Java 8

Sommaire

Petit sondage ?

Qui connait déjà

- Les concepts de la programmation fonctionnelle ?
- Les lambdas ?
- Les streams ?

Attention

Dans la suite de cette présentation les types sont parfois simplifiés pour aider à la compréhension.

Ainsi les signatures ont été simplifiées en supprimant la variance : <`? super`...> et <`? extends`...>

```
Function<? super T, ? extends K> f;
```

Devient :

```
Function<T, K> f;
```

Exemple simple (Java 7)

Imaginons une application qui gère des **personnes** :

```
class Person {
    private String name;
    private int age;
    private double weight;
    private double height;
    // ...
}
```

Nous devons **filtrer** ces personnes selon leur **age** :

```
List<Person> personsFilteredByAge(List<Person> allPersons, int a) {
    List<Person> result = new ArrayList<>();

    for (Person person : allPersons) {

        if (person.getAge() == a) {
            result.add(person);
        }
    }
    return result;
}
```

Nous devons aussi les **filtrer** selon leur **poids**, **taille**, etc. :

```
List<Person> personsFilteredByWeight(List<Person> allPersons, double w) {
    List<Person> result = new ArrayList<>();

    for (Person person : allPersons) {
        if (person.getWeight() == w) {
            result.add(person);
        }
    }
    return result;
}
```

STOP !!!

D.R.Y!

En programmation il faut respecter le plus possible le principe **D.R.Y.**

- **D**on't
- **R**epeat
- **Y**ourself !

> «Measuring programming progress by lines of code is like measuring aircraft building progress by weight.»
> — Bill Gates /div>

Introduisons l'interface `Condition` pour **abstraire le filtre** :

```
interface Condition {
    boolean check(Person p);
}
```

Nous pouvons **généraliser** la fonction de filtrage de `Person` :

```
List<Person> personsFilteredBy(List<Person> allPersons, Condition condition) {
    List<Person> result = new ArrayList<>();

    for (Person person : allPersons) {
        if (condition.check(person)) {
            result.add(person);
        }
    }

    return result;
}
```

Le filtrage des personnes par age **devient** :

```
List<Person> personsFilteredByAged(List<Person> allPersons, int a) {

    return personsFilteredBy(allPersons, new Condition() {
        @Override
        public boolean check(Person p) {
            return p.getAge() == a;
        }
    });

}
```

Il se trouve que notre application gère aussi des **maisons**.

Modifions l'interface `Condition` pour être plus **générique** :

```java
interface Condition<T> {
    boolean check(T value);
}
```

Ecrivons une methode de **filtrage générique** :

```java
<T> List<T> filteredBy(List<T> allItems, Condition<T> condition) {
    List<T> result = new ArrayList<>();

    for (T item : allItems) {
        if (condition.check(item)) {
            result.add(item);
        }
    }

    return result;
}
```

Le filtrage des personnes par age **devient** :

```java
List<Person> personsFilteredByAged(List<Person> allPersons, int a) {

    return filteredBy(allPersons, new Condition<Person>() {
        @Override
        public boolean check(Person p) {
            return p.getAge() == a;
        }
    });

}
```

Grace à la méthode `filteredBy()` nous pouvons :

- filtrer selon **plusieurs critères**.
- filtrer **plusieurs types**.

Quel est le problème avec la `Condition` suivante ?

```java
new Condition<Person>() {
    @Override
    public boolean check(Person p) {
        return p.getAge() == a;
    }
}
```

- Seul `p.getAge() == a` est du code utile.
- Tout le reste est du **Boilerplate Code**, du code **sans aucune valeur ajoutée**, du code trop verbeux.

Conclusion sur Java 7

Java 7 est un language **trop verbeux** qui ne permet pas facilement l'utilisation d'API de plus **hauts niveaux d'abstractions**.

Sommaire

Java 8

Historique sélectif de Java

- 1996 Java 1.0
- 1997 Java **1.1** : anonymous inner class
- 1998 Java **1.2** : `Collection`
- 2000 Java 1.3
- 2002 Java 1.4
- 2004 Java **1.5** : `Generics<T, U>`
- 2006 Java 1.6
- Sun -> Oracle
- **2011** Java 1.7
- 2014 Java **1.8** : Lambda & `Stream`

Interface (méthodes statiques et par défaut)

En java 8 les interfaces peuvent aussi définir :

- des implementations par defaut (`default`) qui peuvent être surchargées (`@override`) par une autre implémentation par défaut dans des sous-interfaces ou par des implementations dans les classes implémentants cette interface.
- des méthodes statiques (`static`).

```java
interface MyInterface {

    long uneMethodeAImplementer(String s, int i);

    default int add(int a, int b) { return a + b; }

    default String addExtension(String name, String ext) {
        return name + "." + ext;
    }

    static Person createPerson(String name, int age) {
        return new Person(name, age);
    }
}
```

Interface (appels aux méthodes statiques et par défaut)

L'utilisateur de cette interface peut les utiliser directement.

Méthodes statiques

```java
Person toto = MyInterface.createPerson("toto", 10);
```

Implémentation par défault

```
MyInterface value = ...;
int sum = value.add(10, 5);
String music = value.addExtension("super-song", "mp3");
```

Interface (@FunctionalInterface)

En Java 8, `@FunctionalInterface` sert à indiquer que l'interface ne doit contenir **qu'une seule** méthode abstraite (non définie dans `Object`, cf. `Comparator`).

Méthode qui devra être implementée par une classe qui implemente l'interface.

C'est utilisé comme **contrainte pour le compilateur**. C'est semblable à `@Override` pour la vérification de la bonne redéfinition de méthode par héritage.

C'est le cas pour notre interface `Condition<T>` :

```
@FunctionalInterface
interface Condition<T> {
     boolean check(T p);
}
```

Dans ce cas on peut simplifier l'écriture de l'implémentation...

Lambda

Partons d'une condition telle qu'on pourrait l'écrire en Java 7 :

```
Condition<Person> cond = new Condition<Person>() {

    @Override
    public boolean check(Person p) {
        return p.getAge() == 18;
    }

};
```

En Java 8 elle peut s'écrire :

```
Condition<Person> cond = (Person p) -> {
    return p.getAge() == 18;
};
```

Lambda (une seule instruction)

```
Condition<Person> cond = (Person p) -> {
    return p.getAge() == 18;
};
```

S'il n'y a qu'**une seule instruction** (ici `return`), on peut transformer le block `{...}` en expression :

```
Condition<Person> cond = (Person p) -> p.getAge() == 18;
```

Lambda (inférence de type)

```
Condition<Person> cond = (Person p) -> p.getAge() == 18;
```

Le **type** du paramètre peut être **inféré** par le compilateur (ici `Person`) :

```
Condition<Person> cond = (p) -> p.getAge() == 18;
```

Lambda (un seul paramètre)

```
Condition<Person> cond = (p) -> p.getAge() == 18;
```

S'il n'y a qu'**un seul paramètre** alors les parenthèses peuvent être supprimées.

```
Condition<Person> cond = p -> p.getAge() == 18;
```

Si la lambda n'a **aucun** paramètre il faut alors utiliser : `() -> ...`

Pour rappel :

```
Condition<Person> cond = new Condition<Person>() {

    @Override
    public boolean check(Person p) {
        return p.getAge() == 18;
    }

};
```

Lambda (scope des variables)

Une lambda n'a accès **qu'en lecture** aux variables qui sont dans son `scope` :

```java
int i = 0;

Consumer<String> consumer = s -> {
    i = i + 1;              // /!\ NE COMPILE PAS /!\
    System.out.println(s);
};
```

La **mutation** de variables est une pratique qu'il faut **éviter**, toutefois si c'est ce que vous voulez faire, il faut utiliser un `AtomicRef<T>` ou dérivés comme `AtomicInteger`, etc. :

```java
AtomicInteger i = new AtomicInteger();

Consumer<String> consumer = s -> {
    i.incrementAndGet();   // Mutation de l'AtomicInteger
    System.out.println(s);
};
```

Lambda (résumé)

- Interface avec méthodes `default` et `static`.
- Interface avec **une seule** méthode **abstraite** est une `@FunctionalInterface`.
- On peut utiliser une **lambda** pour implémenter une `@FunctionalInterface`.

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8
  - Lambda **✔**
  - Interfaces fonctionnelles prédéfinies
  - Référence de méthode
  - Programmation fonctionnelle
  - Optional
  - Types monadiques
  - Comparator
  - Stream
  - Collectors
  - Monoïdes
  - Stream parallèle
  - Autres ajouts de Java 8

Interfaces fonctionnelles prédéfinies (Function)

Dans le JDK il existe des `@FunctionalInterface`s prédéfinies dans le package `java.util.function`.

Elles sont de la forme générale : `f(TypeEntrée) => TypeSortie`

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`

```java
@FunctionalInterface
public interface Function<A, B> {

    B apply(A a);

}
```

Interfaces fonctionnelles prédéfinies (Predicate)

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`
`A`	`boolean`	`Predicate<A>`

```java
@FunctionalInterface
public interface Predicate<A> {

    boolean test(A a);

}
```

Interfaces fonctionnelles prédéfinies (Consumer)

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`
`A`	`boolean`	`Predicate<A>`
`A`	`void`	`Consumer<A>`

```java
@FunctionalInterface
public interface Consumer<A> {

    void accept(A a);

    ...
}
```

Interfaces fonctionnelles prédéfinies (Supplier)

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`
`A`	`boolean`	`Predicate<A>`
`A`	`void`	`Consumer<A>`
`/`	`B`	`Supplier<B>`

```java
@FunctionalInterface
public interface Supplier<B> {

    B get();

}
```

Interfaces fonctionnelles prédéfinies (UnaryOperator)

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`
`A`	`boolean`	`Predicate<A>`
`A`	`void`	`Consumer<A>`
`/`	`B`	`Supplier<B>`
`A`	`A`	`UnaryOperator<A>`

```java
@FunctionalInterface
public UnaryOperator<A> extends Function<A, A> {

    ...

}
```

Interfaces fonctionnelles prédéfinies (Runnable)

Type d'entrée	Type de sortie	Interface
`A`	`B`	`Function<A,B>`
`A`	`boolean`	`Predicate<A>`
`A`	`void`	`Consumer<A>`
`/`	`B`	`Supplier<B>`
`A`	`A`	`UnaryOperator<A>`
`/`	`void`	`Runnable`

```
@FunctionalInterface
public interface Runnable {

    void run();

}
```

Interfaces fonctionnelles prédéfinies à 2 paramètres

Il existe d'autres fonctions (`Bi...`) dans le package `java.util.function` qui prennent **deux paramètres** en entrée.

Types d'entrée	Type de sortie	Interface
`A,B`	`C`	`BiFunction<A,B,C>`
`A,A`	`A`	`BinaryOperator<A>`
`A,B`	`boolean`	`BiPredicate<A,B>`
`A,B`	`void`	`BiConsumer<A,B>`

Interface (exercice)

Retrouver les types définis dans `java.util.function` ayant la même forme que des interfaces existantes.

`IsomorphFunctionsTest`

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8

- Lambda **✔**
- Interfaces fonctionnelles prédéfinies **✔**
- Référence de méthode
- Programmation fonctionnelle
- Optional
- Types monadiques
- Comparator
- Stream
- Collectors
- Monoïdes
- Stream parallèle
- Autres ajouts de Java 8

Référence de méthode statique

Pour une lambda qui ne fait que transmettre le paramètre à une méthode **statique** :

```
1  Function<Integer, String> stringify = i -> String.valueOf(i);
```

Methode définie dans la classe `String` :

```
1  static String valueOf(int i)
```

Peut s'écrire en faisant référence à cette méthode statique :

```
1  Function<Integer, String> stringify = String::valueOf;
```

Référence à une méthode

Pour une lambda utilisant une méthode d'instance **non liée** à un objet en particulier :

```
1  Function<JFrame, String> naming = f -> f.getTitle();
```

Methode définie dans la classe `JFrame` :

```
1  String getTitle()
```

Peut s'écrire en faisant référence à cette méthode :

```
1  Function<JFrame, String> naming = JFrame::getTitle;
```

Référence à une méthode (exemple)

Autre exemple :

```
1  BiConsumer<JFrame, String> titrage = (f, t)-> f.setTitle(t);
```

Methode définie dans la classe `JFrame` :

```
void setTitle(String title)
```

Peut s'écrire en faisant référence à cette méthode.:

```
BiConsumer<JFrame, String> titrage = JFrame::setTitle;
```

Référence à une méthode d'object (exemple)

Pour une lambda utilisant une méthode d'instance **liée** à un objet en particulier :

```
JFrame f = getFrame();

Consumer<String> titrage = t -> f.setTitle(t);
```

Méthode définie dans la classe `JFrame` :

```
void setTitle(String title)
```

Peut s'écrire en faisant référence à cette méthode d'un objet précis :

```
Consumer<String> titrage = f::setTitle;
```

Référence à une méthode d'object (exemple)

Autre exemple :

```
Consumer<String> logger = t -> System.out.println(t);
```

Méthode définie dans la classe `PrintStream` :

```
void println(String x)
```

Peut s'écrire en faisant référence à cette méthode d'un objet précis :

```
Consumer<String> logger = System.out::println;
```

Référence de constructeur

Pour une lambda qui appelle un constructeur par défault :

```
Supplier<Person> personFactory = () -> new Person();
```

Peut s'écrire en faisant référence au constructeur :

```
Supplier<Person> personFactory = Person::new;
```

Autre exemple :

```
Function<String, JFrame> createJFrame = t -> new JFrame(t);
```

Peut s'écrire en faisant référence au constructeur :

```
Function<String, JFrame> createJFrame = JFrame::new;
```

Référence de constructeur de tableau

Pour une lambda qui appelle la creation d'un **tableau** :

```
Function<Integer, String[]> stringArrayFactory = len -> new String[len];
```

Peut s'écrire en faisant référence au constructeur du tableau.

```
Function<Integer, String[]> stringArrayFactory = String[]::new;
```

Référence de méthode (conversion entre @FunctionalInterface)

Parfois il est utile de convertir une `@FunctionalInterface` vers une autre `@FunctionalInterface` **isomorphe**.

Comment passer d'un `Predicate<File>` :

```
Predicate<File> filePredicate = f -> f.isDirectory();
```

A un `FileFilter` :

```
@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}
```

```
FileFilter ff =  f -> filePredicate.test(f);
```

Peut être réécrit en utilisant une référence sur la **seule méthode abstraite** de l'`@FunctionalInterface` source (ici `Predicate.test(T)`):

```
FileFilter ff = filePredicate::test;
```

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8
    - Lambda **✔**
    - Interfaces fonctionnelles prédéfinies **✔**
    - Référence de méthode **✔**
    - Programmation fonctionnelle
    - Optional
    - Types monadiques
    - Comparator
    - Stream
    - Collectors
    - Monoïdes
    - Stream parallèle
    - Autres ajouts de Java 8

Programmation fonctionnelle (introduction)

- La programmation fonctionnelle est un paradigme de programmation qui a comme ambition de **se rapprocher des valeurs des mathématiques**.
- Ses origines peuvent être trouvées dans les travaux d'Alonzo Church sur le **lambda calcul** dans les années 1930, ou ceux de Moses Schönfinkel et d'Haskell Curry sur la logique combinatoire dans les années **1920**.
- Le plus ancien langage fonctionnel est **Lisp** créé par John McCarthy en **1958**, il est aujourd'hui le deuxième plus vieux langage encore utilisé (derrière Fortran 1954). Il a beaucoup évolué depuis le début des années 1960 et a ainsi donné naissance à de nombreux dialectes (Common Lisp, Scheme, Clojure sur la JVM, etc.).

Programmation fonctionnelle (langages pour la JVM)

Voici une liste de langages s'exécutant sur la JVM et ayant des concepts issus de la programmation fonctionnelle :

- 2001 : **Scala** (V1.0 2004)
- 2003 : **Groovy** (V1.0 2007)
- 2007 : **Clojure** (V1.0 2009)
- 2010 : Ceylon (Red Hat V1.0 2013)
- 2012 : Kotlin (JetBrains)
- 2014 : Java 8

Programmation fonctionnelle («What...»)

- Dans le paradigme impératif (C, Java7, etc.), un programme représente **ce que l'on doit faire** :

«**What to do**»

```
List<Person> result = new ArrayList<>();

for (Person person : allPersons) {
    if (person.getAge() > 18) {
        result.add(person);
    }
}
```

- Dans le paradigme fonctionnel, un programme tend à représenter **ce que l'on veut** d'une manière plus **déclarative** :

«**What you want**»

```
List<Person> result = filter(allPersons, p -> p.getAge() >= 18);
```

Programmation fonctionnelle (citations)

> «Functional programming is so called because its fundamental operation is the application of functions to arguments.»
> — Why Functional Programming Matters.

> «La programmation fonctionnelle est un style de programmation qui met l'accent sur l'évaluation d'expressions, plutôt que sur l'execution de commandes.»
> — Graham Hutton.

Programmation fonctionnelle (valeurs)

La programmation fonctionnelle repose sur **des valeurs** formant un **ensemble cohérent** :

- privilégier l'utilisation de **fonctions pures sans effets de bord**
- privilégier l'**immutabilité**
- raisonner grâce à la **transparence référentielle**
- l'utilisation de fonctions d'**ordre supérieur**
- privilégier l'**évaluation paresseuse**
- l'utilisation des types pour avoir plus de contrôle (par exemple pour **ne pas utiliser null**)

Programmation fonctionnelle (fonction pure)

Une fonction est dite **pure** lorsque sa valeur de sortie (son résultat) dépend **uniquement** de ses paramètres d'entrée.

Une fonction **pure** ne produit **aucun effet de bord**.

Par exemple :

```
double moyenne(double a, double b) {
  return (a + b) /2;
}
```

Programmation fonctionnelle (effets de bord)

Voici une liste de ce que recouvre le terme **effet de bord** :

- Entrée / Sortie
  - clavier
  - écran
  - fichier
  - base de données
  - réseau
- Mutation
  - d'une variable globale
  - d'un paramètre
- Lever une exception

Programmation fonctionnelle (fonction pure)

Une fonction pure est **idempotente**, c'est-à-dire retourne toujours le même résultat pour le même jeu de paramètres en entrée.

L'utilisation de fonction pure permet de "***raisonner***" avec le programme, d'écrire des équivalences (un peu comme en maths).

```
1  double z = f(x) + f(x);
```

peut se réécrire **en toute sécurité** :

```
1  double z = 2 * f(x);
```

C'est ce que l'on appelle la **transparence référentielle**, lorsqu'un appel à une fonction peut être remplacé par son résultat sans changer le **comportement** du programme.

Programmation fonctionnelle (exemple de fonction impure)

Voici une fonction **impure** :

```
1  private int i = 0;
2
3  double f(double x) {
4      System.out.println("X vaut " +x); // /!\ Effet de bord /!\
5      return 10 * x + (i++); // Effet de bord (i++) !
6  }
```

Dans ce cas, 2 appels à la fonction impure **ne se comportent pas** comme 2 multiplié par le résultat d'un appel :

```
1  f(x) + f(x) != 2 * f(x)
```

Programmation fonctionnelle (classes immutables du JDK)

Objet dont la valeur est donnée à la construction et qui ne **change plus dans le temps**. Les attributs sont constants, y compris les relations avec d'autres objets (collections d'objets).

- Exemples de classes immutables du JDK :
  - `String`
  - `File`
  - `Integer`, `Double`, etc.
  - `Color`
  - `Font`
  - `UUID`
  - `URL`
  - `java.time.*` (en Java 8)

En programmation fonctionnelle on applique des fonctions pures qui permettent de créer une structure de données immutable à partir de données immutables.

Programmation fonctionnelle (classe immutable)

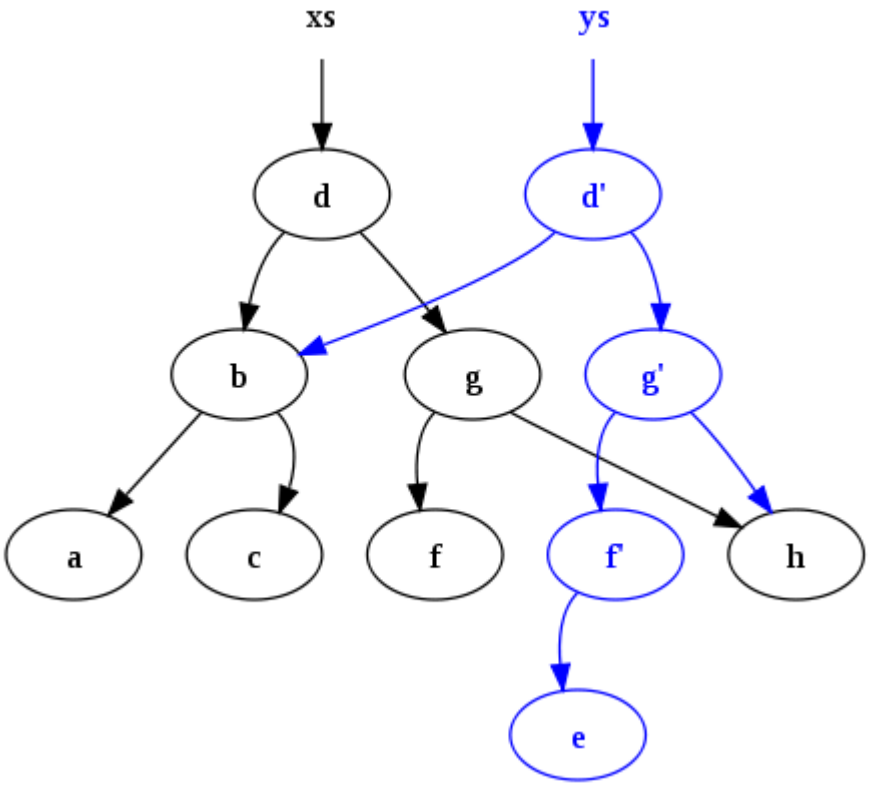
```
1  public final class Personne {
2      private final String nom;
3      private final int age;
4
5      public Personne(String nom, int age) {
6          this.nom = nom;
7          this.age = age;
8      }
9
10     public String getNom() {  return nom; }
11
12     public int getAge() { return age; }
13
14     public Personne vieillir(int nbAnnees) {
15         return new Personne(nom, age + nbAnnees);
16     }
17 }
```

Dans la méthode `vieillir(...)` on crée une nouvelle personne.

Programmation fonctionnelle (structures de données persistantes)

Ici persistante **n'a rien à voir avec le stockage** (JPA, Hibernate, etc.).

La **persistante** signifie que si on modifie une structure de données, on obtient une nouvelle structure de données qui **partage** beaucoup de données avec la structure de données originale.



Seulement un **petit groupe de valeurs ont été copiées**.

La version **originale** n'a absolument **pas été modifiée** et reste utilisable.

Programmation fonctionnelle (librairie proposant l'immutabilité)

La librairie `Google Guava` fournit des implémentations immutables pour les collections Java (cependant cette implémentation n'est pas "persistante") :

```
https://github.com/google/guava/wiki/ImmutableCollectionsExplained
```

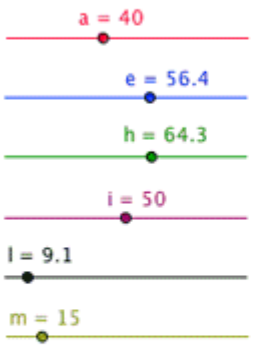
Programmation fonctionnelle (bénéfices de l'immutabilité)

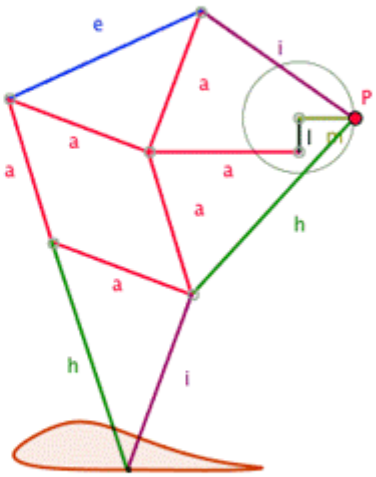
Bénéfices de l'immutabilité :

- Stable dans le temps :
  - Peut être utilisé sans crainte comme clé dans une `Map` ou élément de `Set`.
  - Le `hashcode()` est constant et peut donc n'être calculé qu'une seule fois et stocké.
- Thread safe.
- Peut être passé dans une fonction sans crainte (ne nécessite **pas de copie défensive**).
- Besoin de vérifier les **invariants** qu'à la construction.
- Si un objet immutable génère une exception on est sûr qu'il n'est **pas dans un état indésirable** / instable.

Programmation fonctionnelle (immutabilité, analogie avec la mécanique)

Il est plus simple d'étudier un système quand il y a un nombre limité de **pièces en mouvement**.





Programmation fonctionnelle (danger de la mutabilité)

Le danger avec le partage de structure de données **mutables**, c'est que si un utilisateur la corrompt alors tous sont impactés.

Programmation fonctionnelle (fonction d'ordre supérieur)

Une fonction d'ordre supérieur est une fonction qui prend une fonction en paramètre et/ou retourne une fonction.

Exemple de fonctions d'ordre supérieur :

```
<T> List<T> filter(Collection<T> items, Predicate<T> predicate)
```

```
UnaryOperator<Double> multiplyBy(double coef) {
        return x -> x * coef;
}
```

Ici on n'a pas seulement une fonction qui retourne un résultat, mais une fonction qui retourne une fonction qui pourra évaluer le résultat.

Par exemple on peut imaginer **différer** son évaluation, ou l'évaluer plusieurs fois avec plusieurs paramètres.

Programmation fonctionnelle (exemple de fonctions d'ordre supérieur)

Soit deux `Function` :

```
Function<A, B> aToB = ...;
Function<B, C> bToC = ...;
```

L'interface `Function<T, R>` possède deux fonctions par défault pour composer des fonctions :

- f.compose(g) : x -> f(g(x)), en mathématique (f "rond" g) : f ∘ g

```
default <V> Function<V, R> compose(Function<V, T> before)

Function<A, C> aToC = bToC.compose(aToB);
```

- f.andThen(g) : x -> g(f(x))

```
default <V> Function<T, V> andThen(Function<R, V> after)

Function<A, C> aToC = aToB.andThen(bToC);
```

Programmation fonctionnelle (évaluation paresseuse)

En Java les paramètres sont évalués avant d'être passés à la fonction qui va les utiliser.

```
void logMessage(String msg) {
    if (isLogEnable()) {
        logSystem().append(msg);
    }
}
```

Appel de la méthode :

```
logMessage("v: " + calculLongDeLaValeurDeV());
```

Dans l'appel ci-dessus, le long calcul est effectué et seulement ensuite la méthode `logMessage(...)` est appelée et ne loggera peut être pas le message :-(

Programmation fonctionnelle (évaluation paresseuse)

Refactoring pour avoir une évaluation paresseuse du calcul du message en utilisant un `Supplier<String>` :

```
void logMessage(Supplier<String> msgSupplier) {
    if (isLogEnable()) {
        logSystem().append(msgSupplier.get());
    }
}
```

L'appel de la méthode fournit donc un `Supplier<String>` en paramètre :

```
logMessage(() -> "v: " + calculLongDeLaValeurDeV());
```

La fonction `calculLongDeLaValeurDeV()` n'est appelée **que si c'est nécessaire**.

Programmation fonctionnelle (bannir l'usage de Null)

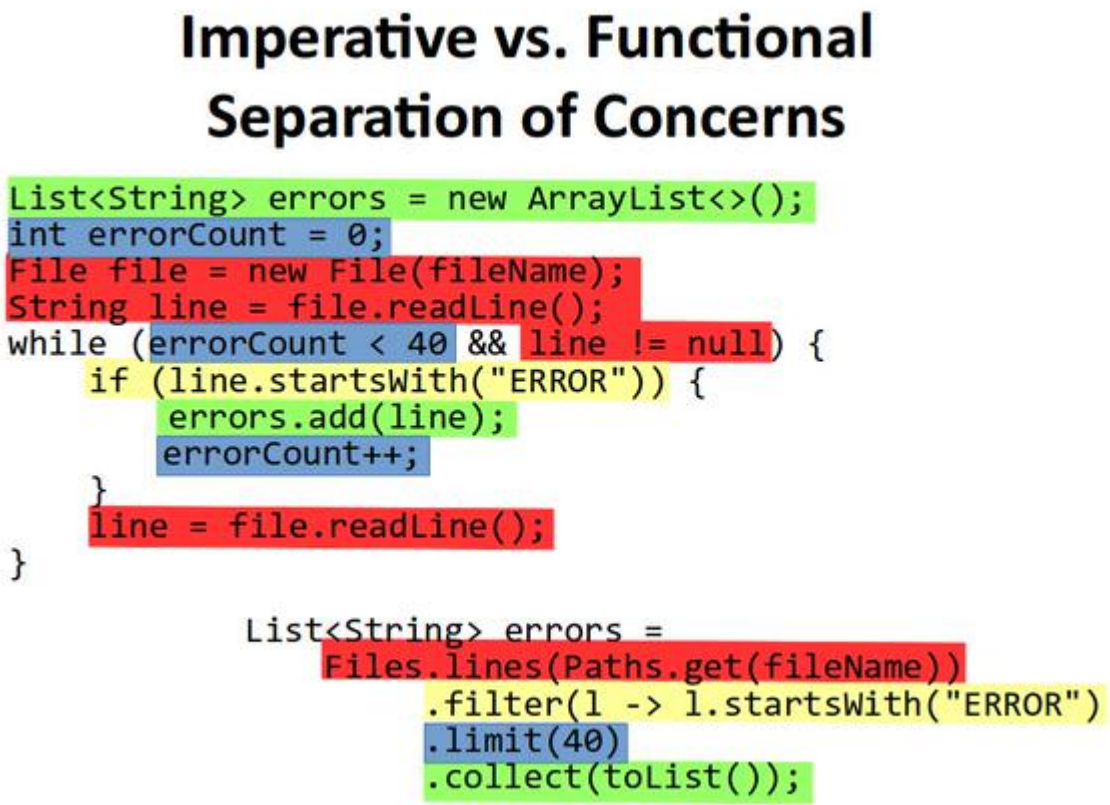
> «Je l'appelle mon **erreur à un milliard de dollars**. Il s'agit de **l'invention de la valeur null** pour un pointeur, en 1965. À l'époque, je concevais le premier système de typage complet pour un langage orienté objet (Algol W). Je voulais m'assurer que tout usage de références était absolument sûr, avec un test effectué automatiquement par le compilateur. Mais je n'ai pas pu résister à ajouter la référence nulle, simplement parce que c'était si facile à implémenter. **Ceci a conduit à un nombre incalculable d'erreurs**, de déficiences, de **plantages** de système, qui ont probablement causé des problèmes et des dommages d'un milliard de dollars dans les quarante dernières années.»
> — Charles Antony Richard Hoare.

Programmation fonctionnelle (Résumé)

Savoir décrire un algo de manière fonctionnelle permet d'avoir accès à des **API de plus hauts niveaux d'abstractions** par combinaison de functions **élémentaires**, facilitant :

- l'**expressivité**
- le **parallélisme** (si c'est possible)
- l'**asynchronisme**

Par ailleurs la programmation fonctionnelle permet une meilleure **séparation des responsabilités** (SoC - Separation of Concerns).



La programmation fonctionnelle permet d'avoir du **code de meilleur qualité**.

Programmation fonctionnelle (comparons un code Java 7...)

```java
String listOldByDep(List<Person> persons) {
   Map<Integer, Person> oldByDep = new HashMap<>();
   for (Person p : persons) {
       Person prevOld = oldByDep.get(p.getDep());

       if (prevOld == null) {
           oldByDep.put(p.getDep(), p);
       } else if (prevOld.getAge() < p.getAge()) {
           oldByDep.put(p.getDep(), p);
       }
   }

   StringBuilder listing = new StringBuilder();
   for (Iterator<Map.Entry<Integer, Person>> iterator = oldByDep.entrySet().iterator();
        iterator.hasNext(); ) {
       Map.Entry<Integer, Person> entry = iterator.next();

       listing.append("Departement ").append(entry.getKey())
              .append(" : ").append(entry.getValue().getName());

       if (iterator.hasNext()) {
           listing.append(", ");
       }
```

```
    }
    return listing.toString();
}
```

Programmation fonctionnelle (... avec un langage moderne)

```
val listing = persons.groupBy(_.dep)
                     .mapValues(_.maxBy(_.age))
                     .map { case (dep, p) => s"Departement $dep : ${p.name}" }
                     .mkString(", ")
```

C'est la même fonctionnalité écrite en **Scala**.

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8
  - Lambda **✔**
  - Interfaces fonctionnelles prédéfinies **✔**
  - Référence de méthode **✔**
  - Programmation fonctionnelle **✔**
  - Optional
  - Types monadiques
  - Comparator
  - Stream
  - Collectors
  - Monoïdes
  - Stream parallèle
  - Autres ajouts de Java 8

Optional (problématique de Java 7)

Exemple de code **Java 7** problématique :

Soit la fonction suivante :

```
Person findPerson(Request req)
```

Exemple d'utilisation :

```
Person p = findPerson(req);
System.out.println(p.getName());
```

Potentiellement la recherche retourne **null** et lors de l'affichage une **NullPointerException** est levée :-(

Optional (introduction)

En programmation fonctionnelle typée, on utilise au maximum le **système de type** pour indiquer le plus de **sémantique** possible.

Java 8 introduit le type `Optional<T>` qui est un conteneur pour une valeur de type `T`, cependant ce conteneur **peut être vide**.

Exemple de signature de fonction retournant un `Optional` :

```
1  Optional<Person> findPersonOpt(Request req)
```

- La fonction ne doit **jamais retourner null** mais `Optional.empty()`.
- Ainsi l'utilisateur de la fonction sait, en lisant sa signature, que la **valeur de retour peut être absente** et doit gérer le cas correctement.

Optional (création)

Instanciation d'un `Optional<T>` :

Un Optional contenant une valeur (`value` doit être non null) :

```
1  static <T> Optional<T> of(T value)
```

Un Optional vide :

```
1  static<T> Optional<T> empty()
```

Un Optional contenant potentiellement une valeur :

```
1  static <T> Optional<T> ofNullable(T value)
```

Optional (présence)

Le type `Optional<T>` aussi simple soit-il, est un très bon point de départ pour appréhender les types utilisés en programmation fonctionnelle.

Tout d'abord, `Optional<T>` est un type **immutable**.

Une série d'opérateurs permettent soit de modifier un `Optional` soit d'utiliser sa valeur.

Opérateurs de base de `Optional` :

`isPresent()` indique la présence ou non d'une valeur.

```
1  boolean isPresent()
```

`orElse(T other)` retourne la valeur contenue dans l'`Optional` sinon retourne la valeur `other`.

```
T orElse(T other)

Optional.of(100).orElse(20); // 100
Optional.empty().orElse(20); // 20
```

Optional (évaluation paresseuse)

En Java l'évaluation des paramètres est dite "***stricte***", c'est-à-dire que tous les paramètres sont évalués et ensuite leurs valeurs sont transmises à la fonction.

```
unOptional.map(...)
          .orElse(uneValeurCouteuseACalculer());
```

Dans l'exemple ci-dessus, la valeur coûteuse est évaluée même si son résultat est inutile quand l'`Optional` contient une valeur.

Dans ce cas il existe une autre méthode : `orElseGet(Supplier<T> other)`.

`orElseGet(...)` retourne la valeur contenue dans l'`Optional` sinon retourne la valeur résultante de l'appel au `Supplier` `other`.

```
T orElseGet(Supplier<T> other)
```

```
unOptional.map(...)
          .orElseGet(() -> uneValeurCouteuseACalculer());
```

Optional (filtrage)

Opérateurs d'ordre supérieur :

`filter(Predicate<T> predicate)`, si une valeur est présente dans l'`Optional` et valide le `Predicate` alors `filter` retourne un `Optional` contenant cette valeur, sinon elle retourne un `Optional` vide :

```
Optional<T> filter(Predicate<T> predicate)
```

```
Optional.of(10).filter(i -> i % 2 == 0);  // Optional.of(10)
Optional.empty().filter(i -> i % 2 == 0); // Optional.empty
Optional.of(33).filter(i -> i % 2 == 0);  // Optional.empty
```

Optional (application)

`map(Function<T, U> mapper)`, si une valeur est présente dans l'`Optional` et `map` retourne un `Optional` correspondant à l'application de cette valeur à la fonction `mapper`, sinon retourne un `Optional` vide.

```
<U> Optional<U> map(Function<T, U> mapper)
```

```
Optional.of(3).map(i -> "_" + (i * 10) + "_");   // Optional.of("_30_")
Optional.empty().map(i -> "_" + (i * 10) + "_"); // Optional.empty
Optional.of(10).map(i -> null);                  // Optional.empty
```

Optional (effet de bord)

La méthode `ifPresent(...)` permet de générer un **effet de bord** à partir de la valeur de l'`Optional` **si elle est présente** :

```
void ifPresent(Consumer<T> consumer)
```

```
uneFonctionQuiRetourneUnOptional().ifPresent( valeur -> {
  System.out.println("On peut faire des effets de bord avec la valeur de l'Optional.");
});
```

Optional (anticipation de Java 9)

En Java 8 il manque une méthode permettant aussi d'effectuer un traitement en cas d'absence de valeur dans l'`Optional`.

Je vous conseille d'**anticiper** la réparation de cet oubli avec l'arrivée de **Java 9** en créant une petite méthode utilitaire.

Ainsi le developpeur n'aura plus à utiliser la méthode `Optional.get()` qui est source d'erreur, en effet `get()` lève une `NullPointerException` si elle est appellée par mégarde sur un `Optional` vide.

```
static <T> void ifPresentOrElse(Optional<T> opt,
                                Consumer<T> ifPresent,
                                Runnable    orElse) {
    if (opt.isPresent()) {
        ifPresent.accept(opt.get());
    } else {
        orElse.run();
    }
}
```

Optional (utilisation de ifPresentOrElse)

```
ifPresentOrElse(uneFonctionQuiRetourneUnOptional(),
  valeur -> {
    System.out.println("On peut faire des effets de bord avec la valeur de l'Optional.");
    ...
  },
  () -> {
    System.out.println("Dommage il n'y a pas de valeur :-(");
    ...
  });
```

Deviendra en **Java 9** :

```
uneFonctionQuiRetourneUnOptional().ifPresentOrElse(
  valeur -> {
    System.out.println("On peut faire des effets de bord avec la valeur de l'Optional.");
    ...
  },
  () -> {
    System.out.println("Dommage il n'y a pas de valeur :-(");
    ...
  });
```

Sommaire

Type monadique (introduction)

Nous allons à présent voir une catégorie particulière de types : **les types monadiques**.

Une monade est une structure de données qui représente un **traitement** dans un certain **contexte**.

- Une monade `M<T>` possède :
  - un moyen de **construction** (constructeur ou factory que nous nommeront `construct(...)`) ayant la signature :
    `A -> M<A>`
  - un opérateur de **composition** (généralement nommé `flatMap(...)` ou bind) permettant des enchaînements et ayant la signature:
    `(M<A>, A -> M<B>) -> M<B>`

Type monadique (3 lois)

- Une monade doit aussi respecter 3 lois (`construct(...)` est une sorte d'élément neutre pour `flatMap(...)`) :
  - Composition à gauche par construct : `construct(x).flatMap(f)` == `f(x)`

- Composition à droite par construct : `m.flatMap(v -> construct(v))` == `m`
- Associativité : `m.flatMap(f).flatMap(g)` == `m.flatMap(x -> f(x).flatMap(g))`

Type monadique (composition)

L'opérateur `flatMap()` :

```
<U> Optional<U> flatMap(Function<T, Optional<U>> mapper)
```

```
Optional.of(1).flatMap(i -> Optional.of("V" + i));   // Optional.of("V1")
Optional.of(1).flatMap(i -> Optional.empty());       // Optional.empty
Optional.empty().flatMap(i -> Optional.of("V" + i)); // Optional.empty
Optional.empty().flatMap(i -> Optional.empty());     // Optional.empty
```

Remarque : Si l'on avait utilisé `map(...)` à la place de `flatMap(...)` :

```
Optional.of(1).map(i -> Optional.of("V" + i));
// Optional.of(Optional.of("V1"))
```

Dans le cas de l'utilisation de `map(...)` on obtient une imbrication de **deux** `Optional` (`Optional<Optional<T>>`) au lieu d'avoir qu'un seul `Optional<T>`.

Type monadique (exemple: Optional)

- Rappel des 3 Lois :
  - Composition à gauche par construct : `construct(x).flatMap(f)` == `f(x)`
  - Composition à droite par construct : `m.flatMap(v -> construct(v))` == `m`
  - Associativité : `m.flatMap(f).flatMap(g)` == `m.flatMap(x -> f(x).flatMap(g))`

Les lois appliquées à `Optional` :

```
Function<Integer, Optional<String>> f = i -> Optional.of("f(" + i + ")");
Function<String, Optional<String>>  g = s -> Optional.of("g(" + s + ")");
```

```
Optional.of(1).flatMap(f);
// Optional.of("f(1)")
```

```
Optional.of(1).flatMap(i -> Optional.of(i));
// Optional.of(1)
```

```
Optional.of(1).flatMap(f).flatMap(g);
// Optional.of("g(f(1))")
Optional.of(1).flatMap(x -> f.apply(x).flatMap(g));
// Optional.of("g(f(1))")
// Egalité
```

Optional est une monade qui gère des traitements dans un **contexte d'absence** de valeur.

Type monadique (exemple de composition avec Optional)

Petit exemple :

```
private Optional<Integer> parseInt(String s) {
    try {
        return Optional.of(Integer.parseInt(s));
    } catch (NumberFormatException ex) {
        return Optional.empty();
    }
}
```

```
int defaultValue = 2015;
String arg = getArg();

int intValue = Optional.ofNullable(arg) // Optional<String>
        .flatMap(s -> parseInt(s))      // Optional<Integer>
        .map(i -> 10 * i)               // Optional<Integer>
        .orElse(defaultValue);          // Integer
```

Type monadique (autre exemple de composition avec Optional)

Modèle de données :

```
+------+          +--------+          +-----+
|Person|----?---->|Vehicule|----?---->|Motor|
+------+          +--------+          +-----+
```

Exemple d'utilisation du modèle en **Java 7** dans un style programmation **impératif** :

```
Person person = getXYZ();

if (person != null) {
    Vehicule vehicule = person.getVehicule();

    if (vehicule != null) {
        Motor motor = vehicule.getMotor();

        if (motor != null) {
            motor.start();
        }
    }
}
```

Optional (autre exemple de composition)

En **Java 8**, avec l'utilisation d'`Optional` dans un style programmation **fonctionnelle** :

Nouvelles signatures :

```java
Optional<Person> getXYZ();
Optional<Vehicule> getVehicule();
Optional<Motor> getMotor();
```

Nouvel enchaînement pour éventuellement appeler la méthode `Motor.start()` :

```java
getXYZ()                        // Optional<Person>
  .flatMap(Person::getVehicule) // Optional<Vehicule>
  .flatMap(Vehicule::getMotor)  // Optional<Motor>
  .ifPresent(Motor::start);     // appelle start() si un motor est present.
```

> «Monads are just types with operators that guide you through the happy path.»
> — Erik Meijer (créateur de Rx, contributeur à Haskell).

Optional (versions spécialisées)

> «Dans la vie on nait tous égaux... mais y en a qui sont plus égaux que d'autres.»
> — Coluche.

Il existe des versions **spécialisées** pour les types primitifs :

- double => `OptionalDouble`
- long => `OptionalLong`
- int => `OptionalInt`
- float => nada :-(
- short => nada :-(
- byte => nada :-(

Sommaire

Comparator (ordre naturel)

En Java 8, l'interface `Comparator` fournie des méthodes pour simplifier la creation de `Comparator`s.

La méthode statique `naturalOrder()` retourne un `Comparator` correspondant à **l'ordre naturel** du type `T` implémentant `Comparable`.

```
static <T extends Comparable<T>> Comparator<T> naturalOrder()
```

```
import static java.util.Comparator.naturalOrder;
Comparator<File> fileCmp = naturalOrder();
```

La méthode statique `reverseOrder()` retourne un `Comparator` correspondant à l'ordre **inverse** de **l'ordre naturel** du type `T` implémentant `Comparable`.

```
static <T extends Comparable<T>> Comparator<T> reverseOrder()
```

```
import static java.util.Comparator.reverseOrder;
Comparator<File> fileCmp = reverseOrder();
```

Comparator (création)

La méthode statique `comparing(...)` retourne un `Comparator` selon l'ordre naturel d'une propriété calculée depuis le type `T`.

```
static <T, U extends Comparable<U>> Comparator<T>
        comparing(Function<T, U> keyExtractor)
```

```
import static java.util.Comparator.comparing;

// Ordre des personnes selon l'ordre naturel (croissant) de leur age.
Comparator<Person> cmpPersonByAge = comparing(p -> p.getAge());
```

Une autre méthode `comparing(...)` retourne un `Comparator` selon l'ordre passé en paramètre d'une propriété calculée depuis le type `T`

```
static <T, U> Comparator<T> comparing(
            Function<T, U> keyExtractor,
            Comparator<U> keyComparator)
```

```
// Ordre des personnes selon l'ordre passé en paramètre
// (ici décroissant) de leur age.
Comparator<Person> cmpPersonByAgeDesc =
        comparing(Person::getAge, reverseOrder());
```

Pour un `Comparator` la méthode `reversed()` retourne un comparator dans l'ordre inverse :

```
default Comparator<T> reversed()
```

```
Comparator<Person> cmpPersonByAgeDesc = cmpPersonByAge.reversed();
```

Comparator (chaînage)

Il est aussi possible de chaîner des `Comparator`s avec la méthode `thenComparing(...)`

```
Comparator<T> thenComparing(Comparator<T> other)
```

```
import static java.util.Comparator.comparing;

Comparator<String> byFirstLetter = comparing(s -> s.charAt(0));
Comparator<String> byNumber      = comparing(s -> s.substring(1));
```

```
List<String> l1 = asList("B05", "B01", "A02", "B02", "A01");
sort(l1, byFirstLetter.thenComparing(byNumber));
// A01, A02, B01, B02, B05

List<String> l2 = asList("B05", "B01", "A02", "B02", "A01");
sort(l2, byNumber.thenComparing(byFirstLetter));
// A01, B01, A02, B02, B05
```

Comparator (gestion des nulls)

Il est aussi possible d'indiquer si nous voulons avoir les valeurs `null`s en **premier** ou en **dernier** grâce aux méthodes statiques `nullsFirst(...)` et `nullsLast(...)` définies dans l'interface `Comparator`:

```
static <T> Comparator<T> nullsFirst(Comparator<T> comparator)
static <T> Comparator<T> nullsLast(Comparator<T> comparator)
```

Comparator (Map.Entry)

L'interface `Map.Entry` fournit des méthodes statiques utiles pour comparer les entrées des `Map`s soit par clés soit par valeur.

```
static <K extends Comparable<K>, V> Comparator<Map.Entry<K,V>>
        comparingByKey()

static <K, V> Comparator<Map.Entry<K, V>>
        comparingByKey(Comparator<K> cmp)

static <K, V extends Comparable<V>> Comparator<Map.Entry<K,V>>
        comparingByValue()

static <K, V> Comparator<Map.Entry<K, V>>
        comparingByValue(Comparator<V> cmp)
```

Comparator (exercice)

Voici la classe pour l'exercice sur les `Comparator`s : `ComparatorTest`.

Sommaire

Stream (introduction)

Un stream n'est pas une collection mais plutôt un enchaînement de traitements à effectuer sur un flux de données typées.

Analogie avec la ligne de commande Linux : C'est aux `Collection`s Java ce que sont pour les fichiers les enchaînements de commandes en **shell** :

```
ls -l /projects | tail -n +2 | sed 's/\s\s*/ /g'
        | cut -d ' ' -f 3 | sort | uniq -c > list.txt
```

Un stream n'est **utilisable q'une seule fois !**

- On peut interagir avec un `Stream` grâce à deux types d'operateurs :
  - opérateurs **intermédiaires** (avec un évaluation paresseuse)
  - opérateurs **terminaux** (un seul par stream)

La définition d'une chaîne de traitement pour un `Stream` respecte la grammaire suivante :

`uneSource` `.opérateurIntermédiaire*` `.opérateurTerminal`

Certains opérateurs permettent de mettre un terme au traitement du stream, ce sont des opérateurs **court-circuit**.

Le traitement d'un `Stream` ne commence que lorsque l'opérateur terminal est appelé.

Stream (versions spécialisées)

> «Dans la vie on nait tous égaux... mais y en a qui sont plus égaux que d'autres.»
> — Coluche.

Il existe des versions **spécialisées** pour les types primitifs :

- double => `DoubleStream`
- long => `LongStream`
- int => `IntStream`
- float => nada :-(
- short => nada :-(
- byte => nada :-(

Stream (creation)

Creation d'un `Stream` :

Dans la classe `java.util.stream.Stream`

```
Stream<String> strings = Stream.empty();
```

```
static<T> Stream<T> of(T... values)
static<T> Stream<T> of(T    value)
```

```
Stream<String> strings = Stream.of("Paris", "Londres", "Madrid");
```

Dans l'interface `java.util.Collection<T>`

```
default Stream<T> stream()
```

```
List<String>   stringList = Arrays.asList("Paris", "Londres", "Madrid");
Stream<String> strings    = stringList.stream()
```

Stream (creation à partir d'un tableau)

Création d'un `Stream` à partir d'un tableau, grâce aux méthodes statiques dans `java.util.Arrays`

```
static <T> Stream<T> stream(T[] array)
```

```
String[] stringArray = new String[]
        {"Paris", "Londres", "Madrid", "Berlin"};
Stream<String> strings = Arrays.stream(stringArray);
```

Création d'un `Stream` à partir d'un sous-ensemble d'un tableau.

```
static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)
```

```
Stream<String> strings = Arrays.stream(stringArray, 1, 3);
// "Londres", "Madrid"
```

Il existe aussi des méthodes `Stream(...)` pour créer des `Stream` s à partir de tableaux d' `int` s, `long` s et `double` s.

```
static IntStream stream(int[] array)
```

```
static IntStream stream(int[] array, int startInclusive, int endExclusive)
```

Stream (creation de Stream infini)

Les `Stream` s peuvent être infinis.

```
static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

```
Stream<Integer> ints = Stream.iterate(10, i -> i + 2);
// 10, 12, 14, etc.
```

```
static<T> Stream<T> generate(Supplier<T> s)
```

```
Stream<Long> times = Stream.generate(() -> System.currentTimeMillis());
```

La classe `java.util.Random` fournit plusieurs méthodes permettant de créer des `Stream` s infinis de valeurs aléatoires.

```
DoubleStream doubles()
```

Stream (creation à partir d'un Iterator)

Fonction utilitaire permettant la création d'un `Stream` à partir d'un `Iterator` :

```
static <T> Stream<T> iteratorToStream(Iterator<T> iterator) {
    Iterable<T> iterable = () -> iterator;
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

Ou construire le `Stream` via un builder :

```
Stream.builder()
    .add(1)
    .add(2)
    .add(3)
    .build()
```

La méthode statique `concat(...)` permet de **concaténer** 2 `Stream` s :

```
static <T> Stream<T> concat(Stream<T> a, Stream<T> b)
```

```
Stream.concat(Stream.of(1, 2, 3), Stream.of(11, 12, 13));
// 1, 2, 3, 11, 12, 13
```

Stream (Operateurs intermédiaires)

Un opérateur intermédiaire retourne **toujours** un `Stream`.

L'opérateur `distinct()` élimine les doublons :

```
Stream<T> distinct()
```

```
Stream.of(1, 2, 3, 2, 4).distinct(); // 1, 2, 3, 4
```

L'opérateur `filter()` élimine les valeurs du `Stream` qui ne correspondent pas au `Predicate` passé en paramètre :

```
Stream<T> filter(Predicate<T> predicate)
```

```
Stream.of(1, 2, 3).filter(i -> i % 2 != 0); // 1, 3
```

L'opérateur `limit(...)` limite la longueur du `Stream` (c'est un opérateur **court-circuit**) :

```
Stream<T> limit(long maxSize)
```

```
Stream.of("a", "b", "c", "d", "e", "f").limit(3);
// "a", "b", "c"
```

Stream (Operateurs intermédiaires)

L'opérateur `skip(long)` permet de sauter les premières valeurs du `Stream` :

```
Stream<T> skip(long n)
```

```
Stream.of("a", "b", "c", "d", "e", "f").skip(2); // "c", "d", "e", "f"
```

L'opérateur `sorted(...)` permet de trier les valeurs du `Stream` selon le `Comparator` passé en paramètre.

L'opérateur `sorted()` sans `Comparator` utilise l'ordre naturel si les éléments implémentent l'interface `Comparable` sinon une `ClassCastException` est levée :

```
Stream<T> sorted(Comparator<T> comparator)
Stream<T> sorted()
```

```
Stream.of(10, 3, 1, 2).sorted(); // 1, 2, 3, 10
```

Stream (Operateurs intermédiaires)

L'opérateur `peek(...)` permet de générer un **effet de bord** pour chaque valeur du `Stream`.

Il peut être inséré au milieu de la définition d'un `Stream` :

```
Stream<T> peek(Consumer<T> action)
```

```
Stream.of(10, 3, 1, 2)
      .peek(i -> System.out.println("Valeur " + i))
      // effet de bord avec 10, 3, 1, 2.
      .sorted(); // 1, 2, 3, 10
```

Stream (opérateur intermédiaire d'application)

L'opérateur `map()` retourne un nouveau `Stream` correspondant à l'application de la fonction passée en paramètre à toutes les valeurs du `Stream` d'origine :

```
<R> Stream<R> map(Function<T,R> mapper)
```

```
Stream.of(1, 2, 3).map(i -> "V-" + i));
// "V-1", "V-2", "V-3"
```

Opérateurs de conversion vers les versions spécialisées des `Stream`s :

```
DoubleStream  mapToDouble(ToDoubleFunction<T> mapper)
IntStream     mapToInt   (ToIntFunction<T>    mapper)
LongStream    mapToLong  (ToLongFunction<T>   mapper)
```

```
// (("Robert", 45), ("Pierre", 30), ("Marie", 25))
IntStream ages = persons.stream().mapToInt(p -> p.getAge());
// 45, 30, 25
```

Stream (opérateur intermédiaire de composition)

Un `Stream<T>` est comme `Optional<T>` c'est aussi un type monadique.

Ainsi `Stream` permet la composition dans un contexte **d'indéterminisme**, c'est à dire : zéro, une ou plusieurs valeurs.

```
<R> Stream<R> flatMap(Function<T, Stream<R>> mapper)
```

```
Stream.of(1, 2, 3)
      .flatMap(i -> Stream.of("A" + i, "B" + i));
      // "A1", "B1", "A2", "B2", "A3", "B3"
```

Opérateurs de conversion vers les versions spécialisées des `Stream`s :

```
DoubleStream  flatMapToDouble(Function<T, DoubleStream> mapper)
IntStream     flatMapToInt   (Function<T, IntStream>    mapper)
LongStream    flatMapToLong  (Function<T, LongStream>   mapper)
```

Stream (Operateurs terminaux)

Un opérateur terminal ne retourne **jamais** un `Stream`, mais soit une valeur soit aucune valeur (`void`).

Un opérateur terminal consomme le `Stream`, c'est-à-dire que la chaîne de traitement commence à être éxécutée.

Stream (Operateurs terminaux)

L'opérateur `allMatch(...)` vérifie que **toutes** les valeurs du `Stream` valident le prédicat passé en paramètre :

```
boolean allMatch(Predicate<T> predicate)
```

```
Stream.of(1, 2, 3).allMatch(i -> i % 2 == 0); // false
Stream.of(1, 2, 3).allMatch(i -> i < 10);     // true
```

L'opérateur `anyMatch(...)` vérifie qu'**au moins une** des valeurs du `Stream` valide le prédicat passé en paramètre :

```
boolean anyMatch(Predicate<T> predicate)
```

```
Stream.of(1, 2, 3).anyMatch(i -> i % 2 == 0); // true
```

L'opérateur `noneMatch(...)` vérifie qu'**aucune** des valeurs du `Stream` ne valide le prédicat passé en paramètre :

```
boolean noneMatch(Predicate<T> predicate)
```

```
Stream.of(1, 2, 3).noneMatch(i -> i % 2 == 0); // false
Stream.of(1, 2, 3).noneMatch(i -> i > 10);     // true
```

Stream (Operateurs terminaux)

L'opérateur `count()` retourne la longueur du `Stream` :

```
long count()
```

```
Stream.of("a", "b", "c").count(); // 3
```

L'opérateur `findFirst()` retourne **le premier** élément du `Stream` si ce dernier n'est pas vide.

C'est un opérateur court-circuit, c'est à dire qu'il est peut **terminer** l'évaluation du `Stream` :

```
Optional<T> findFirst()
```

```
Stream.of(1, 2, 3).findFirst(); // Optional.of(1)
Stream.empty().findFirst();     // Optional.empty
```

L'opérateur `findAny()` retourne **un** élément du `Stream` si ce dernier n'est pas vide.

Cet opérateur peut être utile pour optimiser les traitements de `Stream` **parallèle** :

```
Optional<T> findAny()
```

```
Stream.of(1, 2, 3).findAny(); // peut-être Optional.of(2)
Stream.empty().findAny();     // Optional.empty
```

Stream (Operateurs terminaux)

L'opérateur `forEach(...)` permet de générer un effet de bord pour chaque valeur du `Stream`.

Contrairement à `peek(...)` cet opérateur ne peut pas être au milieu du `Stream` :

```
void forEach(Consumer<T> action)
```

```
Stream.of(1, 2, 3)
      .forEach(i -> System.out.println("Valeur " + i))
      // effet de bord avec 1, 2, 3.
```

Stream (Operateurs terminaux)

Les opérateurs `min(...)` et `max(...)` retournent l'élément maximum/minimum en fonction du `Comparator` passé en paramètre :

```
Optional<T> max(Comparator<T> comparator)
Optional<T> min(Comparator<T> comparator)
```

```
// (Person("Robert", 45), Person("Pierre", 30), Person("Marie", 25))
persons.stream()
       .max(Comparator.comparing(Person::getAge));
       // Optional.of(Person("Robert", 45))

persons.stream()
       .filter(p -> p.getAge() > 100)
       .max(Comparator.comparing(Person::getAge));
       // Optional.empty
```

Stream (Operateurs terminaux)

L'opérateur `toArray()` retourne un tableau d'`Object`s contenant les valeurs du `Stream` :

```
Object[] toArray()
```

```
Object[] valeurs = Stream.of(1, 2, 3).toArray();
```

L'opérateur `toArray(...)` retourne un tableau typé contenant les valeurs du `Stream`, ce tableau est créé par la fabrique passée en paramètre :

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
Integer[] valeurs1= Stream.of(1, 2, 3).toArray(len -> new Integer[len]);

// Plus lisible en utlisant la référence sur le constructeur du tableau.
Integer[] valeurs2 = Stream.of(1, 2, 3).toArray(Integer[]::new);
```

Stream (Operateurs terminaux de réduction)

l'opérateur `reduce(...)` réduit dans un `Optional` le `Stream` en utilisant le `BinaryOperator` passé en paramètre.

Si le `Stream` est vide le résultat est `Optional.empty` :

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

```
Stream.of(1, 2, 3).reduce((a, b) -> a + b);
// Optional.of(1 + 2 + 3)

Stream.of(1).reduce((a, b) -> a + b);
// Optional.of(1)

Stream.of(1, 2, 3)
      .filter(i -> i > 10)
      .reduce((a, b) -> a + b);
// Optional.empty
```

Stream (Opérateurs terminaux de réduction)

Cette version de l'opérateur `reduce(...)` réduit le `Stream` en utilisant le `BinaryOperator` passé en paramètre et en initiant la réduction avec la valeur passée en paramètre (`identity`), il y a forcément un résultat (pas de `Optional` ici) :

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

```
Stream.of(1, 2, 3).reduce(10, (a, b) -> a + b);
// 16 = 10 + 1 + 2 + 3

Stream.of(1, 2, 3)
      .filter(i -> i > 10)
      .reduce(10, (a, b) -> a + b);

// 10
```

Cette réduction retourne un résultat équivalent à ce code impératif :

```
T result = identity;
for (T element : this stream) {
     result = accumulator.apply(result, element)
}
return result;
```

Stream (opérateurs spécifiques pour version spécialisées)

Ci-dessous la liste des opérateurs spécifiques pour `IntStream`.

Creation d'intervalle exclusif et inclusif :

```
static IntStream range(int startInclusive, int endExclusive)
```

```
IntStream.range(1, 4);
// 1, 2, 3

static IntStream rangeClosed(int startInclusive, int endInclusive)
IntStream.rangeClosed(1, 4);
// 1, 2, 3, 4
```

Conversion vers d'autres `Stream`s **spécialisés** :

```
DoubleStream    asDoubleStream()
LongStream      asLongStream()
```

Conversion vers un `Stream` **générique** :

```
Stream<Integer> boxed()
<U> Stream<U>   mapToObj(IntFunction<U> mapper)
```

Stream (opérateurs spécifiques pour version spécialisées)

Opérateur de composition :

```
IntStream flatMap(IntFunction<IntStream> mapper)
```

```
IntStream.of(1, 2, 3)
         .flatMap(i -> IntStream.of(i, 10 * i)); // 1, 10, 2, 20, 3, 30
```

Stream (opérateurs spécifiques pour version spécialisées)

Opérateurs de réduction `reduce`

```
OptionalInt reduce(IntBinaryOperator op)
```

```
IntStream.of(1, 2, 3).reduce((a, b) -> a + b));
// 1 + 2 + 3

IntStream.of(1, 2, 3)
         .filter(i -> i > 10)
         .reduce((a, b) -> a + b));
// OptionalInt.empty
```

Dans cette version le paramètre `identity` est l'initialisation de la réduction.

```
int reduce(int identity, IntBinaryOperator op);
```

```
IntStream.of(1, 2, 3).reduce(10, (a, b) -> a + b);
// 10 + 1 + 2 + 3
IntStream.of(1, 2, 3)
        .filter(i -> i > 10)
        .reduce(10, (a, b) -> a + b);
// 10
```

Stream (opérateurs spécifiques pour version spécialisées)

Calculs statistiques sur les valeurs de l'`IntStream` :

```
OptionalDouble average()
```

```
IntStream.of(1, 3).average(); // OptionalDouble.of(2.0)
```

```
OptionalInt max()
OptionalInt min()
```

```
IntStream.of(1, 2, 3).filter(i -> i > 10).min(); // OptionalInt.empty
```

```
int sum()
int[] toArray()
```

L'opérateur `summaryStatistics()` calcule en **une fois** : le nombre d'éléments, le min, le max, la moyenne et la somme.

```
IntSummaryStatistics summaryStatistics()
```

Stream (Opérateurs terminaux de réduction vers un autre type)

Cette version de l'opérateur `reduce(...)` permet de faire une réduction vers un autre type (ici `U`).

Le dernier paramètre n'est utilisé que pour les `Stream`s parallèles, bien qu'inutile pour un `Stream` séquentiel, il ne doit pas être null :

```
<U> U reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)
```

```
Stream.of(1, 2, 3)
      .reduce("init", (acc, v) -> acc + ", " + v, (a, b) -> "");
// "init, 1, 2, 3"
```

Cette réduction retourne un résultat équivalent à ce code impératif :

```
U result = identity;
for (T element : this stream) {
     result = accumulator.apply(result, element)
}
return result;
```

Stream (Opérateurs terminaux de réduction mutable)

L'opération `collect(...)` est assez proche de `reduce(...)` à la différence près que l'accumulation se fait dans une structure mutable.

```
<R> R collect(Supplier<R> supplier,
        BiConsumer<R,T> accumulator, BiConsumer<R,R> combiner)
```

Cette réduction retourne un résultat équivalent à ce code impératif :

```
R result = supplier.get();
for (T element : this stream) {
     accumulator.accept(result, element);
}
return result;
```

Contrairement à `reduce(...)`, dans la boucle `for` ci-dessus il n'y a pas de réaffectation, mais la **mutation** de `result`.

Stream (analogie avec SQL)

L'écriture d'enchaînement de `Stream` requiert une tournure d'esprit assez proche de celle qu'il faut pour écrire du `SQL`.

```
persons.stream() //
        .filter(p -> p.getAge() >= 18) //
        .sorted(comparing(p -> p.getHeight())) //
        .map(p -> p.getName() + "(" + p.getHeight() + ")");
```

Evidemment c'est dans le même ordre, mais c'est un peu le **même esprit** :

```sql
SELECT p.name, p.height
FROM persons
WHERE p.age >= 18
ORDER BY p.height
```

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8
  - Lambda **✔**
  - Interfaces fonctionnelles prédéfinies **✔**
  - Référence de méthode **✔**
  - Programmation fonctionnelle **✔**
  - Optional **✔**
  - Types monadiques **✔**
  - Comparator **✔**
  - Stream **✔**
  - Collectors
  - Monoïdes
  - Stream parallèle
  - Autres ajouts de Java 8

Collector (introduction)

Un `Collector` est constitué des 4 fonctions suivantes :

- la **création** du nouveau conteneur de résultat (`supplier()`)
- l'**accumulation** de nouvelles données dans le conteneur de résultat (`accumulator()`)
- la **fusion** de deux conteneurs de résultats en un seul (`combiner()`), utilisé pour les `Stream`s parallèles
- la **finalisation** du résultat (`finisher()`)

Les types du `Collector<T, A, R>` :

- `T` : le type des valeurs d'**entrée**.
- `A` : le type de la structure de données **intermédiaires** d'accumulation.
- `R` : le type du **résultat** de la réduction.

Réduire un `Stream` en utilisant un `Collector` doit retourner un résultat équivalent au code ci-dessous :

```
R container = collector.supplier().get();
for (T t : data) {
     collector.accumulator().accept(container, t);
}
return collector.finisher().apply(container);
```

Collector (création)

L'interface `java.util.stream.Collector` contient une méthode statique pour créer un `Collector` :

```
static <T, A, R> Collector<T, A, R> of(Supplier<A>        supplier,
                                       BiConsumer<A, T>   accumulator,
                                       BinaryOperator<A>  combiner,
                                       Function<A, R>     finisher,
                                       Characteristics... characteristics)
```

Les valeurs de l'`enum` `Characteristics` :

- `CONCURRENT` : indique que le `Collector` peut réduire des valeurs d'entrée qui proviennent de **plusieurs** `Thread`s.
- `UNORDERED` : indique que le `Collector` ne tient pas compte de l'**ordre** des valeurs d'entrée.
- `IDENTITY_FINISH` : indique que le `Collector` utilise la fonction `Function.identity()` (`x -> x`) comme finisher.

Une autre méthode `of(...)` utilise **d'office** la fonction `Function.identity()` (`x -> x`) comme finisher :

```
static <T, R> Collector<T, R, R> of(Supplier<R>        supplier,
                                    BiConsumer<R, T>   accumulator,
                                    BinaryOperator<R>  combiner,
                                    Characteristics... characteristics)
```

Collector (opérateur collect)

Un `Collector` est typé, il ne s'applique que sur un `Stream` de **type compatible**.

L'opérateur `collect(...)` permet une réduction **mutable** en utilisant les fonctions définies dans le `Collector` :

```
<R,A> R collect(Collector<T,A,R> collector)
```

Collectors (une classe utilitaire)

La classe `Collectors` est une collection de méthodes statiques permettant de créer des `Collector`s.

Le collector `joining(...)` permet de joindre un `Stream` de `String` en une seule `String` :

```
static Collector<CharSequence, ?, String> joining(CharSequence delimiter,
                                                  CharSequence prefix,
                                                  CharSequence suffix)

static Collector<CharSequence, ?, String> joining(CharSequence delimiter)

static Collector<CharSequence, ?, String> joining()
```

```
Stream.of(1, 2, 3).collect(joining(" - "));
// /!\ NE COMPILE PAS /!\
```

```
Stream.of(1, 2, 3).map(String::valueOf).collect(joining(" - "));
// "1 - 2 - 3"

Stream.of(1, 2, 3).map(String::valueOf)
        .collect(joining(" - ", "-->", "<--"));
// "-->1 - 2 - 3<--"
```

Collectors (vers des Collections)

Les collectors `toList()`, `toSet()` et `toCollection()` accumulent les valeurs d'entrée respectivement dans une `List`, `Set` et une collection créée par la fabrique passée en paramètre :

```
static <T> Collector<T, ?, List<T>> toList()

static <T> Collector<T, ?, Set<T>> toSet()

static <T, C extends Collection<T>> Collector<T, ?, C>
        toCollection(Supplier<C> factory)
```

Collectors (vers une Map)

Le collector `toMap()` accumule les valeurs d'entrée dans une `Map`.

```
static <T, K, U, M extends Map<K, U>> Collector<T, ?, M>
       toMap(Function<T, K>    keyMapper,
             Function<T, U>    valueMapper,
             BinaryOperator<U> mergeFunction,
             Supplier<M>       mapSupplier)
```

- `keyMapper` et `valueMapper` permettent respectivement d'extraire la **clé** et la **valeur** pour créer une `Map.Entry` à partir d'une valeur d'entrée du `Collector`.
- `mergeFunction` permet de résoudre des collisions entre des valeurs d'entrée ayant la même clé.
- `mapSupplier` est une fabrique permettant de créer la `Map` qui accumule les valeurs d'entrée.

Il est parfois pratique de créer un `Tuple<A,B>` pour ensuite créer une `Map` :

```
unStreamDeTuples.collect(toMap(Tuple::getA, Tuple::getB));
```

Collectors (collectors correspondants à des opérateurs de Stream)

- Le collector `counting()` est le pendant de `count()` de `Stream`.
- Les 3 collectors `reducing(...)` sont les pendants des `reduce(...)` de `Stream`.
- Les collectors `minBy(...)` et `maxBy(...)` sont les pendants de `min(...)` et `max(...)` de `Stream`.
- Le collector `summingInt(...)` est l'équivalent de `mapToInt(...).sum()` pour `IntStream`, etc. :

```
// (("Robert", 45), ("Pierre", 30), ("Marie", 25))
persons.stream().mapToInt(p -> p.getAge()).sum();      // 100
persons.stream().collect(summingInt(p -> p.getAge())); // 100
```

- Le collector `summarizingInt(...)` est l'équivalent de `mapToInt(...).summaryStatistics()` pour `IntStream`, etc. :
- Le collector `averagingInt(...)` est l'équivalent de `mapToInt(...).average()` pour `IntStream`, etc. :

Collectors (forme générale de regroupement)

Le collector `groupingBy(...)` permet de grouper les valeurs d'entrée :

```
static <T, K, D, A, M extends Map<K, D>> Collector<T, ?, M>
       groupingBy(Function<T, K>     classifier,
                  Supplier<M>        mapFactory,
                  Collector<T, A, D> downstream)
```

- la `Function` `classifier` permet d'extraire la clé du groupe.
- le `Supplier` `mapFactory` est une fabrique permettant la création de la `Map` résultant du collector.
- le `Collector` `downstream` est le collector permettant de créer le groupe.

Version simplifiée sans `mapFact` qui retourne une `HashMap` :

```
static <T, K, D, A, M extends Map<K, D>> Collector<T, ?, M>
       groupingBy(Function<T, K>     classifier,
                  Collector<T, A, D> downstream)
```

Collectors (regroupement simplifié)

Version encore plus simplifiée qui retourne une `HashMap` de `List`s :

```
static <T, K, D, A, M extends Map<K, D>> Collector<T, ?, M>
       groupingBy(Function<T, K> classifier)
```

```
Stream.of("Paris", "Berlin", "Barcelone")
      .collect(groupingBy(s -> s.charAt(0)));
// {'P'=["Paris"], 'B'=["Berlin", "Barcelone"]}
```

En utilisant un `Collector` pour créer la valeur du groupe (ici `counting()` pour le comptage) :

```
Stream.of("Paris", "Berlin", "Barcelone")
      .collect(groupingBy(s -> s.charAt(0), counting()));
// {'P'=1, 'B'=2}
```

Collectors (partitionnement)

Le collector `partitioningBy(...)` est une version spécialisée de `groupingBy(...)` ou la `Function` `classifier` est remplacée par un `Predicate`.

Le résultat de ce `Collector` est une `Map` ayant toujours 2 `Map.Entry` avec les clés `true` & `false` pour chacun des groupes qui valident ou non le `Predicate`.

```
static <T, D, A> Collector<T, ?, Map<Boolean, D>>
       partitioningBy(Predicate<T>       predicate,
                      Collector<T, A, D> downstream)
```

Version encore plus simplifiée qui retourne une `HashMap` de `List`s :

```
static <T, D, A> Collector<T, ?, Map<Boolean, D>>
       partitioningBy(Predicate<T> predicate)
```

```
Stream.of("Paris", "Berlin", "Barcelone")
      .collect(partitioningBy(s -> s.length() > 6));
// {false=["Paris", "Berlin"], true=["Barcelone"]}
```

Stream (exercice)

Voici le modèle que l'on va utiliser pour les exercices :

```
+----------+        +-------+        +-------+
|  Person  |---?--->| House |---?--->|Garden |
|----------|        +-------+        |-------|
|name      |            |            |surface|
|age       |            |            +-------+
|department|            |
+----------+            |            +-----------+        +--------------+
                        +-------*--->|   Room    |---*--->|     Bed      |
                                     |-----------|        |--------------|
                                     |windowCount|        |forPersonCount|
                                     +-----------+        +--------------+
```

Sommaire

- Exemple très simple en Java 7 **✔**
- Limitations de Java 7 **✔**
- Java 8
  - Lambda **✔**
  - Interfaces fonctionnelles prédéfinies **✔**
  - Référence de méthode **✔**
  - Programmation fonctionnelle **✔**
  - Optional **✔**
  - Types monadiques **✔**
  - Comparator **✔**
  - Stream **✔**

- Collectors **✔**
- Monoïdes
- Stream parallèle
- Autres ajouts de Java 8

Semi-groupe et Monoïde

- Un semi-groupe est une structure algébrique consistant en un ensemble muni d'une loi de composition **interne associative** : `(T, BinaryOperator<T> op)`.
- Loi 1 : t1 `op` t2 `op` t3 `op` t4
  == ((t1 `op` t2) `op` t3) `op` t4 `// traitement incrémental`
  == t1 `op` t(2 `op` t(3 `op` t4))
  == (t1 `op` t2) `op` t(3 `op` t4) `// traitement parallèle`
- Un monoïde est un semi-groupe ayant **un élément neutre** : `(T, BinaryOperator<T> op, T neutre)`.
- Loi 2 : unT `op` neutre = neutre `op` unT

Exemples de Monoïdes

Ensemble	Operation	Elément neutre
Integer	+	0
Integer	*	1
String	+	""
Boolean	and	true
Boolean	or	false
List	append	[]
Set	append	[]
Map	union (avec un semi-groupe pour les valeurs)	{:}

Monoïde pour calculer une moyenne ?

Imaginons que nous ayons une classe de **22 élèves** :

- **20 élèves** ont une **note** de **1**
- **2 élèves** ont une **note** de **19**

Calculons la **moyenne** parallèlement :

- un calcul pour les **mauvaises notes** : moyenne de **1**, (= (1+1+1+...+1+1) / 20)
- un calcul pour les **bonnes notes** : moyenne de **19**, (= (19 + 19) / 2)

Faisons la **moyenne** de la **classe** : **10** (car (1 + 19) / 2)

Evidemment on ne peut **pas** faire **la moyenne des moyennes**.

Car l'opération de calcul de moyenne ne forme **pas un monoïde**.

Monoïde pour calculer une moyenne !

Par contre avec le **monoïde** suivant :

- valeur : Le couple formé pas la **somme** des notes et le **nombre** de notes : `(sum, count)`
- opération : La somme des sommes des notes, et la somme des nombres de notes : `(+, +)`
- élément neutre : `(0, 0)`

Calculons la **moyenne** parallèlement :

- un calcul pour les **mauvaises notes** : `(20, 20)`
- un calcul pour les **bonnes notes** : `(38, 2)`

Faisons la **moyenne** de la **classe** : **2.636** (car `(20 + 38, 20 + 2)`, donc 58 / 22)

Sommaire

Stream parallèle (introduction)

Les `Stream`s parallèles utilisent en interne le mécanisme de **parallélisation** `ForkJoin` introduit en **Java7**.

Pour connaitre le nombre de `Thread`s utilisés :

```
1 ForkJoinPool commonPool = ForkJoinPool.commonPool();
2 System.out.println("Nombre de Threads du pool: " +
3         commonPool.getParallelism());
```

Stream parallèle (création)

Pour créer un `Stream` parallèle on peut utiliser l'opérateur `parallel()` :

```
Stream.of(1, 2, 3).parallel();
```

Ou utiliser `parallelStream()` sur une `Collection` :

```
List<String> list = Arrays.asList("a", "b", "c");

list.parallelStream();
```

Stream parallèle (consommation)

Pour consommer un `Stream` parallèle il existe deux méthodes.

La méthode `forEach(...)` consomme le stream parallèle dans un **ordre** qui peut être **différent** de celui de la source :

```
void forEach(Consumer<T> action)
```

La méthode `forEachOrdered(...)` consomme le stream parallèle dans l'**ordre** de la source (plus coûteux) :

```
void forEachOrdered(Consumer<T> action)
```

Par ailleurs si la source d'un `Stream` ne représente pas une liste ordonnée de données alors on peut utiliser l'opérateur `unordered()` pour permettre certaines optimisations des `Stream`s parallèles.

Stream parallèle (gain ?)

Traiter un `Stream` en parallèle n'est pas forcément plus rapide qu'en séquentiel.

Si le traitement pour chaque valeur du `Stream` est **long**, alors la parallélisation apporte un gain :

Traitement séquentiel :

```
|-----------||-----------||-----------||-----------||-----------|
```

Traitement parallèle :

```
|-----------||-----------|
   |-----------||-----------|
      |-----------|
```

Par contre si le traitement pour chaque valeur du `Stream` est **très léger** alors le coût du mécanisme de parallélisation a un impact significatif sur la durée totale du traitement :

```
|||||
```

```
||
   ||
      |
```

Stream parallèle (ForkJoinPool spécifique)

Tout les Stream`s parallèles partagent le **même** ForkJoinPool`.

Cependant il est possible d'utiliser un `ForkJoinPool` spécifique pour ne **pas perturber** les autres `Stream`s parallèles :

```
ForkJoinPool myForkJoinPool = new ForkJoinPool(2);

Integer sum = myForkJoinPool
              .submit(() -> range(1, 1_000_000)
                            .parallel()
                            .filter(i -> i % 2 == 0)
                            .sum()
                     )
              .get();
```

Stream parallèle (contraintes pour une bonne réduction)

Rappel de la signature de `reduce(...)` :

```
<U> U reduce(U                  identity,
             BiFunction<U, T, U> accumulator,
             BinaryOperator<U>   combiner)
```

Pour faire une réduction avec un `Stream` parallèle il faut obligatoirement utiliser un **monoïde** pour le type `U` :

- `identity` doit être l'élément **neutre** du monoïde.
- `accumulator` doit être de la forme : (acc, v) -> acc `operation` f(v).
- `combiner` doit être l'`opération` du monoïde.

Stream parallèle (exemple d'une mauvaise réduction)

Voici un exemple de réduction qui **ne respecte pas** les règles du **monoïde** :

Réduction **séquentielle** du `Stream` :

```
Stream.of("a", "b", "c").reduce("-debut-",
                                (acc, v) -> acc + ", " + v,
                                (a, b) -> a + " combine " + b));
// "-debut-, a, b, c"
```

Réduction **parallèle** du `Stream` :

```
Stream.of("a", "b", "c").parallel().reduce("-debut-",
                                           (acc, v) -> acc + ", " + v,
                                           (a, b) -> a + " combine " + b));
// "-debut-, a combine -debut-, b combine -debut-, c"
```

Stream parallèle (réduction concurrente)

Lorsque l'on utilise un `Stream` parallèle il est plus **performant** d'utiliser respectivement `groupingByConcurrent(...)` et `toConcurrentMap(...)` au lieu de `groupingBy(...)` et `toMap(...)`.

Ces deux méthodes retournent une `ConcurrentMap` au lieu d'une simple `Map` :

```
static <T, K, A, D, M extends ConcurrentMap<K, D>>
       Collector<T, ?, M> groupingByConcurrent(Function<T, K>     classifier,
                                               Supplier<M>        mapFactory,
                                               Collector<T, A, D> downstream)
```

```
static <T, K, U, M extends ConcurrentMap<K, U>>
       Collector<T, ?, M> toConcurrentMap(Function<T, K>    keyMapper,
                                          Function<T, U>    valueMapper,
                                          BinaryOperator<U> mergeFunction,
                                          Supplier<M>       mapSupplier)
```

Sommaire

Nouvelle API java.time.*

En Java 8, une nouvelle API (package `java.time.*`) pour gérer le temps a été écrite en remplacement de l'ancienne `java.util.Date`.

Cette API est complètement **immutable**.

Bien que différente, elle est dans le **même esprit** que la librairie *open source* Joda-Time.

AtomicRef & Co

En Java 8, des méthodes de mise à jour des valeurs des `AtomicRef<T>` ont été ajoutées.

L'avantage de ces méthodes, c'est qu'elles intègrent les boucles de **retentative** en cas d'échec de l'opération atomique `compareAndSet(...)`.

Pour cette raison les **lambdas** passées en paramètre doivent être **pures (sans effets de bords)**.

```
V getAndUpdate(UnaryOperator<V> updateFunction)
V updateAndGet(UnaryOperator<V> updateFunction)
V getAndAccumulate(V x, BinaryOperator<V> accumulatorFunction)
V accumulateAndGet(V x, BinaryOperator<V> accumulatorFunction)
```

Quelques méthodes utiles sur Map

Méthodes ajoutées dans l'interface `Map` :

La méthode `getOrDefault(...)` est une variante de `get(...)` qui retourne une **valeur par défaut** si aucune valeur ne correspond à la clé passée en paramètre :

```
default V getOrDefault(Object key, V defaultValue)
```

l'opérateur `forEach(...)` permet d'itérer sur toutes les clés/valeurs de la `Map` :

```
default void forEach(BiConsumer<K, V> action)
```

La méthode `merge(...)` permet d'ajouter une valeur dans la `Map` comme `put(...)` mais avec la possibilité de fusionner la valeur avec une éventuelle précédente valeur :

```
default V merge(K key, V value, BiFunction<V, V, V> remappingFunction)
```

Quelques méthodes utiles sur Map (computeIfAbsent)

La méthode `computeIfAbsent(...)` permet de retrouver une valeur associée à une clé comme `get(...)` mais si la valeur n'existe pas, alors une **valeur** est **calculée depuis la clé**, puis ajoutée à la `Map`, puis retournée :

```
default V computeIfAbsent(K key, Function<K, V> mappingFunction)
```

```
Map<Integer, List<String>> citiesByDept = new HashMap<>();

citiesByDept.computeIfAbsent(44, k -> new ArrayList<>()).add("Nantes");
```

Quelques méthodes utiles sur ConcurrentHashMap (recherches)

Dans l'interface `ConcurrentHashMap` :

En Java 8, Il existe 4 fonctions `searchXYZ(...)` qui sont capables d'effectuer une recherche tout en **transformant** éventuellement la valeur trouvée :

```
<U> U search(long parallelismThreshold,
                  BiFunction<K, V, U> searchFunction)

<U> U searchKeys(long parallelismThreshold,
                     Function<K, U> searchFunction)

<U> U searchValues(long parallelismThreshold,
                       Function<V, U> searchFunction)

<U> U searchEntries(long parallelismThreshold,
                    Function<Map.Entry<K,V>, U> searchFunction)
```

La recherche continue **tant que** la `Function` de recherche `searchFunction` retourne **null**.

Si la taille de la `Map` est supérieure au paramètre `parallelismThreshold` alors la recherche s'effectue en parallèle.

Quelques méthodes utiles sur ConcurrentHashMap (réductions)

En Java 8, il existe 19 opérateurs `reduceXYZ(...)`. Si la taille de la `Map` est supérieure au paramètre `parallelismThreshold` alors la réduction s'effectue en parallèle.

```
<U> U reduce(long                parallelismThreshold,
             BiFunction<K, V, U> transformer,
             BiFunction<U, U, U> reducer)
```

LongAdder et DoubleAdder

En Java 8, lorsque vous utilisez un `AtomicLong` ou `AtomicDouble` avec beaucoup de mises à jour faites par **différents** `Thread`s alors il est préférable d'utiliser `LongAdder` ou `DoubleAdder` pour **réduire la contention**.

Exemple de calcul de fréquence de mots :

```
ConcurrentHashMap<String, LongAdder> freqs = new ConcurrentHashMap<>();

words.stream().forEach(w -> freqs.computeIfAbsent(w, k -> new LongAdder()).increment());
```

Voir aussi les versions plus générales d'accumulation avec `LongAccumulator` et `LongAccumulator`.