Chapitre 7

Les streams

Introduction

- Stream est une API java qui a pour but :
  - Traiter les données de manière déclarative comme SQL
  - Éviter aux développeurs d'écrire des boucles complexes
  - Profiter des traitements parallèles en s'appuyant sur les architectures multicœurs
  - Augmenter l'efficacité du code et réduire sa taille
- Les streams travaillent en collaboration avec les expressions Lambdas

Exemple d'un stream

- Comptage des nombre paires dans une liste :

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

public class ExempleStream {
   public static void main(String[] args) {
      // création d'une liste de nombres de 0 à 9
      List<Integer> nombres = Arrays.asList(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
      // création du Stream, filtrage puis comptage des nombre paires
      long count = nombres.stream().filter(x -> x % 2 == 0).count();
      // affichage des résultats
      System.out.println("Nombre d'éléments pairs : " + count);
   }
}
```


Caractéristiques d'un stream

- Un stream est créé à partir d'une source de données (collection, tableau, fichiers, etc.…)
- Il présente les caractéristiques suivantes :
  - Un stream ne stocke pas de données mais les transfère simplement d'une source vers une suite d'opérations.
  - Un stream ne modifie pas les données de sa source initiale.
  - Pour effectuer des modifications, un nouveau Stream est construit à partir de l'initial garantissant la cohérence lors de la parallélisation des modifications.

Caractéristiques d'un stream

- Un stream, une fois parcouru, n'est pas réutilisable. Pour récupérer les données de la source initiale, il est nécessaire de construire un nouveau stream.
- Un stream peut être non borné, mais les opérations doivent se terminer en un temps fini.
- Le chargement des données pour les opérations sur un stream est réalisé de manière "lazy", ainsi les données ne sont traitées que lorsque cela est nécessaire.

Caractéristiques d'un stream

- Les streams sont représentés par l'interface **Stream**
- Le cas particulier des streams d'entiers peuvent être représentés par l'interface **IntStream**
- Le cas particulier des streams de long peuvent être représentés par l'interface **LongStream**
- Le cas particulier des streams de réels peuvent être représentés par l'interface **DoubleStream**

Génération de Streams

- Un stream en Java peut être généré à partir de différentes sources de données.
- Exemples :
  - **À partir d'une Collection :**

```
List<String> myList = Arrays.asList("a", "b", "c");
Stream<String> myStream = myList.stream();
```

  - **À partir d'un Tableau :**

```
String[] myArray = {"a", "b", "c"};
Stream<String> myStream = Arrays.stream(myArray);
```

  - **À partir d'une Séquence de Valeurs :**

```
Stream<String> myStream = Stream.of("a", "b", "c");
```

  - **À partir d'un Fichier :**

```
Path filePath = Paths.get("mon_fichier.txt");
Stream<String> lines = Files.lines(filePath);
```


Génération de Streams

- **Streams Infinis :**

```
// Génère une séquence infinie de nombres pairs
Stream<Integer> nbPairs = Stream.iterate(0, n -> n + 2);
```

```
// Génère une séquence infinie de nombres aléatoires
Stream<Double> nbAleatoires = Stream.generate(Math::random);
```

- **Streams de Texte :**

```
BufferedReader rdr = new BufferedReader(new FileReader("fichier.txt"));
Stream<String> lines = rdr.lines();
```

Opérations sur les streams

- L'interface Stream définit de nombreuses opérations.
- Elles peuvent être classés en deux catégories :
  - **Opérations intermédiaires** : elles peuvent être enchaînées car elles renvoient un Stream
  - **Opérations terminales** : elles renvoient une valeur différente d'un Stream (ou pas de valeur) et ferme le Stream à la fin de leur exécution.

Operations intermédiaires

- Sont effectuées sur un stream et qui renvoient un nouveau stream.
- Elles ne sont évaluées que lorsque une opération terminale est présente dans le pipeline du stream.

Operations intermédiaires

- Les opérations les plus importantes sont :
  - filter(Predicate<T> predicate)
  - map(Function<T, R> mapper)
  - flatMap(Function<T, Stream<R>> mapper)
  - distinct()
  - sorted()
  - peek(Consumer<T> action)
  - limit(long maxSize)
  - skip(long n)
  - takeWhile(Predicate<T> predicate)
  - dropWhile(Predicate<T> predicate)
  - filter (pour les types primitifs : IntStream, LongStream, DoubleStream, etc.)

Operations intermédiaires

- **Opération filter()**
  - **Syntaxe** : filter(Predicate<T> predicate)
  - **Rôle** : Filtre les éléments du stream en fonction d'un prédicat.
  - **Exemple** : stream.filter(x -> x > 5)
- **Opération map()**
  - **Syntaxe** : map(Function<T, R> mapper)
  - **Rôle** : Transforme chaque élément du stream à l'aide d'une fonction.
  - **Exemple** : stream.map(x -> x * 2)

Operations intermédiaires

- **Opération flatMap()**
  - **Syntaxe** : flatMap(Function<T, Stream<R>> mapper)
  - **Rôle** : Transforme chaque élément du stream en un stream de zéro ou plusieurs éléments, puis concatène ces streams résultants.
  - **Exemple** : stream.flatMap(x -> Stream.of(x, x * 2, x * 3))
- **Opération distinct()**
  - **Syntaxe** : distinct()
  - **Rôle** : Retourne un nouveau stream en éliminant les doublons.
  - **Exemple** : stream.distinct()

Operations intermédiaires

- **Opération sorted()**
  - **Syntaxe** : sorted()
  - **Rôle** : Trie les éléments du stream.
  - **Exemple** : stream.sorted()
- **Opération peek()**
  - **Syntaxe** : peek(Consumer<T> action)
  - **Rôle** : Exécute une action pour chaque élément du stream sans modifier le stream.
  - **Exemple** : stream.peek(x -> System.out.println("Traitement: " + x))

Operations intermédiaires

- **Opération limit()**
  - **Syntaxe** : limit(long maxSize)
  - **Rôle** : Limite le stream à un certain nombre maximal d'éléments.
  - **Exemple** : stream.limit(10)
- **Opération skip()**
  - **Syntaxe** : skip(long n)
  - **Rôle** : Ignore les n premiers éléments du stream
  - **Exemple** : stream.skip(5)

Operations intermédiaires

- **Opération takeWhile()**
  - **Syntaxe** : takeWhile(Predicate<T> predicate)
  - **Rôle** : Retourne les éléments du stream tant que le prédicat est vrai.
  - **Exemple** : stream.takeWhile(x -> x < 10)
- **Opération dropWhile()**
  - **Syntaxe** : dropWhile(Predicate<T> predicate)
  - **Rôle** : Ignore les éléments du stream tant que le prédicat est vrai, puis retourne le reste.
  - **Exemple** : stream.dropWhile(x -> x < 5)

Operations intermédiaires

- **Opération filter() pour les types primitifs**
  - **Syntaxe** : LongStream/DoubleStream/IntStream.filter()
  - **Rôle** : Des variantes de filter spécifiques aux types primitifs (int, long, double) sont disponibles pour éviter la conversion automatique des valeurs.
  - **Exemple** :

```
IntStream intStream = IntStream.of(1, 2, 3, 4, 5);
IntStream filteredStream = intStream.filter(x -> x > 2);
```


Operations intermédiaires

- Remarque:
  - mapToInt(..)/mapToLong(..)/mapToDouble(..) retournent respectivement IntStream, LongStream, DoubleStream
  - Exemple :

```
List<String> list = Arrays.asList("3", "6", "8",  "14", "15");
list.stream().mapToInt(num -> Integer.parseInt(num))
             .filter(num -> num % 3 == 0)
             .forEach(System.out::println);
```

Opérations terminales

- Ils produisent un résultat final et marque la fin du pipeline d'opérations sur un stream.
- Après une opération terminale, le stream ne peut plus être utilisé pour appliquer d'autres opérations.
- Les opérations terminales déclenchent l'évaluation « lazy » des opérations intermédiaires associées dans le pipeline du stream.

Opérations terminales

- Les opérations les plus importantes sont :
  - forEach(Consumer<T> action)
  - toArray()
  - reduce(BinaryOperator<T> accumulator)
  - collect(Collector<T, A, R> collector)
  - min(Comparator<T> comparator) et max(Comparator<T> comparator)
  - count()
  - anyMatch(Predicate<T> predicate)
  - allMatch(Predicate<T> predicate)
  - noneMatch(Predicate<T> predicate)
  - findFirst() et findAny()
  - iterator()

Opérations terminales

- **Opération forEach()**
  - **Syntaxe** : forEach(Consumer<T> action)
  - **Rôle** : Applique une action à chaque élément du stream.
  - **Exemple** : stream.forEach(System.out::println);
- **Opération toArray()**
  - **Syntaxe** : toArray()
  - **Rôle** : Convertit les éléments du stream en un tableau.
  - **Exemple** :Object[] array = stream.toArray();

Opérations terminales

- **Opération reduce()**
  - **Syntaxe** : reduce(int Valinit , BinaryOperator<T> accumulator)
  - **Rôle** : Combine des éléments d'un Stream pour produire une valeur unique (de type Optional)
  - **Exemple** : Optional<Integer> sum = stream.reduce(0,Integer::sum);
- **Opération collect()**
  - **Syntaxe** :collect(Collector<T, A, R> collector)
  - **Rôle** : Collecte les éléments du stream en utilisant un objet Collector.
  - **Exemple** : List<String> resultList = stream.collect(Collectors.toList());

Opérations terminales

- **Opération min() et max()**
  - **Syntaxe** : min(Comparator<T> comparator) //(resp. max)
  - **Rôle** : Trouve le plus petit ou le plus grand élément du stream selon le comparateur fourni.
  - **Exemple** :

```
Optional<Integer> minValue = stream.min(Comparator.naturalOrder());
```

- **Opération count()**
  - **Syntaxe** : count()
  - **Rôle** : Retourne le nombre d'éléments dans le stream.
  - **Exemple** : long count = stream.count();

Opérations terminales

- **Opération anyMatch()**
  - **Syntaxe** : anyMatch(Predicate<T> predicate)
  - **Rôle** : Vérifie si au moins un élément du stream satisfait le prédicat.
  - **Exemple** : boolean anyMatch = stream.anyMatch(x -> x > 5);
- **Opération allMatch ()**
  - **Syntaxe** : allMatch(Predicate<T> predicate)
  - **Rôle** : Vérifie si tous les éléments du stream satisfont le prédicat.
  - **Exemple** : boolean allMatch = stream.allMatch(x -> x > 0);

Opérations terminales

- **Opération noneMatch()**
  - **Syntaxe** : noneMatch(Predicate<T> predicate)
  - **Rôle** : Vérifie si aucun élément du stream ne satisfait le prédicat.
  - **Exemple** : boolean noneMatch = stream.noneMatch(x -> x < 0);
- **Opération findFirst() et findAny()**
  - **Syntaxe** : findFirst() ; findAny()
  - **Rôle** : Retournent le premier élément du stream ou n'importe lequel, respectivement.
  - **Exemple** : Optional<String> firstElement = stream.findFirst();

Opérations terminales

- **Opération iterator()**
  - **Syntaxe** : iterator()
  - **Rôle** : Retourne un itérateur pour les éléments du stream.
  - **Exemple** : Iterator<String> iterator = stream.iterator();

Opérations terminales : Exemple

- Soit liste d'entiers. Créez un stream, calculez le carré de chaque nombre, puis trouvez la somme de ces carrés.
- Solution :

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

public class Exercise3 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sumOfSquares = numbers.stream()
                .mapToInt(n -> n * n)
                .sum();
        System.out.println(sumOfSquares);
    }
}
```


Classe Optional

- Optional est une classe qui représente une valeur qui peut être présente ou absente.
- Quelques méthodes :
  - isPresent(): Renvoie true si la valeur est présente, sinon false.
  - get(): Renvoie la valeur si elle est présente.
  - orElse(T other): Renvoie la valeur si elle est présente, sinon renvoie une valeur par défaut spécifiée.
  - orElseGet(Supplier<? extends T> other): Renvoie la valeur si elle est présente, sinon renvoie la valeur générée par le fournisseur spécifié.

Classe Optional : Exemple

```
import java.util.Optional;
import java.util.Random;

public class Main {
 // Méthode pour générer un nombre aléatoire entre 1 et 100 inclus
   private static Optional<Integer> genererNombreAleatoire() {
      Random random = new Random();
      // Simuler la possibilité de ne rien renvoyer avec 50% de chance
      if (random.nextBoolean()) {   // gènère true ou false
         int nombre = random.nextInt(100) + 1;
         return Optional.of(nombre);
      } else { return Optional.empty(); // Aucun nombre généré }
   }
```


Classe Optional : Exemple

```
public static void main(String[] args) {
      // Génération aléatoire d'un nombre
      Optional<Integer> nombreOptionnel = genererNombreAleatoire();

      // Affichage du nombre si présent
      nombreOptionnel.ifPresent(nombre ->
            System.out.println("Nombre généré : " + nombre));

      // Utilisation d'une valeur par défaut si le nombre est absent
      int nombreParDefaut = nombreOptionnel.orElse(0);
      System.out.println("Nombre par défaut si absent : " + nombreParDefaut);
   }

}
```

Classe Collectors

- Fournit un ensemble de méthodes statiques pour créer des objets Collector.
- Souvent utilisé en conjonction avec la méthode collect() des streams.
- Rassemble les éléments d'un stream dans une structure de données (liste, map, etc..)
- Quelques méthodes :
  - toList() : accumule les éléments du stream dans une liste.
  - toSet() : accumule les éléments du stream dans un ensemble.
  - toMap() : accumule les éléments du stream dans une map.

Classe Collectors

- counting() : Crée un Collector qui compte le nombre d'éléments dans le stream.
- joining() : concatène les éléments du stream en une seule chaîne de caractères.
- groupingBy() : groupe les éléments en fonction d'une propriété ou d'une fonction.
- partitioningBy() : partitionne les éléments du stream en deux groupes en fonction d'un prédicat.
- summarizingInt() : fournit des statistiques (min, max, somme, moyenne, compteur) sur un stream d'entiers.
- reducing() : réduit les éléments du stream en utilisant une opération de réduction spécifiée.

Classe Collectors : Exemple 1

- Regroupement de noms par leurs longueurs.

```
import java.util.*;
import java.util.stream.Collectors;
public class Main {
  public static void main(String[] args) {
    List<String> listeNoms = Arrays.asList("Ali", "Omar", "Taha", "Sami",
"Ola");
    // Regrouper les mots par leur longueur
    Map<Integer, List<String>> gr = listeNoms.stream()
            .collect(Collectors.groupingBy(String::length));
   // Afficher les résultats
   gr.forEach((longueur, noms) -> System.out.println("Noms de longueur "
                                                    + longueur + ": " + noms));
  }
}
```

```
Noms de longueur 3: [Ali, Ola]
Noms de longueur 4: [Omar, Taha, Sami]
```


Classe Collectors : Exemple 2

- Soit une liste d'entiers. Créez un stream, filtrez les nombres pairs, doublez-les, puis stockez le résultat dans une liste.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Exercise1 {
   public static void main(String[] args) {
      List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
      List<Integer> resultat = nombres.stream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * 2)
            .collect(Collectors.toList());
      System.out.println(resultat);
   }
}
```

Streams parallèles

- Avec l'augmentation de la puissance de calcul des machines modernes, l'utilisation de plusieurs cœurs devient cruciale pour optimiser les performances.
- Les streams parallèles en Java offrent une approche simple pour tirer parti du parallélisme.
- Les streams parallèles peuvent être créés de deux manières:
  1) À partir d'un stream séquentiel
  2) Directement à partir d'une source de données

Streams parallèles : Création

- À partir d'un stream séquentiel
  - Se fait grâce à la méthode **parallel()**
  - Exemple :

```
List<String> myList = Arrays.asList("Java", "Python", "C++", "JavaScript");

// Création d'un stream séquentiel
Stream<String> sequentialStream = myList.stream();

// Conversion du stream séquentiel en un stream parallèle
Stream<String> parallelStream = sequentialStream.parallel();
```

Streams parallèles : Création

- Directement à partir d'une source de données
  - Se fait grâce à la méthode **parallelStream()**
  - Exemple :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
// Création directe d'un stream parallèle à partir d'une liste
Stream<Integer> parallelNumberStream = numbers.parallelStream();
```


Streams parallèles : Utilisation

- Les streams parallèles sont particulièrement utiles pour les opérations qui peuvent être décomposées en tâches indépendantes et exécutées simultanément sur différents threads.
- Exemple 1 : Filtrage Parallèle

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Filtrage des nombres pairs en parallèle
List<Integer> parallelEvenNumbers = numbers.parallelStream()
                                    .filter(num -> num % 2 == 0)
                                    .collect(Collectors.toList());
```

Streams parallèles : Utilisation

- Exemple 2 : Somme Parallèle

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Calcul de la somme des nombres en parallèle
int sum = numbers.parallelStream()
            .mapToInt(Integer::intValue)
            .sum();
```


Streams parallèles : Utilisation

- Remarque :
  - Le traitement séquentiel s'effectue sur un seul thread, tandis que le traitement parallèle utilise plusieurs threads pour diviser la charge de travail.
- Exemple 3:

```
List<String> myList = Arrays.asList("Java", "Python", "C++", "JavaScript");

// Traitement parallèle pour convertir les éléments en majuscules
List<String> parallelUpperCaseList = myList.parallelStream()
                                           .map(String::toUpperCase)
                                           .collect(Collectors.toList());
```