Comment faire le passage de JDK7 à JDK8 ? ou JDK8, les nouveautés.

TL;DR[1] *JDK8 ce ne sont pas quelques nouveautés . . . ce sont des packages revus, des packages ajoutés et des ajouts importants dans la conception du langage.*

Un peu de cosmétique

Pour commencer doucement, la **javadoc** a reçu un petit coup de *css* et passe aux onglets.

Les optionals

Toujours pour commencer avec des petites choses, *jdk8* voit apparaitre les **optionals**.

Parfois, une variable doit avoir une valeur par défaut et l'on ne sait pas toujours en trouver une. Si c'est le minimum des naturels, 0 convient très bien . . . si c'est le maximum, c'est plus délicat. *jdk8* propose les classes `Optional<E>`, `OptionalDouble`, `OptionalInt`, `OptionalLong` et `OptionalDataException` dans le *package* `java.util`. Ces classes auront, ou pas, une valeur.

[1] Too long; don't read signale que le billet n'est pas lu parce que trop long. J'en fait ici un résumé . . . très très résumé ;-)

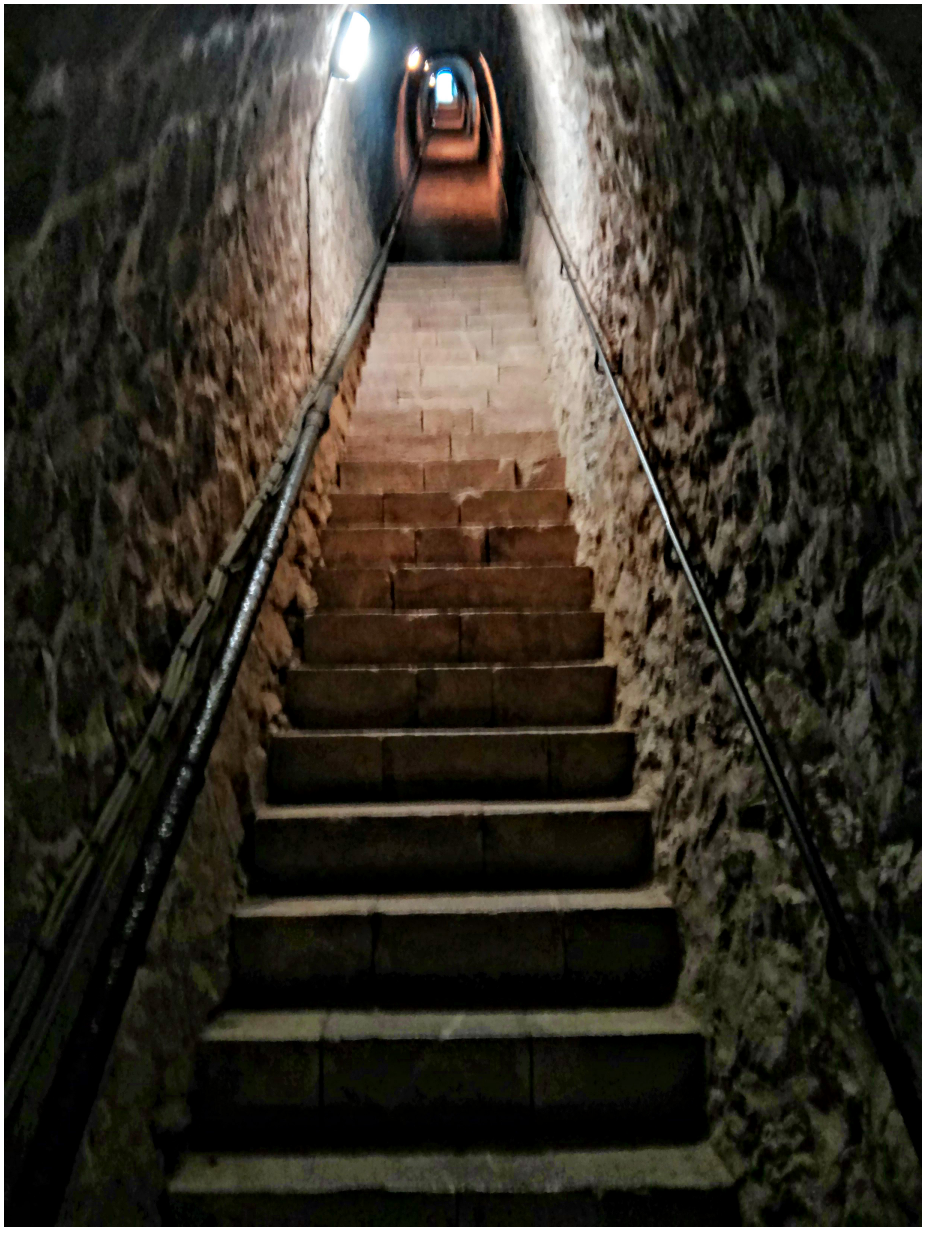


Figure 1: Les escaliers des milles marches

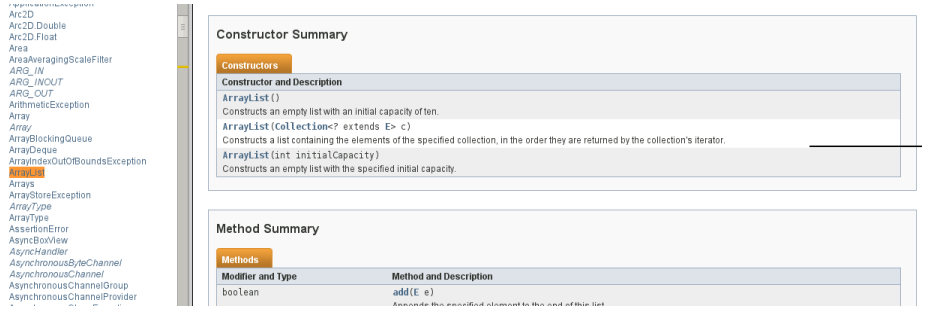


Figure 2: javadoc, JDK7

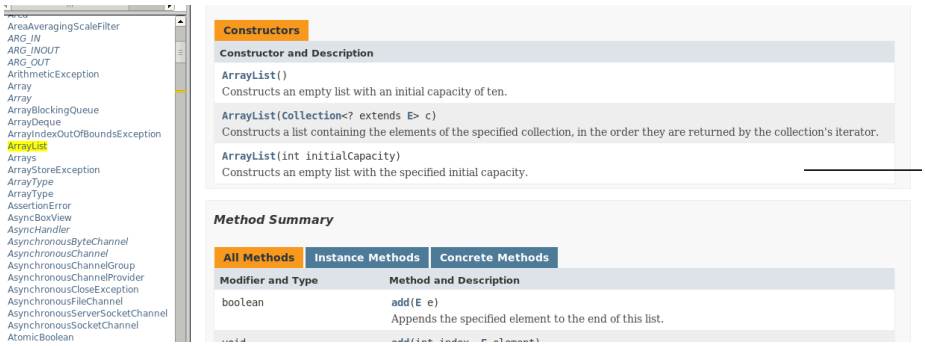


Figure 3: javadoc, JDK8

```
Optional<Integer> oi = Optional.of(7);
//Optional<Integer> oi = Optional.empty();

System.out.printf("Or else: %d\n", oi.orElse(0));
if (oi.isPresent()){
    System.out.printf("Get: %d\n", oi.get());
}
```

L'intérêt est assez moyen à ce stade ... mais se marquera dans la suite ... et puis, c'est pour commencer doucement. L'article est long, les changements sont nombreux.

Où l'on parle du temps

jdk8 voit arriver de nouveaux *packages* pour la gestion des dates (exit donc `GregorianCalendar` et c'est pas trop tôt). `java.time` et ses petits frères `java.time.chrono`, `java.time.format`, `java.time.temporal` et `java.time.zone` vont nous permettre de parler d'**instants** (*instant*), de **durée** (*duration*) et ...

Il sera donc possible de définir des instants ainsi que le laps de temps (la durée) écoulé entre deux instants grâce aux classes `Instant`, `Duration`, ... par exemple comme suit:

```
Instant now = Instant.now();
Instant twominutes = now.plus(Duration.ofMinutes(2));
Instant nowagain = Instant.now();
Duration duration = Duration.between(now, nowagain);
System.out.println("Duration: " + duration.toMillis() + "ms");
```

Si l'on ne s'intéresse pas à l'heure mais aux jours qui passent, les classes `LocalDate`, `Period`, `Month`, ... prendront le relai ...

L'exemple est assez parlant:

```
LocalDate myBirthday = LocalDate.of(1971, Month.AUGUST, 14);
Period age = myBirthday.until(LocalDate.now());
System.out.println("My age: " + age.getYears() + " years old");
System.out.println("My age: " + age.toTotalMonths() + " months");
DayOfWeek myBirthdayDayOfWeek = DayOfWeek.from(myBirthday);
System.out.println(myBirthday.format(
        DateTimeFormatter.ofPattern("d MMMM u"))
    + " was a "
    + myBirthdayDayOfWeek.getDisplayName(TextStyle.FULL,
        Locale.ENGLISH));
```

```
LocalDate firstMonday =
    myBirthday.with(TemporalAdjusters.firstInMonth(
        DayOfWeek.MONDAY));
System.out.println("In 1971, the first monday in august was the "
    + firstMonday.getDayOfMonth());
```

Il est évidemment possible de passer de la nouvelle API à l'ancienne et *vice versa*.

Petites modifications dans la classe String

La concaténation des chaines qui se faisant grâce à `StringBuilder.append()` peut maintenant se faire par le biais de `StringJoiner.add()`. L'avantage réside dans le fait que l'on peut spécifier un délimiteur entre tous les éléments.

Il est donc plus facile d'écrire *one, two, three* (notez la présence de la virgule) et même *[one - two - three]* (notez maintenant la présence des tirets et des crochets).

```
StringJoiner sj123 = new StringJoiner(" - ","[", "]");
sj123.add("one").add("two").add("three");
System.out.println(sj123);

$ [one - two - three]
```

La méthode `String.join()` permet de faire directement le boulot.
Facile !

```
System.out.println(
        String.join(", " ,"one", "two", "three"));
```

Et pourquoi pas directement sur un itérable (*Iterable*) ?
(Je choisis une liste, un tableau ou tout autre itérable fait l'affaire)

```
List<String> strings = Arrays.asList(
    new String[] {"one", "two", "three"});
System.out.println(
    "["
    + String.join(" - ", strings)
    + "]");

$ [one - two - three]
```

Question – Comment savoir si c'est plus cher ou moins cher que d'écrire une boucle pour et de concaténer avec + ?

Certaines nouveautés *jdk8* (*optionnals*, *default methods*) semblent être là uniquement pour permettre l'introduction des *lambdas expressions*. J'en parle volontairement avant pour alléger la section *lambdas*.

Les interfaces sont revues

(**default methods**)

En JDK<=7 une interface définit un contrat que toutes ses implémentations doivent respecter. Dès lors qu'une interface est définie (depuis longtemps), la modifier devient bancal puisque cette modification impacte toutes les implémentations.

Imaginons que l'on veuille ajouter une méthode à une interface.

Cela implique que toutes les classes implémentant l'interface ajoutent une implémentation de la fonction. Ce n'est peut-être pas viable.

jdk8 propose la notion de « méthode par défaut » (**default method**) permettant de donner une implémentation par défaut pour une méthode.
On parlera en détail de la classe `Stream` et de la méthode homonyme plus bas. Imaginons que l'on veuille affubler l'interface *Collection* d'une nouvelle méthode *stream()*. Pour ne pas casser toutes les implémentations de *Collection*, l'interface définit une implémentation par défaut grâce au mot clé `default`.

```
public interface Collection<E> {
    public add(E e);
    // le reste

    default Stream<E> stream() {
        // implémentation par défaut pouvant être réécrite
        return ...
    }
}
```

Ce principe d'implémentation de méthode par défaut résout le problème de compatibilité mais entraine une sorte d'**héritage multiple**. En effet, comme je peux implémenter plusieurs interfaces, rien n'empêche que ces interfaces contiennent des méthodes par défaut.

Java propose donc un héritage multiple en terme d'implémentation de méthode. Cet héritage peut mener à des conflits lors de la compilation qui seront réglés comme suit:

- la classe gagne par rapport aux interfaces qu'elle implémente;
- l'interface la plus spécifique gagne;
- sinon, le compilateur pleure ...

Si les interfaces proposent d'office des méthodes par défaut, il ne sera plus nécessaire, lorsque l'on implémente une interface, d'écrire toutes les méthodes (même avec un corps « bidon ») ... ce qui change (radicalement) l'angle de vue.

lambda expressions (enfin)

Les *lambdas expressions* en Java sont des **méthodes anonymes**. Comme le nom l'indique, ce sont des méthodes qui n'ont pas de nom.

Ce que je pouvais déjà faire dans une classe (j'adapte un peu l'exemple de Wikipedia), définir une méthode:

```
int addition(int a, intb) {
    return a + b;
}
```

Avec les *lambdas expressions* en *jdk8*, je peux maintenant m'abstraire du nom et des types; paramètres et type de retour. Si j'utilise l'opérateur `->` à la place de *return*, il reste[2]:

```
(a,b) -> a + b
```

Je ne peux pas écrire cette *lambda* en l'état. Une *lambda expression* n'est pas une instruction. L'utilisation des *lambdas* sera fortement liée aux interfaces ne contenant qu'une seule méthode abstraite. Ces interfaces me permettront d'écrire:

```
Interface Operation {
    int operation(int a, int b);
}
// ...
Operation addition = (a, b) -> a + b;
Operation substraction = (a, b) -> a - b;
```

Les *lambdas expressions* permettent une écriture plus compacte dans l'écriture des classes internes anonymes.

Si dans une classe j'ai la méthode suivante:

[2] Dès lors que le compilateur peut inférer le type des variables, il peut être omis.

```
public void operate(int a, int b, Operation o){
    return o.operation(a, b);
}
```

Je peux écrire une classe anonyme à l'ancienne ou bien à la manière de *jdk8*:

```
// old way
o.operate(5, 7, new Operation() {

    @Override
        public int operation(int a, int b) {
            return a * b;
        }
 });
// jdk8 way
o.operate(5, 7, (a, b) -> a + b);
```

Les *lambdas expressions* ne seront pleinement utiles que s'il existe une série **d'interfaces pratiques prédéfinies**.

Rassurez vous, elles existent !

Les interfaces fonctionnelles

La notion de lambdas n'a de sens que s'il existe une série d'interfaces utiles prédéfinies ai-je dit.

Ces interfaces sont appelées **interfaces fonctionnelles** (*functional interfaces*). Ces interfaces fonctionnelles ne contiennent qu'une seule méthode appelée **méthode fonctionnelle**.

Le *package* `java.util.fonctions` en fournit 43 de différentes sortes (et il y en aura d'autres); *function*, *consumer*, *predicate*, *supplier*, *bifunction*, *biconsumer*, *bipredicate*, ... (et les types primitifs).

- Fonction (*function*) prend un objet et en retourne un autre `T → R`
- Consommateur (*consumer*) consomme un objet `T → void`
- Prédicat (*predicate*) prend un objet et retourne un booléen `T → boolean`
- Fournisseur (*supplier*) fournit un objet `void → R`
- S'il y a deux arguments au lieu de un, on préfixe par « bi »
- Pour les types primitifs, on aura par exemple `ToIntFunction`, ...

Ces interfaces fonctionnelles ne possèdent qu'une seule méthode abstraite. On pourra par exemple écrire:

```
Predicate<Object> p8 = t -> t.toString().length() > 17;
Function<String, Person> f8 = s -> new Person(s);
BiConsumer<Person, String> bi8 = (t, u) -> t.setDescription(u);
```

Une interface fonctionnelle ne possède qu'une seule méthode abstraite mais elle peut avoir des méthodes par défaut (*default methods*) ! Par exemple `Predicate` a comme méthodes par défaut `and`, `or`, `negate` et `isEquals`. On ne s'amusera pas à les récrire mais il sera facile de les utiliser.

Pour que l'utilisation des interfaces fonctionnelles soit intéressante, il faudrait que l'API fournisse une série de méthodes prenant en paramètre une de ces interfaces fonctionnelles.

Méthode foreach pour les itérables

On a connu le *for* pour parcourir un peu n'importe quoi tant que l'on pouvait y mettre un indice. On a oublié les itérateurs avec les boucles *for* contenant des *hasNext / next* parce qu'est apparu le *enhanced for* (le *foreach* de son petit nom).

On accueille aujourd'hui la méthode *forEach* qui s'applique à un `Iterable` (mais pas à un tableau). Cette méthode aura besoin d'un *consumer* en paramètre. On pourra par exemple écrire:

```
List<Person> wednesday = ...
wednesday.forEach(t -> System.out.println(t));
```

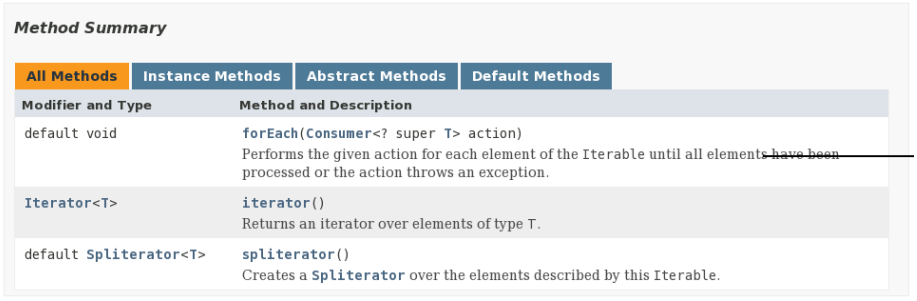
ou dans sa version *old way* (sans *lambdas*),

```
wednesday.forEach(new Consumer<Person>() {

        @Override
        public void accept(Person t) {
            System.out.println(t.getName());
        }
});
```

Sans utiliser les lambdas, l'écriture n'est pas plus compacte que *enhanced for* et l'intérêt est assez limité.

```
for (Person person : wednesday) {
    System.out.println(person.getName());
}
```



Method Summary

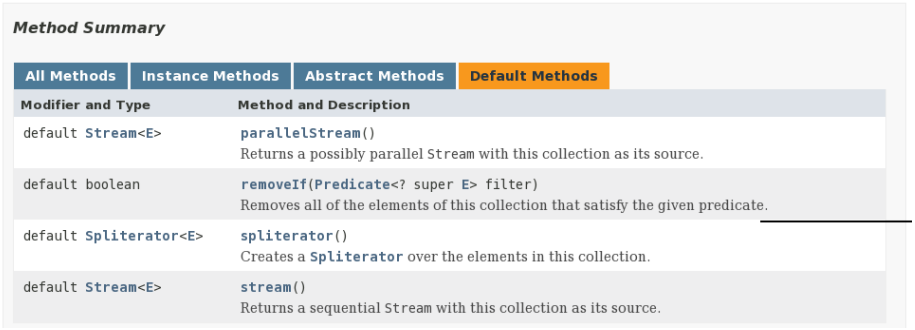
All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
default void	`forEach(Consumer<? super T> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	`iterator()` Returns an iterator over elements of type `T`.
default `Spliterator<T>`	`spliterator()` Creates a `Spliterator` over the elements described by this `Iterable`.

Figure 4: jdk8, iterable

Méthode removeIf pour les collections

Dans le même genre d'approche, *jdk8* voit apparaitre `Collection.removeIf(Predicate)` qui permet la suppression d'un élément ...



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
default `Stream<E>`	`parallelStream()` Returns a possibly parallel `Stream` with this collection as its source.
default boolean	`removeIf(Predicate<? super E> filter)` Removes all of the elements of this collection that satisfy the given predicate.
default `Spliterator<E>`	`spliterator()` Creates a `Spliterator` over the elements in this collection.
default `Stream<E>`	`stream()` Returns a sequential `Stream` with this collection as its source.

Figure 5: jdk8, removeif

Supprimer un élément dans une liste devient assez compact et simple (avant, je ne pouvais pas faire sans –la parcourir– écrire le parcours):

```
wednesday.removeIf(t -> t.getName().contains("Marlène"));
```

La classe Map

La classe `java.util.Map` s'enrichit de quelques méthodes. `putIfAbsent` ne nécessite pas que l'on s'y attarde. `Map` propose un *forEach* sur ses paires clé/valeur utilisant un *biconsumer* (je « consomme » la clé et la valeur pour en faire « quelque chose ») et un *replaceAll* utilisant une *bifunction* (sur base de la clé et de la valeur, je retourne « quelque chose ») sur les paires clé/valeur permettant par exemple d'écrire:

```
Map<Integer, Person> map = ...
map.replaceAll((k, v) -> new GenderPerson(v.getName(), Gender.UNKNOW));
map.forEach((Integer k, Person v) -> {
        System.out.println(v + " (" + k + ")");
    });
```

Au traditionnel `put`, la classe ajoute une méthode `replace` dont je ne vois pas l'utilité, une méthode `compute[IfAbsent|IfPresent]` permettant de modifier une valeur voire de l'ajouter si elle n'existe pas et ajoute à la méthode `remove(Key)` déjà existante une méthode `remove(Key, Value)` qui n'ôtera la paire que si la **paire** existe. À l'inverse de l'ancienne méthode qui ôte la paire quelle que soit la valeur.

La classe Comparator

La classe `java.util.Comparator` (*functional interface*) est munie d'une méthode statique `naturalOrder` retournant un *comparator* basé sur la méthode *compareTo*. Cette classe `Comparator` ne fonctionne donc qu'avec des objets comparables (implémentant `Comparable`).

La classe existe depuis *jdk1.2* et s'utilise avec `Arrays.sort` et `Collections.sort`. Aujourd'hui une méthode *sort* est ajoutée à l'interface *List*.

```
List<Person> wednesday = ...
wednesday.sort(Comparator.naturalOrder());
wednesday.forEach(t -> System.out.println(t.getName()));

wednesday.sort(Comparator.reverseOrder());
wednesday.forEach(t -> System.out.println(t.getName()));

wednesday.sort((Person o1, Person o2)
    -> o1.name.charAt(1) - o2.name.charAt(1));
wednesday.forEach(t -> System.out.println(t.getName()));
```

La méthode compare de la classe Comparator est une bifunction et la classe Comparator une interface fonctionnelle.

Map, filter, reduce pattern

La présentation de José Paumard au BRUJUG commence par une présentation du *design pattern* « Map / (filter) / reduce ».

La première question posée est:

> Calculer la moyenne d'âge des personnes de plus de 20 ans dans cette liste.

Avec un *jdk7* et une approche itérative ce calcul nécessite; un parcours de la liste, un test au niveau de l'âge, l'ajout de l'âge à la somme des âges et, finalement, un calcul de la moyenne par division.

À l'inverse, cette même question posée à un SGBD nous permettrait d'écrire quelque chose du style ci-dessous tout en laissant au SGBD le loisir de faire le calcul comme il l'entend:

```
SELECT AVG() FROM Person WHERE age > 20
```

L'approche que l'on veut avoir et qui nous [est un peu guidée par les algorithmes *big data*][mapreduce] est de faire **correspondre** la liste aux attributs qui vont bien (par exemple les prénoms), de la **filtrer** en ne conservant que les personnes qui nous intéressent et de **réduire** la liste en un seul nombre, dans ce cas, la moyenne d'âge.

Les trois termes importants sont:

- *map* fait correspondre une liste à une autre liste du même nombre d'éléments (une liste de personnes <-> une liste **d'âges** de personne);
- *filter* fait correspondre une liste à une liste de moins d'éléments **du même type** (une liste d'âge → une liste d'âge > 20 ans);
- *reduce* agrège les éléments en un résultat unique (la moyenne)

Il est possible de modéliser cela avec JDK7 par le biais d'interfaces et de classes anonymes comme l'explique très bien José Paumard dans sa présentation des lambdas / streams / collectors.

Je devrais créer une interface me permettant de faire correspondre une personne avec son âge. Nous savons déjà que les **interfaces fonctionnelles** (*functional interfaces*) répondent à la question avec une fonction (*function*). Pour le filtre, un prédicat (*predicate*) précisera si l'on conserve l'élément dans la nouvelle liste ou pas. Tandis que pour l'agrégation, une bifonction (*bifunction*) réduira la valeur de proche en proche.
##La classe Stream {#laclassestream}

```
package java.util.stream
```

Un *stream* est un flux de données, éventuellement infini en provenance d'une source. C'est une interface générique (`Stream<E>`) ou une interface adaptée aux types primitifs (`IntStream`, ...) Un *stream* ne porte pas les données ni ne les contient, il permet de les faire transiter afin d'exprimer un traitement sur celles-ci. Le *stream* ne modifie pas la source, il permet l'expression d'un traitement.

Les opérations sur les *stream* sont de deux types:

- les opérations intermédiaires qui ne font aucun traitement mais qui le définissent (*map*, *filter*) et;
- les opérations terminales qui déclenchent le traitement (*reduce*, « *to collection* »)

L'utilisation d'un *stream* permet d'éviter la duplication de collections en mémoire lorsque le traitement que l'on veut faire nécessite l'utilisation de collections intermédiaires. L'utilisation d'un *stream* permettra au compilateur d'optimiser le traitement en fonction du traitement que l'on désire.

Un *stream* se crée sur base d'une collection, d'un tableau ou d'une des *factory* de la classe `Stream`.
Par exemple:

```
List<Person> wednesday = ...
Stream<Person> stream = wednesday.stream() ...
```

Un *stream* est `Closeable`. Il peut donc se trouver dans un *try-with resources*.

La classe Collectors

La classe `Collectors` du *package* `java.util.stream` propose moult (37) méthodes permettant la « réduction » d'un *stream* dans l'optique du *design pattern map filter reduce*. On y trouve par exemple; *counting*, *summing*, *averaging*, *joining*, *mapping*, ... Ceci permettra parfois d'éviter d'écrire une *lambda*.

```
all = wednesday.stream()
    .map((Person t) -> t.getName())
    .collect(Collectors.joining(", "));
```

pourra remplacer

```
all = wednesday.stream()
    .map((Person p) -> p.getName())
    .reduce("", (String t, String u) -> t + ", " + u);
```

Pour d'autres exemples moins triviaux, on ira voir la présentation de José Paumard.

Les exemples présentés ici et d'autres se trouvent dans ce répo github

Points non abordés

- Si la classe `java.util.concurrent.Future` est présente depuis jdk5, elle se voit ajouter la classe `CompletableFuture` permettant de prévoir et d'organiser les tâches concurrentes.
- Les améliorations sur les classes utilisées en multi-processing.
- ... et (probablement) d'autres choses encore

En espérant que ça favorise votre *upgrade* ... vous pouvez reprendre une activité normale.

Liens / crédits

- Présentation de MapReduce pattern par Ilya Katsoz
- Les supports de José Paumard suite à la [conférence du BRUJUG à Leuven][brujug-jpaumard]: GitHub, Slideshare, lambdas-streams et collections et Slideshare, 50 nouveautés
- Mes exemples chez github
- Crédit photo chez DeviantArt par Pinkilla

Pierre (Pit) Bettens
pb@namok.be

http://blog.namok.be
@pinkilla