JAVA Stream

m2iformation.fr

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA

Stream

Sommaire

Stream

Introduction

Java 8 propose l'API Stream pour mettre en oeuvre une approche de la programmation fonctionnelle sachant que Java est et reste un langage orienté objet.

Le concept de Stream existe déjà depuis longtemps dans l'API I/O, notamment avec les interfaces InputStream et OutputStream. Il ne faut pas confondre l'API Stream de Java 8 avec les classes de type xxxStream

Introduction

L'API Stream facilite l'exécution de traitements sur des données de manière séquentielle ou parallèle.

- Ceci permet au Stream de pouvoir :
  - Optimiser les traitements exécutés grâce au lazyness et à l'utilisation d'opérations de type short-circuiting qui permettent d'interrompre les traitements avant la fin si une condition est atteinte
  - Exécuter certains traitements en parallèle à la demande du développeur

Introduction

L'API Stream permet de réaliser des opérations fonctionnelles sur un ensemble d'éléments. De nombreuses opérations de l'API Stream attendent en paramètre une interface fonctionnelle ce qui conduit naturellement à utiliser les expressions lambdas

Le besoin de l'API Stream

Il est fréquent dans les applications de devoir manipuler un ensemble de données. Pour stocker cet ensemble de données, Java propose, depuis sa version 1.1, l'API Collections.

Le besoin de l'API Stream

Avec Java 5, il est possible d'utiliser un Iterator avec les generics.Avec Java 5, il est possible d'utiliser un Iterator avec les generics.

```
List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
Iterator<Integer> it = entiers.iterator();
long somme = 0;
while (it.hasNext()) {
  int valeur = it.next();
  if (valeur < 10) {
    somme += valeur;
  }
}
System.out.println(somme);
```

Le besoin de l'API Stream

Java 5 a aussi introduit une nouvelle syntaxe de l'instruction for qui permet de faciliter l'écriture du code requis pour parcourir une collection

```
long somme = 0;
 for (int valeur : entiers) {
   if (valeur < 10) {
     somme += valeur;
   }
 }
 System.out.println(somme);
```

Le besoin de l'API Stream

Java 8 propose la méthode forEach() dans l'interface Collection qui attend en paramètre une interface fonctionnelle de type Consumer.

```
LongAdder somme = new LongAdder();
entiers.forEach(valeur -> {
  if (valeur < 10) {
    somme.add(valeur);
  }
});
System.out.println(somme);
```

Le besoin de l'API Stream

L'ajout du support de traitement en parallèle dans l'API Collections aurait été très compliqué : le choix a été fait de définir une nouvelle API permettant de :

- Faciliter l'exécution de traitements sur un ensemble de données
- Réduire la quantité de code nécessaire pour le faire
- Permettre d'exécuter des opérations en parallèle de manière très simple

Le besoin de l'API Stream

```
long somme = entiers.stream()
                        .filter(v -> v <10)
                        .mapToInt(i -> i)
                        .sum();
    System.out.println(somme);
```

Elle permet notamment très facilement d'exécuter ces traitements en parallèle.

```
long somme = entiers.parallelStream()
                        .filter(v -> v <10)
                        .mapToInt(i -> i)
                        .sum();
    System.out.println(somme);
```

Le besoin de l'API Stream

Ainsi avant Java 8, la seule manière de traiter les éléments d'une collection est d'itérer sur ceux-ci.

```
long total = 0;
   int nbPers = 0;
   for (Personne personne : personnes) {
     if (personne.getGenre() == Genre.FEMME) {
       nbPers++;
       total += personne.getTaille();
     }
   }
   double resultat = (double) total / nbPers;
   System.out.println("Taille moyenne des femmes = " + resultat);
```

Le besoin de l'API Stream

Les traitements de l'exemple précédent peuvent être réécrits en utilisant l'API Stream.

```
resultat = personnes
  .stream()
  .filter(p -> p.getGenre() == Genre.FEMME)
  .mapToInt(p -> p.getTaille())
  .average()
  .getAsDouble();
System.out.println("Taille moyenne des femmes = " + resultat);
```

L'API Stream

Interface	Description
BaseStream<T,S extends BaseStream<T,S>>	Interface de base pour les Streams
Collector<T,A,R>	Interface pour une opération de réduction qui accumule les éléments dans un conteneur mutable avec éventuellement une transformation du résultat un fois tous les éléments traités
DoubleStream	Un Stream dont la séquence est composée d'éléments de primitif double
XXXStream	Un Stream dont la séquence est composée d'éléments primitif
Stream	Un stream dont la séquence est composée d'éléments de type T

L'API Stream

L'interface BaseStream<T,S extends BaseStream<T,S>> est l'interface mère des principales interfaces utilisées :

- Stream : interface permettant le traitement d'objets de type T de manière séquentielle ou parallèle
- IntStream : spécialisation pour traiter des données de type int
- LongStream : spécialisation pour traiter des données de type long
- DoubleStream : spécialisation pour traiter des données de type double

L'API Stream

Si les éléments du Stream sont des données numériques alors il n'est pas recommandé d'utiliser un Stream où T est de type Double, Long ou Integer. Les opérations sur ces objets vont nécessiter de l'autoboxing.

Le rôle d'un Stream

Il est très fréquent de vouloir traiter des données stockées dans une collection. Cependant, l'exécution de traitements sur une collection présente plusieurs inconvénients :

- Elle requière de nombreuses lignes de code notamment quelques-unes pour gérer l'itération sur les différents éléments ou pour réaliser des opérations de base
- Il n'est pas facile de faire exécuter ces traitements de manière parallèle

Le rôle d'un Stream

Un Stream peut être vu comme une abstraction qui permet de réaliser des opérations définies sur un ensemble de données.

Le but principal d'un Stream est de pouvoir décrire de manière expressive des traitements à exécuter sur un ensemble de données en limitant au maximum le code à produire pour y arriver.

Le rôle d'un Stream

La déclaration d'un Stream se fait en plusieurs étapes :

- Création d'un Stream à partir d'une source : collection ou autre
- Définition de zéro, une ou plusieurs opérations intermédiaires qui renvoient toutes un Stream
- Définition d'une opération terminale qui va permettre d'obtenir le résultat des traitements et qui va clôturer le Stream

Le rôle d'un Stream

```
List<String> nomEmployes =
      employes.stream()
              .filter(e -> e.getGenre() == Genre.HOMME)
              .sorted(Comparator.comparingInt(Employe::getTaille)
                                .reversed())
              .map(Personne::getNom)
              .collect(Collectors.toList());
```

Le rôle d'un Stream

Le code via le stream est plus compact car il ne contient que les informations indispensables à la bonne exécution des traitements :

- On obtient un Stream à partir de la collection contenant les données
- On applique plusieurs opérations filtre pour ne conserver sur les employés masculins, trie ces employés par taille décroissante et extraction des noms
- On les insère dans une collection de type List

Le rôle d'un Stream

De plus, comme la manière d'exécuter ces traitements est laissée à l'implémentation du Stream, il est possible que celui-ci les exécute en parallèle sur simple demande en remplaçant la méthode stream() par parallelStream().

Un Stream utilise d'autres concepts lors de sa mise en oeuvre :

- Lazyness : les données sont consommées de manière tardive
- Short-circuiting : certaines opérations peuvent interrompre le traitement des éléments
- Internal iteration : c'est l'API qui réalise l'itération sur les données à traiter

Le mode de fonctionnement d'un Stream

Il existe deux types d'opérations : les opérations intermédiaires et les opérations terminales. L'ensemble des opérations effectuées par un Stream est appelé pipeline d'opérations (operation pipeline).
Il est important que :

- Les opérations exécutées par le Stream ne modifient pas la source de données utilisée par le Stream
- Les données de la source de données ne doivent pas être modifiées durant leur traitement par le Stream car cela pourrait avoir un impact sur leur exécution

Les opérations pour définir les traitements d'un Stream

Les opérations d'un Stream permettent de décrire des traitements, potentiellement complexes, à exécuter sur un ensemble de données.
Il existe deux catégories d'opérations :

- 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

Les opérations pour définir les traitements d'un Stream

Les opérations proposées par un Stream sont des opérations communes :

- Pour filtrer des données
- Pour rechercher une correspondance avec des éléments
- Pour transformer des éléments
- Pour réduire les éléments et produire un résultat

Les filtres

- Pour filtrer des données, un Stream propose plusieurs opérations :
  - 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
  - skip(n) : renvoie un Stream dont les n premiers éléments sont ignorés

Les correspondance

Pour rechercher une correspondance avec des éléments, un Stream propose plusieurs opérations :

- anyMatch(Predicate) : renvoie un booléen qui précise si l'évaluation du Predicate sur au moins un élément vaut true
- 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

Les correspondance

- 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 existe

Transformer les données

Pour transformer des données, un Stream propose plusieurs opérations :

- 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

Produire un resultat

Pour réduire les données et produire un résultat, un Stream propose plusieurs opérations :

- 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

La différence entre une collection et un Stream

Il est tentant de voir un Stream comme une collection. Les notions de collections et de Streams concernent toutes les deux une séquence d'éléments. De manière grossière, une collection permet de stocker cette séquence d'éléments et un Stream permet d'exécuter des opérations sur cette séquence d'éléments.

La différence entre une collection et un Stream

- Bien que les Collections et les Streams semblent avoir des similitudes, ils ont des objectifs différents :
  - Les collections permettent de gérer et de récupérer des données qu'elles stockent
  - Les Streams assurent l'exécution lazy de traitements déclarés sur des données d'une source

La différence entre une collection et un Stream

- Les Streams diffèrent donc de plusieurs manières des Collections :
  - Un Stream n'a pas forcement de taille fixe : le nombre d'éléments à traiter peut potentiellement être infini. Ils peuvent consommer des données jusqu'à ce qu'une condition soit satisfaite : des méthodes comme limit() ou findFirst() peuvent alors permettre de définir une condition d'arrêt
  - Une opération ne peut consommer qu'une seule fois un élément. Un nouveau Stream doit être recréer pour traiter de nouveau un élément comme le fait un Iterator

La différence entre une collection et un Stream

- Les Streams ne stockent pas de données. Ils transportent et utilisent les données issues d'une source qui les stockent ou les génèrent
- Les Streams exécutent un pipeline d'opérations sur les données de leur source
- Les Streams sont de nature fonctionnel : les opérations d'un Stream produisent des résultats mais ne devrait pas modifier les données de leur source

La différence entre une collection et un Stream

- Un Stream permet de mettre en oeuvre la programmation fonctionnelle : le résultat d'une opération ne doit pas modifier l'élément sur lequel elle opère ni aucun autre élément de la source. Par exemple, une opération de filtre ne doit pas retirer des éléments de la source mais créer un nouveau Stream qui contient uniquement les éléments filtrés

Le pipeline d'opérations d'un Stream

- Les opérations sont des méthodes définies dans l'interface Stream :
  - Les méthodes intermédiaires : map(), mapToInt(), flatMap(), mapToDouble(), filter(), distinct(), sorted(), peek(), limit(), skip(), parallel(), sequential(), unordered()
  - Les méthodes terminales : forEach(), forEachOrdered(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny(), iterator()

La composition de Collectors

Il est possible de combiner des Collectors pour réaliser des réductions plus complexes : par exemple faire des groupements à plusieurs niveaux.

Ces combinaisons sont similaires à celles utilisables en SQL : il est possible de combiner un GROUP BY avec des opérations telles que COUNT. Il est donc possible d'utiliser un autre Collector pour déterminer la valeur comme par exemple pour compter le nombre d'éléments de chaque groupe.

La composition de Collectors

- Une surcharge de la méthode groupBy() qui attend en second paramètre un objet de type Collector permet d'appliquer le Collector pour réduire les éléments du groupe et ainsi obtenir la valeur associée à la clé.

```
Stream<String> mots = Stream.of("aa", "bb", "aa", "bb","cc", "bb");
Map<String, Long> nbMots = mots.collect(
  Collectors.groupingBy(s -> s.toUpperCase(), Collectors.counting()));
for (Map.Entry entry : nbMots.entrySet()) {
  System.out.println(entry.getKey() + ", " + entry.getValue());
}
```

```
CC, 1
BB, 3
AA, 2
```

Implémentation d'un Collector

Le JDK propose en standard un ensemble d'implémentations de Collector pour des besoins courants. Il est possible de développer sa propre implémentation de l'interface Collector pour définir des opérations de réductions personnalisées si celles-ci ne sont pas fournies par le JDK.

Implémentation d'un Collector

L'interface Collector implémentée doit être typée avec 3 génériques :

- public interface Collector<T, A, R> {...}

Les trois types génériques correspondent aux types utilisés par le Collector :

- T : le type des objets qui seront traités
- A : le type de l'objet utilisé comme accumulator
- R : le type de l'objet qui sera retourné comme résultat

Implémentation d'un Collector

Il est courant que les types A et R soient identiques : c'est par exemple le cas pour un type de l'API Collection. Mais ils peuvent être différents par exemple un accumulator de type StringBuilder et un résultat de type String.

L'implémentation d'un Collector peut se faire de deux manières :

- Utiliser une des surcharges de la fabrique of() de l'interface Collector : elle permet de créer des Collector simples
- Définir une classe qui implémente l'interface Collector : pour des cas plus complexes ou des besoins particuliers

Les Streams pour des données primitives

L'interface Stream utilise un type generic T et s'utilise donc avec des objets de type T. Il existe aussi plusieurs interfaces dédiées à la manipulation de types primitifs int, long et double respectivement IntStream, LongStream et DoubleStream.

Lorsqu'un Stream est utilisé sur des données primitives cela peut engendrer de nombreuses opérations de boxing/unboxing pour encapsuler une valeur primitive dans un objet de type wrapper et vice versa. Ces opérations peuvent être coûteuses lorsque le nombre d'éléments à traiter dans le Stream est important.

Les Streams pour des données primitives

Il est parfois utile voire même nécessaire de transformer un Stream primitif en Stream d'objets et vice versa.
Les méthodes mapToInt(), mapToLong() et mapToDouble() de l'interface Stream fonctionne comme la méthode map() sauf qu'au lieu de renvoyer un Stream, elles renvoient respectivement un IntStream, un LongStream et un DoubleStream.

Le traitement des opérations en parallèle

Une des fonctionnalités les plus mises en avant concernant les Streams est la facilité déconcertante pour exécuter les traitements des opérations en parallèle.

L'avènement des processeurs multi-cour, omniprésents dans les appareils informatique (ordinateurs, tablettes, téléphones mobiles, ...) oblige à mettre en oeuvre des traitements en parallèle pour exploiter leur puissance.

L'exécution de traitements sur des données d'une collection requiert de réaliser une itération sur chacun des éléments. Le traitement d'éléments dans une itération est par définition intrinsèquement séquentiel.

Le traitement des opérations en parallèle

La mise en oeuvre de ces traitements en parallèle est très compliquée même avec l'API Fork/Join. Pourtant elle peut être nécessaire si le volume des données est important. Pour ne pas réécrire intégralement l'API Collection et ainsi maintenir la rétrocompatibilité, Java SE 8 propose dans l'API Streams le traitement en parallèle de données.

Très facilement, l'API Stream permet de réaliser ses traitements en séquentiel ou en parallèle. Seules des méthodes par défaut pour l'obtention d'un Stream à partir d'une Collection ont été ajoutées à cette API.

Le traitement des opérations en parallèle

```
List<String> elements = Arrays.asList("elem1", " elem2", " elem3", "elem4", "elem5");
elements.stream()
        .forEach(System.out::println);
elements.parallelStream()
        .forEach(System.out::println);
```

Les Streams infinis

Un Stream infini fait référence à un Stream dont la source produit de manière continue des éléments à consommer. Sans condition d'arrêt, cette génération peut être infinie comme son l'indique.

La mise en oeuvre de Stream infini est possible car le principe de traitements des éléments d'un Stream est lazy. Les opérations intermédiaires définissent les traitements mais ceux-ci ne sont réellement exécutés que lors de l'invocation de l'opération terminale.

Les Streams infinis

```
Stream<Integer> stream = IntStream.iterate(0, i -> i + 1);
```

Attention lors de l'utilisation de Stream infinis : il est nécessaire de fournir une restriction qui va limiter le nombre d'éléments générés sinon le Stream va exécuter sans arrêt la génération de nouveaux éléments.

```
IntStream.iterate(0, i -> i + 1).limit(5).forEach(System.out::println);
```

Le débogage d'un Stream

Il est possible d'utiliser la méthode peek() pour afficher l'élément en cours de traitement dans le pipeline. C'est ailleurs normalement la seule raison d'utiliser la méthode peek() dans un pipeline d'opérations. Pour simplement afficher l'élément courant, il suffit de lui passer en paramètre la référence de méthode System.out::println. Si l'on doit utiliser la méthode peek() plusieurs fois dans le pipeline, il est préférable de construire une chaîne de caractères qui fournisse des informations complémentaires notamment l'étape courante dans le pipeline.

Les limitations de l'API Stream

L'API Stream permet la mise en oeuvre d'une approche fonctionnelle dans le langage Java. Java reste un langage orienté objet et n'est pas un langage fonctionnel. L'utilisation de l'API Stream n'est pas aussi riche ou poussée que dans d'autres langages qui sont fonctionnels.

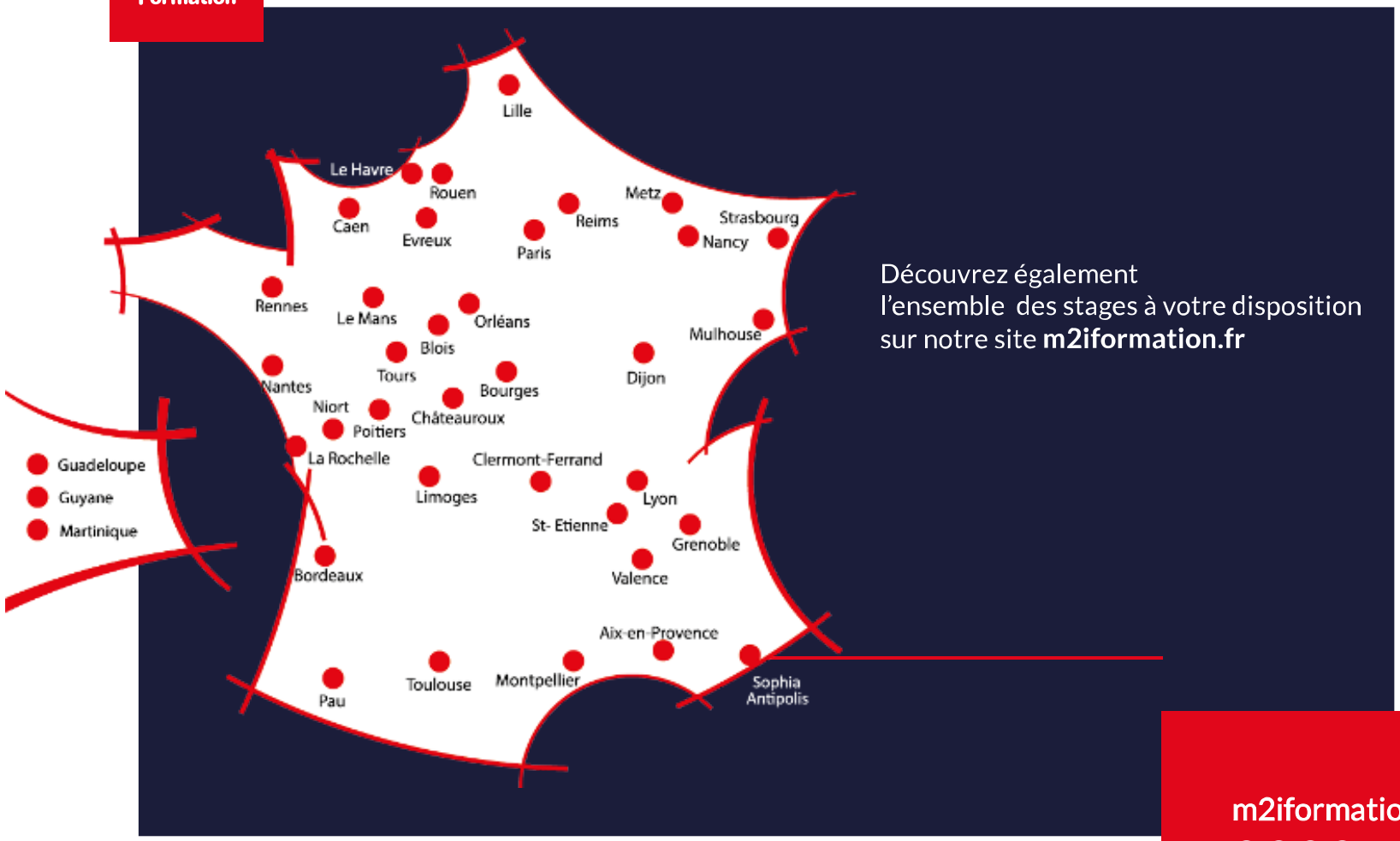
L'API Stream possède aussi quelques limitations. Par exemple, il n'est pas possible de définir ses propres opérations : seules celles définies par l'API peuvent être utilisées.

Un Stream n'est pas réutilisable

Une fois qu'un Stream a été exécuté, il ne peut plus être réutilisé. Dès qu'une opération terminale est invoquée, celle-ci ferme le Stream.

Si le Stream est de nouveau invoqué, une exception de type IllegalStateException est levée.

```
public static void main(String[] args) {
   Stream<String> stream = Stream.of("a1", "a2", "a3")
                                 .filter(s -> s.startsWith("a"));
   stream.forEach(System.out::println);
   stream.forEach(System.out::println);
 }
```



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

m2iformation.fr