Chapitre 15

Les Streams

1 Définition et principes

2 Exemples d’application

1 Définition et principes

2 Exemples d’application

Définition et principes (1/4)

- Un **Stream** est une fonctionnalité introduite dans Java (package *java.util.stream*) à partir de sa version 8.
  - > C'est une séquence d'éléments sur laquelle on peut effectuer un ensemble d'opérations séquentiellement ou parallèlement
  - > Un Stream n'est pas une structure (**collection**, **tableau**, etc) conçue pour le stockage de données/objets
  - > C'est un **flux** appliqué à une **source de données** (collection, tableau, etc) pour réaliser une **série d'opérations** de façon chaînée afin d'obtenir un résultat final. C'est un pipeline de traitements dont chaque étape correspond à une opération.
  - > Les **opérations** fournies sur les Streams sont basées sur les expressions lambda utilisant les interfaces fonc. prédéfinies
- Deux grands groupes d'opérations applicables sur les Streams :
  - ★ **Les opérations intermédiaires :** méthodes prenant en entrée un élément d'un Stream et retournant un Stream
  - ★ **Les opérations terminales :** méthodes marquant la fin d'un Stream. Prend en entrée un Stream et retourne un résultat
- Seule l'application d'une opération terminale déclenche l'exécution des opérations intermédiaires appliquées sur un Stream initial. En d'autres termes, les opérations intermédiaires d'un pipeline ne seront jamais exécutées si le pipeline ne déclare pas en fin de chaine une opération terminale. L'application d'une opération terminale rend un Stream inutilisable après coup

Définition et principes (2/4)

- **Source de données :**

  > Pour créer un Stream initial sur lequel appliquer des opérations diverses, il existe plusieurs possibilités :

  - Exemple à partir d'une collection (ArrayList, HashSet, etc) :

```
List<Integer> liste = Arrays.asList(1, 2, 3);
Stream<Integer> source = liste.stream();          // source correspond à un Stream sur lequel on peut désormais appliquer une série d'opérations
```

  - Exemple à partir d'un tableau :

```
Integer[] values = {1, 2, 3};
Stream<Integer> source = Arrays.stream(values);
```

  - Exemple avec la méthode statique *of* des Streams :

```
1/ Stream<Integer> source = Stream.of(Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(3));

2/ Integer[] values = {1, 2, 3};
   Stream<Integer> source = Stream.of(values);
```

  - Exemple avec la méthode statique *generate* des Streams :

  Signature : *Stream.generate(Supplier s)*

```
Stream<Integer> source = Stream.generate( () -> Math.round( 10*Math.random() ) );
```

génère un nombre infini d'un chiffre choisi entre 0 et 10

Définition et principes (3/4)

- **Quelques opérations intermédiaires :**
  - map() : transforme chaque élément d'un Stream en un unique autre suivant un critère
  - filter() : filtre les éléments d'un Stream respectant un prédicat
  - limit() : limite le nombre d'éléments d'un Stream à la taille indiquée
  - sorted() : tri les éléments d'un Stream
  - flatmap() : transforme chaque élément d'un Stream en zéro ou plusieurs éléments suivant un critère
  - skip() : Permet d'ignorer un nombre d'éléments dans un Stream
  - peek() : méthode passive pouvant servir à loguer/visualiser les éléments traiter dans un Stream

- **Quelques opérations terminales :**
  - min() / max() : retourne le plus petit (resp. le plus grand) élément d'un Stream
  - count() : retourne le nombre d'éléments présents dans un Stream
  - foreach() : boucle sur les éléments d'un Stream
  - collect() : retourne les éléments d'un Stream sous forme d'une collection (ArrayList, HashSet, HashMap, etc)
  - reduce() : retourne une donnée résultante d'un *BinaryOperator*, argument de cette méthode *reduce*
  - anyMatch() : retourne le premier élément qui match un critère donné

Définition et principes (4/4)

- **Quelques opérations intermédiaires :**
  - map() : Stream<R> map(Function<? super T, ? extends R> mapper);
  - filter() : Stream<R> filter(Predicate<? super R> predicate)
  - limit() : Stream<R> limit(long maxSize)
  - sorted() : Stream<R> sorted(Comparator<? super R> comparator) | Stream<R> sorted()
  - flatmap() : Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  - skip() : Stream<R> skip(long n)
  - peek() : Stream<R> peek(Consumer<? super R> action)

- **Quelques opérations terminales :**
  - min() / max() : Optional<R> min(Comparator<? super R> comparator) | Optional<R> max(Comparator<? super R> comparator)
  - count() : long count()
  - foreach() : void forEach(Consumer<? super R> action)
  - collect() : R collect(Collector<? super T, A, R> collector)
  - reduce() : R reduce(R identity, BinaryOperator<R> accumulator)
  - anyMatch() : boolean anyMatch(Predicate<? super R> predicate)

1 Définition et principes

2 Exemples d’application

Exemples d'applications (1/4)

- **Utilisation des opérations *filter* et *foreach* :**

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

public class MainStream {

     public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
        list.stream().filter(num -> num%2 == 0 ).forEach(e -> System.out.println(e));
     }
}
```

- **Résultat :**

```
2
4
```

Exemples d'applications (2/4)

- **Utilisation des opérations *map*, *filter*, *sorted* et *foreach* :**

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

public class MainStream {

     public static void main(String[] args) {
        List<Integer> list = Arrays.asList(5, 3, 4, 1, 2);
        list.stream().map(e -> e + 1)
                     .filter(e -> e%2 !=0)
                     .sorted()
                     .forEach(e -> System.out.println(e));
     }
}
```

- **Résultat :**

```
3
5
```

Exemples d'applications (3/4)

- **Utilisation des opérations *generate*, *limit, skip,* et *reduce* :**

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

public class MainStream {

     public static void main(String[] args) {
        long seuil = Math.round(100*Math.random());
        System.out.print(seuil);
        long nombre = Stream.generate(() -> 100L)).limit(seuil)
                                                   .skip(10)
                                                   .reduce(0L, (a, b) -> a + b);;

        System.out.print(nombre);
     }
}
```

- **Résultats variables**

=> Le programme fait une somme des (***seuil***-10) du nombre 100 (ie, (***seuil***-10)x100).
A chaque exécution, un seuil aléatoire est choisi (0<=seuil <= 100)

Exemples d'applications (4/4)

- **Utilisation des opérations *of, filter, limit, peek*, et *collect* :**

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

public class MainStream {

     public static void main(String[] args) {
        List<String> listeCodePaysCommencantParC = Stream.of(Locale.getISOCountries())
                                                         .filter(e -> e.startsWith("C"))
                                                         .limit(5)
                                                         .peek(e -> System.out.print(e + " | "))
                                                         .collect(Collectors.toList());


        System.out.println("\n\nVérification : ");
        listeCodePaysCommencantParC.forEach(e -> System.out.print(e + " | ") );
     }
}
```

- **Résultat :**

```
CA | CC | CD | CF | CG |

Vérification :
CA | CC | CD | CF | CG |
```


SUCCESS