



FACULTÉ DES SCIENCES
DE TUNIS

30 Novembre 2024

JAVA **Stream**

HJAIEJ Mohamed

 hjayej.mohamed@gmail.com



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 SQL)

JAVA 8

```
Person min = null
for(var person: persons){
    if(person.name().startsWith('A')){
        if(min==null || min.age()> person.age()){
            min = person;
        }
    }
}
return min;
```

API des Stream

```
Optional<Person> result =
persons.stream()
.filter(p->p.name().startsWith('A'))
.min(Collector.comparingInt(Person::age));
```

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

Plusieurs API

Les streams utilisent plusieurs API

L'API des *Stream* (`java.util.stream`)

Operations ensemblistes

L'API des *Collector* (`java.util.stream`)

pour mettre les résultats dans une collection.

L'API `java.util.Optional`:

Abstraction de null + véritable API

L'API `java.util.Comparator`:

créer des comparateurs et les composer.

Opérations sur les Stream

Trois étapes:

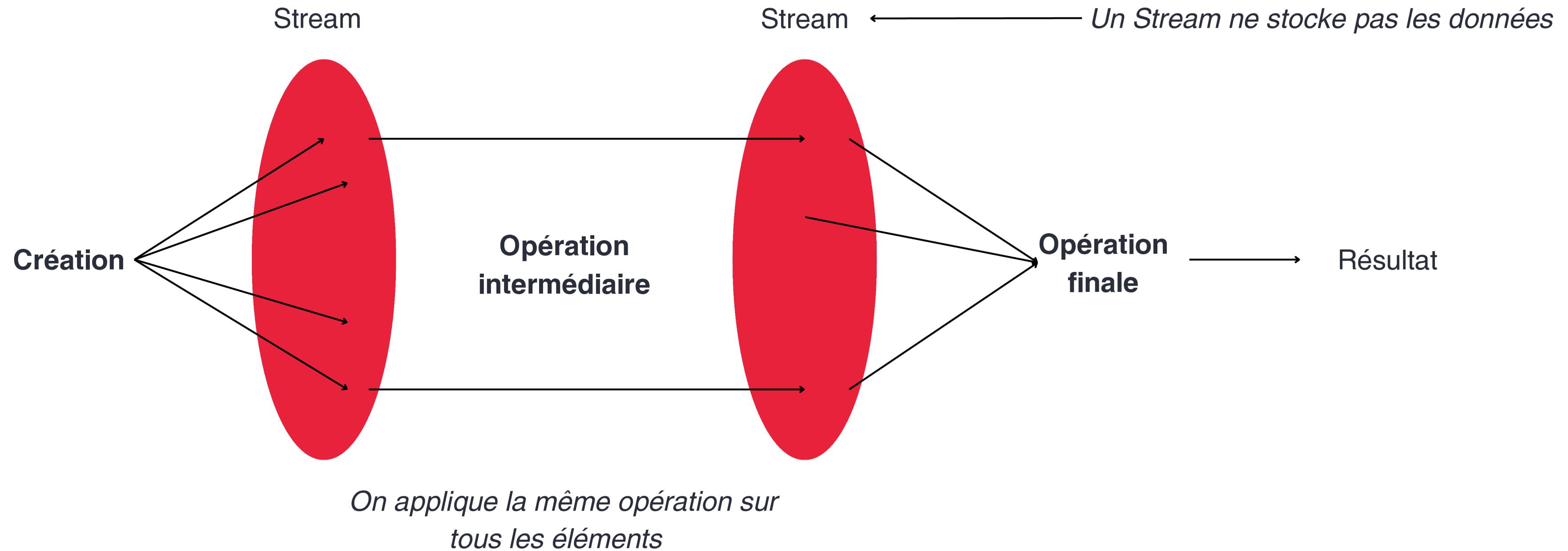
- Création d'un Stream
- Transformations successives du Stream (opération intermédiaires)
- Demander le résultat (opération finale)

Les opérations sur les streams sont paresseuses, on ne fait pas le calcul si cela n'est pas nécessaire:

par exemple: si on n'appelle pas l'opération finale, aucun calcul n'est fait.

Pipeline

Un Stream est organisé comme un pipeline, les stream sont poussées dans le pipeline.



Créer un Stream

A partir d'une collection

collection.stream()

A partir de valeurs

Stream.of(1,2,3,...)

A partir d'un tableau

Arrays.stream(tableau)

Arrays.stream(tableau,strat,end)

Stream de type primitif

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

IntStream, LongStream, DoubleStream

Par exemple: IntStream

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

Boxing

Le terme boxing (ou autoboxing) en java fait référence à la conversion automatique d'un type primitif (int, float, double) en son équivalent objet (Int, Float, Double). Cette conversion est souvent implicite et introduite pour permettre aux types primitifs d'être utilisés là où les objets sont requis, comme dans les collections ou les Streams

Le problème de boxing:

Le boxing peut avoir un impact négative sur les performances car:

- *Il introduit une surcharge aux termes de mémoire: (les objets occupent plus d'espace que les types primitifs).*
- *Il demande des opérations supplémentaires pour effectuer la conversion entre les types primitifs et leurs objets équivalents.*

Exemple Boxing

```
int number = 10;  
//autoboxing: conversion de 'int' en 'Integer'  
Integer boxedNumber = number;  
// unboxing conversion de 'Integer' en 'int'  
int unboxedNumber = boxedNumber;
```

Dans le contexte des Streams; un flux ordinaire comme Stream<Integer> peut impliquer du boxing si vous travaillez avec des types primitifs par exemple:

```
Stream<Integer> stream = List.of(1,2,3).stream();
```

Solutions: Streams Spécialisées

Pour éviter le boxing dans les opérations sur les types primitifs, java offre trois Streams spécialisés:

IntStream: pour les primitifs de type int

FloatStream: pour les primitifs de type float

DoubleStream: pour les primitifs de type double.

Example d'utilisation pour calculer la somme:

// avec boxing

```
Stream<Integer> stream = List.of(1,2,3).stream();
Stream<Integer> stream = Stream.of(1,2,3);
int sum = stream.mapToInt(Integer::intValue).sum();
```

// sans boxing

```
IntStream stream= IntStream.of(1,2,3)
int sum = stream.sum()
```

Stream sur des ressources système

Les Streams sur des ressources system doit être fermés avec close(). on utilise le try-with-ressource.

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

```
try(var stream = Files.lines(path)){ // Try-with-resources pour fermer automatiquement le stream  
    stream.forEach(System.out::println)  
}
```

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

```
try(var stream = Files.list(path)){ // Try-with-resources pour fermer automatiquement le stream  
    stream.forEach(System.out::println);  
}
```



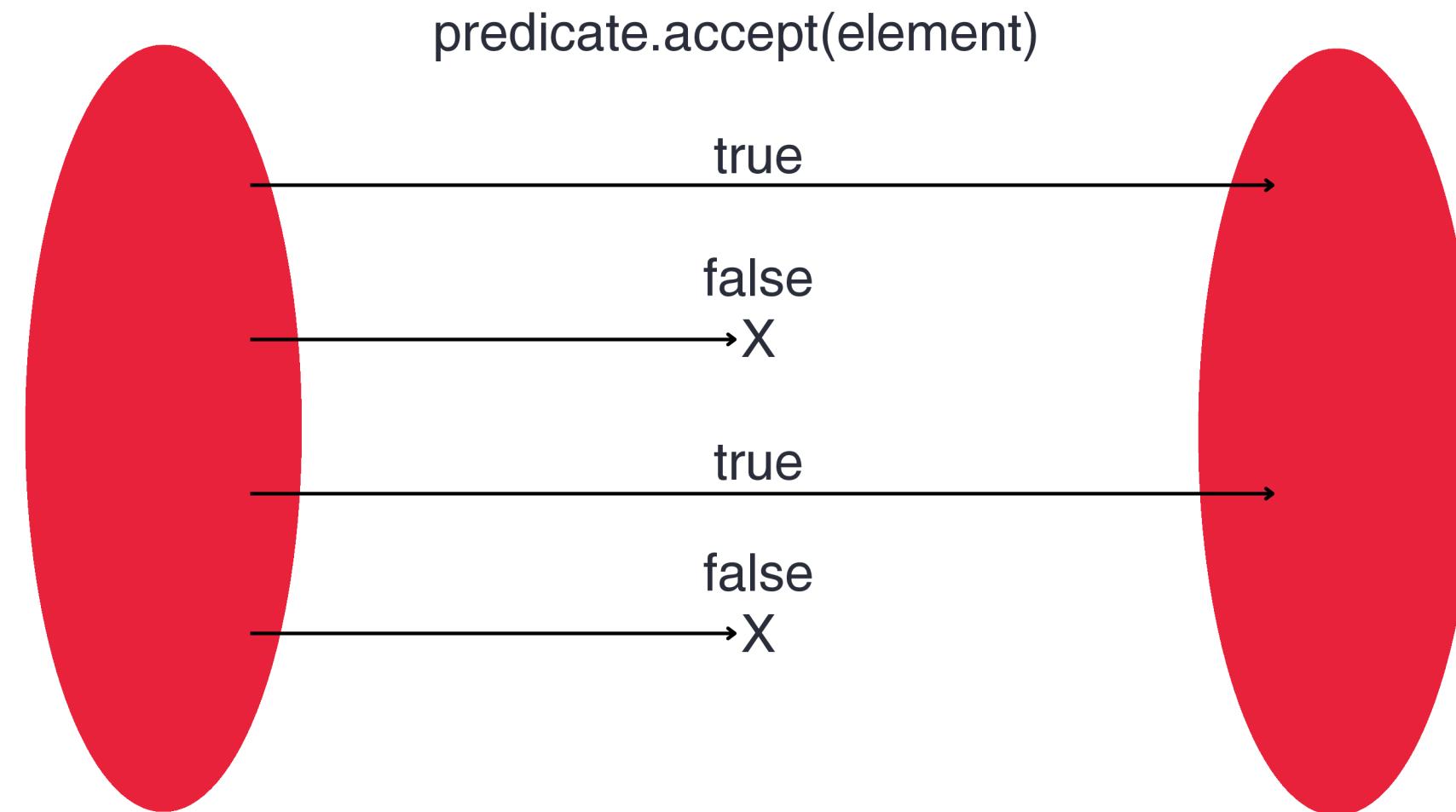
Operations intermédiaires

Opérations intermédiaire

Filter<Predicate<E>>

On appelle un prédicat($E \rightarrow \text{boolean}$) et on ne propage pas l'objet si le prédicat renvoie false.

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

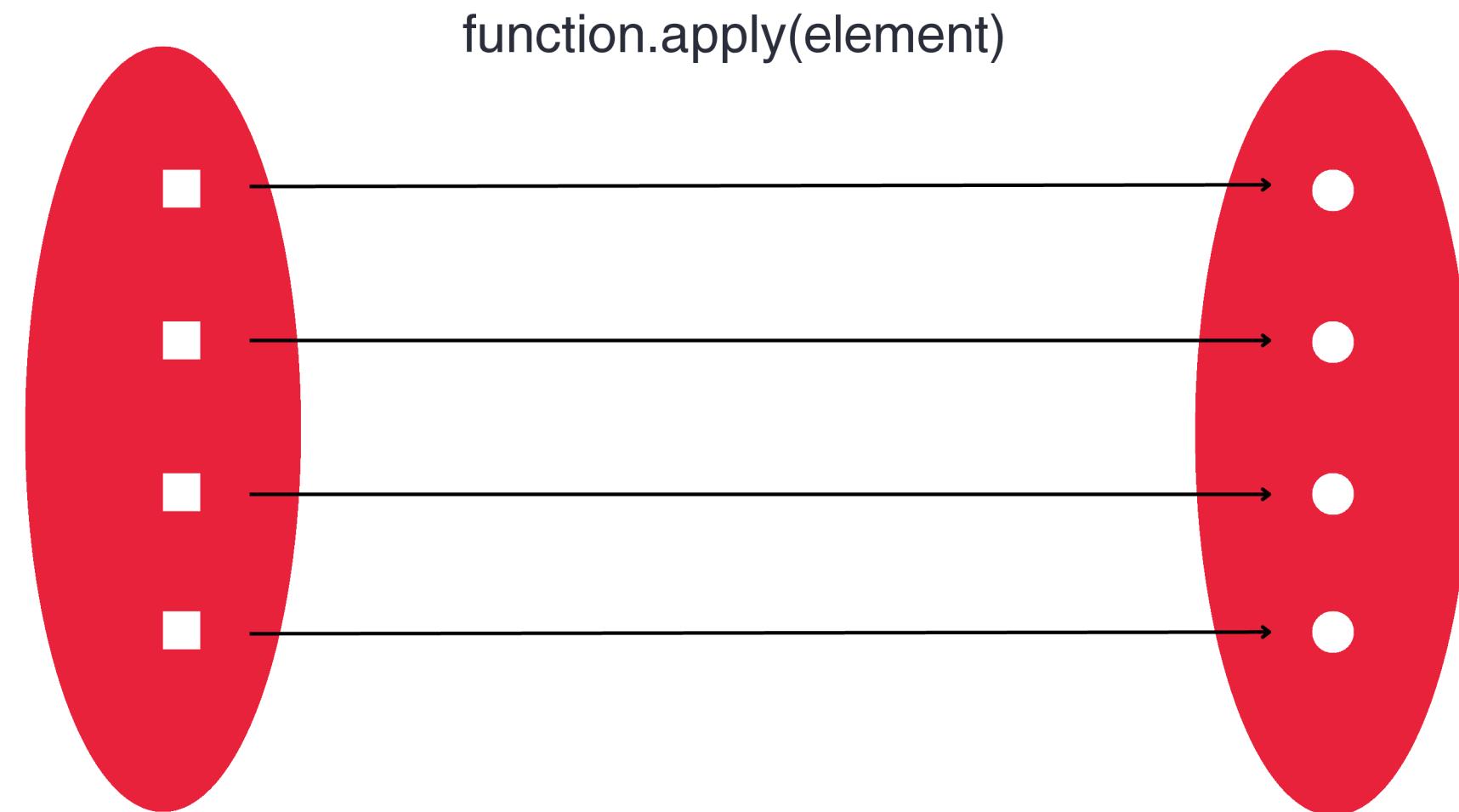


Opérations intermédiaire

Map<Function<E,R>>

Transforme chaque élément en appelant la fonction

stream.map(Person::name)

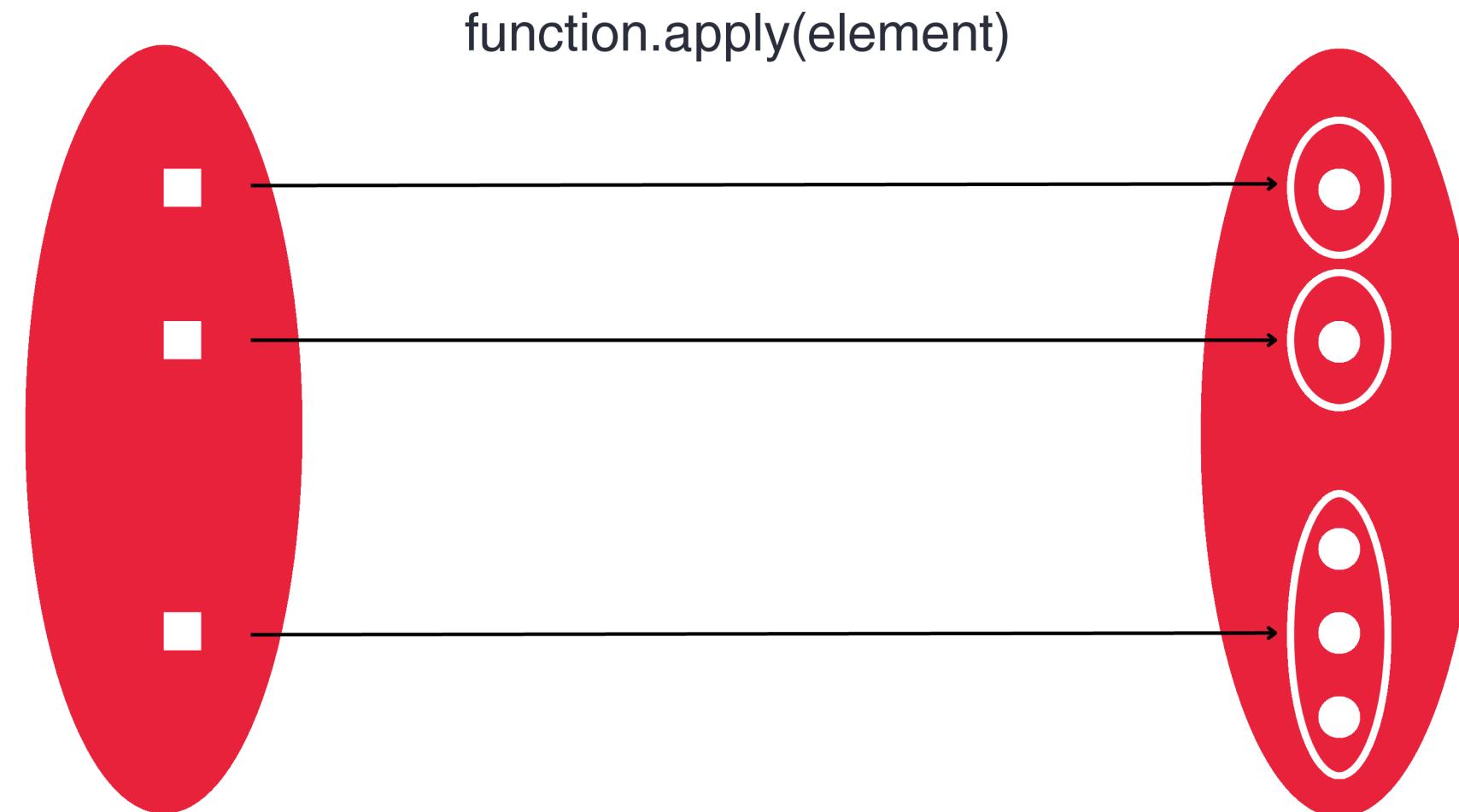


Opérations intermédiaires

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

Transforme chaque élément en appelant la fonction

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



Opérations intermédiaire

flatMap<Function<E, Stream<R>>> (2)

Transforme chaque élément d'un Stream en un autre Stream.

Aplatir les Streams imbriquées en un seul Stream continue.

stream.flatMap(element -> fonctionQuiRenvoieUnStream(element))

Exemple 1: (transformer)

```
List<String> listStr = List.of("string1 st","string2 st2","string3 st3","string4 st4","string5 st5 st6");  
Stream<String> allWords = listLmb.stream().flatMap(phrase -> Stream.of(phrase.split(" ")));
```

Exemple 2: (aplatir)

```
List<List<String>> listLmb = List.of(List.of("string1","string2"),List.of("string3","string4","string5"));  
Stream<String> allWords = listLmb.stream().flatMap(List::stream);
```

Autre Opérations intermédiaire

.limit(value)

Filtre ne laisse passer que value éléments.

.skip(value)

Saute les value éléments.

.distinct()

Filtre qui ne laisse pas passer le même valeur deux fois.

.sorted(comparateur)

Trie les éléments et les propage triés.

map/flatMap et premitif

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

Par ex:

stream.mapToInt(Person::age) -> intStream

Alors que:

stream.map(Person::age) -> Stream<Integer>



Operations finales

toList() / toArray() / findFirst()

Pour sortir les éléments du stream

.toList() stocker les éléments dans une liste non modifiable.

stream.toList()

.toArray(intFunction<E[]>) stocker les éléments dans un tableau

stream.toArray(Person[]::new)

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

stream().findFirst()

allMatch() / anyMatch() / noneMatch

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

.allMatch(Predicat<E>)

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

.anyMatch(Predicat<E>)

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

.noneMatch(Predicat<E>)

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

reduce()/count()

Agréger les valeurs en une seule.

.reduce(intial, combiner) combiner les valeurs 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'éléments

```
stream.count()
```

min()/max()

Calcule l'élément minimum (resp, maximum) en fonction d'un comparateur.

.min(comparateur) calcule l'élément minimum.

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

.max(comparateur) 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 éléments sont des types numériques.

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

Calcule le minimum, maximum, la somme et la moyenne.

.boxed()

Opération intermédiaire qui “box” tous les éléments.

forEach(Consumer<E>)

Appelle le consommateur avec chaque élément: L'ordre est l'ordre de la collection.

stream.forEach(System.out::println)

collect(Collector)

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

- Par exemple, pour une List, les éléments seront d'abord stockés dans une ArrayList et optionnellement List.copyOf est appelée.
- Autre exemple, les chaînes 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”) sontient des méthods statiques qui renvoie des Collector prédéfinies.

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

joining(délimiteur, prefix, suffix)

Permet de joindre les chaînes de caractères d'un stream avec un délimiteur (et optionnellement un préfix et un suffix).

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

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

toList(), toUnmodifiableList()

Collectors.toList() stocke les éléments dans une liste mutable.

stream.collect(Collectors.toList())

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

stream.collect(Collectors.toUnmodifiableList())

toUnmodifiableList() est presque équivalent à stream.toList(), toUnmodifiableList() n'accepte pas les valeurs null contrairement à stream.toList().

toMap(), toUnmodifiableMap()

Collectors.toMap(fonctionCle,fonctionValue) stocke les éléments dans un Map en appelant les fonctions pour obtenir la clé et la valeur d'un élément.

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

=> L'implémentation n'accepte pas d'avoir deux éléments qui renvoient la même clé.

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

groupingBy(fonctionClé,collector)

Groupe les éléments 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 présentée 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)

stream.collect(Collectors.groupingBy(Person::age, Collectors.counting())) //Map<Integer,Long>
si l'élément 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 0 à 1 élément, il posséde 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",23)  
Optional.of(person).mapToInt(Person::age) //OptionalInt[32]
```



L'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, élément est < à other.
- si la valeur renvoyée est > 0, élément est > à other.
- si la valeur renvoyée est == 0, élément est égale 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)



Exercices

Average()

Ecrire un programme java pour calculer la moyenne d'une liste of entiers avec l'API stream.

Solution1

```
import java.util.*;  
List<Integer> numbers = List.of(1,2,3);  
double res = numbers.stream().mapToInt(Integer::intValue).average().orElse(0.0);
```

Solution2

```
import java.util.*;  
import java.util.stream.*;  
IntStream stream = IntStream.range(1,5);  
OptionalDouble res = stream.average();
```

Average()

Ecrire un programme java pour calculer la moyenne d'une liste of entiers avec l'API stream.

Solution1

```
import java.util.*;  
List<Integer> numbers = List.of(1,2,3);  
double res = numbers.stream().mapToInt(Integer::intValue).average().orElse(0.0);
```

Solution2

```
import java.util.*;  
import java.util.stream.*;  
IntStream stream = IntStream.range(1,5);  
OptionalDouble res = stream.average();
```

map()

Ecrire un programme java transformer une liste des mots en Majuscules.

Solution

```
import java.util.*;  
List<String> words = List.of("Optional", "String", "Java");  
List<String> uppercased = words.stream().map(String::toUpperCase).collect(Collectors.toList());
```

groupingBy() / summingInt

Ecrire un programme java qui calcule les la somme des valeurs paires et impaires.

Solution

```
import java.util.*;  
IntStream stream = IntStream.range(1,10);  
Map<Integer,Integer> res = stream.boxed().collect(Collectors.groupingBy(n->n%2==0,  
Collectors.summingInt(n->n));
```

distinct()

Ecrire un programme java qui supprime les duplications dans un liste.

Solution

```
import java.util.*;  
  
List<String> words = List.of("hi","hello","hi","hello");  
List<String> uniqueWords= words.stream().distinct().toList();  
System.out.println(uniqueWords);
```

filter():Stream, counting():Long

Ecrire un programme java qui calcule le nombre des mots qui commencent avec un spécifique lettre.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<String> words = List.of("hi","hello","hi","hello");  
Long count= words.stream().filter(w->w.startsWith("h")).collect(Collectors.counting());  
System.out.println(count);
```

sorted():Stream, Collector.reverseOrder()

Ecrire un programme java qui ordonne les mots d'une liste selon l'ordre alphabétique dans les deux sens descendant et descendant.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<String> words = List.of("hi","hello","hi","hello");  
List<String> orderAsc = words.stream().sorted().toList();  
List<String> orderDesc = words.stream().sorted(Comparator.reverseOrder()).toList();
```

max():Optional<T>, min():Optional<T>

Ecrire un programme java qui donne le maximum et le minimum d'une liste des entiers.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<Integer> numbers = List.of(1,2,3,4,5);  
Integer max = numbers.stream().max(Integer::compare).get();  
Integer min = numbers.stream().min(Integer::compare).get();
```

skip(n):Stream, findFirst():Optional<T>

Ecrire un programme java qui donne le deuxième minimum et maximum d'une liste des entiers.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<Integer> numbers = List.of(1,2,3,4,5);  
Integer secondMax = numbers.stream().sorted(Integer::compare).skip(1).findFirst().orElse(null);  
Integer secondMin = numbers.stream().sorted((i1,i2)->Integer.comapre(i2,i1)).skip(1).findFirst().orElse(null);
```

skip(n):Stream, findFirst():Optional<T>

Ecrire un programme java qui donne le deuxième minimum et maximum d'une liste des entiers.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<Integer> numbers = List.of(1,2,3,4,5);  
Integer secondMax = numbers.stream().sorted(Integer::compare).skip(1).findFirst().orElse(null);  
Integer secondMin = numbers.stream().sorted((i1,i2)->Integer.comapre(i2,i1)).skip(1).findFirst().orElse(null);
```

reduce():Optional<T>

Ecrire un programme java qui donne le premier élément d'une liste des entiers.

Solution

```
import java.util.*;  
import java.util.stream.*;  
List<Integer> numbers = List.of(1,2,3,4,5);  
Integer firstWithReduce = numbers.stream().reduce((first,second)->first).orElse(null);
```

This do the samething as findFirst()

peek():Flux<T> : intercepter le stream

En Java, Stream fournit une alternative puissante pour traiter les données. Nous allons ici discuter de l'une des méthodes les plus fréquemment utilisées, appelée peek(), qui, en tant qu'action de consommateur, renvoie essentiellement un flux composé des éléments de ce flux, en exécutant en outre l'action fournie sur chaque élément au fur et à mesure que les éléments sont consommés à partir du flux résultant. Il s'agit d'une opération intermédiaire, car elle crée un nouveau flux qui, une fois parcouru, contient les éléments du flux initial qui correspondent au prédictat donné. .

Flux<T> peek(Consommateur<? super T> action)

Example

```
import java.util.*;  
import java.util.stream.*;  
Integer max = numbers.stream().peek(System.out::println).max(Integer::compare).get();
```

concat(Stream<T>,Stream<T>):Stream<T>

La méthode Stream.concat() crée un flux concaténé dans lequel les éléments sont tous les éléments du premier flux suivis de tous les éléments du second flux. Le flux résultant est ordonné si les deux flux d'entrée sont ordonnés, et parallèle si l'un des deux flux d'entrée est parallèle.

Example

```
import java.util.*;  
import java.util.stream.*;  
Stream<Integer> stream1 = Stream.of(1, 3, 5);  
Stream<Integer> stream2 = Stream.of(2, 4, 6);  
Stream<Integer> resultingStream = Stream.concat(stream1, stream2);
```