Streams and Lambda expressions

Table des matières

I. Working with lambda :

- ***Intro :***

Principe de base : raccourcir le code.

Définition : c'est une méthode **anonyme** qu'on peut passer en **paramètre**.

Concept : On traite le code comme des données et on peut les passer à une méthode.

- ***Les interfaces fonctionnelles :***

Elles sont prédéfinies par Java et sont des classes interfaces qui comprennent **1 seule** méthode abstraite qu'on peut implémenter avec un lambda.

Un lambda implémente **TOUJOURS** une interface fonctionnelle. (Elle se trouvent dans le pack java.util.function).

- ***Syntaxe :***

L'expression lambda se compose :

- D'une liste de paramètres.
- D'une flèche : ->
- D'un corps.

Elle n'a pas de nom car elle est anonyme.

Elle n'a pas d'accesseur car elle ne fait pas partie d'une classe.

On ne spécifie pas les types car le compilateur le sait déjà (cela dit il est quand même possible de le faire).

Pour rappel, une méthode comprend :

Un accesseur, un type de retour, un nom, une liste de paramètre, un corps.

Si un lambda n'a pas de paramètre, il y'a deux parenthèses vides :

```
Runnable runn = () -> System.out.println("Hello world");
```

Si elle comprend **1 seul** paramètre, pas besoin de parenthèses :

```
Filter filter = file -> file.isHidden();
```

L'expression **retour** est **implicite** :

Ceci entre accolade et avec le mot Return:

```
(p1, p2)->{
    return p1.getPrice().compareTo(p2.getPrice());
}
```

Est la même chose que ceci sans accolade et sans return :

```
(p1, p2 -> p1.getPrice().compareTo(p2.getPrice()))
```

Si elle comprend plusieurs expressions, il faut d'office des accolades après la flèche.

- ***Accéder aux variables locales (capturing variables) :***

Règle principale : Elles doivent être **finales** ou **effectivement définitives** (effectivly final).

Elle doit se comporter comme si elle avait le mot clé finale, c'est-à-dire qu'on lui attribue une valeur qui ne changera pas. Cela est du au fonctionnement des lambdas qui utilisent une copie de la variable plutôt que son original.

L'exécution du lambda ne se fait pas lors de sa déclaration dans une variable mais lorsque la variable dans laquelle elle est déclarée est utilisée.

Si, juste après la déclaration du lambda, on change la valeur de la variable que le lambda utilise plus loin, on génère une exception.

- ***This et Super dans les lambdas :***

Une classe anonyme est comme une classe normale, elle peut avoir des **variables membres**.

On peut y avoir accès en utilisant le mot clé **This** :

```
public static void printMessage(){
    Runnable runnable = new Runnable() {
        private String message ="Hello world";
        @Override
        public void run() {
            System.out.println(this.message);
        }
    };
```

- ***Les exceptions dans les lambdas :***

```
List<Product> products = ExampleData.getProducts();

        try (FileWriter writer = new
FileWriter("products.txt")) {
//            for (Product product : products) {
//                writer.write(product.toString() + "\n");
//            }

            // According to forEach(), the lambda expression
implements interface Consumer. The accept() method
            // of this interface does not declare any checked
exceptions, so the lambda expression is not allowed
            // to throw any checked exceptions. We are forced
to handle the IOException inside the lambda expression.
            products.forEach(product -> {
                try {
                    writer.write(product.toString() + "\n");
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            });
        } catch (IOException | RuntimeException e) {
            System.err.println("An exception occurred: " +
e.getMessage());
        }
```

Ici ce qu'il faut retenir est simplement que selon les cas, il n'est pas toujours utile de travailler avec un lambda car on perd son avantage syntaxique, c'est-à-dire de dire quelque chose en moins de ligne.

- ***Méthodes références :***

Si je veux faire un print, je fais en réalité juste appel à une autre méthode :

```
products.forEach(product -> System.out.println(product));
```

On peut encore supprimer cette ligne en disant à forEach qu'il doit faire appel à printLn directement.

C'est possible si on utilise **une référence de méthode** :

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

C'est en vérité un pointeur qui dit quelle méthode utiliser. On voit qu'il n'y a pas de paramètre ici car il est **IMPLICITE**. Ce sera product

Les méthodes qui peuvent être utilisées dépendent de l'interface fonctionnelle. Dans ce cas-ci c'est une interface **CONSUMER** donc ce sont des **VOID** qui ne prennent qu'**1** paramètre.

Syntaxe :

Après les doubles deux points, se trouve le nom de la méthode à laquelle on fait référence. Cette méthode peut être :

- Static
- Une méthode d'instance
- Un constructeur

Ce qu'on écrit avant les deux points dépend du type de méthode auquel on se réfère. Ici on se réfère à la classe. C'est une référence à un **STATIC**.

Pour la précédente on faisait appel à la méthode println d'un objet PrintStream particulier auquel fait référence System.out. C'est une référence à une méthode d'**INSTANCE**.

```
public class LambdasExample07 {


    static boolean isExpensive(Product product) {
        return product.getPrice().compareTo(new
BigDecimal("5.00")) >= 0
}
//renvoie true quand le produit coute 5 ou plus


        // Method reference to a static method.
        products.removeIf(LambdasExample07::isExpensive);
```

Pour une référence à un constructeur :

```
ProductFactory factory = Product::new;
```

Résumé :

Four Types of Method References
Static method = > ClassName :: methodName
Instance method of a specific object => objectRef :: methodName ex :(System.out ::print)
Instance method not of a specific object => ClassName :: methodName
Constructor => ClassName :: new

II. Les interfaces fonctionnelles :

- ***Compréhension :***

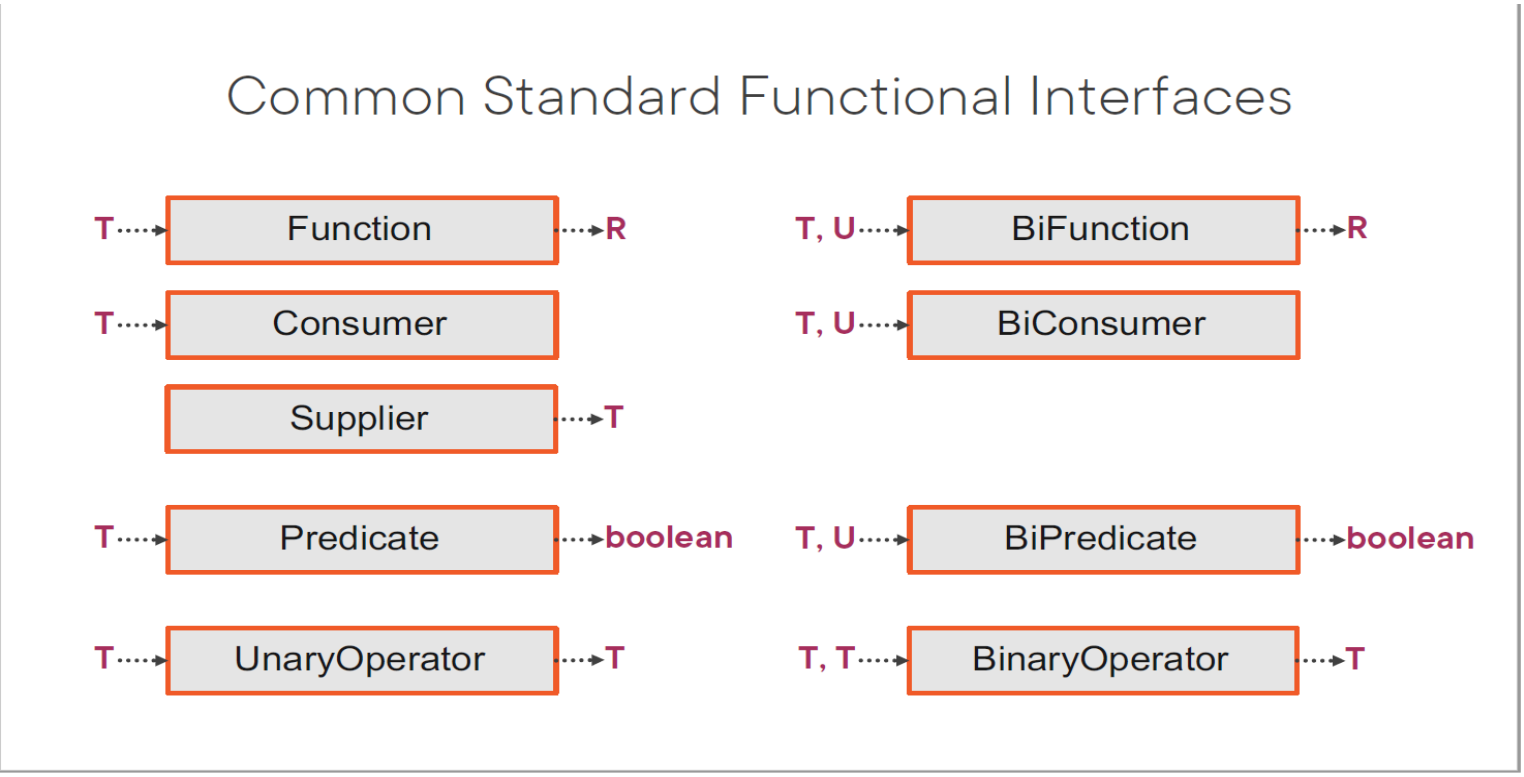
Une interface fonctionnelle **est une interface avec une seule méthode abstraite**.

L'interface peut avoir d'autres méthodes mais **elles ne peuvent pas** être abstraite.

Elles sont distinguées par l'annotation : **@FunctionnalInterface**

Cela a pour but de préciser qu'elles doivent être utilisées comme tels. Si on fait une interface fonctionnelle et qu'on met cette annotation le compilateur vérifiera qu'elle est bien fonctionnelle.

- ***Les interfaces fonctionnelles communes :***



Il y'a les inputs et les outputs. Il peut n'y avoir aucun input ou aucun output mais pas les deux. Runnable ???

Les plus importantes sont Function, Consumer et Supplier.

Toutes ces interfaces ont des méthodes qui ne sont pas vides et qui peuvent être utilisées.

Ex : Predicate.test(product) renvoie true si c'est bien un objet Product.

- ***Les objets optionnels :***

Optional = Object conteneur**. Peut contenir** ou pas une valeur **NON-Nulle**.

C'est une alternative plus sécurisée qu'utiliser null.

Ici on déclare qu'on a potentiellement un string dans la déclaration de méthode avec Optional<String>

```
public Optional<String> method(){
    String returnValue = null;
    // Code
    return Optional.ofNullable(returnValue);
}
```

Si une valeur est présente, optionnal.isPrsent() = TRUE et .get() renvoie sa valeur.

Si il est possible que cela renvoie un null, on utilise la méthode ofNullable() qui renvoie un objet optionnel vide sans exception.

```
private static void Student(String name, String lastName, int
age){
        Optional<String> ln = Optional.ofNullable(lastName);
        String a = ln.isPresent() ? ln.get() : "Not Given";
        System.out.println("name : "+name + ", lastname : "+a
+", age : "+age);
    }
```

- ***Functional Composition :***

Cela consiste à combiner des fonctions dans une nouvelle fonction.

Par exemple ces deux fonctions .map au sein d'une seule :

```
findProduct(products, product ->
product.getName().equals(name))
        .map(Product::getPrice)
        .map(price -> String.format("XXXX"))
```

D'abord on extrait ces deux fonctions vers des variables locales qui reprennent ces lambdas:

```
Function<Product, BigDecimal> productToPrice =
Product::getPrice;

Function<BigDecimal, String> priceToMessage = price ->
String.format("The price of %s is $ %.2f%n", name, price);
```

Ensuite on combine ces deux fonctions en une seule, en utilisant la méthode **andThen** :

```
Function<Product, String> productToMessage =
productToPrice.andThen(priceToMessage);
```

Cela donne comme résultat idem que le premier mais avec une seule ligne .map.

```
findProduct(products, product ->
product.getName().equals(name))
        .map(productToMessage)
```

- ***Specialized standard Functionnal Interfaces :***

Ce sont des versions spécialisées faites pour travailler avec des types primitifs.

Exemple : Arraylist<**Integer** >. On ne peut pas mettre **int**, car on ne prend pas en compte les primitifs.

Exemple d'interfaces fonctionnelles utilisant un int : IntBinaryOperator

C'est la même chose que binaryoperator mais prévu pour des int. L'intérêt est de **gagner en performance** et éviter des étapes supplément pour l'obtention d'un résultat.

Seuls les types **INT, LONG et DOUBLE** sont disponibles. Une seule exception est **BooleanSupplier** qui prend un primitif booléen.

III. Working with Streams – The Basics

- ***Définition :***

But : Traiter une séquence d'élément en exécutant différent types d'opérations dessus.

```
products.stream()
        .filter....
        .map ...
        .forEacht(sout)...
```

Toutes les méthodes qui commencent avec un point composent le **pipeline de flux** (**stream Pipeline**).

Ce pipeline **utilise la programmation fonctionnelle** car on a des expressions lambdas et des références de méthode.

Exemple d'une mauvaise façon de faire qui va contre la programmation fonctionnelle car on modifie une liste extérieure en itérant dans le flux avec le forEach.

```
ArrayList<String> prenomSorted = new ArrayList<>();
ArrayList<String> prenom = new
ArrayList<>(Arrays.asList("Seb", "Jean ", "Francois", "Santi",
"Sacha")) {};
prenom.stream()
        .filter(item -> item.contains("S"))
        .forEach(prenomSorted::add);
```

Le stream peut être créer à partir de tout ce qui fournit une **séquence d'élément**. Par exemple une collection ou un io channel.

Il y'a deux types d'opération dans le pipeline :

Opérations **intermédiaires** : Ce sont celles du milieu (Ici map et filter).

Opérations **terminales** : C'est celle de la fin et elle est obligatoire sinon les intermédiaires ne sont pas faites. (Ici c'est la forEach)

Pour savoir si elles sont terminales ou non il faut regarder la documentation. De manière générale celle qui **renvoie un stream** sont intermédiaires et ce qui **renvoie autre chose** sont terminales.

- ***Différences avec les collections :***

Les streams :

Ne **stockent rien**.

Sont **fainéants** (font rien si pas de méthode terminale).

Désigné pour la **programmation fonctionnelle** (veut dire qu'on veut éviter d'avoir un état mutable, cad on applique des fonctions à des valeurs immuables qui renvoient d'autres valeurs immuables).

Ne **modifie jamais la source** du stream : du à la nature fonctionnelle. Si on utilise sorted, par exemple, il ne trie pas les éléments du stream mais créé un nouveau stream avec ces éléments triés.

Itérer sur un stream le **consume**. Pas de retour en arrière donc.

Peuvent être **infinis.** Car il ne doit pas les stocker.

Les itérations :

Externe : Avec une boucle for par exemple car l'itération n'est pas contrôlée par la collection.

Interne : On appel une opération terminale comme .forEach.

- ***Obtenir un stream :***

```
// Get a stream from a collection
Stream<Product> stream1 = products.stream();

// Get a stream from an array
String[] array = new String[]{"one", "two", "three"};
Stream<String> stream2 = Arrays.stream(array);

// Create a Stream from elements directly
Stream<String> stream3 = Stream.of("one", "two", "three");

// Create a Stream with zero or one elements with ofNullable()
Stream<String> stream4 = Stream.ofNullable("four");

// Create an empty Stream with Stream.empty()
Stream<?> stream5 = Stream.empty();
```

Il existe encore d'autre moyen d'obtenir un stream.

La méthode ofNullable peut être utile si on a besoin d'un stream vide. C'est d'ailleurs la même chose que la dernière ligne.

- ***Filtrer et transformer un stream :***

Pour filtrer on appel **.filter** qui prend un **predicate**.

```
products.stream()
        .filter(product -> product.getCategory() ==
Category.FOOD)
        .forEach(System.out::println);
```

Pour transformer les éléments on fait appel à **.map** qui prend un **function**.

```
products.stream()
        .map(product -> String.format("The price of %s is $
%.2f", product.getName(), product.getPrice()))
        .forEach(System.out::println);
```

Un autre moyen de transformation est .flatMap qui transforme **plusieurs élément en un seul**.

```
//Le nom des produits se compose parfois de pllusieurs mots
//Pour certain produit cela renverra plusieurs String
products.stream()
        //avec map on a un flux contenant des flux de Strings
        //avec flatmap on a juste un flux contenant des
Strings
        .flatMap(product ->
spaces.splitAsStream(product.getName()))
        //renvoie un flux de valeurs
       .forEach(System.out::println);
```

- Rechercher dans les Streams

Il existe plusieurs moyens :

Il faut avant tout filtrer le flux :

```
       .filter(product -> product.getCategory() ==
Category.OFFICE)
```

Attention aux **renvois d'optionnels** !!! Obligé car il pourrait ne pas y avoir de résultat.

FindFirst : va trouver le premier élément dans le flux. **Pas de paramètre** et c'est une **opération terminale**. Ici on aura comme résultat : Seb

```
ArrayList<String> prenom = new
ArrayList<>(Arrays.asList("Seb", "Jean ", "Francois", "Santi",
"Sacha")) {};
Optional <String> result  = prenom.stream()
        .filter(item -> item.contains("S"))
        .findFirst();
System.out.println(result);
```

FindAny() : fait quasiment la même chose que findFirst mais utilisé lorsque le stream est issu de collection non-ordonnée. On demande de trouver tout élément qui correspond aux critères de recherche. On ne se soucie pas exactement lequel. Un hashSet par exemple est non ordonnée.

AnyMAtch : Prend un prédicat donc n'a pas besoin d'un filter avant. Elle combine le filter et le FindAny. Renvoie un true si un seul élément correspond au critère.

allMatch : Renvoie un true si tous les élément sont correspondent au critère.

noneMatch : idem que précédent mais en sens inverse.

Certaine opération de recherche sont en **court-circuit** : veut dire qu'elle stopperont dès qu'elles auront un résultat : findFirst, anyMatch et findAny.

- ***Reducing and Collecting Streams :***

Principe de base, on itère parmi des éléments qu'on fini par combiner en un résultat final qui est réduit.

Une des opérations les plus courantes est de stocker cette réduction **dans une collection** en utilisant to list ou .collect(Collectors.toList())

On voit ici que to list renvoie une List et non un ArrayList.

```
ArrayList<String> prenom = new
ArrayList<>(Arrays.asList("Seb", "Jean ", "Francois", "Santi",
"Sacha")) {};
List<String>result = prenom.stream()
        .filter(item -> item.contains("S"))
        .toList();
```

Dinstinct() : permet de supprimer les duplicats, donc si j'ai deux fois seb, je n'en aurai qu'un seul.

IV. Working with Streams in depth :

- ***Générer et construire des Stream :***

Il existe d'autres méthode pour en créer :

Generate : génère un flux infini de valeurs produites par un fournisseur.

Ici on générère un stream de **UUID** qui sont des objets de 36 charactères : 123e4567-e89b-12d3-a456-556642440000 et on montre les dix premiers.

```
Stream<UUID> uuids = Stream.generate(UUID::randomUUID);
uuids.limit(10).forEach(System.out::println);
```

Iterate : la première version génère à l'infini comme Generate. Il prend deux paramètres :

un **seed** : qui est le premier élément du stream

un **unaryOperator** : la manière d'itérer

Exemple avec un objet de type BigInteger qui permet d'avoir de très gros nombre :

```
Stream<BigInteger> powersOfTwo =
Stream.iterate(BigInteger.ONE, n ->
n.multiply(BigInteger.TWO));
powersOfTwo.limit(20).forEach(System.out::println);
```

Iterate : la deuxième version prend un troisième paramètre : **un prédicat** qui va déterminer s'il y'a un élément suivant qui est ici après letter<='z ' . On récupère les lettres de l'alphabet.

Cette ligne de code peut se lire comme une **boucle for**.

```
Stream<Character> alphabet = Stream.iterate('A', letter ->
letter <= 'Z', letter -> (char) (letter + 1));
```

Builder : Permet d'ajouter des élément au flux.

```
Stream.Builder<String> builder = Stream.builder();
builder.add("one");
builder.add("two");
builder.add("three");
Stream<String> stream = builder.build();
stream.forEach(System.out::println);
```

- ***Reducing Stream in details :***

En plus de ceux déjà vu il en existe d'autres :

Reduce : prend un **binaryOperator,** le sous-total et le prochaine élément et renvoie un optionnel. Il y'a aussi une version avec un deuxième opérateur, qui prend la valeur initiale du résultat final. Cette version retournera une **valeur** et non un optionnal.

```
Optional<BigDecimal> opt = products.stream()
        .map(Product::getPrice)
        .reduce((result, element) -> result.add(element)); //
Can also be written with a method reference: BigDecimal::add
```

```
// The third version of reduce() is the most general
// The type of its result value may be different than the type
of the elements in the stream
// The third parameter is a combiner function to combine
intermediate results; this is useful for example for a
parallel
// stream, where different threads compute intermediate
results that have to be combined into a final result
BigDecimal total2 = products.stream().reduce(
      BigDecimal.ZERO,
      (result, product) -> result.add(product.getPrice()),
      BigDecimal::add);
System.out.printf("The total value of all products is: $
%.2f%n", total2);
```

- ***Collecting Streams in details :***

Une opération de collecte est une **opération de réduction** dan laquelle le résultat fini dans un conteneur de résultat mutable (ex une collection).

Méthode **collec**t() : Prend trois paramètres :

Un fournisseur : L'élément (la liste) qui contiendra le résultat.

Un accumulateur : Accumule les éléments du flux dans le conteneur, c'est un **Biconsumer**. (BiFunction chez Reduce).

Un combineur : Emerge le contenu des deux conteneur de résultats intermédiaires en un seul. C'est aussi un **Biconsume**r (BinaryOperator chez Reduce).

Il y'a similitude avec la méthode Reduc qui prend trois trois opérateurs. La différence est que Reduce est prévu pour une **Immutable reduction** et collect pour une **Mutable reduction**.

Faire cette opération avec un reduc aurait été bcp plus long car il travaille qu'avec des objets immuables, il aurait fallu que je fasse une liste temporaire pour stocker.

```
ArrayList<String> prenom = new
ArrayList<>(Arrays.asList("Seb", "Jean ", "Francois", "Santi",
"Sacha")) {};
List<String>result = prenom.stream().collect(
        ArrayList::new, //FOURNISSEUR
        (list, name)->list.add(name), //ACCUMULATEUR
        ArrayList::addAll); //COMBINEUR
```

Lequel utiliser ?

Si le conteneur du résultat est modifiable = collect().

Sinon : reduce().

- ***Working with Collector :***

Quand on travail avec collector, pas besoin d'implémenter l'interface nous-même.

Toset et toList on déjà été vu.

Voici un exemple avec **toMap**, et comme cela crée un objet map on a besoin de clef, on aura ici le total par catégorie.

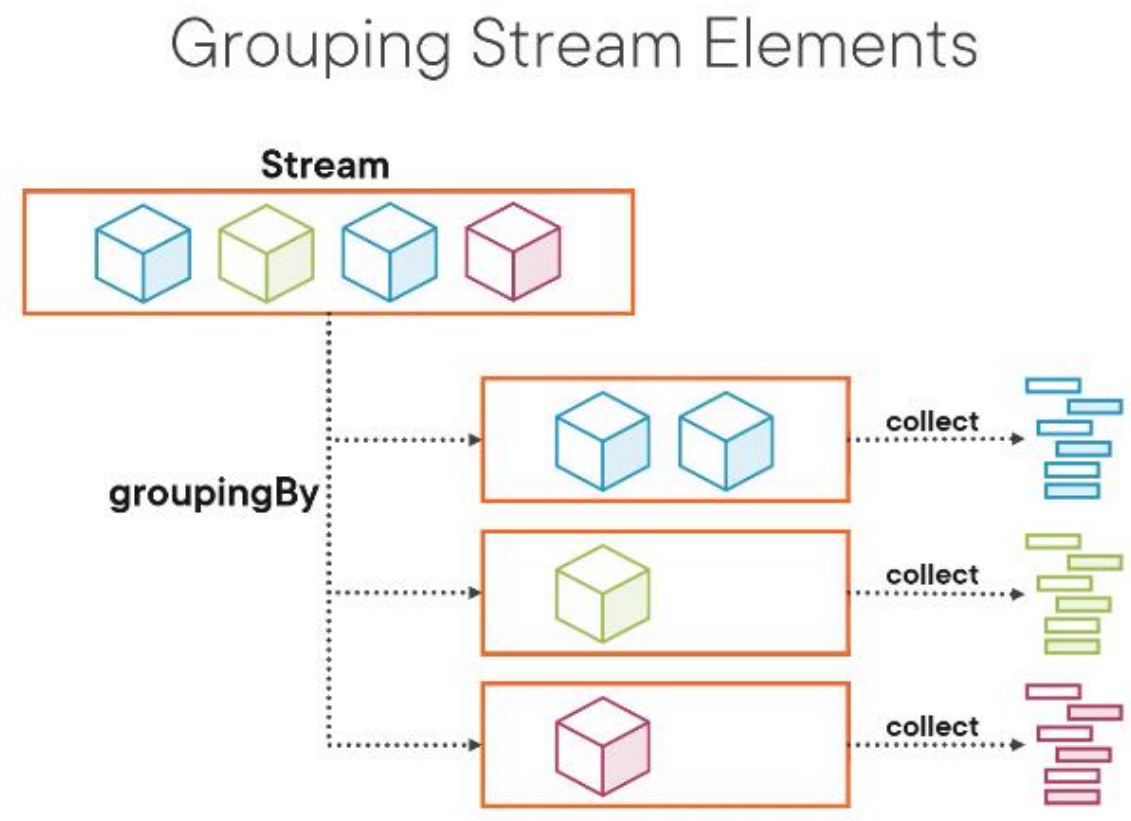
```
// Using Collectors.toMap() to compute the total price of
products per category.
Map<Category, BigDecimal> totalPerCategory = products.stream()
        .collect(Collectors.toMap(
                Product::getCategory, // Key mapper function
                Product::getPrice,    // Value mapper function
                BigDecimal::add));      // Merge function
Result :
{CLEANING=12.16, UTENSILS=56.49, FOOD=45.50, OFFICE=25.25}
```

- ***Grouping Stream Elements :***

On utilise ici **groupingBy**() :

Cela nous permet de regrouper les élément par catégorie qui seront des cléfs dans une map.

```
Map<Category, List<Product>> productsByCategory =
products.stream()
        .collect(Collectors.groupingBy(Product::getCategory));
```



Le grouppinBy fait une **division du stream en multiples Stream**, un par catégorie. Pour créer les **valeurs** de la map il faut aussi collecter chacun des flux des différentes catégorie.

Il faut donc appliquer une map() opération pour chacun de ces flux, ici pour les transformer en flux de noms de produits et ensuite les collecter en liste. C'est la méthode **mapping**() qui s'occupera de ça. Elle prend deux paramètres : Une function et un collecteur.

```
Map<Category, List<String>> productNamesByCategory =
products.stream()
        .collect(Collectors.groupingBy(Product::getCategory,
Collectors.mapping(Product::getName, Collectors.toList())));
```

- ***Partitioning Stream Elements :***

Il s'agit de séparer les éléments selon des clefs qui seront des predicats à la base. Ici on sépare les éléments en bon marché, et cher avec **partitioningBy**.

Ici priceLimit est un BigDecimal de valeur 5.00.

```
Map<Boolean, List<Product>> partitionedProducts =
products.stream()
        .collect(Collectors.partitioningBy(
             product -> product.getPrice()
             .compareTo(priceLimit) < 0));
```

Si on veut inspecter le résultat :

```
System.out.println("Cheap products: ");
partitionedProducts.get(true).forEach(System.out::println);

System.out.println("Expensive products: ");
partitionedProducts.get(false).forEach(System.out::println);
```

Il existe d'autre possibilité d'opération identique à celle rencontrée avec groupingBy.

- ***Parallel Streams :***

Les flux avec lesquels on travaillé jusqu'à présent étaient **SEQUENTIELS**.

Ici il s'agit de travailler avec des streams **PARALLEL**.

Il suffit de changer une seule chose dans le code : stream() devient **paralellStream()**.

Cette simplicité est du au fait qu'on utilise une itération interne. Quand l'itération est externe, on doit programmer tous les détails nous-même, exemple avec une boucle for. Si on devait en plus gérer le multiThreading cela rendrait le code extrêmement compliqué.

Avec une itération interne, on ne s'occupe pas des détails. Il fera alors du multiThreading de bas niveau lui-même.

Cela a un cout et n'est pas forcément nécessaire. Cela peut être limité par le cpu. Pour savoir quelle méthode est la plus rapide, il faut la mesurer.

La méthode grouppingBy n'est pas efficace en parallèle, au lieu de ça il faut utiliser **groupingByConcurrent** mais c'est non ordonné.

- ***Specialized Streams :***

Ils sont spécialisés pour les primitifs : IntStream, LongStream, DoubleStream.

Ils ont quelques méthodes supplémentaires :

Comparé à Stream<Double> prices, cette façon de faire évite le **boxing/unboxing** problème.

```
DoubleStream prices = products.stream()
        .mapToDouble(product ->
product.getPrice().doubleValue());
```

Ici la méthode sum est propre à ce stream.

```
double total = prices.sum();
```

Ici on le met dans un objet d'un autre type qui comporte un tas de méthode utile pour avoir les sommes, les maximum, les minimum, etc… avec statistic.get…Max,Min…etc.

```
DoubleSummaryStatistics statistics = products.stream()
        .mapToDouble(product ->
product.getPrice().doubleValue())
        .summaryStatistics();
```