Stream

Rémi Forax

Opération ensembliste sur des collections

Exemple, avec un record et une liste

```
public record Person(String name, int age) {}
var persons = List.of(new Person("Ana", 23), …);
```

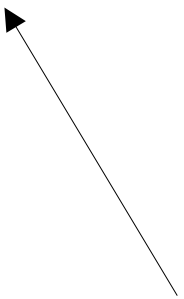
On veut répondre à une question comme

- Quel est la plus jeune personne dont le nom commence par “A”
  - Un peu comme en SQL

Avant Java 8

Quelle est la plus jeune personne dont le nom commence par “A” ?

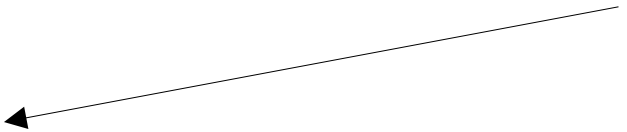
```
Person min = null;
for(var person: persons) {  // utilise un itérateur
  if (person.name().starsWith("A")) {
    if (min == null || person.age() < min.age()) {
      min = person;
    }
  }
}
return min;
```



En termes de maintenance,
le code est assez loin de la question
car on explique comment on fait pour obtenir ce que l’on veut et pas ce que l’on veut

Avec l'API des Stream

Quelle est la plus jeune personne dont le nom commence par "A" ?




```
Optional<Person> result =
    persons.stream()   // on utilise un Stream
      .filter(p - > p.name().startsWith("A"))
      .min(Comparator.comparingInt(Person::age));
```

L'API des Stream est déclarative, on dit le résultat que l'on veut, pas comment on l'obtient

- Plus facile pour la maintenance du code

Plusieurs APIs

Les streams utilisent plusieurs API

- L'API des Stream (java.util.stream)
  - Opération ensembliste
- L'API des Collector (java.util.stream)
  - Pour mettre le résultat dans une collection
- L'API java.util.Optional
  - Abstraction de null + véritable API
- L'API java.util.Comparator
  - Créer des comparateurs et les composer

API des Stream

Opérations sur les Stream

Trois étapes

- Création d'un Stream
- Transformations successives du Stream (opérations intermédiaires)
- Demander le résultat (opération finale)

Les opérations sur les streams sont paresseuses

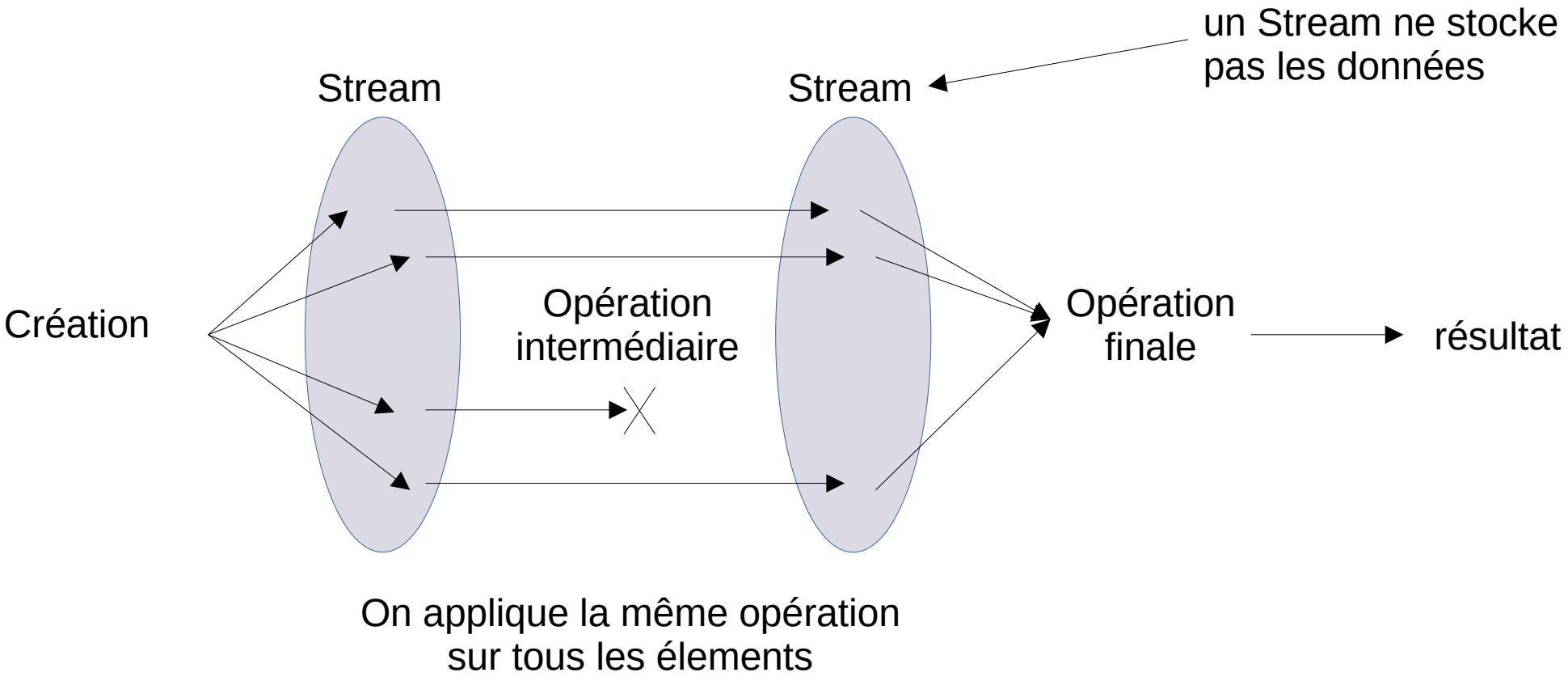
on ne fait pas le calcul si cela n'est pas nécessaire

par ex: si on n'appelle pas d'opération finale,
aucun calcul n'est fait

Pipeline

Un Stream est organisé comme un pipeline

Les données sont poussées (*push*) dans le pipeline



Creation d’un Stream

Créer un Stream

A partir d’une collection

```
collection.stream()
```

A partir de valeurs

```
Stream.of(1, 2, 3, …)
```

A partir d’un tableau

```
Array.stream(tableau)
```

Stream de type primitif

Pour éviter le boxing, il existe trois Stream spécialisés pour les types primitifs

- IntStream, LongStream, DoubleStream

par ex: IntStream

- IntStream.range(start, end)
  - renvoie toutes les valeurs de start à end (non compris)

Stream sur des ressources système

Les Streams sur des ressources systèmes doivent être fermés avec close()

on utilise le *try-with-resources*

- Files.lines(path) renvoie les lignes d'un fichier

```
try(var stream = Files.lines(path)) {
  stream. ...
}
```

- Files.list(path) renvoie les fichiers d'un répertoire

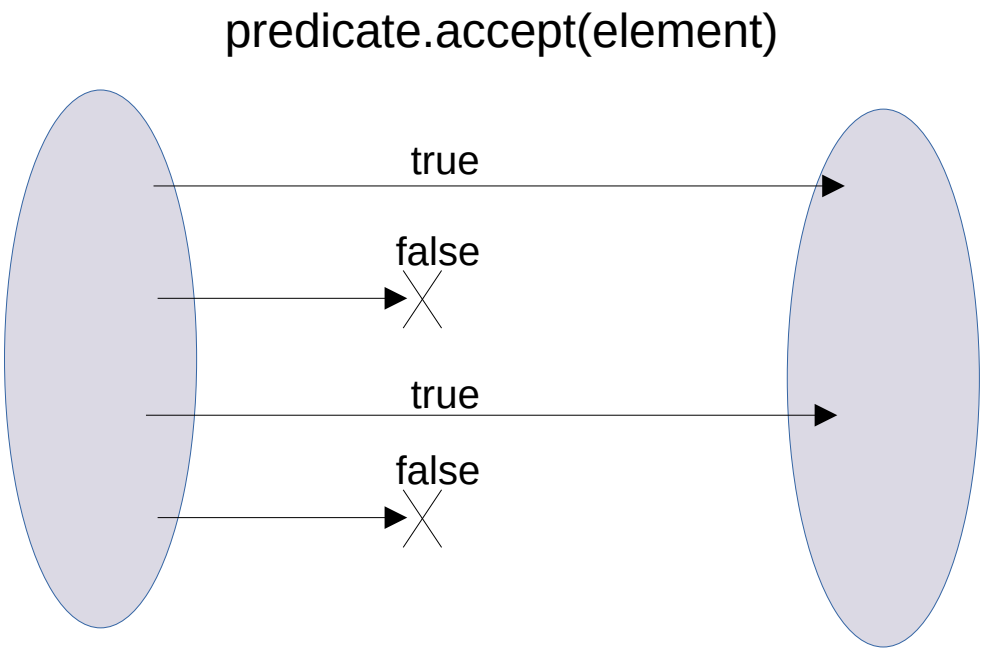
```
try(var stream = Files.list(path)) {
  stream. ...
}
```

Opérations intermédiaires

filter(Predicate<E>)

On appelle un prédicat (E → boolean) et on ne propage pas l'objet si le prédicat renvoie false

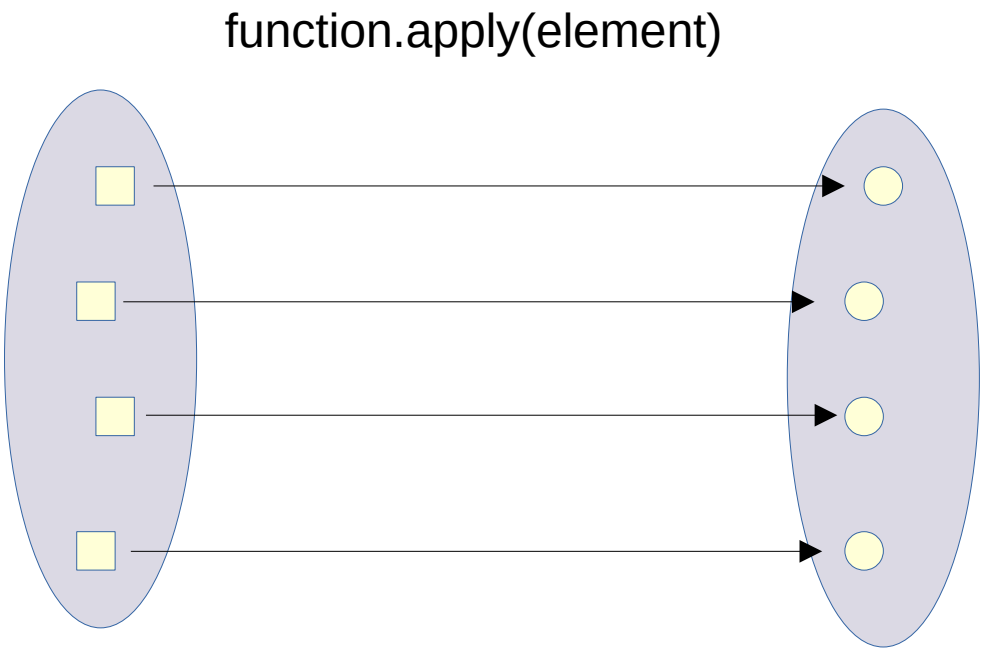
```
stream.filter(person - > person.age() >= 18)
```



map(Function<E,R>)

Transforme chaque élement en appelant la fonction

```
stream.map(Person::name)
```

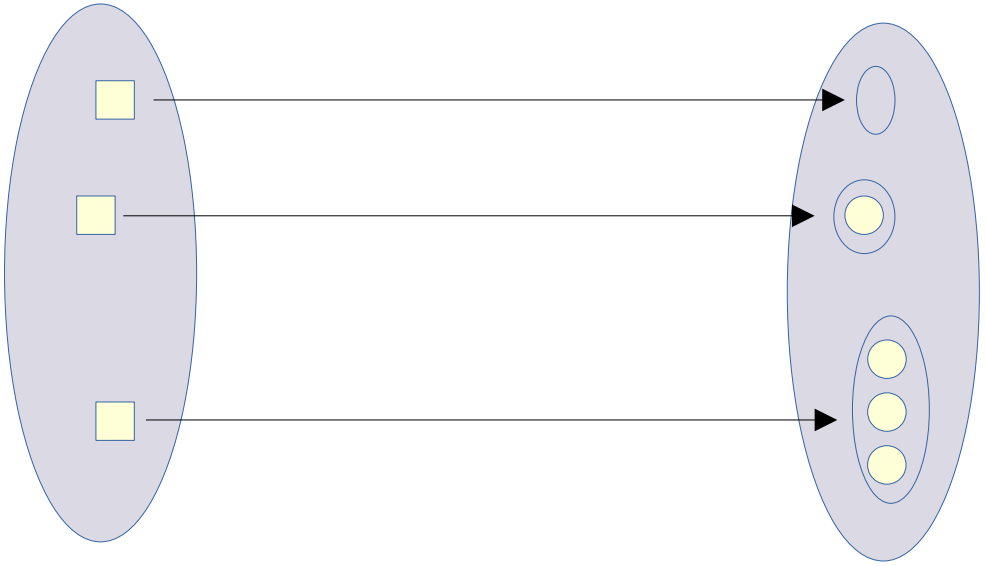


flatMap(Function<E, Stream<R>>)

Transforme chaque élement en appelant la fonction

```
stream.map(p - >
        Stream.of(p, new Person(p.name(), p.age() + 1)))
```

function.apply(element)



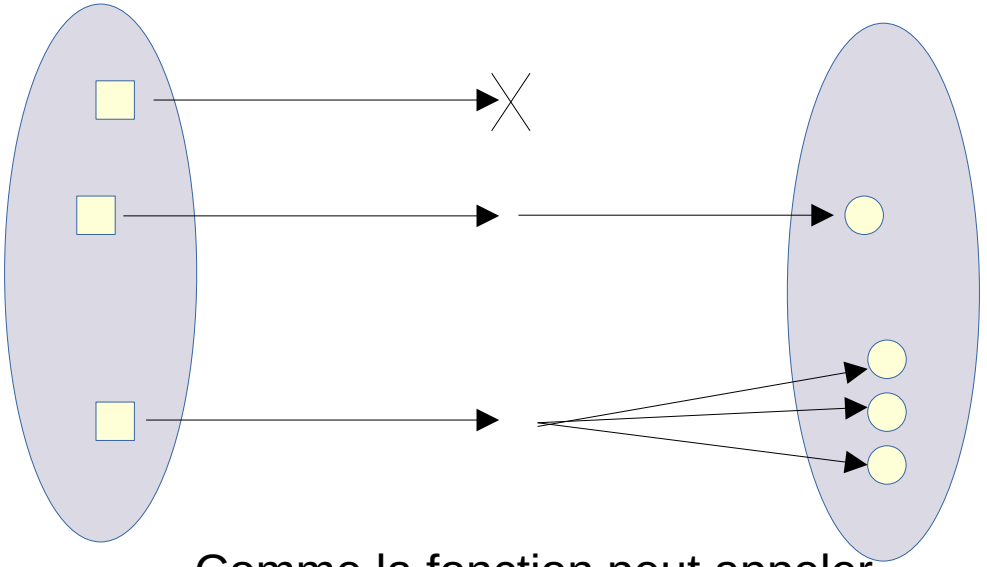
Comme la fonction renvoie un Stream,
elle renvoie de 0 à n élements

mapMulti(BiConsumer<E,Consumer<R>>)

Pour chaque objet, peut propager zéro ou plusieurs objets

```
stream.mapMulti((person, sink) - > {
  sink.accept(person);
  sink.accept(new Person(person.name(), person.age() + 1));
})
```

biConsumer.accept(element, sink)



Comme la fonction peut appeler
le consommateur autant de fois qu'elle
le souhaite, elle propage de 0 à n élements

Autre opérations intermédiaires

.**limit**(value)

Filtre qui ne laisse passer que *value* élements

.**skip**(value)

Saute les *value* premiers élements

.**distinct**()

Filtre qui ne laisse pas passer la même valeur deux fois

.**sorted**(comparateur)

Trie les élements et les propage triés

map/flatMap/mapMulti et primitif

Les opérations intermédiaires map, flatMap et mapMulti ont des versions particulières pour les int, long et double (pour éviter le boxing)

Par ex:

stream.**mapToInt**(Person::age) → IntStream

alors que

Stream.map(Person::age) → Stream<Integer>

Opérations finales

toList() / toArray() / findFirst()

Pour sortir les élements du stream

.**toList**() stocker les élements dans une liste non modifiable

```
stream.toList()
```

.**toArray**(IntFunction<E[]>) stocker les élements dans un tableau

```
stream.toArray(Person[]::new)
```

.**findFirst**() renvoyer le premier élement en tant que Optional<E>

```
stream.findFirst()
```

allMatch, anyMatch, noneMatch

Savoir si tous (*all*), au moins un (*any*) ou aucun (*none*) élements sont vrais pour un prédicat

.**allMatch**(Predicate<E>)

```
stream.allMatch(person -> person.age() >= 18)
```

.**anyMatch**(Predicate<E>)

```
stream.allMatch(person -> person.age() >= 18)
```

.**noneMatch**(Predicate<E>)

```
stream.noneMatch(person -> person.age() >= 18)
```

reduce() / count()

Agréger les valeurs en une seule

.**reduce**(initial, combiner) combiner les élements deux à deux

```
record Stat(long sum, long count) {
  private Stat merge(Stat stat) {
    return new Stat(sum + stat.sum, count + stat.count);
  }
  private double average() { return sum / (double) count; }
}
stream.map(p -> new Stat(p.age(), 1))
  .reduce(new Stat(0, 0), Stat::merge)
  .average()
```

.**count**() compter le nombre d’élements

```
stream.count()
```

min/max(Comparator<E>)

Calcule l’élement minimum (resp. maximum) en fonction d’un comparateur

.**min**(comparator) calcule l’élement minimum

```
stream.min((p1, p2) - > Integer.compare(p1.age(), p2.age()))
```

.**max**(comparator) calcule l’élément maximum

```
stream.max(Comparator.comparingInt(Person::age));
```

IntStream, LongStream et DoubleStream

Les streams de type primitif ont des opérations supplémentaires

car les élement sont des types numériques

.**min**(), .**max**(), .**sum**(), .**average**()

Calcule le minimum, maximum, la somme et la moyenne

.**boxed**()

Opération intermédiaire qui “box” tous les élements

forEach(Consumer<E>)

Appelle le consommateur avec chaque élement

- L’ordre est l’ordre de la collection (si il existe)
- stream.forEach(System.out::println);

Attention, ne pas créer un stream juste pour appeler forEach

- collection.stream().forEach(…)
- n’est pas nécessaire car il y a déjà une méthode forEach(…) sur les collections
- collection.forEach(...)

API des Collector

La méthode **collect**(Collector)

Un collecteur est un objet qui sait créer un objet mutable (souvent une collection), ajouter les élements du Stream et optionnellement créer une version non mutable de l'objet

- Par exemple, pour une List, les élements seront d'abord stockés dans une ArrayList et optionnellement List.copyOf est appelée
- Autre exemple, les chaines de caractères sont jointes dans un StringMerger qui est transformé en String à la fin

java.util.stream.Collectors

La classe Collectors (avec un ‘s’) contient des méthodes statiques qui renvoie des Collector prédéfinies

- joining()
- toList(), toUnmodifiableList()
- toMap(), toUnmodifiableMap()
- groupingBy()

joining(delimiteur, prefix, suffix)

Permet de joindre les chaînes de caractères d’un stream avec un délimiteur (et optionnellement un préfixe et un suffixe)

```
stream.map(Person::toString)
  .collect(Collectors.joining(", "))
```

Ne marche que sur les Stream de sous-types de CharSequence (interface commune de String et StringBuilder)

toList(), toUnmodifiableList()

Collectors.**toList**() stocke les élements dans une liste mutable

```
stream.collect(Collectors.toList())
```

Collectors.**toUnmodifiableList**() stocke les élements dans une liste non modifiable

```
stream.collect(Collectors.toUnmodifiableList())
```

toUmodifiableList() est presque équivalent à stream.toList(), toUnmodifiableList() n’accepte pas les valeur **null** contrairement à stream.toList()

toMap(), toUnmodifiableMap()

Collectors.**toMap**(fonctionCle, fonctionValeur) stocke les élements dans un Map en appelant les fonctions pour obtenir la clé et la valeur d'un élement

```
stream.collect(
  Collectors.toMap(Person::name, Person::age))
    // Map<String, Integer>
```

L'implantation n'accepte pas d'avoir deux élements qui renvoient la même clé

Collectors.**toUmodifiableMap**() renvoie une version non modifiable de la Map

groupingBy(fonctionClé, collector)

Groupe les élements dans une Map dont les clés sont données par la fonctionClé. Les éléments qui ont la même valeur de clé sont stockés dans une collection créée par le collecteur passé en paramètre

```
stream.collect(
  Collectors.groupingBy(Person::age, Collectors.toList()))
    // Map<Integer, List<Person>>
```

```
stream.collect(
  Collectors.groupingBy(Person::age, Collectors.counting()))
    // Map<Integer, Long>
```

L’API Optional

java.util.Optional<E>

Représente une valeur qui est présente ou pas

Permet d'indiquer que la valeur de retour d'une méthode peut ne pas exister

- C'est mieux que de renvoyer null et voir les codes clients planter
- On force l'utilisateur de l'API à explicitement gérer le cas où la valeur n'existe pas

Création

Optional.**empty**(), Optional.**of**(E), Optional.**ofNullable**(E)

si l'élement en paramètre est **null**, Optional.Of() lève une NPE, alors que Optional.ofNullable() renvoie Optional.empty()

Méthodes de Optional

.**isPresent**()/.**isEmpty**() permet de tester si il existe une valeur

```
Optional.empty().isEmpty()  // true
Optional.of("foo").isEmpty() // false
```

.**orElseThrow**() permet d'obtenir la valeur (il plante sinon)

```
Optional.of("foo").orElseThrow()  // foo
Optional.empty.orElseThrow()  // NoSuchElementException
```

.**orElse**()/.**orElseGet**() indique une valeur/un supplier par défaut

```
Optional.of("foo").orElseGet(() -> "bar")  // foo
Optional.empty().orElseGet(() -> "bar")  // bar
Optional.empty().orElse("bar")  // bar
```

Optional en tant que monade

Optional est aussi considéré comme une sorte de stream à 0 ou 1 élément, il possède les méthodes

filter, map, flatMap et ifPresent (l'équivalent de forEach)

Comme pour les streams, Optional a des versions spécialisées pour éviter le boxing des types primitifs

```
var person = new Person("Ana", 32);
Optional.of(person).mapToInt(Person::age)
  // OptionalInt[32]
```

API des comparateurs

java.util.Comparator

Interface fonctionnelle qui permet de comparer les éléments 2 à 2

```
public interface Comparator<T> {
  int compare(T element, T other);
}
```

La sémantique de “compare” est la même que strcmp en C,

- si la valeur renvoyée est < 0, *element* est < à *other*
- si la valeur renvoyée est > 0, element est > à other
- si la valeur renvoyée est == 0, *element* est égal à *other*

Comparator.comparing()

Méthode static qui permet de créer un comparateur en indiquant une fonction de projection dont le résultat est utilisé pour comparer les éléments

```
Comparator.comparing(Person::name)
```

La méthode comparing() a des versions spécifiques pour les types primitifs pour éviter le boxing

```
Comparator.comparingInt(Person::age)
```

En résumé

L’API des Stream

Permet de faire des requètes sur les élements d’une collection de façon déclarative

- Code plus facile à lire et maintenir
  (une fois que l’on connait l’API)

L’API des Collector permet de faire les opérations de haut niveau: joindre des chaines, convertir vers une autre collection, grouper des élements, etc ...