



COURS JAVA AVANCÉ PARTIE 2 - SUITE

Mme Nouria Sana



COURS JAVA AVANCÉ -

LES STREAMS

Mme Nouria Sana

API STREAM : UNE NOUVELLE FAÇON DE GÉRER LES COLLECTIONS EN JAVA 8

- Cette nouvelle API fournie par le JDK 8 modifier fondamentalement notre façon de traiter les Collections.
- Elle propose une alternative au pattern Iterator relativement lourd à mettre en place.
- Celle-ci tire profit de la nouvelle syntaxe des **lambdas expressions** pour réduire le code un maximum tout en améliorant ses performances.
- De plus, la classe **Collectors** offre de nombreux patterns qui remplacent dans de nombreux cas le pattern Iterator.



QU'EST-CE QU'UN STREAM ?

- Un Stream n'est pas une collection ou une structure de donnée de manière générale.
- C'est une séquence d'éléments sur laquelle on peut effectuer un groupe d'opérations de manière séquentielle ou parallèle.

public interface Stream<T>

Permet de remplacer le pattern Iterator

- Il existe deux types d'opérations sur les Stream :
 - **Les opérations intermédiaires** : elles transforment un Stream en autre Stream : ce genre d'opération conserve le stream ouvert ce qui permet d'effectuer d'autre opérations dessus. Exemple : Methodes **map()** et **filter()**
 - **Les opérations finales (terminale)**: elles produisent un résultat ou un « side-effect » . C'est l'opération finale du stream, c'est ce qui lance la « consommation » du stream. Exemple : **Reduce()**



JAVA.UTIL.STREAM

- Les streams utilisent les interfaces fonctionnelles
- un Stream ne stocke aucune donnée il se contente de les transférer vers une suite instruction à opérer
- Un stream ne modifie pas les données
- Un stream est à usage unique : une fois utilisé complètement impossible de l'utiliser une seconde fois




EXEMPLE CLASS TESTSTREAM (VOIR NETBEANS)

```
public enum Couleur {  
    MARRON("marron"),  
    BLEU("bleu"),  
    VERT("vert"),  
    VERRON("verron"),  
    INCONNU("non déterminé"),  
    ROUGE("rouge mais j'avais piscine...");
```

```
    private String name = "";  
    Couleur(String n){name = n;}  
    public String toString()  
    {return name;}  
}
```

```
public class Personne {  
    public Double taille = 0.0d, poids = 0.0d;  
    public String nom = "", prenom = "";  
    public Couleur yeux = Couleur.INCONNU;  
    public Personne() {  
        public Personne(double taille, double poids, String nom, String prenom, Couleur yeux)  
        {  
            super();  
            this.taille = taille;    this.poids = poids;  
            this.nom = nom;        this.prenom = prenom;  
            this.yeux = yeux;    }  
    }  
  
    public String toString() {  
        String s = "Je m'appelle " + nom + " " + prenom;  
        s += ", je pèse " + poids + " Kg";  
        s += ", et je mesure " + taille + " cm.";  
        return s;  
    }  
    // getter et setter ....}}
```



```
public class TestStream {  
    public static void main(String[] args) {  
        List<Personne> listP = Arrays.asList(  
            new Personne(1.80, 70, "A", "Noura", Couleur.BLEU),  
            new Personne(1.56, 50, "B", "Nouira", Couleur.VERRON),  
            new Personne(1.75, 65, "C", "Ali", Couleur.VERT),  
            new Personne(1.68, 50, "D", "Sonia", Couleur.ROUGE),  
            new Personne(1.96, 65, "E", "Karima", Couleur.BLEU),  
            new Personne(2.10, 120, "F", "Emna", Couleur.ROUGE),  
            new Personne(1.90, 90, "G", "Skander", Couleur.VERRON)        );
```

//parcours stream

```
Stream<Personne> sp = listP.stream();  
sp.forEach(System.out::println);
```

Je m'appelle A Noura, je pèse 70.0 Kg, et je mesure 1.8 cm.
Je m'appelle B Nouira, je pèse 50.0 Kg, et je mesure 1.56 cm.
Je m'appelle C Ali, je pèse 65.0 Kg, et je mesure 1.75 cm.
Je m'appelle D Sonia, je pèse 50.0 Kg, et je mesure 1.68 cm.
Je m'appelle E Karima, je pèse 65.0 Kg, et je mesure 1.96 cm.
Je m'appelle F Emna, je pèse 120.0 Kg, et je mesure 2.1 cm.
Je m'appelle G Skander, je pèse 90.0 Kg, et je mesure 1.9 cm.

// operation Filter

```
System.out.println("\nAprès le filtre filtrer personne dont poids est >50");
```

```
sp = listP.stream(); //N'oubliez surtout pas de recréer le stream car le premier est entièrement consommé  
sp.filter(x -> x.getPoids() > 50).forEach(System.out::println);
```

Après le filtre

Je m'appelle A Noura, je pèse 70.0 Kg, et je mesure 1.8 cm.
Je m'appelle C Ali, je pèse 65.0 Kg, et je mesure 1.75 cm.
Je m'appelle E Karima, je pèse 65.0 Kg, et je mesure 1.96 cm.
Je m'appelle F Emna, je pèse 120.0 Kg, et je mesure 2.1 cm.
Je m'appelle G Skander, je pèse 90.0 Kg, et je mesure 1.9 cm., et je mesure 1.9 cm.



//Operation map

//appliquer une opération sur chaque élément afin de ne récupérer que ce qui nous intéresse.

//Par exemple, en ne conservant que le poids des personnes que nous avons filtré.

```
System.out.println("\nAprès le filtre et le map");
```

```
sp = listP.stream();
```

```
sp.filter(x -> x.getPoids() > 50)
```

```
    .map(x -> x.getPoids())
```

```
    .forEach(System.out::println);
```

Après le filtre et le map

70.0

65.0

65.0

120.0

90.0



//operation reduce

//Cette opération a pour but d'agréger le contenu de votre stream pour fournir un résultat unique.

//Exemple calculer la somme des poids des personnes que nous //avons filtré précédemment

```
System.out.println("\nAprès le filtre et le map et reduce");
```

```
sp = listP.stream();
```

```
Double sum = sp.filter(x -> x.getPoids() > 50)
```

```
    .map(x -> x.getPoids())
```

```
    .reduce(0.0d, (x,y) -> x+y);
```

```
    System.out.println(sum);
```

Après le filtre et le map et reduce
410.0



```
System.out.println("\nAprès le filtre et le map et reduce");
```

```
sp = listP.stream();
```

```
Optional<Double> sum = sp.filter(x -> x.getPoids() > 250)
```

```
    .map(x -> x.getPoids())
```

```
    .reduce((x,y) -> x+y);
```

```
if(sum.isPresent())
```

```
    System.out.println(sum.get());
```

```
else
```

```
    System.out.println("Aucun agrégat de poids...");
```

Après le filtre et le map et reduce
Aucun agrégat de poids...



```
sp = listP.stream();
```

```
long count = sp.filter(x -> x.getPoids() > 50)  
    .map(x -> x.getPoids())  
    .count();
```

Nombre d'éléments : 5

```
System.out.println("Nombre d'éléments : " + count);
```

```
sp = listP.stream();
```

```
List<Double> ld = sp.filter(x -> x.getPoids() > 50)  
    .map(x -> x.getPoids())  
    .collect(Collectors.toList());
```

```
System.out.println(ld);
```

[70.0, 65.0, 65.0, 120.0, 90.0]



EXEMPLE ORDER(NETBEANS)

- Manipuler des ordres d'achats et de ventes de produits. Un ordre sera soit d'achat ou de vente d'un produit quelconque (mobile, souris, vélo,...) avec un prix donné.
- Nous avons donc un objet Order qui définit quatre propriétés.
- L'exercice consiste à récupérer dans la liste ci-dessous les produits vendus et les trier par prix croissant.

id	type	price	product
1	BUY	100	phone
2	SELL	50	mouse
3	SELL	150	bike
4	BUY	500	laptop
5	SELL	40	keyboard



```

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import static java.util.stream.Collectors.toList;

public class Order {
    public enum OrderType { BUY, SELL };
    int id;
    OrderType type;
    Double price;
    String product;

    public Order(int id, OrderType type, Double price, String product) {
        this.id = id;
        this.type = type;
        this.price = price;
        this.product = product;
    }

    public int getId() {return id; }
    public OrderType getType() { return type; }
    public Double getPrice() { return price; }
    public String getProduct() { return product; }
    public String toString(){ return id + " " + type + " " + price + " " + product; }

    public static void main(String[] args) {
        List<Order> orderList = new ArrayList<>();

```

```

        orderList.add(new Order(1, OrderType.BUY, 100.0, "phone"));
        orderList.add(new Order(2, OrderType.SELL, 50.0, "mouse"));
        orderList.add(new Order(3, OrderType.SELL, 150.0, "bike"));
        orderList.add(new Order(4, OrderType.BUY, 500.0, "laptop"));
        orderList.add(new Order(5, OrderType.SELL, 40.0, "keyboard"));

```

•`orderList.stream()` renvoie un stream d'objets `Order` soit Stream

•// Récupération de la liste triée sans l'utilisation des Stream

```

List<Order> sellOrderList = new ArrayList<>();
for (Order order : orderList) {
    if (order.getType() == Order.OrderType.SELL)
    {    sellOrderList.add(order); }
}

```

```

Collections.sort(sellOrderList, (o1, o2) -> o1.getPrice().compareTo(o2.getPrice()));
List<String> products = new ArrayList<>();
for (Order o : sellOrderList) {
    products.add(o.getProduct());
    System.out.println(o.getProduct());
}

```

//*Resultat :
keyboard
mouse
bike*/

•//Récupération de la liste triée avec l'utilisation de Stream

```

List<String> products2 = orderList.stream()
    .filter(o-> o.getType()==OrderType.SELL)
    .sorted(Comparator.comparing(Order::getPrice))
    .map(Order::getProduct)
    .collect(toList());
System.out.println(products2);

```

//Resultat
[keyboard, mouse, bike]
}

AVEC STREAM :

- ArrayList temporaire disparaît
- L'application d'opérations successives sur la liste est plus claire
- Le code se réduit à une ligne

• `filter()` permet de sélectionner dans le stream tous les éléments respectant un prédicat, en retour on obtient toujours un Stream. C'est une fonction intermédiaire.

utilisation de la lambda `o -> o.getType() == OrderType.SELL`. Appliquée à chaque élément du Stream, elle vérifiera le type de l'ordre. Ici par exemple nous souhaitons ne sélectionner que les ordres de vente.

• `sorted()` va trier les éléments du Stream en utilisant un comparator passé en argument, ici encore j'ai fait le choix d'utiliser une lambda:

`(o1, o2) -> o1.getPrice().compareTo(o2.getPrice())`

• `map()` est bien connue en programmation fonctionnelle, elle va appliquer une transformation à chaque élément, ici on va récupérer le produit sur lequel porte un ordre et ainsi récupérer en retour un stream de produits, Stream.

D'AUTRES OPÉRATIONS

- Peek : operation intermediaire permettant de « déboguer » entre chaque opération.
- Limit ou skip : opération également intermédiaire permettant de limiter ou bloquer des éléments d'un stream.
- findFirst, findAny : opérations terminales qui permet de trouver des éléments particuliers d'un stream



```
public class ProjetStreams1 {
```

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

```
        // interfaces fonctionnelles utiles
```

```
        Function <Integer,Double> fun = new Function <Integer,Double> (){      public Double apply (Integer i){return i*10.0;} };
        Consumer <String> cons = new Consumer <String> (){      public void accept (String s){System.out.println(s+" is consumed");} };
        Supplier <String> supp = new Supplier <String> (){      public String get(){return "Success";} };
        Predicate <Double> pre = new Predicate <Double> (){      public boolean test (Double n){return n<20;} };
```

```
    // écriture par expressions Lambda
```

```
        Function <Integer,Double> fun1 = i->i*10.0;
        Consumer <String> cons1 = s->System.out.println(s+" is consumed");
        Supplier <String> supp1 = ()-> "Suceess";
        Predicate <Double> pre1 = n-> n<20;
```

```
    // création de d'une liste avant Java 9
```

```
        List <String> liste = new ArrayList();
        liste.add("Camion");liste.add("voiture"); liste.add("train");
        liste.forEach(System.out::println);
```

Camion
Voiture
train

```
    // en une seule instruction
```

```
        Arrays.asList("Camion","voiture","train").forEach(System.out::println);
```

Camion
voiture
Train

```
    // autres collections
```

```
        Map <Integer,String> map = new HashMap();
        map.put(1,"Camion"); map.put(2,"voiture"); map.put(3,"train");
        map.entrySet().forEach(System.out::println);
```

1=Camion
2=voiture
3=train

```
    // interface fonctionnelle UnaryOperator
```

```
        UnaryOperator<Integer> operator = t -> t * 2;
        System.out.println(operator.apply(5));
        System.out.println(operator.apply(10));
        System.out.println(operator.apply(15));
        UnaryOperator<Integer> operator1 = t -> t + 10;
        UnaryOperator<Integer> operator2 = t -> t * 10;
        int a = operator1.andThen(operator2).apply(5);
        System.out.println(a);
        int b = operator1.compose(operator2).apply(5);
        System.out.println(b);
```

10
20
30

150
60

```
    // autres
```

```
        DoubleBinaryOperator operator3 = (p, q) -> (p + q);
        System.out.println(operator3.applyAsDouble(5, 6));
```

11.0

```
    // Il existe des interfaces fonctionnelles spécifiques au Java 8:
```

```
    // LongUnaryOperator, LongToIntFunction, IntSupplier, DoubleConsumer,
```

```
    // IntToDoubleFunction, LongPredicate
```

```
    }
}
```




```

public class ProjetStreams2 {

    public static void main(String[] args) {
        Stream <Integer> str = Stream.of (3,9,7,5,1);
        str. forEach (e->System.out.print(e+ " ")); // 3 9 7 5 1

        IntStream str1 = IntStream.of(4,3,2,9,4);
        str1.forEach(e-> System.out.print(e+" ")); //4 3 2 9 4

        // generate (Supplier <T> s)
        Stream.generate(()-> "bonjour "); // Stream infini de String (bonjour)
        Stream.generate(Math::random); // Stream infini
        Stream.generate(Math::random).limit(8).forEach(e-> System.out.print(e+" "));
        // 0.9779699305994837 0.7703551433855831 0.6167040553120374 0.7910158538290651
        // 0.8796445810342365 0.3634712383918377 0.9174521161507594 0.9529296661567758

        // iterate (T seed, UnaryOperator <T> f)
        Stream.iterate(5,(e-> e+2)).limit(10).forEach(e-> System.out.print(e+" "));
        // 5 7 9 11 13 15 17 19 21 23

        IntStream.range(20,23).forEach(e-> System.out.print(e+" "));
        // 20 21 22

        IntStream.rangeClosed(20,23).forEach(e-> System.out.print(e+" "));
        //20 21 22 23

    }
}

```

