Les Streams en Java

> Programmation Fonctionnelle

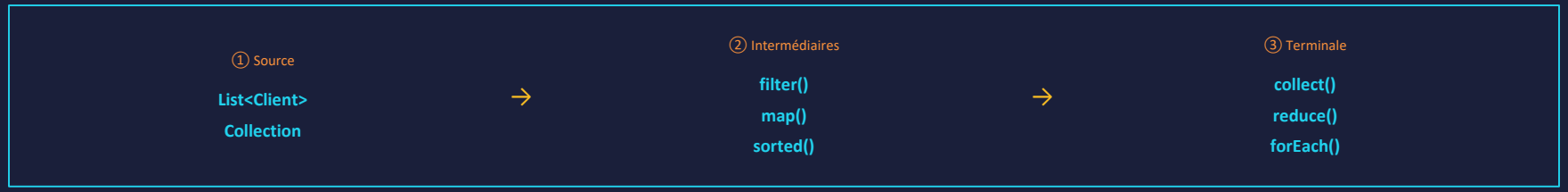
Auteur: A, Larhlimi

Novembre 2025

```
// Pipeline Stream
clients.stream()
.filter(c → c.CA > 500k)
.sorted(byName)
.collect(toList())

// Résultat
[Client, Client, ...]
```

> Qu'est-ce qu'un Stream ?



Définition

Un Stream est une **séquence d'éléments** supportant des opérations agrégées. Introduit avec **Java 8**.

Programmation Déclarative

Exprimer **QUOI faire** plutôt que **COMMENT le faire**. Code plus concis et lisible.

Lazy Evaluation

Les opérations intermédiaires sont **paresseuses**. Elles ne s'exécutent que lors de l'appel de l'opération terminale.

Immutabilité

Les Streams ne modifient **jamais** la source de données. Ils produisent de nouveaux résultats.

> La Classe Client : Modèle de Données

Structure de la Classe

```
public class Client {
private final int idClient;
private final String nom;
private final String adresse;
private final double chiffreAffaire;
// Constructeur, Getters
// equals(), hashCode()
}
```

Exemples de Données (Clients Marocains)

ID	Nom	Ville	CA
101	Ahmadi	Casablanca	150k
102	Bennani	Fès	550k
104	Tijani	Marrakech	1.2M
105	Boutaleb	Fès	320k
108	Fellah	Fès	600k
110	Khallouki	Casablanca	180k

> Opérations Intermédiaires : Filtrage et Transformation

① filter()

Sélectionne les éléments qui satisfont une condition (Predicate).

```
// Clients avec CA > 500k
clients.stream()
.filter(c →
c.getChiffreAffaire() > 500000)
.collect(...)
```

↓ Résultat

Output

Client{id=102, nom='Martin', ...}
Client{id=104, nom='Dubois', ...}
Client{id=108, nom='Petit', ...}

② map()

Transforme chaque élément en un nouvel élément (Function).

```
// Extraire les noms
clients.stream()
.map(Client::getNom)
.collect(
Collectors.toList())
```

↓ Résultat

Output

[Dupont, Martin, Bernard,
Dubois, Thomas, Robert, ...]

Lazy Evaluation (Évaluation Paresseuse)

Les opérations **filter()** et **map()** ne s'exécutent **jamais** tant qu'une opération terminale (collect, forEach, etc.) n'est pas appelée. Cela optimise les performances en évitant les traitements inutiles.

> Opérations Intermédiaires : Tri et Limitation

① sorted()

```
clients.stream()
.sorted(
Comparator
.comparing(
Client::getNom
)
)
```

Résultat (Trié par nom):

```
Alaoui (Fès)
Ahmadi (Casablanca)
Benjelloun (Casablanca)
```

② limit(n)

```
clients.stream()
.sorted(
Comparator
.comparing(
Client::getCA
).reversed()
).limit(3)
```

Résultat (Top 3 CA):

```
Tijani: 1.2M
Ahmadi: 600k
Alaoui: 550k
```

③ skip(n)

```
clients.stream()
.skip(5)
.limit(3)
// Pagination
```

Résultat (Éléments 6-8):

```
Benjelloun (Casablanca)
Chraibi (Agadir)
Ahmadi (Fès)
```

④ distinct()

```
clients.stream()
.map(
Client::getAdresse
)
.distinct()
```

Résultat (Villes uniques):

```
Casablanca, Fès, Marrakech, Agadir
```

> collect() : Accumulation des Résultats

① Collectors.toList()

Créer une Liste

```
clients.stream()
.filter(c → c.CA > 500k)
.collect(
Collectors.toList()
)
```

Résultat

```
List<Client>
[Client(102), Client(104), ...]
```

② Collectors.toSet()

Créer un Ensemble Unique

```
clients.stream()
.map(Client::getAdresse)
.collect(
Collectors.toSet()
)
```

Résultat

```
Set<String>
{Paris, Lyon, Marseille}
```

③ Collectors.toMap()

Créer une Map (Clé-Valeur)

```
clients.stream()
.collect(
Collectors.toMap(
Client::getId,
Client::getNom
)
)
```

Résultat

```
Map<Integer, String>
{101→Dupont, 102→Martin, ...}
```

④ Collectors.groupingBy()

Regrouper par Clé

```
clients.stream()
.collect(
Collectors.groupingBy(
Client::getAdresse
)
)
```

Résultat

```
Map<String, List<Client>>
{Paris→[...], Lyon→[...]}
```

> Opérations Terminales : Réduction & Statistiques

reduce()

```
// Combine les
// éléments en 1
double caTotal =
clients.stream()
.mapToDouble(
Client::getCA
)
.reduce(
0.0,
Double::sum
);
```

Résultat

5,537,500.00 €

Statistiques

count()

Nombre total d'éléments

min()

Élément minimum selon un Comparateur

max()

Élément maximum selon un Comparateur

average()

Moyenne (DoubleStream)

sum()

Somme (Streams Primitifs)

Exemples

```
// Client max CA
Optional<Client>
maxClient =
clients.stream()
.max(
Comparator
.comparing(
Client::getCA
)
);
```

Résultat

Dubois (1.2M €)

> Opérations Terminales : Recherche et Vérification

① anyMatch()

```
boolean existe =
clients.stream()
.anyMatch(c →
c.CA > 1000000);
```

Retourne

boolean

Vrai si au moins un client a CA > 1M

② allMatch()

```
boolean tous =
clients.stream()
.allMatch(c →
c.CA > 50000);
```

Retourne

boolean

Vrai si TOUS les clients ont CA > 50k

③ findFirst()

```
Optional<Client> first =
clients.stream()
.filter(c →
"Lyon".equals(c.ville))
.findFirst();
```

Retourne

Optional<Client>

Premier client de Lyon (ou vide)

④ findAny()

```
Optional<Client> any =
clients.parallelStream()
.filter(c →
c.CA > 500000)
.findAny();
```

Retourne

Optional<Client>

N'importe quel client avec CA > 500k

> Exercice 1 : Filtrage et Tri

📋 Énoncé du Problème

> Trouver tous les clients résidant à **"Fès"** et les trier par **chiffre d'affaires décroissant**. Afficher le résultat dans une liste.

💻 Solution (Pipeline Stream)

```
List <Client> fesClients =
ClientData. getClients ().stream ()
. filter (c → "Fès" .equals(c.getAdresse()))
. sorted (
Comparator. comparing (Client::getChiffreAffaire)
. reversed ()
)
. collect (Collectors. toList ());
```

✅ Résultat Attendu

ID	Nom	CA
108	Ahmadi	**600 000 €**
102	Alaoui	**550 000 €**
105	Boutaleb	**320 000 €**

🔍 Explication de la Solution

Étape 1 : filter()

Sélectionne uniquement les clients dont l'adresse est **"Fès"**. Les autres clients sont ignorés.

Étape 2 : sorted()

Trie les clients de Fès par chiffre d'affaires en ordre **décroissant** (reversed()). Le client avec le CA le plus élevé apparaît en premier.

> Exercice 2 : Regroupement et Agrégation

Énoncé du Problème

Regrouper les clients par **ville** et calculer le **chiffre d'affaires moyen** pour chaque ville. Le résultat doit être une **Map<String, Double>** où la clé est le nom de la ville et la valeur est le CA moyen.

Solution : Pipeline Stream

```
Map<String, Double>
caMoyenParVille =
ClientData.getClients()
.stream()
.collect(
Collectors.groupingBy(
Client::getAdresse,
Collectors
.averagingDouble(
Client::getChiffreAffaire
)
)
);
```

Explication du Pipeline

groupingBy() regroupe les clients par ville. **averagingDouble()** est un **Downstream Collector** qui calcule la moyenne du CA pour chaque groupe de clients.

Résultat Attendu

Ville	CA Moyen
Agadir	210,000.00 €
Casablanca	151,250.00 €
Fès	505,000.13 €
Marrakech	825,000.00 €

Interprétation des Résultats

Casablanca a 4 clients avec un CA moyen de 151k. Fès a 4 clients avec un CA moyen de 505k. Marrakech a 2 clients avec un CA moyen de 825k (le plus élevé). Agadir a 1 client avec un CA de 210k.

> Bonnes Pratiques et Performance

① Immutabilité

Les Streams **ne modifient jamais** la source de données. Ils produisent de nouveaux résultats.

```
// ✓ Correct
List<Client> filtered =
clients.stream()
.filter(...)
.collect(...)
// clients inchangée
```

Avantage

Pas d'effets de bord, code plus sûr et prévisible.

② Lazy Evaluation

Les opérations intermédiaires ne s'exécutent que lors de l'appel d'une opération **terminale**.

```
// Pas exécuté
clients.stream()
.filter(...)
.map(...)
// Exécuté ici
.collect(...)
```

Optimisation

Évite les traitements inutiles et améliore les performances.

③ Streams Parallèles

Utiliser **parallelStream()** pour les gros volumes de données, mais attention aux coûts de synchronisation.

```
// Parallèle
double avg =
clients.parallelStream()
.mapToDouble(...)
.average()
.orElse(0.0);
```

④ Éviter l'Autoboxing

Préférer les **Streams Primitifs** (IntStream, DoubleStream) pour les calculs numériques.

```
// ✓ Meilleur
double sum =
clients.stream()
.mapToDouble(...)
.sum();
```

Performance

> Conclusion et Ressources

🎯 Points Clés à Retenir

① Pipeline Stream

Toujours structurer en trois étapes : **Source** → **Intermédiaires** → **Terminale**.

② Lazy Evaluation

Les intermédiaires ne s'exécutent que lors de l'appel d'une opération **terminale**. Cela optimise les performances.

③ Immutabilité

Les Streams ne modifient jamais la source. Ils produisent de **nouveaux résultats** sans effets de bord.

④ Collectors

Maîtriser **groupingBy()**, **toMap()**, et les **Downstream Collectors** pour les opérations complexes.

📚 Ressources Supplémentaires

Documentation Oracle

Référence officielle complète sur l'API Stream Java 8+. Incontournable pour approfondir.

Tutoriel Baeldung

Tutoriels pratiques et exemples détaillés sur les Streams, Collectors, et opérations parallèles.

Java 8+ Lambdas

Comprendre les expressions lambda est essentiel pour maîtriser les Streams. Pratiquer régulièrement.

Exercices Pratiques

Créer vos propres projets avec des Streams. La pratique est la meilleure façon d'apprendre.

✨ Prochaines Étapes

Vous avez maintenant les connaissances fondamentales sur les Streams en Java. **Pratiquez** avec la classe **Client** et explorez les **Streams Parallèles** pour les gros volumes de données. N'hésitez pas à consulter la documentation officielle pour les cas d'usage avancés.