

λ

Rémi Forax

Wildcards ?

Attention, ce cours est sur les lambdas et pas sur les méthodes paramétrées même si les deux sont liées

La plupart des signatures des méthodes de ce cours sont fausses car il manque les wildcards (qui sont vus dans le cours sur les types / méthodes paramétrés)

Universalité

En POO, l'universalité est le fait qu'un objet puisse représenter à la fois

- des données
- du code

Voir du code (une fonction) comme un objet permet d'abstraire en séparant

- la partie générique et réutilisable d'un algo
- la partie spécifique non réutilisable

Exemple

List.sort(comparator)

Séparée en 2 parties

- L'algo de trie en lui-même (sort) qui est ré-utilisable
- La fonction de comparaison (le comparator) qui spécialise sort en fonction d'un order particulier

La fonction de comparaison est **passée en paramètre** de l'algorithme générique

Avec les mains ...

Spécifier un type de fonction en Java

Par ex: (String, String) → int
pour pouvoir déclarer la méthode générique
`void sort((String, String) → int comparator) { ... }`

Voir une méthode comme une fonction

Pour pouvoir appeler la méthode générique
avec une fonction

```
class MyComparators {  
    static int compareByLength(String s1, String s2) { ... }  
}  
...  
list.sort(& compareByLength);
```

En Java

Être capable de spécifier un type de fonction en Java

~~(String, String) → int~~

On utilise une interface fonctionnelle

```
interface Comparator<T> { int compare(T t1, T t2); }  
void sort(Comparator<String> comparator) {  
}
```

Être capable de voir une méthode comme une fonction

On utilise un nouvel **opérateur ::** (*colon colon*)

```
class MyComparators {  
  static int compareByLength(String s1, String s2) { ... }  
}  
...  
list.sort(& compareByLength);  
list.sort(MyComparators::compareByLength);
```

Pas un truc nouveau

Tous les langages ont un moyen d'abstraire du code

- En C: type pointeur de fonction + une fonction est une valeur
- En Python/JavaScript: une fonction est une valeur
- En C#: un delegate est un type de fonction + conversion automatique
- En Java: interface fonctionnelle est un type fonction + conversion une méthode vers une instance

Interface Fonctionnelle

Interface fonctionnelle

Une interface fonctionnelle est une interface avec une seule méthode abstraite

Représente un type de fonction (astuce !)

```
public interface BiFunction<T, U, V> {  
    public V apply(T t, U u);  
}
```

représente le type $(\mathbf{T}, \mathbf{U}) \rightarrow \mathbf{V}$

```
BiFunction<String, Integer, Double> fun =  
    (String s, Integer i) -> s.length() + i + 5.0;
```

@FunctionalInterface

Le compilateur vérifie qu'il y a qu'une seule méthode abstraite

Les méthodes privées, statiques ou par défaut ne comptent pas

Documente (javadoc) que l'interface représente un type de fonction

```
@FunctionalInterface
public interface BiFunction<T, U, V> {
    public V apply(T t, U u);
}
```

Interface Fonctionnelle vs Type de fonction

Problème

Cela veut dire que à chaque fois que je veux un type de fonction, il faut que je crée une interface ?

Oui mais

Le JDK fourni déjà un ensemble d'interfaces fonctionnelle pour les types de fonction les plus fréquents dans le package `java.util.function`

Mais cela veut dire que je doit les connaitres

Oui !

Toutes ?

Oui !

Package java.util.function

Types de fonction

- de 0 à 2 paramètres
 - Runnable, Supplier, Consumer, Function, BiFunction, ...
- spécialisé pour les types primitifs
 - IntSupplier () → int, LongSupplier () → long, DoubleSupplier () → double
 - Predicate<T> (T) → boolean
 - IntFunction<T> (int) → T
 - ToIntFunction<T> (T) → int
- spécialisé si même type en paramètre et type de retour
 - UnaryOperator (T) → T ou BinaryOperator (T, T) → T
 - DoubleBinaryOperator (double, double) → double, ...

Runnable et Supplier

`java.lang.Runnable` est équivalent à `() → void`

```
Runnable code = () -> { System.out.println("hello"); }  
code.run();
```

`Supplier<T>` est équivalent à `() → T`

```
Supplier<String> factory = () -> "hello";  
System.out.println(factory.get());
```

`[Int|Long|Double]Supplier`

```
IntSupplier factory = () -> 42;  
System.out.println(factory.getAsInt());
```

Consumer

Consumer<T> est équivalent à **(T) → void**

```
Consumer<String> printer =  
    s -> System.out.println(s);  
printer.accept("hello");
```

[Int|Long|Double]Consumer

```
DoubleConsumer printer =  
    d -> System.out.println(d);  
printer.accept(42.0);
```

Predicate

Predicate<T> représente **(T) → boolean**

```
Predicate<String> isSmall = s -> s.length() < 5;  
System.out.println(isSmall.test("hello"));
```

[Int|Long|Double]Predicate

```
LongPredicate isPositive = v -> v >= 0;  
System.out.println(isPositive.test(42L));
```

Function

Function<T,U> représente **(T) → U**

```
Function<String, String> fun = s -> "hello " + s;  
System.out.println(fun.apply("function"));
```

[Int|Long|Double]Function<T>

```
IntFunction<String[]> arrayCreator =  
    size -> new String[size];  
System.out.println(arrayCreator.apply(5).length);
```

To[Int|Long|Double]Function<T>

```
ToIntFunction<String> stringLength = s -> s.length();  
System.out.println(stringLength.applyAsInt("hello"));
```

UnaryOperator

UnaryOperator<T> représente **(T) → T**

```
UnaryOperator<String> op = s -> "hello " + s;  
System.out.println(op.apply("unary operator"));
```

[Int|Long|Double]UnaryOperator

```
IntUnaryOperator negate = x -> - x;  
System.out.println(negate.applyAsInt(7));
```

BiPredicate et BiFunction

BiPredicate<T, U> représente

(T, U) → boolean

BiPredicate<String, String> isPrefix =

(s, prefix) -> s.startsWith(prefix);

System.out.println(isPrefix.test("hello", "hell"));

BiFunction<T, U, V> représente **(T, U) → V**

BiFunction<String, String, String> concat =

(s1, s2) -> s1 + " " + s2;

System.out.println(concat.apply("hello", "Bob"));

BinaryOperator

BinaryOperator<T> représente $(T, T) \rightarrow T$

BinaryOperator<String> concat =

(s1, s2) -> s1 + " " + s2;

System.out.println(concat.**apply**("hello", "binop"));

[Int|Long|Double]BinaryOperator

IntBinaryOperator add = (a, b) -> a + b;

System.out.println(add.**applyAsInt**(40, 2));

Opérateur ::
(method reference)

L'opérateur `::` (*method reference*)

Permet de voir une méthode comme une instance d'une interface fonctionnelle

```
class Text {  
    static boolean startsWith(String text, String prefix) {  
        ...  
    }  
}
```

même classe

même nom

```
BiPredicate<String, String> function = Text::startsWith;
```

interface fonctionnelle

Trouver la bonne méthode

Le compilateur utilise le type des paramètres/type de retour de la méthode abstraite de l'interface fonctionnel pour trouver la méthode référencée par l'opérateur ::

```
interface BiPredicate<T, U> {  
    boolean test(T t, U u);  
}  
  
class Text {  
    static boolean startsWith(String text, String prefix) {  
        ...  
    }  
}  
  
BiPredicate<String, String> function = Text::startsWith;
```

The diagram illustrates how the compiler finds the correct method for a lambda expression. It shows three code snippets with arrows indicating the mapping of types:

- The `BiPredicate` interface has a `test` method with parameters `T` and `U` and a `boolean` return type.
- The `Text` class has a `startsWith` method with parameters `String` and `String` and a `boolean` return type.
- The lambda expression `BiPredicate<String, String> function = Text::startsWith;` has parameters `String` and `String` and a `BiPredicate` return type.

Arrows point from the `boolean` return type in the interface to the `boolean` return type in the class. Arrows also point from the `T` parameter in the interface to the `String` parameter in the class, and from the `U` parameter in the interface to the `String` parameter in the class. Finally, an arrow points from the `BiPredicate` return type in the lambda expression to the `BiPredicate` return type in the interface.

Opérateur :: et interface fonctionnelle

L'opérateur :: ne marche que si le compilateur peut calculer que le type *cible* est une interface fonctionnelle

```
static void foo(ToIntFunction<String> f) { ... }
```

```
static void bar(Object f) { ... }
```

```
foo(Integer::parseInt); // Ok !
```

```
bar(Integer::parseInt); // compile pas
```

Les différents operateurs ::

Il existe 5 formes de *method reference*

- Méthode statique

```
ToDoubleFunction<String> parseDouble = Double::parseDouble;
```

- Méthode d'instance

```
ToIntFunction<String> stringLength = String::length;
```

- Méthode d'instance + receveur attaché (*bound*)

```
String s = "hello";  
IntSupplier helloLength = s::length;
```

- Constructeur

```
record Person(String name) { }  
Function<String, Person> factory = Person::new;
```

- Construction de Tableau

```
IntFunction<String[]> arrayFactory = String[]::new;
```

:: et méthode d'instance

L'opérateur :: voit une méthode comme une fonction donc pour les méthodes d'instance, il ne faut pas oublier le type de **this**

```
record Person(String name, int age) {  
    boolean isOlderThan(Person person) {  
        return age > person.age;  
    }  
    // est équivalent à  
    boolean isOlderThan(Person this, Person person) {  
        return this.age > person.age;  
    }  
}  
  
BiPredicate<Person, Person> older = Person::isOlderThan;
```


Methode avec receveur attaché (*bound*)

L'opérateur `::` permet de spécifier une valeur qui sera utilisée comme receveur (valeur de `this`) d'une méthode d'instance

```
record Person(String name, int age) {  
    boolean isOlderThan(Person person) {  
        return age > person.age;  
    }  
}
```

```
var john = new Person("John", 37);  
Predicate<Person> isYoungerThanJohn =  
john::isOlderThan;
```

C'est la même chose que l'application partiel de Haskell !

Opérateur :: et conversions

L'opérateur :: effectue les conversions suivantes

- Contravariance des paramètres
- Covariance du type de retour
- Conversion des types primitifs
- Boxing/unboxing et varargs

les mêmes conversions que lors d'un appel de méthode

Par exemple,

```
ToDoubleFunction<String> fun = Integer::parseInt;
```

```
Function<String, Integer> fun = Integer::parseInt;
```

Conversion entre interfaces fonctionnelles

Même si deux interfaces fonctionnelles représentent le même type fonction, elle ne sont pas le même type

```
void foo(Runnable runnable) { ... }
```

```
interface Code { void execute(); }
```

```
Code code = ...
```

```
foo(code); // compile pas
```

On utilise l'opérateur `::` pour faire la conversion

```
foo(code::execute); // Ok !
```

Lambda

Opérateur :: et lambda

L'opérateur :: force à avoir une méthode nommée déjà définie dans une classe

```
class Utils {  
    static boolean startsWithAStar(String s) {  
        return s.startsWith("*");  
    }  
}  
Predicate<String> predicate = Utils::startsWithAStar;
```

Une lambda permet de déclarer une méthode anonyme

```
Predicate<String> predicate = (String s) -> s.startsWith("*");
```

Lambda → code simplifié

Le compilateur *de-sugar* la lambda en une méthode avec un nom générée

```
Predicate<String> predicate =  
    (String s) -> s.startsWith("*");
```

est transformé en

```
static boolean lambda$1(String s) {  
    return s.startsWith("*");  
}
```

```
Predicate<String> predicate = Utils::lambda$1;
```

Inférence du type des paramètres

Il n'est pas obligatoire de spécifier le types des paramètres

Comme pour l'opérateur ::, le type des paramètres est déjà présent dans l'interface fonctionnelle

```
interface Comparator<T> {  
    int compare(T t, T t2);  
}
```

```
Comparator<String> c =  
    (s1, s2) -> s1.compareToIgnoreCase(s2);
```

Syntaxes des lambdas

Lambda block

```
Comparator<String> c = (s1, s2) -> {  
    return s1.compareToIgnoreCase(s2);  
};
```

Lambda expression

```
Comparator<String> c =  
    (s1, s2) -> s1.compareToIgnoreCase(s2);
```

pas de return car il y a une seule expression

Syntaxe en fonction du nombre de paramètres

La syntaxe est “optimisée” pour 1 paramètre
comme en JavaScript, C#, Scala

1 paramètre, parenthèse pas nécessaire

`s -> s.length()`

0 paramètre

`() -> 42`

2 ou + paramètres

`(s1, s2) -> s1.compareTo(s2);`

Capture de la valeur des variables locales

Les variables capturées doivent être assignées une seule fois (effectivement final)

```
static void foo() {  
    for(var loop = 1; loop <= 5; loop++) {  
        Runnable runnable = () -> {  
            System.out.print(loop); // compile pas, loop est modifiée  
        };  
        ...  
    }  
}
```

Capture de la valeur des variables locales (2)

Les variables capturées doivent être assigné une seule fois
(effectivement final)

```
static void printTriangle() {  
    for(var loop = 1; loop <= 5; loop++) {  
        var copyOfLoop = loop;  
        Runnable runnable = () -> {  
            System.out.print(copyOfLoop); // ok  
        };  
        ...  
    }  
}
```

C# et JavaScript(avec **let**) sont plus malins et ont un cas spécial pour les variable de boucle, en Java il faut le faire à la main :(

Capture de **this**

Une lambda peut aussi capturer **this** (implicitement)

```
record Person(String name, int age) {  
    Runnable printer() {  
        return () -> {  
            // System.out.println(name + " " + age);  
            System.out.println(this.name + " " + this.age);  
        };  
    }  
}
```

Lorsque l'on accède à des champs de l'objet englobant
seul **this** est capturé

En résumé

Avoir des objets qui représente des fonctions
(du code)

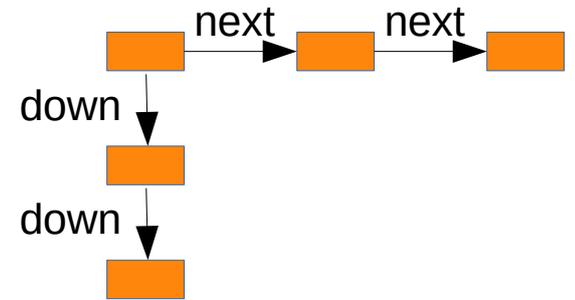
Utilise une interface fonctionnelle pour le
typage

2 syntaxes

- Le code est simple, on utilise les lambdas
- La méthode existe déjà ou le code est plus compliqué, on donne un nom et utilise l'opérateur ::

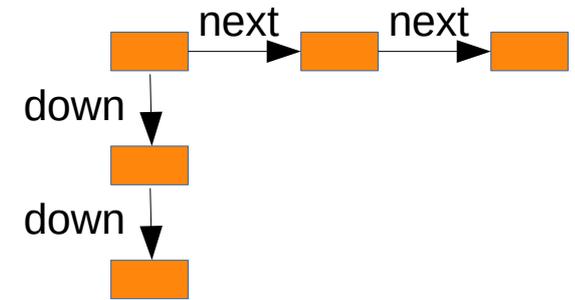
Exemples de factorisation de code

Exemple Simple



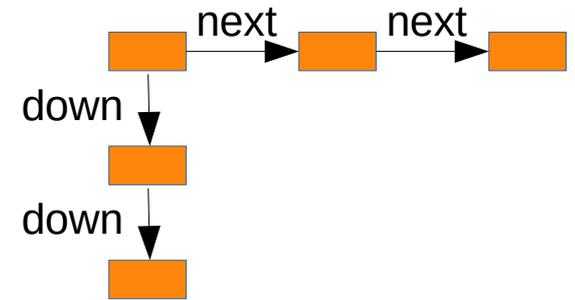
```
record Link(int value, Link next, Link down) {  
    public void printAllNext() {  
        for(var link = this; link != null; link = link.next) {  
            System.out.println(link.value);  
        }  
    }  
  
    public void printAllDown() {  
        for(var link = this; link != null; link = link.down) {  
            System.out.println(link.value);  
        }  
    }  
}
```

Factorisation



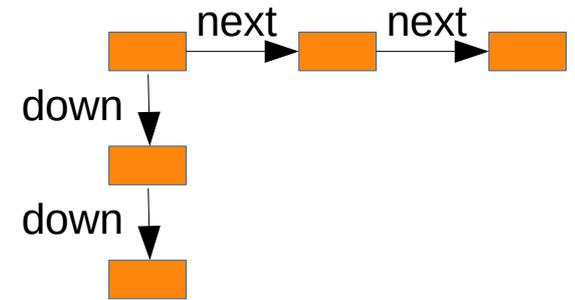
```
record Link(int value, Link next, Link down) {  
  private void printAll(???) {  
    for(var link = this; link != null; ???) {  
      System.out.println(link.value);  
    }  
  }  
}  
  
public void printAllNext() {  
  printAll(???)  
}  
  
public void printAllDown() {  
  printAll(???)  
}  
}
```

Interface fonctionnelle



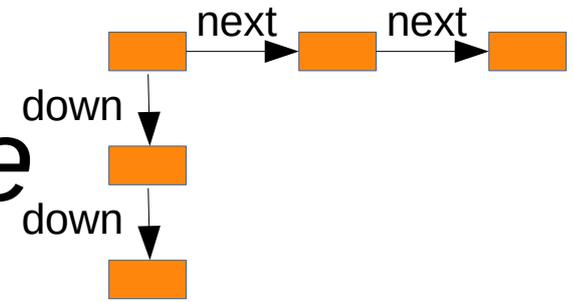
```
record Link(int value, Link next, Link down) {  
  private void printAll(UnaryOperator<Link> op) {  
    for(var link = this; link != null; link = op.apply(link)) {  
      System.out.println(link.value);  
    }  
  }  
}  
  
public void printAllNext() {  
  printAll(???);  
}  
  
public void printAllDown() {  
  printAll(???);  
}  
}
```

Lambda



```
record Link(int value, Link next, Link down) {  
    private void printAll(UnaryOperator<Link> op) {  
        for(var link = this; link != null; link = op.apply(link)) {  
            System.out.println(link.value);  
        }  
    }  
}  
  
public void printAllNext() {  
    printAll(l -> l.next);  
}  
  
public void printAllDown() {  
    printAll(l -> l.down);  
}  
}
```

Lambda *method reference*



```
record Link(int value, Link next, Link down) {  
    private void printAll(UnaryOperator<Link> op) {  
        for(var link = this; link != null; link = op.apply(link)) {  
            System.out.println(link.value);  
        }  
    }  
}
```

```
public void printAllNext() {  
    printAll(Link::next);  
}
```

```
public void printAllDown() {  
    printAll(Link::down);  
}
```

Link est un record,
il existe les accesseurs
next() et down()

Lambda et java.util

Collection et Map

Le package `java.util` est un gros consommateur de lambdas

Quelques exemples

Supprimer les chaînes de caractères vides ("")

`Collection<E>.removeIf(Predicate<E> predicate)`

- `list.removeIf(String::isEmpty);`

Parcourir les éléments d'une table de hachage

`Map<K, V>.forEach(BiConsumer<K, V> consumer)`

- `map.forEach((key, value) -> {
 ...
});`

Trie

java.util.List a une méthode sort(Comparator)

```
List<String> strings = ...  
strings.sort(String::compareToIgnoreCase);
```

Et pour une liste triée par nom

```
record Person(String name) { }  
List<Person> persons = ...  
list.sort((p1, p2) -> {  
    return p1.name().compareTo(p2.name());  
});
```

Comparateur

L'interface `Comparator` possède des méthodes statique et par défaut (HoFs) qui permettent de simplifier l'écriture

```
record Person(String name, int age) { }
```

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

```
// comparaison par noms
```

```
list.sort(Comparator.comparing(Person::name));
```

```
// comparaison par noms puis par age
```

```
list.sort(
```

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

```
    thenComparingInt(Person::age));
```

`java.util.stream.Stream`

Opérations sur les Stream

API Introduite en Java 8

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

Exemple

Trouver la liste des noms de tous les employés qui ont plus de 50 ans

```
record Employee(String name, int age) { }  
List<String> findEmployeeNameOlderThan(  
    List<Employee> employees, int  
age) {  
    var list = new ArrayList<String>();  
    for(var employee: employees) {  
        if (employee.age() >= age) {  
            list.add(employee.name());  
        }  
    }  
    return list;  
}
```

Exemple avec un Stream

Trouver la liste des noms de tous les employés qui ont plus de 50 ans

```
record Employee(String name, int age) { }  
List<String> findEmployeeNameOlderThan(  
    List<Employee> employees, int age) {  
    return employees.stream()  
        .filter(e -> e.age() >= age)  
        .map(Employee::name)  
        .toList();  
}
```

On indique ce que l'on veut comme résultat
pas comment on fait pour l'obtenir

Stream vs SQL

L'interface Stream permet d'effectuer des opérations ensemblistes comme SQL

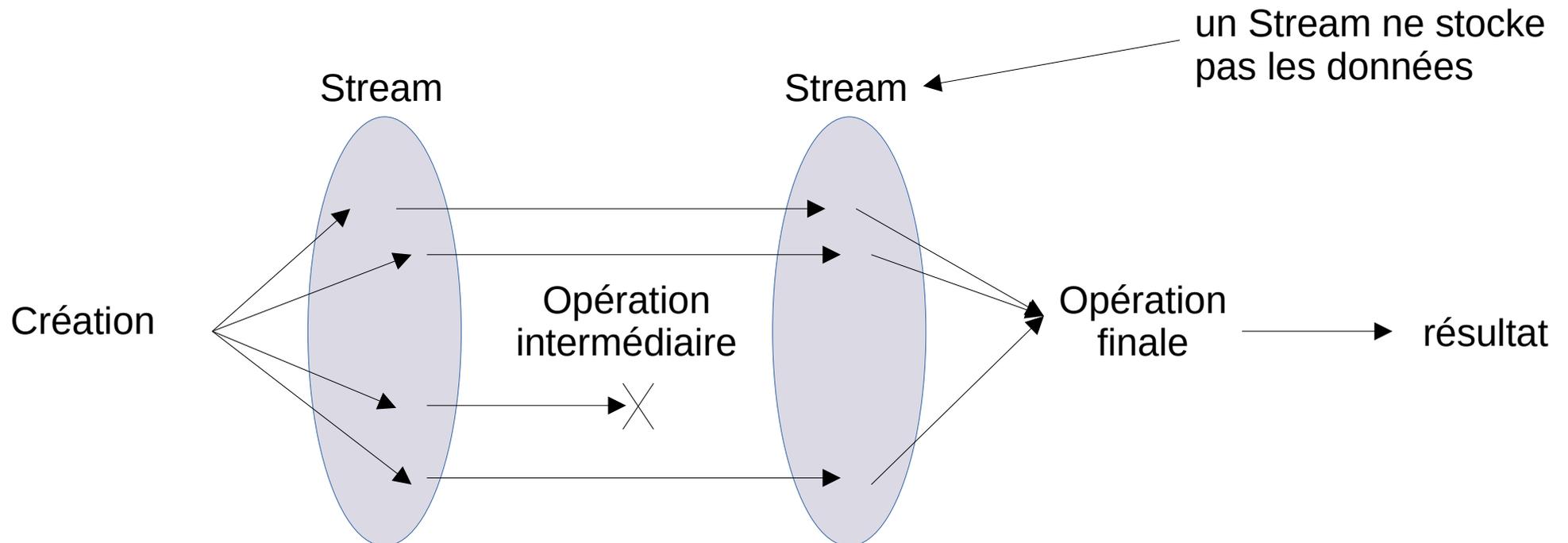
```
SELECT name FROM users  
WHERE country = 'France' ORDER BY id
```

```
users.stream()  
  .filter(user -> user.country().equals("France"))  
  .sorted(Comparator.comparing(User::id))  
  .map(User::name)  
  .toList();
```

Pipeline

Un Stream est organisé comme un pipeline

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



On applique la même opération sur tous les éléments

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

```
Arrays.stream(array)
```

```
Arrays.stream(array, start, end)
```

Stream de type primitif

Pour éviter le boxing, il existe trois streams 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

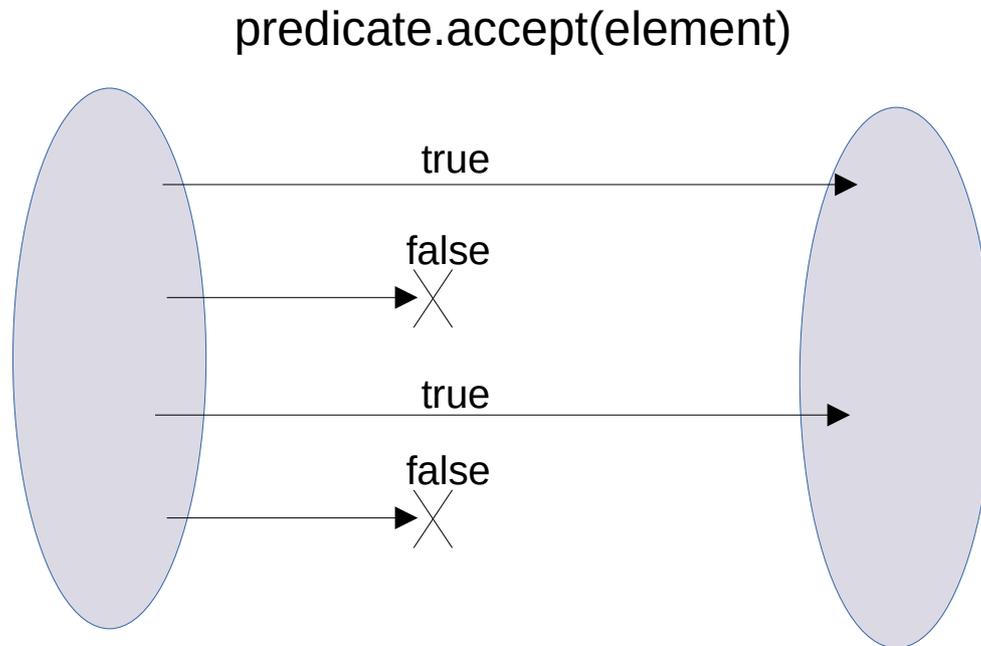
```
try(var stream = Files.list(path)) {  
    stream. ...  
}
```

Opérations intermédiaires

filter(Predicate<E>)

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

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



map(Function<E,R>)

Transforme chaque élément en appelant la fonction

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

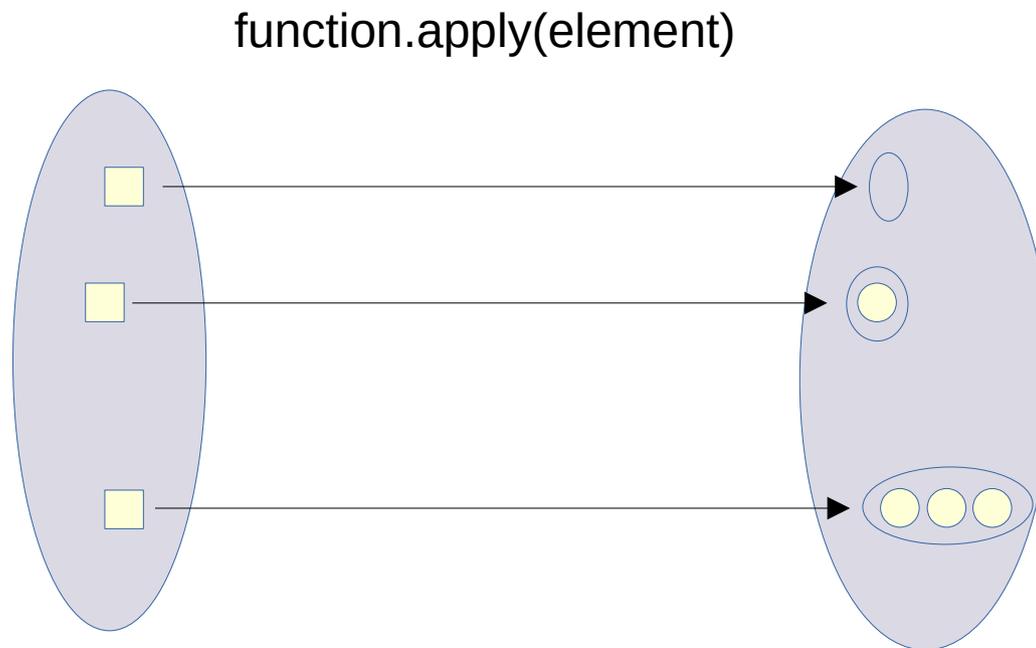
function.apply(element)



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

Transforme chaque élément en aplatissant le stream

```
stream.flatMap(p -> p.children.stream())
```



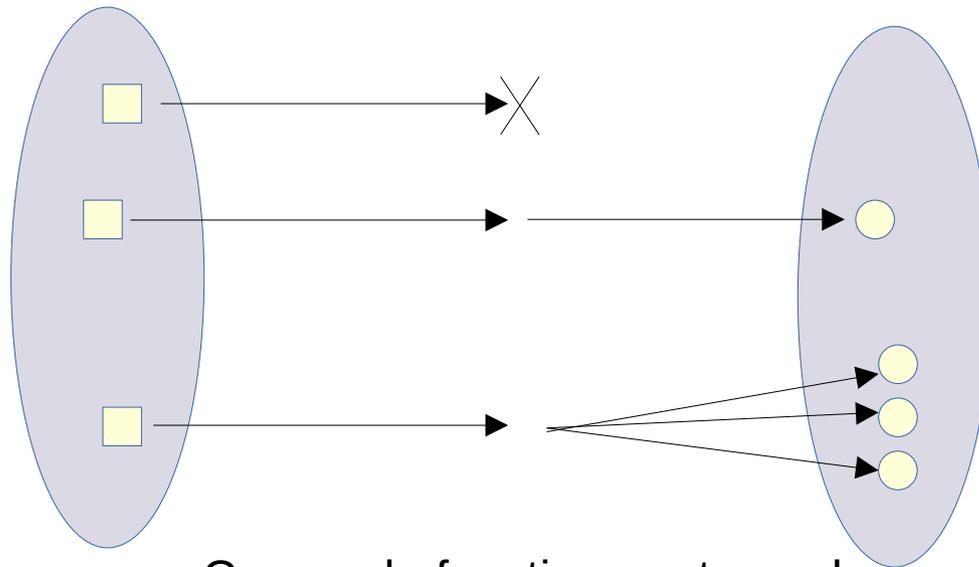
Comme la fonction renvoie un Stream,
elle renvoie de 0 à n éléments

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

Pour chaque objet, peut propager zéro ou plusieurs objets

```
stream.mapMulti((Person p, Consumer<Person> consumer) -> {  
    for (var child : p.children) {  
        consumer.accept(child);  
    }  
})
```

biConsumer.accept(element, consumer)



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

Autre opérations intermédiaires

.limit(value)

Filtre qui ne laisse passer que *value* éléments

.skip(value)

Saute les *value* premiers éléments

.distinct()

Filtre qui ne laisse pas passer la même valeur deux fois

.sorted(comparateur)

Trie les éléments 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 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(Predicate<E>)

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

.anyMatch(Predicate<E>)

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

.noneMatch(Predicate<E>)

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

reduce() / count()

Agréger les valeurs en une seule

.reduce(T initial, BinaryOperatot<T> combiner)
combiner les éléments deux à deux

```
stream.mapToInt(Person::salary)  
      .reduce(0, Integer::sum)
```

.count() compter le nombre d'éléments

```
stream.count()
```

min/max(Comparator<E>)

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

.min(comparator) calcule l'élément 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 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 (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 Collectors

La méthode **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 StringJoiner qui est transformé en String à la fin

java.util.stream.Collectors

La classe Collectors (avec un 's') contient des méthodes statiques qui renvoient 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 éléments dans une liste modifiable

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

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

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

Collectors.toUnmodifiableList() 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 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'implantation n'accepte pas d'avoir deux éléments qui renvoient la même clé

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

groupBy(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 passé à un Collector sous-jacent

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

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

downstream collectors

