Fonctions anonymes (ou lambdas)

Pratique de la programmation orientée-objet

Michel Schinz – 2015-03-23

Tri inverse

Admettons que l'on désire écrire une classe `Sorter` dotée d'une méthode `sortInv` qui trie, en ordre lexicographique inverse, la liste de chaînes de caractères qu'on lui passe :

```
public final class Sorter {
  public static void sortInv(List<String> l){
    // … code
  }
}
```

Comment faire ?

Comparateur inversant

Trier la liste par ordre naturel puis inverser l'ordre de ses éléments n'est pas une bonne solution, car elle nécessite deux parcours de ceux-ci.

Une bien meilleure solution consiste à utiliser la méthode `sort` de `Collections` en lui passant un « comparateur inversant » qui inverse l'ordre naturel des chaînes :

```
public static void sortInv(List<String> l) {
  Collections.sort(l, ???);
}
```

Reste à savoir comment écrire le comparateur.

Comparateur inversant

Le comparateur inversant peut se définir sous la forme d'une classe séparée, privée et imbriquée statiquement dans la classe `Sorter` :

```
public static void sortInv(List<String> l) {
  Collections.sort(l, new InvComparator());
}
private static class InvComparator
    implements Comparator<String> {
  @Override
  public int compare(String s1, String s2) {
    // ???
  }
}
```

Comparateur inversant

Pour simplifier ce code, on peut bien entendu utiliser une classe anonyme, imbriquée dans la méthode `sortInv` :

```
public static void sortInv(List<String> l) {
  Collections.sort(l,
    new Comparator<String>() {
      @Override
      public int compare(String s1,
                         String s2) {
        return s2.compareTo(s1);
      }
    });
}
```

Cela reste lourd pour quelque chose de si simple…

Comparateur inversant

Depuis la version 8 de Java, une syntaxe beaucoup plus légère permet de définir une telle instance d'une classe anonyme implémentant une interface dotée d'une seule méthode. Elle permet de récrire `sortInv` ainsi :

```
public static void sortInv(List<String> l) {
  Collections.sort(l,
    (String s1, String s2) -> {
      return s2.compareTo(s1);
    });
}
```

Cette construction est connue sous le nom de *fonction anonyme* ou *lambda expression*.

Comparateur inversant

Le comparateur inversant peut encore se simplifier en tirant parti du fait que, dans une fonction anonyme :

- le type des paramètres est optionnel car inféré,
- si le corps de la fonction est composé d'une seule expression, elle peut être écrite telle quelle, sans accolades englobantes ni `return`.

On obtient une version simple et élégante de `sortInv` :

```
public static void sortInv(List<String> l) {
  Collections.sort(l,
    (s1, s2) -> s2.compareTo(s1));
}
```

type des paramètres inféré

`return` implicite

Fonctions anonymes

Interface fonctionnelle

Une interface est appelée **interface fonctionnelle** (*functional interface*) si elle ne possède qu'une seule méthode abstraite.

Par exemple, l'interface `Comparator` est une interface fonctionnelle, car elle ne possède qu'une seule méthode abstraite, à savoir `compare`.

Un autre exemple d'interface fonctionnelle est l'interface `RealFunction` ci-dessous, qui pourrait représenter une fonction mathématique de $\mathbb{R}$ dans $\mathbb{R}$:

```
public interface RealFunction {
  public double valueAt(double x);
}
```

Fonctions anonymes

En Java, une **fonction anonyme** (*anonymous function*), aussi appelée **lambda expression** ou parfois **fermeture** (*closure*) est une expression créant une instance d'une classe anonyme qui implémente une interface fonctionnelle.
La fonction anonyme spécifie uniquement les arguments et le corps de l'unique méthode abstraite de l'interface fonctionnelle. Ceux-ci sont séparés par une flèche symbolisée par la chaîne -> :

arguments -> *corps*

Fonctions anonymes

Attention : une fonction anonyme ne peut apparaître que dans un contexte qui attend une valeur dont le type est une interface fonctionnelle.

Par exemple, l'expression suivante est valide :

```
Comparator<Integer> c =
    (x, y) -> x.compareTo(y);
```

car `Comparator` est une interface fonctionnelle. Par contre, l'expression suivante ne l'est pas :

```
Object c =
    (x, y) -> x.compareTo(y);
```

car `Object` n'est pas une interface fonctionnelle. Java ne peut donc savoir à quelle méthode doit correspondre le code de la fonction anonyme.

Paramètres

Dans leur forme générale, les paramètres d'une fonction anonyme sont entourés de parenthèses et leur type est spécifié avant leur nom, comme d'habitude.

Par exemple, une fonction anonyme à deux arguments, le premier de type `Integer` et le second de type `String`, peut s'écrire ainsi :

```
(Integer x, String y) -> // … corps
```

Cette notation peut être allégée de deux manières :

1. le type des arguments peut généralement être omis, car inféré par le compilateur,
2. lorsque la fonction ne prend qu'un seul paramètre, les parenthèses peuvent être omises.

Fonctions-blocs

Dans sa forme la plus générale, le corps d'une fonction anonyme est constitué d'un bloc entouré d'accolades. Comme toujours, si la fonction anonyme retourne une valeur – c-à-d que son type de retour est autre chose que `void` – celle-ci l'est via l'énoncé `return`.

Par exemple, le comparateur ci-dessous compare deux chaînes d'abord par longueur puis par ordre alphabétique :

```
Comparator<String> c = (s1, s2) -> {
  int lc = Integer.compare(s1.length(),
                           s2.length());
  return lc != 0 ? lc : s1.compareTo(s2);
};
```

Fonction-expression

Si le corps de la fonction anonyme est constitué d'une seule expression, alors on peut l'utiliser en lieu et place du bloc. Cela implique de supprimer les accolades englobantes et l'énoncé `return`.

Par exemple, le comparateur ci-dessous compare deux chaînes uniquement par longueur :

```
Comparator<String> c = (s1, s2) ->
  Integer.compare(s1.length(), s2.length());
```

sous forme de bloc, le corps ce même comparateur s'écrit :

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

Méthodes d'interfaces

Note : contrairement à ce que vous avez appris jusqu'à présent, les interfaces peuvent – depuis Java 8 – contenir des méthodes concrètes, qui peuvent être :

- des méthodes statiques, ou
- des **méthodes par défaut** (*default methods*), non statiques, qui sont héritées par toutes les classes qui implémentent l'interface et ne les redéfinissent pas.

Pour être considérée comme fonctionnelle, une interface doit avoir exactement une méthode *abstraite* mais peut avoir un nombre quelconque de telles méthodes concrètes.

Méthodes par défaut

L'interface `Comparable`, par exemple, comporte un grand nombre de méthodes statiques et de méthodes par défaut, qui n'ont pas été présentées jusqu'ici.

Parmi les méthodes par défaut, on trouve p.ex. `reversed`, qui retourne un comparateur inverse de celui auquel on l'applique. Elle pourrait être définie ainsi :

```
public interface Comparator<T> {
  public int compare(T o1, T o2);
  public default Comparator<T> reversed() {
    return (o1, o2) -> compare(o2, o1);
  }
  // … autres méthodes par défaut/statiques
}
```

Interfaces fonctionnelles de l'API Java

`java.util.function`

On l'a vu, les fonctions anonymes ne peuvent être utilisées que dans un contexte qui attend une valeur dont le type est une interface fonctionnelle. Dès lors, il est utile d'avoir à disposition un certain nombre de telles interfaces, couvrant les principaux cas d'utilisation.
Le paquetage `java.util.function` a pour but de définir un ensemble d'interfaces fonctionnelles, et les principales d'entre-elles sont présentées ci-après.

Interface Function

L'interface `Function` représente une fonction à un argument. Le type de cet argument et le type de retour de la fonction sont les paramètres de type de cette interface, nommés respectivement T et R :

```
public interface Function<T, R> {
  public R apply(T x);
}
```

Par exemple, la fonction qui retourne la longueur d'une chaîne pourrait se définir ainsi :

```
Function<String, Integer> stringLength =
  s -> s.length();
```

et s'utiliser de la sorte :

```
stringLength.apply("bonjour"); // → 7
```

Composition de fonctions

En plus de la méthode abstraite `apply`, l'interface `Function` offre une méthode `compose` permettant de composer deux fonctions entre elles, au sens mathématique.

Par exemple, deux fonctions sur les entier f et g et leur composition $f \circ g$ peuvent se définir ainsi :

```
Function<Integer,Integer> f = x -> x + x;
Function<Integer,Integer> g = x -> x + 1;
Function<Integer,Integer> fg = f.compose(g);
```

et la composition peut s'utiliser ensuite comme toute autre fonction :

```
fg.apply(10); // → 22
```

Interface UnaryOperator

L'interface `UnaryOperator` représente un cas particulier de fonction à un argument dont le type de l'argument et le type de retour sont identiques :

```
public interface UnaryOperator<T>
  extends Function<T, T> {}
```

Par exemple, la fonction de calcul de valeur absolue sur les réels est un opérateur unaire qui pourrait se définir ainsi :

```
UnaryOperator<Double> abs =
  x -> Math.abs(x);
```

et s'utiliser de la sorte :

```
abs.apply(-1.2);    // → 1.2
abs.apply(Math.PI); // → 3.1415…
```

Interface `BiFunction`

L'interface `BiFunction` représente une fonction à *deux* arguments dont les types sont donnés par les paramètres de type `T` et `U`, le type du résultat étant donné par `R` :

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

Par exemple, la fonction prenant une chaîne et un index et retournant le caractère de la chaîne à cet index peut se définir ainsi :

```
BiFunction<String,Integer,Character> charAt=
  (s, i) -> s.charAt(i);
```

et s'utiliser de la sorte :

```
charAt.apply("hello", 2); // → l
```

Interface BinaryOperator

L'interface `BinaryOperator` représente un cas particulier de fonction à deux arguments dont le type des arguments et le type de retour sont identiques :

```
public interface BinaryOperator<T>
  extends BiFunction<T, T, T> {}
```

Par exemple, l'addition sur les entiers est un opérateur binaire qui pourrait se définir ainsi :

```
BinaryOperator<Integer> plus =
  (x, y) -> x + y;
```

et s'utiliser de la sorte :

```
plus.apply(8, 9); // → 17
```

Interface Predicate

L'interface `Predicate` représente un prédicat à un argument, c-à-d une fonction à un argument retournant un booléen :

```
public interface Predicate<T> {
  public boolean test(T x);
}
```

Par exemple, le prédicat déterminant si une chaîne de caractères est vide pourrait se définir ainsi :

```
Predicate<String> stringIsEmpty =
  x -> x.isEmpty();
```

et s'utiliser de la sorte :

```
stringIsEmpty.test("not empty!"); // → false
stringIsEmpty.test("");           // → true
```

Interface BiPredicate

L'interface `BiPredicate`, similaire à `Predicate`, représente un prédicat à deux arguments :

```
public interface BiPredicate<T, U> {
  public boolean test(T x, U y);
}
```

Par exemple, le prédicat testant si une chaîne est la représentation textuelle d'un objet pourrait se définir ainsi :

```
BiPredicate<String, Object> isTextReprOf =
  (s, o) -> s.equals(o.toString());
```

et s'utiliser de la sorte :

```
isTextReprOf.test("1", 1);   // → true
isTextReprOf.test("2", "a"); // → false
```

Composition de prédicats

Les interfaces `Predicate` et `BiPredicate` offrent des méthodes par défaut pour obtenir des prédicats à partir de prédicats existants :

- `and` calcule la conjonction de deux prédicats,
- `or` calcule la disjonction de deux prédicats,
- `negate` calcule la négation d'un prédicat.

On peut les utiliser ainsi :

```
Predicate<Integer> p = x -> x >= 0;
Predicate<Integer> q = x -> x <= 5;
Predicate<Integer> r = p.and(q);
Predicate<Integer> s = p.or(q);
Predicate<Integer> t = s.negate();
```

Consumer

L'interface `Consumer` représente un consommateur de valeur, c-à-d une fonction à un argument ne retournant rien :

```
public interface Consumer<T> {
  public void accept(T x);
}
```

Par exemple, la fonction imprimant une chaîne à l'écran est un consommateur de chaîne et peut se définir ainsi :

```
Consumer<String> printString =
  s -> { System.out.println(s); };
```

et s'utiliser de la sorte :

```
printString.accept("hé"); // affiche hé
```

Supplier

L'interface `Supplier` représente un fournisseur de valeurs, c-à-d une fonction sans argument retournant une valeur d'un type donné :

```
public interface Supplier<T> {
  T get();
}
```

Par exemple, un fournisseur constant ne produisant que l'entier 0 pourrait se définir ainsi :

```
Supplier<Integer> zero =
  () -> 0;
```

et s'utiliser de la sorte :

```
zero.get(); // → 0
```

Fonctions
et collections

Fonctions et collections

Lors de l'introduction des fonctions anonymes dans la version 8 de Java, quelques méthodes les utilisant ont été ajoutées aux collections.
Nous n'en examinerons que quelques-unes ici, pour donner une idée de ce qu'apportent les fonctions anonymes dans ce domaine.

`Iterable.forEach`

La méthode `forEach` de l'interface `Iterable` prend un consommateur en argument et l'applique à chaque élément de l'entité itérable.

Etant donné que l'interface `Collection` étend l'interface `Iterable`, cette méthode est disponible – entre autres – sur les listes et les ensembles.

Par exemple, pour afficher les éléments d'une liste – chacun sur une ligne séparée – on peut écrire :

```
List<Integer> l =
  Arrays.asList(1, 2, 3, 4, 5);
l.forEach(x -> { System.out.println(x); });
```

`Collection.removeIf`

La méthode `removeIf` de l'interface `Collection` prend un prédicat en argument et supprime tous les éléments de la collection qui le satisfont.

Par exemple, l'extrait de programme ci-dessous supprime tous les nombres pairs de la liste :

```
List<Integer> l = new ArrayList<>(
  Arrays.asList(1, 2, 3, 4, 5, 6));
l.removeIf(x -> x % 2 == 0);
System.out.println(l); // affiche [1, 3, 5]
```

`List.replaceAll`

La méthode `replaceAll` de l'interface `List` prend un opérateur unaire en argument et remplace chaque élément de la liste par le résultat de cet opérateur appliqué à l'élément en question.
Par exemple, l'extrait de programme ci-dessous remplace tous les nombres par leur carré :

```
List<Integer> l = new ArrayList<>(
  Arrays.asList(1, 2, 3, 4, 5));
l.replaceAll(x -> x * x);
System.out.println(l); // [1, 4, 9, 16, 25]
```

`Map.computeIfAbsent`

La méthode `computeIfAbsent` de l'interface `Map` retourne la valeur associée à une clef, si elle existe ; sinon, elle utilise la fonction qu'on lui a passée pour déterminer la valeur associée à la clef, l'ajoute à la table, puis la retourne.

Par exemple, l'extrait de programme ci-dessous ajoute la valeur v à l'ensemble associé à la clef k, le créant et l'insérant dans la table au besoin :

```
Map<Integer, Set<Integer>> m =
  new HashMap<>();
m.computeIfAbsent(k, k1 -> new HashSet<>())
 .add(v);
```

`Map.computeIfAbsent`

L'intérêt de la méthode `computeIfAbsent` est visible lorsqu'on compare la version du code l'utilisant, présentée à l'instant :

```
m.computeIfAbsent(k, k1 -> new HashSet<>())
  .add(v);
```

avec l'équivalent écrit sans cette méthode :

```
if (!m.containsKey(k))
  m.put(k, new HashSet<>());
m.get(k).add(v);
```

Programmation par flots

Flots

Rappel : un **flot** (*stream*) est une séquence de valeurs auxquelles on accède l'une après l'autre, de la première à la dernière.

En Java, cette notion apparaît dans deux contextes :

1. les entrées/sorties, puisque les données lues ou écrites par le programme sont modélisées par des flots d'octets ou des caractères,
2. ce que l'on nomme parfois la **programmation par flots**, décrite ci-après.

Programmation par flots

La programmation par flots consiste à exprimer un calcul sur des données sous la forme d'un enchaînement d'opérations simples appliquées au flot de ces données.
Comme nous allons le voir, les fonctions anonymes permettent d'exprimer de tels enchaînements d'opérations de manière concise. Pour cette raison, plusieurs classes et méthodes ont été introduites en Java 8 – en même temps que les fonctions anonymes – pour faciliter la programmation par flots.

Conversion °F en °C

Admettons que l'on désire convertir un fichier contenant des températures en degrés Fahrenheit en un fichier contenant ces mêmes températures en degrés Celsius, en ignorant les lignes vides.

Par exemple, si le fichier d'entrée contient les trois lignes :

```
0
100

```

(la dernière étant vide), le fichier de sortie doit contenir les deux lignes :

```
-17.777
37.777
```

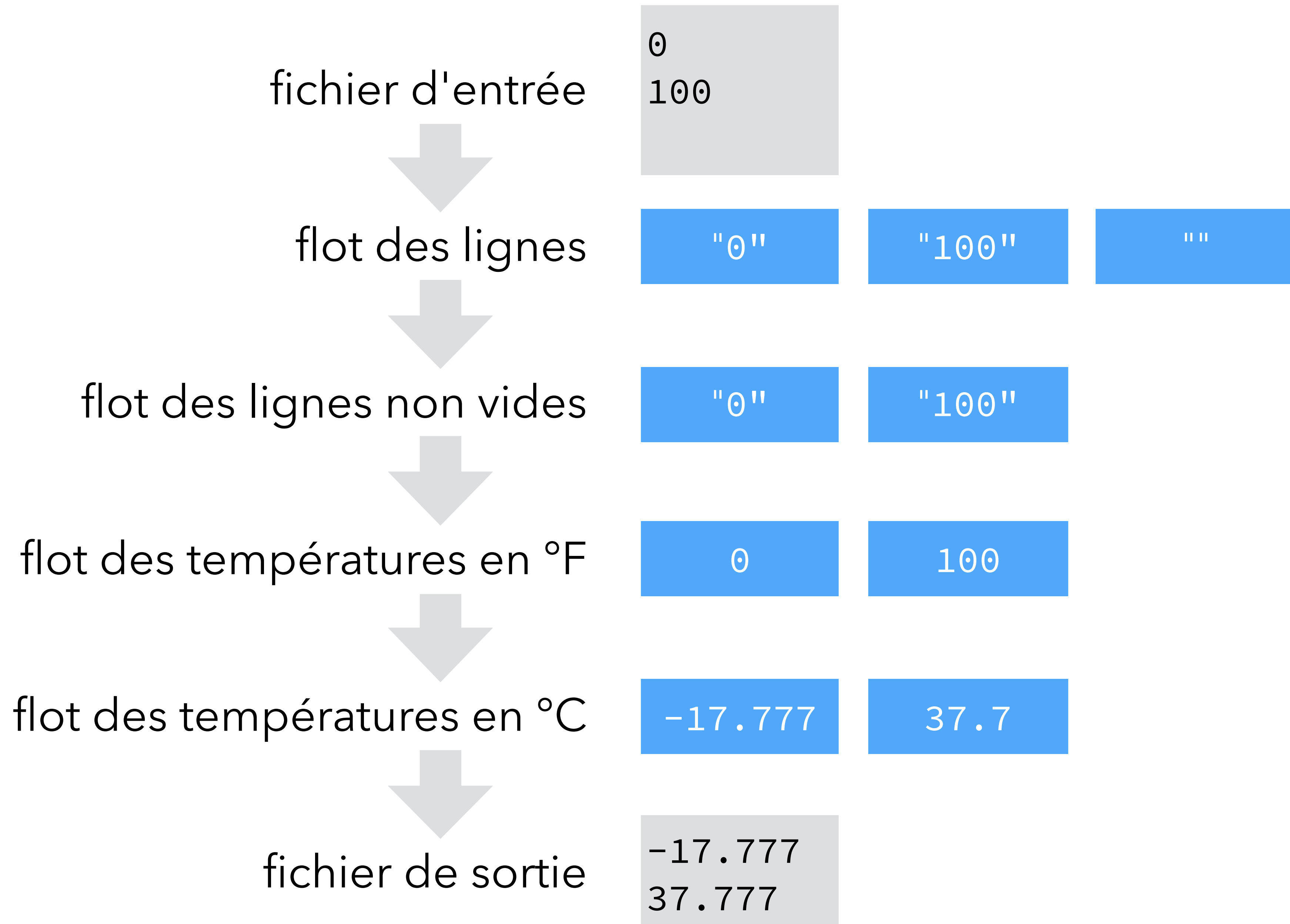
Conversion °F en °C

La conversion du fichier °F en °C peut s'exprimer sous la forme d'une transformation de flots ainsi :

1. obtenir le flot des lignes du fichier d'entrées,
2. filtrer ce flot pour ne garder que les lignes non vides,
3. convertir le flot de lignes – des chaînes de caractères – en un flot de températures en °F – des nombres réels,
4. convertir le flot des températures en °F en flot des températures en °C, au moyen de la formule de conversion [°C = (°F − 32) × 5/9],
5. écrire chaque valeur du flot dans le fichier de sortie, une par ligne.

Cette transformation de flot peut également se visualiser facilement.

Conversion °F en °C



Conversion °F en °C

En utilisant les classes et méthodes que nous allons examiner, cette conversion peut se traduire très simplement en un programme Java :

```
try(BufferedReader r = new BufferedReader(
      new FileReader("f.txt"));
    PrintWriter w = new PrintWriter(
      new FileWriter("c.txt"))) {
  r.lines()
   .filter(l -> !l.isEmpty())
   .map(l -> Double.parseDouble(l))
   .map(f -> (f - 32d) * (5d / 9d))
   .forEach(c -> { w.println(c); });
}
```

`java.util.stream`

Le paquetage `java.util.stream` – nouveauté de la version 8 de Java – définit plusieurs classes et interfaces permettant de faire de la programmation par flots.

Nous n'examinerons ici qu'un sous-ensemble des méthodes de la principale interface de ce paquetage, nommée `Stream` et représentant un flot de données.

De plus, nous verrons quelques méthodes servant de pont entre le monde des flots et celui des collections ou des lecteurs.

Interface `Stream`

L'interface `Stream` du paquetage `java.util.stream` représente un flot de valeurs d'un type donné. Elle est naturellement générique, son paramètre de type représentant le type des valeurs du flot :

```
public interface Stream<T> {
  // … méthodes
}
```

Par exemple, le flot des lignes d'un fichier aura le type `Stream<String>`, celui des températures – en °F ou °C – le type `Stream<Double>`, etc.

Types de méthodes

Les méthodes qui travaillent sur les flots – celles de l'interface `Stream` ou d'autres classes – appartiennent à l'une des trois catégories suivantes :

1. les **méthodes sources**, qui produisent un flot de valeurs à partir d'une source qui peut p.ex. être une collection, un fichier, etc.
2. les **méthodes intermédiaires**, qui transforment les valeurs du flot,
3. les **méthodes terminales**, qui consomment les valeurs du flot, p.ex. en les écrivant dans un fichier, en les réduisant à une valeur unique, etc.

Pipelines

Les trois types de méthodes mentionnés sont utilisés pour construire ce que l'on nomme parfois des **pipelines**. Un pipeline est formé de :

- *une* méthode source, qui produit un flot de valeurs,
- *zero ou plusieurs* méthodes intermédiaires, qui transforment les valeurs,
- *une* méthode terminale, qui consomme les valeurs.

La transformation de °F en °C présentée précédemment est un tel pipeline, dans lequel la méthode `lines` est la méthode sources, les méthodes `filter` et `map` sont les méthodes intermédiaires, et la méthode `forEach` la méthode terminale.

Stream.of

La méthode statique `of` de l'interface `Stream` est une méthode source qui prend un nombre quelconque d'arguments et les retourne sous forme d'un flot.

Par exemple, un flot composé des voyelles non accentuées de l'alphabet latin peut s'obtenir ainsi :

```
Stream<Character> vowels =
  Stream.of('a', 'e', 'i', 'o', 'u', 'y');
```

Stream.iterate

La méthode statique `iterate` de l'interface `Stream` est une méthode source qui produit un flot infini de valeurs, obtenu par application successive d'un opérateur unaire à une valeur initiale.
Par exemple, le flot (infini) des entiers strictement positifs peut s'écrire :

```
Stream<Integer> posInts =
  Stream.iterate(1, i -> i + 1); // 1, 2, …
```

`Stream.filter`

La méthode `filter` de l'interface `Stream` est une méthode intermédiaire qui produit un flot filtré ne contenant que les valeurs qui satisfont un prédicat qu'on lui passe.

Par exemple, pour ne garder que les entiers divisibles par 3 du flot des entiers positifs, on peut écrire :

```
Stream<Integer> multiplesOfThree =
  posInts.filter(x -> x % 3 == 0); // 3,6,…
```

Stream.map

La méthode map de l'interface Stream est une méthode intermédiaire qui applique une fonction à un argument aux éléments du flot et produit le flot des résultats.

Par exemple, le flot infini des carrés des entiers positifs peut s'obtenir ainsi :

```
Stream<Integer> posSqrs =
  posInts.map(i -> i * i); // 1, 4, 9, 16, …
```

tandis que celui des valeurs réciproques peut s'obtenir ainsi :

```
Stream<Double> posRecs =
  posInts.map(i -> 1.0 / i); // 1.0, 0.5, …
```

`Stream.limit`

La méthode `limit` de la classe `Stream` est une méthode intermédiaire qui limite le nombre maximum de valeurs que peut produire le flot auquel on l'applique.

Par exemple, le flot des 10 premiers entiers strictement positifs s'obtient ainsi :

```
Stream<Integer> posInts10 =
  posInts.limit(10); // 1, 2, …, 10
```

Stream.reduce

La méthode `reduce` de l'interface `Stream` est une méthode terminale qui réduit à une valeur unique les valeurs du flot, au moyen d'une valeur initiale et d'un opérateur binaire.

Par exemple, la somme des 10 premiers entiers strictement positifs peut s'obtenir en réduisant le flot de ces entiers au moyen de l'opérateur d'addition et de la valeur initiale 0 :

```
int posInts10Sum =
  posInts10.reduce(0, (x, y) -> x + y);
```

et le produit – qui vaut 10! = 3'628'800 – s'obtient de manière similaire :

```
int posInts10Prod =
  posInts10.reduce(1, (x, y) -> x * y);
```

`Stream.forEach`

La méthode `forEach` de l'interface `Stream` est une méthode terminale qui fournit, l'une après l'autre, les valeurs du flot à un consommateur.
Par exemple, pour afficher à l'écran les 10 premiers entiers strictement positifs – un par ligne – on peut écrire :

```
posInts10.forEach(
  x -> { System.out.println(x); });
```

`Collection.stream`

La méthode `stream` de l'interface `Collection` retourne un flot avec les éléments de la collection. Elle sert donc de pont entre le monde des collections et celui des flots.

Par exemple, pour calculer la somme des valeurs d'une table associative dont les valeurs sont des nombres à virgule flottante, on peut écrire :

```
Map<String, Double> m = …;
m.values()
  .stream()
  .reduce(0d, (x, y) -> x + y);
```

`BufferedReader.lines`

La méthode `lines` de `BufferedReader` retourne le flot des lignes du lecteur auquel on l'applique. Elle sert donc de pont entre le monde des lecteurs et celui des flots.

Par exemple, pour afficher toutes les lignes non vides du fichier `in.txt`, on peut écrire simplement :

```
try (BufferedReader r =
       new BufferedReader(
         new FileReader("in.txt"))) {
  r.lines()
   .filter(l -> !l.isEmpty())
   .forEach(l -> {System.out.println(l);});
}
```

Références de méthodes

Référence de méthode

Il arrive souvent que l'on veuille écrire une fonction anonyme qui se contente d'appeler une méthode en lui passant les arguments qu'elle a reçus.

Par exemple, le comparateur d'entiers ci-dessous appelle simplement la méthode statique `compare` de `Integer` pour comparer ses arguments :

```
Comparator<Integer> c =
  (i1, i2) -> Integer.compare(i1, i2);
```

Pour simplifier l'écriture de telles fonctions anonymes, Java offre la notion de **référence de méthode** (*method reference*). En l'utilisant, le comparateur ci-dessus peut se récrire simplement ainsi :

```
Comparator<Integer> c = Integer::compare;
```

Référence de méthode

Il existe plusieurs formes de références de méthodes, mais toutes utilisent une notation similaire, basée sur un double deux-points (`::`).
Nous n'examinerons ici que les trois formes de références de méthodes les plus fréquentes, à savoir :

- les références de méthodes statiques,
- les références de constructeurs,
- les références de méthodes non statiques, dont il existe deux variantes.

Référence statique

Une référence à une méthode statique s'obtient simplement en séparant le nom de la classe et celui de la méthode par un double deux-points.

Par exemple, comme on l'a vu, un comparateur ne faisant rien d'autre qu'utiliser la méthode statique `compare` de la classe `Integer` peut s'écrire ainsi :

```
Comparator<Integer> c = Integer::compare;
```

ce qui est équivalent à, mais plus concis que :

```
Comparator<Integer> c =
  (s1, s2) -> Integer.compare(s1, s2);
```

Référence constructeur

Il est également possible d'obtenir une référence de méthode sur un constructeur, en utilisant le mot-clef **new** en lieu et place du nom de méthode statique.

Par exemple, un fournisseur de nouvelles instances vides de `ArrayList` peut s'écrire :

```
Supplier<List> lists = ArrayList::new;
```

ce qui est équivalent à, mais plus concis que :

```
Supplier<List> lists =
   () -> new ArrayList();
```

Référence non statique 1

Aussi étrange que cela puisse paraître, une référence à une méthode *non statique* peut également s'obtenir en séparant le nom de la classe du nom de la méthode par un double deux-points.

Par exemple, un comparateur sur les chaînes ne faisant rien d'autre qu'utiliser la méthode (non statique !) `compareTo` des chaînes peut s'écrire :

```
Comparator<String> c = String::compareTo;
```

ce qui est équivalent à, mais plus concis que :

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

Notez que l'objet auquel on applique la méthode devient le premier argument de la fonction anonyme !

Référence (non) statique

Notez la différence cruciale entre une référence à une méthode statique et la première variante d'une référence à une méthode non statique que nous venons d'examiner :

- une référence à une méthode statique produit une fonction anonyme ayant *le même nombre d'arguments* que la méthode,
- une référence à une méthode non statique produit une fonction anonyme ayant *un argument de plus* que la méthode, cet argument supplémentaire étant le récepteur, c-à-d l'objet auquel on applique la méthode.

Référence (non) statique

Par exemple, la méthode statique `compare` de la classe `Integer` prend *deux* arguments. Dès lors, une référence vers cette méthode est une fonction à *deux* arguments :

```
BiFunction<Integer, Integer, Integer> c1 =
  Integer::compare;
```

La méthode non statique `compareTo` de la même classe `Integer` prend *un seul* argument. Mais comme il s'agit d'une méthode non statique, une référence vers cette méthode est aussi une fonction à *deux* arguments :

```
BiFunction<Integer, Integer, Integer> c2 =
  Integer::compareTo;
```

Référence non statique 2

Une seconde variante de référence à une méthode non statique permet de spécifier le récepteur. Avec cette variante, la fonction anonyme a le même nombre d'arguments que la méthode.

Par exemple, une fonction permettant d'obtenir le n^e caractère de l'alphabet (en partant de 0) peut s'écrire :

```
Function<Integer, Character> alphabetChar =
  "abcdefghijklmnopqrstuvwxyz"::charAt;
```

ce qui est équivalent à, mais plus concis que :

```
Function<Integer, Character> alphabetChar =
  i -> "abcdefghijklmnopqrstuvwxyz".charAt(i);
```

Transformation °F en °C

La transformation °F en °C peut se simplifier au moyen de références de méthodes :

```
try(BufferedReader r = new BufferedReader(
      new FileReader("f.txt"));
    PrintWriter w = new PrintWriter(
      new FileWriter("c.txt"))) {
  r.lines()
   .filter(l -> !l.isEmpty())
   .map(Double::parseDouble)
   .map(f -> (f - 32d) * (5d / 9d))
   .forEach(w::println);
}
```