

L3. Programmation orientée objet. Cours 8

Marie-Pierre Béal (Cours de Rémi Forax)

Stream. Optional. Comparateurs.



Stream

Stream

Opérations ensemblistes sur des collections

Exemple, avec un record et une liste

```
public record Person(String name, int age) {}  
var persons = List.of(new Person("Ana", 23), ...);
```

On veut répondre à une question comme

- Quel est la plus jeune personne dont le nom commence par "A" ?
 - Un peu comme en SQL

Avant Java 8

Quelle est la plus jeune personne dont le nom commence par "A" ?

```
Person min = null;
for(var person: persons) { // utilise un itérateur
    if (person.name().startsWith("A")) {
        if (min == null || person.age() < min.age()) {
            min = person;
        }
    }
}
return min;
```

En termes de maintenance, le code est assez loin de la question car on explique comment on fait pour obtenir ce que l'on veut et pas ce que l'on veut.

Avec l'API des Stream

Quelle est la plus jeune personne dont le nom commence par "A" ?

```
Optional<Person> result =  
    // Optional veut dire que la Person  
    // peut exister ou non  
persons.stream() // on utilise un Stream  
    .filter(p -> p.name().startsWith("A"))  
    .min(Comparator.comparingInt(Person::age));
```

L'API des Stream est **déclarative**, on dit le résultat que l'on veut, pas comment on l'obtient

- Plus facile pour la maintenance du code

Plusieurs APIs

Les streams utilisent plusieurs API

- L'API des Stream (`java.util.stream`)
 - Opérations ensemblistes
- L'API des Collector (`java.util.stream`)
 - Pour mettre le résultat dans une collection
- L'API `java.util.Optional`
 - Abstraction de null + véritable API
- L'API `java.util.Comparator`
 - Créer des comparateurs et les composer

L'API des Stream

L'API des Stream

Opérations sur les Stream

Trois étapes

- Création d'un Stream
- Transformations successives du Stream (opérations intermédiaires)
- Demander le résultat (opération finale)

Les opérations sur les Stream sont paresseuses

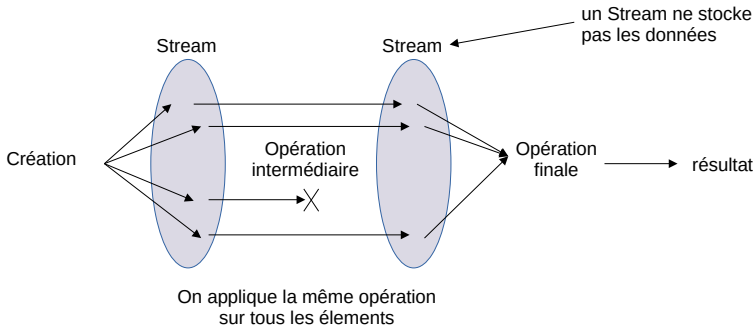
On ne fait pas le calcul si cela n'est pas nécessaire

- par exemple : si on n'appelle pas d'opération finale, aucun calcul n'est fait

Pipeline

Un Stream est organisé comme un pipeline

Les données sont poussées (*push*) dans le pipeline



Création d'un Stream

Création d'un Stream

Créer un Stream

A partir d'une collection

```
collection.stream()
```

A partir de valeurs

```
Stream.of(1, 2, 3, ...)
```

A partir d'un tableau

```
Arrays.stream(tableau)  
Arrays.stream(tableau, start, end)
```

Stream de type primitif

Pour éviter le boxing, il existe trois Stream spécialisés pour les types primitifs

- `IntStream`, `LongStream`, `DoubleStream`

Par exemple `IntStream`

- `IntStream.range(start, end)`
 - renvoie un stream contenant tous les entiers de `start` à `end` (`end` non compris)

Stream sur des ressources système

Les Streams sur des ressources système doivent être fermés avec `close()`

- On utilise le try-with-resources
- `Files.lines(path)` renvoie un stream contenant les lignes d'un fichier

```
try(var stream = Files.lines(path)) {  
    stream.    ...  
}
```

- `Files.list(path)` renvoie un stream contenant les noms des fichiers d'un répertoire

```
try(var stream = Files.list(path)) {  
    stream. ...  
}
```

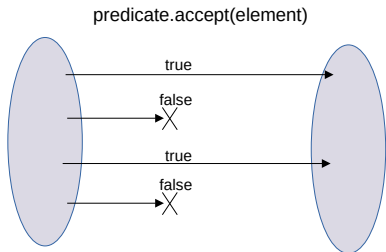
Opérations intermédiaires

Opérations intermédiaires

filter(Predicate<E>)

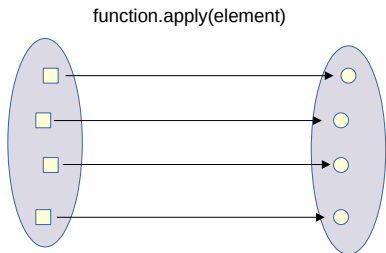
On appelle un prédicat ($E \rightarrow \text{boolean}$) et on ne propage pas l'objet si le prédicat renvoie false

```
stream.filter(person -> person.age() >= 18)
```



map(Function<E,R>)

Transforme chaque élément en appelant la fonction
`stream.map(Person::name)`

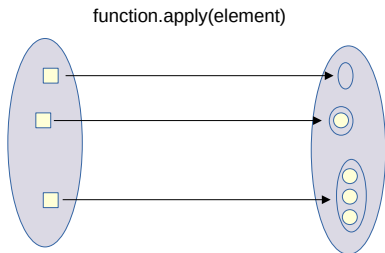


flatMap(Function<E, Stream<R>>)

Transforme chaque élément en appelant la fonction

```
stream.flatMap(p ->
```

```
    Stream.of(p, new Person(p.name(), p.age() + 1)))
```



Comme la fonction renvoie un Stream,
elle renvoie de 0 à n éléments

Exemple de flatMap

```
public record Person(String name, int age) { }  
...  
var list = Stream.of(new Person("toto", 30), new Person("titi", 40))  
    .flatMap(p -> Stream.of(p, new Person(p.name(), p.age() + 1)))  
    .toList();  
// [Person[name=toto, age=30], Person[name=toto, age=31],  
// Person[name=titi, age=40], Person[name=titi, age=41]]
```

Autres opérations intermédiaires

- `.limit(value)`
 - Filtre qui ne laisse passer que value éléments
- `.skip(value)`
 - Saute les value premiers éléments
- `.distinct()`
 - Filtre qui ne laisse pas passer la même valeur deux fois
- `.sorted(comparateur)`
 - Trie les éléments et les propage triés

Les opérations `distinct()` et `sorted()` coûtent cher : `distinct()` demande de stocker tous les éléments dans un Set et `sorted()` de stocker tous les éléments dans un tableau avant de trier. Donc ce sont des opérations où l'on perd le streaming car on doit utiliser de la mémoire intermédiaire.

map/flatMap et primitif

Les opérations intermédiaires map et flatMap ont des versions particulières pour les int, long et double (pour éviter le boxing)

Par exemple

```
stream.mapToInt(Person::age) // renvoie un IntStream
```

alors que

```
stream.map(Person::age) // renvoie un Stream<Integer>
```

Opérations finales

Opérations finales

`toList()` / `toArray()` / `findFirst()`

Pour sortir les éléments du stream

- `.toList()` pour stocker les éléments dans une liste non modifiable
 - `stream.toList()`
- `.toArray(IntFunction<E[]>)` pour stocker les éléments dans un tableau
 - `stream.toArray(Person[]::new)`
- `.findFirst()` pour renvoyer le premier élément en tant que `Optional<E>`
 - `stream.findFirst()`

allMatch, anyMatch, noneMatch

Savoir si tous (all), au moins un (any) ou aucun (none) éléments sont vrais pour un prédicat

- `.allMatch(Predicate<E>)`
 - `stream.allMatch(person -> person.age() >= 18)`
- `.anyMatch(Predicate<E>)`
 - `stream.anyMatch(person -> person.age() >= 18)`
- `.noneMatch(Predicate<E>)`
 - `stream.noneMatch(person -> person.age() >= 18)`

reduce() / count()

Agréger les valeurs en une seule

- `.reduce(initial, combiner)` pour combiner les éléments deux à deux

```
record Stat(long sum, long count) {  
    public Stat merge(Stat stat) {  
        return new Stat(sum + stat.sum, count + stat.count);  
    }  
    public double average() {  
        return sum / (double) count;  
    }  
}
```

```
var stream = Stream.of(new Person("toto", 30), new Person("titi", 40));  
var average = stream.map(p -> new Stat(p.age(), 1))  
                    .reduce(new Stat(0, 0), Stat::merge)  
                    .average();    // 35.0
```

- `.count()` pour compter le nombre d'éléments

```
var count = stream.count();
```

min/max(Comparator<E>)

Calcule l'élément minimum (resp. maximum) en fonction d'un comparateur

- `.min(comparator)` calcule l'élément minimum modifiable

```
stream.min((p1, p2) -> Integer.compare(p1.age(), p2.age()))
```

- `.max(comparator)` calcule l'élément maximum

```
stream.max(Comparator.comparingInt(Person::age))}
```

IntStream, LongStream et DoubleStream

Les streams de type primitif ont des opérations supplémentaires car les éléments sont des types numériques

`.min()`, `.max()`, `.sum()`, `.average()`

- Calcule le minimum, maximum, la somme et la moyenne

`.boxed()`

- Opération intermédiaire qui "box" tous les éléments

forEach(Consumer<E>)

Appelle le consommateur avec chaque élément
L'ordre est l'ordre de la collection (s'il existe)

```
stream.forEach(I0::println);
```

Attention, ne pas créer un stream juste pour appeler forEach

```
collection.stream().forEach(...);
```

n'est pas nécessaire car il y a déjà une méthode forEach(...) sur les collections

```
collection.forEach(...);
```

API des Collector

API des Collector

La méthode `collect(Collector)`

Un collecteur est un objet qui sait créer un objet mutable (souvent une collection), ajouter les éléments du `Stream` et optionnellement créer une version non mutable de l'objet.

- Par exemple, pour une `List`, les éléments seront d'abord stockés dans une `ArrayList` et optionnellement `List.copyOf` est appelée.
- Autre exemple, les chaînes de caractères sont jointes dans un `StringMerger` qui est transformé en `String` à la fin.

java.util.stream.Collectors

La classe Collectors (avec un 's') contient des méthodes statiques qui renvoient des Collector prédéfinis.

- joining()
- toList(), toUnmodifiableList()
- toMap(), toUnmodifiableMap()
- groupingBy()

joining(delimiteur, prefix, suffix)

Permet de joindre les chaînes de caractères d'un stream avec un délimiteur (et optionnellement un préfixe et un suffixe)

```
stream.map(Person::name)
      .collect(Collectors.joining(", "));
```

Ne marche que sur les Stream de sous-types de CharSequence (interface commune de String et StringBuilder).

`toList()`, `toUnmodifiableList()`

`Collectors.toList()` stocke les éléments dans une liste mutable.

```
stream.collect(Collectors.toList())
```

`Collectors.toUnmodifiableList()` stocke les éléments dans une liste non modifiable.

```
stream.collect(Collectors.toUnmodifiableList())
```

`toUnmodifiableList()` est presque équivalent à `stream.toList()`
`toUnmodifiableList()` n'accepte pas les valeurs null contrairement à `stream.toList()`

toMap(), toUnmodifiableMap()

Collectors.**toMap**(fonctionCle, fonctionValeur) stocke les éléments dans une Map en appelant les fonctions pour obtenir la clé et la valeur d'un élément

```
stream.collect(Collectors.toMap(Person::name, Person::age))  
// Map<String, Integer>
```

- L'implantation n'accepte pas d'avoir deux éléments qui renvoient la même clé

Collectors.**toUnmodifiableMap**(fonctionCle, fonctionValeur) renvoie une version non modifiable de la Map

groupBy(fonctionClé, collector)

Groupe les éléments dans une Map dont les clés sont données par la fonctionClé. Les éléments qui ont la même valeur sont envoyés au collecteur pris en paramètre.

```
stream.collect(  
    Collectors.groupingBy(Person::age, Collectors.toList())  
    // Map<Integer, List<Person>>
```

```
stream.collect(  
    Collectors.groupingBy(Person::age, Collectors.counting())  
    // Map<Integer, Long>
```

L'API Optional

L'API Optional

java.util.Optional<E>

Représente une valeur qui est présente ou pas

- Permet d'indiquer que la valeur de retour d'une méthode peut ne pas exister
 - C'est mieux que de renvoyer null et voir les codes clients planter
 - On force l'utilisateur de l'API à explicitement gérer le cas où la valeur n'existe pas

Création

- `Optional.empty()`, `Optional.of(E)`, `Optional.ofNullable(E)`
 - si l'élément en paramètre est null, `Optional.of()` lève une `NullPointerException`, alors que `Optional.ofNullable()` renvoie `Optional.empty()`

Méthodes de Optional

`.isPresent()/.isEmpty()` permet de tester s'il existe une valeur

```
Optional.empty().isEmpty() // true  
Optional.of("foo").isEmpty() // false
```

`.orElseThrow()` permet d'obtenir la valeur (il plante sinon)

```
Optional.of("foo").orElseThrow() // foo  
Optional.empty().orElseThrow() // NoSuchElementException
```

`.orElse()/.orElseGet()` indique une valeur/un supplier par défaut

```
Optional.of("foo").orElseGet(() -> "bar") // foo  
Optional.empty().orElseGet(() -> "bar") // bar  
Optional.empty().orElse("bar") // bar
```

Optional en tant que monade

Optional est aussi considéré comme une sorte de stream à 0 ou 1 élément. Il possède les méthodes

- filter, map, flatMap et ifPresent (l'équivalent de forEach)

Un Optional ne possède pas toutes les méthodes de Stream, mais on peut convertir un Optional en Stream avec la méthode stream()

```
var person = new Person("Ana", 32);  
Optional.of(person).mapToInt(Person::age) // compile pas  
Optional.of(person).stream().mapToInt(Person::age) // ok !  
// IntStream
```

API des comparateurs

API des comparateurs

java.util.Comparator

Interface fonctionnelle qui permet de comparer les éléments deux à deux

```
public interface Comparator<T> {  
    int compare(T element, T other);  
}
```

La sémantique de "compare" est la même que strcmp en C

- si la valeur renvoyée < 0 est, element est $<$ à other
- si la valeur renvoyée est > 0 , element est $>$ à other
- si la valeur renvoyée est $= 0$, element est égal à other

Comparator.comparing()

Méthode `static` qui permet de créer un comparateur en indiquant une fonction de projection dont le résultat est utilisé pour comparer les éléments

```
Comparator.comparing(Person::name)
```

La méthode `comparing()` a des versions spécifiques pour les types primitifs pour éviter le boxing

```
Comparator.comparingInt(Person::age)
```

Exemple de Comparator.comparingInt()

```
var persons = List.of(new Person("toto", 30),  
                      new Person("titi", 40));  
Optional<Person> result =  
    persons.stream()  
        .min(Comparator.comparingInt(Person::age));  
IO.println(result.isEmpty()); //false  
IO.println(result.orElseThrow()); //Person[name=toto, age=30]
```

En résumé

En résumé

L'API des Stream

Permet de faire des requêtes sur les éléments d'une collection de façon déclarative

- Code plus facile à lire et à maintenir (une fois que l'on connaît l'API)

L'API des Collector permet de faire les opérations de haut niveau : joindre des chaînes, convertir vers une autre collection, grouper des éléments, etc ...