λ - Lambda

Rémi Forax

Wildcards ?

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

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

Universalité

En prog. Objet, l'universalité est le fait qu'un objet puisse représenter

- des données (data + method)
- du code (fonction)

Voir du code comme un objet permet d'abstraire en séparant dans un algo une partie générique, ré-utilisable, et une partie spécifique

Pas nouveau

Tous les languages ont un moyen d'abstraire le code

- C: pointer de fonction
- Lisp: tout est une liste :)
- Python/Javascript: une fonction est une valeur
- Java, C#: conversion automatique d'une fonction annonyme vers un objet

Exemple

List.sort(comparator)

- Séparée en 2 parties
- L'algo de tri en lui-même qui est réutilisable (quicksort, mergesort, timsort, etc)
- La fonction de comparaison qui spécialise l'algo en fonction d'un ordre particulier

La fonction de comparaison est passée en paramètre de l'algo générique

Avec une méthode déjà existante

```
public class Example {
  public static int lengthCmp(String s1, String s2) {
    int diff = Integer.compare(s1.length(), s2.length());
    if (diff != 0) {
      return diff;
    }
    return s1.compareTo(s2);
  }

  public static void main(String[] args) {
     Comparator<String> c = …
     Arrays.sort(args, c);
  }
}
```

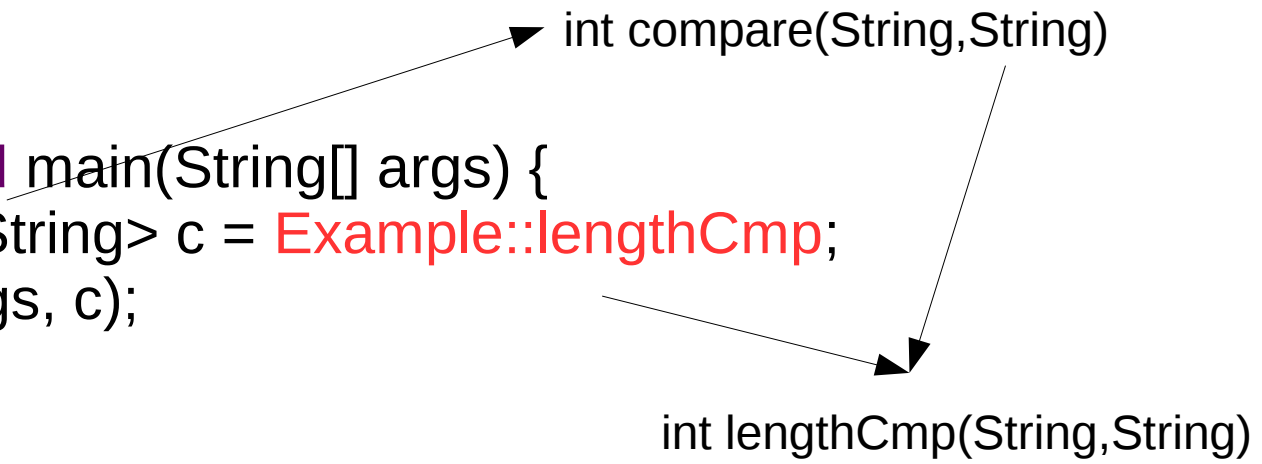
Avec une méthode déjà existante
Method reference

```
public interface Comparator<T> {
  int compare(T t1, T t2);       // functional interface
}

public class Example {
  public static int lengthCmp(String s1, String s2) {
    ...
  }

  public static void main(String[] args) {
     Comparator<String> c = Example::lengthCmp;
     Arrays.sort(args, c);
  }
}
```

Method reference
Inference de la signature

La syntaxe :: ne permet pas de spécifier le type des paramètres
Le type des paramètres est déjà exprimé par l'interface

```
public interface Comparator<T> {
  int compare(T t1, T t2);       // functional interface
}

public class Example {
  public static int lengthCmp(String s1, String s2) {
    ...
  }

  public static void main(String[] args) {
     Comparator<String> c = Example::lengthCmp;
     Arrays.sort(args, c);
  }
}
```



Interface fonctionnelle

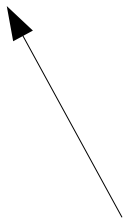
Une interface fonctionnelle est une interface avec une seule méthode abstraite

Même chose qu'un pointeur de fonction en C

```
int *(String,String)
```

En Java :

```
public interface StringComparator {
  int compare(String s1, String s2);       // functional interface
}
```



Le nom de la méthode sert juste à la documentation

@FunctionalInterface

Le compilo vérifie qu'il n'y a qu'une seule méthode abstraite

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

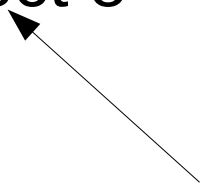
```
@FunctionalInterface
public interface StringComparator {
  int compare(String s1, String s2);
}
```

Auto conversion vers un objet

En Java, une *méthode reference* est automatiquement convertie en une instance d'un objet qui implante l'interface fonctionnelle

Il faut que le *target type* soit une interface fonctionnelle

```
StringComparator c = Example::lengthCmp; // ok
Comparator<String> c2 = Example::lengthCmp; // ok
Object o = Example::lengthCmp; // compile pas :(
```



On ne peut pas extraire le type fonction !

Fonction anonyme (ou lambda)

Autre façon de transformer un code en Objet

```
public interface Comparator<T> {
  int compare(T t1, T t2);       // functional interface
}

public class Example {
  public static void main(String[] args) {
     Comparator<String> c = (String s1, String s2) -> {
       return …;
     };
     Arrays.sort(args, c);
  }
}
```

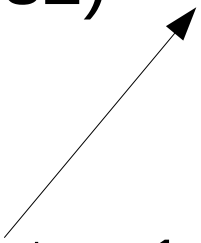
2 syntaxes de lambda

Lambda block

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

Lambda expression

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



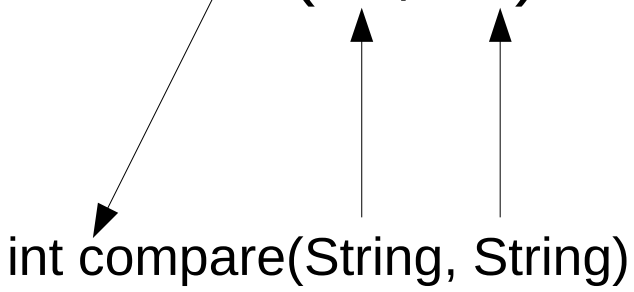
Pas d'accolades, pas de return, 1 seule expression

Inférence du type des paramètres

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

car comme pour les *methodes références* le type est porté par l'interface fonctionnelle

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



Syntaxe en fonction du nombre de paramètres

Comme la plupart des lambdas ont 1 paramètre, la syntaxe est “optimisée” pour 1 paramètre

1 paramètre, parenthèses pas nécessaire

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

0 paramètre

```
() -> 12
```

2 ou + paramètre

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

lambda vs classe anonyme

Syntaxe des classes anonyme (pre-Java 8)

```
Comparator<String> c = new Comparator<String>() {
  @Override
  public boolean compare(String s1, String s2) {
    return s1.compareIgnoreCase(s2);
  }
}
```

Conceptuellement verbeux, on veut une fonction anonyme pas une classe anonyme, et syntaxiquement verbeux

lambda vs classe anonyme (2)

Syntaxe des classes anonyme (pre-Java 8)

```
Comparator<String> c = new Comparator<String>() {
  @Override
  public boolean compare(String s1, String s2) {
    return s1.compareIgnoreCase(s2);
  }
}
```

La VM (ou le JDK) doit pouvoir ré-utiliser la même instance

Il existe 1 seule méthode donc pas nécessaire

déjà dit ! cf type à gauche

lambda vs classe anonyme (3)

Syntaxe des classes anonyme (pre-Java 8)

```
Comparator<String> c = new Comparator<String>() {
  @Override
  public boolean compare(String s1, String s2) {
    return s1.compareIgnoreCase(s2);
  }
}
```

Syntaxe des lambdas:

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

En résumé

On veut avoir des objets qui représentent du code

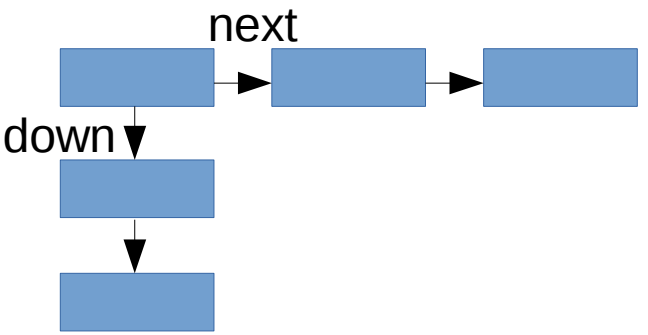
2 syntaxes

- Le code est simple, x -> lambda()
- La méthode existe déjà ou le code est plus compliqué, on donne un nom, Method::reference

Le *target type* doit être une interface fonctionnelle

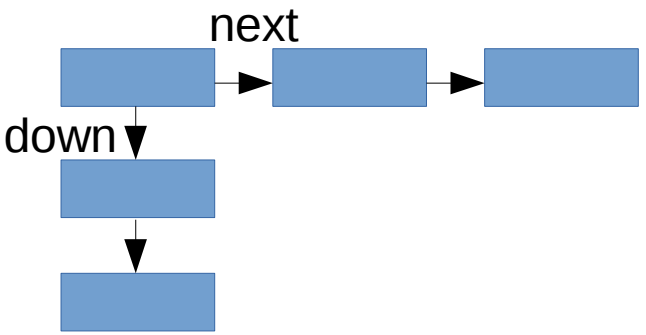
On peut implanter une interface sans classe !

Exemple



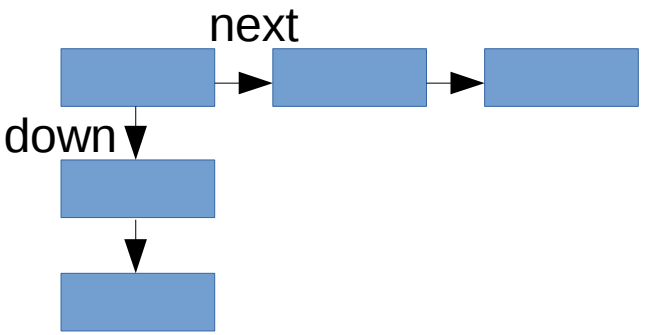

```
class Link {
  private final int value;
  private final Link next;
  private final Link down;

  public void printAllNext() {
    for(Link l = this; l != null; l = l.next) {
      System.out.println(l.value);
    }
  }

  public void printAllDown() {
    for(Link l = this; l != null; l = l.down) {
      System.out.println(l.value);
    }
  }
}
```

Exemple (factorisation de code)




```
class Link {
  private final int value;
  private final Link next;
  private final Link down;

  private void printAll(???) {
    for(Link l = this; l != null; l = ???) {
      System.out.println(l.value);
    }
  }

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

Exemple (factorisation de code)

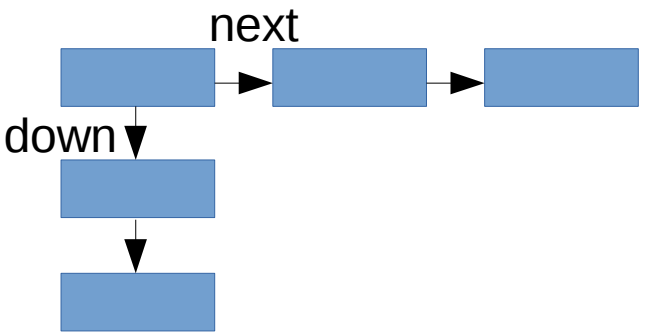



```
class Link {
  private final int value;
  private final Link next;
  private final Link down;

  private void printAll(???) {
    for(Link l = this; l != null; l = ???) {
      System.out.println(l.value);
    }
  }

  public void printAllNext() {
    printAll(l -> l.next);
  }
  public void printAllDown() {
    printAll(l -> l.down);
  }
}
```

Type Fonction correspondant
(Link) → Link

lambdas

Exemple (factorisation de code)




```
class Link {
  private final int value;
  private final Link next;
  private final Link down;

  private void printAll(Fun fun) {
    for(Link l = this; l != null; l = fun.call(l)) {
      System.out.println(l.value);
    }
  }

  public void printAllNext() {
    printAll(l - > l.next);
  }

  public void printAllDown() {
    printAll(l - > l.down);
  }
}
```

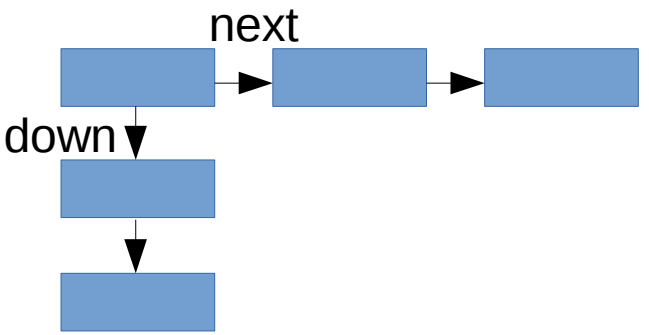
On définie une nouvelle interface

```
@FunctionalInterface
interface Fun {
  Link call(Link l);
}
```

Interfaces fonctionnelles prédéfinies dans java.util.function

signature	*interface*	*method*
() → void	Runnable	run
() → T	Supplier<T>	get
T → void	Consumer<T>	accept
int → void	IntConsumer	accept
T → boolean	Predicate<T>	test
int → boolean	IntPredicate	test
T → R	Function<T,R>	apply
int → R	IntFunction<R>	apply
T → int	ToIntFunction<T>	applyAsInt

Exemple (factorisation de code)

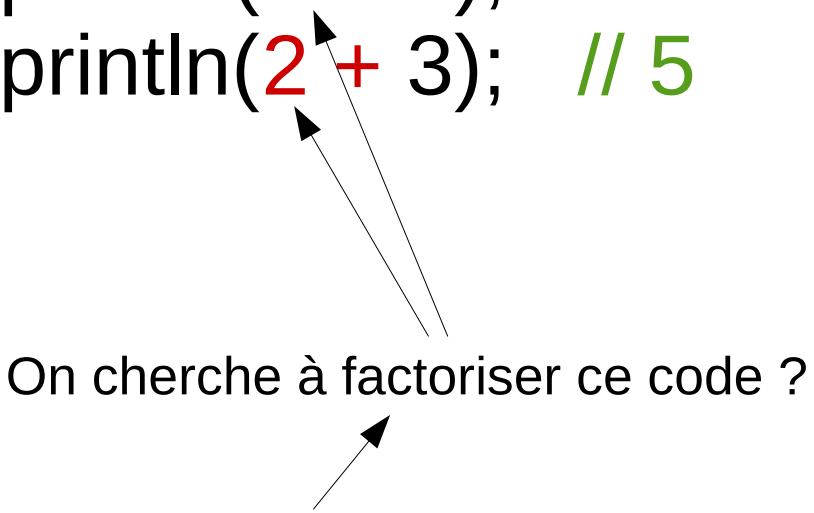



```
class Link {
  private final int value;
  private final Link next;
  private final Link down;

  private void printAll(java.util.Function<Link, Link> fun) {
    for(Link l = this; l != null; l = fun.apply(l)) {
      System.out.println(l.value);
    }
  }

  public void printAllNext() {
    printAll(l - > l.next);
  }

  public void printAllDown() {
    printAll(l - > l.down);
  }
}
```

```
package java.util.function;

@FunctionalInterface
interface Function<T, R> {
  R apply(T t);
}
```

Autre Exemple

```
class Example {


  public static void main(String[] args) {
    System.out.println(2 + 1);   // 3
    System.out.println(2 + 3);   // 5
  }
}
```



Autre Exemple

```
class Example {
```

Ici, le compilo utilise le type de retour qui est une interface fonctionnelle

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

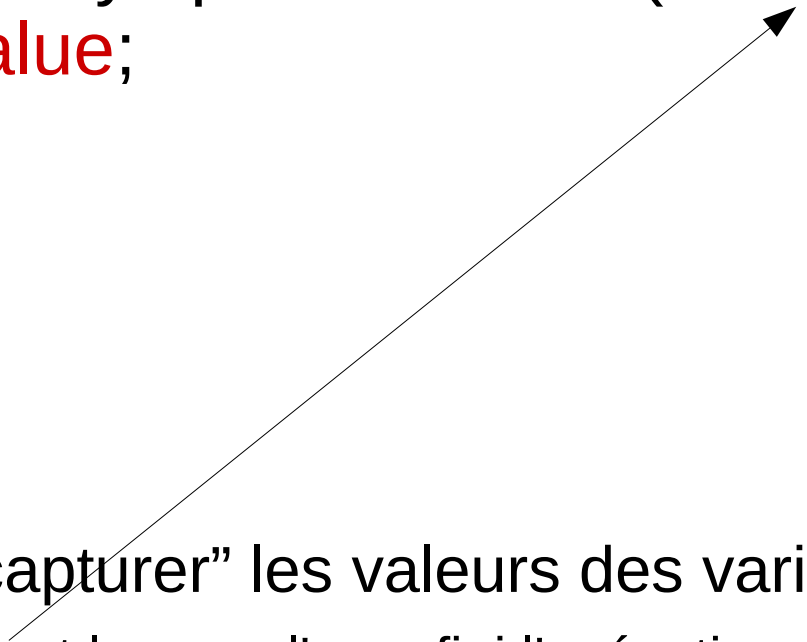
```
package java.util.function;

interface IntUnaryOperator {
  int applyAsInt(int x);
}
```

Et si on paramétrait par la valeur que l'on additionne ?

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

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

Capture !

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

  ...
}
```



une lambda peut “capturer” les valeurs des variables locales

Question: value meurt lorsque l'on a fini l'exécution de adder, donc comment ça marche ???

On capture des valeurs pas des variables !

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

  ...
}
```

une lambda peut “capturer” les valeurs des variables locales

La valeur est recopiée dans l'objet qui instancie l'interface fonctionnelle, donc la valeur existe toujours lorsque l'on sort de la méthode constant

Capture et modification

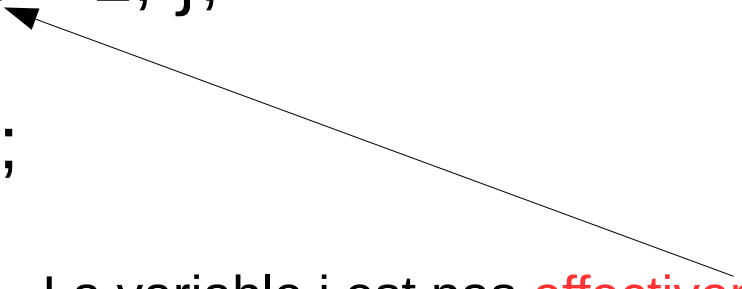
Que se passe t'il si l'on écrit ?

```
public interface Runnable {
  public void run();
}
...
public static void main(String[] args) {
  int i = 0;
  Runnable r = () -> { i = 1; };
  r.run();
  System.out.println(i);
}
```

Capture et modification

Le compilateur râle !

```
public interface Runnable {
  public void run();
}
...
public static void main(String[] args) {
  int i = 0;
  Runnable r = () -> { i = 1; };
  r.run();
  System.out.println(i);
}
```



Effectivement final

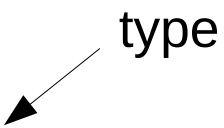
Une variable locale capturée par une lambda est considérée comme non-modifiable (final) implicitement

Car la valeur stockée dans la lambda est une copie de la valeur de la variable

Donc pas d'effet de bord dans les lambdas !

Method reference vs function

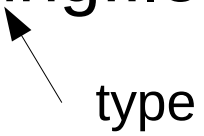
Si la méthode est statique:




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

Si la méthode est une méthode d'instance:

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



Une méthode d'instance est une fonction avec en premier paramètre un object du type de la classe

Method reference/capture de “this”

Si la méthode est statique:

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

type

Si la méthode est une méthode d'instance:

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

type

Si on utilise :: sur une instance, celle-ci est capturée

```
String hello = "hello";
Predicate<Object> p = hello::equals;
```

instance

Method reference/capture de “this”

Exemple, au lieu de

```
list.forEach(e - > System.out.println(e));
```

on peut utiliser la capture d'instance

```
PrintStream out = System.out;
list.forEach(out::println);
```

que l'on peut écrire plus simplement, comme

```
list.forEach(System.out::println);
```

Autre Exemple

Supposons que je veuille valider une liste d'entiers

```
public static boolean inBetween(int value, int min, int max) {
  return value >= min && value <= max;
}
```

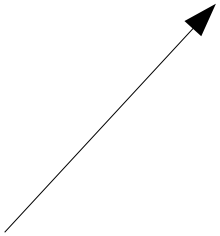
Supposons que min/max soient accessibles à partir de la ligne de commande tandis que les entiers (value) sont obtenus en parsant un fichier

On peut écrire mais ...

```
public static boolean inBetween(int value, int min, int max) { … }
public static boolean parseAndValidate(Path path, int min, int max) {
  for(String line: … from path …) {
    int value = Integer.parseInt(line);
    if (!inBetween(value, min, max)) {
      return false;
    }
  }
  return true;
}
public static void main(String[] args) {
  int min = …
  int max = …
  Path path = ...
  … = parseAndValidate(path, min, max);
}
```

On peut écrire mais … (2)

```
public static boolean inBetween(int value, int min, int max) { … }

public static boolean parseAndValidate(Path path, int min, int max) {
  for(String line: … from path …) {
    int value = Integer.parseInt(line);
    if (!inBetween(value, min, max)) {
      return false;
    }
  }
  return true;
}

public static void main(String[] args) {
  int min = …
  int max = …
  Path path = ...
  … = parseAndValidate(path, min, max);
}
```

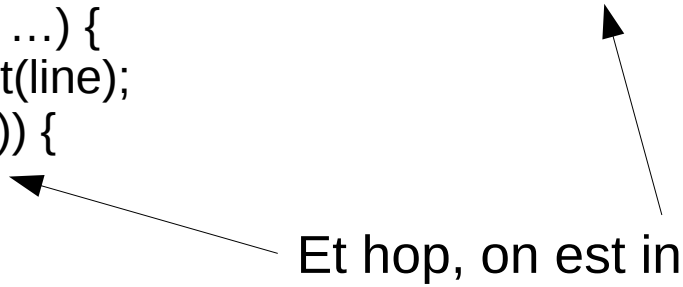


Mais les paramètres min et max n'ont rien à faire ici !

Et si on change la façon dont on valide

Mais c'est mieux d'écrire

```
@FunctionalInterface
public interface IntPredicate {
  boolean accept(int value);
}

public static boolean inBetween(int value, int min, int max) { … }

public static boolean parseAndValidate(Path path, IntPredicate predicate) {
  for(String line: … from path …) {
    int value = Integer.parseInt(line);
    if (!predicate.accept(value)) {
      return false;
    }
  }
  return true;
}

public static void main(String[] args) {
  int min = …
  int max = …
  Path path = ...
  … = parseAndValidate(path, value -> inBetween(value, min, max));
}
```



Lambda et java.util

Collection et Map

Le package java.util est un gros consommateur de lambda

Quelques exemples:

Supprimer les chaines de caractères vide ("")
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) - > {
  ...
  });

Tri

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

- List<String> list = ...
  list.sort((s1, s2) -> s1.compareIgnoreCase(s2));
- que l'on peut aussi écrire
  list.sort(String::compareIgnoreCase);

Et pour une liste de personnes triée par nom

- List<Person> list = …
  list.sort((p1, p2) - > {
    return p1.getName().compareTo(p2.getName());
  });

Trie et Comparateur

L'interface Comparator possède des méthodes static comparing() qui permettent de simplifier l'écriture en indiquant une projection

Une list de personne trier par nom

- List<Person> list = …
  list.sort((p1, p2) - > {
    return p1.getName().compareTo(p2.getName());
  });

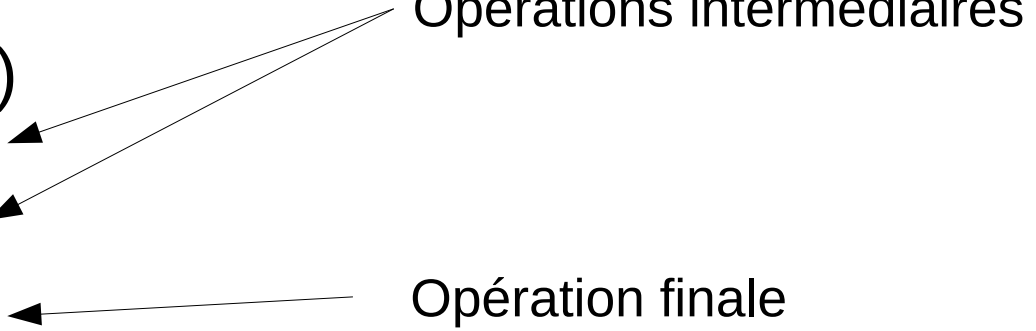
- list.sort(Comparator.comparing(Person::getName));

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 terminale

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



Exemple

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

```
List<String> findEmployeeNameOlderThan50(
   List<Employee> employees) {
   ArrayList<String> list = new ArrayList<>();
   for(Employee employee: employees) {
      if (employee.getAge() >= 50) {
        list.add(employee.getName());
      }
   }
   return list;
}
```

Exemple avec des Streams

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

```
List<String> findEmployeeNameOlderThan50(
   List<Employee> employees) {
   return employees.stream()
     .filter(e -> e.getAge() >= 50)
     .map(Employee::getName)
     .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>

- Choisit les élements si le predicat est vrai

.map(Function<E, R>) → Stream<R>

- Transforme les élements

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

- Applatit l'ensemble des Streams de chaque élement

.skip(), .limit()

- Saute des élements, reduit le nombre d'élement

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

- Sans doublon (par rapport à equals), trié

Opération finale

.count()

- Compte les élements

.forEach(Consumer<E>)

- Applique le consumer pour chaque élement

.allMatch(Predicate<E>), .anyMatch(Predicate<E>)

- Vrai si tout les/au moins un élement(s) match

.findAny(), .findFirst() → Optional<E>

- Trouver un ou le premier, renvoie une posibilité d'élement

.toArray(IntFunction)

- Crée un tableau ex: .toArray(String[]::new)

.reduce(), collect(Collector)

- Aggège les éléments en 1 valeur (resp. non-mutable, mutable)

Stream et SQL

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

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

```
persons.stream()
   .filter(p - > p.getCountry().equals("France"))
   .sorted(Comparator.comparing(Person::getId))
   .map(Person::getName)
   .collect(toList());
```

Effet de bord

Aucun effet de bord sur la collection à l'origine du Stream est autorisé !

- list.stream().forEach(list::add) // pas fun

Aucun effet de bord sur d'autres variables sauf dans

- forEach(Consumer)
- peek(Consumer)
  - Opération intermédiaire appelée pour chaque élément, pratique pour le debug

Spécialisation des types primitifs

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

IntStream, LongStream et DoubleStream

Les méthodes map(), flatMap() sont aussi spécializées

ex: Stream.mapToInt(Person::getAge) → IntStream

Comme les types sont numériques, quelques méthodes ont été ajoutées

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

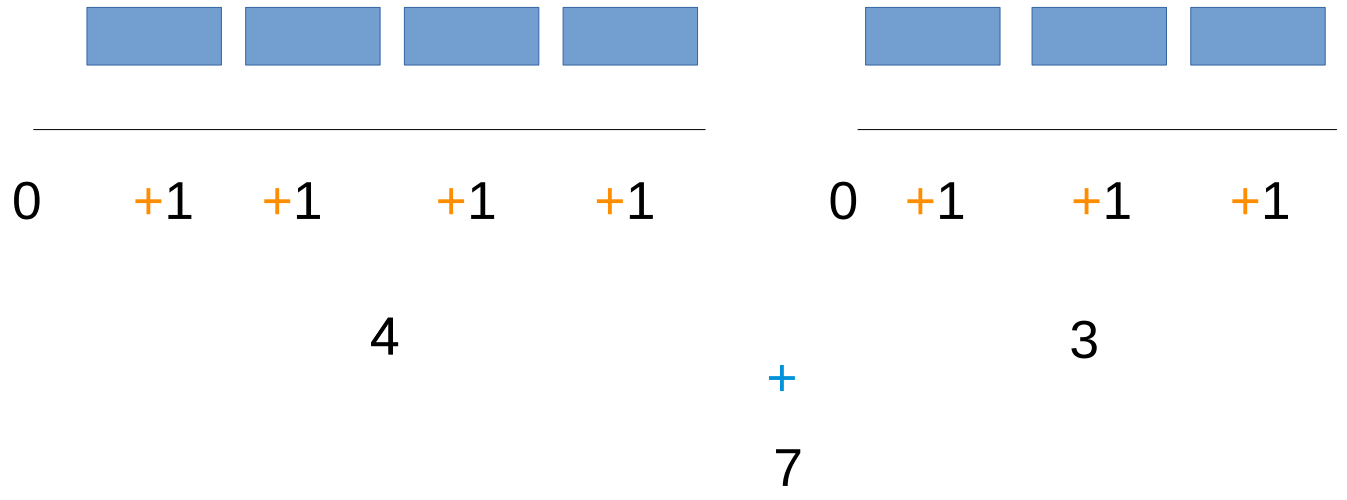
Reduction

Aggrège les données en 1 valeur

.reduce(U initial, BiFunction<U,T,U> aggregate, BinaryOperator<U> combine)

Exemple: // compte les élements

reduce(0, (val, e) - > val + 1,
(val1, va2) - >val1 + val2)



java.util.stream.Collector<T>

Aggège les données de façon mutable

- Supplier qui crée un container ()→A
- Un accumulateur mutable (A, T) →void
- Un combiner (A, A) → A

Exemples:

- Collector.of(ArrayList::new,
  ArrayList::add,
  (x, y) -> { x.addAll(y); return x; });
- Collector.of(StringBuilder::new,
  StringBuilder::append,
  StringBuilder::append);

Collectors

java.util.stream.Collectors contient un ensemble de Collector prédéfinis

toList(), **toSet**()

toCollection(Supplier<Collection>)

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

toMap(Function<T, K> keyMapper, Function<T,V> valueMapper)

- Map<..., ...> map = entryStream.toMap(Entry::getKey, Entry::getValue)

groupingBy(Function<...> mapper)

- Map<String, **List**<Foo>> map = fooStream.collect(groupingBy(Foo::getName))

joining(separator, first, last)

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

Stream Parallel

Permet de distribuer le calcul sur plusieurs coeurs

stream.parallel()

ou collection.parallelStream()

Attention !

- Pas d'effet de bord dans les lambdas (sauf forEach)
- le temps de distribution et de réagrégation doit être plus court que le temps de calcul
- Marche bien si bcp de donnée (100 000?) ou si calcul lent pour chaque élément

Stream.forEach() vs Collection.forEach()

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

Au lieu d'écrire

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

il est possible d'écrire directement

```
collection.forEach(…)
```

qui évite de créer un Stream pour rien.

Lambda et java.io

Fichier et Stream

java.nio.file.Files

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

Stream et Ressources Fichiers

Les Streams gérant les fichiers doivent être **fermés** en appelant close()

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

- Pas de *resource leak*:
  ```
  Stream<String> lines = lines(path);
  try {
    lines.forEach(System.out::println);
  } finally {
     lines.close();
  }
  ```

Stream et Ressources Fichiers

Préférable d'utiliser un try-with-resources

- Pas de *resource leak*:

```
try(Stream<String> lines = lines(path)) {
  lines.forEach(System.out::println);
}
```

Attention !

Préférable d'utiliser un try-with-resources

- *Resource leak*:

```
try(Stream<String> lines = lines(path).map(...)) {
  lines.forEach(System.out::println);
}
```

Si le map() plante le close() est pas fait

- Pas de *resource leak*:

```
try(Stream<String> lines = lines(path)) {
  lines.map(...).forEach(System.out::println);
}
```

Principes de la prog fonctionnelle

Application partielle

Composition

Curryfication

Application partielle

On appelle application partielle le fait de calculer partiellement certains arguments d'une fonction

```
interface UnaryOperator {
  int apply(int v);
}
interface IntBinaryOperator {
  int apply(int v1, int v2);
}

IntBinaryOperator add = (a, b) - > a + b;
UnaryOperator plus1 = ...
```

Application partielle (2)

On appelle application partielle le fait de calculer partiellement certains arguments d'une fonction

```
interface UnaryOperator {
  int apply(int v);
}
interface IntBinaryOperator {
  int apply(int v1, int v2);
}

IntBinaryOperator add = (a, b) - > a + b;
UnaryOperator plus1 = a -> add.apply(a, 1);
```

Exemple (sans lambda)

On a un code qui permet de créer des jouets ou des chaises pour une couleur donnée

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }


static <T> List<T> createAllToys(int count, Color color) {
  return range(0, count).mapToObj(i -> createToy(color)).collect(toList());
}
static <T> List<T> createAllChairs(int count, Color color) {
  return range(0, count).mapToObj(i -> createChair(color)).collect(toList());
}


public static void main(String[] args) {
  Color color = …
  ... = (...)? createAllToys(10_000, color):
               createAllChairs(10_000, color);
```

Exemple

On partage le code de création des Toy|Chair

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }

static <T> List<T> createAll(int count, Color color,
                             Function<Color, T> factory) {
  return range(0, count)
        .mapToObj(i - > factory.apply(color))
        .collect(toList());
}

public static void main(String[] args) {
  Color color = …
  Function<Color, Object> factory = (...)?
      Example::createToy: Example::createChair;
  … = createAll(10_000, color, factory);
```

Exemple (2)

Attention, la couleur ne devrait pas être un paramètre de createAll !

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }

static <T> List<T> createAll(int count, Color color,
                             Supplier<T> factory) {
  return range(0, count)
        .mapToObj(i - > factory.create())
        .collect(toList());
}

public static void main(String[] args) {
  Color color = …
  Function<Color, Object> factory = (...)?
     Example::createToy: Example::createChair;

  Supplier<Object> supplier = ??
  … = createAll(10_000, supplier);
```

Euh ?

Exemple (3)

withColor est une application partielle de la factory avec la couleur

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }

static <T> Supplier<T> withColor(Function<Color, T> fun, Color color) {
  return ??
}
static <T> List<T> createAll(int count, Supplier<T> factory) {
  return range(0, count)
        .mapToObj(i - > factory.create())
        .collect(toList());
}
public static void main(String[] args) {
  Color color = …
  Function<Color, Object> factory = (...)?
     Example::createToy: Example::createChair;
  Supplier<Object> supplier = withColor(factory, color);
  … = createAll(10_000, supplier);
```

Exemple (4)

withColor est une application partielle de la factory avec la couleur

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }

static <T> Supplier<T> withColor(Function<Color, T> fun, Color color) {
  return () -> fun.apply(color);
}
static <T> List<T> createAll(int count, Supplier<T> factory) {
  return range(0, count)
        .mapToObj(i - > factory.create())
        .collect(toList());
}
public static void main(String[] args) {
  Color color = …
  Function<Color, Object> factory = (...)?
     Example::createToy: Example::createChair;
  Supplier<Object> supplier = withColor(factory, color);
  … = createAll(10_000, supplier);
```

Exemple (5)

En fait, le mécanisme est plus général et indépendant du fait que cela soit une couleur

```
static Toy createToy(Color color) { … }
static Chair createChair(Color color) { … }

static <T, V> Supplier<T> partial(Function<V, T> fun, V value) {
  return () -> fun.apply(value);
}
static <T> List<T> createAll(int count, Supplier<T> factory) {
  return range(0, count)
        .mapToObj(i - > factory.create())
        .collect(toList());
}
public static void main(String[] args) {
  Color color = …
  Function<Color, Object> factory = (...)?
      Example::createToy: Example::createChair;
  Supplier<Object> supplier = partial(factory, color);
  … = createAll(10_000, supplier);
```

Method Reference d'instance

Voir une instance d'une interface fonctionnelle comme une instance d'une autre interface fonctionnelle *compatible*

Par ex:

```
interface Function<T, R> { R apply(T t); }
interface Predicate<T> { boolean accept(T t); }


Function<String, Boolean> fun = s -> true;
Predicate<String> pred = ??
```

Method Reference d'instance (2)

Par ex:

```
interface Function<T, R> { R apply(T t); }
interface Predicate<T> { boolean accept(T t); }

Function<String, Boolean> fun = s -> true;
Predicate<String> pred = fun::apply;

Predicate<Integer> pred2 = i -> false;
Function<Integer, Boolean> fun2 = pred2::accept;
```

Composition de fonction

On veut appliquer deux transformations successives sur les éléments d'une liste

```
static List<String> transform(List<String> list,
        Function<String, String> transform1,
        Function<String, String> transform2) {
  return list.stream()
        .map(transform1)
        .map(transform2)
        .collect(toList());
}

… = transform(String::toUpperCase, String::toLowerCase);
```

Composition de fonction

Idiot !, il vaux mieux faire 1 map() et dire que le résultat de la première fonction est l'argument de la seconde

```
static Function<String, String> compose(
    Function<String, String> fun1,
    Function<String, String> fun2) {
  return ??
}
```

```
static List<String> transform(List<String> list,
        Function<String, String> transform) {
  return list.stream().map(transform).collect(toList());
}
```

```
… = transform(l, compose(String::toUpperCase, String::toLowerCase));
```

Composition de fonction

Idiot !, il vaux mieux faire 1 map() et dire que le résultat de la première fonction est l'argument de la seconde

```
static Function<String, String> compose(
    Function<String, String> fun1,
    Function<String, String> fun2) {
  return s -> fun2.appply(fun1.apply(s));
}
```

```
static List<String> transform(List<String> list,
        Function<String, String> transform) {
  return list.stream().map(transform).collect(toList());
}
```

```
… = transform(l, compose(String::toUpperCase, String::toLowerCase));
```

Composition de fonction

Il existe déjà deux méthodes par défaut compose() et andThen() sur java.util.function.Function mais ...

```
static List<String> transform(List<String> list,
        Function<String, String> transform) {
  return list.stream().map(transform).collect(toList());
}

… = transform(l,
      String::toUpperCase.andThen(String::toLowerCase));
```

ne compile pas car le compilo ne connait pas le type de String::toUpperCase et donc il ne peut pas vérifier si c'est une interface fonctionnelle

Composition de fonction

Il existe déjà deux méthodes par défaut compose() et andThen() sur java.util.function.Function mais ...

```
static List<String> transform(List<String> list,
        Function<String, String> transform) {
  return list.stream().map(transform).collect(toList());
}

Function<String, String> toUpperCase= String::toUpperCase;
… = transform(l, toUpperCase.andThen(String::toLowerCase));
```

Il faut déclarer une variable locale intermédiaire
(on peut faire un cast aussi mais le code est crado)

Curryfication

Transformer une fonction à plusieurs paramètres en une fonction de fonction de …

L'appel de fonction devient l'application partielle

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

static Function<Integer, ...> curry(BiFunction<...> binOp) {
  return …; //TODO
}

BiFunction<Integer, Integer, Integer> op = (a, b) -> a + b;
Function<Integer, … /*TODO*/> c = curry(op);
System.out.println(c.apply(3).apply(2)); // 5
```

Curryfication (2)

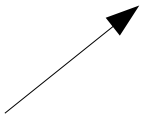
```
interface Function<T,R> { R apply(T t); }
interface BiFunction<T,U,R> { R apply(T t, U u); }

static Function<Integer, Function<Integer, Integer>>
    curry(BiFunction<Integer, Integer, Integer> binOp) {
  return …; //TODO
}

BiFunction<Integer, Integer, Integer> op = (a, b) -> a + b;
Function<Integer, Function<Integer, Integer>> c =
        curry(op);
System.out.println(c.apply(3).apply(2)); // 5
```

Curryfication (3)

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

static <T,U,R> Function<T, Function<U, R>>
    curry(BiFunction<T, U, R> binOp) {
  return …; //TODO
}

BiFunction<Integer, Integer, Integer> op = (a, b) -> a + b;
Function<Integer, Function<Integer, Integer>> c =
        curry(op);
System.out.println(c.apply(3).apply(2)); // 5
```

Curryfication (4)

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

static <T,U,R> Function<T, Function<U, R>>
    curry(BiFunction<T, U, R> binOp) {
  return t -> u -> binOp.apply(t, u);
}

BiFunction<Integer, Integer, Integer> op = (a, b) -> a + b;
Function<Integer, Function<Integer, Integer>> c =
        curry(op);
System.out.println(c.apply(3).apply(2)); // 5
```



Le premier appel à apply est une application partielle !