Java 8 Streams & Collectors

Patterns, performance, parallélisation

@JosePaumard

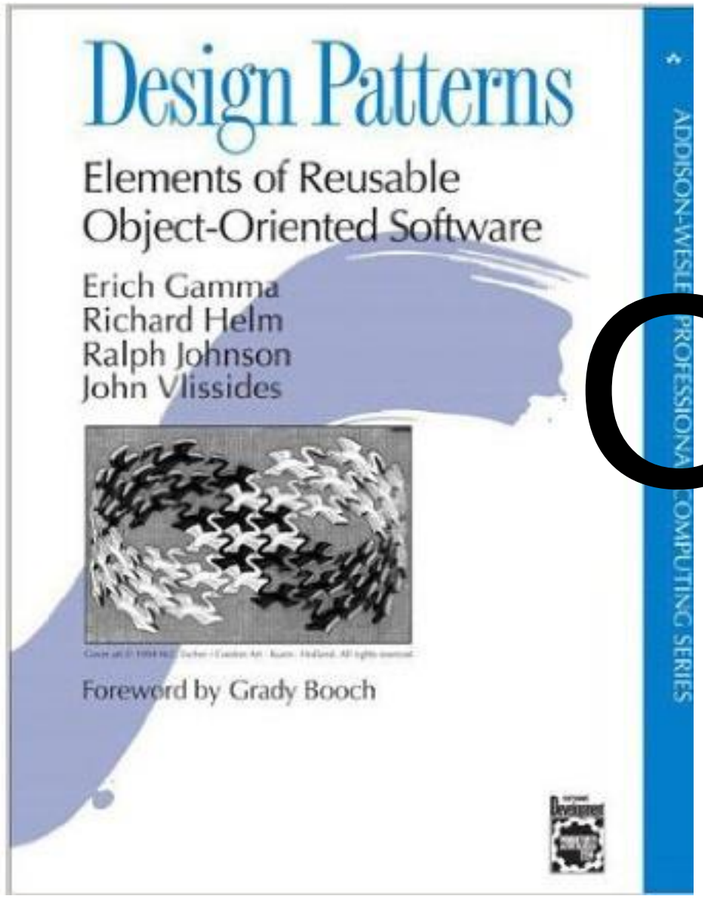
@JosePaumard

Stream

@JosePaumard

Stream Collectors

@JosePaumard



Stream Collectors

@JosePaumard

José PAUMARD

MCF Um. Paris 13

PhD Appl C.S.

Open source dev.

Independant

José PAUMARD

Open source dev.

Independant

José PAUMARD

Paris JUG

Devoxx FRANCE

Du code et des slides

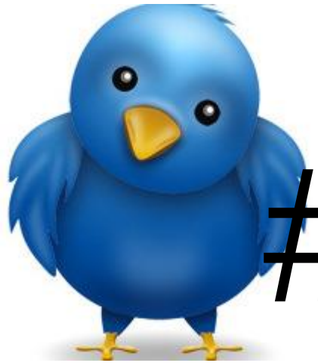
1ère partie : des slides

- Stream
- Opérations
- État
- Réduction
- Collector

2ème partie : code + slides

- Houston
- Shakespeare
- Actors

Questions ?



#Stream8

@JosePaumard

Stream

Qu’est-ce qu’un Stream ?

Qu’est-ce qu’un Stream ?

Techniquement : une interface paramétrée

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

   // ...
}
```

Qu’est-ce qu’un Stream ?

Techniquement : une interface paramétrée

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

   // ...
}
```

Pratiquement : un nouveau concept

Qu’est-ce qu’un Stream ?

À quoi un Stream sert-il ?

Qu’est-ce qu’un Stream ?

À quoi un Stream sert-il ?

Réponse : à traiter efficacement les grands volumes de données, et aussi les petits

Qu’est-ce qu’un Stream ?

Que signifie *efficacement* ?

Qu’est-ce qu’un Stream ?

Que signifie *efficacement* ?

Deux choses :

Qu’est-ce qu’un Stream ?

Que signifie *efficacement ?*

Deux choses :

1) en parallèle, pour exploiter le multicœur

Qu’est-ce qu’un Stream ?

Que signifie *efficacement ?*

Deux choses :

1) en parallèle, pour exploiter le multicœur
2) en pipeline, pour éviter les intermédiaires de calcul

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Il y a des arguments pour qu’une collection soit un Stream !

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Il y a des arguments pour qu’une collection soit un Stream !

1) mes données sont dans des collections (ou tables de hachage)

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Il y a des arguments pour qu’une collection soit un Stream !

1) mes données sont dans des collections (ou tables de hachage)
2) on connaît bien l’API Collection…

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Deux raisons :

1) ça permet d’avoir les mains libres

Qu’est-ce qu’un Stream ?

Pourquoi une collection ne peut-elle constituer un Stream ?

Deux raisons :

1) ça permet d’avoir les mains libres
2) ça permet de ne pas polluer l’API Collection avec ces nouveaux concepts

Donc : qu’est-ce qu’un Stream ?

Réponses :

Donc : qu’est-ce qu’un Stream ?

Réponses :

1) un objet qui sert à définir des opérations

Donc : qu’est-ce qu’un Stream ?

Réponses :

1) un objet qui sert à définir des opérations
2) qui ne possède pas les données qu’il traite (source)

Donc : qu’est-ce qu’un Stream ?

Réponses :

1) un objet qui sert à définir des opérations
2) qui ne possède pas les données qu’il traite (source)
3) qui s’interdit de modifier les données qu’il traite

Donc : qu’est-ce qu’un Stream ?

Réponses :

1) un objet qui sert à définir des opérations
2) qui ne possède pas les données qu’il traite (source)
3) qui s’interdit de modifier les données qu’il traite
4) qui traite les données en « une passe »

Donc : qu'est-ce qu'un Stream ?

Réponses :

1) un objet qui sert à définir des opérations
2) qui ne possède pas les données qu'il traite (source)
3) qui s'interdit de modifier les données qu'il traite
4) qui traite les données en « une passe »
5) qui est optimisé du point de vue algorithmique et qui est capable de calculer en parallèle

Comment construit-on un Stream ?

Plein de patterns !

Comment construit-on un Stream ?

Plein de patterns !

Choisissons-en un :

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
```

Opérations sur un Stream

Première opération : forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(p -> System.out.println(p)) ;
```

… affiche chaque personne sur la console

Opérations sur un Stream

Première opération : forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(System.out::println) ;
```

… affiche chaque personne sur la console

Opérations sur un Stream

Première opération : forEach()

```
List<Person> persons = ... ;

Stream<Person> stream = persons.stream() ;
stream.forEach(System.out::println) ;
```

Method reference :

```
o -> System.out.println(o) ≡ System.out::println
```

Opération forEach()

Première opération : forEach()

forEach() : prend un Consumer<T> en paramètre

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;
}
```

Opération forEach()

Première opération : forEach()

forEach() : prend un Consumer<T> en paramètre

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;
}
```

Interface fonctionnelle ?

Sauf qu'en fait…

Consumer est un peu plus complexe que ça :

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Sauf qu'en fait…

Consumer est un peu plus complexe que ça :

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Méthode par défaut ?

Interface fonctionnelle

= une interface qui ne possède qu'une seule méthode

Interface fonctionnelle

= une interface qui ne possède qu'une seule méthode

« peut » être annotée par @FunctionalInterface

Interface fonctionnelle

= une interface qui ne possède qu'une seule méthode

« peut » être annotée par @FunctionalInterface

Les méthodes de Object ne « comptent » pas

Interface fonctionnelle

= une interface qui ne possède qu'une seule méthode

« peut » être annotée par @FunctionalInterface

Les méthodes de Object ne « comptent » pas

Peut comporter des méthodes « par défaut »

Méthodes par défaut

Nouveauté Java 8 !

Méthodes par défaut

Nouveauté Java 8 !

Intégrée pour permettre de faire évoluer des vieilles interfaces

Méthodes par défaut

Nouveauté Java 8 !

Intégrée pour permettre de faire évoluer des vieilles interfaces

Cas de Collection : on a ajouté stream()

Méthodes par défaut

Quid de l’héritage multiple ?

Méthodes par défaut

Quid de l'héritage multiple ?

On a déjà l'héritage multiple en Java !

```
public final class String
implements Serializable, Comparable<String>, CharSequence {

   // ...
}
```

Méthodes par défaut

Quid de l'héritage multiple ?

On a déjà l'héritage multiple en Java !

```
public final class String
implements Serializable, Comparable<String>, CharSequence {

   // ...
}
```

On a l'héritage multiple de *type*

Méthodes par défaut

Java 8 amène l'héritage multiple d'*implémentation*

Méthodes par défaut

Java 8 amène l'héritage multiple d'*implémentation*

Ce que l'on n'a pas, c'est l'héritage multiple d'*état*

Méthodes par défaut

Java 8 amène l'héritage multiple d'*implémentation*

Ce que l'on n'a pas, c'est l'héritage multiple d'*état*

et d'ailleurs… on n'en veut pas !

Méthodes par défaut

Conflits ?

Méthodes par défaut

Conflits ?

oui…

Méthodes par défaut

```
public class A
implements B, C {

}
```

```
public interface B {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

Méthodes par défaut

```
public class A
implements B, C {

}
```

```
public interface B {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

Conflit : erreur de compilation !

Méthodes par défaut

```
public class A
implements B, C {

}
```

```
public interface B {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

Pour lever l’erreur : deux solutions

Méthodes par défaut

```
public class A
implements B, C {
   public String a() {...}
}
```

```
public interface B {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

1) La classe gagne !

Méthodes par défaut

```
public class A
implements B, C {
   public String a() { B.super.a() ; }
}
```

```
public interface B {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

1) La classe gagne !

Méthodes par défaut

```
public class A
implements B, C {

}
```

```
public interface B extends C {

   default String a() {...}
}
```

```
public interface C {

   default String a() {...}
}
```

2) Le plus spécifique gagne !

Méthodes par défaut

Conflits ?

oui…

2 règles pour les gérer :

Méthodes par défaut

Conflits ?

oui…

2 règles pour les gérer :

1) La classe gagne !

Méthodes par défaut

Conflits ?

oui…

2 règles pour les gérer :

1) La classe gagne !
2) Le plus spécifique gagne !

Retour sur Consumer

```
@FunctionalInterface
public interface Consumer<T> {

   void accept(T t) ;

   default Consumer<T> andThen(Consumer<? super T> after) {
      Objects.requireNonNull(after);
      return (T t) -> { accept(t); after.accept(t); };
   }
}
```

Retour sur Consumer

```
List<String> liste = new ArrayList<>() ;

Consumer<String> c1 = s -> liste.add(s) ;
Consumer<String> c2 = s -> System.out.println(s) ;
```

Retour sur Consumer

```
List<String> liste = new ArrayList<>() ;

Consumer<String> c1 = liste::add ;
Consumer<String> c2 = System.out::println ;
```

Retour sur Consumer

```
List<String> liste = new ArrayList<>() ;

Consumer<String> c1 = liste::add ;
Consumer<String> c2 = System.out::println ;

Consumer<String> c3 = c1.andThen(c2) ; // et on pourrait continuer
```

Retour sur Consumer

Attention à la concurrence !

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

Consumer<String> c1 = liste::add ;

persons.stream()
       .forEach(c1) ; // concurrence ?
```

Retour sur Consumer

Attention à la concurrence !

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

Consumer<String> c1 = liste::add ;

persons.stream().parallel()
       .forEach(c1) ; // concurrence ?
```

Retour sur Consumer

Attention à la concurrence !

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

Consumer<String> c1 = liste::add ;

persons.stream()
       .forEach(c1) ; // concurrence ? Baaad pattern !
```

Retour sur Consumer

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

Consumer<String> c1 = liste::add ;
Consumer<String> c2 = System.out::println

persons.stream()
       .forEach(c1.andThen(c2)) ;
```

Problème : forEach() ne retourne rien

Retour sur Consumer

Peek à la rescousse !

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(resultat::add) ; // Baaad pattern !
```

Retour sur Stream

Donc on a :

- une méthode forEach(Consumer)
- une méthode peek(Consumer)

Retour sur Stream

3ème méthode : filter(Predicate)

Méthode filter()

3ème méthode : filter(Predicate)

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Méthode filter()

3ème méthode : filter(Predicate)

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

```
Predicate<Person> p = person -> person.getAge() > 20 ;
```

Interface Predicate

Interface fonctionnelle

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;
}
```

Interface Predicate

En fait …

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;

   default Predicate<T> and(Predicate<? super T> other) { ... }

   default Predicate<T> or(Predicate<? super T> other) { ... }

   default Predicate<T> negate() { ... }
}
```

Interface Predicate

```
default Predicate<T> and(Predicate<? super T> other) {
   return t -> test(t) && other.test(t) ;
}
```

```
default Predicate<T> or(Predicate<? super T> other) {
   return t -> test(t) || other.test(t) ;
}
```

```
default Predicate<T> negate() {
   return t -> !test(t) ;
}
```

Interface Predicate : patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;
```

Interface Predicate : patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;

Predicate<Integer> p = p1.and(p2).or(p3) ; // (p1 AND p2) OR p3
```

Interface Predicate : patterns

```
Predicate<Integer> p1 = i -> i > 20 ;
Predicate<Integer> p2 = i -> i < 30 ;
Predicate<Integer> p3 = i -> i == 0 ;

Predicate<Integer> p = p1.and(p2).or(p3) ; // (p1 AND p2) OR p3
Predicate<Integer> p = p3.or(p1).and(p2) ; // (p3 OR p1) AND p2
```

Interface Predicate

En fait (bis) …

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t) ;

   // méthodes par défaut

   static <T> Predicate<T> isEqual(Object o) { ... }
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

En fait :

```
static <T> Predicate<T> isEqual(Object o) {

   return (null == o)
            ? obj -> Objects.isNull(obj)
            : t -> o.equals(t) ;
}
```

Interface Predicate

```
static <T> Predicate<T> isEqual(Object o) {

   return t -> o.equals(t) ;
}
```

En fait :

```
static <T> Predicate<T> isEqual(Object o) {

   return (null == o)
            ? Objects::isNull
            : o::equals ;
}
```

Interface Predicate : patterns

```
Predicate<String> p = Predicate.isEqual("deux") ;
```

Interface Predicate : patterns

```
Predicate<String> p = Predicate.isEqual("deux") ;

Stream<String> stream1 = Stream.of("un", "deux", "trois") ;

Stream<String> stream2 = stream1.filter(p) ;
```

Interface Predicate : remarque

Dans ce code :

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Les deux streams *stream* et *filtered* sont différents

La méthode filter() retourne un nouvel objet

Interface Predicate : remarque

Question : qu’est-ce que j’ai dans cet objet ?

Interface Predicate : remarque

Question : qu’est-ce que j’ai dans cet objet ?

Réponse : mes données filtrées

Interface Predicate : remarque

Question : qu’est-ce que j’ai dans cet objet ?

Réponse : mes données filtrées

vraiment ?

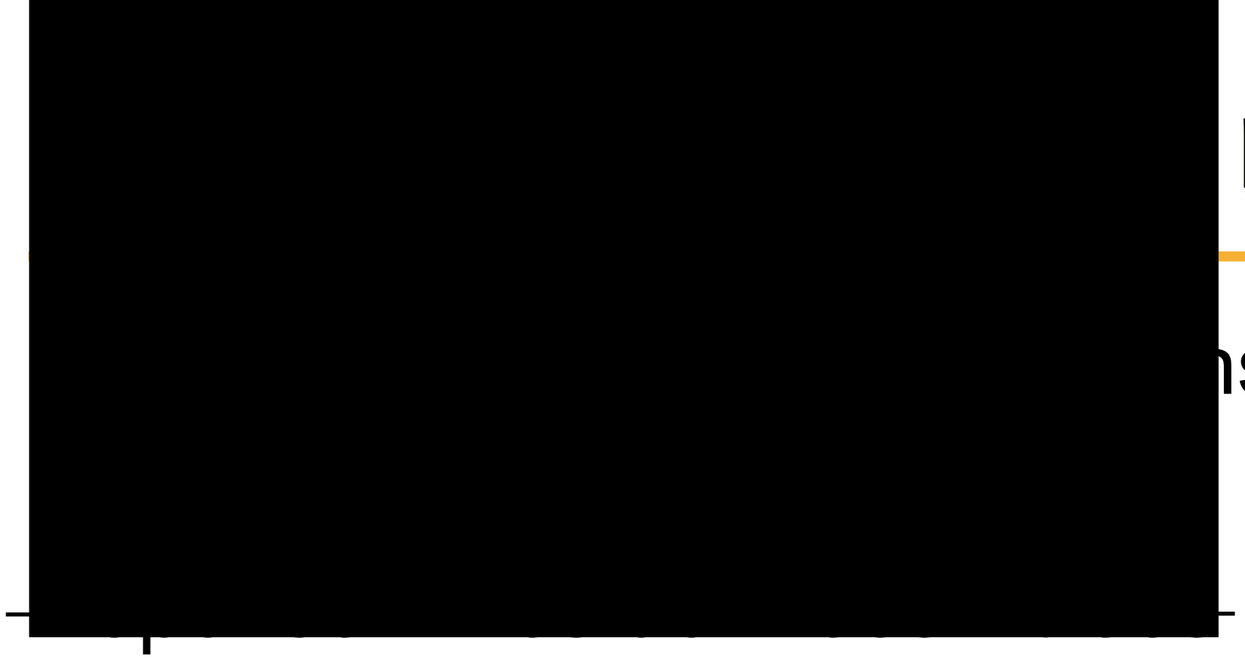
Interface Predicate : remarque

Question : qu'est-ce que j'ai dans cet objet ?

Réponse : mes données filtrées

vraiment ?

« un stream ne porte pas de données »…



remarque

ns cet objet ?

remarque

ns cet objet ?

Interface Predicate : remarque

Question 2 :

que se passe-t-il lors de l'exécution de ce code ?

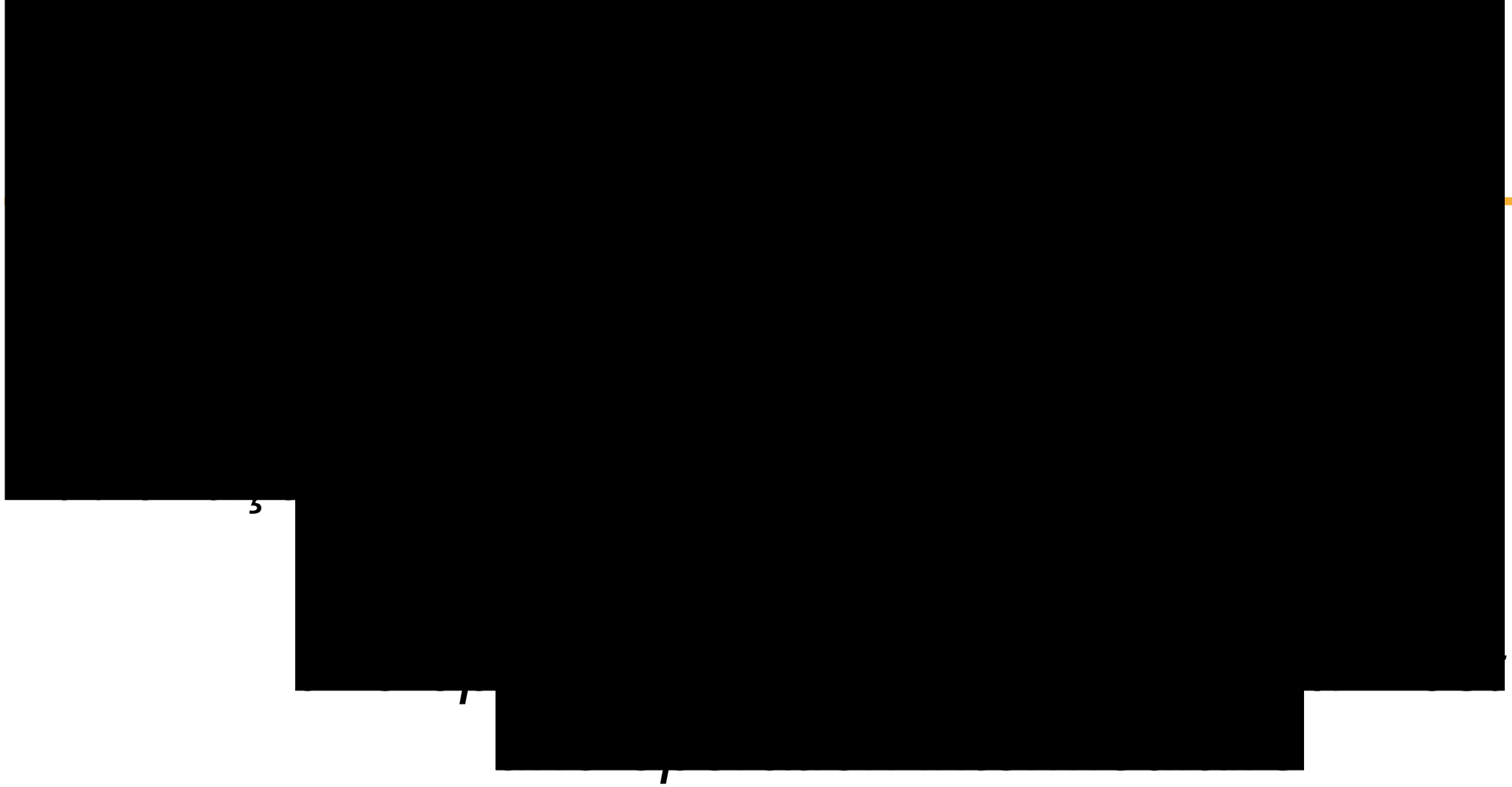
```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

Predicate : remarque

l lors de l'exécution de ce code ?

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<Person> filtered =
   stream.filter(person -> person.getAge() > 20) ;
```



Appels intermédiaires

Le fait de choisir qu'un stream ne porte pas ses données crée la notion d'opération intermédiaire

Appels intermédiaires

Le fait de choisir qu'un stream ne porte pas ses données crée la notion d'opération intermédiaire

On doit bien avoir des opérations terminales quelque part…

Retour sur Consumer (bis)

Que se passe-t-il dans ce code ?

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(resultat::add) ; // Baaad pattern !
```

Retour sur Consumer (bis)

Que se passe-t-il dans ce code ?

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

persons.stream()
       .peek(System.out::println)
       .filter(person -> person.getAge() > 20)
       .peek(resultat::add) ; // Baaad pattern !
```

Rien !

Retour sur Consumer (bis)

Que se passe-t-il dans ce code ?

```
List<String> resultat = new ArrayList<>() ;
List<Person> persons = ... ;

persons.stream()
        .peek(System.out::println)
        .filter(person -> person.getAge() > 20)
        .peek(resultat::add) ; // Baaad pattern !
```

1) rien ne s'affiche !

2) resultat reste vide !

Retour sur Stream

Donc on a :

- une méthode forEach(Consumer)
- une méthode peek(Consumer)
- une méthode filter(Predicate)

Continuons

Opération de mapping

Continuons

Opération de mapping

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<String> names =
   stream.map(person -> person.getNom()) ;
```

Continuons

Opération de mapping

```
List<Person> liste = ... ;
Stream<Person> stream = liste.stream() ;
Stream<String> names =
   stream.map(person -> person.getNom()) ;
```

Retourne un stream (différent du stream original)

Donc opération intermédiaire

Mapper : interface Function

Interface fonctionnelle

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;
}
```

Mapper : interface Function

En fait

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

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

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

Mapper : interface Function

En fait

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

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

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

Gare aux génériques !

Mapper : interface Function

En fait

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   default <V> Function<V, R> compose(
      Function<? super V, ? extends T> before) ;

   default <V> Function<T, V> andThen(
      Function<? super R, ? extends V> after) ;
}
```

Mapper : interface Function

En fait

```
@FunctionalInterface
public interface Function<T, R> {

   R apply(T t) ;

   // méthodes par défaut

   static <T> Function<T, T> identity() {
      return t -> t ;
   }
}
```

Retour sur Stream

Donc on a :

- une méthode forEach(Consumer)
- une méthode peek(Consumer)
- une méthode filter(Predicate)
- une méthode map(Function)

Continuons

Opération flatMap

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

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

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

La fonction mapper transforme

des éléments T

en éléments Stream<R>

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

S’il s’agissait d’un mapping classique
flatMap() retournerait Stream<Stream<R>>

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

S’il s’agissait d’un mapping classique
flatMap() retournerait Stream<Stream<R>>

Donc un « stream de streams »

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

Mais il s’agit d’un flatMap !

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

Mais il s’agit d’un flatMap !

Et le flatMap met le Stream<Stream> « à plat »

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

Mais il s’agit d’un flatMap !

Et le flatMap met le Stream<Stream> « à plat »

Méthode flatMap()

flatMap() = mettre à plat

Signature :

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

Retourne un stream, donc opération intermédiaire

Bilan sur Stream

On a 3 types de méthodes :

- forEach() : consomme
- peek() : regarde et passe à un consommateur
- filter() : filtre
- map() : mappe = transforme
- flatMap() : mappe et met à plat

Bilan sur Stream

On a 3 types de méthodes :

- forEach() : consomme
- peek() : regarde et passe à un consommateur
- filter() : filtre
- map() : mappe = transforme
- flatMap() : mappe et met à plat

Réduction ?

Réduction

Méthode de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer somme =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

Réduction

Méthode de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer somme =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

Expression lambda sur deux éléments : applicable à tout un stream

3 | 4 | 2 | 6 | 7 | . . . | 9 | 0

| ٣ | ١ | ٤ | ٢ | ١ | ٦ | ١ | ٧ | ... | ١ | ٩ | ٥ |

$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$

3	1	4	1	2	1	6	1	7	1	...	1	9	1	0

i_1 i_2

$i_1 + i_2$

STEP 1

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

| 3 | 1 | 4 | 1 | 2 | 1 | 6 | 1 | 7 | 1 | ... | 1 | 9 | 1 | 0 |

STEP 2

$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$

| 3 | 4 | 2 | 6 | 7 | ... | 9 | 0 |

Que calo dormo-t-il en
parallèle ?

٣	١	٤	٢	١	٦	١	٧	...	١	٩	١	٥

$i_1\ i_2$ — $i_1 + i_2$

لـ i_1, i_2 — $i_1 + i_2$

$$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$$

| ٣ | ١٤ | ٢ | ١٦ | ٧ | ... | ٩ | ٥ |

$i_1\ i_2$ $\quad$ $i_1\ i_2$ $\quad$ لـ

$i_1 + i_2$ $\quad$ $i_1 + i_2$

$i_1 \quad + \quad i_2$

c_3

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

3	1	4	1	5	9	2	6	5	3	...	1	9	1	0

$i_1\ i_2$ $\quad$ $i_1\ i_2$

$i_1 + i_2$ $\quad$ $i_1 + i_2$

$i_1 \quad + \quad i_2$

c_3

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

$\Rightarrow i_1 + (i_2 + i_3)$

$=$

$(i_1 + i_2) + i_3$

3	1	4	1	2	1	6	1	7	1	...	1	9	1	0

$\Rightarrow$ $Red(i_1, Red(i_2, i_3))$

$=$

$Red(Red(i_1, i_2), i_3)$

$\lambda = (i_1, i_2) \rightarrow i_1 + i_2$

Réduction

Doit être associative…

```
Reducer r1 = (i1, i2) -> i1 + i2 ;          // ok
```

Réduction

Doit être associative…

```
Reducer r1 = (i1, i2) -> i1 + i2 ;         // ok
Reducer r2 = (i1, i2) -> i1*i1 + i2*i2 ; // oooops...
```

Réduction

Doit être associative…

```
Reducer r1 = (i1, i2) -> i1 + i2 ;         // ok
Reducer r2 = (i1, i2) -> i1*i1 + i2*i2 ; // oooops...
Reducer r3 = (i1, i2) -> (i1 + i2)/2 ;    // re-oooops...
```

Réduction

Méthode de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer somme =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

Expression lambda sur deux éléments : applicable à tout un stream

Réduction

Méthode de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer somme =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

0 : valeur par défaut pour le cas où la source est vide

Réduction

Autre méthode de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
... max =
   stream.max(Comparator.naturalOrder()) ;
```

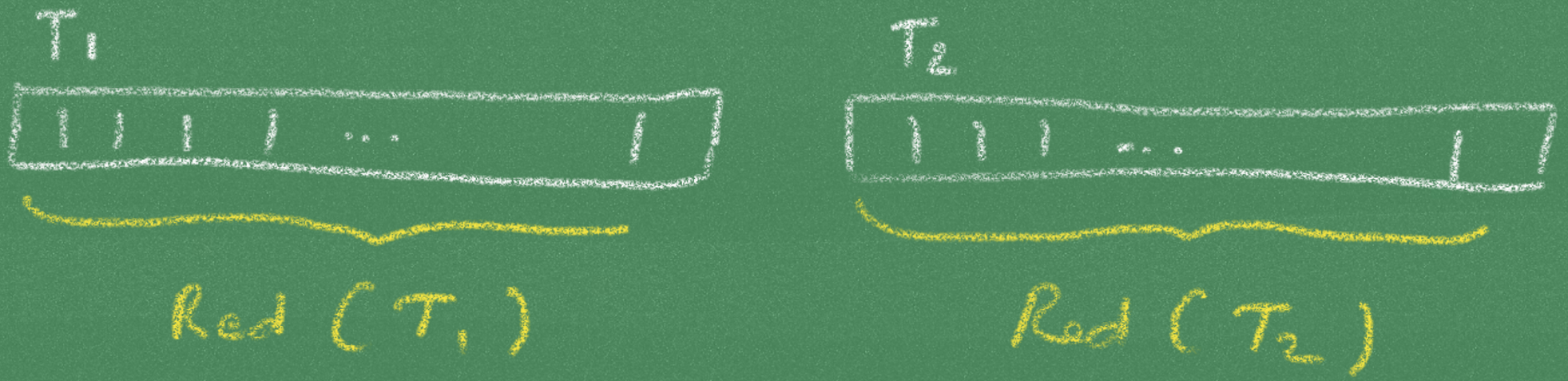
Quel type de retour pour max ? (ou min…)

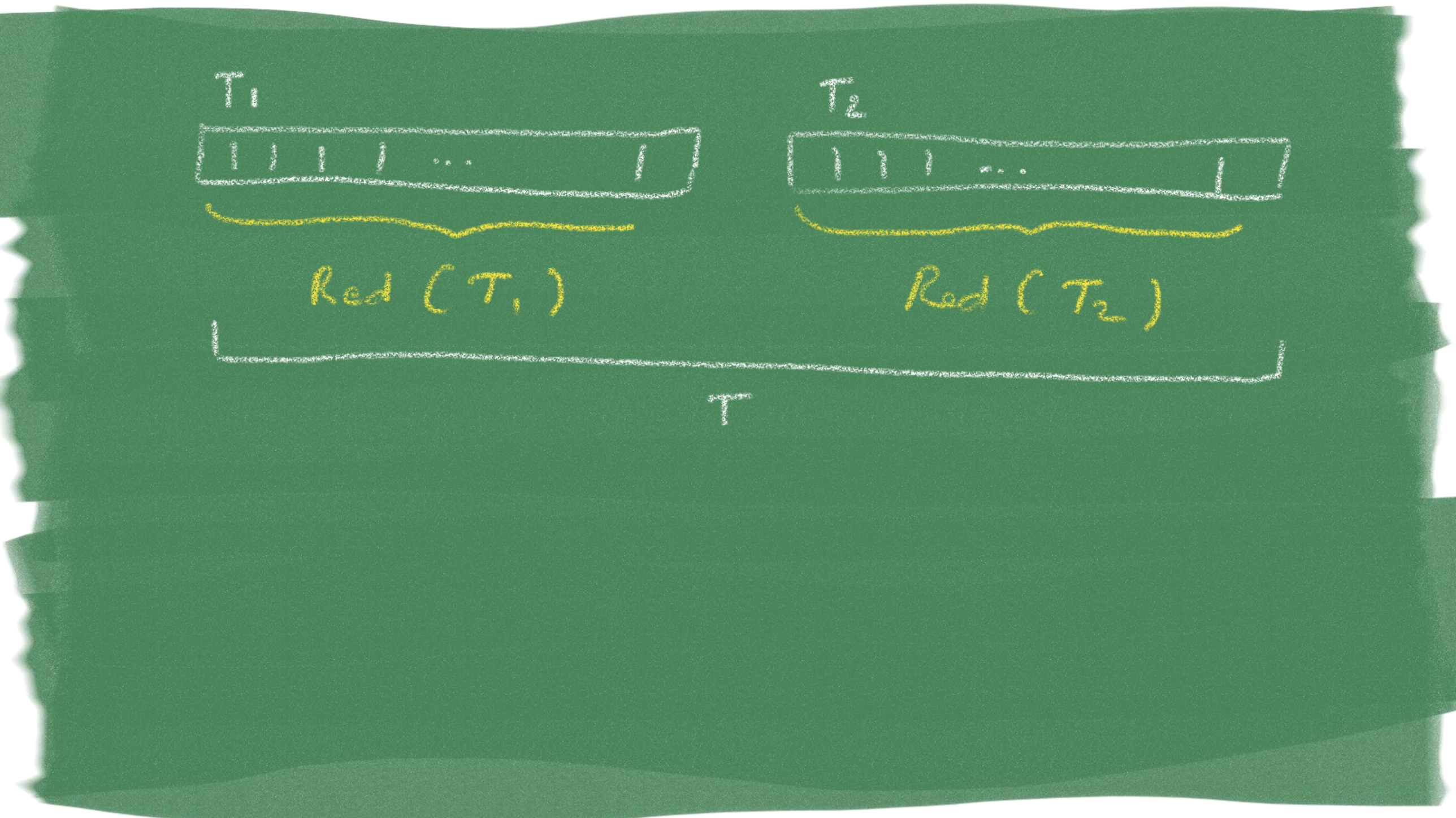
Type de retour pour la réduction

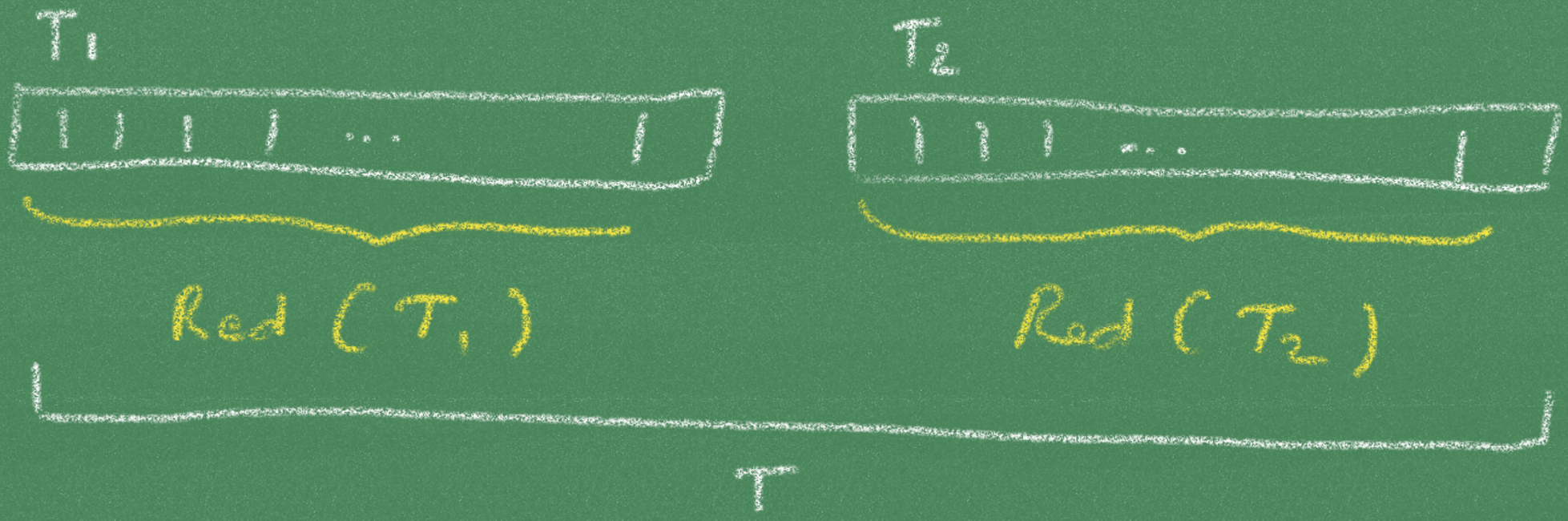
Quel est la valeur par défaut du max ?

1) la valeur par défaut est la réduction de l'ensemble vide
2) mais c'est aussi l'élément neutre de la réduction

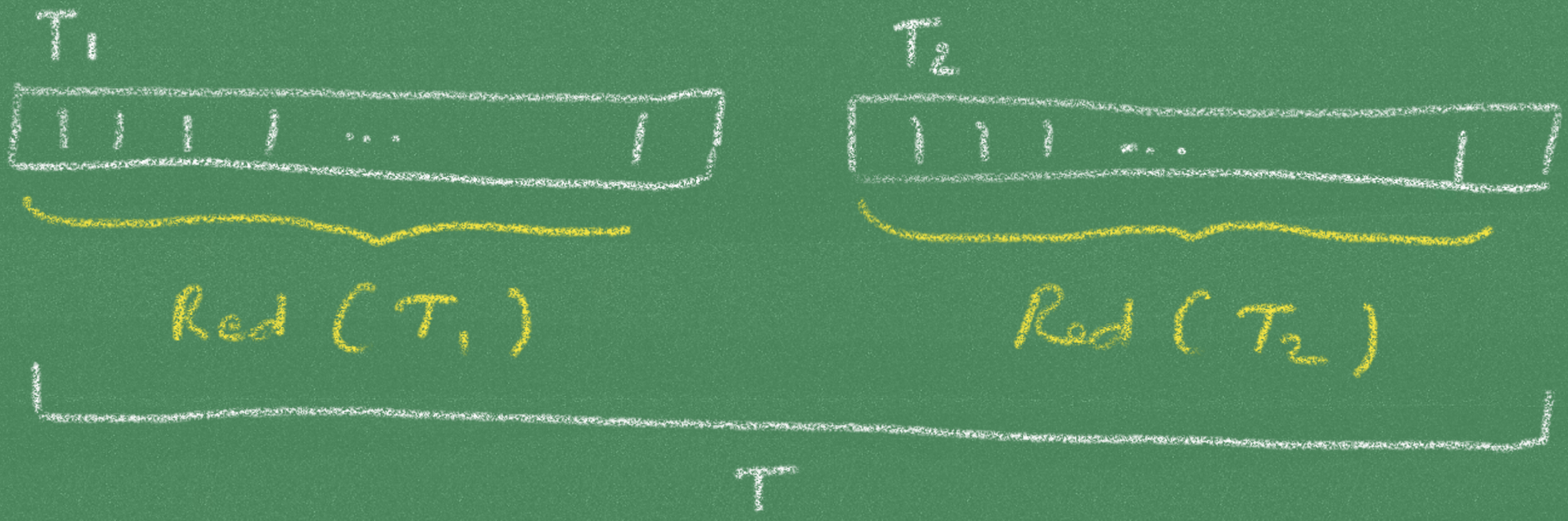

T1
T2


T1
T2
Red (T1)
Red (T2)


T1
T2
Red (T1)
Red (T2)
T



$$T = T_1 \cup T_2$$



$$T = T_1 \cup T_2$$

$$Red(T) = Red(Red(T_1), Red(T_2))$$

Suponemos que T_1 está viva $T_1 = \phi$

Suponemos que T_1 está vida $T_1 = \phi$

$T = T_2$

Suponemos que T_1 está vacía $T_1 = \emptyset$

$T = T_2$

$Red(T) = Red(Red(T_1), Red(T_2))$

Suppongan que T_1 está vacía $T_1 = \phi$

$T = T_2$

$Red(T) = Red(Red(T_1), Red(T_2))$

$Red(T) = Red(Red(\phi), Red(T))$

Supposons que T_1 est vide $T_1 = \phi$

$T = T_2$

$Red(T) = Red(Red(T_1), Red(T_2))$

$Red(T) = Red(Red(\phi), Red(T))$

$Red(\phi)$ est bien également membre de Red

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

0 ?

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

0 ?

max(0, -1) n'est pas égal à 0

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

~~0 ?~~

max(0, -1) n'est pas égal à 0

-∞ ?

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

~~0 ?~~

max(0, -1) n'est pas égal à 0

-∞ ?

souci : ce n'est pas un entier

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

~~0 ?~~

max(0, -1) n'est pas égal à 0

~~-∞ ?~~

souci : ce n'est pas un entier

Integer.*MIN_VALUE* ?

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

~~0 ?~~

max(0, -1) n'est pas égal à 0

~~-∞ ?~~

souci : ce n'est pas un entier

~~Integer.*MIN_VALUE* ?~~

souci pour convertir en long, et vice-versa

Type de retour pour la réduction

Problème : quel est l'élément neutre pour le max ?

Réponse : il n'y a pas d'élément neutre pour le max

Type de retour pour la réduction

Problème : quel est la valeur par défaut pour le max ?

Type de retour pour la réduction

Problème : quel est la valeur par défaut pour le max ?

Réponse : il n’y a pas de valeur par défaut pour le max

Type de retour pour la réduction

Quelle est alors le type de retour pour max() ?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
... max =
   stream.max(Comparator.naturalOrder()) ;
```

Type de retour pour la réduction

Quelle est alors le type de retour pour max() ?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
... max =
   stream.max(Comparator.naturalOrder()) ;
```

Si on prend int, alors la valeur par défaut est 0

Type de retour pour la réduction

Quelle est alors le type de retour pour max() ?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
... max =
   stream.max(Comparator.naturalOrder()) ;
```

Si on prend int, alors la valeur par défaut est 0

Si on prend Integer, alors la valeur par défaut est null

Type de retour pour la réduction

Quelle est alors le type de retour pour max() ?

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Optional<Integer> max =
   stream.max(Comparator.naturalOrder()) ;
```

On a besoin d'un nouveau concept : Optional

Optional

Un optional encapsule un objet

Peut être vide

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

Optional

Un optional encapsule un objet

Peut être vide

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

```
String s = opt.orElse("") ; // défini une valeur par défaut
                            // applicative
```

Optional

Un optional encapsule un objet

Peut être vide

```
Optional<String> opt = ... ;
if (opt.isPresent()) {
   String s = opt.get() ;
} else {
   ...
}
```

```
String s = opt.orElseThrow(MyException::new) ; // construction lazy
```

Retour sur la réduction

Méthodes de réduction :

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Integer somme =
   stream.reduce(0, (age1, age2) -> age1 + age2) ;
```

```
List<Integer> ages = ... ;
Stream<Integer> stream = ages.stream() ;
Optional<Integer> opt =
   stream.reduce((age1, age2) -> age1 + age2) ;
```

Remarque sur la réduction

Une réduction ne retourne pas de Stream :

- max(), min()
- count()

Réduction booléennes :

- allMatch(), noneMatch, anyMatch()

Retourne un Optional

- findFirst(), findAny() (si le Stream est vide ?)

Remarque sur la réduction

Dans tous les cas, ces réductions retournent une valeur

Elles ne peuvent donc pas être évaluées de façon *lazy*

Elles déclenchent les opérations !

Ce sont les opérations *terminales*

Opération terminale

```
List<Person> persons = ... ;

... =
persons.map(person -> person.getAge())   // retourne Stream<Integer>
       .filter(age -> age > 20)          // retourne Stream<Integer>
       .min(Comparator.naturalOrder()) ;
```

Opération terminale

Écriture d'un map / filter / reduce

```
List<Person> persons = ... ;

Optional<Integer> age =
persons.map(person -> person.getAge())   // retourne Stream<Integer>
       .filter(age -> age > 20)          // retourne Stream<Integer>
       .min(Comparator.naturalOrder()) ; // opération terminale
```

Opération terminale

Écriture d'un map / filter / reduce

```
List<Person> persons = ... ;

persons.map(person -> person.getLastName())
       .allMatch(length < 20) ;                    // opération terminale
```

Opération terminale

Écriture d'un map / filter / reduce

```
List<Person> persons = ... ;

persons.map(person -> person.getLastName())
       .allMatch(length < 20) ;                    // opération terminale
```

Intérêt de tout faire dans la même boucle

Qu’est-ce qu’un Stream ?

Un objet qui permet de définir des traitements sur des jeux de données arbitrairement grands

Typiquement : map / filter / reduce

Approche pipeline :

1) on définit des opérations
2) on lance les opérations

Comment un Stream est-il fait ?

Que se passe-t-il lors de la construction d'un Stream ?

```
List<Person> persons = new ArrayList<>() ;
Stream<Person> stream = persons.stream() ;
```

```
// interface Collection
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

Comment un Stream est-il fait ?

Que se passe-t-il lors de la construction d'un Stream ?

```
// classe StreamSupport
public static <T> Stream<T> stream(
   Spliterator<T> spliterator, boolean parallel) {

   Objects.requireNonNull(spliterator) ;
   return new ReferencePipeline.Head<>(
                  spliterator,
                  StreamOpFlag.fromCharacteristics(spliterator),
                  parallel) ;
}
```

Comment un Stream est-il fait ?

Que se passe-t-il lors de la construction d'un Stream ?

```
// classe ArrayList
@Override
public Spliterator<E> spliterator() {
   return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

Comment un Stream est-il fait ?

Que se passe-t-il lors de la construction d'un Stream ?

```
// classe ArrayList
@Override
public Spliterator<E> spliterator() {
   return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

Le Spliterator encapsule la logique d'accès aux données

Celui d'ArrayList manipule le tableau

Fonction du Spliterator

Méthodes à implémenter

```
// interface Spliterator
boolean tryAdvance(Consumer<? super T> action);
```

Consomme le prochain élément s’il existe

Fonction du Spliterator

Méthodes à implémenter

```
// interface Spliterator
Spliterator<T> trySplit();
```

Utilisée par le parallélisme : divise les données en deux, suivant des règles précises

Fonction du Spliterator

Méthodes à implémenter

```
// interface Spliterator
long estimateSize();
```

Retourne une estimation du nombre d’éléments de ce Stream

Fonction du Spliterator

Méthodes par défaut

```
// interface Spliterator
default void forEachRemaining(Consumer<? super T> action) {
   do { } while (tryAdvance(action));
}
```

```
// interface Spliterator
default long getExactSizeIfKnown() {
   return (characteristics() & SIZED) == 0 ? -1L : estimateSize();
}
```

Fonction du Spliterator

Méthodes à implémenter

```
// interface Spliterator
int characteristics();
```

Implémentations

```
// pour ArrayList
public int characteristics() {
    return Spliterator.ORDERED |
           Spliterator.SIZED |
           Spliterator.SUBSIZED;
}
```

Fonction du Spliterator

Méthodes à implémenter

```
// interface Spliterator
int characteristics();
```

Implémentations

```
// pour HashSet
public int characteristics() {
   return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
           Spliterator.DISTINCT;
}
```

Caractéristiques d'un Stream

Un Stream porte des caractéristiques

Caractéristique	
ORDERED	L'ordre a un sens
DISTINCT	Pas de doublon
SORTED	Trié
SIZED	Le cardinal est connu
NONNULL	Pas de valeurs nulles
IMMUTABLE	Ne peut pas être modifié
CONCURRENT	Autorise le parallélisme
SUBSIZED	Le cardinal est connu

Caractéristiques d'un Stream

Les opérations changent la valeur des caractéristiques

Méthode	Mets à 0	Mets à 1
Filter	SIZED	-
Map	DISTINCT, SORTED	-
FlatMap	DISTINCT, SORTED, SIZED	-
Sorted	-	SORTED, ORDERED
Distinct	-	DISTINCT
Limit	SIZED	-
Peek	-	-
Unordered	ORDERED	-

Caractéristiques d'un Stream

Les caractéristiques d'un Stream sont prise en compte à la consommation

```
// HashSet
HashSet<String> strings = ... ;
strings.stream()
       .distinct() // ne déclenche pas de traitement
       .sorted()
       .collect(Collectors.toList()) ;
```

Implémentations d’un Stream

Complexe

Partagé en deux :

1) partie algorithmique, on n’a pas envie d’y toucher
2) partie accès aux données : faite pour être surchargée !

Apparté sur les Comparator

On a écrit :

```
// interface Comparator
Comparator cmp = Comparator.naturalOrder() ;
```

Apparté sur les Comparator

On a écrit :

```
// interface Comparator
Comparator cmp = Comparator.naturalOrder() ;
```

```
// interface Comparator
@SuppressWarnings("unchecked")
public static <T extends Comparable<? super T>>
Comparator<T> naturalOrder() {

   return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
}
```

Apparté sur les Comparator

```
// classe Comparators
enum NaturalOrderComparator
implements Comparator<Comparable<Object>> {

   INSTANCE;

   public int compare(Comparable<Object> c1, Comparable<Object> c2) {
      return c1.compareTo(c2);
   }

   public Comparator<Comparable<Object>> reversed() {
      return Comparator.reverseOrder();
   }
}
```

Apparté sur les Comparator

On peut écrire :

```
// interface Comparator
Comparator cmp =
   Comparator.comparing(Person::getLastName)
             .thenComparing(Person::getFirstName)
             .thenComparing(Person::getAge) ;
```

Apparté sur les Comparator

Méthode comparing()

```
// interface Comparator
public static
   <T, U> Comparator<T>
   comparing(Function<T, U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Apparté sur les Comparator

Méthode comparing()

```
// interface Comparator
public static
   <T, U extends Comparable<U>> Comparator<T>
   comparing(Function<T, U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return (Comparator<T>)
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Apparté sur les Comparator

Méthode comparing()

```
// interface Comparator
public static
   <T, U extends Comparable<? super U>> Comparator<T>
   comparing(Function<? super T, ? extends U> keyExtractor) {

   Objects.requireNonNull(keyExtractor);
   return (Comparator<T> & Serializable)
      (c1, c2) ->
      keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Apparté sur les Comparator

Méthode thenComparing()

```
// interface Comparator
default
   <U> Comparator<T>
   thenComparing(Function<T, U> keyExtractor) {

   return thenComparing(comparing(keyExtractor));
}
```

Apparté sur les Comparator

Méthode thenComparing()

```
// interface Comparator
default
   <U extends Comparable<? super U>> Comparator<T>
   thenComparing(Function<? super T, ? extends U> keyExtractor) {

   return thenComparing(comparing(keyExtractor));
}
```

Apparté sur les Comparator

Méthode thenComparing()

```
// interface Comparator
default Comparator<T> thenComparing(Comparator<? super T> other) {

   Objects.requireNonNull(other);
   return (Comparator<T> & Serializable) (c1, c2) -> {
      int res = compare(c1, c2);
      return (res != 0) ? res : other.compare(c1, c2);
   };
}
```

Petit bilan

API Stream

- opérations intermédiaires
- opérations terminales
- implémentations à deux niveaux

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
Stream<Persons> stream = persons.limit(1_000) ;
```

… sélectionne les 1000 *premières* personnes

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
Stream<Persons> stream = persons.limit(1_000) ;
```

… sélectionne les 1000 *premières* personnes

Cette opération a besoin d'un compteur

Parallélisme ?

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

« les noms des personnes apparaissent dans le même ordre que les personnes »

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .collect(Collectors.toList()) ;
```

« les noms des personnes apparaissent dans le même ordre que les personnes »

Parallélisme ?

Opérations stateless / stateful

Le code suivant :

```
ArrayList<Person> persons = ... ;
List<String> names =
   persons.map(Person::getLastName)
          .unordered()
          .collect(Collectors.toList()) ;
```

« les noms des personnes apparaissent dans le même ordre que les personnes »

Parallélisme ?

Bilan sur les Stream

Un Stream a un état

On peut définir des opérations sur un Stream :

- intermédiaires & terminales
- stateless & stateful

Les traitements sur un Stream peuvent être parallélisés

Stream & performance

Deux éléments :

- traitements *lazy*
- traitements parallèles

Stream & performance

Deux éléments :

- traitements *lazy*
- traitements parallèles

Stream<T> versus IntStream, LongStream, DoubleStream

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()
       .map(Person::getAge)
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)         // Stream<Integer> re-boxing re-☹
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)         // Stream<Integer> re-boxing re-☹
       .sum() ;                         // pas de méthode sum() sur Stream<T>
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)  // Stream<Integer> re-boxing re-☹
       .mapToInt(age -> age.getValue()) // IntStream ☺
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .map(Person::getAge)             // Stream<Integer> boxing ☹
       .filter(age -> age > 20)  // Stream<Integer> re-boxing re-☹
       .mapToInt(Integer::getValue) // IntStream ☺
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;


persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

int sum =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .sum() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

??? =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .max() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

OptionalInt opt =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .max() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

??? =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .average() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

OptionalInt opt =
persons.stream()                        // Stream<Person>
       .mapToInt(Person::getAge) // IntStream ☺
       .filter(age -> age > 20)  // IntStream ☺
//       .mapToInt(Integer::getValue)
       .average() ;
```

Stream & performance

Retour sur l'exemple

```
ArrayList<Person> persons = ... ;

IntSummaryStatistics stats =
persons.stream()
       .mapToInt(Person::getAge)
       .filter(age -> age > 20)
       .summaryStatistics() ;
```

Calcule en une passe : count, sum, min, max

Et donc aussi average

Parallélisation

Construction d'un Stream parallèle

```
ArrayList<Person> persons = ... ;

Stream<Person> stream1 = persons.parallelStream() ;

Stream<Person> stream2 = persons.stream().parallel() ;
```

Construit sur un Fork / Join construit au niveau de la JVM

Parallélisation

Construction d'un Stream parallèle

```
ArrayList<Person> persons = ... ;

Stream<Person> stream1 = persons.parallelStream() ;

Stream<Person> stream2 = persons.stream().parallel() ;
```

Construit sur un Fork / Join construit au niveau de la JVM

Parallélisation

On peut imposer deux choses :

- le nombre de cœurs sur lequel le FJ Pool va s’installer
- Le FJ Pool lui-même !

Parallélisation

Par défaut les ForkJoinTask s'exécutent dans le « common pool », un FJ Pool créé au niveau de la JVM

Le taux de parallélisme de ce pool est le nombre de cœurs

On peut le fixer :

```
System.setProperty(
   "java.util.concurrent.ForkJoinPool.common.parallelism", 2) ;
```

Parallélisation

On peut aussi imposer le FJ Pool dans lequel les traitements sont faits

```
List<Person> persons = ... ;

ForkJoinPool fjp = new ForkJoinPool(2) ;
fjp.submit(
   () ->                                  //
   persons.stream().parallel()            // implémentation de
          .mapToInt(p -> p.getAge())      // Callable<Integer>
          .filter(age -> age > 20)        //
          .average()                      //
).get() ;
```

Réduction

Généralisons la réduction

Réduction

Généralisons la réduction

Bien sûr, une réduction peut être vue comme une agrégation au sens SQL (sum, min, max, avg, etc…)

Réduction

Généralisons la réduction

Bien sûr, une réduction peut être vue comme une agrégation au sens SQL (sum, min, max, avg, etc…)

On peut voir les choses de façon plus générale

Réduction

Exemple : la somme

Réduction

Exemple : la somme

Un « container » résultat : un entier, de valeur initiale 0

Réduction

Exemple : la somme

Un « container » résultat : un entier, de valeur initiale 0

Une opération « ajouter » : ajoute un élément au container

Une opération « combiner » : fusionner deux containers

utilisée si l'on parallélise le traitement

Réduction

Remarque sur la valeur initiale

Réduction

Remarque sur la valeur initiale

Il s’agira de la valeur retournée en cas de réduction sur un ensemble vide !

Réduction

Remarque sur la valeur initiale

Il s’agira de la valeur retournée en cas de réduction sur un ensemble vide !

… c’est donc l’élément neutre de la réduction …

Réduction

Remarque sur la valeur initiale

Certaines réduction ont donc un problème :

- max,min
- average

Réduction

Donc une réduction peut se définir par trois opérations :

- Création d’un container résultat
- Ajout d’un élément à un container
- Fusion de deux container

Réduction

Modélisation, trois opérations :

- constructeur : Supplier

- accumulateur : Function

- combiner : Function

Réduction

Exemple 1 :

- constructeur : Supplier

```
() -> new StringBuffer() ;
```

- accumulateur : Function

- combiner : Function

Réduction

Exemple 1 :

- constructeur : Supplier

```
() -> new StringBuffer() ;
```

- accumulateur : Function

```
(StringBuffer sb, String s) -> sb.append(s) ;
```

- combiner : Function

Réduction

Exemple 1 :

- constructeur : Supplier

```
() -> new StringBuffer() ;
```

- accumulateur : Function

```
(StringBuffer sb, String s) -> sb.append(s) ;
```

- combiner : Function

```
(StringBuffer sb1, StringBuffer sb2) -> sb1.append(sb2) ;
```

Réduction

Exemple 1 : concaténation de chaîne de caractères

- constructeur : Supplier

```
StringBuffer::new
```

- accumulateur : Function

```
StringBuffer::append
```

- combiner : Function

```
StringBuffer::append
```

Réduction : mise en œuvre

Exemple 1 : accumulation dans un StringBuffer

```
List<Person> persons = ... ;

StringBuffer result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          StringBuffer::new,     // constructor
          StringBuffer::append, // accumulator
          StringBuffer::append  // combiner
       ) ;
```

Réduction : collectors

Exemple 1 : utilisation d'un collector

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining()
       ) ;
```

Réduction

Exemple 2 :

- constructeur : Supplier

```
() -> new ArrayList() ;
```

- accumulateur : Function

- combiner : Function

Réduction

Exemple 2 :

- constructeur : Supplier

```
() -> new ArrayList() ;
```

- accumulateur : Function

```
(ArrayList list, E e) -> list.add(e) ;
```

- combiner : Function

Réduction

Exemple 2 :

- constructeur : Supplier

```
() -> new ArrayList() ;
```

- accumulateur : Function

```
(ArrayList list, E e) -> list.add(e) ;
```

- combiner : Function

```
(ArrayList list1, ArrayList list2) -> list1.addAll(list2) ;
```

Réduction

Exemple 2 : accumulation dans une liste

- constructeur : Supplier

```
ArrayList::new
```

- accumulateur : Function

```
ArrayList::add
```

- combiner : Function

```
ArrayList::addAll
```

Réduction : mise en œuvre

Exemple 2 : accumulation dans une liste

```
List<Person> persons = ... ;

ArrayList<String> names =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          ArrayList::new,    // constructor
          ArrayList::add,    // accumulator
          ArrayList::addAll // combiner
       ) ;
```

Réduction : mise en œuvre

Exemple 2 : accumulation dans une liste

```
List<Person> persons = ... ;

ArrayList<String> names =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          ArrayList::new,   // constructor
          Collection::add,   // accumulator
          Collection::addAll // combiner
       ) ;
```

Réduction : mise en œuvre

Exemple 2 : accumulation dans une liste

```
List<Person> persons = ... ;

HashSet<String> names =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          HashSet::new,   // constructor
          Collection::add,   // accumulator
          Collection::addAll // combiner
       ) ;
```

Réduction : collectors

Exemple 1 : utilisation d'un collector

```
List<Person> persons = ... ;

List<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toList()
       ) ;
```

Réduction : collectors

Exemple 1 : utilisation d'un collector

```
List<Person> persons = ... ;

Set<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toSet()
       ) ;
```

Réduction : collectors

Exemple 1 : utilisation d'un collector

```
List<Person> persons = ... ;

TreeSet<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toCollection(TreeSet::new)
       ) ;
```

Collector

Réduction : collectors

Exemple 1 : utilisation d'un collector

```
List<Person> persons = ... ;

String result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.joining(", ")
       ) ;
```

Réduction : collectors

Exemple 2 : utilisation d'un collector

```
List<Person> persons = ... ;

List<String> result =
persons.stream()
        .filter(person -> person.getAge() > 20)
        .map(Person::getLastName)
        .collect(
           Collectors.toList()
        ) ;
```

Réduction : collectors

Exemple 2 : utilisation d'un collector

```
List<Person> persons = ... ;

Set<String> result =
persons.stream()
       .filter(person -> person.getAge() > 20)
       .map(Person::getLastName)
       .collect(
          Collectors.toSet()
       ) ;
```

Collectors

Classe Collectors : 33 méthodes statiques

Boite à outils !

Collectors : groupingBy

Table de hachage : age / liste des personnes

```
List<Person> persons = ... ;

Map<Integer, List<Person>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(Person::getAge)
        ) ;
```

Collectors : groupingBy

Table de hachage : age / nombre de personnes

```
Map<Integer, Long> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.counting() // « downstream collector »
           )
        ) ;
```

Collectors : groupingBy

Table de hachage : age / liste des noms des personnes

```
Map<Integer, List<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping(     //
                 Person::getLastName // downstream collector
              )                      //
           )
        ) ;
```

Collectors : groupingBy

Table de hachage : age / liste des noms des personnes

```
Map<Integer, String> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping( // 1er downstream collector
                 Person::getLastName
                 Collectors.joining(", ") // 2ème downstream collector
              )
           )
        ) ;
```

Collectors : groupingBy

Table de hachage : age / liste des noms des personnes

```
Map<Integer, TreeSet<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              Collectors.mapping(
                 Person::getLastName
                 Collectors.toCollection(TreeSet::new)
              )
           )
        ) ;
```

Collectors : groupingBy

Table de hachage : age / liste des noms des personnes

```
TreeMap<Integer, TreeSet<String>> result =
persons.stream()
        .collect(
           Collectors.groupingBy(
              Person::getAge,
              TreeMap::new,
              Collectors.mapping(
                 Person::getLastName
                 Collectors.toCollection(TreeSet::new)
              )
           )
        ) ;
```

Bilan intermédiaire

Stream + Collectors =

Nouveaux outils pour le map / filter / reduce

1) exécutions lazy
2) exécutions parallèle

Des exemples à venir…

… mais pour le moment …


Coffee time !
#Stream8
@JosePaumard

Java 8 Streams & Collectors

Patterns, performance, parallélisation

#Stream8

@JosePaumard

Cas d’utilisation

« Houston, we have a problem »

https://github.com/JosePaumard/jdk8-lambda-tour

Collectors : groupingBy

Fichier CSV contenant les MacDo US

```
-149.95038,61.13712,"McDonalds-Anchorage,AK","3828 W Dimond Blvd, Anchorage,AK, (907) 248-0597"
-149.93538,61.18167,"McDonalds-Anchorage,AK","4350 Spenard Rd, Anchorage,AK, (907) 243-5122"
```

http://introcs.cs.princeton.edu/java/data/

Collectors : groupingBy

Fichier CSV contenant les MacDo US

```
-149.95038,61.13712,"McDonalds-Anchorage,AK","3828 W Dimond Blvd, Anchorage,AK, (907) 248-0597"
-149.93538,61.18167,"McDonalds-Anchorage,AK","4350 Spenard Rd, Anchorage,AK, (907) 243-5122"
```

```
try (Stream<String> lines =
        Files.lines(Paths.get("files", "mcdonalds.csv"))) {
    ...
}
```

McDonald : compter les villes

1ère solution

```
long nTowns =
    mcdos.stream()
         .map(McDonald::city) // mcdo -> mcdo.city()
```

McDonald : compter les villes

1ère solution

```
long nTowns =
    mcdos.stream()
         .map(McDonald::city) // mcdo -> mcdo.city()
         .collect(Collectors.toSet())
```

McDonald : compter les villes

1ère solution

```
long nTowns =
    mcdos.stream()
         .map(McDonald::city) // mcdo -> mcdo.city()
         .collect(Collectors.toSet())
         .size() ;
```

McDonald : compter les villes

2ème solution :

```
long nTowns =
    mcdos.stream()
         .map(McDonald::city) // mcdo -> mcdo.city()
         .distinct()
         .collect(Collectors.counting()) ;
```

McDonald : compter les villes

3ème solution :

```
long nTowns =
    mcdos.stream()
         .map(McDonald::city) // mcdo -> mcdo.city()
         .distinct()
         .count() ;
```

McDonald : la ville qui en a le plus

1ère étape : histogramme ville / # de mcdos

C'est une table de hachage !

```
Map<String, Long> map =
    mcdos.stream()
         .collect(
             Collectors.groupingBy(  // table de hachage



             )
        ) ;
```

McDonald : la ville qui en a le plus

1ère étape : histogramme ville / # de mcdos

C'est une table de hachage !

```
Map<String, Long> map =
   mcdos.stream()
        .collect(
            Collectors.groupingBy(
               McDonald::city,        // les clés sont les villes

            )
         ) ;
```

McDonald : la ville qui en a le plus

1ère étape : histogramme ville / # de mcdos

C'est une table de hachage !

```
Map<String, Long> map =
   mcdos.stream()
        .collect(
            Collectors.groupingBy(
               McDonald::city,
               Collectors.counting()  // downstream
            )
         ) ;
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

Il s'agit d'un max

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

Il s'agit d'un max

- Sur quel ensemble ?

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

Il s'agit d'un max

- Sur quel ensemble ?

- Au sens de quel comparateur ?

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

Il s'agit d'un max

- Sur quel ensemble ?
  - L'ensemble des paires clés / valeurs
- Au sens de quel comparateur ?

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

Il s'agit d'un max

- Sur quel ensemble ?
  - L'ensemble des paires clés / valeurs
- Au sens de quel comparateur ?
  - Le comparateur compare les valeurs

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
map.entrySet() // Set des paires clés / valeurs
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .max(       // on prend le max

    ) ;
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .max(        // on prend le max
       Comparator.comparing(entry -> entry.getValue())
    ) ;
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .max(        // on prend le max
       Comparator.comparing(Map.Entry::getValue)
    ) ;
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .max(        // on prend le max
       Map.Entry.comparingByValue() // comparingByKey()
    ) ;
```

McDonald : la ville qui en a le plus

2ème étape : extraire la clé associée à la plus grande valeur

```
Optional<Map.Entry<String, Long>> opt =
map.entrySet() // Set des paires clés / valeurs
   .stream()
   .max(       // on prend le max
      Map.Entry.comparingByValue() // comparingByKey()
   ) ;
```

McDonald : *n* villes qui en ont le plus

Plus compliqué…

1) Trier les paires clés / valeurs

McDonald : *n* villes qui en ont le plus

Plus compliqué…

1) Trier les paires clés / valeurs
2) Prendre les *n* plus grandes

McDonald : la ville qui en a le plus

Tri de l'ensemble des paires clés / valeurs

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .sorted(    // tri
       Comparator.comparing(entry -> -entry.getValue())
    )
```

McDonald : la ville qui en a le plus

Tri de l'ensemble des paires clés / valeurs

```
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .sorted(    // tri
       Comparator.comparing(entry -> -entry.getValue())
    )
    .limit(n)   // n : nombre de valeurs que l'on veut
```

McDonald : la ville qui en a le plus

Tri de l'ensemble des paires clés / valeurs

```
List<Map.Entry<String, Long>> result =
map.entrySet() // Set des paires clés / valeurs
    .stream()
    .sorted(    // tri
       Comparator.comparing(entry -> -entry.getValue())
    )
    .limit(n)   // n : nombre de valeurs que l'on veut
    .collect(Collectors.toList()) ;
```

Cas d’utilisation

« Shakespeare joue au Scrabble »

https://github.com/JosePaumard/jdk8-lambda-tour

Scrabble

Deux fichiers :

- les mots autorisés au Scrabble
- les mots utilisés par Shakespeare

Scrabble

Deux fichiers :

- les mots autorisés au Scrabble
- les mots utilisés par Shakespeare

Un peu de doc (Wikipedia) :

```
private static final int [] scrabbleENScore = {
// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,  q, r, s, t, u, v, w, x, y,  z
   1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10} ;
```

Scrabble : score d’un mot

1ère question : calculer le score d’un mot

```
B O N J O U R
3 1 1 8 1 1 1 // lecture du tableau scrabbleENScore
```

Scrabble : score d’un mot

1ère question : calculer le score d’un mot

```
B O N J O U R
3 1 1 8 1 1 1 // lecture du tableau scrabbleENScore
SUM
= 16
```

Scrabble : score d'un mot

1ère question : calculer le score d'un mot

```
B O N J O U R // stream des lettres du mot « bonjour »
3 1 1 8 1 1 1 // mapping : lettre -> score de chaque lettre
SUM
= 16
```

Scrabble : score d'un mot

1ère question : calculer le score d'un mot

```
B O N J O U R // stream des lettres du mot « bonjour »
3 1 1 8 1 1 1 // mapping : lettre -> score de chaque lettre
sum()
= 16
```

Scrabble : score d'un mot

1ère question : calculer le score d'un mot

```
B O N J O U R // stream des lettres du mot « bonjour »
3 1 1 8 1 1 1 // mapping : lettre -> score de chaque lettre
sum()
= 16
```

```
word -> word.chars() // stream des lettres du mot
            .map(lettre -> scrabbleENScore[lettre - 'a'])
```

Scrabble : score d'un mot

1ère question : calculer le score d'un mot

```
B O N J O U R // stream des lettres du mot « bonjour »
3 1 1 8 1 1 1 // mapping : lettre -> score de chaque lettre
sum()
= 16
```

```
word -> word.chars() // stream des lettres du mot, type IntStream ☺
            .map(lettre -> scrabbleENScore[lettre - 'a'])
            .sum() ; // existe sur IntStream
```

Scrabble : score d'un mot

1ère question : calculer le score d'un mot

```
B O N J O U R // stream des lettres du mot « bonjour »
3 1 1 8 1 1 1 // mapping : lettre -> score de chaque lettre
sum()
= 16
```

```
Function<String, Integer> score =
word -> word.chars() // stream des lettres du mot, type IntStream ☺
            .map(lettre -> scrabbleENScore[lettre - 'a'])
            .sum() ; // existe sur IntStream
```

Scrabble : score d’un mot

1ère question : calculer le score d’un mot

= map / reduce

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

1) Histogramme des mots en fonction de leur score

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

1) Histogramme des mots en fonction de leur score

2) max sur les clés

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

1) Histogramme des mots en fonction de leur score
   - table de hachage : *groupingBy*
2) max sur les clés
   - on sait faire

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

```
shakespeareWords.stream()
    .collect(
       Collectors.groupingBy(
          score                          // fonction de calcul de score
       )
    )
```

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

```
Map<Integer, List<String>> map =
shakespeareWords.stream()
    .collect(
       Collectors.groupingBy(
          score                          // fonction de calcul de score
       )
    ) ;
```

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

```
shakespeareWords.stream()
    .collect(
       Collectors.groupingBy(
          score
       )                                  // Map<Integer, List<String>>
    )
    .entrySet()
    .stream()                             // Map.Entry<Integer, List<String>>
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : score de Shakespeare

2ème question : calculer le meilleur score de Shakespeare

```
Optional<Map.Entry<Integer, List<String>> opt =
shakespeareWords.stream()
    .collect(
       Collectors.groupingBy(
          score
       )                                  // Map<Integer, List<String>>
    )
    .entrySet()
    .stream()                             // Map.Entry<Integer, List<String>>
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : score de Shakespeare

3ème question : limiter aux mots autorisés au Scrabble

```
Optional<Map.Entry<Integer, List<String>> opt =
shakespeareWords.stream()
    .filter(word -> scrabbleWords.contains(word))
    .collect(
       Collectors.groupingBy(
          score
       )                                  // Map<Integer, List<String>>
    )
    .entrySet()
    .stream()                             // Map.Entry<Integer, List<String>>
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : score de Shakespeare

3ème question : limiter aux mots autorisés au Scrabble

```
Optional<Map.Entry<Integer, List<String>> opt =
shakespeareWords.stream()
    .filter(scrabbleWords::contains)
    .collect(
       Collectors.groupingBy(
          score
       )                                  // Map<Integer, List<String>>
    )
    .entrySet()
    .stream()                             // Map.Entry<Integer, List<String>>
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : score de Shakespeare

3ème question : limiter aux mots autorisés au Scrabble

= problème de filter / reduce

Scrabble : score de Shakespeare

Question : le mot en question est-il possible au Scrabble ?

Scrabble : score de Shakespeare

Question : le mot en question est-il possible au Scrabble ?

```
private static final int [] scrabbleENDistribution = {
// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,  q, r, s, t, u, v, w, x, y, z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Il n’y a qu’un seul ‘Z’ au Scrabble anglais…

Scrabble : score de Shakespeare

Question : le mot en question est-il possible au Scrabble ?

```
private static final int [] scrabbleENDistribution = {
// a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p,  q, r, s, t, u, v, w, x, y,  z
   9, 2, 2, 1, 12, 2, 3, 2, 9, 1, 1, 4, 2, 6, 8, 2, 1, 6, 4, 6, 4, 2, 2, 1, 2, 1} ;
```

Il n’y a qu’un seul ‘Z’ au Scrabble anglais…

Il faut vérifier le nombre d’utilisation de chaque lettre

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
B U Z Z A R D // le mot
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre -> groupingBy
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre
2 4 1 9 6 1   // les lettres disponibles
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre
2 4 1 9 6 1   // les lettres disponibles
T T F T T T   // il faut que tout soit à TRUE -> allMatch
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
word -> word.chars()                          // IntStream ☹
            .mapToObj(Integer::new)           // Stream<Integer>
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
word -> word.chars()                          // IntStream ☹
             .mapToObj(Integer::new)          // Stream<Integer>
             .collect(
                Collectors.groupingBy(



                )
             ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
word -> word.chars()                          // IntStream ☹
            .mapToObj(Integer::new)           // Stream<Integer>
            .collect(                         // Map<Integer, Long>
               Collectors.groupingBy(
                  letter -> letter,
                  Collectors.counting()
               )
            ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
word -> word.chars()                          // IntStream ☹
            .mapToObj(Integer::new)           // Stream<Integer>
            .collect(                         // Map<Integer, Long>
               Collectors.groupingBy(
                  Function.identity(),
                  Collectors.counting()
               )
            ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Function<String, Map<Integer, Long>> letterHisto =
    word -> word.chars()                          // IntStream ☹
                .mapToObj(Integer::new)           // Stream<Integer>
                .collect(                         // Map<Integer, Long>
                   Collectors.groupingBy(
                      Function.identity(),
                      Collectors.counting()
                   )
                ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Predicate<String> canWrite =
word -> letterHisto
    .apply(word) // Map<Integer, Long>
    .entrySet()
    .stream()    // Map.Entry<Integer, Long>
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Predicate<String> canWrite =
word -> letterHisto
    .apply(word) // Map<Integer, Long>
    .entrySet()
    .stream()    // Map.Entry<Integer, Long>
    .allMatch(




    ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Predicate<String> canWrite =
word -> letterHisto
    .apply(word) // Map<Integer, Long>
    .entrySet()
    .stream()    // Map.Entry<Integer, Long>
    .allMatch(
       entry ->


    ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Predicate<String> canWrite =
word -> letterHisto
    .apply(word) // Map<Integer, Long>
    .entrySet()
    .stream()    // Map.Entry<Integer, Long>
    .allMatch(
       entry ->
          entry.getValue() <=                    // besoin en lettres

    ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Predicate<String> canWrite =
word -> letterHisto
    .apply(word) // Map<Integer, Long>
    .entrySet()
    .stream()    // Map.Entry<Integer, Long>
    .allMatch(
       entry ->
          entry.getValue() <=                    // besoin en lettres
          scrabbleENDistrib[entry.getKey() - 'a'] // lettres dispos
    ) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

```
Optional<Map.Entry<Integer, List<String>> opt =
shakespeareWords.stream()
    .filter(scrabbleWords::contains)
    .filter(canWrite)
    .collect(
       Collectors.groupingBy(
          score
       )                                  // Map<Integer, List<String>>
    )
    .entrySet()
    .stream()                             // Map.Entry<Integer, List<String>>
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : mot possible ?

4ème question : peut-on écrire ce mot ?

= problème de map / reduce

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Deux impacts :

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Deux impacts :

- impact sur les mots que l'on peut faire

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Deux impacts :

- impact sur les mots que l'on peut faire
- impact sur le calcul du score

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Comment déterminer si un mot est accepté ?

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Comment déterminer si un mot est accepté ?

Critère : pas plus de deux blancs

Scrabble : et si l'on prend les blancs ?

5ème question : et si l'on utilise les blancs disponibles ?

Comment déterminer si un mot est accepté ?

Critère : pas plus de deux blancs

On peut écrire une fonction qui calcule le nombre de blancs utilisés pour un mot

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre
2 4 1 9 6 1   // les lettres disponibles
```

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre
2 4 1 9 6 1   // les lettres disponibles
0 0 1 0 0 0   // nombre de blancs nécessaires
              // il s'agit d'un max !
```

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

```
B U Z Z A R D // le mot
B U Z A R D   // les lettres utilisées
1 1 2 1 1 1   // l'occurrence de chaque lettre
2 4 1 9 6 1   // les lettres disponibles
0 0 1 0 0 0   // nombre de blancs nécessaires
              // il s'agit d'un max !
```

```
Integer.max(
    0,
    entry.getValue() -                          // lettres nécessaires
    scrabbleEnDistrib[entry.getKey() - 'a'] // lettres dispo
```

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

```
Function<String, Integer> nBlanks =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(          // IntStream -> sum() dispo
                  Integer.max(
                     0,
                     entry.getValue() -
                     scrabbleEnDistrib[entry.getKey() - 'a']
                  )
               )
               .sum() ;
```

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

```
Optional<Map.Entry<Integer, List<String>> opt =
shakespeareWords.stream()
    .filter(scrabbleWords::contains)
    .filter(word -> nBlanks.apply(word) <= 2)
    .collect(
       Collectors.groupingBy(
          score
       )
    )
    .entrySet()
    .stream()
    .max(Map.Entry.comparingByKey()) ;
```

Scrabble : mot possible ?

5ème question : et si l'on utilise les blancs disponibles ?

= problème de map / reduce

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

= problème de map / reduce

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
B U Z  Z A R D // le mot
B U Z  A R D   // les lettres utilisées
1 1 2  1 1 1   // l'occurrence de chaque lettre
2 4 1  9 6 1   // les lettres disponibles
1 1 1  1 1 1   // lettres effectivement utilisées
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
B U Z  Z A R D // le mot
B U Z  A R D   // les lettres utilisées
1 1 2  1 1 1   // l'occurrence de chaque lettre
2 4 1  9 6 1   // les lettres disponibles
1 1 1  1 1 1   // lettres effectivement utilisées
               // il s'agit d'un min !
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
B U Z  Z A R D // le mot
B U Z  A R D   // les lettres utilisées
1 1 2  1 1 1   // l'occurrence de chaque lettre
2 4 1  9 6 1   // les lettres disponibles
1 1 1  1 1 1   // lettres effectivement utilisées
* * *  * * *
3 1 10 1 1 2   // score individuel de chaque lettre
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
B U Z  Z A R D // le mot
B U Z  A R D   // les lettres utilisées
1 1 2  1 1 1   // l'occurrence de chaque lettre
2 4 1  9 6 1   // les lettres disponibles
1 1 1  1 1 1   // lettres effectivement utilisées
* * *  * * *
3 1 10 1 1 2   // score individuel de chaque lettre
SUM
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
Function<String, Integer> score =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(









               .sum() ;
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
Function<String, Integer> score =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(
                  entry ->









               .sum() ;
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
Function<String, Integer> score =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(
                  entry ->
                  scrabbleENScore[entry.getKey() - 'a']* // score de
                                                         // la lettre



               .sum() ;
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

```
Function<String, Integer> score =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(
                  entry ->
                  scrabbleENScore[entry.getKey() - 'a']*
                  Integer.min(



                  )
               .sum() ;
```

Scrabble : score avec les blancs ?

6ème question : compter les points avec les blancs ?

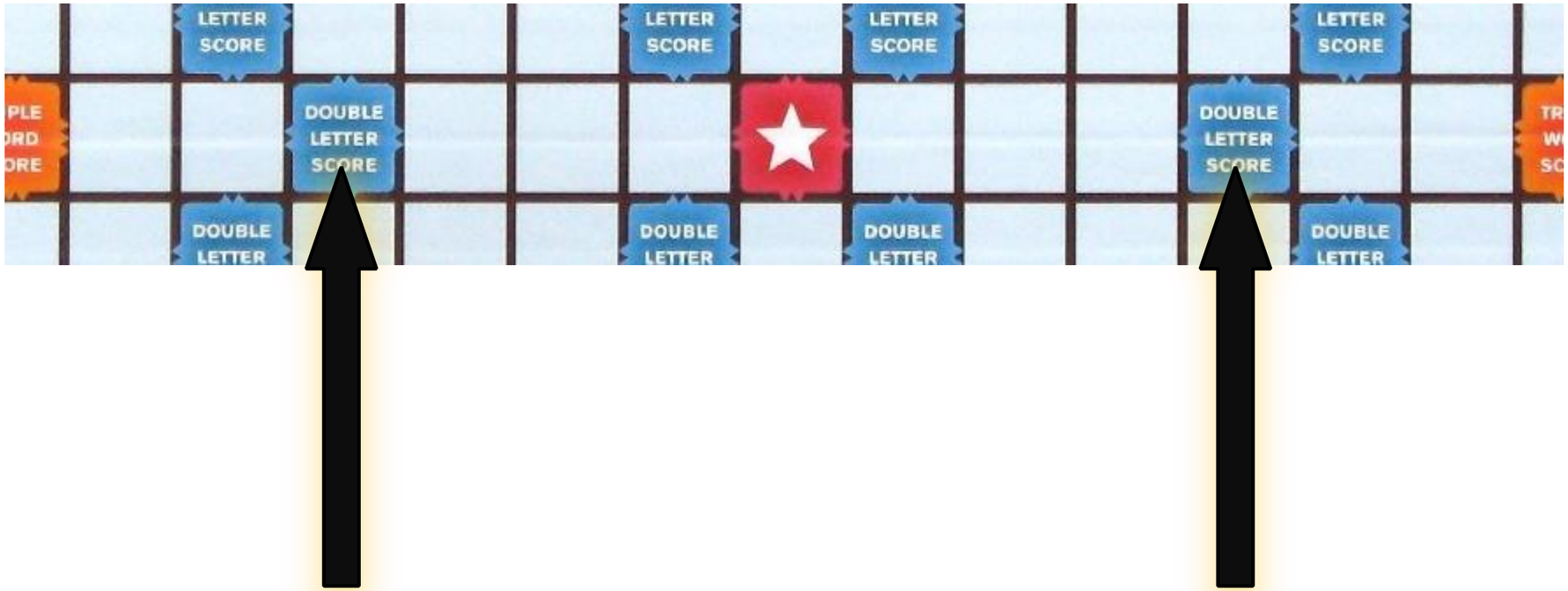
```
Function<String, Integer> score =
   word -> letterHisto.apply(word)
               .entrySet()
               .stream()
               .mapToInt(
                  entry ->
                  scrabbleENScore[entry.getKey() - 'a']*
                  Integer.min(
                     entry.getValue(), // nombre de lettres dans le mot
                     scrabbleENDistrib[entry.getKey() - 'a']
                  )
               .sum() ;
```

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

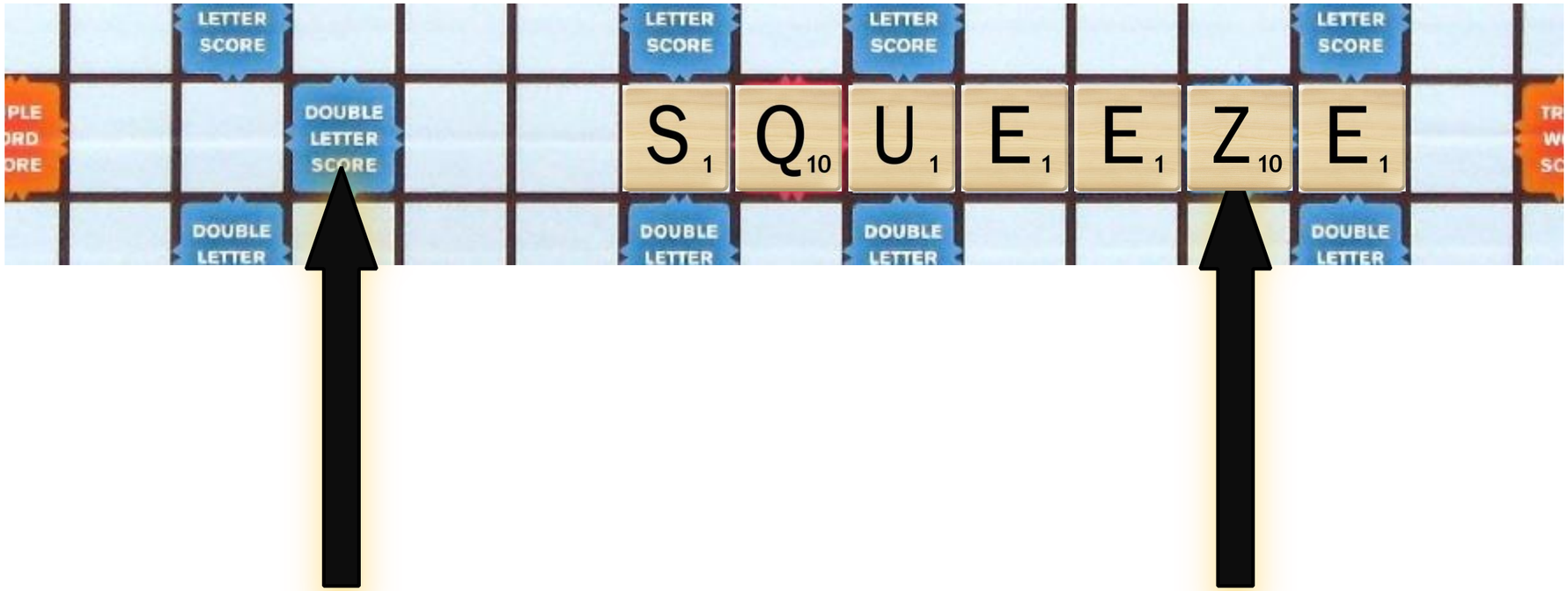
Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?



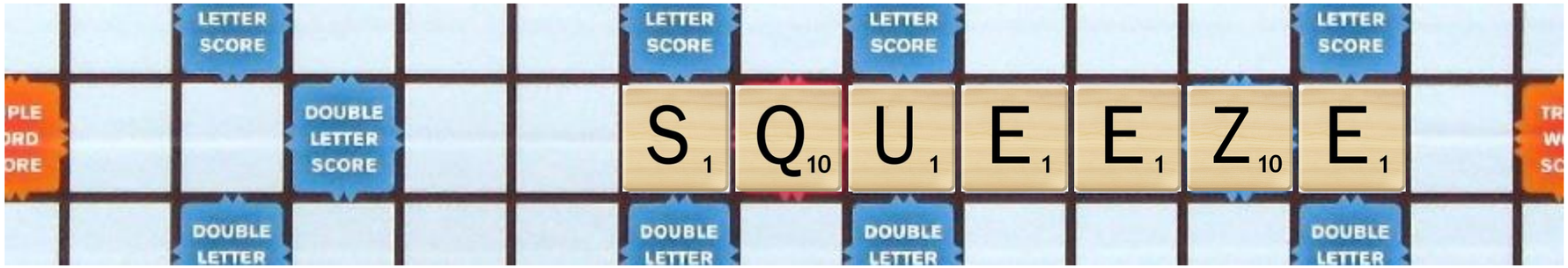
Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?



Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

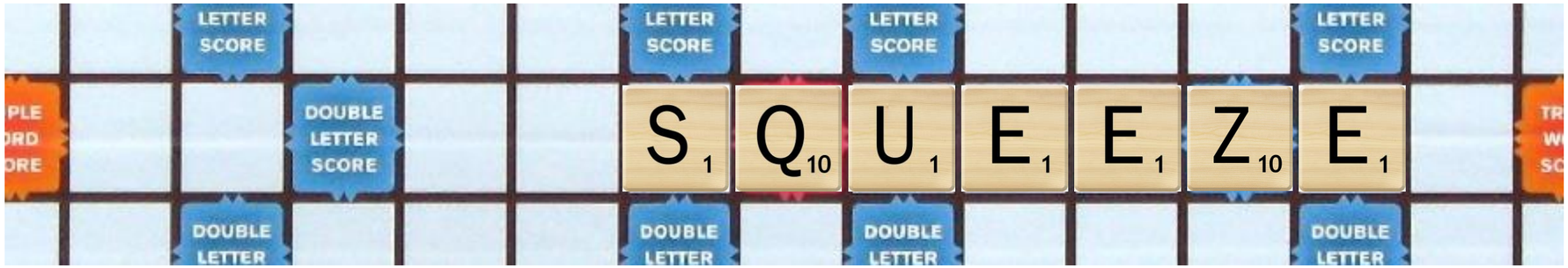


6 solutions pour poser le mot :

- utilisation de la case de droite = 3 lettres de la fin du mot
- utilisation de la case de gauche = 3 lettres du début

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?



Solution :

- prendre le max des 3 dernières lettres
- avec le max des 3 premières lettres

…si le mot fait 7 lettres !

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

Traduit en Java, le max est pris sur que ensemble ?

```
word.chars().skip(4)                                    // 1er flux
word.chars().limit(Integer.max(0, word.length() - 4)) // 2ème flux
```

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

Traduit en Java, le max est pris sur que ensemble ?

```
word.chars().skip(4)                                    // 1er flux
word.chars().limit(Integer.max(0, word.length() - 4)) // 2ème flux
```

Le max de ces deux flux, puis le max des ces deux max

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

Traduit en Java, le max est pris sur que ensemble ?

```
word.chars().skip(4)                                    // 1er flux
word.chars().limit(Integer.max(0, word.length() - 4)) // 2ème flux
```

Concaténer ces flux ?

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

```
IntStream.concat(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.map(scrabbleENScore)
.max()
```

Scrabble : et sur le plateau de jeu ?

7ème question : le « lettre compte double » ?

```
IntStream.concat(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.map(scrabbleENScore)
.max()
```

Problème : concat ne se parallélise pas bien…

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Stream.of(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
) // retourne un Stream de Stream !
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Stream.of(
    word.chars().skip(4)
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Stream.of(
    word.chars().skip(4)
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
```

Que faire de cet Optional ?

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Stream.of(
    word.chars().skip(4)
    word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
```

Que faire de cet Optional ? … qui peut être vide !

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Stream.of(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
.ifPresent(          )
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
List<Integer> list = new ArrayList<>() ;

Stream.of(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
.ifPresent(list::add)
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
List<Integer> list = new ArrayList<>() ;

Stream.of(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
.ifPresent(list::add) ;
list.stream().max(Comparator.naturalOrder()).get()
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
List<Integer> list = new ArrayList<>() ;
list.add(0) ;
Stream.of(
   word.chars().skip(4)
   word.chars().limit(Integer.max(0, word.length() - 4))
)
.flatMap(Function.identity())
.map(scrabbleENScore)
.max()
.ifPresent(list::add) ;
list.stream().max(Comparator.naturalOrder()).get()
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

```
Function<String, Integer> scoreAtPosition =
   word -> {
      List<Integer> list = new ArrayList<>() ;
      list.add(0) ;
      Stream.of(
         word.chars().skip(4)
         word.chars().limit(Integer.max(0, word.length() - 4))
      )
      .flatMap(Function.identity()).map(scrabbleENScore).max()
      .ifPresent(list::add) ;
      return list.stream().max(Comparator.naturalOrder()).get()
   }
```

Scrabble : et sur le plateau de jeu ?

7ème question : « lettre compte double » ?

= problème de map / reduce !

Cas d’utilisation

« des films et des acteurs »

https://github.com/JosePaumard/jdk8-lambda-tour

Des films et des acteurs

Fichier : « movies-mpaa.txt »

Contient 14k films américains de 1916 à 2004

Avec :

- le titre
- l’année de sortie
- la liste des acteurs

Des films et des acteurs

Fichier : « movies-mpaa.txt »

Contient 14k films américains de 1916 à 2004

Avec :

- le titre
- l‘année de sortie
- la liste des acteurs

Référence 170k acteurs différents

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

Construire une table de hachage :

- les clés sont les acteurs
- les valeurs sont le nombre de films

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

Construction de l'ensemble des acteurs

```
Set<Actor> actors =
   movies.stream()
   .flatMap(movie -> movie.actors().stream())
   .collect(Collector.toSet()) ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
actors.stream()
 .collect(









 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
actors.stream()
 .collect(
    Collectors.toMap(




    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
actors.stream()
 .collect(
    Collectors.toMap(
       Function.identity(),




    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
actors.stream()
 .collect(
    Collectors.toMap(
       Function.identity(),
       actor -> movies.stream()
                      .filter(movie -> movie.actors().contains(actor))
                      .count()
    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
actors.stream().parallel() // T = 40s ☹
 .collect(
    Collectors.toMap(
       Function.identity(),
       actor -> movies.stream()
                      .filter(movie -> movie.actors().contains(actor))
                      .count()
    )
 )
 .entrySet().stream()
 .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
movies.stream()
    .map(movie -> movie.actors().stream()) // stream de stream !
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream d'acteurs
    .collect(




    )
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream d'acteurs
    .collect(
       Collectors.groupingBy(
          Function.identity(),
          Collectors.counting()
       )
    )
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream d'acteurs
    .collect(
       Collectors.groupingBy(
          Function.identity(),
          Collectors.counting()
       )
    )
    .entrySet().stream()
    .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

Des films et des acteurs

1ère question : quel acteur a joué dans le plus de films ?

Réponse : Frank Welker

Des films et des acteurs

2ème question : idem, en une année ?

Des films et des acteurs

2ème question : idem, en une année ?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream d'acteurs
    .collect(
       Collectors.groupingBy(
          Function.identity(),
          Collectors.counting()
       )
    )
    .entrySet().stream()
    .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

2ème question : idem, en une année ?

```
movies.stream()
    .flatMap(movie -> movie.actors().stream()) // stream d'acteurs
    .collect(
       Collectors.groupingBy(
          Function.identity(),
          // construire une table de hachage année / nombre
       )
    )
    .entrySet().stream()
    .max(Map.Entry.comparingByValue()).get() ;
```

Des films et des acteurs

2ème question : idem, en une année ?

On peut construire une table de hachage :

- les clés sont les années
- les valeurs sont … des tables de hachage

Des films et des acteurs

2ème question : idem, en une année ?

On peut construire une table de hachage :

- les clés sont les années
- les valeurs sont … des tables de hachage
    - dont les clés sont les acteurs
    - les valeurs sont le nombre de films

Des films et des acteurs

2ème question : idem, en une année ?

On peut construire une table de hachage :

- les clés sont les années
- les valeurs sont … des tables de hachage
    - dont les clés sont les acteurs
    - les valeurs sont le nombre de films

Et on veut le max par nombre de films

Des films et des acteurs

2ème question : idem, en une année ?

On va construire le collector à la main !

```
movies.stream().collect(
    Collector.groupingBy(
        movie -> movie.releaseYear(),
        // downstream construit à la main
    )
)
```

Des films et des acteurs

2ème question : idem, en une année ?

Ce collector construit une table de hachage :

- les clés sont les acteurs
- les valeurs sont le nombre de films joués dans l'année

Des films et des acteurs

2ème question : idem, en une année ?

1) la construction de la table résultat

```
Supplier<Map<Actor, AtomicLong>> supplier =
    () ->
    new HashMap<Actor, AtomicLong>() ;
```

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

```
BiConsumer<Map<Actor, AtomicLong>, Movie> accumulator =
   (map, movie) -> movie.actors().forEach(
      actor ->


   ) ;
```

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

```
BiConsumer<Map<Actor, AtomicLong>, Movie> accumulator =
   (map, movie) -> movie.actors().forEach(
      actor ->
         map.get(actor).incrementAndGet()  // map.getActor != null

   ) ;
```

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

```
BiConsumer<Map<Actor, AtomicLong>, Movie> accumulator =
   (map, movie) -> movie.actors().forEach(
      actor ->
         map.put(actor, new AtomicLong())  // map.getActor == null

   ) ;
```

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

```
BiConsumer<Map<Actor, AtomicLong>, Movie> accumulator =
   (map, movie) -> movie.actors().forEach(
      actor ->  map.computeIfAbsent(
            actor, // la clé
          // exécute ce code si pas de clé
        ) // et retourne la valeur
        .incrementAndGet()
   ) ;
```

Des films et des acteurs

2ème question : idem, en une année ?

2) l'ajout d'un élément à une table partiellement remplie

```
BiConsumer<Map<Actor, AtomicLong>, Movie> accumulator =
   (map, movie) -> movie.actors().forEach(
      actor ->  map.computeIfAbsent(
            actor,
            a -> new AtomicLong() // exécute ce code si absent
         ) // et retourne la valeur
         .incrementAndGet()
   ) ;
```

Des films et des acteurs

2ème question : idem, en une année ?

3) la fusion de deux tables partiellement remplies

Des films et des acteurs

2ème question : idem, en une année ?

3) la fusion de deux tables partiellement remplies

```
BinaryOperator<Map<Actor, AtomicLong>> combiner =
    (map1, map2) -> {









        return map1 ;
    } ;
```

Des films et des acteurs

2ème question : idem, en une année ?

3) la fusion de deux tables partiellement remplies

```
BinaryOperator<Map<Actor, AtomicLong>> combiner =
    (map1, map2) -> {
        map2.entrySet().stream().forEach(




        ) ;
        return map1 ;
    } ;
```

Des films et des acteurs

2ème question : idem, en une année ?

3) la fusion de deux tables partiellement remplies

```
BinaryOperator<Map<Actor, AtomicLong>> combiner =
   (map1, map2) -> {
      map2.entrySet().stream().forEach(
         entry -> map1.computeIfAbsent(
                     entry.getKey(), a -> new AtomicLong()
                  )
      ) ;
      return map1 ;
   } ;
```

Des films et des acteurs

2ème question : idem, en une année ?

3) la fusion de deux tables partiellement remplies

```
BinaryOperator<Map<Actor, AtomicLong>> combiner =
   (map1, map2) -> {
      map2.entrySet().stream().forEach(
         entry -> map1.computeIfAbsent(
                     entry.getKey(), a -> new AtomicLong()
                  ).addAndGet(entry.getValue().get())
      ) ;
      return map1 ;
   } ;
```

Des films et des acteurs

2ème question : idem, en une année ?

```
movies.stream().collect(
    Collector.groupingBy(
        movie -> movie.releaseYear(),
        // downstream construit à la main
        Collector.of(
            supplier, accumulator, combiner,




        )
    )
) // Map<Integer, Map<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

```
movies.stream().collect(
    Collector.groupingBy(
        movie -> movie.releaseYear(),
        // downstream construit à la main
        Collector.of(
            supplier, accumulator, combiner,
            new Collector.Characteristics [] {
                Collector.Characteristics.CONCURRENT.CONCURRENT
            }
        )
    )
) // Map<Integer, Map<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

```
movies.stream().collect(
    Collector.groupingBy(
        movie -> movie.releaseYear(),
        // downstream construit à la main
        Collector.of(
            supplier, accumulator, combiner,
            new Collector.Characteristics [] {
                Collector.Characteristics.CONCURRENT.CONCURRENT
            }
        )
    )
) // Map<Integer, Map<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

Dernière étape : extraire le max

```
// Map<Integer, Map<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

Dernière étape : extraire le max

```
// Map<Integer, Map<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

Dernière étape : extraire le max

```
// Map<Integer, Map<Actor, AtomicLong>>
```

1) extraire Map<Integer, Map.Entry<Actor, AtomicLong>>

Des films et des acteurs

2ème question : idem, en une année ?

Dernière étape : extraire le max

```
// Map<Integer, Map<Actor, AtomicLong>>
```

1) extraire Map<Integer, Map.Entry<Actor, AtomicLong>>
2) extraire le max de cette table

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map<Actor, AtomicLong>>
map.entrySet().stream().collect(
    Collectors.toMap(
        entry -> entry.getKey(), // on ne change pas la clé
                                 // Map.Entry<Actor, AtomicLong>
    )
)
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map<Actor, AtomicLong>>
map.entrySet().stream().collect(
    Collectors.toMap(
        entry -> entry.getKey(),
        entry -> entry.getValue().entrySet().stream()
                      .max(


                      )
                      .get()
    )
)
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map<Actor, AtomicLong>>
map.entrySet().stream().collect(
    Collectors.toMap(
        entry -> entry.getKey(),
        entry -> entry.getValue().entrySet().stream()
                    .max(
                        Map.Entry.comparingByValue(

                    )
                    .get()
    )
)
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map<Actor, AtomicLong>>
map.entrySet().stream().collect(
    Collectors.toMap(
        entry -> entry.getKey(),
        entry -> entry.getValue().entrySet().stream()
                    .max(
                        Map.Entry.comparingByValue(
                            Comparator.comparing(AtomicLong::get))
                    )
                    .get()
    )
)
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map<Actor, AtomicLong>>
map.entrySet().stream().collect(
    Collectors.toMap(
        entry -> entry.getKey(),
        entry -> entry.getValue().entrySet().stream()
                    .max(
                        Map.Entry.comparingByValue(
                            Comparator.comparing(AtomicLong::get))
                    )
                    .get()
    )
) // Map<Integer, Map.Entry<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map.Entry<Actor, AtomicLong>>
map2.entrySet().stream()
     .max(
        Comparator.comparing(

        )
     )
     .get() ;
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map.Entry<Actor, AtomicLong>>
map2.entrySet().stream()
    .max(
       Comparator.comparing(
          entry -> entry.getValue().getValue().get())
       )
    )
    .get() ;
```

Des films et des acteurs

2ème question : idem, en une année ?

```
// Map<Integer, Map.Entry<Actor, AtomicLong>>
map2.entrySet().stream()
    .max(
       Comparator.comparing(
          entry -> entry.getValue().getValue().get())
       )
    )
    .get() ; // Map.Entry<Integer, Map.Entry<Actor, AtomicLong>>
```

Des films et des acteurs

2ème question : idem, en une année ?

Des films et des acteurs

2ème question : idem, en une année ?

Réponse : Phil Hawn, en 1999
a joué dans 24 films

Cas d’utilisation

https://github.com/JosePaumard/jdk8-lambda-tour

Conclusion

Streams + Collectors = obsolescence du pattern Iterator !

Conclusion

Streams + Collectors = obsolescence du pattern Iterator !

Combinés avec les lambdas =
nouvelle façon d’écrire les applications en Java

Conclusion

Streams + Collectors = obsolescence du pattern Iterator !

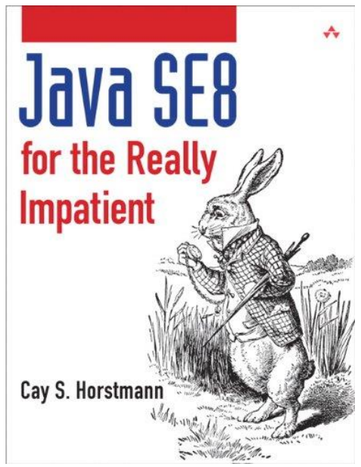
Combinés avec les lambdas =

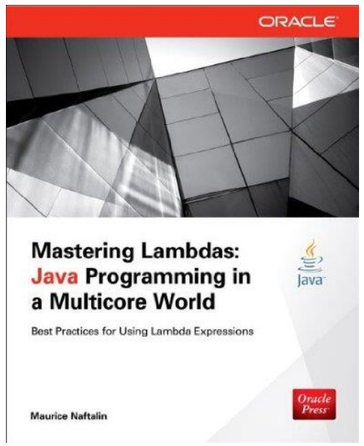
nouvelle façon d’écrire les applications en Java

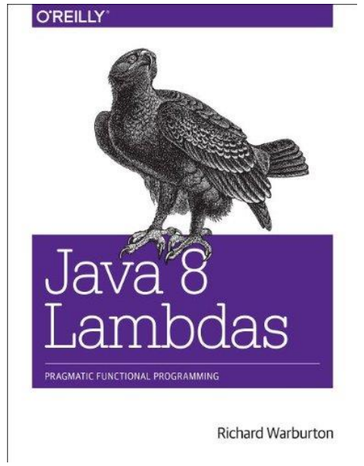
- le développement
- l’architecture

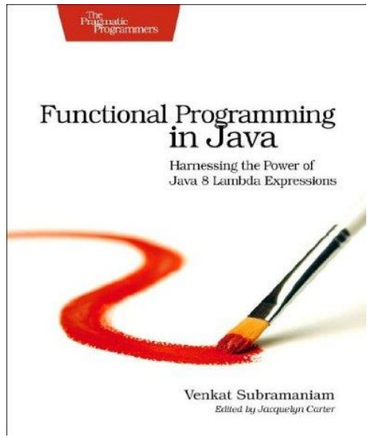
Conclusion

Un peu de doc :










Merci !
#Stream8
@JosePaumard


Q/R
#Stream8
@JosePaumard