Stream et Optional

Rémi Forax

java.util.stream.Stream

Abstraction d’un flux d’élements sur lequel on veut faire des calculs

- Ce n’est pas une Collection car un Stream ne contient pas d’élément
- Ce n’est pas un Iterator car un Stream correspond à un calcul complet et pas à une étape du calcul
  - Un Stream à une vision plus globale du traitement

Sources d’un Stream

Comme un Stream ne stocke pas les données, elles proviennent d’une source

- À partir de valeurs
  - Stream.empty(),
    Stream.of(E… element),
    Stream.ofNullable(E element)
- À partir d’une collection
  - collection.stream(), collection.parallelStream()

Sources d’un Stream (2)

À partir d’un fichier

Files.lines(Path path)

À partir d’un intervalle

IntStream.range(int start, int end)

À partir d’un tableau

Arrays.stream(E[] array)

À d’une liste chaînée

Stream.iterate(head, e -> e != null, e -> e.next)

Ré-utilisation d’un Stream

Un Stream permet de faire un calcul, pas plusieurs

```
Stream<String> stream = …
Stream<String> s1 = stream
        .map(String::toLowerCase);
IntStream s2 = stream
        .mapToInt(String::length);  // IllegalStateException
```

Il n’est donc pas possible de réutiliser un Stream, il faut créer plusieurs streams différents sur la même source

Ré-utilisation d’un Stream (2)

On ne peut pas réutiliser un Stream !

=> Eviter de stocker les Streams
dans des variables locales

=> Utiliser le chainage des méthodes

```
list.stream()
  .filter(Predicate.not(String::isEmpty))
  .map(Stream::toLowerCase)
  .collect(Collectors.toList());
```

Parcourt et modifications

Comme le parcourt avec un iterator() ou la méthode forEach(), un Stream ne doit pas faire d'effet de bord sur la source de ses élements

```
List<String> list = …
list.stream()
    .forEach(list::add);   // aaaaaaah
```

Pour les collections non concurrentes,
=> ConcurrentModificationException

L’API des Streams

Basée sur les opérations sans état (*stateless*)

- filter, map/flatMap, reduce

Capable d’arrêter le calcul si résultat est trouvé (*short circuit*)

- limit(), findFirst()/findAny(), takeWhile()

Les opérations intermédiaires modifient le calcul, les opérations terminales lancent le calcul

Opérations intermédiaires (1/2)

Sélectionne si un élement reste dans le Stream

```
Stream<E> filter(Predicate<? super E>)
```

Transforme un élément

```
<R> Stream<R> map(
                    Function<? super E, ? extends R)
```

Tranforme un élement en une série d'élements

```
<R> Stream<R> flatMap(
                    Function<? super E, ? extends Stream<R>>)
```

Saute des éléments, Sélectionne les premiers éléments

```
Stream<E> skip(int length)
Stream<E> limit(int maxSize)
```

Opérations intermédiaires (2/2)

Supprime les doublons

Stream<E> distinct()

Trie les élements

Stream<E> sorted(Comparator<? super E>)

Obtient les éléments au milieu du Stream (pour débugger)

Stream<E> peek(Consumer<? super E>)

Sélectionne/supprime des élements (stop après)

Stream<E> takeWhile(Predicate<? super E>)
Stream<E> dropWhile(Predicate<? super E>)

Opérations Terminales (1/2)

Compte les éléments

long count()

Appel le consumer pour chaque élement

Stream<E> forEach(Consumer<? super E>)
Stream<E> forEachOrdered(Consumer<? super E>)

Vrai si tout les/au moins un élement(s) vérifie le prédicat

allMatch(Predicate<? super E>)
anyMatch(Predicate<? super E>)

Opérations Terminales (2/2)

Trouve le/un premier élement

Stream<E> findFirst()
Stream<E> findAny()

Créé un tableau

E[] toArray(IntFunction<E[]>)

aggège les données (sans mutation)

reduce(T seed, BinaryOperator<T> reducer)

aggège les données (mutable)

T collect(Collector<E, A,T> collector)

Où est le bug ?

Le code suivant ne compile pas :(

```
public static int sum(int[][] table) {
    return Arrays.stream(table)
                 .flatMap(Arrays::stream)
                 .sum();
 }
```

Le résultat doit être un IntStream

Il faut utiliser flatMapToInt, pas flatMap !

```
public static int sum(int[][] table) {
    return Arrays.stream(table)
                 .flatMapToInt(Arrays::stream)
                 .sum();
 }
```

Des Streams de type primitif

IntStream, LongStream et DoubleStream

- Évite le boxing
- Possède des méthodes spécifiques (sum, average, etc.)

Sur un Stream, il existe plusieurs versions de map(), mapToInt, mapToDouble, mapToLong qui renvoie des Stream de type primitif (même chose pour flatMap)

Un Stream peut être infinie

Afficher les valeurs inférieur à 100 de la suite

- $U_0 = 1$
- $U_n = 2 * U_{n-1} + 1$

```
public static void main(String[] args) {
   IntStream
      .iterate(1, n -> 2 * n + 1)         // stream infini !
      .takeWhile(v -> v < 100)
      .forEach(System.out::println);
}
```

Operations “statefull”

Certaines opérations demande à garder un état pour effectuer le calcul

- distinct()
  - On doit garder les éléments déjà vu dans un Set
- sorted()
  - On doit garder tous les éléments dans une List avant de pouvoir les trier

Ces opérations vont allouer de la mémoire linéairement par rapport au nombre d’élements

Stream parallèle

L’API des Streams est la même que le calcul soit fait séquentiellement ou en parallèle

On obtient un stream parallèle explicitement

- Soit en appelant collection.parallelStream()
- Soit rendant le Stream parallèle Stream.parallel()

Attention, un Stream parallèle ne va pas forcément plus vite qu’un Stream séquentiel

- Le temps de distribution du calcul et de ré-agréggation peut être plus lent que le temps du calcul lui-même

Reduce

Permet de calculer une valeur à partir de l'ensemble des valeurs d'un Stream<T>

- Reduce vers un T

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

```
streamOfInt.reduce(0 , Integer::sum);
```

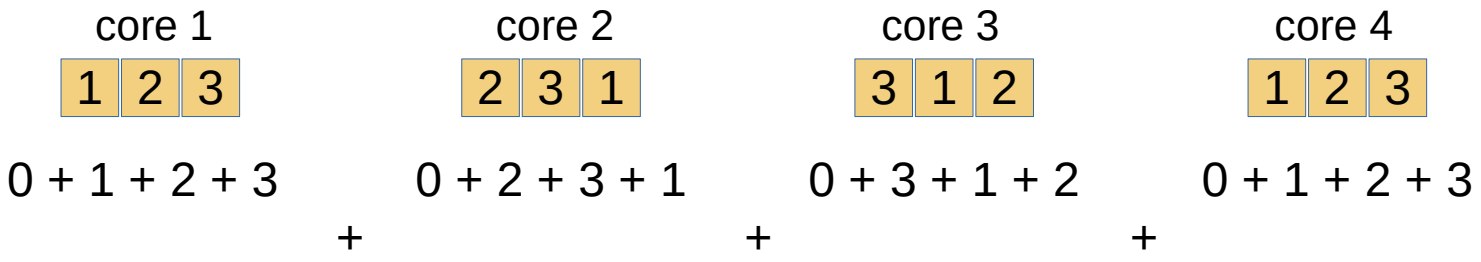
- Reduce vers autre chose qu'un T

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

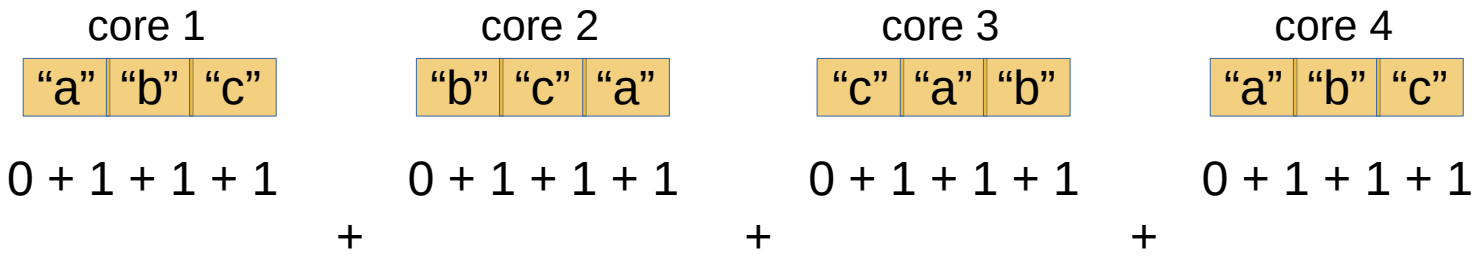
```
streamOfString.reduce(0, (acc, s) - > acc + 1, Integer::sum);
```

Reduce

- streamOfInt.reduce(0 , Integer::sum);



- streamOfStrings.reduce(0,
  (acc, s) - > acc + 1,
  Integer::sum);



Effet de bord

ForEach() + effet de bord avec un Stream parallèle fait n’importe quoi

```
List<String> list = …
ArrayList<String> result = new ArrayList<>();
list.parallelStream()
    .map(String::toLowerCase)
    .forEach(result::add);    // aaaaaaaaaaaaaaaah
```

Effet de bord : Solution !

Le collecteurs savent faire des opérations en parallèle

privilégier la méthode collect() et les Collectors !

```
List<String> list = …
    list.parallelStream()
        .map(String::toLowerCase)
        .collect(Collectors.toList());
```

collect

Permet de renvoyer un objet mutable “contenant” tous les élements du Stream

- équivalent à faire un reduce() mais pour les objets mutables

L’interface java.util.stream.Collector définie ce qu’est un collecteur

- La classe java.util.stream.Collectors contient des Collector prédéfinies

Collector<E, A, T>

Aggrège les données de façon mutable

- Un supplier qui crée un container () → A
- Un accumulateur mutable (A, E) → void
- Un combiner fonctionnel (A, A) → A
- Un finisher (optionnel) (A) → T

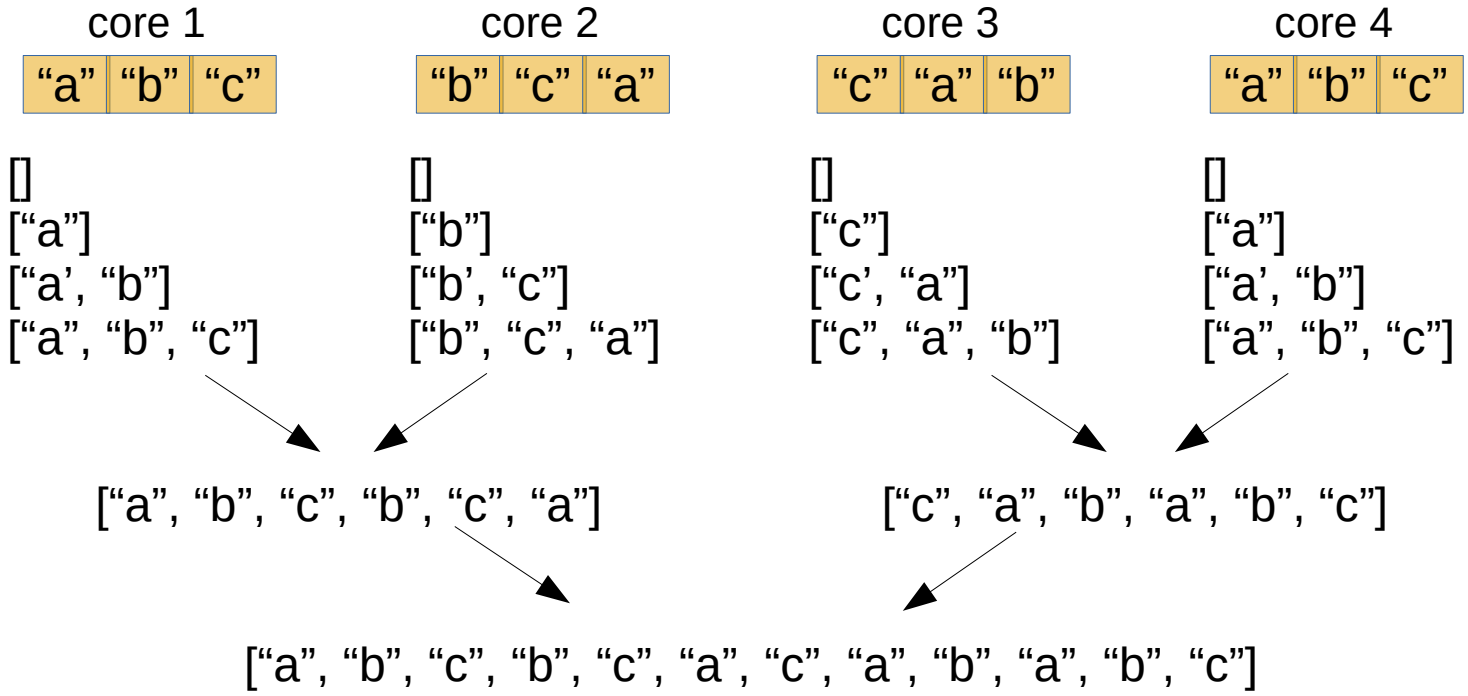
Exemple:

Le Collectors.toList() est équivalent à

```
Collector.of(ArrayList::new,
    ArrayList::add,
   (a1, a2) → { a1.addAll(a2); return a1; })
```

Stream.collect(toList())

streamOfStrings.collect(Collectors.toList());
- supplier: ArrayList::new
- accumulator: ArrayList::add
- combiner: (a1, a2) -> { a1.addAll(a2); return a1; }
- finisher: pas définie donc identity



java.util.stream.Collectors

- joining(separator, prefix, suffix) → String
- toList(), toUnmodifiableList(), toSet(), toUnmodifiableSet()
- toCollection(Supplier<? extends Collection<E>>>) → Collection<E>
- toMap(Function<? super T, ? extends K> keyMapper,
  Function<? Super T, ? extends V> valueMapper) → Map<K,V>
- groupingBy(Function<? super T, ? extends K>) → Map<K,List<V>>
- partition(Predicate<? Super T>) → Map<Boolean,List<T>>

Downstream Collector

Un Collector peut lui même dépendre d'un autre Collector (*downstream collector*)

Par exemple, le *downstream collector* par défaut de groupingBy() est le résultat de toList()

```
List<String> list = List.of("foo", "bar", "boo");
Map<Character, List<String> map =
    list.stream()
        .collect(groupingBy(s - > s.charAt(0)));
```

mais on spécifier un autre collector

```
Map<Character, Long> countMap =
    list.stream()
        .collect(groupingBy(s - > s.charAt(0), counting());
```

Opération pas efficace en parallèle

Paralléliser un calcul est pas efficace

- si le résultat dépend d'un ordre
  - On peut relacher la contrainte d'ordre !
- si il faut stocker des résultats intermédiaires

- distinct() / sorted()

  Statefull, on a un état partagé :(
- limit()

  On peut utiliser unordered() pour indiquer que l'ordre est pas important
- findFirst()

  On utilise findAny() à la place
- forEachOrdered()

  On utilise forEach() à la place

Spliterator

java.util.Spliterator

Abstraction caché derrière un Stream

Cela évite de ré-implanter les 30+ méthodes de l'interface Stream pour chaque implantation

Un Spliterator est un

- iterateur *push* (java.util.Iterator est *pull*)
- qui sait se séparer en 2 (*split*)

Un Spliterator retient les caractéristiques de la source du Stream pour optimiser les traitements

Il essaye aussi d'estimer sa taille pour séparer en 2 parties à peu près égales

Implanter un Spliterator<T>

trySplit(): Spliterator<T>

Essaye de se couper en 2, renvoie l'autre partie ou null

tryAdvance(Consumer<? super T> action)

Si il reste au moins un élément, appel le consumer sur l'élément courant, passe au suivant et renvoie vrai sinon renvoie faux

int characteristics()

Renvoie les caractéristiques du spliterator

long estimateSize()

La taille estimée ou Long.MAX_VALUE
(si infinie ou trop long à calculer)

Exemple à partir d'un Iterateur

```
public static <T> Spliterator<T> fromIterator(Iterator<? extends T> it) {
    return new Spliterator<T>() {
      @Override
      public Spliterator<T> trySplit() { return null; }

      @Override
      public boolean tryAdvance(Consumer<? super T> consumer) {
        if (it.hasNext()) {
          consumer.accept(it.next());
          return true;
        }
        return false;
      }

      @Override
      public int characteristics() { return 0; }

      @Override
      public long estimateSize() { return Long.MAX_VALUE; }
    };
  }
```

Exemple à partir d’un Iterateur

java.util.stream.StreamSupport permet de créer un Stream à partir d’un Spliterator

```
Spliterator<String> spliterator = …
boolean parallel = ...
Stream<String> stream =
    StreamSupport.stream(spliterator, parallel);
```

Caracteristiques d’un Spliterator

CONCURRENT (La source est concurrente)

IMMUTABLE (La source est non mutable)

NONNULL (Pas d’élement null)

ORDERED (Les éléments sont ordonnées)

DISTINCT (Pas deux fois le même élement)

SORTED (Les données sont triées)

SIZED (Le nombre d’éléments est connu)

SUBSIZED (Après un trySplit(), le nombre d’éléments est connu)

Exemple à partir d'un Tableau

```
@SafeVarargs
public static <T> Spliterator<T> fromArray(int start, int end, T... array) {
  return new Spliterator<T>() {
     private int i = start;

    @Override
    public Spliterator<T> trySplit() { return null; }

    @Override
    public boolean tryAdvance(Consumer<? super T> consumer) {
      if (i < end) { consumer.accept(array[i++]);  return true; }
      return false;
    }

    @Override
    public int characteristics() { return SIZED; }

    @Override
    public long estimateSize() { return end - i; }
  };
}
```

Avec le trySplit

```
@SafeVarargs
public static <T> Spliterator<T> fromArray(int start, int end, T... array) {
  return new Spliterator<>() {
    private int i = start;

    @Override
    public Spliterator<T> trySplit() {
      var middle = (i + end) >>> 1;
      if (middle == i) {
        return null;
      }
      var spliterator = fromArray(i, middle, array);
      i = middle;
      return spliterator;
    }

    @Override
    public int characteristics() { return SIZED | SUBSIZED; }

    ...
  };
}
```

Methodes additionnelles

Methodes dont ils existent une implantation par défaut et qui peuvent être redéfinie si il existe une implantation plus efficace

- forEachRemaning(Consumer<? super T> consumer)

  Parcours les éléments restants comme un forEach

- Comparator<? super T> getComparator()

  Renvoie le comparateur utiliser par la collection (pour TreeSet ou TreeMap.keySet())

java.util.Spliterators

Fournit des implantations de Spliterator par défaut

- Spliterator<T> spliterator(T[] array)
- Spliterator<T> spliteratorUnknownSize(
  Iterator<? extends T> iterator)

et une méthode inverse (pas efficace)

- Iterator<T> iterator(
  Spliterator<? extends T> spliterator)

Exemple complet: FizzBuzz

Comment écrire FizzBuzz avec des Streams ?

FizzBuzz:
pour les nombres de 1 à 100

- On affiche Fizz, si le nombre est un multiple de 3
- On affiche Buzz, si le nombre est un multiple de 5
- On affiche FizzBuzz, si c’est un multiple de 15
- Sinon on affiche le nombre

Note: il existe des solutions très simples
pour implanter FizzBuzz sans Stream

FizzBuzz

On utilise plusieurs streams infinis

ints	cycle("Fizz")	cycle("Buzz")	zip(+)
0	Fizz	Buzz	FizzBuzz
1	""	""	""
2	""	""	""
3	Fizz	""	Fizz
4	""	""	""
5	""	Buzz	Buzz
6	Fizz	""	Fizz
7	""	""	""
8	""	""	""
9	Fizz	""	Fizz
…	…	…	…

FizzBuzz: cycle

```
public static <T> Stream<T> cycle(
                    int length, T element, T empty) {

    return IntStream
        .iterate(0, x -> (x + 1) % length)
        .mapToObj(x -> x == 0? element: empty);
  }
```

FizzBuzz: zip()

```
public static <T, U, V> Stream<V> zip(Stream<T> s1, Stream<U> s2,
                                      BiFunction<? super T, ? super U, ? extends V> merger) {
    var spliterator1 = s1.spliterator();
    var spliterator2 = s2.spliterator();
    return StreamSupport.stream(new Spliterator<>() {
      @Override
      public int characteristics() {
        return spliterator1.characteristics() & spliterator2.characteristics();
      }

      @Override
      public long estimateSize() {
        var size1 = spliterator1.estimateSize();
        var size2 = spliterator2.estimateSize();
        return (size1 == Long.MAX_VALUE || size2 == Long.MAX_VALUE)?
                Long.MAX_VALUE: Math.min(size1, size2);
      }

      ...
    }, false);
  }
```

FizzBuzz: zip()

```
public static <T, U, V> Stream<V> zip(Stream<T> s1, Stream<U> s2,
                                      BiFunction<? super T, ? super U, ? extends V> merger) {
    var spliterator1 = s1.spliterator();
    var spliterator2 = s2.spliterator();
    return StreamSupport.stream(new Spliterator<>() {
      ...
      @Override
      public Spliterator<V> trySplit() { return null; }

      @Override
      public boolean tryAdvance(Consumer<? super V> action) {
        var box = new Object() { T value; }
        return spliterator1.tryAdvance(t -> box.value = t) &&
          spliterator2.tryAdvance(u -> action.accept(merger.apply(box.value, u)));
      }
    }, false);
  }
```

FizzBuzz: main

```
Stream<String> fizzbuzz =
    zip(
      IntStream.iterate(0, x -> x + 1).boxed(),
      zip(
        cycle(3, "Fizz", ""),
        cycle(5, "Buzz", ""),
        String::concat),
        (i, s) -> s.isEmpty()? "" + i: s);


fizzbuzz.skip(1)
            .limit(100)
            .forEach(System.out::println);
```

Optional

java.util.Optional

Si le résultat d'un calcul peut ne pas exister,
au lieu d'utiliser null, on utilise Optional qui oblige l'utilisateur à gérer le cas où il n'y a pas de valeur

Exemples:

Trouver la première valeur d'un Stream<E>

- Stream.findFirst() renvoie Optional<E>

Calculer le maximum d'un IntStream

- intStream.max() renvoie un OptionalInt

Comme les Stream, il existe des Optional pour les types primitif (OptionalInt, OptionalLong, OptionalDouble)

Créer un Optional

Optional possède des méthodes statiques de création (**static factory methods**)

- Un Optional sans valeur
  - Optional.**empty**()
- A partir d'une valeur non null
  - Optional.**of**(E element)
- A partir d'une valeur qui peut être null
  - Optional.**ofNullable**(E elementOrNull)

API

La valeur existe ou pas ?

isEmpty(), **isPresent**()

Obtenir la valeur

~~**get**~~()/**orElseThrow**() throws NSEE,
orElse(T),
orElseGet(Supplier<? extends T>),
orElseThrow(Supplier<? extends Throwable>)

orElseThrow (ex: ~~get~~()) peut lever une exception (à utiliser avec précaution)

Optional et calcul

En plus de représenter une valeur ou non d'un calcul, il est possible d'effectuer des opérations directement sur un Optional

Au lieu de

```
Optional<String> opt = …
if (opt.isPresent()) {
  System.out.println(opt.orElseThrow());
}
```

on demande à l'Optional de faire le calcul

```
Optional<String> opt = …
opt.ifPresent(System.out::println);
```

API (2)

Si présent

void **ifPresent(**Consumer<? super T>)

Transformation

Optional<R> **map**(Function<? super T, ? extends R>),

Optional<R> **flatMap**(Function<? super T,
? extends Optional<R>)

Composition

Optional<T> **or**(Supplier<? extends Optional<T>>)

Pont vers un Stream

Stream<T> **stream**()

Exemple

Au lieu de

```
OptionalInt result = ...
if (result.isPresent()) {
  System.out.println("result is " + result.orElseThrow());
} else {
  System.out.println("result not found");
}
```

On écrit

```
OptionalInt result = …
System.out.println(result
        .map(value - > "result is " + value)
        .orElse("result not found"));
```

Ne pas stocker Optional dans un champ

Problème avec

```
public class Foo {
  private final Optional<Bar> bar;

  public Foo(Optional<Bar> bar) { this.bar = bar; }

  public Optional<Bar> getBar() { return bar; }
}
```

Fait un double dé-référencement inutile de la mémoire (comme Integer à la place de int)

Ne pas stocker Optional dans un champ

On utilise l'encapsulation, on stocke null en interne mais on export un Optional au lieu de null

```
public class Foo {
  private final Bar bar;  // maybe null

  public Foo(Optional<Bar> bar) {
    this.bar = bar.orElse(null);
  }

  public Optional<Bar> getBar() {
    return Optional.ofNullable(bar);
  }
}
```

Pas de Collection ou Map d'Optional

On va éviter de stocker des trucs qui peuvent ne pas exister

avec class Foo { Optional<Bar> getBar() { … }}

On évite

```
List<Foo> foos = …
List<Optional<Bar>> bars =
    foos.stream()
            .map(Foo::getBar)
            .collect(Collectors.toList())
```

Pas de Collection ou Map d'Optional

On utilise flatMap() pour faire disparaitre les Optional (en les transformant en Stream)

avec class Foo { Optional<Bar> getBar() { … }}

On écrit

```
List<Foo> foos = …
List<Bar> bars =
    foos.stream()
        .flatMap(foo - > foo.getBar().stream())
        .collect(Collectors.toList())
```