream` et ses spécialisations :
    - l'interface `java.util.stream.IntStream`,
    - l'interface `java.util.stream.LongStream`,
    - l'interface `java.util.stream.DoubleStream`,
  - la classe `java.util.stream.Collectors`.