

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

Pas un truc nouveau

Tous les langages 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 paramètre** 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 $(T, U) \rightarrow V$

```
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
 $(T, U) \rightarrow \text{boolean}$

```
BiPredicate<String, String> isPrefix =  
    (s, prefix) -> s.startsWith(prefix);  
System.out.println(isPrefix.test("hello", "hell"));
```

BiFunction<T, U, V> représente $(T, U) \rightarrow V$

```
- 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

The diagram illustrates the relationship between a class method and a functional interface method reference. It shows a code snippet for a class `Text` with a static method `startsWith`. Below this, a functional interface `BiPredicate` is shown with a method reference `Text::startsWith`. Arrows indicate that the `Text` class is the same as the class in the method reference, and the `startsWith` method name is the same as the one in the functional interface. The `BiPredicate` interface is identified as a functional interface.

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 resolves the lambda expression `Text::startsWith`. It shows the `BiPredicate` interface with a `test` method that takes two generic parameters `T` and `U` and returns a `boolean`. Below it, the `Text` class has a `startsWith` method that takes two `String` parameters and returns a `boolean`. At the bottom, a lambda expression `BiPredicate<String, String> function = Text::startsWith;` is shown. Arrows indicate the matching process: one arrow points from the `boolean` return type of the interface to the `boolean` return type of the class method; another arrow points from the `T` parameter of the interface to the `String` parameter of the class method; and a third arrow points from the `U` parameter of the interface to the `String` parameter of the class method.

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*

- Methode statique

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

- Methode d'instance

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

- Methode 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 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 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 des variables locales

Les lambdas sont plus que des fonctions anonymes car elles peuvent “capturées” les valeur des variables locales (ou paramètres) dans le scope

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

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 capturer 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)

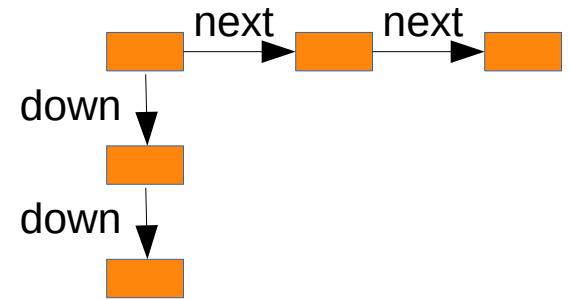
Utilise une interface fonctionnelle pour le
typage

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 ::

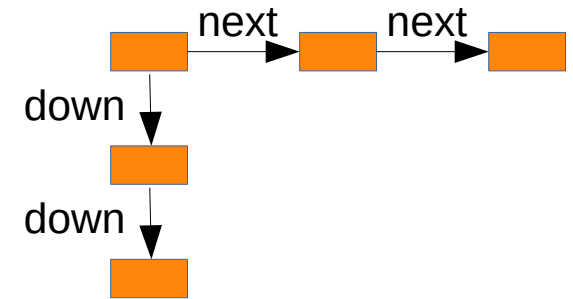
Exemples de factorisation de code

Exemple Simple



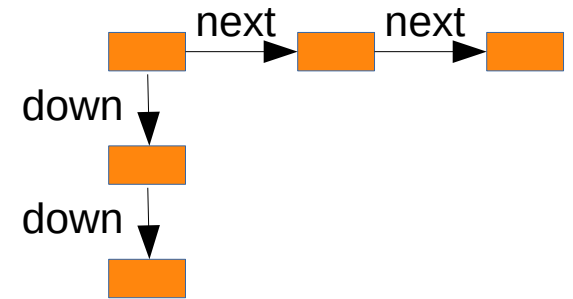
```
record Link(int value, Link next, Link down) {  
    public void printAllNext() {  
        for(var link = this; link != null; link = link.next) {  
            System.out.println(value);  
        }  
    }  
  
    public void printAllDown() {  
        for(var link = this; link != null; link =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

Factorisation



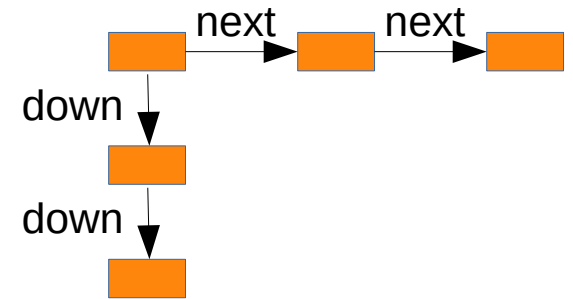
```
record Link(int value, Link next, Link down) {  
    private void printAll(???) {  
        for(var link = this; link != null; ???) {  
            System.out.println(value);  
        }  
    }  
}  
  
public void printAllNext() {  
    printAll(???);  
}  
  
public void printAllDown() {  
  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

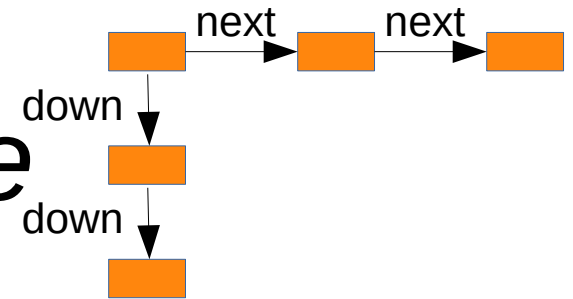
Lambda



```
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(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(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 définir 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);
```

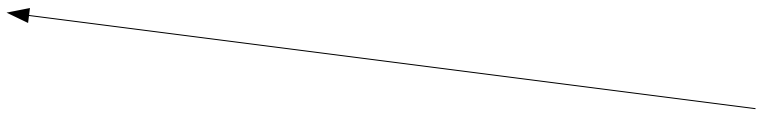
HoF: Methode par défaut

Ou plus naturellement en utilisant une méthode par défaut

@FunctionalInterface

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

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



...

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

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);
```


Type structurel

Il est possible de voir une instance d'une classe comme une instance d'une interface même si la classe n'implante pas l'interface

```
record Dog(String name) {  
    void bark() { ... }  
}
```

```
record Cat(String name) {  
    void meow() { ... }  
}
```

```
var scooby = new Dog("scooby");  
var garfield = new Cat("garfield");  
List<Runnable> sounds = List.of(scooby::bark, garfield::meow);  
sounds.forEach(Runnable::run);
```

Lambda et java.util

Collection et Map

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

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());  
});
```

Compareur

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

API introduite dans la version 1.8

Un Stream ne stocke pas les données mais effectue des transformations jusqu'à une opération terminales

```
List<String> list = ...  
int count = list.stream()  
    .map(...) ←  
    .filter(...) ←  
    .count();
```

Opérations intermédiaires

Opération finale

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)  
        .collect(Collectors.toList());  
}
```

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

Opérations intermédiaires

`.filter(Predicate<E>): Stream<E>`

- garde les éléments pour lequel le prédicate est vrai

`.map(Function<E, R>): Stream<R>`

- transforme les éléments un par un

`.flatMap(Function<E, Stream<R>): Stream<R>`

- tranforme en 0 à n éléments

`.skip(long), limit(long)`

- Saute des éléments, garde uniquement les premiers éléments

`.distinct(), .sorted(Comparator<E>)`

- Sans doublon (cf equals), trié (par le comparateur)

Opérations finales

`.count(): long`

- Compte les éléments

`.forEach(Consumer<E>): void`

- Appel le consumer pour chaque élément

`.findAny(), .findFirst(): Optional<E>`

- Trouver un ou le premier élément (ou `Optional.empty()`)

`.toArray(IntFunction<A>): A`

- Créer un tableau ex: `toArray(String[]::new)`

`.collect(Collector<...>): ...`

- Aggère les éléments en une valeur

java.util.stream.Collectors

La classe Collectors contient un ensemble de Collector prédéfinis

toList(), toSet(), toUnmodifiableList(), to...

stream.collect(toList())

toCollection(Supplier<Collection<E>>)

stream.collect(toCollection(ArrayList::new))

groupingBy(Function<...> mapper): Map<..., List<...>>

Map<String, List<Foo>> map =

stream.collect(groupingBy(Foo::name))

joining(separateur, prefix, suffix): String

stream.map(Object::toString).collect(joining(", ", "[", "]"));

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)  
    .collect(toList());
```

Spécialisation des types primitifs

Pour éviter le boxing (allocation), l'interface Stream est spécialisée pour certains types primitifs

IntStream, LongStream et DoubleStream

La méthode map() est spécialisée

ex: `Stream.mapToInt(Person::age): IntStream`

Comme les types sont numériques, quelques méthodes ont été ajoutées

- `max()`, `sum()`, `average()`, etc
- `IntStream.range(0, 100)`

Stream.forEach() vs Collection.forEach()

java.util.Collection possède déjà une méthode
forEach() (héritée de Iterable)

Au lieu d'écrire

```
collection.stream().forEach(...)
```

il est plus simple d'écrire

```
collection.forEach(...)
```

Lambda et java.io

Fichier et Stream

Méthodes *static* dans `java.nio.file.Files`

- `Files.lines(Path path): Stream<String>`
 - Envoie les lignes une par une
- `Files.list(Path directory): Stream<Path>`
 - Envoie les fichiers (Path) du répertoire un par un
- `Files.walk(Path directory): Stream<Path>`
 - Parcours récursifs des fichiers du répertoire et des sous-répertoires

Stream et Ressources Fichiers

Les Streams gérant des fichiers doivent être **explicitement fermés** en appelant `close()`

- *Resource leak* (ahhhh)
 - `Files.lines(path).forEach(System.out::println);`

- Bonne pratique

```
var lines = Files.lines(path);  
try {  
    lines.forEach(System.out::println);  
} finally {  
    lines.close();  
}
```

Stream et Ressources Fichiers

Il est préférable d'utiliser un try-with-resources

```
try(var lines = Files.lines(path)) {  
    lines.forEach(System.out::println);  
}
```

Attention !

Il faut pas utiliser le try-with resources n'importe comment

- Resource leak

```
try(var lines = Files.lines(path).map(...)) {  
    lines.forEach(System.out::println);  
}
```

si l'appel à map() plante, close() est pas appelée

- Bonne pratique

```
try(var lines = Files.lines(path)) {  
    lines.map(...).forEach(System.out::println);  
}
```