Formation Java 8

TP n°4 : L'api STREAM

SOMMAIRE

1. Prérequis :

- JDK 8.
- Eclipse.

2. Objectifs

- ✓ Comprendre le besoin et le fonctionnement de l'API Stream.
- ✓ Comprendre le pipeline d'opérations d'un Stream.
- ✓ Maitriser les opérations intermédiaires.
- ✓ Maitriser les opérations terminales.
- ✓ Maîtriser Les Collectors.
- ✓ Comprendre comment utiliser les Streams avec les opérations I/O.
- ✓ Comprendre l'exécution des Streams en mode parallèle.
- ✓ Apprendre à débugger les Streams.
- ✓ Quelques bonnes pratiques.

3. Introduction

❖ En programmation fonctionnelle, on décrit le résultat souhaité mais pas comment on obtient le résultat. Ce sont les fonctionnalités sous-jacentes qui se chargent de réaliser les traitements requis en tentant de les exécuter de manière optimisée. Ce mode de fonctionnement est similaire à SQL : le langage SQL permet d'exprimer une requête mais c'est le moteur de la base de données qui choisir la meilleure manière d'obtenir le résultat décrit. Comme avec SQL, la manière dont on exprime le résultat peut influencer la manière dont le résultat va être obtenu notamment en termes de performance.

❖ Java 8 propose l'API Stream pour mettre en œuvre 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 de Java I/O. Les streams de Java I/O permettent de lire ou écrire des données dans un flux (sockets, fichiers, ...). Le concept de Stream de Java 8 est différent du concept de flux (stream) de l'API I/O même si l'approche de base est similaire : manipuler un flux d'octets ou de caractères pour l'API I/O et manipuler un flux de données pour l'API Stream. Cette dernière repose sur le concept de flux (stream en anglais) qui est une séquence d'éléments.

❖ L'API Stream facilite l'exécution de traitements sur des données de manière séquentielle ou parallèle. Les Streams permettent de laisser le développeur se concentrer sur les données et les traitements réalisés sur cet ensemble de données sans avoir à se préoccuper de détails techniques de bas niveau comme l'itération sur chacun des éléments ou la possibilité d'exécuter ces traitements de manière parallèle.

❖ L'API Stream de Java 8 propose une approche fonctionnelle dans les développements avec Java. Elle permet de décrire de manière concise et expressive un ensemble d'opérations dont le but est de réaliser des traitements sur les éléments d'une source de données. Cette façon de faire est

complètement différente de l'approche itérative utilisée dans les traitements d'un ensemble de données avant Java 8. 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

❖ 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 et les références de méthodes dans la définition des Streams. Un Stream permet donc d'exécuter des opérations standards dont les traitements sont exprimés grâce à des expressions lambdas ou des références de méthodes.

❖ Un Stream permet d'exécuter une agrégation d'opérations de manière séquentielle ou en parallèle sur une séquence d'éléments obtenus à partir d'une source dans le but d'obtenir un résultat.

❖ Dans l'exemple ci-dessus, une collection d'éléments de type Emp est utilisée comme source pour un Stream qui va effectuer des opérations de type filter-map-reduce pour calculer la moyenne de l'âge des employés de sexe masculin. La moyenne calculée par la méthode average() est une opération dite de réduction.

Le code utilisé est :

- Concis et clair : car il repose sur l'utilisation d'opérations prédéfinies.
- Exprimé de manière déclarative (c'est une description du résultat attendu) plutôt que de manière impérative (c'est une description des différentes étapes nécessaires à l'exécution des traitements)

```
package ma.formation.stream;

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

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

public class Test1 {

	public static void test1() {
		List<Emp> employes = Arrays.asList(
				new Emp("emp_1", 33, Genre.HOMME),
				new Emp("emp_2", 43, Genre.HOMME),
				new Emp("emp_3", 52, Genre.HOMME),
				new Emp("emp_4", 50, Genre.HOMME),
				new Emp("emp_5", 50, Genre.HOMME),
				new Emp("emp_6", 49, Genre.FEMME),
				new Emp("emp_7", 41, Genre.FEMME),
				new Emp("emp_8", 39, Genre.FEMME),
				new Emp("emp_9", 37, Genre.FEMME),
				new Emp("emp_10", 25, Genre.FEMME));
```

```
            double ageMoyeDesHommes = employes.stream().
                        filter(e -> e.getGenre() == Genre.HOMME).
                        mapToInt(e -> e.getAge())
                        .average().orElse(0);
            System.out.println(ageMoyeDesHommes);
      }
}
@Data
@NoArgsConstructor
@AllArgsConstructor
class Emp {
      String nom;
      Integer age;
      Genre genre;
}
enum Genre {
      HOMME, FEMME
}
```

❖ Les éléments traités par le Stream sont fournis par une source qui peut être de différents types :

- ✓ Une collection
- ✓ Un tableau
- ✓ Un flux I/O
- ✓ Une chaîne de caractères
- ✓ ...

❖ Avec l'API Stream, il est possible de déclarer de manière concise des traitements sur une source de données qui seront exécutés de manière séquentielle ou parallèle. Le traitement en parallèle d'un ensemble de données requiert plusieurs opérations :

- ✓ La séparation des données en différents paquets plus petits (forking)
- ✓ L'exécution dans un thread dédié des traitements sur chaque élément d'un paquet, généralement en utilisant un pool de threads pour limiter la quantité de ressources utilisées
- ✓ La combinaison des résultats de chaque paquet pour obtenir le résultat final (joining)

❖ Java 7 propose le framework Fork/Join pour faciliter la mise en œuvre de ces opérations en parallèle. L'API Stream utilise ce framework pour l'exécution des traitements en parallèle.

❖ L'utilisation de l'API Stream possède donc plusieurs avantages :

- ✓ L'exécution, sur un ensemble de données, de traitements définis de manière déclarative.
- ✓ La quantité de code à produire pour obtenir un traitement similaire reposant sur sa propre itération est réduite.
- ✓ La possibilité d'exécuter ses traitements en parallèle.

4. 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.

Différentes évolutions ont permis de rendre de plus en plus expressif le code pour traiter ces données. Celles-ci vont être illustrées dans les exemples ci-dessous à l'aide d'un petit algorithme qui va calculer la somme des valeurs inférieures à 10 dans une collection contenant les 12 premiers entiers.

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

```
public static void test2() {
             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);
       }
```

❖ 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 :

```
public static void test3() {
       List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
       long somme = 0;
          for (int valeur : entiers) {
            if (valeur < 10) {
              somme += valeur;
            }
          }
          System.out.println(somme);
}
```

Dans les exemples précédents, le parcours des éléments de la collection est fait à la main sous la forme d'une itération externe (external iteration) qui contient la logique de traitements. Finalement le code à produire pour réaliser cette tâche relativement simple est assez important.

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

```
public static void test4() {
       List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
        LongAdder somme = new LongAdder();
          entiers.forEach(valeur -> {
            if (valeur < 10) {
              somme.add(valeur);
            }
          });
          System.out.println(somme);}
```

❖ Dans les exemples précédents, le code de l'algorithme est exécuté séquentiellement. Il serait beaucoup plus complexe et verbeux pour arriver à le faire exécuter en parallèle (en imaginant que le volume de données à traiter soit beaucoup plus important).

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.

Pour offrir une solution à cette problématique et proposer la possibilité de simplifier la réalisation d'opérations plus ou moins complexes sur des données, Java 8 propose l'API Stream.

```
public static void test5() {
	List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
	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.

```
public static void test6() {
List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
long somme = entiers.parallelStream().filter(v -> v < 10).mapToInt(i -> i).sum();
System.out.println(somme);
}
```

❖ L'API Stream propose donc d'ajouter une manière plus expressive et une approche fonctionnelle au langage Java pour le traitement de données :

- ✓ Il n'y a plus de code pour itérer sur chacun des éléments.
- ✓ Des opérations sont utilisées en leur passant en paramètre des expressions lambdas ou des références de méthode pour indiquer le détail de la tâche à accomplir.

L'interface Stream<T> propose une liste définie d'opérations qu'il sera possible de réaliser. La plupart de ces opérations renvoient une instance de type Stream<T> pour permettre de combiner ces opérations.

a. L'API STREAM

L'API Stream est contenue dans le package java.util.stream. Ce package est composée de 10 interfaces, deux classes et une énumération.

Les interfaces de l'API sont :

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.
DoubleStream.Builder	Un builder pour un DoubleStream.
IntStream	Un Stream dont la séquence est composée d'éléments de primitif int.
IntStream.Builder	Un builder pour un IntStream.
LongStream	Un Stream dont la séquence est composée d'éléments de primitif long.
LongStream.Builder	Un builder pour un LongStream.
Stream<T>	Un stream dont la séquence est composée d'éléments de type T.
Stream.Builder<T>	Un builder pour un Stream.

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

- ✓ Stream<T> : 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

Si les éléments du Stream sont des données numériques alors il n'est pas recommandé d'utiliser un Stream<T> où T est de type Double, Long ou Integer. Les opérations sur ces objets vont nécessiter de l'autoboxing. Dans ces cas, il est préférable d'utiliser un DoubleStream, IntStream ou LongStream pour améliorer les performances des traitements de ces données et profiter de certaines opérations de réductions dédiés aux type primitifs tels que la somme ou la moyenne.

L'API définit plusieurs classes :

Classe	Description
Collectors	Propose plusieurs fabriques pour obtenir des implémentations de l'interface Collector qui implémentent des opérations de réduction communes
StreamSupport	Propose des méthodes de bas niveau pour créer des Streams à partir d'un Spliterator fourni en paramètre

b. 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. Globalement cela passe par une description de ce que l'on veut faire mais pas comment cela va se faire.

De la même manière que SQL permet de définir les données à obtenir et la façon dont on souhaite qu'elles soient restituées, les Streams permettent de décrire des opérations à réaliser sur un ensemble de données.

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.

```
public static void test7() {
List<Emp> employes = Arrays.asList(
new Emp("emp_1", 33, Genre.HOMME),
new Emp("emp_2", 43, Genre.HOMME),
new Emp("emp_3", 52, Genre.HOMME),
new Emp("emp_4", 50, Genre.HOMME),
 new Emp("emp_5", 50, Genre.HOMME),
new Emp("emp_6", 49, Genre.FEMME),
 new Emp("emp_7", 41, Genre.FEMME),
new Emp("emp_8", 39, Genre.FEMME),
new Emp("emp_9", 37, Genre.FEMME),
new Emp("emp_10", 25, Genre.FEMME));
 List<String> nomEmployes =     employes.stream()
                                  .filter(e -> e.getGenre() == Genre.HOMME)
                                  .sorted(Comparator.comparingInt(Emp::getAge)
                                 .reversed())
                                  .map(Emp::getNom)
                                  .collect(Collectors.toList());

                              for (String nom : nomEmployes) {
                                System.out.println(nom);
                              }
        }
```

L'exemple ci-dessus est très simple et lisible 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 âge décroissant et extraction des noms.
- On les insère dans une collection de type List.

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().

```
      public static void test8() {
List<Emp> employes = Arrays.asList(
new Emp("emp_1", 33, Genre.HOMME),
new Emp("emp_2", 43, Genre.HOMME),
new Emp("emp_3", 52, Genre.HOMME),
new Emp("emp_4", 50, Genre.HOMME),
new Emp("emp_5", 50, Genre.HOMME),
new Emp("emp_6", 49, Genre.FEMME),
new Emp("emp_7", 41, Genre.FEMME),
new Emp("emp_8", 39, Genre.FEMME),
new Emp("emp_9", 37, Genre.FEMME),
new Emp("emp_10", 25, Genre.FEMME));
List<String> nomEmployes =
      employes.parallelStream()
      .filter(e -> e.getGenre() == Genre.HOMME)
      .sorted(Comparator.comparingInt(Emp::getAge)
      .reversed())
      .map(Emp::getNom)
      .collect(Collectors.toList());

  for (String nom : nomEmployes) {
      System.out.println(nom);
   }
}
```

c. Les concepts mis en œuvre par les Streams

Une définition simple d'un Stream pourrait être : « l'application d'un ensemble d'opérations sur une séquence d'éléments issus d'une source dans le but d'obtenir un résultat ».

Cette définition met en avant plusieurs concepts :

- ✓ Un ensemble d'opérations : ce sont les traitements ordonnés qui seront exécutés. Chaque opération permet d'effectuer une tâche. Les opérations communes en programmation fonctionnelle sont proposées : filtre (filter), transformation (map), réduction (reduce), recherche (find), correspondance (match), tri (sorted), ...
- ✓ Pipeline d'opérations : de nombreuses opérations renvoient un Stream ce qui permet de les enchaîner et éventuellement de réaliser des optimisations.
- ✓ Une séquence d'éléments : ce sont les données d'un certain type sur lesquelles les opérations vont être appliquées mais le Stream ne stocke pas ses données car elles sont obtenues à la demande.
- ✓ Une source : elle permet de fournir les données à traiter au Stream comme une collection, un tableau ou tout autre ressource capable de fournir un Stream.

De plus, un Stream utilise d'autres concepts lors de sa mise en œuvre :

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

1. Le mode de fonctionnement d'un Stream

Un Stream permet d'exécuter des opérations sur un ensemble de données obtenues à partir d'une source afin de générer un résultat.

```
public static void test9() {
            List<String> maListe = Arrays.asList("a1", "a2", "b2", "b1", "c1");
          maListe.stream()
                 .filter(s -> s.startsWith("b"))
                 .map(String::toUpperCase)
                 .sorted()
                 .forEach(System.out::println);
        }
```

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).

La plupart des opérations d'un Stream attendent en paramètre une interface fonctionnelle pour définir leur comportement. Ces paramètres peuvent ainsi être exprimés en utilisant une expression lambda ou une référence de méthode.

Une opération peut être avec ou sans état.

Il est important que 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.

2. 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. Elles sont définies dans l'interface java.util.stream.Stream grâce à de nombreuses méthodes.

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 intermédiaires ne réalisent aucun traitement tant que l'opération terminale n'est invoquée. Elles sont dites lazy ce qui permettra éventuellement d'effectuer des optimisations dans leur exécution.

```
public static void test10() {
               List<Integer> nombres = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

                  List<Integer> troisPremierNombrePairAuCarre =
                    nombres.stream()
                           .filter(n -> {
                               System.out.println("filter " + n);
                               return n % 2 == 0;
                            })
                           .map(n -> {
                               System.out.println("map " + n);
                               return n * n;
                            })
                           .limit(3)
                           .collect(toList());

                  System.out.println("");
                  troisPremierNombrePairAuCarre.forEach(System.out::println);
}
```

Le résultat est :

```
filter 1
filter 2
map 2
filter 3
filter 4
map 4
filter 5
filter 6
map 6

4
16
36
```

Dans l'exemple ci-dessus, plusieurs optimisations sont mises en œuvre par le Stream. Tous les éléments ne sont pas parcourus. Les opérations ne sont invoquées que le nombre de fois requis sur des éléments pour obtenir le résultat :

L'opération de filtre n'est invoquée que 6 fois

L'opération de transformation n'est invoquée que 3 fois

Ceci est possible car l'opération limit(3) ne renvoie qu'un Stream qui contient au maximum le nombre d'éléments précisé en paramètre. Les opérations sont regroupées en une seule itération qui traite chaque élément en lui application le pipeline d'opérations : les opérations n'itèrent pas individuellement sur les éléments du Stream. Ceci permet d'optimiser au mieux les traitements à réaliser pour obtenir le résultat de manière efficiente.

L'utilisation d'un Stream implique généralement trois choses :

- ✓ Une source qui va alimenter le Stream avec les éléments à traiter.
- ✓ Un ensemble d'opérations intermédiaires qui vont décrire les traitements à effectuer.
- ✓ Une opération terminale qui va exécuter les opérations et produire le résultat.

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.

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 qui ne contient comme éléments que le nombre fourni en paramètre.
- ✓ **skip(n)** : renvoie un Stream dont les n premiers éléments sont ignorés.

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.
- ✓ **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.

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.

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.

d. La différence entre une collection et un Stream

Les Streams diffèrent de plusieurs manières des Collections :

- ✓ Un Stream n'est pas une structure qui stocke des données. Elle permet de traiter les différents éléments d'une source en appliquant différentes opérations.
- ✓ Un Stream permet de mettre en œuvre 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.
- ✓ De nombreuses opérations d'un Stream attendent en paramètre une expression lambda.
- ✓ Il n'est pas possible d'accéder à un élément grâce à un index : seul le premier peut être obtenu ou il faut exporter le résultat sous la forme d'une collection ou d'un tableau.
- ✓ Il est possible de paralléliser l'exécution des opérations. Cette exécution parallèle utilise alors le framework Fork/Join.
- ✓ 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.
- ✓ Les Streams ne stockent pas de données. Ils transportent et utilisent les données issues d'une source qui les stockent où les génèrent.
- ✓ Les Streams exécutent un pipeline d'opérations sur les données de leur source.
- ✓ Les Streams sont de nature fonctionnelle : les opérations d'un Stream produisent des résultats mais ne devrait pas modifier les données de leur source.
- ✓ Les opérations intermédiaires des Streams sont exécutées de manière lazy. Ce mode de fonctionnement permet d'effectuer des optimisations lors de l'exécution en un seul passage du pipeline d'opérations.

e. L'obtention d'un Stream

La création d'un Stream se fait nécessairement à partir d'une source de données. Cette source va permettre de fournir les différents éléments à la demande du Stream pour pouvoir les traiter.

Les Streams peuvent utiliser des données issues d'une source finie ou infinie d'éléments.

Il existe différentes façons de construire un Stream à partir de différentes sources :

- Collection
- Tableau
- Ensemble de données

- Fichier
- ...

Il est aussi possible d'utiliser une Function pour produire un nombre infini d'éléments comme source de données.

Chacune des classes pouvant servir de source, propose une ou plusieurs méthodes pour créer un Stream :

Méthode	**Source**
stream() ou parallelStream() d'une Collection	Les éléments de la collection Stream<String> chaines = Arrays.asList("a1", "a2", "a3").stream();
Stream<T> Arrays.stream(T[])	Les éléments du tableau String[] valeurs = {"a1", "a2", "a3"}; Stream<String> chaines = Arrays.stream(valeurs);
Stream<T> Stream.of(T)	L'élément
Stream<T> Stream.of(T…)	Les éléments passés en paramètre dans le varargs Stream<Integer> stream = Stream.of(1,2,3,4);
IntStream.range(int, int)	Les valeurs entières incluse entre les bornes inférieure et supérieure exclue fournies
IntStream.rangeClosed(int, int)	Les valeurs entières incluse entre les bornes inférieures et supérieures fournies
Stream<T> Stream.iterate(T, UnaryOperator<T>)	Des valeurs générées en appliquant successivement la fonction à partir de la valeur initiale fournie en paramètres Stream<Integer> stream = Stream.iterate(0, (i) -> i + 1);
Stream.empty()	Aucun élément
Stream.generate(Supplier<T>)	Un nombre infini d'éléments créés par le Supplier fourni en paramètre Stream<UUID> stream = Stream.generate(UUID::randomUUID);
Stream<String> BufferedReader.lines()	Les lignes d'un fichier texte
Stream<String> Files.lines	Les lignes d'un fichier texte
IntStream Random.ints()	Un nombre infini d'entiers aléatoires

IntStream BitSet.stream()	Les indices des bits positionnés à 1 dans le BitSet
Stream<String> Pattern.splitAsStream(CharSequence)	Les chaînes de caractères issues de l'application du motif String str = "chaine1,chaine2,chaine3"; Pattern.compile(",") .splitAsStream(str) .forEach(System.out::println);
ZipFile.stream() JarFile.stream()	Les éléments contenus dans l'archive
IntStream String.chars	Les caractères de la chaîne Remarque le Stream obtenu est de type IntStream IntStream stream = "abcdef".chars();

1. La création d'un Stream à partir de ses fabriques

L'interface Stream propose plusieurs fabriques pour créer un Stream.

La méthode of() de l'interface Stream attends en paramètre un varargs d'objets.

```
public static void test11() {
            Stream<Integer> stream = Stream.of(1, 2, 3);

            Stream<int> stream = Stream.of(1, 2, 3); //ne se compile pas !!

            IntStream stream2 = IntStream.of( 1, 2, 3);

            stream.forEach(System.out::println);
            stream2.forEach(System.out::println);

}
```

- Les méthodes iterate() et generate() de l'interface Stream attendent en paramètre une Function. Les éléments générés sont calculés à la demande en invoquant la fonction pour générer la prochaine valeur. Sans contrainte particulière, cette génération se poursuit à l'infini. Dans ce cas, on parle de Stream infini : c'est un Stream dont la source ne connait pas le nombre d'éléments qui sera traité à l'avance.
  Par exemple, la méthode iterate() attend en paramètre une valeur initiale et un UnaryOperator qui sera invoqué pour générer chaque nouvelle valeur.

```
public static void test12() {
    Stream<Integer> entiers = Stream.iterate(0, n -> n + 1);
    entiers.forEach(System.out::println);
}
```

Ce traitement va incrémenter une valeur entière et l'afficher sur la console.

Il est bien sûr possible de définir une condition d'arrêt des traitements du Stream basée sur le nombre d'éléments traités en utilisant la méthode limit() avec en paramètre le nombre souhaité.

```
public static void test13() {
            Stream<Integer> entiers = Stream.iterate(0, n -> n + 1);
          entiers.limit(10)
                 .forEach(System.out::println);

}
```

2. L'obtention d'un Stream à partir d'une collection

```
public static void test14() {
            List<String> elements = new ArrayList<String>();
            elements.add("element1");
            elements.add("element2");
            elements.add("element3");
            Stream<String> stream = elements.stream();
}
```

5. Le pipeline d'opérations d'un Stream

Un Stream exécute une combinaison d'opérations pour former un pipeline.

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 plupart des opérations d'un Stream attendent en paramètres une interface fonctionnelle qui sont généralement définies dans le package java.util.function. Cela permet d'utiliser une expression Lambda pour décrire les traitements qui seront réalisés par l'opération.

Le traitement de données par un Stream se fait en deux étapes :

- Configuration en invoquant des méthodes intermédiaires.
- Exécution des traitements en invoquant la méthode terminale.

Remarque : aucun traitement n'est effectué lors de l'invocation des méthodes intermédiaires. Ces méthodes sont dites lazy.

Dès qu'une méthode terminale est invoquée, le Stream est considéré comme consommé et plus aucune de ces méthodes ne peut être invoquée. Une méthode terminale peut produire une ou plusieurs valeurs ou opérer des effets de bord (forEach).

L'exemple ci-dessous créé une liste des 5 premières personnes dont le nom commence par un 'A'.

```
public static void test15() {
            List<String> prenomsCommencantParA = personnes
                        .stream()
                        .map(Personne::getNom)
                        .filter(nom->nom.startsWith("a"))
                        .limit(5)
                        .collect(Collectors.toList());

      }
```

Il est important de comprendre que les opérations intermédiaires s'exécutent en mode lazy. Leur invocation ne provoque aucun traitement : c'est l'invocation de la méthode terminale qui provoque le traitement des données par les opérations du Stream.

Ce mode de fonctionnement possède plusieurs avantages :

- Généralement permet l'invocation des opérations en une passe.
- Ceci est particulièrement utile lorsque la quantité de données à traiter par le Stream est importante.
- L'API Stream peut optimiser les opérations exécutées notamment en utilisant des opérations de type short-circuiting. Ce type d'opération peut interrompre les traitements du Stream lorsqu'une condition est satisfaite.

Pour illustrer ce comportement, l'exemple ci-dessous utilise un Stream sur une collection de prénoms pour les mettre en majuscule, ne conserver que ceux qui commencent par un 'A' et ne prendre que les deux premiers pour les afficher.

```
public static void test16() {
            List<String>
prenoms=Arrays.asList("Hassan","Anas","Jamal","Ahmed","Mohammed");
            prenoms.stream()
        .map(String::toUpperCase)
        .filter(p -> p.startsWith("A"))
        .limit(2)
        .forEach(System.out::println);

}
```

Bien que l'opération limit() soit la dernière opération intermédiaire, elle a une influence sur l'exécution. Une fois que la condition est atteinte (deux prénoms commençant par A) alors les traitements effectués par le Stream sont interrompus. Ceci est possible car le pipeline d'opérations est appliqué sur chaque donnée.

```java
public static void test17() {
List<String> prenoms = Arrays.asList("Hassan", "Anas", "Jamal", "Ahmed", "Mohammed");
            prenoms.stream().map(p -> {
                  afficher("map    ", p);
                  return p.toUpperCase();
            }).filter(p -> {
                  afficher("filter ", p);
                  return p.startsWith("A");
            }).peek(p -> {
                  afficher("limit ", p);
            }).limit(2).forEach(System.out::println);

      }

      public static void afficher(String message, String p) {
            System.out.println(message + " : " + p);
      }
```

```
map     : Hassan
filter  : HASSAN
map     : Anas
filter  : ANAS
limit   : ANAS
ANAS
map     : Jamal
filter  : JAMAL
map     : Ahmed
filter  : AHMED
limit   : AHMED
AHMED
```

6. Les opérations intermédiaires

Une opération intermédiaire renvoi toujours un Stream.

Le pipeline d'opérations d'un Stream peut avoir zéro, une ou plusieurs opérations intermédiaires.

Les opérations intermédiaires peuvent être réparties en deux groupes :

- ✓ **Les opérations stateless** ne conservent pas d'état lors du traitement d'un élément par rapport au précédent. Chaque élément est donc traité indépendamment des autres.
- ✓ **Les opérations stateful** conserve un état par rapport aux éléments traités précédemment lors du traitement d'un élément. Certaines opérations stateful doivent traiter l'ensemble des éléments avant de pouvoir créer leur résultat. Ceci peut avoir des conséquences lors de l'utilisation d'opérations stateful dans un Stream avec traitements en parallèle : cela peut nécessiter plusieurs itérations sur les données.

L'API Stream définit plusieurs opérations intermédiaires :

Opération	Stateless/Stateful	Rôle
filter	Stateless	Stream<T> filter(Predicate<? super T> predicate) Filtrer tous les éléments pour n'inclure dans le Stream de sortie

		que les éléments qui satisfont le Predicat
map	Stateless	<R> Stream<R> map(Function<? super T,? extends R> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un élément de type R
mapToxxx(Int, Long or Double)	Stateless	xxxStream mapToxxx(ToxxxFunction<? super T> mapper) Renvoyer un Stream qui contient le résultat de la transformation de chaque élément de type T en un type primitif xxx
flatMap	Stateless	<R> Stream<R> flatMap(Function<T,Stream<? extends R>> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les Stream<R> retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type R.
flatMapToxxx (Int, Long or Double)	Stateless	xxxStream flatMapToxxx(Function<? super T,? extends xxxStream> mapper) Renvoyer un Stream avec l'ensemble des éléments contenus dans les xxxStream retournés par l'application de la Function sur les éléments de type T. Ainsi chaque élément de type T peut renvoyer zéro, un ou plusieurs éléments de type primitif xxx.
Distinct	Stateful	Stream<T> distinct() Renvoyer un Stream<T> dont les doublons ont été retirés. La détection des doublons se fait en invoquant la méthode equals()
sorted	Stateful	Stream<T> sorted() Stream<T> sorted(Comparator<? super T>) Renvoyer un Stream dont les éléments sont triés dans un certain ordre. La surcharge sans paramètre tri dans l'ordre naturel : le type T doit donc implémenter l'interface Comparable car c'est sa méthode compareTo() qui est utilisée pour la comparaison des éléments deux à deux La surcharge avec un Comparator l'utilise pour déterminer l'ordre de tri.
peek	Stateless	Stream<T> peek(Consumer <? super T>)

		Renvoyer les éléments du Stream et leur appliquer le Consumer fourni en paramètre
limit	Stateful Short-Circuiting	Stream<T> limit(long) Renvoyer un Stream qui contient au plus le nombre d'éléments fournis en paramètre
skip	Stateful	Stream<T> skip(long) Renvoyer un Stream dont les n premiers éléments ont été ignorés, n correspondant à la valeur fournie en paramètre
sequential		Stream<T> sequential() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est séquentiel
parallel		Stream<T> parallel() Renvoyer un Stream équivalent dont le mode d'exécution des opérations est en parallèle
Unordered		Stream<T> nnordered() Renvoyer un Stream équivalent dont l'ordre des éléments n'a pas d'importance
onClose		Stream<T> onClose(Runnable) Renvoyer un Stream équivalent dont le handler fourni en paramètre sera exécuté à l'invocation de la méthode close(). L'ordre d'exécution de plusieurs handlers est celui de leur déclaration. Tous les handlers sont exécutés même si un handler précédent a levé une exception.

Les opérations sans état (stateless) comme map() ou filter() renvoient simplement le résultat de l'exécution sur l'élément courant dans le Stream pour être traité par l'opération suivante.

Les opérations avec état (stateful) peuvent avoir besoin de conserver des informations relatives au traitement des éléments précédents voir de tous les éléments pour pouvoir produire le Stream contenant les éléments à traiter par l'opération suivante.

a. Les méthodes map(), mapToInt(), mapToLong et mapToDouble()

Le type de la valeur de retour de l'opération map() peut être du même type que la valeur traitée ou d'un type différent.

```
public static void test18() {
            Stream<String> chaines = Stream.of("aaa", "bbb", "ccc");
            chaines.map(chaine -> chaine.toUpperCase()).forEach(System.out::println);

      }
```

Cette méthode peut aussi être une méthode contenant des traitements métiers de l'application.

```
import java.util.Random;
public class Test {
public static void test19() {
            Long[] idEmployes = { 12345L, 67890L };
          List<Emp> listeEmployes = Stream
              .of(idEmployes)
              .map(EmployeeService::rechercherParId)
              .collect(Collectors.toList());
          System.out.println(listeEmployes);
      }
}
class EmployeeService {

        private static Random rnd = new Random();

        public static Emp rechercherParId(Long id) {
          return new Emp();//implémenter ici votre régle métier
        }
}
```

La méthode mapToInt() applique une Function à chaque élément pour produire un IntStream à la place d'un Stream<Integer>

La méthode mapToLong() est similaire à la méthode mapToInt() mais elle renvoie un LongStream.

La méthode mapToDouble() est similaire à la méthode mapToInt() mais elle renvoie un DoubleStream.

Lorsque le résultat de la transformation renvoie une valeur entière ou flottante, il est possible d'utiliser la méthode map() qui va renvoyer un Stream du type du wrapper de la valeur primitive.

```
public static void test20() {
            Double[] valeurs = { 1.0, 2.0, 3.0, 4.0, 5.0 };
            Double[] valeursAuCarre = Stream.of(valeurs).map(v -> v *
```

```
v).toArray(Double[]::new);
            System.out.println(Arrays.deepToString(valeursAuCarre));

}
```

```
[1.0, 4.0, 9.0, 16.0, 25.0]
```

Mais l'utilisation des méthodes mapToDouble(), mapToInt() et mapToLong() améliorent les performances essentiellement car elles évitent des opérations coûteuses de boxing et unboxing. Ceci est particulièrement vrai pour très grande quantité d'éléments.

```
public static void test21() {
Double[] valeurs = { 1.0, 2.0, 3.0, 4.0, 5.0 };
double[] valeursAuCarre = Stream.of(valeurs).mapToDouble(v -> v * v).toArray();
System.out.println(Arrays.toString(valeursAuCarre));
}
```

b. La méthode flatMap()

Les opérations map() et flatMap() servent toutes les deux à réaliser des transformations mais il existe une différence majeure :

- ✓ La méthode map() renvoie un seul élément.
- ✓ La méthode flatMap() renvoie un Stream qui peut contenir zéro, un ou plusieurs éléments.

L'opération map() permet de transformer un élément du Stream mais elle possède une limitation : le résultat de la transformation doit obligatoirement produire un unique élément qui sera ajouté au Stream renvoyé.

L'exemple ci-dessous créé une paire de valeur (valeur -1 et valeur) à partir d'une collection de nombres. La valeur retournée est ainsi une List de List d'Integer.

```
public static void test22() {
            List<Integer> nombres = Arrays.asList(1, 3, 5, 7, 9);
            List<List<Integer>> tuples = nombres
                        .stream()
                        .map(nombre -> Arrays.asList(nombre - 1, nombre))
                        .collect(Collectors.toList());
            System.out.println(tuples);
}
```

```
[[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
```

Le même exemple que précédemment mais utilisant une opération flatMap() renvoie une List d'Integer dont toutes les valeurs ont été mises à plat.

```
public static void test23() {
            List<Integer> nombres = Arrays.asList(1, 3, 5, 7, 9);
              List<Integer> nombresDesTuples =
              nombres.stream()
                        .flatMap(nombre -> Arrays.asList(nombre - 1, nombre).stream())
```

```
                        .collect(Collectors.toList());
                System.out.println(nombresDesTuples);
        }
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'opération flatMap() attend en paramètre une Function qui renvoie un Stream. La Function est appliquée sur chaque élément qui produit chacun un Stream. Le résultat renvoyé par la méthode est un Stream qui est l'agréation de tous les Streams produits par les invocations de la Function.

```
package ma.formation.stream;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Departement {
        private String nom;
        private List<Etudiant> etudiants = new ArrayList<>();

        public Departement(String nom, List<Etudiant> etudiants) {
                super();
                this.nom = nom;
                this.etudiants = etudiants;
        }

        public String getNom() {
                return nom;
        }

        public void setNom(String nom) {
                this.nom = nom;
        }

        public List<Etudiant> getEtudiants() {
                return etudiants;
        }

        public static void test24() {
                List<Departement> departements = Arrays.asList(
                                new Departement("depatement_1",
                                                Arrays.asList(new Etudiant("ETUDIANT_1"), new
Etudiant("ETUDIANT_2"),
                                                                new Etudiant("ETUDIANT_3"), new
Etudiant("ETUDIANT_4"))),
                                new Departement("depatement_2", Arrays.asList(new
Etudiant("ETUDIANT_5"), new Etudiant("ETUDIANT_6"),
                                                new Etudiant("ETUDIANT_7"), new
Etudiant("ETUDIANT_8")))

                );

                departements.stream().flatMap(d ->
d.getEtudiants().stream()).forEach(System.out::println);
        }

        public static void main(String[] args) {
                test24();
        }
```

```
}

class Etudiant {
	private String nom;

	public Etudiant(String nom) {
		this.nom = nom;
	}

	@Override
	public String toString() {
		return nom;
	}
}
```

```
ETUDIANT_1
ETUDIANT_2
ETUDIANT_3
ETUDIANT_4
ETUDIANT_5
ETUDIANT_6
ETUDIANT_7
ETUDIANT_8
```

L'exemple ci-dessous affiche tous les mots d'un fichier texte.

```
public static void test25() {
		try {
			Files.lines(Paths.get("fichier.txt"))
				.map(ligne -> ligne.split("\\s+"))
				.flatMap(Arrays::stream)
				.map(mot -> mot.replaceAll("[^A-Za-z]+", ""))
				.distinct()
				.forEach(System.out::println);
		    } catch (IOException e) {
			e.printStackTrace();
		    }
	}
```

L'exemple ci-dessous créé un Stream à partir d'un Stream de List en utilisant la méthode flatMap().

```
public static void test26() {
Stream<List<String>> listesIngredients = Stream.of(Arrays.asList("carottes", "poireaux",
"navets"),
Arrays.asList("eau"), Arrays.asList("sel", "poivre"));
Stream<String> ingredients = listesIngredients.flatMap(strList -> strList.stream());
ingredients.forEach(System.out::println);
}
```

c. La méthode filter()

La méthode filter() attend en paramètre une interface fonctionnelle de type Predicate : elle permet de filter les éléments du Stream selon la condition fournie en paramètre par l'interface fonctionnelle de type Predicate. Le Predicate est appliqué sur chaque élément du Stream : s'il renvoie true, l'élément est ajouté dans le Stream de sortie, sinon l'élément est ignoré.

```
public static void test27() {
List<String> prenoms = Arrays.asList("mohammed", "ali", "samy", "adam", "said", "jean");
prenoms.stream().filter(p -> p.startsWith("a")).forEach(System.out::println);
	}
```

Il est possible d'utiliser plusieurs opérations filter() dans un même pipeline d'opérations.

```
public static void test28() {
List<String> prenoms = Arrays.asList("mohammed", "ali", "samy", "adam", "said", "jean");
			prenoms.stream()
				.filter(p -> p.startsWith("a"))
				.filter(p -> p.length() == 3)
				.forEach(System.out::println);
	}
```

Il est cependant préférable de regrouper les filtres lorsque cela est possible.

```
public static void test29() {
List<String> prenoms = Arrays.asList("mohammed", "ali", "samy", "adam", "said", "jean");
		prenoms.stream()
			.filter(p -> p.startsWith("a") && p.length() == 3 )
			.forEach(System.out::println);
	}
```

d. La méthode distinct()

L'opération distinct() permet de supprimer les doublons contenus dans un Stream. Les éléments contenus dans le Stream retournés sont donc uniques. La méthode distinct() n'attend aucun paramètre et renvoie un Steam avec les éléments dont les doublons ont été retirés.

```
public static void test30() {
		List<Integer> entiers = Arrays.asList(1, 2, 3, 1, 2, 3);
		entiers.stream()
			.distinct()
			.forEach(System.out::println);
	}
```

La détection des doublons est réalisée en utilisant la méthode equals() des éléments. Il est donc important que celle-ci soit correctement implémentée.

L'exemple suivant ne donne pas le bon résultat car il manque la redéfinition de la méthode hashCode() :

```
package ma.formation.stream;

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

```java
public class MaClasse {

	private String nom;

	public MaClasse(String nom) {
		this.nom = nom;
	}

	public String getNom() {
		return nom;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (obj == null) {
			return false;
		}
		if (getClass() != obj.getClass()) {
			return false;
		}
		MaClasse other = (MaClasse) obj;
		if (nom == null) {
			if (other.nom != null) {
				return false;
			}
		} else if (!nom.equals(other.nom)) {
			return false;
		}
		return true;
	}

	@Override
	public String toString() {
		return "MaClasse [nom=" + nom + "]";
	}

	public static void test31() {
		List<MaClasse> maClasses = Arrays.asList(new MaClasse("aaa"), new
MaClasse("bbb"), new MaClasse("aaa"));
		maClasses.stream().distinct().forEach(System.out::println);

	}

	public static void main(String[] args) {
		test31();
	}
}
```

```
MaClasse [nom=aaa]
MaClasse [nom=bbb]
MaClasse [nom=aaa]
```

Si vous ajoutez la méthode hashCode() ci-dessous à l'exemple ci-dessus, vous aurez le bon résultat :

```java
package ma.formation.stream;

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

public class MaClasse {

	private String nom;

	public MaClasse(String nom) {
		this.nom = nom;
	}

	public String getNom() {
		return nom;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((nom == null) ? 0 : nom.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		MaClasse other = (MaClasse) obj;
		if (nom == null) {
			if (other.nom != null)
				return false;
		} else if (!nom.equals(other.nom))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "MaClasse [nom=" + nom + "]";
	}

	public static void test31() {
		List<MaClasse> maClasses = Arrays.asList(new MaClasse("aaa"), new
MaClasse("bbb"), new MaClasse("aaa"));
		maClasses.stream().distinct().forEach(System.out::println);

	}

	public static void main(String[] args) {
		test31();
	}
}
```

```
MaClasse [nom=aaa]
MaClasse [nom=bbb]
```

Cette opération intermédiaire est stateful : elle nécessite de stocker tous les éléments qui seront renvoyés : cela peut donc requérir une quantité de mémoire dépendante du nombre d'éléments traités par le Stream. Tous les éléments du Stream en entrée doivent être traités avant qu'un premier élément ne soit envoyés dans le Stream de retour.

Si le Stream est ordonné, l'élément conservé parmi les doublons est toujours le premier. Si le Stream n'est pas ordonné, l'élément conservé parmi les doublons n'est pas prédictible.

L'utilisation d'un Stream non ordonné ou l'utilisation de la méthode unordered() peut améliorer les performances lors de l'utilisation de distinct() dans un Stream parallèle, sous réserve que le contexte le permette. Si l'ordre doit être préservé, il arrive qu'un Stream séquentiel puisse être plus performant qu'un Stream parallèle équivalent.

e. La méthode limit()

L'opération intermédiaire limit() permet de limiter le nombre d'éléments contenu dans le Stream retourné. Le nombre d'éléments est passé en paramètre de la méthode limit().

C'est une opération de type short-circuit : dès que le nombre d'éléments contenus dans le Stream retourné est atteint, l'opération met fin à la consommation d'éléments de la source par le Stream.

```
	public static void test32() {
		List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
	    entiers.stream()
			.limit(3)
			.forEach(System.out::println);
	  }
```

f. La méthode skip()

L'opération skip() permet d'ignorer un certain nombre d'éléments du Stream. Les premiers éléments du Stream sont ignorés, les autres sont ajoutés dans le Stream retourné par la fonction. Si le nombre d'éléments fournis en paramètre est supérieur ou égal au nombre d'éléments du Stream, alors le Stream retourné est vide.

La méthode skip() attend en paramètre une valeur entière qui correspond au nombre d'éléments à ignorer.

```
public static void test33() {
		List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
	    entiers.stream()
			.skip(3)
			.forEach(System.out::println);
	}
```

```
4
5
6
```

g. La méthode sorted()

L'opération sorted() permet de trier les éléments du Stream : elle renvoie donc un Stream dont les éléments sont triés. Elle utilise un tampon qui doit avoir tous les éléments avant de pouvoir effectuer le tri.

Par défaut, la méthode sorted() sans paramètre utilise l'ordre naturel des éléments.

```
public static void test34() {
List<String> prenoms = Arrays.asList("kadiri", "ahmadi", "zalzouli", "benoma",
"mohammadi");
prenoms.stream().sorted().forEach(System.out::println);
	}
```

Une surcharge de la méthode sorted() attend en paramètre un Comparator qui sera utilisé pour comparer les éléments deux à deux lors du tri.

```
public static void test35() {
List<String> prenoms = Arrays.asList("kadiri", "ahmadi", "zalzouli", "benoma",
"mohammadi");
prenoms.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println);
}
```

```
zalzouli
mohammadi
kadiri
benoma
ahmadi
```

Il est possible de construire des Comparator évolués en utilisant les méthodes static et par défaut de l'interface Comparator.

```
public static void test36() {
List<String> prenoms = Arrays.asList("kadiri", "ahmadi", "zalzouli", "benoma",
"mohammadi");
prenoms.stream()
		.sorted(Comparator.comparingInt(String::length)
		.thenComparing(Comparator.naturalOrder())
		.reversed())
		.forEach(System.out::println);
	}
```

```
mohammadi
zalzouli
kadiri
benoma
ahmadi
```

h. La méthode peek()

L'opération peek() renvoie un Stream contenant tous les éléments du Stream courant en appliquant le Consumer fournit en paramètre sur chacun des éléments.

Le but de cette opération **est essentiellement le débogage**, par exemple pour afficher l'élément en cours de traitement.

```
public static void test37() {
            List<String> prenoms = Arrays.asList("kadiri", "ahmadi", "zalzouli",
"alali", "mohammadi");
                        prenoms.stream()
                                .map(p -> {
                                  afficher("map    ", p);
                                  return p.toUpperCase();
                                })
                                .filter(p -> {
                                   afficher("filter ", p);
                                   return p.startsWith("A");
                                })
                                .peek(p -> {
                                  afficher("limit ", p);
                                })
                                .limit(2)
                                .forEach(System.out::println);
            }
```

```
map    : kadiri
filter  : KADIRI
map    : ahmadi
filter  : AHMADI
limit  : AHMADI
AHMADI
map    : zalzouli
filter  : ZALZOULI
map    : alali
filter  : ALALI
limit  : ALALI
ALALI
```

i. Les méthodes qui modifient le comportement du Stream

Plusieurs méthodes sont des opérations intermédiaires qui modifient certaines caractéristiques du Stream et donc peuvent avoir un impact sur la manière dont le Stream va traiter les données.

La méthode sequential() permet de demander l'exécution des traitements du Stream en séquentiel dans un seul thread.

La méthode parallel() permet de demander l'exécution des traitements du Stream en parallèle en utilisant plusieurs threads du pool Fork/Join.

Les méthodes sequential() et parallel() sont des opérations intermédiaires : elles permettent donc simplement de configurer les traitements qui seront effectivement exécutés par l'invocation de la

méthode terminale. Les traitements ainsi exécutés ne peuvent l'être dans leur intégralité qu'en séquentiel ou en parallèle même si les méthodes sequential() et parallel() sont invoqués dans le Stream.

```
public static void test38() {
             Stream.of(9, 3, 4, 2)
         .parallel()
         .sorted()
         .peek((v) -> System.out.println(Thread.currentThread()
             .getName() + " " + v))
         .sequential()
         .forEach((v) -> System.out.println(Thread.currentThread()
            .getName() + " " + v));
      }
```

```
main 2
main 2
main 3
main 3
main 4
main 4
main 9
main 9
```

C'est l'état indiqué par la dernière méthode sequential() ou parallel() qui sera utilisée pour déterminer le mode d'exécution des traitements du Stream.

```
public static void test39() {
             Stream.of(1, 3, 4, 2)
         .sequential()
         .sorted()
         .peek((v) -> System.out.println(Thread.currentThread()
             .getName() + " premier " + v))
         .parallel()
         .forEach((v) -> System.out.println(Thread.currentThread()
             .getName() + " " + v));

      }
```

```
ForkJoinPool.commonPool-worker-3 premier 1
ForkJoinPool.commonPool-worker-3 1
ForkJoinPool.commonPool-worker-2 premier 4
ForkJoinPool.commonPool-worker-2 4
main premier 3
main 3
ForkJoinPool.commonPool-worker-1 premier 2
ForkJoinPool.commonPool-worker-1 2
```

7. Les opérations terminales

Les traitements d'un Stream sont démarrés à l'invocation de son unique opération terminale.

Une seule opération terminale ne peut être invoquée sur un même Stream : une fois cela fait, le Stream ne sera plus utilisable.

Il n'est pas possible de réutiliser un Stream une fois que son opération terminale est invoquée. Le Stream est alors considéré comme consommé. Il faut obligatoirement obtenir un nouveau Stream à partir de la source pour chaque traitement. Il n'est possible que d'invoquer une seule fois un pipeline d'opérations sur un Stream sinon une exception de type IllegalStateException est levée.

```
public static void test40() {

	String[] fruits = { "orange", "citron", "pamplemousse", "banane","fraise",
			"groseille", "raisin", "pomme", "poire", "abricot", "cerise",
"peche", "clementine" };

		Stream<String> stream = Stream.of(fruits);
		stream.filter(s -> s.startsWith("p"))
			.forEach(System.out::println);
		try {
		  stream.filter(s -> s.startsWith("c"))
			.forEach(System.out::println);
		} catch (IllegalStateException e) {
		  e.printStackTrace(System.out);
		}


	}
```

```
pamplemousse
pomme
poire
peche
java.lang.IllegalStateException: stream has already been operated upon or closed
	at java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)
	at java.util.stream.ReferencePipeline.<init>(ReferencePipeline.java:94)
	at java.util.stream.ReferencePipeline$StatelessOp.<init>(ReferencePipeline.java:618)
	at java.util.stream.ReferencePipeline$2.<init>(ReferencePipeline.java:163)
	at java.util.stream.ReferencePipeline.filter(ReferencePipeline.java:162)
	at ma.formation.stream.Test1.test40(Test1.java:349)
	at ma.formation.stream.Test1.main(Test1.java:358)
```

Cela ne pose pas de soucis pour créer un nouveau Stream à chaque fois puisque le Stream pointe simplement sur sa source de données mais ne les copie pas. Il est par exemple possible d'utiliser un Supplier pour créer des instances d'un Stream avec les mêmes éléments.

```
public static void test41() {
		String[] fruits = { "orange", "citron", "pamplemousse", "banane","fraise",
				"groseille", "raisin", "pomme", "poire", "abricot", "cerise",
"peche", "clementine" };
		 Supplier<Stream<String>> streamSupplier = () -> Stream.of(fruits);
```

```
                streamSupplier.get()
                              .filter(s -> s.startsWith("p"))
                              .forEach(System.out::println);
                streamSupplier.get()
                              .filter(s -> s.startsWith("c"))
                              .forEach(System.out::println);

}
```

a. Les méthodes forEach() et forEachOrdered()

L'API Stream propose deux opérations pour exécuter des traitements sur les éléments du résultat de l'exécution des opérations intermédiaires :

- ✓ void forEach(Consumer<? super T> action) : cette opération exécute l'action passée en paramètre sur chacun des éléments du Stream. Le comportement de la méthode forEach() sur un Stream parallèle n'est pas déterministe : elle ne garantit pas dans ce cas le respect de l'ordre des éléments puisque ceux-ci sont traités par différents threads.
- ✓ void forEachOrdered(Consumer<? super T> action) : cette opération est similaire à l'opération forEach() mais elle garantit l'ordre des éléments du Stream. Pour cela, elle n'utilise qu'un seul thread pour exécuter l'action sur tous les éléments. C'est l'API qui détermine quel unique thread exécute l'action sur chacun des éléments

Ces deux méthodes attendent en paramètre un java.util.function.Consumer qui contient l'action à réaliser sur chacun des éléments.

Remarque : il n'est pas possible d'utiliser les instructions break ou return dans l'expression lambda pour interrompre l'itération sur les éléments.

L'exemple ci-dessous illustre les différences lors de l'utilisation de ces méthodes :

```
public static void test42() {
               List<String> prenoms = Arrays.asList("ali", "samir", "mohammed", "karim",
"zoubir");
                  Consumer<String> afficherElement = s -> System.out.println(s + " - " +
                    Thread.currentThread().getName());
                  prenoms.stream().sorted().forEach(afficherElement);
                  System.out.println();
                  prenoms.parallelStream().sorted().forEach(afficherElement);
                  System.out.println();
                  prenoms.parallelStream().sorted().forEachOrdered(afficherElement);
                }
```

```
ali - main
karim - main
mohammed - main
samir - main
zoubir - main

mohammed - main
ali - ForkJoinPool.commonPool-worker-3
samir - main
zoubir - ForkJoinPool.commonPool-worker-2
karim - ForkJoinPool.commonPool-worker-1

ali - ForkJoinPool.commonPool-worker-2
karim - ForkJoinPool.commonPool-worker-3
mohammed - ForkJoinPool.commonPool-worker-3
samir - ForkJoinPool.commonPool-worker-3
zoubir - ForkJoinPool.commonPool-worker-3
```

Lors de l'invocation de la méthode forEach() sur un Stream séquentiel, l'ordre des éléments n'est préservé. Bien que la méthode forEach() ne respecte aucun ordre, le parcours d'un Stream séquentiel se fait dans un unique thread ce qui permet de conserver l'ordre des éléments.

Lors de l'invocation de la méthode forEach() sur un Stream parallèle, l'ordre des éléments n'est pas préservé. Ceci est lié au fait de l'utilisation de plusieurs threads pour exécuter l'action sur les différents éléments.

Pour respecter l'ordre des éléments dans un Stream parallèle, il est nécessaire d'utiliser la méthode forEachOrdered() qui va réaliser l'exécution de l'action dans un thread unique. Evidemment les performances de la méthode forEachOrdered() sont moins bonnes que celles de la méthode forEach().

Il faut éviter d'utiliser ces méthodes. C'est particulièrement vrai si celles-ci effectuent des traitements qui modifient une donnée.

```java
package ma.formation.stream;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.DoubleAdder;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

public class TestEmployeeStream {

	public static void test43() {
		List<Employe> employes = new ArrayList<>(6);
		employes.add(new Employe("e1", Genre.HOMME, 1000d));
		employes.add(new Employe("e2", Genre.HOMME, 2000d));
		employes.add(new Employe("e3", Genre.FEMME, 3000d));
		employes.add(new Employe("e4", Genre.FEMME, 4000d));
		employes.add(new Employe("e5", Genre.HOMME, 1000d));
		employes.add(new Employe("e6", Genre.FEMME, 5000d));
```

```
		DoubleAdder total = new DoubleAdder();
		employes.stream().forEach(e -> total.add(e.getSalaire()));

		System.out.println("total salaire = " + total.doubleValue());
	}

	public static void main(String[] args) {
		test43();
	}

}

@NoArgsConstructor
@AllArgsConstructor
@Data
class Employe {
	String nom;
	Genre genre;
	Double salaire;
}
```

Attention : lors de l'utilisation sur un Stream parallèle, si l'action d'une opération forEach() modifie une ressource partagée, il est nécessaire de gérer explicitement les accès concurrents.

Cette approche n'est cependant pas fonctionnelle : il préférable d'utiliser une approche de type map / reduce. L'interface DoubleStream propose d'ailleurs une méthode sum() qui effectue ce traitement.

```
public static void test44() {
		List<Employe> employes = new ArrayList<>(6);
		employes.add(new Employe("e1", Genre.HOMME, 1000d));
		employes.add(new Employe("e2", Genre.HOMME, 2000d));
		employes.add(new Employe("e3", Genre.FEMME, 3000d));
		employes.add(new Employe("e4", Genre.FEMME, 4000d));
		employes.add(new Employe("e5", Genre.HOMME, 1000d));
		employes.add(new Employe("e6", Genre.FEMME, 5000d));

		Double total = employes.stream().mapToDouble(e -> e.getSalaire()).sum();

		System.out.println("total salaire = " + total);
	}
```

Ces méthodes impliquent nécessairement des effets de bords puisqu'elle ne renvoie rien (void).

Pour respecter une approche fonctionnelle, il est préférable d'éviter de modifier la source de données ou ses éléments.

```
public static void test45() {
		List<Employe> employes = new ArrayList<>(6);
		employes.add(new Employe("e1", Genre.HOMME, 1000d));
		employes.add(new Employe("e2", Genre.HOMME, 2000d));
		employes.add(new Employe("e3", Genre.FEMME, 3000d));
		employes.add(new Employe("e4", Genre.FEMME, 4000d));
		employes.add(new Employe("e5", Genre.HOMME, 1000d));
```

```
            employes.add(new Employe("e6", Genre.FEMME, 5000d));

employes.stream().forEach(e -> e.setSalaire(e.getSalaire() + (e.getSalaire() * 0.1)));
employes.stream().forEach(System.out::println);
}
```

Une meilleure approche dans ce cas, pourrait être de ne pas utiliser un Stream mais d'utiliser la méthode forEach() de la collection.

```
public static void test46() {
            List<Employe> employes = new ArrayList<>(6);
            employes.add(new Employe("e1", Genre.HOMME, 1000d));
            employes.add(new Employe("e2", Genre.HOMME, 2000d));
            employes.add(new Employe("e3", Genre.FEMME, 3000d));
            employes.add(new Employe("e4", Genre.FEMME, 4000d));
            employes.add(new Employe("e5", Genre.HOMME, 1000d));
            employes.add(new Employe("e6", Genre.FEMME, 5000d));
            employes.forEach(e -> {
                  e.setSalaire(e.getSalaire() + (e.getSalaire() * 0.1));
                  System.out.println(e);
            });
      }
```

Le résultat de l'exécution est le même que pour l'exemple précédent.

Les méthodes forEach() et forEachOrdered() sont des opérations terminales : une fois leur exécution terminée, le Stream est consommé et ne peut plus être utilisé. Il n'est donc pas possible d'invoquer deux fois ces méthodes sur un même Stream.

```
public static void test47() {
            List<Employe> employes = new ArrayList<>(6);
            employes.add(new Employe("e1", Genre.HOMME, 1000d));
            employes.add(new Employe("e2", Genre.HOMME, 2000d));
            employes.add(new Employe("e3", Genre.FEMME, 3000d));
            employes.add(new Employe("e4", Genre.FEMME, 4000d));
            employes.add(new Employe("e5", Genre.HOMME, 1000d));
            employes.add(new Employe("e6", Genre.FEMME, 5000d));

            Stream<Employe> stream=employes.stream();

            stream.forEach(e -> e.setSalaire(e.getSalaire() + (e.getSalaire() * 0.1)));
            stream.forEach(System.out::println);
      }
```

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
        at java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
        at java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:580)
        at ma.formation.stream.TestEmployeeStream.test47(TestEmployeeStream.java:83)
        at ma.formation.stream.TestEmployeeStream.main(TestEmployeeStream.java:87)
```

b. La méthode collect()

La méthode collect() de l'interface Stream est une opération terminale qui permet de réaliser une agrégation mutable des éléments. Cette agrégation peut prendre différentes formes : stocker les éléments dans une structure de données telle qu'une collection, concaténer les éléments qui sont des chaînes de caractères, ...

L'opération collect() permet de transformer les éléments d'un Stream pour produire un résultat sous différentes formes telles que List, Set, Map, ... ou plus généralement sous la forme d'une réduction dans un conteneur mutable.

Méthode	Rôle
<R,A> R collect(Collector<? super T,A,R> collector)	Effectuer une opération de réduction mutable sur les éléments du Stream en utilisant le Collector fourni
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	Effectuer une opération de réduction mutable sur les éléments du Stream en utilisant les fonctions fournies en paramètres

Les traitements réalisés par la méthode collect() peuvent être exécutés en parallèle sur plusieurs threads dans un Stream parallèle car chacun d'eux va utiliser sa propre instance du conteneur et la méthode de combinaison permet de fusionner le contenu des différents conteneurs.

La surcharge <R> collect(Supplier<R> resultSupplier, BiConsumer<R, T> accumulator, BiConsumer<R, R> combiner) de la méthode collect() attend trois paramètres :

- ✓ supplier : pour fournir une instance du conteneur vide.
- ✓ accumulator : pour ajouter un élément dans le conteneur.
- ✓ combiner : pour combiner deux conteneurs lors de l'utilisation du Stream en parallèle.

L'exemple ci-dessous utilise la méthode collect() pour mettre tous les éléments du Stream dans une collection de type HashSet grâce aux trois fonctions fournies en paramètre :

- ✓ supplier : renvoie une nouvelle instance de type HashSet
- ✓ accumulator : ajoute l'élément dans la collection
- ✓ combiner : ajoute tous les éléments d'un HashSet dans l'autre

```
public static void test48() {

List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
Set<String> ensemble = elements.stream()
                        .collect(() -> new HashSet<String>(),
                        (s, e) -> s.add(e),
```

```
                        (s1, s2) -> s1.addAll(s2));
System.out.println(ensemble);
}
```

```
[elem2, elem1, elem4, elem3]
```

Comme les expressions correspondent toutes à l'invocation d'une seule méthode, il est possible d'utiliser des références de méthodes pour obtenir le même résultat.

```
public static void test49() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4")
Set<String> ensemble =
elements.stream().collect(HashSet::new,HashSet::add,HashSet::addAll);
System.out.println(ensemble);
}
```

La mise en œuvre de ces deux exemples est préférable à l'exemple ci-dessous.

```
public static void test50() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
// NE PAS FAIRE COMME CELA
Set<String> ensemble = new HashSet<>();
elements.stream().forEach(s -> ensemble.add(s));
System.out.println(ensemble);
}
```

Cet exemple fonctionne mais c'est un antipattern. Il fonctionne correctement uniquement en mono thread.

La méthode collect() peut aussi être utilisée pour effectuer une opération de réduction qui va permettre de combiner deux éléments pour obtenir un nouvel élément qui sera combiné à son tour à l'élément suivant et ainsi de suite jusqu'à ce que tous les éléments soient traités.

L'exemple ci-dessous concatène les éléments qui sont de type chaîne de caractères.

```
public static void test51() {
      List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
                StringBuilder elmtSb =
                  elements.stream()
                          .collect(() -> new StringBuilder(),
                                   (sb, s) -> sb.append(s),
                                   (sb1, sb2) -> sb1.append(sb2));
                System.out.println(elmtSb.toString());
      }
```

```
elem1elem2elem2elem3elem4
```

Comme chaque fonction est l'invocation d'une seule méthode, il est possible d'utiliser des références de méthodes.

```
public static void test52() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3", "elem4");
StringBuilder elmtSb =  elements.stream().collect(StringBuilder::new,
StringBuilder::append,StringBuilder::append);
System.out.println(elmtSb.toString());

	}
```

Fréquemment, il est nécessaire d'ajouter un séparateur entre chacun des éléments concaténés.

```
public static void test53() {
		List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3",
"elem4");
		StringBuilder concat =
		  elements.stream()
				.collect(() -> new StringBuilder(), (sb, s) -> {
					if (sb.length() != 0) {
					  sb.append(";");
					}
					  sb.append(s);
					} , (sb1, sb2) -> sb1.append(sb2));
		System.out.println(concat.toString());

	}
```

```
elem1;elem2;elem2;elem3;elem4
```

Cette solution utilise un bloc code ce qui la rend plus complexe et empêche l'utilisation d'une référence de méthode. Pour simplifier le code, il est possible d'utiliser la classe StringJoiner introduite dans Java 8.

```
public static void test54() {
		List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3",
"elem4");
	  StringJoiner elmtJoiner =
		elements.stream()
			.collect(() -> new StringJoiner(";"),
				(j, e) -> j.add(e),
				(j1, j2) -> j1.merge(j2));
	  System.out.println(elmtJoiner.toString());
	}
```

c. Les méthodes findFirst() et findAny(

L'interface Stream possède les méthodes findFirst() et findAny() pour obtenir un élément du Stream qui respecte le Predicate qui leur est fourni en paramètre.

Les méthodes findFirst() et findAny() sont des opérations terminales de type short-circuiting qui renvoient une instance de type Optional car il est possible que le Stream ne possède aucun élément et dans ce cas l'instance retournée encapsule l'absence de valeur.

La méthode findFirst() renvoie un objet de type Optional qui encapsule le premier élément dont le Predicate fourni en paramètre est validé.

Si aucun élément n'est trouvé, l'Optional retourné est vide.

```
public static void test55() {
		List<Employe> employees = new ArrayList<>(6);
		employees.add(new Employe("p1", Genre.HOMME, 10000d));
		employees.add(new Employe("p2", Genre.HOMME, 15000d));
		employees.add(new Employe("p3", Genre.FEMME, 100000d));
		employees.add(new Employe("p4", Genre.FEMME, 1000d));
		employees.add(new Employe("p5", Genre.HOMME, 2000d));
		employees.add(new Employe("p6", Genre.FEMME, 5000d));

		Optional<Employe> unGrandSalaire = employees.stream().filter(p ->
p.getSalaire() >= 100000).findFirst();

		if (unGrandSalaire.isPresent()) {
			System.out.println("Employé avec grand salaire trouvee : " +
unGrandSalaire);
		} else {
			System.out.println("Aucune Employé avec grand salaire trouvee");
		}
	}
```

```
Employé avec grand salaire trouvee : Optional[Employe(nom=p3, genre=FEMME, salaire=100000.0)]
```

L'opération findAny() renvoie n'importe quel élément du Stream dont le Predicate fourni en paramètre est validé.

L'utilisation de l'une ou l'autre de ces méthodes requière une attention particulière sur un Stream en parallèle.

La méthode findAny() est plus performante que la méthode findFirst() notamment lorsque le Stream est traité en mode parallèle.

d. Les méthodes xxxMatch()

L'API propose plusieurs méthodes xxxMatch() pour déterminer si aucun, au moins un ou tous les éléments respectent une certaine condition.

La méthode anyMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si au moins un élément du Stream respecte le Predicate.

```
public static void test56() {
		Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
	boolean auMoinsUnEgalATrois = entiers.anyMatch(e -> e == 3);
	System.out.println(auMoinsUnEgalATrois);
}
```

La méthode allMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si tous les éléments du Stream respectent le Predicate.

```
public static void test57() {
          Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
          boolean tousInferieursADix = entiers.allMatch(e -> e < 10);
          System.out.println(tousInferieursADix);
}
```

La méthode noneMatch(Predicate<? super T> predicate) renvoie un booléen qui précise si aucun élément du Stream respecte le Predicate.

```
public static void test58() {
          Stream<Integer> entiers = Stream.of(1, 2, 3, 4, 5);
          boolean tousDifferentsDeDix = entiers.noneMatch(e -> e == 10);
          System.out.println(tousDifferentsDeDix);
}
```

e. La méthode count()

La méthode count() renvoie un entier long qui est le nombre d'éléments contenu dans le Stream.

```
public static void test59() {
          Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5);
         System.out.println("Nb elements=" + nombres.count());
}
```

La méthode count() est une opération terminale donc elle effectue ces traitements sur le Stream issu de l'exécution des opérations intermédiaires.

```
public static void test60() {
Stream<Integer> nombres = Stream.of(1, 2, 3, 4, 5).filter(e -> (e % 2) ==
0);
System.out.println("Nb elements=" + nombres.count());
}
```

f. La méthode reduce

La réduction, aussi appelée folding en programmation fonctionnelle, permet de réaliser une opération d'agrégation sur un ensemble d'éléments afin de produire un résultat. Une opération de réduction applique de manière répétée une opération sur chacun des éléments pour les combiner afin de produire un unique résultat.

Par exemple, pour des éléments de type entier, si l'opérateur binaire fait une addition alors l'opération de réduction calcule la somme des éléments. Mais les traitements d'une réduction ne se limite pas à ce type de traitements. Une opération de réduction peut aussi être la recherche de la plus grande valeur en utilisant une expression lambda (x,y) -> Math.max(x,y) ou avec la référence de méthode équivalente Math::max.

Une réduction peut aussi s'appliquer sur tout type d'objet : dans ce cas, il est nécessaire de préciser le comportement des traitements de l'opération d'agrégation.

La méthode reduce() effectue une opération de type réduction. L'interface Stream<T> possède trois surcharges de la méthode reduce() :

Méthode	Rôle
Optional<T> reduce(BinaryOperator<T> accumulator)	Effectue la réduction des éléments du Stream en appliquant la fonction fournie en paramètre. Elle renvoie un Optional qui encapsule la valeur ou l'absence de valeur
T reduce(T identity, BinaryOperator<T> accumulator)	Effectue la réduction des éléments du Stream en appliquant la fonction fournie en paramètre à partir de la valeur fournie en premier paramètre
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	Effectue la réduction des éléments du Stream en appliquant la fonction d'accumulation et de combinaison fournies en paramètre à partir de la valeur fournie en premier paramètre

La surcharge qui attend en paramètre un BinaryOperator<T> renvoie comme résultat au plus un élément encapsulé dans un Optional. Elle attend en paramètre un opérateur binaire associatif qui effectue la réduction des éléments du Stream. Elle renvoie un Optional qui est vide si le Stream ne contient aucun élément. Si le Stream ne possède qu'un seul élément, alors c'est cet élément qui est retourné.

```
public static void test61() {
List<Emp> employes = Arrays.asList(
new Emp("emp_1", 33, Genre.HOMME),
new Emp("emp_2", 43, Genre.HOMME),
new Emp("emp_3", 52, Genre.HOMME),
new Emp("emp_4", 52, Genre.HOMME),
new Emp("emp_5", 50, Genre.HOMME),
new Emp("emp_6", 49, Genre.FEMME),
new Emp("emp_7", 41, Genre.FEMME),
new Emp("emp_8", 39, Genre.FEMME),
new Emp("emp_9", 37, Genre.FEMME),
new Emp("emp_10", 25, Genre.FEMME));

employes.stream().reduce((p1, p2) -> p1.getAge() > p2.getAge() ? p1 :
```

```
p2).ifPresent(System.out::println);
      }
```

L'exemple ci-dessus permet de déterminer l'employé ayant l'âge le plus grand.

La fonction de type BinaryOperator<T> attend en paramètres deux objets de type T et renvoie un objet de T. Le premier paramètre est la valeur accumulée et le second paramètre est l'élément courant en cours de traitement dans le Stream.

```
public static void test62() {
String lettres = Stream.of("a", "b", "c", "d").reduce((accumulator, item)
-> accumulator + ", " + item)
                        .orElse("");
           System.out.println(lettres);

}
```

```
<terminated> Test1 (3) [Java Application] C
a, b, c, d
```

L'opération de réduction peut utiliser un des BinaryOperator fournis par le JDK. L'exemple ci-dessous recherche l'employée la plus grande.

```
public static void test63() {
List<Emp> employes = Arrays.asList(
new Emp("emp_1", 33, Genre.HOMME),
new Emp("emp_2", 43, Genre.HOMME),
new Emp("emp_3", 52, Genre.HOMME),
new Emp("emp_4", 52, Genre.HOMME),
new Emp("emp_5", 50, Genre.HOMME),
new Emp("emp_6", 49, Genre.FEMME),
new Emp("emp_7", 41, Genre.FEMME),
new Emp("emp_8", 39, Genre.FEMME),
new Emp("emp_9", 37, Genre.FEMME),
new Emp("emp_10", 25, Genre.FEMME));

Comparator<Emp> comparateurParAge = Comparator.comparingInt(Emp::getAge);
BinaryOperator<Emp> lePlusGrand = BinaryOperator.maxBy(comparateurParAge);
Optional<Emp> plusGrandEmploye = employes.stream().reduce(lePlusGrand);
System.out.println(plusGrandEmploye);
}
```

La seconde surcharge de la méthode reduce() attend en paramètre la valeur initiale et une fonction de type BinaryOperator qui sera invoquée pour effectuer la réduction.

```
      public static void test64() {
```

```
            List<Integer> entiers = Arrays.asList(1, 2, 3, 4, 5, 6);
        Integer total = entiers.stream()
                                .reduce(0, (valeurAccumulee, valeur) ->
valeurAccumulee + valeur);
        System.out.println("total=" + total);
}
```

a fonction de type BinaryOperator peut être fournie sous la forme d'une référence de méthode. L'exemple ci-dessous détermine la valeur maximum d'éléments de type Integer.

```
public static void test65() {
            Integer max = Stream.of(1, 2, 3, 4, 5).reduce(0, Integer::max);
            System.out.println(max);
}
```

Si le Stream ne contient aucun élément, alors c'est la valeur initiale qui est renvoyée.

```
public static void test66() {
            String lettres = Stream.of("a", "b", "c",
"d").limit(0).reduce("X",
                        (accumulator, item) -> accumulator + ", " + item);
            System.out.println(lettres);
}
```

La troisième surcharge de la méthode reduce() attend en paramètre la valeur initiale, une BiFunction qui sera utilisée comme un accumulateur et une BinaryOperator qui sera utilisée comme combinateur.

Cette opération peut être définie explicitement de manière différente au moyen de deux opérations de type map() et reduce().

L'exemple ci-dessous calcule la taille des employés. Ce traitement aurait pu être réalisé différemment par le Stream mais il est utilisé pour illustrer le mode de fonctionnement de la méthode reduce().

```
public static void test67() {
            List<Emp> employes = Arrays.asList(
                        new Emp("emp_1", 33, Genre.HOMME),
                        new Emp("emp_2", 43, Genre.HOMME),
                        new Emp("emp_3", 52, Genre.HOMME),
                        new Emp("emp_4", 52, Genre.HOMME),
                        new Emp("emp_5", 50, Genre.HOMME),
                        new Emp("emp_6", 49, Genre.FEMME),
                        new Emp("emp_7", 41, Genre.FEMME),
                        new Emp("emp_8", 39, Genre.FEMME),
                        new Emp("emp_9", 37, Genre.FEMME),
                        new Emp("emp_10",25, Genre.FEMME));
```

```
            Integer ageTotal = employes.stream().reduce(0, (somme, e) ->
somme += e.getAge(),
                        (somme1, somme2) -> somme1 + somme2);
            System.out.println("Age total = " + ageTotal);
      }
```

Le traitement exécuté est similaire à celui ci-dessous.

```
public static void test68() {
            List<Emp> employes = Arrays.asList(
                        new Emp("emp_1", 33, Genre.HOMME),
                        new Emp("emp_2", 43, Genre.HOMME),
                        new Emp("emp_3", 52, Genre.HOMME),
                        new Emp("emp_4", 52, Genre.HOMME),
                        new Emp("emp_5", 50, Genre.HOMME),
                        new Emp("emp_6", 49, Genre.FEMME),
                        new Emp("emp_7", 41, Genre.FEMME),
                        new Emp("emp_8", 39, Genre.FEMME),
                        new Emp("emp_9", 37, Genre.FEMME),
                        new Emp("emp_10",25, Genre.FEMME));

            BiFunction<Integer, Emp, Integer> accumulator = (somme, e) ->
somme += e.getAge();
           Integer resultat = 0;
           for (Emp e : employes) {
             resultat = accumulator.apply(resultat, e);
           }
         System.out.println("Taille totale = " + resultat);
}
```

Pour comprendre le mode de fonction de la méthode reduce() et l'utilisation qu'elle fait des fonctions d'accumulation et de combinaison, il faut ajouter des traces dans leurs expressions Lambdas.

```
public static void test69() {

            List<Emp> employes = Arrays.asList(
                        new Emp("emp_1", 33, Genre.HOMME),
                        new Emp("emp_2", 43, Genre.HOMME),
                        new Emp("emp_3", 52, Genre.HOMME),
                        new Emp("emp_4", 52, Genre.HOMME),
                        new Emp("emp_5", 50, Genre.HOMME),
                        new Emp("emp_6", 49, Genre.FEMME),
                        new Emp("emp_7", 41, Genre.FEMME),
                        new Emp("emp_8", 39, Genre.FEMME),
                        new Emp("emp_9", 37, Genre.FEMME),
                        new Emp("emp_10",25, Genre.FEMME));

Integer ageTotal = employes.stream().reduce(0, (somme, e) -> {
afficher("accumulator", " somme=" + somme + " e=" + e.getAge());
```

```
                return somme += e.getAge();
            } , (somme1, somme2) -> {
                afficher("combiner", " somme1=" + somme1 + " somme2=" +
somme2);
                return somme1 + somme2;
            });
            System.out.println("Age total = " + ageTotal);

    }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (25 jan. 2022 à 15:41:25)
accumulator :  somme=0 e=33
accumulator :  somme=33 e=43
accumulator :  somme=76 e=52
accumulator :  somme=128 e=52
accumulator :  somme=180 e=50
accumulator :  somme=230 e=49
accumulator :  somme=279 e=41
accumulator :  somme=320 e=39
accumulator :  somme=359 e=37
accumulator :  somme=396 e=25
Age total = 421
```

La fonction de combinaison n'est jamais invoquée lorsque le Stream est exécuté en séquentiel. Le comportement est différent lorsque le Stream est invoqué en parallèle.

```
public static void test70() {
            List<Emp> employes = Arrays.asList(
                        new Emp("emp_1", 33, Genre.HOMME),
                        new Emp("emp_2", 43, Genre.HOMME),
                        new Emp("emp_3", 52, Genre.HOMME),
                        new Emp("emp_4", 52, Genre.HOMME),
                        new Emp("emp_5", 50, Genre.HOMME),
                        new Emp("emp_6", 49, Genre.FEMME),
                        new Emp("emp_7", 41, Genre.FEMME),
                        new Emp("emp_8", 39, Genre.FEMME),
                        new Emp("emp_9", 37, Genre.FEMME),
                        new Emp("emp_10",25, Genre.FEMME));

             Integer ageTotal = employes.parallelStream().reduce(0, (somme,
e) -> {
                afficher("accumulator", " somme=" + somme + " e=" +
e.getAge());
                return somme += e.getAge();
            } , (somme1, somme2) -> {
                afficher("combiner", " somme1=" + somme1 + " somme2=" +
somme2);
                return somme1 + somme2;
```

```
                });
                System.out.println("Age total = " + ageTotal);

    }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (25 jan. 2022 à 15:44:30)
accumulator :  somme=0 e=52
accumulator :  somme=0 e=37
accumulator :  somme=0 e=50
accumulator :  somme=0 e=41
accumulator :  somme=0 e=43
accumulator :  somme=0 e=33
accumulator :  somme=0 e=52
combiner :  somme1=33 somme2=43
accumulator :  somme=0 e=25
combiner :  somme1=37 somme2=25
accumulator :  somme=0 e=39
combiner :  somme1=52 somme2=50
accumulator :  somme=0 e=49
combiner :  somme1=49 somme2=41
combiner :  somme1=52 somme2=102
combiner :  somme1=39 somme2=62
combiner :  somme1=76 somme2=154
combiner :  somme1=90 somme2=101
combiner :  somme1=230 somme2=191
Age total = 421
```

Le comportement des traitements est différent : comme l'accumulateur est invoqué en parallèle, ce n'est plus lui qui fait la somme mais la fonction de combinaison.

Une opération de réduction est particulièrement intéressante lors du traitement du Stream en parallèle. Lors de l'exécution des traitements en parallèle, il est nécessaire de partager une variable qui stocke la valeur cumulée et de gérer les accès concurrents à cette variable par les différents threads. Généralement, cette gestion des accès concurrents inhibe voire dégrade les performances gagnées par la parallélisation.

Pour limiter cet impact, l'opération reduce() d'un Stream, traite et cumule les éléments dans chaque threads : ainsi, il n'y a pas de variable partagée et donc aucune contention liée aux accès concurrents.

Les résultats de chaque threads sont ensuite combinés pour obtenir le résultat final.

Pour permettre à une opération de réduction d'être exécutée en parallèle, il est nécessaire que celle-ci soit associative.

Une opération # est associative si (a # b) # c = a # (b # c).

L'addition et la multiplication sont des opérations associatives. La soustraction n'est pas une opération associative :

(3 - 2) -1 = 0

3 - (2 -1) = 1

Avec une opération associative, la réduction peut être effectuée dans n'importe quel ordre. Ceci est important pour une exécution en parallèle dans laquelle les éléments sont traités par lots, la réduction se faisant pour chacun des éléments des lots puis il y a une combinaison des résultats intermédiaires pour renvoyer le résultat de la réduction.

Si l'opération n'est pas associative, les résultats obtenus ne seront pas ceux escomptés.

```
public static void test71() {
OptionalInt res = IntStream.range(1, 100).parallel().reduce((a, b) -> a -
b);
System.out.println(res.getAsInt());

}
```

```
<terminated> Test1 (3) [Java Application] C:\
24
```

Alors, si on exécute le même traitement en séquentiel, on devrait avoir comme résultat :

```
public static void test72() {
OptionalInt res = IntStream.range(1, 100).reduce((a, b) -> a - b);
System.out.println(res.getAsInt());

}
```

```
<terminated> Test1 (3) [Java Applic
-4948
```

g. Les méthodes min() et max()

Les méthodes min() et max() permettent respectivement de retourner la valeur minimale et maximale issue des traitements du Stream.

Elles attendent en paramètre un Comparator qui permet de préciser l'ordre de comparaison des éléments.

Elles renvoient une instance de type Optional<T>

Le plus simple est d'utiliser la méthode comparing() de l'interface Comparator : elle renvoie un objet de type Comparator qui compare les clés extraites grâce à l'expression Lambda fournie en paramètre. Le paramètre de la méthode comparing() est une Function qui permet d'extraire la clé sur laquelle le Comparator retourné va faire la comparaison.

```
public static void test73() {
List<String> prenoms = Arrays.asList("samir", "mohammed",
"ali","abdellah", "sami", "imad");
           Optional<String> plusPetitPrenom =
prenoms.stream().min(Comparator.comparing(element -> element.length()));
           System.out.println(plusPetitPrenom.orElseGet(() -> "aucun
prenom trouve"));

      }
```

```
<terminated> Test1 (3) [Java Application
ali
```

h. La méthode toArray()

La méthode toArray() permet de renvoyer les éléments du Stream dans un tableau. Elle possède deux surcharges :

Méthode	Rôle
Object[] toArray()	Renvoyer un tableau contenant les éléments du Stream
<A> A[] toArray(IntFunction<A[]> generator)	Renvoyer un tableau contenant les éléments du Stream. Le tableau renvoyé est créé par la Function fournie en paramètre

Si le type du tableau n'est pas important, il est possible d'utiliser la méthode toArray() sans paramètre.

```
public static void test74() {
Stream<String> stream = Stream.of("a", "b", "c");
Object[] strings = stream.toArray();
System.out.println(Arrays.deepToString(strings));
}
```

La Function attendue en paramètre de la surcharge de la méthode toArray() attend en paramètre un entier qui est la taille du tableau et renvoie un tableau dont la taille est celle passée en paramètre. Elle permet de préciser le type du tableau à utiliser et qui sera renvoyé par la méthode.

```
public static void test75() {
           Stream<String> stream = Stream.of("a", "b", "c");
           String[] strings = stream.toArray(size -> new String[size]);
            System.out.println(Arrays.deepToString(strings));
      }
```

i. La méthode iterator

La méthode iterator() de l'interface BaseStream renvoie un Iterator<T> qui permet de parcourir dans une itération extérieure tous les éléments d'un Stream.

```
public static void test76() {
		Stream<String> stream = Stream.of("a", "b", "c");
		Iterator<String> it = stream.iterator();
		while (it.hasNext()) {
			String s = it.next();
			System.out.println(s);
		}
}
```

Attention, comme un Stream ne stocke pas ses éléments, il n'est possible de parcourir les éléments avec l'Iterator qu'une seule fois.

```
public static void test77() {
		Stream<String> stream = Stream.of("a", "b", "c");

		Iterator<String> it = stream.iterator();
		while (it.hasNext()) {
			String s = it.next();
			System.out.println(s);
		}

		it = stream.iterator();
		while (it.hasNext()) {
			String s = it.next();
			System.out.println(s);
		}

	}
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (25 jan. 2022 à 16:56:28)
a
b
c
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
	at java.util.stream.AbstractPipeline.spliterator(AbstractPipeline.java:343)
	at java.util.stream.ReferencePipeline.iterator(ReferencePipeline.java:139)
	at ma.formation.stream.Test1.test77(Test1.java:624)
	at ma.formation.stream.Test1.main(Test1.java:633)
```

8. Les Collectors

Une des surcharges de la méthode collect() attend en paramètre un objet de type Collector. Les traitements à appliquer sont alors définis par l'interface Collector.

La classe java.util.stream.Collectors propose un ensemble de fabriques qui renvoient des implémentations de Collector pour des opérations de réduction communes.

a. L'interface Collector

Un Collector permet de réaliser une opération de réduction qui accumule les éléments d'un Stream dans un conteneur mutable. Il peut éventuellement appliquer une transformation pour permettre de fournir le résultat final dans un type différent de celui du conteneur dans lequel les éléments sont accumulés. Les traitements des opérations de réduction peuvent être exécutés de manière séquentielle ou parallèle.

Les traitements d'un Collector sont définis grâce à 4 fonctions qui sont utilisées pour agréger les éléments du Stream dans un conteneur mutable, avec éventuellement une transformation optionnelle pour produire le résultat final :

- Un supplier : permet de renvoyer une nouvelle instance du conteneur mutable
- Un accumulator : accumule un élément dans le conteneur
- Un combiner : combine une instance du type du conteneur pour en produire un seul
- Un finisher : transforme le conteneur pour renvoyer une instance du type du résultat renvoyé (cette fonction est optionnelle)

L'interface Collector<T,A,R> possède trois types génériques :

- T : le type des éléments utilisé par l'opération de réduction
- A : le type de l'objet mutable utilisé par l'opération de réduction
- R : le type du résultat de l'opération de réduction

L'utilisation d'une implémentation d'un Collector dans des traitements séquentiels réalise basiquement plusieurs traitements :

- Utilise le Supplier pour créer une instance du conteneur
- Invoque la fonction accumulator pour chaque élément
- Invoque éventuellement la fonction finisher si nécessaire à la fin

L'utilisation d'une implémentation d'un Collector dans des traitements parallèles réalise basiquement plusieurs traitements :

- Utilise le Supplier pour créer une instance du conteneur pour chaque lot
- Invoque la fonction accumulator pour chaque élément du lot
- Invoque la fonction combiner pour fusionner les conteneurs de chaque lot
- Invoque éventuellement la fonction finisher si nécessaire à la fin

b. La classe Collectors

La classe Collectors propose des fabriques pour obtenir des instances de Collector qui réalisent des agrégations communes telles que l'ajout des éléments dans une collection, la concaténation de chaînes de caractères, des réductions, des calculs numériques et statistiques, des groupements, ...

1. Les fabriques pour des Collector vers des collections

```
public static void test78() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
List<String> resultats = elements.stream().collect(toList());
System.out.println(resultats);
}
```

```
public static void test79() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem2", "elem3",
"elem4");
Set<String> resultats = elements.stream().collect(toSet());
System.out.println(resultats);
}
```

Il n'y a aucune garantie sur l'instance de type Set retournée par le Collector de la méthode toSet().

Remarque : il n'est donc pas possible avec cette méthode de pouvoir préciser le type de l'implémentation de la collection. Pour cela, il faut utiliser la méthode toCollection().

La méthode toCollection() permet de renvoyer les éléments du Stream dans une collection dont l'implémentation est fournie par le Supplier.

```
     public static void test80() {
List<String> elements = Arrays.asList("elem4", "elem2", "elem2", "elem1",
"elem3");
Set<String> resultats =
elements.stream().collect(toCollection(TreeSet::new));
         System.out.println(resultats);
}
```

```
public static void test81() {
           List<String> elements = Arrays.asList("elem1", "elem2",
"elem3", "elem4");
         Map<String, Integer> resultats =
elements.stream().collect(toMap(Function.identity(),

String::length));
```

```
        System.out.println(resultats);
    }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bir
{elem2=5, elem1=5, elem4=5, elem3=5}
```

Dans l'exemple ci-dessus, comme les clés sont identiques, la fonction de fusion des doublons renvoie simplement le premier des deux. Evidemment, la fonction peut être adaptée selon les besoins, par exemple pour cumuler les valeurs des différents éléments de chaque clé.

```
public static void test82() {
      List<String> elements = Arrays.asList("elemxcc", "elem2", "elem2",
"elem3", "elem4");
        Map<Integer, String> resultats =
          elements.stream().collect(toMap(String::length,
Function.identity(), (s1, s2) -> s1+"*"+s2));
        System.out.println(resultats);

    }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\ja
{5=elem2*elem2*elem3*elem4, 7=elemxcc}
```

2. La fabrique pour des Collector qui exécutent une action complémentaire

La méthode collectingAndThen() permet d'exécuter une action supplémentaire sous la forme d'une fonction après l'exécution du Collector fourni en paramètre.

```
public static void test83() {
          List<String> elements = Arrays.asList("elem1", "elem2",
"elem3", "elem4");

          List<String> resultats =
elements.stream().collect(Collectors.collectingAndThen(toList(),
Collections::unmodifiableList));

System.out.println(resultats.getClass().getName());
//génére l'exception java.lang.UnsupportedOperationException
resultats.add("");
}
```

3. Les fabriques qui renvoient des Collector pour réaliser une agrégation

Les surcharges de la méthode joining() de la classe Collectors permettent de concaténer les éléments d'un Stream<String>.

La surcharge sans paramètre joining() renvoie un Collector qui permet de concaténer les éléments d'un Steam<String>

```
public static void test84() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
String resultat = elements.stream().collect(joining());
System.out.println(resultat);
}
```

La surcharge joining(CharSequence delimiter) renvoie un Collector qui permet de concaténer les éléments d'un Stream<String> avec le séparateur fourni en paramètre.

```
public static void test85() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
String resultat = elements.stream().collect(joining(","));
System.out.println(resultat);
}
```

La surcharge joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix) renvoie un Collector qui permet de concaténer les éléments d'une Steam<String> avec le séparateur, le préfixe et le suffixe fournis en paramètre.

```
public static void test86() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
String resultat = elements.stream().collect(joining(", ", "[", "]"));
System.out.println(resultat);
}
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_12
[elem1, elem2, elem3, elem4]
```

4. Les fabriques qui renvoient des Collectors pour effectuer des opérations numériques

La méthode counting() renvoie un Collector qui permet de compter le nombre d'éléments du Stream.

```
public static void test87() {
List<String> elements = Arrays.asList("elem1", "elem2", "elem3", "elem4");
Long resultat = elements.stream().collect(counting());
System.out.println(resultat);
```

```
}
```

Les méthodes summarizingInt(), summarizingLong() et summarizingDouble() de la classe Collectors renvoient des Collector qui calculent des informations statistiques basiques sur les données numériques extraites des éléments du Stream : le nombre d'éléments, les valeurs min et max, la moyenne et la somme.

Ces données sont respectivement encapsulées dans des objets de type IntSummaryStatistics, LongSummaryStatistics, et DoubleSummaryStatistics.

```
public static void test88() {

		List<Emp> employes = Arrays.asList(
				new Emp("emp_1", 33, Genre.HOMME),
				new Emp("emp_2", 43, Genre.HOMME),
				new Emp("emp_3", 52, Genre.HOMME),
				new Emp("emp_4", 52, Genre.HOMME),
				new Emp("emp_5", 50, Genre.HOMME),
				new Emp("emp_6", 49, Genre.FEMME),
				new Emp("emp_7", 41, Genre.FEMME),
				new Emp("emp_8", 39, Genre.FEMME),
				new Emp("emp_9", 37, Genre.FEMME),
				new Emp("emp_10", 25, Genre.FEMME));


		IntSummaryStatistics salaireSummary = employes
				.stream()
				.collect(Collectors.summarizingInt(Emp::getAge));
			System.out.println(salaireSummary);
			System.out.println(salaireSummary.getCount());
			System.out.println(salaireSummary.getMin());
			System.out.println(salaireSummary.getMax());
			System.out.println(salaireSummary.getAverage());
			System.out.println(salaireSummary.getSum());

	}
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (26 jan. 2022 à 14:57:27)
IntSummaryStatistics{count=10, sum=421, min=25, average=42,100000, max=52}
10
25
52
42.1
421
```

Les méthodes averagingInt(), averagingLong() et averagingDouble() de la classe Collectors renvoient un Collector qui calcule la moyenne des données numériques extraites des éléments du Stream.

```
public static void test89() {
Stream<String> notes = Stream.of("10.5", "18", "20");
Double moyenne =
notes.collect(Collectors.averagingDouble(Double::valueOf));
            System.out.println(moyenne);
      }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (26 jan. 2022 à 15:17:00)
16.166666666666668
```

Les méthodes summingInt(), summingLong() et summingDouble() de la classe Collectors renvoient un Collector qui calcule la somme des données numériques extraites des éléments du Stream.

```
public static void test90() {
Stream<String> notes = Stream.of("12", "20", "30");
long somme = notes.collect(Collectors.summingLong(Long::valueOf));
System.out.println(somme);
}
```

Les méthodes minBy() et maxBy() de la classe Collectors renvoient un Collector qui détermine respectivement le plus petit et le plus grand élément du Stream. Elles attendent en paramètre un Comparator qui sera utilisé pour déterminer l'élément concerné. Elles renvoient un Optional qui encapsule éventuellement cet élément s'il existe.

```
public static void test91() {
Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
Optional<String> lePlusGrand =
chaines.collect(Collectors.maxBy(Comparator.naturalOrder()));
System.out.println(lePlusGrand.get());
}
```

Le Comparator utilisé peut par exemple être plus générique en utilisant la méthode static comparing() de l'interface Comparator.

```
public static void test92() {
Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
Optional<String> lePlusLong =
chaines.collect(Collectors.maxBy(Comparator.comparingInt(String::length)));
System.out.println(lePlusLong.get());
}
```

5. Les fabriques qui renvoient des Collectors pour effectuer des groupements

La méthode groupingBy() renvoie un Collector qui va regrouper les éléments du Stream dans une Map.

```
public static void test93() {
Stream<String> chaines = Stream.of("aaa", "bb", "ccccc","dd");
Map<Integer, List<String>> map =
chaines.collect(Collectors.groupingBy(String::length,
Collectors.toList()));
for (Map.Entry entry : map.entrySet()) {
System.out.println(entry.getKey() + ", " + entry.getValue());
}
}
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (26 jan. 2022 à 16:24:09)
2, [bb, dd]
3, [aaa]
5, [ccccc]
```

La méthode partitioningBy() est une spécialisation de la méthode groupingBy(). Elle attend en paramètre un Predicate pour grouper les éléments selon la valeur booléenne retournée par le Predicate. La collection de type Map retournée possède donc forcément un booléen comme clé.

```
public static void test94() {
Stream<String> chaines = Stream.of("aaa", "bb", "ccccc");
Map<Boolean, List<String>> map =
chaines.collect(Collectors.partitioningBy(s -> s.length() >= 3));
for (Map.Entry entry : map.entrySet()) {
     System.out.println(entry.getKey() + ", " + entry.getValue());
}
}
```

```
<terminated> Test1 (3) [Java Application] (
false, [bb]
true, [aaa, ccccc]
```

6. Les fabriques qui renvoient des Collectors pour effectuer des transformations

La méthode mapping() renvoie un Collector qui va transformer les objets de type T en type U grâce à la Function avant d'appliquer le downstream Collector.

public static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)

```
public static void test95() {
           List<Emp> employes = Arrays.asList(
                       new Emp("emp_1", 33, Genre.HOMME),
                       new Emp("emp_2", 43, Genre.HOMME),
                       new Emp("emp_3", 52, Genre.HOMME),
                       new Emp("emp_4", 52, Genre.HOMME),
                       new Emp("emp_5", 50, Genre.HOMME),
                       new Emp("emp_6", 49, Genre.FEMME),
                       new Emp("emp_7", 41, Genre.FEMME),
                       new Emp("emp_8", 39, Genre.FEMME),
                       new Emp("emp_9", 37, Genre.FEMME),
                       new Emp("emp_10", 25, Genre.FEMME));

           Map<Genre, Set<String>> nomsEtudiantsParGenre
          = employes.stream()
                       .collect(groupingBy(Emp::getGenre,
                                   mapping(Emp::getNom, toSet())));
           for (Map.Entry entry : nomsEtudiantsParGenre.entrySet()) {
                System.out.println(entry.getKey() + ", " +
entry.getValue());
           }
     }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (26 jan. 2022 à 16:46:55)
HOMME, [emp_1, emp_2, emp_5, emp_3, emp_4]
FEMME, [emp_9, emp_7, emp_8, emp_10, emp_6]
```

c. 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.

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.

```
public static void test96() {
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());
}
}
```

```
<terminated> Test1 (3) [Ja
CC, 1
BB, 3
AA, 2
```

Comme la méthode groupingBy() renvoie un Collector, il est possible d'utiliser un groupingBy() comme downstream Collector et ainsi réaliser un groupement à deux niveaux. Dans l'exemple ci-dessous, on effectue un groupement par âge et par genre.

```
public static void test97() {
          List<Emp> employes = Arrays.asList(
                    new Emp("emp_1", 30, Genre.HOMME),
                    new Emp("emp_2", 60, Genre.HOMME),
                    new Emp("emp_3", 70, Genre.HOMME),
                    new Emp("emp_4", 60, Genre.HOMME),
                    new Emp("emp_5", 70, Genre.HOMME),
                    new Emp("emp_6", 30, Genre.FEMME),
                    new Emp("emp_7", 30, Genre.FEMME),
                    new Emp("emp_8", 70, Genre.FEMME),
                    new Emp("emp_9", 30, Genre.FEMME),
                    new Emp("emp_10", 30, Genre.FEMME));

           Map<Integer, Map<Genre, List<Emp>>> employesParAgeEtParGenre =
employes
                         .stream()
                         .collect(groupingBy(Emp::getAge,
                                   groupingBy(Emp::getGenre)));
                   System.out.println("resultat = "+
employesParAgeEtParGenre);
     }
```

```
resultat = {70={HOMME=[Emp(nom=emp_3, age=70, genre=HOMME), Emp(nom=emp_5,
age=70, genre=HOMME)], FEMME=[Emp(nom=emp_8, age=70, genre=FEMME)]},
60={HOMME=[Emp(nom=emp_2, age=60, genre=HOMME), Emp(nom=emp_4, age=60,
genre=HOMME)]}, 30={HOMME=[Emp(nom=emp_1, age=30, genre=HOMME)],
FEMME=[Emp(nom=emp_6, age=30, genre=FEMME), Emp(nom=emp_7, age=30,
```

```
genre=FEMME), Emp(nom=emp_9, age=30, genre=FEMME), Emp(nom=emp_10, age=30,
genre=FEMME)]}}
```

d. L'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.

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

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

e. Les fabriques of() pour créer des instances de Collector

Il est possible de créer une instance de type Collector en utilisant sa fabrique of() qui possède deux surcharges.

L'exemple ci-dessous créé une instance de type Collector pour concaténer les noms d'un Stream de d'objet de type Emp mis en majuscules. Le séparateur utilisé est une virgule.

```
public static void test98() {
            List<Emp> employes = Arrays.asList(
                        new Emp("emp_1", 30, Genre.HOMME),
                        new Emp("emp_2", 60, Genre.HOMME),
                        new Emp("emp_3", 70, Genre.HOMME),
                        new Emp("emp_4", 60, Genre.HOMME),
                        new Emp("emp_5", 70, Genre.HOMME),
                        new Emp("emp_6", 30, Genre.FEMME),
                        new Emp("emp_7", 30, Genre.FEMME),
                        new Emp("emp_8", 70, Genre.FEMME),
                        new Emp("emp_9", 30, Genre.FEMME),
                        new Emp("emp_10", 30, Genre.FEMME));
```

```
            Collector<Emp, StringJoiner, String> nomEmpCollector =
                        Collector.of(
                         () -> new StringJoiner(","),                        // supplier
                         (sj, p) -> sj.add(p.getNom().toUpperCase()),  // accumulator
                         (sj1, sj2) -> sj1.merge(sj2),                       // combiner
                         StringJoiner::toString);                           // finisher
                    String noms = employes
                     .stream()
                     .collect(nomEmpCollector);
                    System.out.println(noms);
      }
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (27 jan. 2022 à 01:53:22)
EMP_1,EMP_2,EMP_3,EMP_4,EMP_5,EMP_6,EMP_7,EMP_8,EMP_9,EMP_10
```

Le Collector utilise la classe StringJoiner de Java 8 pour concaténer les prénoms des personnes dans ses traitements :

- Le supplier fournit une instance de StringJoiner avec le délimiteur à utiliser
- L'accumulator utilise la méthode add() du StringJoiner pour concaténer chaque prénom mis en majuscules en les séparant par une virgule
- Le combiner utilise la méthode merge() pour fusionner les deux StringJoiner fournis en paramètre
- Le finisher utilise la méthode toString() du StringJoiner pour fournir le résultat final sous la forme d'une chaîne de caractères. Son utilisation est nécessaire car le type du Supplier et le type renvoyé du Collector sont différents

9. Les Streams pour des données primitives

L'interface Stream<T> 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.

a. Les interfaces IntStream, LongStream et DoubleStream

```
public static void test99() {
            IntStream.rangeClosed(1, 5).forEach(n -> System.out.print(n));
            System.out.println();
            IntStream.range(1, 5).forEach(n -> System.out.print(n));
```

```
    }
```

```
<terminated> Test1 (3) [Java
12345
1234
```

L'exemple ci-dessous détermine les nombres impairs compris entre 0 et 10 inclus.

```
public static void test100() {
            IntStream nombresImpairs = IntStream.rangeClosed(0,
10).filter(nombre -> (nombre % 2) == 1);
            nombresImpairs.forEach(System.out::println);
      }
```

La méthode average() est une opération de réduction qui calcule la moyenne des éléments du Stream. Elle renvoie un objet de type OptionalDouble. Elle renvoie un objet de type xxxOptional ou xxx est le type primitif du Stream.

```
public static void test101() {
            IntStream.of(1, 2, 3, 4, 5)
         .map(n -> 2 * n)
         .average()
         .ifPresent(System.out::println);
      }
```

10. L'utilisation des Streams avec les opérations I/O

a. La création d'un Stream à partir d'un fichier texte

L'API NIO 2 de Java 7, propose une méthode pour lire l'intégralité des lignes d'un fichier texte.

List<String> lines = Files.readAllLines(somePath, someCharset);

L'utilisation d'un Stream permet de ne pas avoir besoin de stocker l'intégralité du contenu du fichier en mémoire : la lecture dans le fichier se fait au fur et à mesure de la consommation par le Stream. Ceci peut être intéressant notamment pour le traitement de gros fichiers.

La méthode lines() de la classe Files attend en paramètre le fichier sous la forme d'un objet de type Path. Elle permet de renvoyer un Stream<String> qui va lire de manière lazy les lignes d'un fichier pour alimenter le Stream au fur et à mesure de la consommation des lignes.

Contrairement à la méthode readAllLines() qui lit l'intégralité du fichier, la méthode lines() peut se contenter de ne lire que les lignes que lors de l'invocation de la méthode terminale et la lecture peut être interrompue par l'exécution d'une méthode de type short-circuiting.

Les caractères sont décodés en utilisant le ChartSet UTF-8 par défaut. La méthode lines() possède une seconde surcharge qui attend en paramètre deux objets de type Path et CharSet.

Lorsque la source de données du Stream effectue des opérations de type I/O, il est nécessaire de libérer les ressources allouées pour lire les données. L'interface BaseStream définit la méthode close() et la classe Stream implémente l'interface AutoCloseable.

La plupart des Streams n'ont pas besoin d'avoir leur méthode close() invoquée sauf dans certains cas notamment lorsque la source réalise des opérations de type I/O. La méthode close() doit être explicitement invoquée si nécessaire pour libérer les ressources ouvertes par la source.

Pour éviter de laisser le système libérer les ressources au bout d'un certain timeout, il faut invoquer la méthode close() de l'objet responsable de la lecture des données. Ceci peut éviter des fuites de ressources. Il est donc important d'invoquer la méthode close() du Stream pour qu'elle invoque la méthode close() de la source utilisée pour lire les lignes du fichier.

La méthode onClose() de l'interface BaseStream est une opération intermédiaire qui permet d'exécuter un traitement lorsque la méthode close() du Stream est invoquée. Il est possible d'invoquer plusieurs fois cette méthode : les traitements seront alors exécutés dans leur ordre d'ajout.

```
public static void test102() {
          try {
               Stream<String> lignes =
Files.lines(Paths.get("c:/tmp/fichier.txt"), Charset.defaultCharset());
               long nbLignes = lignes.count();
               lignes.close();
               System.out.println("Nb lignes = " + nbLignes);
          } catch (IOException e) {
               e.printStackTrace();
          }
     }
```

```
<terminated> Test1 (3) [Java Appl
Nb lignes = 2
```

Il est aussi possible d'utiliser un try with resources sur un Stream qui va invoquer sa méthode close(). Cette dernière va se charger de libérer les éventuelles ressources ouvertes par la source de données.

```
public static void test103() {
          try (Stream<String> lignes =
```

```
Files.lines(Paths.get("c:/tmp/fichier.txt"))) {
                    long nbLignes = lignes.count();
                    System.out.println("Nb lignes = " + nbLignes);
                } catch (IOException e) {
                    e.printStackTrace();
                }
      }
```

b. La création d'un Stream à partir du contenu d'un répertoire

Plusieurs méthodes ont été ajoutées à la classe Files dans la version 8 de Java.

Il est important d'invoquer la méthode close() du Stream à la fin de son utilisation. Le plus simple est d'utiliser une instruction try with resources.

```
public static void test104() {
            String folder = "c:/tmp";
            try (Stream<Path> paths = Files.list(Paths.get(folder))) {
                  paths.filter(p ->
p.toString().endsWith(".txt")).forEach(System.out::println);
            } catch (IOException ioe) {
                  ioe.printStackTrace();
            }
      }
```

11. 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.

Java 7 a introduit le framework Fork/Join pour faciliter la parallélisation de tâches. Ce framework est utilisé par l'implémentation de l'API Stream pour permettre l'exécution de ses traitements en parallèle.

Globalement aussi, le volume des données à traiter augmente. Généralement ces données sont stockées dans une collection. La conception de l'API Collection étant assez ancienne (Java 1.2), il n'est pas facile de proposer l'intégration de fonctionnalités en parallèle dans ces classes.

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.

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.

```
public static void test105() {
List<String> elements = Arrays.asList("elem1", " elem2", " elem3",
"elem4", "elem5");
          elements.stream().forEach(System.out::println);
          elements.parallelStream().forEach(System.out::println);
     }
```

a. La mise en oeuvre des Streams parallèles

La plupart des Streams créés dans l'API du JDK ont leurs opérations qui par défaut seront exécutées de manière séquentielle dans le thread courant. Il est alors nécessaire de préciser explicitement que l'on souhaite que les opérations du Stream soient exécutées de manière parallèle.

Puisque c'est l'API qui se charge d'itérer sur les différents éléments pour exécuter le pipeline d'opérations, il est facilement possible de demander l'exécution de ces traitements en parallèle. Le fonctionnement interne de l'API masque alors toute la complexité de l'exécution en parallèle des traitements.

Pour le développeur, cela consiste simplement :

- Soit à demander la création d'un Stream parallèle en utilisant la méthode parallelStream() des collections
- Soit à convertir un Stream séquentiel en Stream parallèle invoquant la méthode parallel() de l'interface BaseStream qui est une opération intermédiaire

Remarque : il est aussi possible de transformer un Stream parallèle en Stream séquentiel en invoquant la méthode sequential().

b. Le fonctionnement interne d'une Stream

Le découpage en sous-lots est réalisé grâce à un objet de type Spliterator. La suite d'opérations du pipeline pour chacun de ces sous-lots sera alors exécuté par un thread libre du pool du framework Fork/Join.

12. 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.

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.

```
public static void test106() {
IntStream.iterate(0, i -> i + 1).limit(5).forEach(System.out::println);
      }
```

13. Le débogage d'un Stream

Le débogage d'un Stream n'est pas simple car une majorité des traitements est réalisée en interne par l'API.

Lorsqu'une anomalie survient dans les traitements d'un Stream, la stacktrace n'est généralement pas d'une grande aide.

```
public static void test107() {
          Double tailleMoyenne = Stream.of("texte", null, "grand
texte").mapToInt(String::length).average().getAsDouble();
          System.out.println("taille moyenne = " + tailleMoyenne);
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at java.util.stream.ReferencePipeline$4$1.accept(ReferencePipeline.java:210)
        at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
        at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
        at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
        at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
        at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
        at java.util.stream.IntPipeline.collect(IntPipeline.java:472)
        at java.util.stream.IntPipeline.average(IntPipeline.java:434)
        at ma.formation.stream.Test1.test107(Test1.java:875)
        at ma.formation.stream.Test1.main(Test1.java:881)
```

Une exception de type NullPointerException est levée mais la lecture de la stacktrace ne fournit aucune information pour permettre de déterminer l'origine du problème.

Il est possible d'utiliser la méthode peek() pour afficher l'élément en cours de traitement dans le pipeline. 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.

```
public static void test108() {
		Double tailleMoyenne = Stream.of("texte", null, "grand texte")
				.peek(e -> System.out.println("map : " + e))
				.mapToInt(String::length)
				.peek(e -> System.out.println("average : " + e))
				.average().getAsDouble();
		System.out.println("taille moyenne = " + tailleMoyenne);
	}
```

```
<terminated> Test1 (3) [Java Application] C:\Java\jdk1.8.0_121\bin\javaw.exe (28 jan. 2022 à 12:18:47)
map : texte
Exception in thread "main" average : 5
map : null
java.lang.NullPointerException
	at java.util.stream.ReferencePipeline$4$1.accept(ReferencePipeline.java:210)
	at java.util.stream.ReferencePipeline$11$1.accept(ReferencePipeline.java:373)
	at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)
	at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
	at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
	at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
	at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
	at java.util.stream.IntPipeline.collect(IntPipeline.java:472)
	at java.util.stream.IntPipeline.average(IntPipeline.java:434)
	at ma.formation.stream.Test1.test108(Test1.java:885)
	at ma.formation.stream.Test1.main(Test1.java:890)
```

14. 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 ces propres opérations : seules celles définies par l'API peuvent être utilisées.

a. 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.

```java
public static void test109() {
		Stream<String> stream = Stream.of("a1", "a2", "a3").filter(s ->
s.startsWith("a"));
		stream.forEach(System.out::println);
		stream.forEach(System.out::println);
	}
```

```
a1
a2
a3
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
	at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
	at java.util.stream.ReferencePipeline.forEach(ReferencePipeline.java:418)
	at ma.formation.stream.Test1.test109(Test1.java:889)
	at ma.formation.stream.Test1.main(Test1.java:893)
```

Pour contourner cette limitation, il faut créer un nouveau Stream pour chaque utilisation. Si la configuration du Stream est complexe, il est possible de définir un Supplier dont le rôle sera de créer une instance à chaque invocation.

```java
public static void test110() {
		Supplier<Stream<String>> supplier = () -> Stream.of("a1", "a2",
"a3").filter(s -> s.startsWith("a"));
		supplier.get().forEach(System.out::println);
		supplier.get().forEach(System.out::println);
	}
```

15. Quelques recommandations sur l'utilisation de l'API Stream

L'API Stream permet de réaliser certains traitements sur un ensemble de données de manière déclarative, ce qui réduit la quantité de code à produire. La déclaration de certains traitements peut se faire de différentes manières pouvant avoir des impacts sur les performances.

Il ne faut pas utiliser systématiquement l'API Stream mais plutôt favoriser son utilisation lorsque celle-ci apporte une plus-value. Si l'API est utilisée, il est préférable de le faire de manière optimale.

- ✓ Il n'est pas utile d'avoir recours à un Stream pour appliquer un traitement sur les éléments d'une collection. C'est d'autant plus vrai si aucune opération intermédiaire n'est utilisée dans le Stream.
- ✓ Le remplacement d'une boucle for par un Stream n'est réellement intéressant que si l'approche fonctionnelle s'appuyant sur une itération interne est utilisée pour par exemple chaîner plusieurs opérations dans le but d'obtenir un résultat.
- ✓ Il n'est pas utile d'avoir recours à un Stream pour convertir les éléments d'un tableau en une collection de type List. Il est préférable d'utiliser la méthode toList() de la classe Arrays.

- ✓ Il n'est pas nécessaire d'utiliser un Stream pour uniquement trouver le plus grand élément dans une collection. Il est préférable d'utiliser la méthode max() de la classe Collections.
- ✓ Il n'est pas nécessaire d'utiliser un Stream pour uniquement déterminer le nombre d'éléments d'une collection. Il est préférable d'utiliser la méthode size() de l'interface Collection.
- ✓ Il n'est pas nécessaire d'utiliser un Stream pour uniquement vérifier la présence d'un élément dans une collection. Il est préférable d'utiliser la méthode contains() de l'interface Collection.
- ✓ Il n'est pas utile de filtrer les éléments et de vérifier s'il y a un premier élément restant pour s'assurer qu'un élément est présent dans une collection. Il est préférable d'utiliser l'opération terminale anyMatch() du Stream.
- ✓ Il n'est pas utile de trier les éléments dans un ordre croissant et de prendre le premier pour rechercher le plus petit élément. Il est préférable d'utiliser l'opération min() du Stream.
- ✓ Il est préférable d'utiliser les fabriques dédiées de l'API Stream plutôt que de créer une collection à partir de laquelle on obtient un Stream pour traiter les éléments.
- ✓ Il n'est pas utile de créer une collection vide et de demander un Stream sur celle-ci pour obtenir un Stream vide. Il est préférable d'utiliser la méthode empty() de l'interface Stream.
- ✓ …

Fin du TP 4.