Stream Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 Objectifs d'apprentissage 1
2 Les Streams en Java : une approche fonctionnelle pour traiter les données 2
3 Pourquoi utiliser les streams ? 4
4 Création et Utilisation de Base des Streams 4
4.1 Les Streams : générateurs 1/4 4
4.2 Les Streams : générateurs 2/4 5
4.3 Les Streams : générateurs 3/4 5
4.4 Les Streams : générateurs 4/4 6
4.5 Filtrage 6
4.6 Map 6
5 Exceptions dans les Interfaces Fonctionnelles 7
6 Agrégation des Streams 9
6.1 Production d'une collection à partir d'un Stream 9
6.2 Opérations de groupement 9
6.3 Concaténation en String 10
7 Parallélisme des Streams 10
8 Stream API Enhancements 13
8.1 iterate() avec prédicat 14
8.2 Collectors améliorés 15

1 Objectifs d'apprentissage

À l'issue de cette leçon, vous serez capable de :

- **OOC-STREAM-01** — Expliquer le modèle de traitement déclaratif des streams et le distinguer de l'itération impérative sur collections.
- **OOC-STREAM-02** — Catégoriser les sources de création d'un flux — collections et générateurs — et écrire la construction adaptée à chaque cas.
- **OOC-STREAM-03** — Composer des pipelines de filtrage et de transformation (filter, map, iterate avec prédicat) en maîtrisant le traitement des exceptions.

- **OOC-STREAM-04** — Sélectionner l'opération terminale d'agrégation appropriée — reduce, collect, groupingBy, joining ou toCollection — et en analyser le résultat.
- **OOC-STREAM-05** — Analyser les apports et les limites du parallélisme (parallelStream) ainsi que les évolutions récentes de la Stream API.

```
//|echo: false
//|output: false
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

final static Logger logger = LoggerFactory.getLogger("notebook");
```

```
SLF4J(I): Connected with provider of type
[ch.qos.logback.classic.spi.LogbackServiceProvider]
```

Ce notebook présente brièvement la programmation en pipeline (Stream) à partir de Java 8.

La suite sera illustrée sur un exemple simple composé d'une liste de personnes.

```
record Person(String email, String prenom, String nom, int age) {}

//Immutable List of Records
final List<Person> personnes = List.of(
     new Person("pierre.durand@a.fr","Pierre", "Durand", 20),
     new Person("marie.durand@b.fr","Marie", "Durand", 14),
     new Person("albert.martin@c.fr","Albert", "Martin", 30)
    // ...
);
personnes;
```

```
[Person[email=pierre.durand@a.fr,    prenom=Pierre,    nom=Durand,    age=20],
Person[email=marie.durand@b.fr,     prenom=Marie,     nom=Durand,     age=14],
Person[email=albert.martin@c.fr, prenom=Albert, nom=Martin, age=30]]
```

2 Les Streams en Java : une approche fonctionnelle pour traiter les données

Java 8 a introduit les **streams**, une abstraction puissante pour traiter des collections de données de manière déclarative et fonctionnelle. Contrairement aux collections traditionnelles, un stream ne stocke pas les données mais fournit une vue séquentielle sur une source de données.

Caractéristiques clés des streams :

- **Pipeline de traitement:** Les opérations sur un stream sont enchaînées pour former un pipeline. Chaque étape du pipeline transforme le stream en un nouveau stream.
- **Exécution paresseuse:** Les opérations ne sont exécutées que lorsque le résultat final est consommé (par exemple, en collectant les éléments dans une liste).
- **Parallélisme:** Les streams peuvent être parallélisés pour tirer parti de plusieurs cœurs de processeur.
- **Sources variées:** Les streams peuvent être créés à partir de collections, de tableaux, de fichiers, et même générés à la volée.

```
//| echo: true
//| output: true
List<String> words = Arrays.asList("Java", "est", "un", "langage", "puissant");
List<String> uppercaseWords = words.stream()
                                   .map(String::toUpperCase)
                                   .collect(Collectors.toList());
uppercaseWords.forEach(logger::info);
```

```
15:21:49.220 [IJava-executor-7] INFO  notebook -- JAVA
15:21:49.235 [IJava-executor-7] INFO  notebook -- EST
15:21:49.236 [IJava-executor-7] INFO  notebook -- UN
15:21:49.236 [IJava-executor-7] INFO  notebook -- LANGAGE
15:21:49.236 [IJava-executor-7] INFO  notebook -- PUISSANT
```

Dans cet exemple :

- `words.stream()` crée un stream à partir de la liste `words`.
- `map(String::toUpperCase)` transforme chaque élément en majuscules.
- `collect(Collectors.toList())` collecte les éléments du stream dans une nouvelle liste.
- La liste est ensuite traitée en flux avec une lambda expression

Les opérations principales sur les streams :

- **`filter`:** Filtre les éléments selon un prédicat.
- **`map`:** Applique une fonction à chaque élément.
- **`reduce`:** Réduit les éléments à une valeur unique.
- **`sorted`:** Trie les éléments.
- **`collect`:** Collecte les éléments dans une structure de données.

3 Pourquoi utiliser les streams ?

- **Code plus concis et expressif:** Les opérations sur les streams sont souvent plus courtes et plus faciles à comprendre que les boucles traditionnelles.
- **Parallélisme facile à mettre en œuvre:** Le parallélisme peut être ajouté simplement en appelant la méthode `parallel()`.
- **Intégration avec les fonctionnalités fonctionnelles de Java 8:** Les streams s'intègrent bien avec les lambda expressions et les références de méthode.

4 Création et Utilisation de Base des Streams

Les streams en Java offrent une manière élégante et expressive de traiter des collections de données. Ils permettent de réaliser des opérations telles que le filtrage, la transformation et la réduction de données de manière déclarative. La création d'un stream peut se faire à partir de diverses sources : collections, tableaux, fichiers, ou encore à l'aide de méthodes de fabrique. Par exemple, pour créer un stream à partir d'une liste, il suffit d'appeler la méthode `stream()` sur cette liste. Les streams sont particulièrement utiles pour effectuer des traitements en chaîne, en combinant plusieurs opérations les unes à la suite des autres.

Une fois un stream créé, il est possible d'appliquer différentes opérations sur ses éléments. L'une des opérations les plus simples consiste à parcourir tous les éléments du stream et à effectuer une action sur chacun d'eux. C'est le rôle de la méthode `forEach`. Par exemple, pour afficher en majuscules tous les mots d'une liste, on peut utiliser `forEach` en combinaison avec la méthode `map` pour transformer chaque mot en majuscules. Les streams offrent également de nombreuses autres opérations puissantes, telles que `filter` pour filtrer les éléments, `map` pour transformer les éléments, `reduce` pour réduire un stream à une seule valeur, et bien d'autres encore.

```
//| echo: true
//| output: true
personnes.stream().forEach(p→logger.info("{}, {}",
                                p.nom().toUpperCase(),
                                p.prenom()));
```

```
15:28:55.382 [IJava-executor-0] INFO  notebook -- DURAND, Pierre
15:28:55.392 [IJava-executor-0] INFO  notebook -- DURAND, Marie
15:28:55.393 [IJava-executor-0] INFO  notebook -- MARTIN, Albert
```

4.1 Les Streams : générateurs 1/4

Cet exemple de code Java utilise un stream d'entiers pour afficher les nombres de 10 à 14 sur la console.

```
//| echo: true
//| output: true
import java.util.stream.IntStream;
IntStream.range(10,15)
    .forEach(System.out::println);
```

```
10
11
12
13
14
```

4.2 Les Streams : générateurs 2/4

Cette ligne de code parcourt chaque élément de la liste personnes et affiche chacun d'eux sur la console.

```
//| echo: true
//| output: true
personnes.stream()
    .forEach(System.out::println);
```

```
Person[email=pierre.durand@a.fr, prenom=Pierre, nom=Durand, age=20]
Person[email=marie.durand@b.fr, prenom=Marie, nom=Durand, age=14]
Person[email=albert.martin@c.fr, prenom=Albert, nom=Martin, age=30]
```

4.3 Les Streams : générateurs 3/4

Ce code lit chaque ligne du fichier /etc/legal et l'affiche sur la console.

```
//| echo: true
//| output: true
import java.util.stream.Stream;
try (Stream stream = Files.lines(Paths.get("/etc/legal"))) {
            stream.forEach(System.out::println);
}
```

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

4.4 Les Streams : générateurs 4/4

ou les flux d'entrée/sortie (I/O stream)

```
//| echo: true
//| output: true
try (FileReader fileReader = new FileReader("/etc/legal");
  BufferedReader br = new BufferedReader(fileReader)) {
  br.lines().forEach(System.out::println);
}
```

```
The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

4.5 Filtrage

Ce code utilise un prédicat pour filtrer les personnes majeures d'une liste, puis les affiche à l'aide d'un stream.

```
//| echo: true
//| output: true
personnes.stream()
    .filter(p -> p.age() > 18)
    .forEach(System.out::println);
```

```
Person[email=pierre.durand@a.fr, prenom=Pierre, nom=Durand, age=20]
Person[email=albert.martin@c.fr, prenom=Albert, nom=Martin, age=30]
```

4.6 Map

Il est aussi possible d'appliquer une Function à chaque élément du flux avec `map()`.

Par exemple pour transformer un `Stream<Person>` en `Stream<String>` pour afficher les noms formattés des personnes de plus de 18 ans.

```
//| echo: true
//| output: true
// 1. Créer un stream à partir de la liste "personnes"
personnes.stream()

    // 2. Filtrer les personnes majeures (âge supérieur à 18 ans)
```

```
    .filter(p -> p.age() > 18)

    // 3. Transformer chaque personne en une chaîne de caractères formatée
              .map(p    ->    String.format("%s    %s",    p.nom().toUpperCase(),
p.prenom().toLowerCase()))

    // 4. Afficher chaque chaîne de caractères formatée sur la console
    .forEach(System.out::println);
```

```
DURAND pierre


MARTIN albert
```

Il est aussi possible de passer d'un flux d'objet à un flux de primitif.

Un ensemble d'opération aggrégations (disposant d'un systeme d'itération interne) est proposé (average, sum, min, max, count, ...)

```
//| echo: true
//| output: true
double ageMoyen = personnes.stream()
    .mapToInt(Person::age)
    .average()
    .getAsDouble();
ageMoyen;
```

```
21.333333333333332
```

5 Exceptions dans les Interfaces Fonctionnelles

Lorsqu'une opération sur un élément d'un stream peut lever une exception, la gestion de cette exception au sein d'une expression lambda peut complexifier le code. En effet, les expressions lambda sont conçues pour être concises et fonctionnelles, et l'ajout de blocs `try-catch` peut réduire leur lisibilité. De plus, les exceptions vérifiées (celles qui doivent être déclarées dans la signature d'une méthode) ne peuvent pas être directement levées depuis une expression lambda.

Plusieurs stratégies peuvent être envisagées pour gérer ces situations :

- **Envelopper l'exception dans une `RuntimeException` :** Cette approche permet d'éviter de déclarer l'exception dans la signature de la lambda, mais elle peut masquer les erreurs et rendre le débogage plus difficile.

- **Utiliser des méthodes de référence :** Si l'opération qui lève l'exception est déjà implémentée dans une méthode, une référence de méthode peut être utilisée pour l'invoquer. Cela évite de réécrire le code de gestion des exceptions.
- **Créer des interfaces fonctionnelles personnalisées :** Pour des besoins spécifiques, il est possible de définir des interfaces fonctionnelles qui déclarent des exceptions vérifiées. Cela permet de gérer les exceptions de manière plus explicite.
- **Utiliser des bibliothèques tierces :** Certaines bibliothèques proposent des fonctionnalités pour faciliter la gestion des exceptions dans les streams, comme la possibilité de définir des stratégies de récupération ou de propagation des exceptions.

```
//| echo: true
//| output: true
//Production de 10 entiers à raison d'un par 1/10s et calcul de la somme.
IntStream.range(0, 10)
        .map(i -> {try {
                        Thread.sleep(100);
                        }
                                        catch  (java.lang.InterruptedException  e)
{logger.info("{}",e.getMessage());}
                    return i*2;})
        .sum();
```

```
90
```

```
import java.util.concurrent.CompletableFuture;
import java.util.function.IntFunction;
```

```
//| echo: true
//| output: true

IntFunction<Integer> processElement = i -> {
            try {
                Thread.sleep(100);
                return i * 2;
            } catch (InterruptedException e) {
                // Log the exception with details
            logger.info("Error during calculation {}",e.getMessage());
                return null; // Indicate failure by returning null
```

```
            }
        };

IntStream.range(0, 10)
                .mapToObj(processElement)
               .mapToInt(Integer::intValue) // Convert results to int primitives
                .sum();
```

```
90
```

Une solution élégante est d'écrire une classe wrapper qui gère l'exception par exemple en l'encapsulant dans une runtime exception cf. https://www.baeldung.com/java-lambda-exceptions.

6 Agrégation des Streams

Pour agréger les éléments d'un stream, Java 8 introduit les méthodes reduce() et collect(). Bien qu'elles offrent une grande flexibilité, leur utilisation peut s'avérer complexe pour les débutants. C'est pourquoi, dans la plupart des cas, les méthodes utilitaires de la classe Collectors sont préférables. Ces méthodes fournissent des solutions prêtes à l'emploi pour des opérations d'agrégation courantes comme le calcul de sommes, de moyennes, le regroupement d'éléments, etc.

6.1 Production d'une collection à partir d'un Stream

```
//| echo: true
//| output: true
import java.util.stream.Collectors;

List<String> AdultNames = personnes.stream()
    .filter(p -> p.age() > 18)
    .map(p-> "%s %s".formatted(p.nom().toUpperCase(),
                               p.prenom().toLowerCase()))
    .collect(Collectors.toList());

AdultNames;
```

```
[DURAND pierre, MARTIN albert]
```

6.2 Opérations de groupement

```
//| echo: true
//| output: true
```

```
//Regrouper les prénoms par nom dans une Map.
Map<String, List<String>> PersonfirstnamesGroupedByNames =
    personnes.stream()
        .collect(
            Collectors.groupingBy(
                Person::nom,
                Collectors.mapping(
                    Person::prenom,
                    Collectors.toList())));

PersonfirstnamesGroupedByNames;
```

```
{Durand=[Pierre, Marie], Martin=[Albert]}
```

6.3 Concaténation en String

```
//| echo: true
//| output: true
personnes.stream()
    .filter(p → p.age() > 18)
    .map(p→ "%s %s".formatted(p.nom().toUpperCase(),
                              p.prenom().toLowerCase()))
    .collect(Collectors.joining(",","⟶","⟵"));
```

```
⟶DURAND pierre,MARTIN albert⟵
```

7 Parallélisme des Streams

- **Avantages:**
  - Accélération des traitements sur de grands ensembles de données
  - Meilleure utilisation des ressources matérielles
- **Inconvénients:**
  - Coût de gestion des threads
  - Complexité accrue du code
- **Quand utiliser ?**
  - Lorsque les opérations sur les éléments sont indépendantes
  - Pour des jeux de données volumineux
  - Lorsque les gains de performance sont significatifs par rapport au coût

Le parallélisme des streams permet d'accélérer les traitements en répartissant les calculs sur plusieurs threads. Cependant, cette approche n'est pas sans conséquences. La création et la gestion de threads engendrent un coût en termes de

ressources système. De plus, la complexité du code peut augmenter, notamment lorsqu'il s'agit de synchroniser les différentes tâches parallèles. Il est donc essentiel d'évaluer soigneusement les avantages et les inconvénients du parallélisme avant de l'utiliser.

Pour tirer pleinement parti du parallélisme, il est recommandé d'appliquer cette technique à des opérations indépendantes sur des éléments d'un stream. Les traitements longs et les grands volumes de données sont des cas d'utilisation typiques. Il est également important de mesurer les gains de performance réels obtenus avec le parallélisme, afin de s'assurer que les coûts associés sont justifiés. Dans l'exemple ci-dessus, la méthode `parallelStream()` est utilisée pour créer un stream parallèle. La méthode `map()` applique une fonction à chaque élément du stream, simulant un traitement long avec la méthode `sleep()`. Enfin, la méthode `sum()` réduit le stream à une valeur unique en calculant la somme des éléments.

Dans les exemples ci-dessous une pause artificielle est ajoutée pour favoriser le parallelisme. cf. https://www.baeldung.com/java-when-to-use-parallel-stream

---

```
//| echo: true
//| output: true

import java.util.function.BiFunction;

BiFunction<Random, Integer, Stream<Person>> personGenerator=(random, size)-
>IntStream.range(0, /* limit */ size) //TRICK TO KNOW THE SIZE OF THE STREAM
                    .mapToObj(unused -> {try {Thread.sleep(10);} catch
(java.lang.InterruptedException e) {};
                            return new Person("pierre.durand@a.fr","Pierre",
"Durand", random.nextInt(100));});

long seed = System.currentTimeMillis(); // Get current time in milliseconds
Random random = new Random(seed);

double ageMoyen;
long startTime = System.nanoTime();
ageMoyen = personGenerator.apply(random, 200)
        .mapToInt(Person::age)
        .average()
        .getAsDouble(); //Il s'agit d'un Optionnal
long endTime = System.nanoTime();
System.out.println("moyenne calculée %1.0f en %dms".formatted(ageMoyen, (endTime
- startTime)/1000000));
```

```
moyenne calculée 47 en 2463ms
```

```
//| echo: true
//| output: true

Random random = new Random(seed);

double ageMoyen;
long startTime = System.nanoTime();
ageMoyen =  personGenerator.apply(random, 200)
        .parallel()
        .mapToInt(Person::age)
        .average()
        .getAsDouble(); //Il s'agit d'un Optionnal
long endTime = System.nanoTime();
System.out.println("moyenne calculée %1.0f en %dms".formatted(ageMoyen, (endTime
- startTime)/1000000));
```

```
moyenne calculée 47 en 318ms
```

Les streams parallèles en Java exploitent le modèle de concurrence ForkJoin pour paralléliser les traitements sur les collections. Le framework ForkJoin divise récursivement les tâches en sous-tâches jusqu'à un niveau de granularité optimal, puis les exécute en parallèle. Cette approche permet de tirer pleinement parti des ressources multi-cœurs des machines modernes.

```
//| echo: true
//| output: true
ForkJoinPool pool = new ForkJoinPool();
System.out.println("Parallelism = " +pool.getParallelism());
```

```
Parallelism = 8
```

```
//| echo: true
//| output: true
public static class AgeSumTask extends RecursiveTask<Long> {
    private final List<Person> persons;
    private final int start;
    private final int end;
```

```
    public AgeSumTask(List<Person> persons, int start, int end) {
        this.persons = persons;
        this.start = start;
        this.end = end;
    }

    protected Long compute() {
        if (end - start ⩽ 10_000) { // Seuil pour le découpage
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += persons.get(i).age();
            }
            return sum;
        } else {
            int middle = (start + end) / 2;
            AgeSumTask left = new AgeSumTask(persons, start, middle);
            AgeSumTask right = new AgeSumTask(persons, middle, end);
            left.fork();
            long rightResult = right.compute();
            long leftResult = left.join();
            return leftResult + rightResult;
        }
    }

    }

    public static double calculateAverageAge(List<Person> persons) {
        ForkJoinPool pool = new ForkJoinPool();
        AgeSumTask task = new AgeSumTask(persons, 0, persons.size());
        long totalAge = pool.invoke(task);
        return (double) totalAge / persons.size();
}

double averageAge = calculateAverageAge(
               personnes);System.out.println("La  moyenne  d'âge  est  de  %1.0f
an(s).".formatted(averageAge));
```

```
La moyenne d'âge est de 21 an(s).
```

8 Stream API Enhancements

- Stream API Enhancements (Java 9+)

- takeWhile() et dropWhile()

```
Stream.of(1, 2, 3, 4, 1, 2, 3)
    .takeWhile(n -> n < 3)    // [1, 2]
    .forEach(System.out::println);

Stream.of(1, 2, 3, 4, 1, 2, 3)
    .dropWhile(n -> n < 3)    // [3, 4, 1, 2, 3]
    .forEach(System.out::println);
```

```
1
2



3
4
1
2
3
```

8.1 iterate() avec prédicat

```
Stream.iterate(0, n -> n < 10, n -> n + 2)
    .forEach(System.out::println);

// Crée un Stream vide si null, sinon Stream d'un élément
Stream.ofNullable(null).count();    // 0
Stream.ofNullable("value").count(); // 1

//Optional to Stream
Optional.of("value")
    .stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

```
0
2
4
6
8
VALUE
```

8.2 Collectors améliorés

```
//| echo: false
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.List;
```

```
// filtering
// Create a list of integers from 1 to 9
List<Integer> evenNums = IntStream.range(1, 10)
    .boxed() // Convert IntStream to Stream<Integer>
    .collect(Collectors.filtering(
        n -> n % 2 == 0, // Filter to keep only even numbers
        Collectors.toList() // Collect the results into a List
    ));

evenNums;
```

```
[2, 4, 6, 8]
```

```
// Définition d'une personne avec son nom et sa liste de téléphones
record Person(String nom, List<String> phones) {}

// Jeu de données avec doublons intentionnels (2 "Bob")
List<Person> persons = List.of(
    new Person("Alice", List.of("123", "456")),
    new Person("Bob", List.of("789")),
    new Person("Bob", List.of("555"))
);

// Regroupement et aplatissement des téléphones par nom:
// 1. groupingBy: crée un Map avec les noms comme clés
// 2. flatMapping: fusionne toutes les listes de téléphones en une seule liste
Map<String, List<String>> phonesByName = persons.stream()
    .collect(Collectors.groupingBy(
        Person::nom,                          // Groupe par nom
        Collectors.flatMapping(               // Pour chaque groupe:
            p -> p.phones().stream(),        // - transforme la liste en stream
            Collectors.toList()               // - collecte tous les numéros
        ))
    );
```

```
// Affiche: {Bob=[789, 555], Alice=[123, 456]}
System.out.println(phonesByName);
```

```
{Bob=[789, 555], Alice=[123, 456]}
```