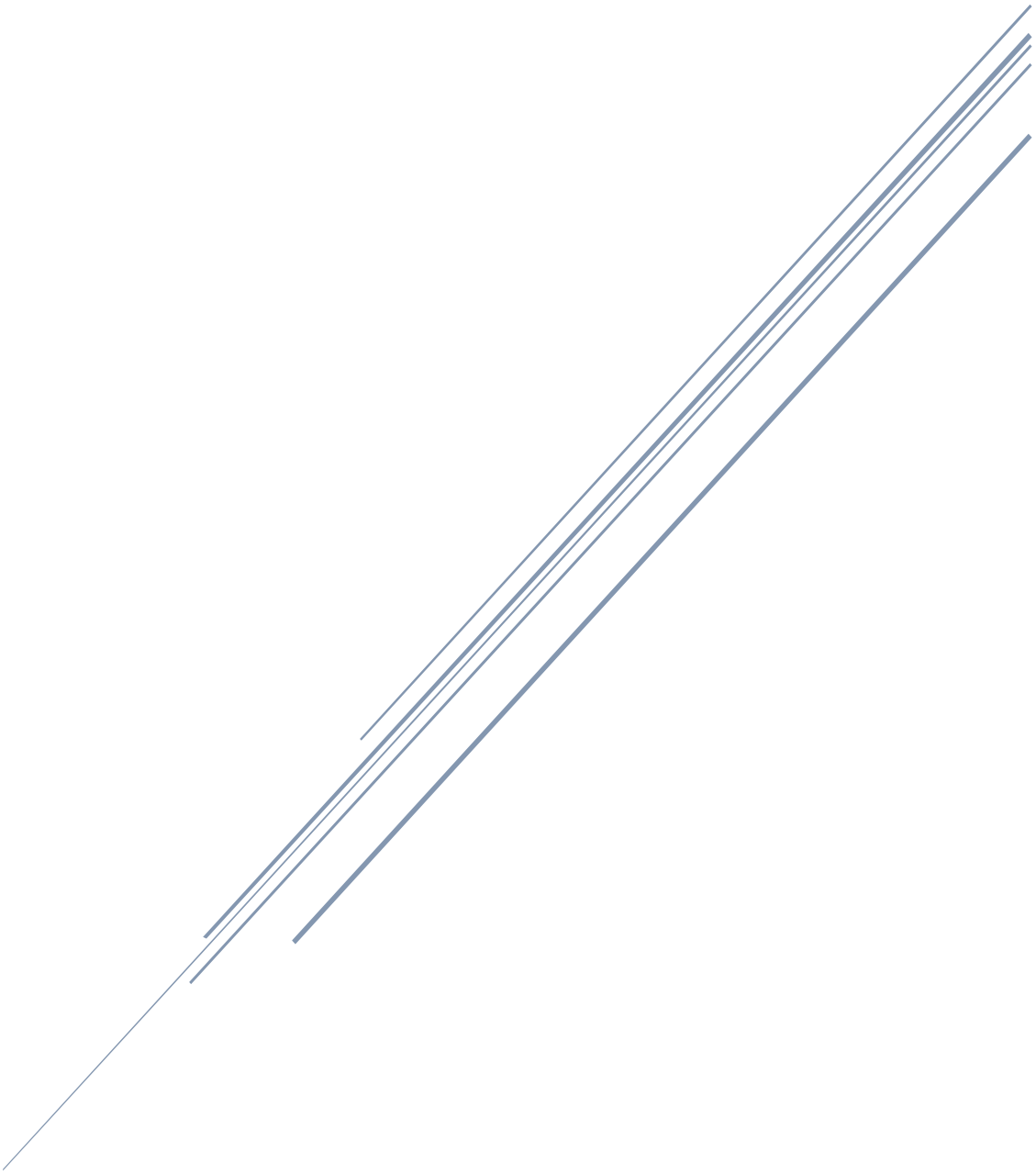
JAVA 8

LES FONDAMENTAUX DU LANGAGE



Attia Mehdi

Table des matières

II. Lambda expressions (JSR 335)

Dans les versions antérieures du Java, il n'existe que deux types de références : des références vers des valeurs primitives, et des références vers des instances d'objets. Dans d'autres langages (Groovy, Scala, Haskell...), il est également possible d'établir des références vers des closures, c'est-à-dire des blocs de code anonymes. Une référence de ce type peut alors, comme toutes les autres, être utilisée en tant que champ d'une classe ou en paramètre d'une méthode. Sous la pression des langages "alternatifs" et de la communauté Java, Oracle s'est enfin décidé à intégrer les closures dans le langage.

Dans sa forme la plus simple, une expression lambda pourrait être représentée comme une liste des paramètres séparés par des virgules, le symbole de → et le corps.

Lambdas expression est considérée comme la plus grosse nouveauté de Java 8, c'est une forme de méthode, plus compacte qu'une méthode standard, pour laquelle le nom est implicite et les paramètres peuvent être omis, tout comme leur type ou une valeur de retour. C'est une manière simplifiée d'écrire l'implémentation d'une interface qui n'a qu'une seule méthode abstraite (SAM).

Exemple 1 : Trier une liste de chaines de caractères

```
List<String> names = Arrays.asList("Lionel", "Andres", "Neymar",
        "Xavi,Busquets");
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

Au lieu de créer des objets anonymes toute la journée, Java 8 est livré avec une syntaxe beaucoup plus courte et plus flexible, les expressions lambdas :

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

Comme vous pouvez le voir le code est beaucoup plus court et plus facile à lire, et il peut être encore plus court puisque il est possible d'omettre les types de paramètres et la valeur de retour.

Remarque : Le type de retour d'une expression lambda est celui de la méthode abstraite de l'interface fonctionnelle associée à cette expression.

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Exemple 2 : Les expressions lambda dans les applications graphiques (GUI)

Pour traiter les événements dans une interface utilisateur graphique (GUI), telles que les actions de clavier et les actions de la souris, vous créez généralement des gestionnaires d'événements, ce qui implique généralement la mise en œuvre d'une interface particulière. Souvent, les interfaces de gestionnaire d'événement sont des interfaces fonctionnelles; ont tendance à avoir une seule méthode.

Voici une classe anonyme qui définit une action de la souris en Java FX :

```
btn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

L'invocation de méthode ***"btn.setOnAction"*** spécifie ce qui se passe lorsque vous sélectionnez le bouton représenté par l'objet de ***"btn"***. Cette méthode nécessite un objet de type ***"EventHandler<ActionEvent>".*** L'interface ***"EventHandler<ActionEvent>"*** contient une seule méthode abstraite, ***"void handle(T event)"*** (Functional interface), donc il est possible d'utiliser l'expression lambda mis en évidence ci-dessous pour la remplacer.

```
btn.setOnAction(event -> System.out.println("Hello World!"));
```

I.1 Lambda Target Type

Dans d'autres langages, le type de l'expression lambda serait une fonction; mais dans Java, les expressions lambda sont représentées comme des objets, donc ils doivent avoir un type d'objet, particulier connu sous le nom d'une interface fonctionnelle. Ce qu'on appelle le type de cible "target type".

Puisque une interface fonctionnelle ne peut avoir qu'une seule méthode abstraite, les types des paramètres de l'expression Lambda doivent correspondre aux paramètres de cette méthode, et le type du corps de lambda doit correspondre au type de retour de la méthode. En outre, toutes les exceptions émises dans le corps lambda doivent être autorisés par la clause "***throws***" de cette méthode dans l'interface fonctionnelle.

Considérons les deux interfaces fonctionnelles suivantes ***(java.lang.Runnable*** et ***java.util.concurrent.Callable<V>):***

```
public interface Runnable {
    void run();
}
public interface Callable<V> {
    V call();
}
```

La méthode "***Runnable.run***" n'a pas une valeur de retour contrairement à la méthode *"**CallableV>.call**"*

Supposons qu'une méthode nomée "***invoke***" sera surchargée comme suit :

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Quelle méthode sera appelée par l'instruction suivante :

```
String s = invoke(() -> "done");
```

La méthode ***"invoke (Callable <T>)"*** sera invoquée parce que cette méthode renvoie une valeur. Dans ce cas, le type de l'expression lambda ***" () -> done"*** est ***Callable <T>.***

I.2 Lambda Scopes

Accéder à des variables externes à partir des expressions lambda est très similaire aux objets anonymes. Il est possible d'accéder aux variables locaux finals (finals) ainsi que les variables d'instance (instance fields) et les variables statiques.

- **Accéder à des variables locales**

Nous pouvons lire des variables locales finales à partir de la portée externe des expressions lambda.

```
public static interface Converter<F, T> {
    T convert(F from);
}

public static void main(String[] args) {
    final int num = 1;
    Converter<Integer, String> stringConverter =  (from) -> String.valueOf(from + num);
    stringConverter.convert(2);     // 3
}
```

Mais contrairement aux objets anonymes la variable ***"num"*** peut être déclarée non final. Ce code est également valable:

```
int num = 1;
Converter<Integer, String> stringConverter = (from) -> String.valueOf(from + num);
stringConverter.convert(2);     // 3
```

Cependant la variable ***"num"*** doit être implicitement finale. Le code suivant ne compile pas:

```
int num = 1;
Converter<Integer, String> stringConverter =  (from) -> String.valueOf(from + num);
stringConverter.convert(2);
num = 3;
```

- **Accéder aux fields et aux variables statiques**

Contrairement aux variables locales nous avons l'accès en lecture et écriture aux variables d'instance et aux variables statiques de l'intérieur des expressions lambda.

I.3 Ce que Lambda ne peut pas faire

Il y a quelques caractéristiques que lambdas ne fournissent pas. Elles ont été considérées pour Java 8, mais n'ont pas été incluses, pour plus de simplicité et en raison de contraintes de temps.

- **Accès aux variables locales non finales**

Comme il est mentionné dans la section (I.2) si une nouvelle valeur est affectée à une variable, elle ne peut pas être utilisée dans une expression lambda. Le mot-clé "final" n'est pas nécessaire, mais la variable doit être "effectivement finale» (voir plus haut). Ce code ne compile pas:

```
int count = 0;
List<String> strings = Arrays.asList("a", "b", "c");
strings.forEach(s -> {
    count++; // erreur: Impossible de modifier la valeur du count
});
```

- **La transparence d'exception**

Si une exception contrôlée peut être lancée à l'intérieur d'une expression lambda, l'interface fonctionnelle doit également déclarer que cette exception contrôlée peut être lancée aussi. L'exception n'est pas propagée à la méthode contenant l'expression lambda. Ce code ne compile pas:

```
void appendAll(Iterable<String> values, Appendable out)
        throws IOException { // n'aide pas à interpréter l'erreur.
    values.forEach(s -> {
        out.append(s); // Erreur: Impossible de lancer IOException ici
                       // Consumer.accept (T) ne le permet pas
    });
}
```

- **Flux de contrôle (break, return)**

Il n y a pas un moyen de sortir d'une boucler et retourner une valeur comme résultat de la méthode qui contient l'expression lambda, par exemple :

```
final String secret = "foo";
boolean containsSecret(Iterable<String> values) {
    values.forEach(s -> {
        if (secret.equals(s)) {
            ??? /* Tu veux mettre fin à la boucle et retourner true,
                   mais ce n'est pas possible */
        }
    });
}
```

III. Interfaces fonctionnelles

Une interface fonctionnelle est une interface disposant une unique méthode abstraite. Le but est de définir la signature d'une méthode qui pourra être utilisée pour passer en paramètre :

- Une référence vers une méthode statique
- Une référence vers une méthode d'instance
- Une référence vers un constructeur
- Une expression lambda.

Même si l'annotation ***@FunctionalInterface*** n'est pas obligatoire, elle a un rôle important permettant au compilateur de forcer l'interface afin qu'elle ne contienne qu'une seule méthode abstraite. Si nous essayons de définir plus d'une seule méthode abstraite dans une interface annotée avec ***@FunctionalInterface***, le compilateur affichera une erreur.

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
```

Une interface ne peut pas fournir une implémentation par défaut de l'une des méthodes de la classe Object. En particulier, cela signifie qu'il n'est pas possible de fournir une implémentation pour les méthodes **"*equals*"**, **"*hashCode*"**, ou **"*toString*"** à partir d'une interface.

II.1 Méthodes statiques et les méthodes par défaut.

Parmi les nouveautés apportées par Java 8, on en trouve deux qui concernent les interfaces: les méthodes statiques et les méthodes par défaut. Les méthodes statiques définies sur les interfaces fonctionnent exactement de la même façon que celles portées par les classes, il n'y a donc pas grand-chose à en dire. En revanche, les méthodes par défaut risquent de modifier assez profondément notre façon de concevoir nos API.

En Java 7 et antérieur, une méthode déclarée dans une interface ne fournit pas d'implémentation. Ce n'est qu'une signature, un contrat auquel chaque classe dérivée doit se conformer en fournissant une implémentation propre. Mais il arrive que plusieurs classes similaires souhaitent partager une même implémentation de l'interface. Dans ce cas, deux stratégies sont possibles :

- Factoriser le code commun dans une classe abstraite, mais il n'est pas toujours possible de modifier la hiérarchie des classes
- Extraire le code commun dans une classe utilitaire, sous forme de méthode statique (ex: ***Collections.sort()***).

On conviendra qu'aucune des deux n'est réellement satisfaisante. Heureusement, Java 8 nous offre maintenant une troisième possibilité en proposant une solution plus propre permettant aux méthodes déclarées dans les interfaces d'avoir une implémentation, il s'agit bien des méthodes par défaut. Une méthode par défaut est une méthode d'instance définie dans une interface et commence par le mot-clé "default", elle contient également un corps de code. Chaque classe qui implémente l'interface hérite ses méthodes par défaut et peut les redéfinir.

Prenons un coup d'œil sur exemple ci-dessous.

```java
interface Formule {
    double calculer(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

L'interface Formule définit la méthode abstraite "***calculer***" la méthode par défaut "***sqrt***". Classes concrètes n'ont qu'à implémenté la méthode abstraite calculer.

```java
Formule formule = new Formule() {
    @Override
    public double calculer(int a) {
        return sqrt(a * 100);
    }
};

formule.calculer(100); // 100.0
formule.sqrt(16); // 4.0
```

L'interface formule est implémentée en tant qu'objet anonyme. Le code est assez verbeux: 6 lignes de code pour une simple calcule de ***"sqrt (a * 100)"***. Dans les prochaines sections (II.3, III) nous présentons un moyen plus agréable d'implémentation d'interfaces qui possèdent une seule méthode abstraite (Single Abstract Method interface) en Java 8.

II.2 Référence de méthodes et de constructeurs

Une référence de méthode ou de constructeur est utilisée pour définir une méthode ou un constructeur en tant qu'implémentation de la méthode abstraite d'une interface fonctionnelle, à l'aide du nouvel opérateur « **::** ».

Tous constructeurs et méthodes peuvent être utilisés comme référence, à condition qu'il n'y ait aucune ambigüité. Lorsque plusieurs méthodes ont le même nom, ou lorsqu'une classe a plus d'un constructeur, la méthode ou le constructeur approprié est sélectionné en fonction de la méthode abstraite de l'interface fonctionnelle à laquelle fait référence l'expression.

Exemples :

```java
public class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}

Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");

System.out.println(converted);    // "J"
```

Voyons comment le mot-clé « **::** » fonctionne pour les constructeurs. Nous définissons d'abord une classe avec différents constructeurs:

```java
class Person {
    String firstName;
    String lastName;

    Person() {
    }

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Ensuite, nous spécifions une interface "***PersonneFactory***" pour créer de nouvelles personnes:

```java
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

Nous créons une référence au constructeur personne via ***"Person::new"***. Le compilateur Java choisit automatiquement le bon constructeur en faisant correspondre la signature de ***"PersonFactory.create"***.

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Lionel", "Messi");
```

Le tableau suivant présente quatre types de références de méthode.

Type de référence de méthode	Syntaxe	Exemple	Equivalence en Lambda
Référence d'une méthode statique	className**::**staticMethodName	String::valueOf	x -> String.valueOf(x)
Référence d'une méthode d'instance	objectName::instanceMethodName	s::toString	() -> s.toString()
Référence d'une méthode d'instance arbitraire	className::instanceMethodName	Object::toString	x -> x.toString()
Référence du constructeur	className::new	ArrayList::new LinkedList<Employee>::new	() -> new ArrayList<> () -> new LinkedList<Employee>()

Les Références des méthodes est un moyen facile pour accéder aux méthodes existantes et les utiliser d'une manière fonctionnelle.

II.3 Les interfaces fonctionnelles les plus utilisées

Le JDK 8 apporte par défaut plusieurs interfaces fonctionnelles, D'anciennes interfaces de l'API ont été migrées telles que **"*Comparable*"** et **"*Runnable*"**.

Nous citons ci-dessous les interfaces les plus utilisées :

- ***Function***

Une interface "***Function***" prend un argument et retourne un résultat dont la méthode fonctionnelle est ***"apply(Object)".***

```
Function<Integer, String> toString = n -> String.valueOf(n);
```

@FunctionalInterface
public interface Function<T,R>

T - le type d'entrée de la fonction
R - le type du résultat de la fonction

L'interface "***Function***" contient des méthodes par défaut (***compose***, ***andthen***) qui peuvent être utilisées pour chaîner plusieurs fonctions ensemble.

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);
backToString.apply("123"); // "123"
```

- ***Supplier***

L'interface **"*Supplier*"** produise un résultat d'un type donnée. Contrairement à l'interface **"*Function*"**, **"*Supplier*"** n'accepte pas les arguments. La méthode abstraite de cette interface est ***"get()".***

@FunctionalInterface
public interface Supplier<T>

T - Le type du résultat fournit par ce Supplier

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

- ***Consumer***

L'interface **"*Consumer*"** représente une opération qui accepte un seul argument et ne renvoie aucun résultat. La méthode abstraite de cette interface est **"*accept(Object)*"**.

```
Consumer<Person> personConsumer= (p) -> System.out.println("Hello," + p.firstName);
personConsumer.accept(new Person("Lionel", "Messi"));
```

@FunctionalInterface
public interface Consumer<T>

T - le type d'entrée de la fonction

- ***Predicate***

L'interface **"*Predicate*"** représente une opération qui accepte un seul argument et renvoie un résultat de type **"*Boolean*"**. La méthode abstraite de cette interface est **"*test(Object)*"**.

@FunctionalInterface
public interface Predicate<T>

T - le type d'entrée de la fonction

```
Predicate<String> predicate = (s) -> s.length() > 0;
predicate.test("goal");              // true
```

L'interface **"*Predicate*"** contient quelques méthodes non abstraites (par défaut) telles que :

"***and (Predicate<? super T> other)***", "***isEqual (Object targetRef)***", "***negate()***" ,"***or (Predicate<? super T> other)***".

```
Predicate<String> predicate = (s) -> s.length() > 0;
predicate.negate().test("Goal");          // false
```

- ***Comparator***

L'interface **"*Comparator*"** est bien connue dans les anciennes versions du Java. C'est une fonction de comparaison, qui impose un ordre total sur certains objets de collection via sa méthode abstraite ***"compare(T o1, T o2)"*.**

@FunctionalInterface
public interface Comparator<T>

T - le type d'objets qui peuvent être comparés par ce comparateur.

Java 8 ajoute d'autres méthodes par défaut à l'interface **"*Comparator*"** telles que "***reversed*()**" et "***thenComparing(Comparator<? super T> other)***".

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName
        .compareTo(p2.firstName);
Person p1 = new Person("Lionel", "messi");
Person p2 = new Person("Andres", "iniesta");
comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2); // < 0
```

- ***UnaryOperator***

Représente une opération sur un opérande unique qui produit un résultat du même type que son opérande. C'est une spécialisation de l'interface **"*Function*"** dans le cas où l'opérande et le résultat sont du même type. L'interface abstraite de cette interface est ***"apply(Object)".***

@FunctionalInterface
public interface UnaryOperator<T>
extends Function<T,T>

T - le type de l'opérande et du résultat de l'opération

```
UnaryOperator<Integer> add = n -> n + 1;
add.apply(5); // 6
```

```
Function<Integer, Integer > add = n -> n + 1;
add.apply(5); // 6
```

- ***BiFunction***

Représente une fonction qui accepte deux arguments et produit un résultat. C'est la spécialisation de deux arités de l'interface **"*Function*"**. La méthode abstraite de cette interface est ***"apply(T t, U u)".***

@FunctionalInterface
public interface BiFunction<T,U,R>

- T - le type du premier argument de la fonction
- U - le type du deuxième argument de la fonction
- R - le type du résultat de la fonction

```
BiFunction<String,String,Integer> sum = (x, y)
        -> Integer.valueOf(x) + Integer.valueOf(y);
System.out.println(sum.apply("5", "6")); //11
```

- ***BinaryOperator***

Représente une opération à deux opérandes de même type, qui produise un résultat du même type que les opérandes. C'est une spécialisation de l'interface **"*BiFunction*"** dans le cas où les opérandes et le résultat sont tous du même type.

@FunctionalInterface
public interface BinaryOperator<T>
extends BiFunction<T, T, T>

T - le type des opérandes et du résultat de l'opération

```
BinaryOperator<Integer> sum = (x, y) -> x + y;
System.out.println(sum.apply(5, 6)); //11
```

On a cité les interfaces fonctionnelles les plus utilisées en java 8, mais il existe plusieurs d'autres tels que ***"BiPredicate<T, U>", "BiConsumer<T, U>…"***

IV. Streams

Java 8 arrive avec une toute nouvelle API Stream qui utilise les Lambda. Cette nouvelle API offre la possibilité de simplifier l'écriture, d'améliorer la performance ainsi d'augmenter la lisibilité d'un certain nombre de code.

Un "***Stream<T>***" décrit une séquence d'éléments (de type T) supportant des opérations agrégées en séquence ou en parallèle.

En voici un exemple :

```
Collection<Player> players = playerService.getPlayers();
int salaire = players.stream()
        .filter(p -> p.getName().equals("lionel"))
        .mapToInt(Player::getSalaire).sum();
```

On récupère un "***Stream<Player>***" à partir d'une collection que l'on filtre en ne prenant que les joueurs dont le nom est "***lionel***". On crée ensuite un "***IntStream***" (stream de int) des salaires de ces joueurs filtrés que l'on additionne. En bref, on a calculé le salaire total des joueurs dont le nom est "***lionel***".

Il est important de ne pas voir les "***streams***" comme une alternative aux collections mais plutôt comme un moyen de traitement associé :

- Un "***stream***" n'est pas un moyen de stockage, il ne fait que transmettre des données provenant d'une source (collection, tableau, générateur de données, canal d'E/S, …) au travers d'une file d'opérations.
- Un "***stream***" est fonctionnel par nature, une opération sur un "***stream***" produit un résultat mais ne modifie jamais sa source. Un filtrage, par exemple, crée un nouveau "***stream***" avec les éléments filtrés, il ne supprime rien de la source.
- Un "***stream***" a un mode d'exécution paresseuse. Les opérations d'un "***stream***" sont de deux types : intermédiaires (elles renvoient un "***stream***" comme le filtrage par exemple) ou terminales (comme la somme vue plus haut). Les opérations intermédiaires sont toujours implémentées de manière paresseuse, afin de profiter au maximum d'une éventuelle optimisation.
- Un "***stream***" est un consommateur, ses éléments ne peuvent être visités qu'une seule fois pendant sa durée de vie, un nouveau "***stream***" doit donc être créé pour parcourir à nouveau la source.

III.1 Obtention d'un stream

Il existe plusieurs façons de créer un "***stream***". La plus simple consiste à appeler la méthode "***stream()***" ou "***parallelStream()***" sur une collection, de plus un certain nombre de méthodes ont été ajoutées aux classes déjà existantes.

Un "***stream***" s'obtient de la manière suivante :

- Depuis une Collection via la méthode "***stream()***" ou "***parallelStream()***"
- Depuis un tableau via "***Arrays.stream(Object[])*****.**
- Depuis des méthodes statiques de l'interface "***Stream<T>*** comme "***Stream.of(Object[])*****",** "***IntStream.range(int,int)***" ou "***Stream.iterate(Object, UnaryOperator)*** **".**
- Lors de la lecture d'un fichier via "***BufferedReader.lines()*****".**

III.2 Opérations sur les stream

L'intérêt principal d'un "***stream***" réside dans la possibilité d'effectuer plusieurs opérations plus ou moins complexes en les enchaînant en une seule instruction. Cet enchaînement, appelé pipeline, est composé de plusieurs opérations intermédiaires et d'une seule opération terminale.

- **Opérations intermédiaires**
- **Du même type :**

Chaque opération intermédiaire retourne un nouveau "***stream***" (permettant ainsi l'enchaînement des opérations).

Soit "***stringCollection***" une liste des String définie comme suit :

```
List<String> stringCollection = Arrays.asList("ddd2", "aaa2", "bbb1",
        "aaa1", "bbb3", "ccc", "bbb2", "ddd1");
```

- **Stream<T> filter(Predicate<T>)** : retourne un Stream<T> dont les éléments correspondent au test du prédicat.

```
stringCollection.stream().filter((s) -> s.startsWith("a"))
        .forEach(System.out::println);
// "aaa2", "aaa1"
```

- **Stream<T> sorted()** : retourne un Stream<T> trié selon l'ordre naturel des éléments.

  Gardez en tête que "***sorted***" créée seulement une vue de Stream triée sans manipuler l'ordre de la collection "***stringCollection***".

```
stringCollection.stream().sorted().forEach(System.out::println);
// aaa1, aaa2, bbb1, bbb2,bbb3,ccc,ddd1,ddd2

System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

NB : Si T n'implémente pas l'interface Comparable, une **"*ClassCastException*"** est levée. Pour les "***streams***" ordonnés, le tri est stable, dans le cas contraire, rien n'est garanti.

- **Stream<T> sorted(Comparator<T> comparator)** : identique à la méthode précédente mais en précisant le comparateur.
- **Stream<T> distinct()** : retourne un Stream<T> dont les éléments sont tous distincts.

```
Stream.of("aaa", "bbb", "aaa", "ccc").distinct()
        .forEach(System.out::println);
//"aaa" , "bbb" , "ccc"
```

  NB : T doit obligatoirement implémenter la méthode "***hashCode()***"**.**

- **Stream<T> limit(long maxSize)** : retourne un Stream<T> limité à la taille donnée.

```
stringCollection.stream().limit(4).forEach(System.out::println);
// ddd2, aaa2, bbb1, aaa1
```

- **Stream<T> peek(Consumer<T> action)** : retourne un Stream<T> avec les mêmes éléments que la source mais dont l'action spécifiée est exécutée lors de leur consommation.

  Cette méthode est très pratique pour débugger pendant la visite d'un pipeline de plusieurs opérations.

```
.peek(System.out::println)
.peek(it -> System.out.printf("it is %s%n", a))
```

- **Stream<T> skip(long n)** : retourne un Stream<T> sans les n premiers éléments.

```
stringCollection.stream().skip(3).forEach(System.out::println);
// aaa1, bbb3, ccc, bbb2, ddd1
```

NB : Si le "***stream***" contient moins de n éléments, un "***stream***" vide est renvoyé.

- **De type différent**

Des opérations intermédiaires plus complexes peuvent « changer » le type du Stream :

- **<R> Stream<R> map(Function<T, R> mapper)** : retourne un Stream<R> dont chaque élément est une obtenu à partir d'un élément de la source (via le mapper qui est une fonction de transformation d'un objet de type T en objet de type R).

```
stringCollection.stream().map(s -> s.length())
        .forEach(System.out::println);
// 4,4,4,4,4,3,4,4
```

- **R> Stream<R> flatMap(Function<T, Stream<R>> mapper)** : retourne un **Stream<R>** contenant tous les éléments résultants de l'application du mapper appliqué à chaque élément de la source. En bref, on applique une transformation 1-n à chaque élément puis on concatène le tout.

Nous avons déjà appris comment transformer les objets d'un "***stream***" à un autre type d'objets en utilisant l'opération "***map***". "***map***" est un peu limité parce que chaque objet transformé exactement à un autre objet. Mais que faire si nous voulons transformer un objet en plusieurs autres? C'est là que "***flatMap***" vient à la rescousse.

"***flatMap***" transforme chaque élément du "***stream***" à un "***stream***" d'autres objets. Donc, chaque objet sera transformé en zéro, un ou plusieurs autres objets sauvegardés par des "***streams***". Le

contenu de ces "***streams***" sera par la suite placé dans le "***stream***" renvoyé par l'opération de "***flatMap***".

Avant de voir en action "***flatMap***" nous avons besoin d'une hiérarchie de type approprié:

```java
public class Bar {
    String name;

    Bar(String name) {
        this.name = name;
    }
}
```

```java
public class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();

    Foo(String name) {
        this.name = name;
    }
}
```

Ensuite, nous utilisons nos connaissances sur les "***stream***" d'instancier un couple d'objets:

```java
List<Foo> foos = new ArrayList<>();
// create foos
IntStream.range(1, 4).forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f -> IntStream.range(1, 4).forEach(
        i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

Maintenant, nous avons une liste de trois "***foos***" constituées chacune de trois "***bars***".

"***FlatMap***" accepte une fonction qui doit retourner un "***stream***" d'objets. Donc, afin de résoudre les objets bar de chaque "***foo***", nous passons la fonction appropriée:

```java
foos.stream().flatMap(f -> f.bars.stream())
        .forEach(b -> System.out.println(b.name));
```

```
// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

Comme vous pouvez le voir, nous avons transformé un "***stream***" de trois objets "***foo***" à un "***stream***" de neuf objets "***bar***". Enfin, l'exemple ci-dessus peut être simplifié en un seul pipeline d'opérations de "***stream***" :

```
IntStream
        .range(1, 4)
        .mapToObj(i -> new Foo("Foo" + i))
        .peek(f -> IntStream.range(1, 4)
                .mapToObj(i -> new Bar("Bar" + i + " <- " + f.name))
                .forEach(f.bars::add)).flatMap(f -> f.bars.stream())
        .forEach(b -> System.out.println(b.name));
```

NB : Les mêmes opérations complexes existent sur et vers les "***streams***" de type (***IntStream***, ***LongStream*** et ***DoubleStream***) telles que ***mapToInt***, ***flatMapToInt***, etc.

- **Opérations terminales**

L'opération terminale d'un pipeline est l'opération finale, celle qui déclenche l'exécution du pipeline d'opérations (et donc consomme le flux d'entrée), produit (éventuellement) un résultat puis ferme le "***stream***".

- **Booléennes**

Les opérations booléennes "***allMatch***", "***anyMatch***" et "***noneMatch***", qui prennent toutes les trois un "***Predicate<T>***" en paramètre, sont vraies lorsque, respectivement, tous les éléments, au moins un ou aucun satisfont le prédicat donné.

```java
boolean anyStartsWithA = stringCollection.stream().anyMatch(
        (s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

boolean allStartsWithA = stringCollection.stream().allMatch(
        (s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

boolean noneStartsWithZ = stringCollection.stream().noneMatch(
        (s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true
```

- **Numériques**

Seules 2 opérations numériques existent :

- **long count()** : retourne le nombre d'éléments du "***stream***".
- **type sum()** : retourne la somme des éléments du "***stream***" (type étant ***int***, double ou long), cette méthode n'existe que sur les "***streams***" typés.

```java
long startsWithB =
        stringCollection
            .stream()
            .filter((s) -> s.startsWith("b"))
            .count();
    System.out.println(startsWithB);    // 3
```

- **Simples**

Certaines opérations renvoient un "***Optional<T>***" c'est-à-dire un objet qui peut être soit ***(null)***, soit un élément du "***stream***":

"***Optional<T>***" est un conteneur d'objet permettant de gérer différemment la nullité d'une valeur. La classe "***Optional***" pourrait être manipulée via plusieurs méthodes. La méthode

"***isPresent***" nous renseigne sur l'existence d'une valeur dont l'accès est donné par "***get***". De plus, il est possible d''ajouter une valeur par défaut ou de lever une exception via les méthodes "***orElse***" ou "***orElseThrow***").

- **Optional<T> findAny()** : retourne un élément (n'importe lequel) du "***stream***".
- **Optional<T> findFirst()** : retourne le premier élément du "***stream***".
- **Optional<T> max (Comparator<T>comparator) et Optional<T> min (Comparator<T>comparator)** retournent respectivement le maximum et le minimum des éléments du "***stream***" en accord avec le comparateur donné.

- **Complexes**

La plupart des exemples de code de cette section utilisent la liste suivante des personnes pour la démonstration.

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

```
List<Person> persons =
        Arrays.asList(
            new Person("Neymar", 23),
            new Person("Messi", 27),
            new Person("Pique", 27),
            new Person("Mascherano",
```

Collect

"***Collect***" est une opération terminale extrêmement utile pour transformer les éléments d'un "***stream***" à un autre type de résultat, par exemple une "***List***", "***Set***" ou "***Map***". "***Collect***" accepte un "***collector***" qui se compose de quatre opérations différentes: un "***supplier***", un "***accumulator***", un "***combiner***" et un "***finisher***". Cela semble super-compliqué au premier abord, mais l'atout de Java 8 c'est qu'il prend en charge diverses collectors intégrés via la classe Collectors. Donc, pour les opérations les plus courantes vous n'avez pas à mettre en œuvre un "***Collector***" par vous-même.

Commençons par un cas d'utilisation très courant:

```
List<Person> filtered =
        persons
            .stream()
            .filter(p -> p.name.startsWith("M"))
            .collect(Collectors.toList());

    System.out.println(filtered);    // [Messi, Mascherano]
```

Comme vous pouvez le voir, c'est très simple à construire une liste à partir des éléments d'un "***stream***". Si vous avez besoin d'un Set au lieu de la liste - il suffit d'utiliser "***Collectors.toSet()***".

L'exemple suivant regroupe toutes les personnes selon l'âge:

```
Map<Integer, List<Person>> personsByAge = persons.stream().collect(
        Collectors.groupingBy(p -> p.age));

personsByAge.forEach((age, p) -> System.out.format("age %s: %s\n", age,
        p));

// age 23: [Neymar]
// age 27: [Messi, Pique]
// age 30: [Mascherano
```

"***Collectors***" sont extrêmement polyvalent. Vous pouvez également créer des agrégations sur les éléments du "***stream***", par exemple déterminer l'âge moyen de toutes les personnes:

```
Double averageAge = persons
        .stream()
        .collect(Collectors.averagingInt(p -> p.age));

    System.out.println(averageAge);     // 26.75
```

Si vous êtes intéressés par des statistiques plus complètes, vous pouvez utiliser les méthodes "***summarizing***" de la classe **collectors** tels que "***summarizingInt***",

```java
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(
                Collectors.joining(" and ", "In Barcelone ",
                        " are of legal age."));
System.out.println(phrase);
// In Barcelone Neymar and Messi and Pique and Mascherano are of legal age.
```

"***summarizingLong***" et "***summarizingDouble***". Donc, nous pouvons tout simplement déterminer le min, max, somme, nombre et moyennes des âges des personnes.

```java
IntSummaryStatistics ageSummary = persons.stream().collect(
        Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=107, min=23, average=26,750000, max=30}
```

L'exemple suivant ajoute les noms des personnes à une chaine de caractère.

La méthode "***joining***" accepte un séparateur et optionnellement un préfixe et un suffixe. Afin de transformer les éléments du "***stream***" en une "***Map***", nous devons préciser comment les clés et les valeurs devraient être mappées. Gardez à l'esprit que les clés mappées doivent être uniques, sinon une exception de type "***IllegalStateException***" est levée. Vous pouvez éventuellement passer une fonction de fusion comme un paramètre supplémentaire pour contourner l'exception:

```java
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
            p -> p.age,
            p -> p.name,
            (name1, name2) -> name1 + ";" + name2));

    System.out.println(map);
    // {23=Neymar, 27=Messi;Pique, 30=Mascherano}
```

Maintenant que nous connaissons les principales méthodes de la classe Collectors, nous allons essayer de construire notre propre "***Collector***". Nous voulons transformer toutes les personnes du "***stream***" en une seule chaîne constituée de tous les noms en lettres majuscules séparées par le caractère "|". Pour y arriver, nous créons un nouveau "***collector***" via "***Collector.of()***". Nous devons passer les quatre paramètres d'un "***collector***": un "***supplier***", un "***accumulator***", un "***combiner***" and un "***finisher***".

Puisque les String en Java sont immuables, nous avons besoin d'une classe comme "***StringJoiner***" pour les construire avec un "***collector***". Le supplier construit initialement un "***StringJoiner***" avec le délimiteur approprié. "***L'accumulator***" est utilisé pour ajouter chaque nom en majuscule à la "***StringJoiner***". Le "**combiner**" sait comment fusionner deux "***StringJoiner***" en une seule. Dans la dernière étape le "***finisher***" construit la chaîne désirée à partir de la "***StringJoiner***".

- **Reduce**

L'opération de réduction combine tous les éléments du "***stream***" en un seul résultat. Java 8 prend en charge trois types différents pour réduire les méthodes. Le premier réduit un "***stream***"

```
Collector<Person, StringJoiner, String> personNameCollector = Collector
        .of(() -> new StringJoiner(" | "), // supplier
                (j, p) -> j.add(p.name.toUpperCase()), // accumulator
                (j1, j2) -> j1.merge(j2), // combiner
                StringJoiner::toString); // finisher

String names = persons.stream().collect(personNameCollector);

System.out.println(names); // NEYMAR | MESSI | PIQUE | MASCHERANO
```

d'éléments a exactement un élément du "***stream***". Voyons comment nous pouvons utiliser cette méthode pour déterminer la personne la plus âgée:

```
persons
.stream()
.reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
.ifPresent(System.out::println);    // Mascherano
```

Dans l'exemple ci-dessus, la méthode "***reduce***" accepte un "***BinaryOperator***" pour comparer les âges des personnes afin de retourner la personne avec l'âge maximum. La deuxième méthode de réduire accepte à la fois une valeur d'identité "***identity***" et un accumulateur "***BinaryOperator***". Cette méthode peut être utilisée pour construire une nouvelle personne avec les noms et les âges d'agrégées toutes les autres personnes dans le "***stream***":

```
Person result =
        persons
            .stream()
            .reduce(new Person("", 0), (p1, p2) -> {
                p1.age += p2.age;
                p1.name += p2.name;
                return p1;
            });
    System.out.format("name=%s; age=%s", result.name, result.age);
    //name=NeymarMessiPiqueMascherano; age=107
```

La troisième méthode de réduire accepte trois paramètres: une valeur d'identité "***identity***", un accumulateur "***BiFunction***" et une fonction de combinaison de type "***BinaryOperator***". Comme le type de valeurs d'identité n'est pas limité au type de personne, nous pouvons utiliser cette réduction pour déterminer la somme des âges de toutes les personnes:

```
Integer ageSum = persons
        .stream()
        .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);
    System.out.println(ageSum);  // 107
```

Comme vous pouvez voir le résultat est 107, mais qu'est ce qui se passe exactement en background? Etendons le code ci-dessus par un affichage de débogage:

```java
Integer ageSum = persons.stream().reduce(0, (sum, p) -> {
    System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
    return sum += p.age;
}, (sum1, sum2) -> {
    System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
    return sum1 + sum2;
});
/*
 * accumulator: sum=0; person=Neymar
 * accumulator: sum=23; person=Messi
 * accumulator: sum=50; person=Pique
 * accumulator: sum=77; person=Mascherano
 */
```

Comme vous pouvez voir la fonction de l'accumulateur "***accumulator***" fait tout le travail. Elle a d'abord être appelée avec la valeur d'identité initiale 0 et la première personne Max. Dans les trois prochaines étapes la somme augmente avec l'âge de la dernière personne jusqu'à atteindre le total de 76. La combinaison "combiner" ne sera jamais appelé? L'exécution du même "***stream***" en parallèle va lever le secret:

```java
Integer ageSum = persons.parallelStream().reduce(0, (sum, p) -> {
    System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
    return sum += p.age;
}, (sum1, sum2) -> {
    System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
    return sum1 + sum2;
});

/*
 * accumulator: sum=0; person=Mascherano
 * accumulator: sum=0; person=Neymar
 * accumulator: sum=0; person=Pique
 * accumulator: sum=0;person=Messi
 * combiner: sum1=27; sum2=30
 * combiner: sum1=23; sum2=27
 * combiner: sum1=50; sum2=57
 */
```

L'exécution de ce "***stream***" en parallèles implique un comportement d'exécution complétement différent. Maintenant, la combinaison est en fait appelée. Lorsque

l'accumulateur est appelé en parallèle, le combinateur est nécessaire pour additionner les valeurs accumulées séparés.

- **Order**

Maintenant que nous avons appris comment créer et travailler avec différents types de "***stream***", nous allons plonger plus profondément dans la façon avec laquelle les opérations du "***stream***" sont traités sous le capot.

Une caractéristique importante des opérations intermédiaires est la paresse ("***laziness***"). Regardez cet exemple où une opération terminale est manquante:

```
Stream.of("d2", "a2", "b1", "b3", "c")
.filter(s -> {
    System.out.println("filter: " + s);
    return true;
});
```

On exécutant ce bout de code, rien ne sera affiché à la console. C'est parce que les opérations intermédiaires ne seront exécutés que lorsque une opération terminale est présente.

Étendons l'exemple ci-dessus par l'opération terminale "***forEach***":

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
    System.out.println("filter: " + s);
    return true;
}).forEach(s -> System.out.println("forEach: " + s));
```

```
/*
 * filter: d2
 * forEach: d2
 * filter: a2
 * forEach: a2
 * filter: b1
 * forEach: b1
 * filter: b3
 * forEach: b3
 * filter: c
 * forEach: c
 */
```

L'ordre du résultat peut être surprenant. Une approche naïve consiste à exécuter les opérations horizontalement, l'une après l'autre sur tous les éléments du "***stream***". Mais au lieu que chaque élément se déplace le long de la chaîne verticalement, le premier élément String "d2" passe de la méthode ***"filtre"*** à la méthode "***forEach***", et ce n'est qu'à ce moment-là que le deuxième élément "A2" peut être traité. Ce comportement peut réduire le nombre réel des opérations effectuées sur chaque élément, comme nous le voyons dans l'exemple suivant:

```
Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
}).anyMatch(s -> {
    System.out.println("anyMatch: " + s);
    return s.startsWith("A");
});
```

```
/*
 * map: d2
 * anyMatch: D2
 * map: a2
 * anyMatch: A2
 */
```

L'opération "***AnyMatch***" renvoie vrai dès que le prédicat s'applique à l'élément d'entrée donné. Cela est vrai pour le second élément passé "A2". En raison de l'exécution verticale de la chaîne de "***stream***", dans ce cas la méthode "***map***" sera exécutée seulement deux fois. Donc, plutôt que de mapper tous les éléments du "***stream***", la méthode "***map***" sera appelé aussi peu que possible.

Pourquoi l'ordre est important

L'exemple suivant est constitué de deux opérations intermédiaires "***map***" et "***filtre***" et une opération terminale "***forEach***". Essayons de nouveau de voir de près la manière dont ces opérations sont exécutées:

```
Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
}).filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("A");
}).forEach(s -> System.out.println("forEach: " + s));
```

```
/*
 * map: d2
 * filter: D2
 * map: a2
 * filter: A2
 * forEach: A2
 * map: b1
 * filter: B1
 * map: b3
 * filter: B3
 * map: c
 * filter: C
 */
```

Comme vous l'aurez deviné les opérations "***map***" et "***filtre***" sont appelées cinq fois dans la collection alors que la méthode "***forEach***" est appelée une seule fois.

Nous pouvons réduire le nombre réel d'exécutions si nous changeons l'ordre des opérations, en déplaçant "***filtre***" au début de la chaîne:

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
}).map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

```
/*
 * filter: d2
 * filter: a2
 * map: a2
 * forEach: A2
 * filter: b1
 * filter: b3
 * filter: c
 */
```

Maintenant, la méthode "***map***" est appelée une seule d'où les opérations du pipe-line beaucoup plus rapide pour un nombre d'éléments d'élément plus grand. Gardez cela à l'esprit lors de la composition d'une chaîne des méthodes complexes.

Étendons l'exemple ci-dessus par une opération supplémentaire de tri "***sorted***":

```
Stream.of("d2", "a2", "b1", "b3", "c").sorted((s1, s2) -> {
    System.out.printf("sort: %s; %s\n", s1, s2);
    return s1.compareTo(s2);
}).filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
}).map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

```
/*
 * sort: a2; d2
 * sort: b1; a2
 * sort: b1; d2
 * sort: b1; a2
 * sort: b3; b1
 * sort: b3; d2
 * sort: c; b3
 * sort: c; d2
 * filter: a2
 * map: a2
 * forEach: A2
 * filter: b1
 * filter: b3
 * filter: c
 * filter: d2
 */
```

Le tri est un type spécial de fonctionnement intermédiaire. C'est une opération dite ***"stateful"*** car pour trier une collection d'éléments vous devez maintenir l'état lors du tri.

Tout d'abord, l'opération de tri est exécutée sur tous les éléments du ***"stream".*** En d'autres termes "***sorted***" est exécutée horizontalement. Donc dans ce cas "***sorted***" est appelé huit fois pour plusieurs combinaisons sur chaque élément du "***stream***".

Une fois de plus, nous pouvons optimiser les performances en réorganisant la chaîne:

```
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> {
    System.out.println("filter: " + s);
    return s.startsWith("a");
}).sorted((s1, s2) -> {
    System.out.printf("sort: %s; %s\n", s1, s2);
    return s1.compareTo(s2);
}).map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
}).forEach(s -> System.out.println("forEach: " + s));
```

```
/*
 * filter: d2
 * filter: a2
 * filter: b1
 * filter: b3
 * filter: c
 * map: a2
 * forEach: A2
 */
```

Dans cet exemple "***sorted***" n'est jamais été appelé parce "filtre" réduit le nombre des éléments à un seul élément. Donc la performance est considérablement améliorée notamment pour les grand nombre d'éléments.

V. Date/Time API

Java 8 arrive avec une nouvelle API pour la gestion du temps, via le package "***java.time***". Elle est inspirée de la librairie "***Joda-Time***". "***Stephen Colebourne***", créateur de "***JodaTime***", a participé à l'élaboration de celle-ci. Son but est de combler les défauts des vieillissantes API "***Date***" (JDK 1.0) et "***Calendar***" (JDK 1.1) en introduisant de nouveaux concepts.

Les exemples suivants couvrent les parties les plus importantes de cette nouvelle API.

- **Clock**

Horloge permet d'accéder à la date et l'heure courante. ***Clocks*** ont un fuseau horaire et peuvent être utilisés à la place du "***System.currentTimeMillis()"*** pour récupérer les

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant); // legacy java.util.Date
```

millisecondes actuelles. Un point instantané sur la ligne de temps est également représenté par la classe "***Instant***". Instants peuvent être utilisés pour créer des objets du package ***"java.util.Date"*.**

- **Timezones**

"Timezones" sont représentés par un ***"ZoneId"***. Ils peuvent être facilement accessibles via des méthodes statiques. Ils définissent les décalages horaires ce qui est important pour convertir entre les instants et les dates et heures locales.

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Paris");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

- **LocalTime**

"LocalTime" représente un temps sans fuseau horaire ("***timezone***"), par exemple 10pm ou 17:30:15. L'exemple suivant crée deux heures locales pour les fuseaux horaires définis ci-dessus. Puis nous comparons les deux temps et calculons la différence en heures et en minutes entre eux.

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
```

"LocalTime" est livré avec diverses méthodes pour simplifier la création de nouvelles instances, tel que le parsing des chaines de caractères.

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

*"**LocalDate**"* représente une date distincte, par exemple 11/03/2014. C'est une date immuable et fonctionne exactement d'une manière analogue au *"**LocalTime**"*. L'exemple montre comment calculer des nouvelles dates en ajoutant ou soustrayant des jours, mois ou années. Gardez à l'esprit que chaque manipulation retourne une nouvelle instance.

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```

Créer une *"**LocalDate**"* à partir d'une chaine de caractère est aussi simple que la création d'un *"**LocalTime**"*:

```
DateTimeFormatter germanFormatter =
        DateTimeFormatter
            .ofLocalizedDate(FormatStyle.MEDIUM)
            .withLocale(Locale.GERMAN);

   LocalDate xmas = LocalDate.parse("19.03.2015", germanFormatter);
   System.out.println(xmas);   // 2015-03-19
```

- **LocalDateTime**

"LocalDateTime" combine la date et l'heure comme dans une instance. ***"LocalDateTime"*** est immuable et fonctionne de manière similaire à ***"LocalTime"*** et ***"LocalDate"***. Nous pouvons utiliser des méthodes pour récupérer certains champs d'une date-heure:

```
LocalDateTime maga = LocalDateTime.of(2015, Month.MARCH, 19, 23,
        59, 59);

DayOfWeek dayOfWeek = maga.getDayOfWeek();
System.out.println(dayOfWeek); // THURSDAY

Month month = maga.getMonth();
System.out.println(month); // MARCH

long minuteOfDay = maga.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay); // 1439
```

Avec l'information supplémentaire d'un fuseau horaire ("***TimeZone***"), ***LocalDateTime"*** peut être converti en un Instant. Instants peuvent être facilement convertis en dates de type ***"java.util.Date"***.

```
Instant instant = maga.atZone(ZoneId.systemDefault()).toInstant();

Date magaDate = Date.from(instant);
System.out.println(magaDate); // Thu Mar 19 23:59:59 CET 2015
```

Le formatage de ***"LocalDateTime"*** fonctionne exactement comme ***"LocalDate"*** ou ***"LocalTime"***. Au lieu d'utiliser les formats prédéfinis, nous pouvons créer formateurs de modèles personnalisés.

```java
DateTimeFormatter formatter = DateTimeFormatter
        .ofPattern("MM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("03 19, 2015 - 07:13",
        formatter);
String string = formatter.format(parsed);
System.out.println(string); // 03 19, 2015 - 07:13
System.out.println(parsed); // 2015-03-19T07:13
```

Contrairement ***"java.text.NumberFormat"*** la nouvelle ***"DateTimeFormatter"*** est immuable et thread-safe.

VI. Repeating annotations

En Java 7 et les versions antérieures, il est interdit d'attacher plus qu'une annotation du même type à la même partie du code (par exemple, une classe ou une méthode). Par conséquent, les développeurs ont dû les regrouper en simple annotation appelé annotation conteneur, en tant que solution de contournement:

```java
@Authors({
    @Author(name = "Andres"),
    @Author(name = "Lionel")
    })
public class Book {

}
```

Java 8 a apporté un petit mais très utile amélioration appelé "***repeating annotations***" (annotations répétées) qui nous permet de réécrire le même code sans utiliser explicitement l'annotation conteneur:

```java
@Author(name = " Andres ")
@Author(name = " Lionel ")
public class Book {

}
```

Pour des raisons de compatibilité, l'annotation conteneur est encore utilisée, mais cette fois le compilateur Java est responsable de l'enveloppement des annotations répétées dans l'annotation conteneur. Les annotations définies par l'utilisateur ne sont pas reproductibles par défaut et doivent être annotée avec l'annotation **@Repeatable**:

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {

    Author[] value();

}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(Authors.class)
public @interface Author {

    String name() default "Lionel";

}
```

La valeur de l'annotation **@Repeatable** représente le type de l'annotation conteneur:

Lors l'annotation "***Author***" est utilisé plusieurs fois sur la même partie du code, le compilateur Java crée automatiquement l'annotation conteneur "***Authors***" et mémorise toutes annotations ***"Author"*** dans son élément ***"value".***

- **Accès aux annotations via l'API reflection**

Les annotations ***"@Author"*** peuvent être accessibles de deux façons :

La première est d'y accéder à partir de l'annotation conteneur ***"@Authors"*** en utilisant ***"getAnnotation()"*** ou ***"getDeclaredAnnotation()"*** de l'interface ***"AnnotatedElement":***

```java
Class<Book> klazz = Book.class;
Authors authors = klazz.getAnnotation(Authors.class);
for (Author author : authors.value())
    System.out.println("Author=" + author.name());
    // Author= Andres
    // Author= Lionel
```

La deuxième est à travers la nouvelle méthode de l'API ***"Reflection : getAnnotationsByType()"*** qui scanne l'annotation conteneur afin d'extraire et retourne les annotations répétés sous forme d'un tableau.

```java
Author[] authors = klazz.getAnnotationsByType(Author.class);
for (Author author : authors)
    System.out.println("Author=" + author.name());

    // Author= Andres
    // Author= Lionel
```

La Répétition des annotations est un petit plus en Java qui simplifie l'utilisation de certaines annotations dans divers cadres (surtout dans Hibernate et JPA). Parce que la fonction est tout à fait nouvelle, vous aurez toujours besoin d'utiliser explicitement une annotation conteneur pour un certain temps (jusqu'à **@Repeatable** est ajouté à tous les types d'annotation nécessaires).