Streams en Java

Support complet

A. Larhlimi

9 novembre 2025

Introduction

- **Les Streams en Java** : Introduits avec Java 8 pour le traitement moderne et fonctionnel des collections.
- Permettent des traitements séquentiels et parallèles.
- Objectifs du cours :
  - Comprendre les principes de base et la motivation.
  - Savoir manipuler l'API Streams et ses principales opérations.
  - Appliquer les Streams sur des cas réels en informatique et réseaux.

Notion de Stream

- Une séquence d'éléments provenant d'une source, traitée en pipeline.
- Immutabilité : la source n'est jamais modifiée.
- Exécution paresseuse : une opération terminale déclenche l'exécution réelle.

Motivation

- Simplifier la manipulation des collections.
- Programmer de manière fonctionnelle, plus concise et lisible.
- Faciliter le parallélisme.

Avant : boucles for/while, code verbeux
Avec Streams : pipelines déclaratifs

Interfaces et classes essentielles

- `Stream<T>` - flux d'objets génériques
- `IntStream`, `LongStream`, `DoubleStream` - pour types primitifs
- `Collectors` - utilitaires pour collecter les flux

Interface	Description
Stream<T>	Flux générique
IntStream, LongStream	Pour primitives
BaseStream	Interface commune
Collectors	Agrégation des résultats

Opérations intermédiaires et terminales

Opérations intermédiaires

- Transforment ou filtrent le flux, renvoient un Stream.
- Exécution paresseuse.
- Exemples : `filter()`, `map()`, `distinct()`, `sorted()`, `flatMap()`

Opérations terminales

- Déclenchent l'exécution, produisent un résultat final (pas un Stream).
- Exemples : `forEach()`, `collect()`, `reduce()`, `count()`, `findFirst()`

Exemples d'utilisation

```
List<String> noms = Arrays.asList("ali", "asma",
    "reda", "amina");
List<String> res = noms.stream()
.filter(n → n.length() > 3)
.map(String::toUpperCase)
.collect(Collectors.toList());
// [ASMA, REDA, AMINA]
```

```
int sommePairs = IntStream.range(1, 11)
.filter(n → n % 2 == 0)
.sum(); // 2+4+6+8+10 = 30
```

Cas d'usage : informatique et réseaux

```java
// Extraction d'erreurs des logs
List<String> erreurs = logs.stream()
.filter(l → l.startsWith("[ERR]"))
.collect(Collectors.toList());

// Extraction IP uniques
List<String> ips = lignes.stream()
.map(l → extraireIP(l))
.filter(Objects::nonNull)
.distinct()
.collect(Collectors.toList());
```

Parallélisme avec Streams

- Utiliser `parallelStream()` pour répartir le traitement sur plusieurs threads.

```
long t1 = System.currentTimeMillis();
long count = clients.parallelStream()
.filter(c → c.getBalance() > 10000)
.count();
long t2 = System.currentTimeMillis();
System.out.println("Temps: " + (t2-t1) + " ms");
```

Points de vigilance :

- Ordre non garanti
- Risque de conflit si effets de bord
- Tester l'apport réel de la parallélisation

Traitements avancés

```
// Stream infini
Stream<Integer> entiers = Stream.iterate(0, n →
    n + 1).limit(10);
entiers.forEach(System.out::println);

// Agr gation
Map<Genre, List<Employe>> parGenre = employes.
    stream()
.collect(Collectors.groupingBy(Employe::getGenre
    ));

// Pipeline optimis
public List<ExpensiveResult> optPipeline(List<
    RawData> data) {
  return data.stream()
  .filter(this::isValid)
  .filter(this::isRelevant)
  .map(this::expensiveTransform)
  .collect(Collectors.toList());
```

Bonnes pratiques et limites

- Privilégier Streams pour pipelines, surtout sur grands ensembles
- Éviter les side effects (modification de variables extérieures)
- Tester le comportement en parallèle
- Un Stream n'est consommé qu'une fois

Exercices pratiques

1. Noms : majuscules, longueur > 5
2. Nombres impairs, tri décroissant
3. Prénoms uniques < 25 ans
4. Logs « CRITICAL », doublons supprimés, nombre total
5. Moyenne prix catégorie « Informatique »
6. Commandes > 1000€, dates triées croissante
7. Max/min/moyenne d'un flux de notes
8. Map service $\rightarrow$ liste noms employés
9. Chaînes avec 'a', triées par longueur, doublons supprimés
10. Titres livres < 2000, triés alphabétiquement

2

```
nombres.stream()
.filter(n → n % 2 != 0)
.sorted(Comparator.reverseOrder())
.collect(Collectors.toList());
```

3

```
utilisateurs.stream()
.filter(u → u.getAge() < 25)
.map(Utilisateur::getPrenom)
.distinct()
.sorted()
.collect(Collectors.toList());
```

5

```
produits.stream()
.filter(p → p.getCategorie().equals("
    Informatique"))
.mapToDouble(Produit::getPrix)
.average();
```

7

```
        notes.stream().mapToInt(Integer::intValue).
            summaryStatistics();
```

Classe Utilisateur et test (noms marocains)

```
public class Utilisateur {
  private String prenom, nom;
  private int age;
  public Utilisateur(String prenom, String nom,
      int age) {
    this.prenom = prenom; this.nom = nom; this.
        age = age;
  }
  public String getPrenom() { return prenom; }
  public String getNom() { return nom; }
  public int getAge() { return age; }
}
```

```
List<Utilisateur> utilisateurs = Arrays.asList(
new Utilisateur("Yassine", "El Amrani", 22),
new Utilisateur("Fatima", "Benhadda", 28),
new Utilisateur("Yassine", "Ait Lahcen", 19),
new Utilisateur("Jamila", "Kabbaj", 24),
new Utilisateur("Omar", "Berrada", 22);
```

Classe Produit et test (noms marocains)

```
public class Produit {
  private String nom, categorie;
  private double prix;
  public Produit(String nom, String categorie,
      double prix) {
    this.nom = nom; this.categorie = categorie;
       this.prix = prix;
  }
  public String getNom() { return nom; }
  public String getCategorie() { return
      categorie; }
  public double getPrix() { return prix; }
}
```

```
List<Produit> produits = Arrays.asList(
new Produit("PC Dell Mohammedia", "Informatique"
    , 10500.0),
new Produit("Souris Casablanca", "Informatique",
    190.0),
```

Conclusion et approfondissement

- Les Streams permettent des traitements performants, parallèles et concis.
- Pour progresser : découvrir les Reactive Streams (Java 9+), tuning ForkJoinPool, et l'intégration data/réseau.
- Expérimentez chaque opération et adaptez aux besoins métiers !