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 fausse 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

Pas un truc nouveau

Tous les languages ont un moyen d’abstraire du code (une fonction)

- En C: type pointeur de fonction + une fonction est une valeur
- En Python/JavaScript: une fonction est une valeur
- En Java / C#: il existe une conversion automatique pour voir une méthode comme un objet

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 paramétre de l’algorithme générique

Avec les mains ...

Être capable de spécifier un type de fonction en Java

Par ex: (String, String) → int
pour pouvoir déclarer la méthode générique

```
void sort((String, String) → int comparator) {
}
```

Être capable de 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

```
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);
```

Et c’est quoi une lambda

La syntaxe

```
list.sort((String s1, String s2) -> { … });
```

est une écriture simplifiée

- d’une méthode sans nom
- de l’opérateur :: sur cette méthode

```
static int lambda$1(String s1, String s2) { // nom généré
   ...
}
…
list.sort(ClassEnglobante::lambda$1)
```

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 (T, U) → 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 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 fréquent 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) → 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));
```

Operateur ::
(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) {
  ...
  }
}

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

même classe

même nom

interface fonctionnelle

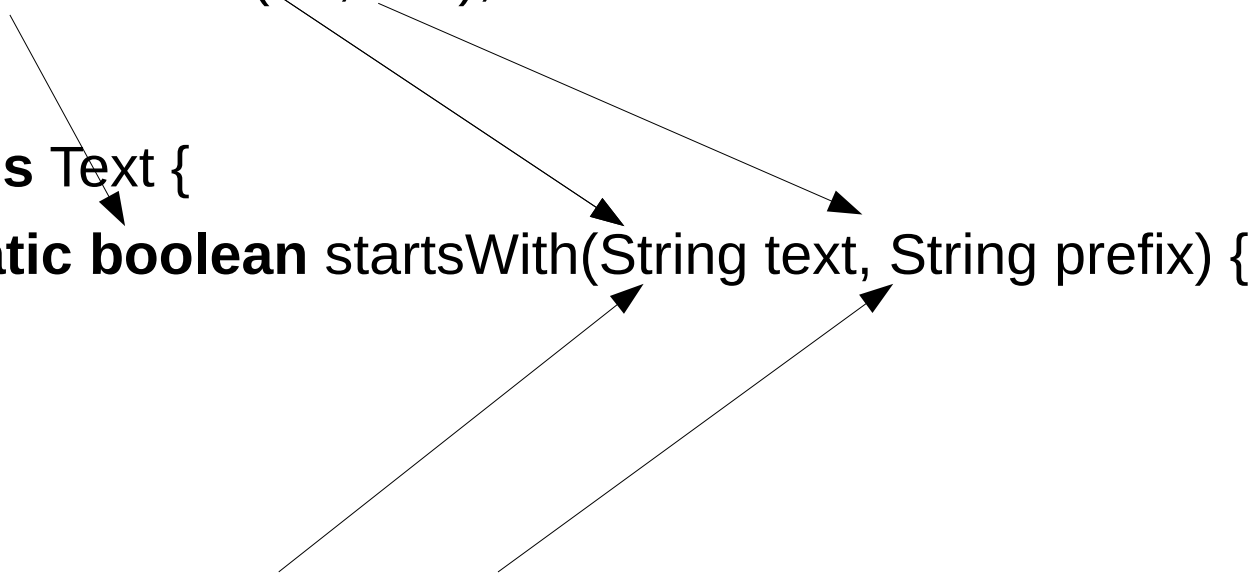
Trouver la bonne méthode

Le compilateur utilise le type des paramétres/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;
```



Opérateur :: et interface fonctionnelle

L’opérateur :: ne marche que si le compilateur peut calculer que le type *target* 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 statique vs Methode d'instance

Attention !
la syntax Class**::**method **ne veut pas** dire que la methode "method" est statique mais juste que la méthode est dans la classe "Class"(ne pas confondre avec Class**.**method)

On peut avoir des conflits

```
class A {
  void m() { … }
  static void m(A a) { }
}
Consumer<A> consumer = A::m; // compile pas car
                             // les 2 méthodes sont valides
```

Methode avec receveur attachée *(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 OCaml !

Opérateur :: et conversions

L’opérateur :: effectue les conversion suivantes

- Contravariance des paramétres
- 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

Operateur :: 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 paramétres

Il n’est pas obligatoire de spécifier le types des paramétres

Comme pour l’opérateur ::, le type des paramétres est déjà présent dans l’interface fonctionnelle

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

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

Syntaxes des lambdas

Lambda block

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

Lambda expression

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

pas de return car il y a une seule expression

Syntaxe en fonction du nombre de paramétres

La syntaxe est “optimisée” pour 1 paramétre (comme en JavaScript, C#, Scala)

1 paramètre, parenthèse pas necessaire

```
s -> s.length()
```

0 paramétre

```
() -> 42
```

2 ou + paramétres

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

Capture de la valeur des variables locales

Les variables capturées doivent être assigné 1 seul fois (effectivement final)

```
static void printTriangle() {
  for(var loop = 1; loop <= 5; loop++) {
    IntStream.range(0, 10).forEach(i -> {
      System.out.print(loop);  // compile pas, loop est modifiée
    });
    System.out.println();
  }
}
```

Capture de la valeur des variables locales (2)

Les variables capturées doivent être assigné 1 seul fois (effectivement final)

```
static void printTriangle() {
  for(var loop = 1; loop <= 5; loop++) {
    var copyOfLoop = loop;
    IntStream.range(0, 10).forEach(i -> {
      System.out.print(copyOfLoop);  // ok
    });
    System.out.println();
  }
}
```

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

Capture de **this**

Une lambda peut aussi capturée this (implicitement)

```
record Person(String name) {
  void printTen() {
    IntStream.range(0, 10).forEach(i -> {
      System.out.println(name + " " + i);
    });
  }
}
```

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)

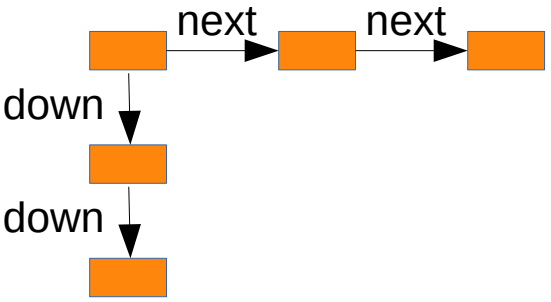
Utilise une interface fonctionnelle pour le typage

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

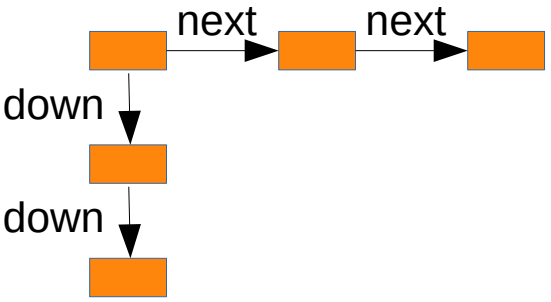
Exemple Simple




```
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

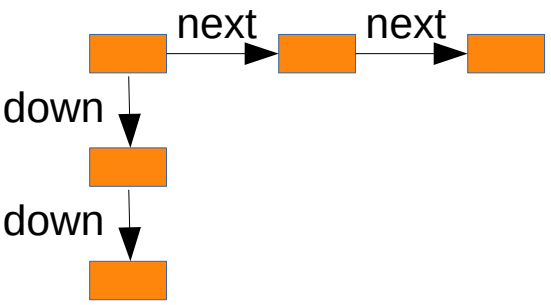



```
record Link(int value, Link next, Link down) {
  private void printAll(???) {
    for(var link = this; link != null; ???) {
      System.out.println(link.value);
    }
  }

  public void printAllNext() {
    printAll(???);
  }

  public void printAllDown() {
    printAll(???);
  }
}
```

Interface fonctionnelle

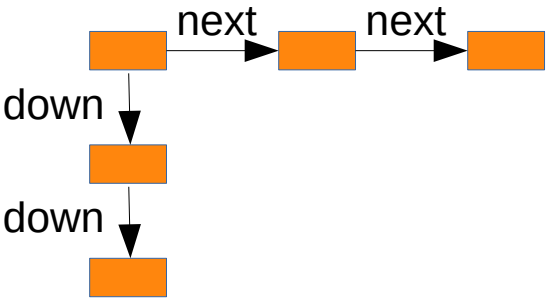



```
record Link(int value, Link next, Link down) {
  private void printAll(UnaryOperator<Link> op) {
    for(var link = this; link != null; link = op.apply(link)) {
      System.out.println(link.value);
    }
  }

  public void printAllNext() {
    printAll(???);
  }

  public void printAllDown() {
    printAll(???);
  }
}
```

Lambda

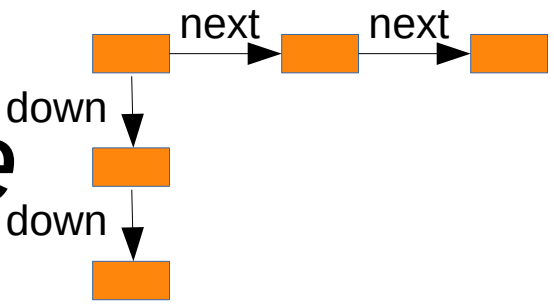



```
record Link(int value, Link next, Link down) {
  private void printAll(UnaryOperator<Link> op) {
    for(var link = this; link != null; link = op.apply(link)) {
      System.out.println(link.value);
    }
  }

  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()

Autre Exemple

Transformer la méthode plus2() en lambda

```
class Example {
  private static int plus2(int x) {
    return x + 2;
  }

  public static void main(String[] args) {
    System.out.println(plus2(1));  // 3
    System.out.println(plus2(3));  // 5
  }
}
```

Autre Exemple (lambda)

Avec une lambda

```
class Example {
    public static void main(String[] args) {
      IntUnaryOperator plus2 = x -> x + 2;
      System.out.println(plus2.applyAsInt(1));  // 3
      System.out.println(plus2.applyAsInt(3));  // 5
    }
  }
}
```

Autre Exemple (suite)

Ecrire une méthode capable de renvoyer une fonction ajoutant une valeur

```
class Example {
  private static ??? adder(int value) {
    return ???;
  }

  public static void main(String[] args) {
    IntUnaryOperator plus2 = adder(2);
    System.out.println(plus2.applyAsInt(1));  // 3
    System.out.println(plus2.applyAsInt(3));  // 5
  }
}
```

Autre Exemple (capture)

Ecrire une méthode capable de renvoyer une fonction ajoutant une valeur

```
class Example {
  private static IntUnaryOperator adder(int value) {
    return x -> x + value;
  }

  public static void main(String[] args) {
    IntUnaryOperator plus2 = adder(2);
    System.out.println(plus2.applyAsInt(1));  // 3
    System.out.println(plus2.applyAsInt(3));  // 5
  }
}
```

Higher Order Functions

Higher order Function

Comme une fonction est une instance d'une interface, on peut definir des méthodes sur des fonctions

```
@FunctionalInterface
interface Code {
  void execute();

  static Code andThen(Code code1, Code code2) {
    return () -> {
      code1.execute();
      code2.execute();
    };
  }
}

…
Code code1 = () → System.out.println("code1");
Code code2 = () → System.out.println("code2");
Code concat = Code.andThen(code1, code2);
```

HoF: Methode par défaut

Ou plus naturellement en utilisant une méthode par défaut

```
@FunctionalInterface
interface Code {
  void execute();

  default Code andThen(Code code) {
    return () -> {
      execute();
      code.execute();
    };
  }
}

…
Code code1 = () → System.out.println("code1");
Code code2 = () → System.out.println("code2");
Code concat = code1.andThen(code2);
```

Code existe déjà dans l'API sous le nom java.lang.Runnable

Classe vs Lambda

Classe vs Lambda

En terme de design, si une interface a une seule méthode

Elle peut être implanter par une classe

```
class MyRunnable implements Runnable {
  private final int id;
  public void run() {
    System.out.println("id " + id);
  }
}
Runnable runnable = new MyRunnable(42);
```

Elle peut être implanter par une lambda

```
Runnable myRunnable(int id) {
  return () → System.out.println("id " + id);
}
Runnable runnable = myRunnable(42);
```

Lambda et java.util

Collection et Map

Le package java.util est un gros consommateur de lambda

Quelques exemples

Supprimer les chaines de caractères vides ("")
Collection<E>.removeIf(Predicate<E> predicate)

- list.removeIf(String::isEmpty);

Parcourir les élements 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::compareIgnoreCase);
```

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 employées 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 employées 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

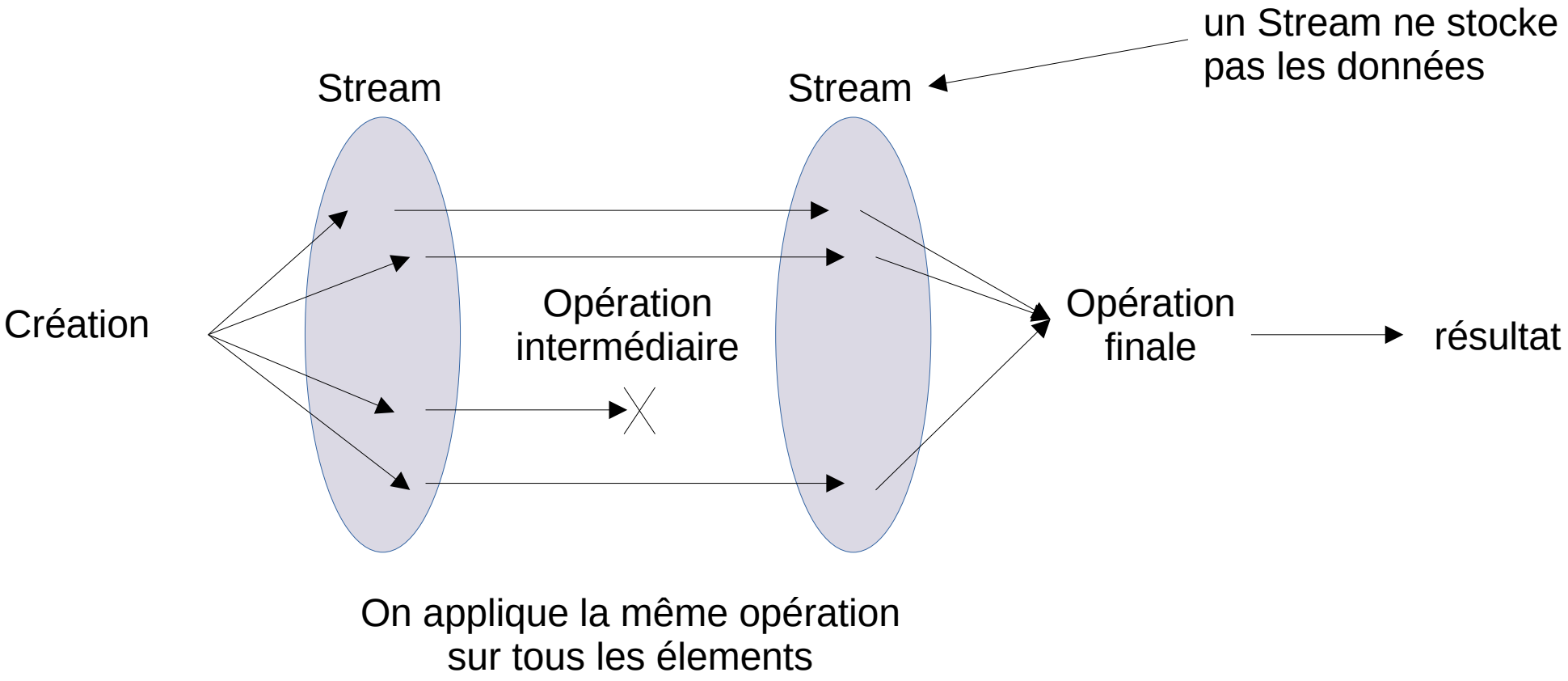
```
SELECT name From persons
  WHERE country = ‘France’ ORDER BY id
```

```
persons.stream()
    .filter(p -> p.country().equals("France"))
    .sorted(Comparator.comparing(Person::id))
    .map(Person::name)
    .toList();
```

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

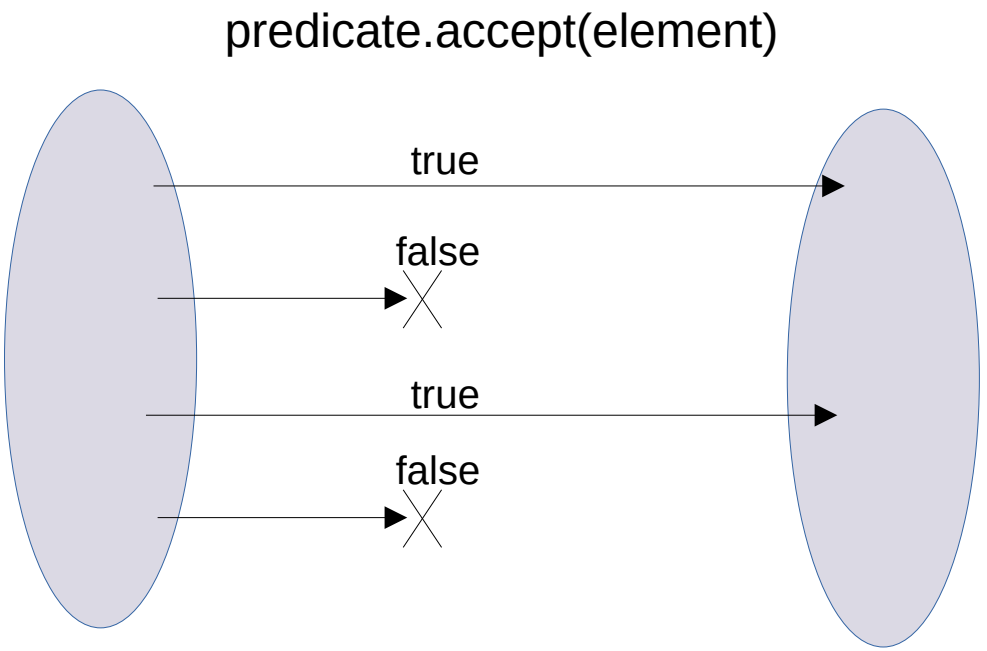
```
try(var stream = Files.list(path)) {
  stream. ...
}
```

Opérations intermédiaires

filter(Predicate<E>)

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

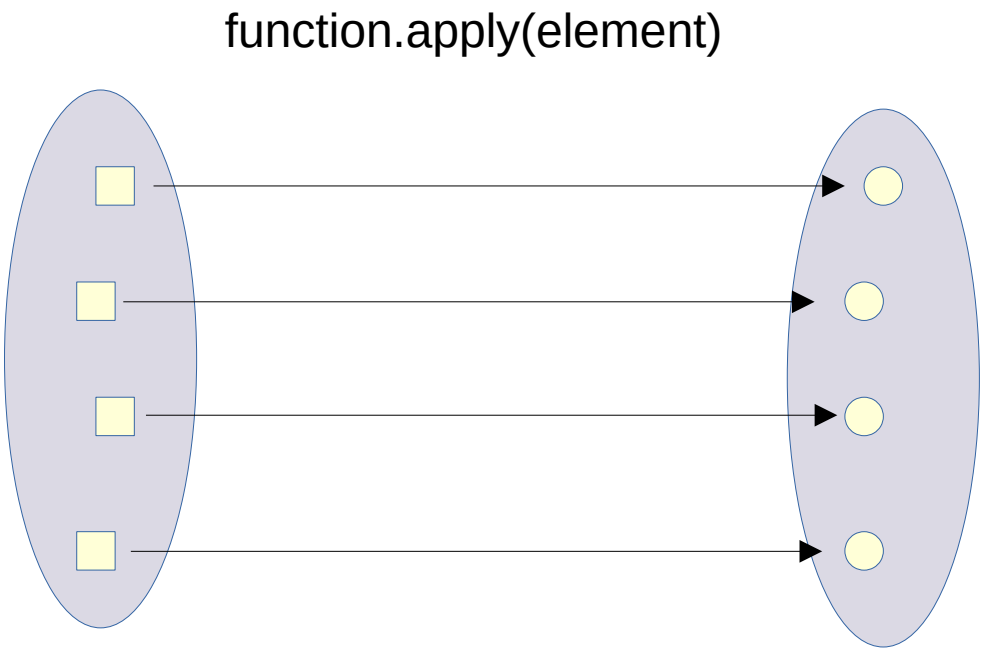
```
stream.filter(person - > person.age() >= 18)
```



map(Function<E,R>)

Transforme chaque élement en appelant la fonction

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

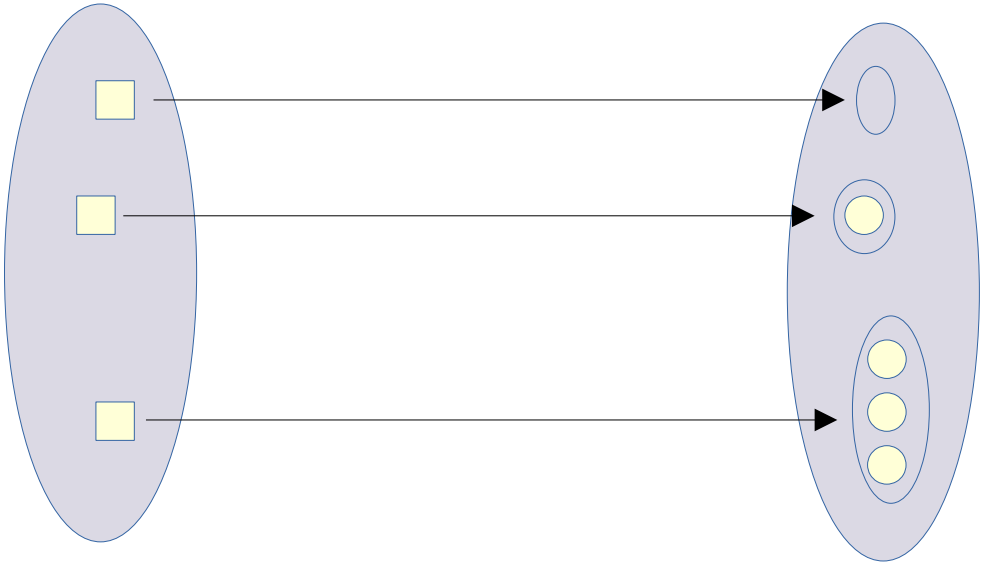


flatMap(Function<E, Stream<R>>)

Transforme chaque élement en appelant la fonction

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

function.apply(element)



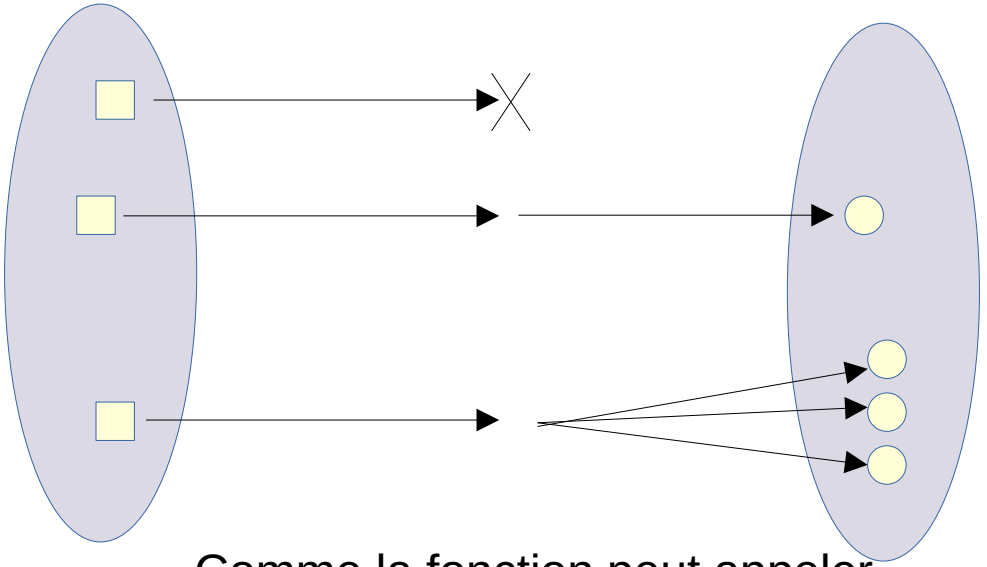
Comme la fonction renvoie un Stream,
elle renvoie de 0 à n élements

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 Collectors

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>
```