Stream JAVA

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA

API Stream

Sommaire

- Introduction
- Création d'un stream
- Opérations intermédiaires
- Opérations terminales
- Optional
- Traitement de fichier par flux

Introduction 1/3

L'introduction de l'API Stream dans Java 8 a révolutionné la manière de **traiter les Collections et les tableaux**, traditionnellement gérés par le pattern Iterator. Cette nouvelle API propose une approche simplifiée et efficace pour effectuer des traitements sur ces structures de données.

L'API Stream se caractérise par plusieurs propriétés distinctives. Tout d'abord, **un Stream ne stocke pas de données**, ce qui le différencie des Collections classiques. De plus, il **ne modifie pas les données de la source originale**, ce qui garantit la cohérence des données lors du traitement.

Introduction 2/3

Un autre aspect de l'API Stream est son **chargement paresseux** des données. Cela signifie que **les données ne sont chargées que lorsque nécessaire**, ce qui permet d'optimiser les performances des applications en évitant le chargement inutile de données.

Les **opérations sur les Streams** sont divisées en **deux catégories** : les **opérations intermédiaires** et les **opérations terminales**. Les opérations **intermédiaires**, telles que map ou filter, sont effectuées de manière paresseuse et **renvoient un nouveau Stream**, tandis que les opérations **terminales** consomment le Stream en produisant un **résultat final**.

Introduction 3/3

En résumé, l'API **Stream** de Java 8 offre une approche moderne et flexible pour **manipuler les données**. En adoptant un modèle paresseux et en **fournissant des opérations puissantes**, elle permet de **simplifier et d'optimiser les traitements sur les Collections et les tableaux en Java**.

Pour plus de détails sur les stream java : Doc Stream Java

Création d'un stream

Un stream est représenté par une instance de l'interface générique Stream.

- On peut créer un Stream en utilisant un objet de type builder

```
Stream<String> stream = Stream.<String>builder().add("element1").add("element2").build();
```

- On peut créer un stream à partir d'un tableau grâce aux méthodes Arrays.stream

```
int[] array = {1, 2, 3, 4, 5};
IntStream stream = Arrays.stream(array);
```

- On peut créer un stream à partir d'une collection car l'interface Collection définit la méthode Collection.stream

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> stream = list.stream();
```

Création d'un stream

Après la création d'un **stream**, nous pourrons appliquer une **succession d'opérations intermédiaires** (filter(), map(),...) ainsi qu'une **opération terminale** (forEach(), collect()). Les **opérations intermédiaires** renvoient systématiquement un stream, ce qui permet de **les enchaîner les unes après les autres**. En revanche, l'**opération terminale** renvoie un autre type (elle peut également renvoyer void).

Exemple

- Création d'une liste de prénoms
- Utilisation d'un stream pour obtenir la liste des prénoms commençant par 'A'
- Utilisation d'un stream pour compter le nombre de prénoms qui contiennent la lettre 'c'

```java
// Création d'une liste de prénoms
List<String> prenoms = Arrays.asList(
  "Alice", "Bob", "Caroline", "David", "Anna", "Catherine");

// Filtrer pour ne garder que les prénoms commençant par "A"
List<String> prenomsCommencantParA = prenoms.stream()
            .filter(prenom -> prenom.startsWith("A"))
            .collect(Collectors.toList());

// Compter les prénoms contenant un "v"
long countWithC = prenoms.stream()
            .filter(prenom -> prenom.contains("v"))
            .count();

System.out.println("Prénoms commençant par 'A': "
                   + prenomsCommencantParA);
System.out.println("Nombre de personnes dont le prénom contient un 'v': "
                   +countWithC);
```

Opérations intermédiaires 1/2

Afin de manipuler un stream plusieurs opérations intermédiaires sont possibles, parmi elles :

- **filter(Predicate)** : renvoie un Stream qui contient les éléments pour lesquels l'évaluation du Predicate passé en paramètre vaut true
- **distinct()** : renvoie un Stream qui ne contient que les éléments uniques (elle retire les doublons). La comparaison se fait grâce à l'implémentation de la méthode equals()
- **limit(n)** : renvoie un Stream que ne contient comme éléments que le nombre fourni en paramètre

Opérations intermédiaires 2/2

- **skip(n)** : renvoie un Stream dont les n premiers éléments sont ignorés
- **map(Function)** : applique la Function fournie en paramètre pour transformer l'élément en créant un nouveau
- **flatMap(Function)** : applique la Function fournie en paramètre pour transformer l'élément en créant zéro, un ou plusieurs éléments

Opérations terminales 1/2

Les **opérations terminales** réalisent des actions qui **consomment le stream** (ne peut pas être réutilisé pour effectuer d'autres opérations) et génèrent un **résultat**.

- **reduce()** : applique une Function pour combiner les éléments afin de produire le résultat
- **collect()** : permet de transformer un Stream qui contiendra le résultat des traitements de réduction dans un conteneur mutable
- **anyMatch(Predicate)** : renvoie un booléen qui précise si l'évaluation du Predicate sur au moins un élément vaut true

Opérations terminales 2/2

- **allMatch(Predicate)** : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut true
- **noneMatch(Predicate)** : renvoie un booléen qui précise si l'évaluation du Predicate sur tous les éléments vaut false
- **findAny()** : renvoie un objet de type Optional qui encapsule un élément du Stream s'il existe
- **findFirst()** : renvoie un objet de type Optional qui encapsule le premier élément du Stream s'il exist

Optional

Optional est une **classe** introduite dans **Java** pour **représenter une valeur qui peut être absente**. Elle permet de rendre explicite la **possibilité qu'un résultat puisse être nul**, aidant ainsi à **éviter** les erreurs de type **NullPointerException**.

Les **Optional** sont souvent utilisés **en conjonction avec les Stream** pour représenter une valeur qui peut être absente. Par exemple, lorsqu'une **opération de recherche dans un Stream ne trouve pas de résultat**, elle peut **renvoyer un Optional vide**

Optional

Les **Optional** nous fournisse **plusieurs methodes** interessantes les plus communes etant :

- **isPresent()** : Vérifie si une valeur est présente.
- **get()** : Récupère la valeur si présente, sinon lance une exception.

Pour plus de détails : Doc Optional

Exemple Optional

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class Main {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("apple", "banana", "cherry");

        // Recherche d'un élément dans la liste
        Optional<String> result = list.stream()
                                      .filter(s -> s.startsWith("p")) // Filtre pour les éléments commençant par "p"
                                      .findFirst(); // Récupère le premier élément correspondant

        // Vérification de la présence de l'élément
        if (result.isPresent()) {
            System.out.println("Premier élément commençant par 'p': " + result.get());
        } else {
            System.out.println("Aucun élément commençant par 'p' trouvé.");
        }
    }
}
```

Traitement de fichier par flux

Il est assez courant d'utiliser les **streams** pour le **traitement de flux de données** en Java, y compris pour la **lecture et le traitement de fichiers**. Les streams offrent une **approche fonctionnelle et expressive pour traiter des collections de données**, des flux d'entrée/sortie et d'autres sources de données.

Cependant, il est important de noter que **les streams ne conviennent pas à tous les types de traitement de flux**, en particulier pour les opérations nécessitant un accès aléatoire aux données ou pour **les fichiers de très grande taille** où la gestion de la mémoire pourrait devenir un problème. Dans de tels cas, d'autres approches comme l'utilisation de flux d'entrée/sortie traditionnels pourraient être plus appropriées.

Exemple Traitement de fichier par flux

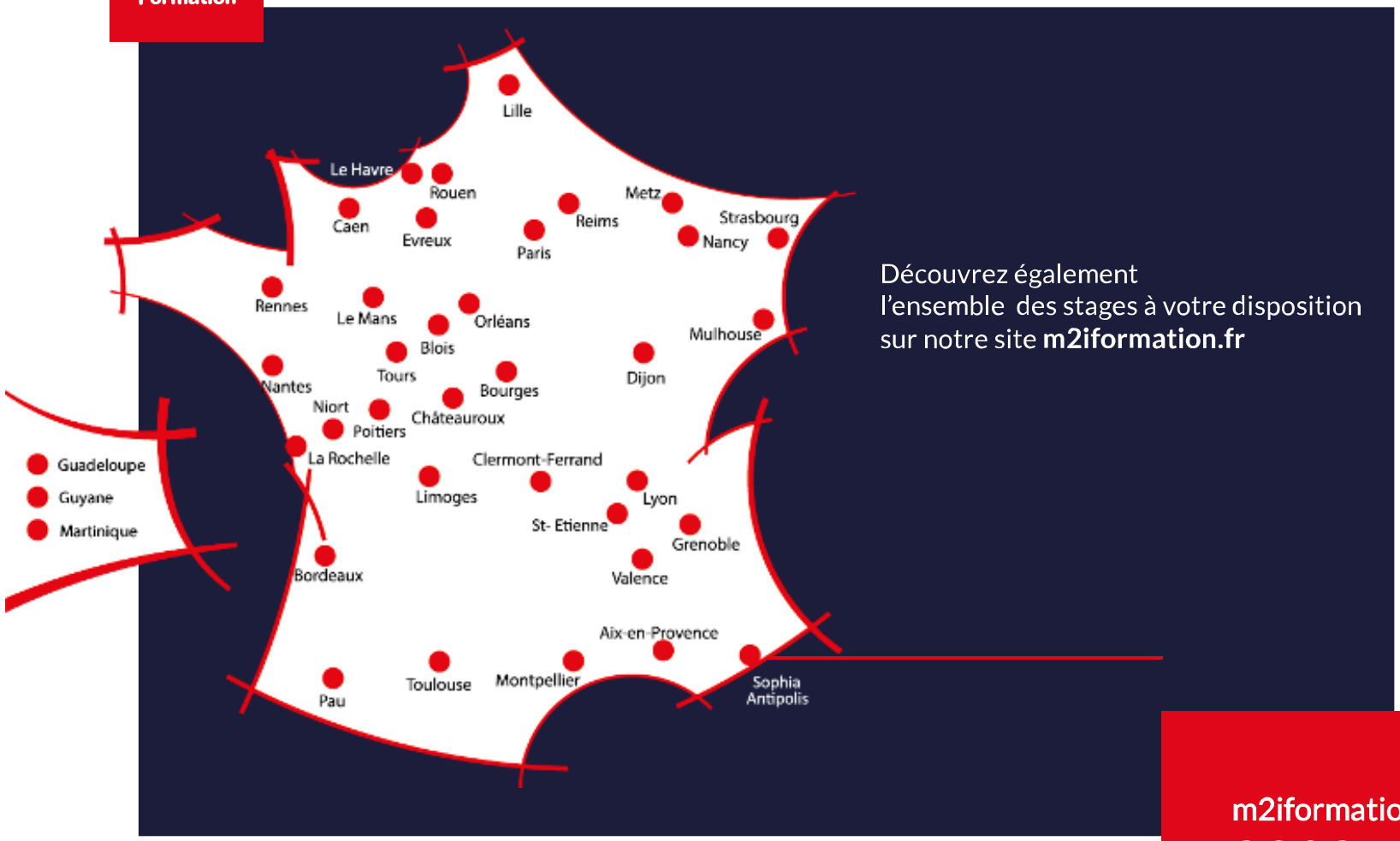
```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String csvFile = "src/main/resources/users.csv";

        try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {
            br.lines() // Crée un stream de lignes du fichier
              .skip(1) // ne tiens pas compte de la premiére ligne
              .map(line -> line.split(",")) // Sépare chaque ligne en un tableau de valeurs
              .forEach(array -> System.out.println(Arrays.toString(array))); // Affiche chaque tableau
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr