Support de Cours : Les Streams en Java

Auteur : Manus AI **Date :** Novembre 2025

I. Introduction aux Streams Java

1. Qu'est-ce qu'un Stream ?

Un **Stream** (flux) en Java, introduit dans Java 8, est une séquence d'éléments qui supporte des opérations séquentielles et parallèles. Il ne s'agit pas d'une structure de données en soi, mais plutôt d'une **abstraction fonctionnelle** permettant de traiter des données de manière déclarative.

> *"Un stream est une séquence d'éléments sur laquelle on peut effectuer des opérations de calcul agrégées." [1]*

2. Pourquoi utiliser les Streams ?

L'utilisation des Streams présente plusieurs avantages majeurs par rapport aux boucles traditionnelles (`for`, `while`) :

Caractéristique	Streams	Boucles Traditionnelles
Style de Programmation	Déclaratif (Quoi faire)	Impératif (Comment le faire)
Lisibilité	Code plus concis et expressif	Code potentiellement verbeux
Parallélisme	Facile à implémenter (`parallelStream()`)	Nécessite une gestion manuelle des threads
Mutabilité	Les Streams ne modifient pas la source de données	Les boucles peuvent modifier la source

3. Différence entre Collections et Streams

Il est crucial de distinguer les **Collections** des **Streams** :

- **Collections** : Sont des structures de données (comme `List`, `Set`, `Map`) qui **stockent** des éléments. Elles sont orientées vers la gestion des données.
- **Streams** : Sont des outils qui **traitent** les données. Ils sont orientés vers le calcul et le pipeline d'opérations.

4. Les trois étapes d'un Stream

Tout pipeline de Stream suit généralement trois étapes :

1. **Source** : La source de données (Collection, Tableau, Fichier I/O, Générateur).
2. **Opérations Intermédiaires** : Opérations qui transforment le Stream en un autre Stream. Elles sont **lazy** (paresseuses), c'est-à-dire qu'elles ne s'exécutent que lorsqu'une opération terminale est appelée. Exemples : `filter()`, `map()`, `sorted()`.
3. **Opération Terminale** : Opération qui produit un résultat ou un effet de bord. Elle est **eager** (ardente) et déclenche l'exécution de toutes les opérations intermédiaires. Exemples : `forEach()`, `collect()`, `reduce()`.

II. Préparation : La Classe `Client`

Pour illustrer les concepts, nous utiliserons la classe `Client` définie comme suit :

```
public class Client {
    private final int idClient;
    private final String nom;
    private final String adresse; // Ville
    private final double chiffreAffaire;

    // Constructeur, Getters, toString(), equals(), hashCode()
    // ... (voir fichier Client.java)
}
```

Nous utiliserons une liste de clients d'exemple fournie par la classe `ClientData` :

```
List<Client> clients = ClientData.getClients();
// Exemple de données :
// Client{id=101, nom='Dupont', adresse='Paris', CA=150000.00}
// Client{id=104, nom='Dubois', adresse='Marseille', CA=1200000.75}
```

III. Les Opérations Intermédiaires (Lazy Operations)

Ces opérations retournent un nouveau Stream et permettent de construire le pipeline.

1. Filtrage : `filter(Predicate<T> predicate)`

Permet de sélectionner les éléments qui correspondent à une condition.

Exemple : Filtrer les clients dont le chiffre d'affaires (CA) est supérieur à 500 000.

```
clients.stream()
    .filter(c -> c.getChiffreAffaire() > 500000)
    .forEach(System.out::println);
```

2. Transformation : `map(Function<T, R> mapper)`

Applique une fonction à chaque élément du Stream pour produire un nouveau Stream d'un type potentiellement différent.

Exemple : Obtenir la liste des noms des clients.

```
List<String> noms = clients.stream()
    .map(Client::getNom) // Équivalent à c -> c.getNom()
    .collect(Collectors.toList());
```

3. Tri : `sorted()`

Trie les éléments du Stream. Peut prendre un `Comparator` en argument.

Exemple : Trier les clients par nom.

```
clients.stream()
    .sorted(Comparator.comparing(Client::getNom))
    .forEach(System.out::println);
```

4. Limitation/Saut : `limit(long maxSize)` et `skip(long n)`

- `limit()` : Tronque le Stream pour qu'il ne contienne pas plus que le nombre d'éléments spécifié.
- `skip()` : Ignore les `n` premiers éléments du Stream.

Exemple : Afficher les 3 clients ayant le plus grand CA.

```
clients.stream()
    .sorted(Comparator.comparing(Client::getChiffreAffaire).reversed())
    .limit(3)
    .forEach(System.out::println);
```

5. Déduplication : `distinct()`

Retourne un Stream avec les éléments uniques, en utilisant les méthodes `equals()` et `hashCode()` de l'objet.

Exemple : Afficher les villes uniques des clients.

```
clients.stream()
    .map(Client::getAdresse)
    .distinct()
    .forEach(System.out::println);
```

IV. Les Opérations Terminales (Eager Operations)

Ces opérations consomment le Stream et produisent un résultat final.

1. Collecte : `collect(Collector<T, A, R> collector)`

C'est l'opération terminale la plus courante, utilisée pour accumuler les éléments du Stream dans une Collection ou une Map. La classe `java.util.stream.Collectors` fournit de nombreuses implémentations de `Collector`.

Collector	Description	Exemple d'utilisation
`toList()`	Accumule les éléments dans une `List`.	`collect(Collectors.toList())`
`toSet()`	Accumule les éléments dans un `Set` (éléments uniques).	`collect(Collectors.toSet())`
`toMap()`	Accumule les éléments dans une `Map`.	`collect(Collectors.toMap(Client::getIdClient, c -> c))`
`groupingBy()`	Regroupe les éléments selon une clé.	`collect(Collectors.groupingBy(Client::getAdresse))`

Exemple : Regrouper les clients par ville.

```
Map<String, List<Client>> clientsParVille = clients.stream()
    .collect(Collectors.groupingBy(Client::getAdresse));
```

2. Réduction : `reduce()`

Combine les éléments du Stream en un seul résultat.

Exemple : Calculer le chiffre d'affaires total de tous les clients.

```
double caTotal = clients.stream()
    .map(Client::getChiffreAffaire)
    .reduce(0.0, Double::sum); // 0.0 est l'identité
```

3. Recherche et Vérification

Ces opérations sont utiles pour vérifier des conditions ou trouver un élément. Elles retournent souvent un `Optional<T>`.

Opération	Description	Retourne
`findFirst()`	Retourne le premier élément du Stream.	`Optional<Client>`
`findAny()`	Retourne n'importe quel élément du Stream (utile pour les Streams parallèles).	`Optional<Client>`
`anyMatch()`	Vérifie si au moins un élément correspond au prédicat.	`boolean`
`allMatch()`	Vérifie si tous les éléments correspondent au prédicat.	`boolean`

Exemple : Vérifier si au moins un client a un CA supérieur à 1 000 000.

```
boolean grosClientExiste = clients.stream()
    .anyMatch(c -> c.getChiffreAffaire() > 1000000); // true
```

4. Statistiques

Les opérations comme `count()`, `min()`, `max()` permettent d'obtenir des valeurs statistiques.

Exemple : Trouver le client avec le CA maximum.

```
Optional<Client> clientMaxCA = clients.stream()
    .max(Comparator.comparing(Client::getChiffreAffaire));
```

V. Les Streams Primitifs

Pour éviter l'autoboxing/unboxing coûteux lors du traitement de grands nombres, Java fournit des Streams spécialisés pour les types primitifs : `IntStream`, `LongStream`, et `DoubleStream`.

Exemple : Calculer la moyenne du CA (qui est un `double`).

```
double caMoyen = clients.stream()
    .mapToDouble(Client::getChiffreAffaire) // Conversion en DoubleStream
    .average() // Opération spécifique au DoubleStream
    .orElse(0.0);
```

VI. Exercices Pratiques (Basés sur la classe `Client`)

Les exercices suivants utilisent la liste de clients fournie par `ClientData.getClients()`.

Exercice 1 : Filtrage et Tri

Objectif : Trouver tous les clients résidant à **"Lyon"** et les trier par **chiffre d'affaires décroissant**.

Exercice 2 : Transformation et Collecte

Objectif : Créer une `Map<Integer, String>` où la clé est l'**ID du client** et la valeur est son **nom**.

Exercice 3 : Agrégation

Objectif : Calculer le **chiffre d'affaires total** des clients de **"Paris"**.

Exercice 4 : Opérations Complexes (Regroupement)

Objectif : Regrouper les clients par **ville** et, pour chaque ville, calculer le **chiffre d'affaires moyen** de ses clients. Le résultat doit être une `Map<String, Double>`.

Exercice 5 : Vérification et Transformation

Objectif : Vérifier si **tous** les clients ont un CA supérieur à 50 000. Si oui, afficher la liste des noms des clients en **majuscules**.

VII. Solutions des Exercices

(Les solutions seront fournies dans la section suivante.)

VIII. Conclusion

L'API Stream de Java est un outil puissant qui permet d'écrire du code plus **fonctionnel**, **lisible** et **maintenable**. En maîtrisant les opérations intermédiaires et terminales, vous pouvez transformer des tâches complexes de manipulation de collections en pipelines de données élégants.

Ressources supplémentaires :

- [1] Documentation officielle Oracle sur les Streams Java.
- [2] Tutoriel Baeldung sur l'API Stream Java 8.

VII. Solutions des Exercices

Pour exécuter ces solutions, vous aurez besoin des classes `Client` et `ClientData` ainsi que des imports suivants :

```
import com.manus.streams.Client;
import com.manus.streams.ClientData;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.OptionalDouble;
import java.util.stream.Collectors;
```

Solution Exercice 1 : Filtrage et Tri

Objectif : Trouver tous les clients résidant à **"Lyon"** et les trier par **chiffre d'affaires décroissant**.

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

System.out.println("Clients de Lyon (CA décroissant) :");
lyonClients.forEach(System.out::println);
```

Solution Exercice 2 : Transformation et Collecte

Objectif : Créer une `Map<Integer, String>` où la clé est l'**ID du client** et la valeur est son **nom**.

```
Map<Integer, String> idToNameMap = ClientData.getClients().stream()
    .collect(Collectors.toMap(
        Client::getIdClient, // Clé : idClient
        Client::getNom       // Valeur : nom
    ));

System.out.println("\nMap ID -> Nom :");
idToNameMap.forEach((id, nom) -> System.out.println("ID " + id + " : " +
nom));
```

Solution Exercice 3 : Agrégation

Objectif : Calculer le **chiffre d'affaires total** des clients de **"Paris"**.

```
double caTotalParis = ClientData.getClients().stream()
    .filter(c -> "Paris".equals(c.getAdresse()))
    .mapToDouble(Client::getChiffreAffaire) // Utilisation d'un DoubleStream pour la performance
    .sum(); // Opération terminale sum()

System.out.printf("\nChiffre d'affaires total des clients de Paris : %.2f\n", caTotalParis);
```

Solution Exercice 4 : Opérations Complexes (Regroupement)

Objectif : Regrouper les clients par **ville** et, pour chaque ville, calculer le **chiffre d'affaires moyen** de ses clients. Le résultat doit être une `Map<String, Double>`.

```
Map<String, Double> caMoyenParVille = ClientData.getClients().stream()
    .collect(Collectors.groupingBy(
        Client::getAdresse, // Clé de regroupement : adresse (ville)
        Collectors.averagingDouble(Client::getChiffreAffaire) // Downstream Collector : calcul de la moyenne
    ));

System.out.println("\nCA Moyen par Ville :");
caMoyenParVille.forEach((ville, moyenne) -> System.out.printf("%s : %.2f\n", ville, moyenne));
```

Solution Exercice 5 : Vérification et Transformation

Objectif : Vérifier si **tous** les clients ont un CA supérieur à 50 000. Si oui, afficher la liste des noms des clients en **majuscules**.

```
boolean caSup50k = ClientData.getClients().stream()
    .allMatch(c -> c.getChiffreAffaire() > 50000);

System.out.println("\nTous les clients ont-ils un CA > 50 000 ? " +
caSup50k);

if (caSup50k) {
    List<String> nomsMajuscules = ClientData.getClients().stream()
        .map(Client::getNom)
        .map(String::toUpperCase)
        .collect(Collectors.toList());

    System.out.println("Noms en majuscules : " + nomsMajuscules);
}
```