Lambdas

Michel Schinz

2019-04-01

1 Introduction

Admettons que l'on désire écrire une classe `Sorter` dotée d'une méthode statique `sortDescending` qui trie par ordre décroissant les éléments d'une liste d'entiers qu'on lui passe en argument. Cette classe pourrait ressembler à ceci :

```
public final class Sorter {
  public static void sortDescending(List<Integer> a){
    // … code
  }
}
```

Pour ce faire, on peut s'aider de la méthode (statique) `sort` de la classe `Collections`, qui prend en arguments :

1. un tableau dynamique à trier,
2. un comparateur sachant comparer les éléments du tableau deux à deux,

et trie les éléments du tableau en ordre *croissant*, en fonction des informations fournies par le comparateur.

Avant de voir comment utiliser cette méthode pour arriver à nos fins, il importe de se souvenir ce qu'est, exactement, un comparateur.

1.1 Comparateurs

Pour mémoire, en Java, un comparateur n'est rien d'autre qu'une instance d'une classe qui implémente l'interface `java.util.Comparator`, définie ainsi :

```
public interface Comparator<T> {
  int compare(T v1, T v2);
}
```

Il s'agit d'une interface générique, dont le paramètre de type `T` représente le type des valeurs que le comparateur sait comparer. La méthode `compare` prend les deux valeurs à comparer, `v1` et `v2` ci-dessus, et retourne un entier exprimant leur relation, selon la convention suivante :

- si l'entier retourné est négatif, la première valeur est strictement plus petite que la seconde (`v1 < v2`),
- si l'entier retourné est nul, les deux valeurs sont égales (`v1 = v2`),
- si l'entier retourné est positif, la première valeur est strictement plus grande que la seconde (`v1 > v2`).

A noter que l'utilisation d'un entier pour exprimer ces trois possibilités est un accident historique. Il aurait été préférable d'utiliser une énumération comportant trois valeurs, mais les énumérations n'existaient pas encore en Java au moment où la notion de comparateur a été introduite.

1.2 Comparateur d'entier (inverse)

Pour trier un tableau d'entiers en ordre décroissant au moyen de la méthode `sort`, il suffit de définir un comparateur qui inverse l'ordre habituel des entiers. Par exemple, ce comparateur doit déclarer que 5 est *plus grand* que 10, et pas plus petit. Etant donné que la méthode `sort` utilise le comparateur pour trier les éléments en ordre croissant, ce comparateur inversé nous permet effectivement d'obtenir, de manière détournée, un tri par ordre décroissant.

1.2.1 Comparateur imbriqué statiquement

La classe du comparateur à utiliser, que nous appellerons `InvIntC` (*inverse integer comparator*) peut se définir comme une classe privée, imbriquée statiquement dans la classe `Sorter`. On obtient alors le résultat suivant :

```
import java.util.*;
import static java.util.Collections.sort;

public final class Sorter {
  public static void sortDescending(ArrayList<Integer> a){
    sort(a, new InvIntC());
  }

  private static class InvIntC implements
                                Comparator<Integer> {
    @Override
```

```
    public int compare(Integer i1, Integer i2) {
      if (i2 < i1)
        return -1;
      else if (i2.equals(i1))
        return 0;
      else // i2 > i1
        return 1;
    }
  }
}
```

A noter que pour tester si les deux entiers reçus sont égaux, il faut impérativement utiliser la méthode `equals` ici, et pas l'opérateur d'égalité `==`. La raison en est que les arguments de `compare` sont des *objets* (de type `Integer`) contenant des entiers, et pas des entiers de type `int`. Dès lors, l'opérateur `==` appliqué à de tels objets fait une comparaison par *référence*, ce qui n'est pas correct ici : deux instances différentes de `Integer` contenant le même entier doivent être considérées comme égales. Pour que ce soit bien le cas, il faut donc utiliser la méthode `equals`, qui est redéfinie dans ce but par la classe `Integer`.

Ce code peut être simplifié au moyen de la méthode (statique) `compare` de la classe `Integer`, qui compare deux entiers et retourne un entier exprimant leur relation, exactement comme la méthode `compare` de l'interface `Comparator`. En lui passant les entiers à comparer dans l'ordre inverse, on obtient bien le résultat escompté. La méthode `compare` du comparateur se simplifie alors ainsi :

```
public final class Sorter {
  public static void sortDescending(ArrayList<Integer> a){
    sort(a, new InvIntC());
  }

  private static class InvIntC implements
                                Comparator<Integer> {
    @Override
    public int compare(Integer i1, Integer i2) {
      return Integer.compare(i2, i1);
    }
  }
}
```

Même s'il fonctionne, le code ci-dessus est relativement lourd à écrire au vu de la simplicité de la tâche à réaliser.

1.2.2 Comparateur anonyme

Une première manière d'alléger le code ci-dessus consiste à utiliser ce que l'on nomme en Java une **classe intérieure anonyme** (*anonymous inner class*).

Comme ce nom le suggère, une classe intérieure anonyme n'est pas nommée, contrairement à `InvIntC` plus haut. Au lieu de cela, sa définition apparaît directement après l'énoncé `new` qui crée son instance. Dès lors, une telle classe n'est utile que s'il n'y a qu'un seul endroit dans tout le programme où l'on désire créer une de ses instances. Comme c'est le cas dans l'exemple plus haut, il est possible de récrire la classe `Sorter` ainsi :

```
public final class Sorter {
  public static void sortDescending(ArrayList<Integer> l){
    Collections.sort(l, new Comparator<Integer>() {
        @Override
        public int compare(Integer i1, Integer i2) {
          return Integer.compare(i2, i1);
        }
      });
  }
}
```

Il peut sembler étrange que le mot-clef `new` soit suivi du nom `Comparator`, puisque `Comparator` désigne une interface, et que les interfaces ne sont pas instanciables. La raison en est que, lorsqu'on utilise une classe intérieure anonyme, le nom qui suit le mot-clef `new` n'est pas le nom de la classe dont on désire créer une instance—puisqu'elle est anonyme—mais bien le nom de sa super-classe ou, comme ici, de l'interface qu'elle implémente. Ce nom est suivi des éventuels paramètres de son constructeur entre parenthèses, puis du corps de la classe entre accolades.

Cette nouvelle version de la classe `Sorter` est préférable à la précédente car elle est plus simple et la totalité du code lié au tri par ordre décroissant se trouve à l'intérieur de la méthode `sortDescending`. Néanmoins, la définition de la classe intérieure anonyme reste lourde.

1.2.3 Comparateur lambda

Heureusement, depuis la version 8 de Java, une syntaxe beaucoup plus légère permet d'écrire ce comparateur sans devoir définir explicitement une classe auxiliaire, nommée ou non.

En utilisant cette notation, on peut définir la méthode `sortDescending` simplement ainsi :

```
public final class Sorter {
  public static void sortDescending(ArrayList<Integer> a){
    sort(a, (i1, i2) -> Integer.compare(i2, i1));
```

```
  }
}
```

En comparant cette version à la précédente, on constate que le comparateur passé à `sort` est obtenu simplement en écrivant le corps de sa méthode `compare` (en omettant le `return`) précédé d'une flèche (`->`) et du nom de ses deux arguments entre parenthèses (ici `i1` et `i2`). Cette construction est connue sous le nom de **lambda**.[1]

2 Lambdas

Avant de pouvoir décrire les lambdas en détail, il faut examiner un type particulier d'interface, celles dites *fonctionnelles*.

2.1 Interface fonctionnelle

Pour mémoire, depuis Java 8, les interfaces peuvent 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.

Une interface est appelée **interface fonctionnelle** (*functional interface*) si elle possède exactement *une* méthode abstraite. Elle peut néanmoins posséder un nombre quelconque de méthodes concrètes, statiques ou par défaut.

Par exemple, l'interface `Comparator` est une interface fonctionnelle, car elle ne possède qu'une seule méthode abstraite, à savoir `compare`. Le fait qu'elle possède aussi plusieurs méthodes concrètes (à la fois statiques et par défaut) n'y change rien.

Un autre exemple d'interface fonctionnelle est l'interface `RealFunction` ci-dessous, qui pourrait représenter une fonction mathématique des réels vers les réels :

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

Comme cet exemple l'illustre, les interfaces fonctionnelles peuvent être annotées au moyen de l'annotation `@FunctionalInterface`. Cette annotation est optionnelle, mais utile car elle garantit qu'une erreur est produite si l'interface à laquelle on l'attache n'est pas fonctionnelle, p.ex. car elle comporte plusieurs méthodes abstraites.

1. Le terme vient du lambda-calcul (ou λ-calcul), un système formel inventé dans les années 30 par le mathématicien américain Alonzo Church afin de décrire les fonctions et leur application. Le lambda-calcul a eu un impact important sur les langages de programmation, en particulier ceux dits *fonctionnels*.

2.2 Lambda

En Java, une **lambda**, aussi appelée **fonction anonyme** (*anonymous function*) ou parfois **fermeture** (*closure*), est une expression créant une instance d'une classe anonyme qui implémente une interface fonctionnelle.

La lambda 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 `->` :

arguments `->` *corps*

Une lambda ne peut apparaître que dans un contexte qui attend une valeur dont le type est une interface fonctionnelle. La raison de cette restriction est claire : étant donné que la lambda ne spécifie pas le nom de la méthode qu'elle implémente, il faut que le compilateur puisse la déterminer sans ambiguïté. Or cela n'est possible que si la lambda apparaît dans un contexte dans lequel la valeur attendue a pour type une interface fonctionnelle, c-à-d dotée d'une et une seule méthode abstraite.

Par exemple, l'expression suivante est valide :

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

car `Comparator` est une interface fonctionnelle, et il est donc clair que la lambda correspond à la méthode `compare` de cette interface. Par contre, l'expression suivante n'est pas valide :

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

car `Object` n'est pas une interface fonctionnelle et le compilateur ne peut donc savoir à quelle méthode correspond la lambda.

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

Dans sa forme la plus générale, le corps d'une lambda est constitué d'un bloc entouré d'accolades. Comme toujours, si la lambda 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 (via la méthode `compareTo` des chaînes) :

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

Si le corps de la lambda 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());
}
```

2.3 Accès à l'environnement

Une des raisons pour lesquelles les lambdas sont puissantes est qu'elles ont accès à toutes les entités visibles à l'endroit de leur définition : paramètres et variables locales de la méthode dans laquelle elles sont éventuellement définies, attributs et méthodes de la classe englobante, etc[2].

Pour illustrer cette possibilité, admettons que l'on désire généraliser la méthode de tri en lui ajoutant un argument supplémentaire spécifiant si le tri doit se faire par ordre croissant ou décroissant. On peut écrire simplement :

```
public final class Sorter {
  enum Order { ASCENDING, DESCENDING };

  public static void sort(ArrayList<Integer> l, Order o) {
    Collections.sort(l, (i1, i2) ->
                     o == Order.ASCENDING
                     ? Integer.compare(i1, i2)
                     : Integer.compare(i2, i1));
  }
}
```

2. Il y a néanmoins une restriction concernant les variables locales, qui doivent être ce que Java appelle *effectively final*. En gros, cela signifie qu'elles doivent être immuables. Pour plus de détails, consulter la section 4.12.4 de *The Java Language Specification*.

Comme on le constate, le comparateur créé par la lambda utilise l'argument `o` de la méthode dans laquelle il est défini. Attention, cela implique que le test pour déterminer l'ordre de tri est effectué à *chaque* comparaison de deux éléments du tableau à trier, ce qui est peu efficace. Il serait donc préférable de récrire l'exemple ci-dessus en sortant le test de la lambda, ce qui est laissé en exercice.

Pour terminer, il faut noter que les classes intérieures anonymes ont également cette capacité d'accéder à toutes les entités visibles à l'endroit de leur définition.

3 Interfaces fonctionnelles de l'API Java

On l'a vu, les lambdas 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-dessous. Ces interfaces fonctionnelles représentent :

- les fonctions unaires (à un argument) et binaires (à deux arguments),
- les opérateurs (fonctions dont le type de retour est identique au type de l'argument) unaires et binaires,
- les prédicats (fonctions dont le type de retour est booléen) unaires et binaires,
- les producteurs et consommateurs (fonctions respectivement sans arguments et sans valeur de retour).

3.1 Fonctions unaires et binaires

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
```

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
```

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”, 4); // => o
```

3.2 Opérateurs unaires et binaires

L'interface `UnaryOperator` représente un opérateur unaire, c-à-d 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
```

L'interface `BinaryOperator` représente un opérateur binaire, c-à-d 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
```

3.3 Prédicats unaires et binaires

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

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
```

Les interfaces `Predicate` et `BiPredicate` offrent des méthodes par défaut permettant d'obtenir de nouveaux prédicats à partir de prédicats existants. Ces méthodes sont :

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

On peut les utiliser ainsi :

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

3.4 Producteurs et consommateurs

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

```
public interface Supplier<T> {
  public 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
```

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
```

4 Références de méthodes

Il arrive souvent que l'on veuille écrire une lambda qui se contente d'appeler une méthode en lui passant les arguments qu'elle a reçu. 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;
```

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 :

1. les références de méthodes statiques,
2. les références de constructeurs,
3. les références de méthodes non statiques, dont il existe deux variantes.

4.1 Référence statique

Une référence à une méthode statique s'obtient simplement en séparant le nom de la classe de 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);
```

4.2 Référence de 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<Integer>` peut s'écrire :

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

ce qui est équivalent à, mais plus concis que :

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

4.3 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 lambda ! Il y a donc une différence cruciale entre une référence à une méthode statique et la première variante d'une référence à une méthode non statique ci-dessus :

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

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;
```

4.4 Référence non statique (2)

Une seconde variante de référence à une méthode non statique permet de spécifier le récepteur. Avec cette variante, la lambda 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);
```

5 Références

- la documentation de l'API Java, en particulier :
  - l'interface `java.util.Comparator`,
  - le paquetage `java.util.function`, en particulier les interfaces fonctionnelles suivantes :
    - `Function` et `BiFunction`,
    - `Predicate` et `BiPredicate`,
    - `Supplier` et `Consumer`,
  - l'annotation `FunctionalInterface`.
- *The Java® Language Specification*, de James Gosling et coauteurs, en particulier :
  - §15.9, *Class Instance Creation Expressions*,
  - §9.8, *Functional Interfaces*,
  - §15.13, *Method Reference Expressions*,
  - §15.27, *Lambda Expressions*,
  - §4.12.4, *`final` Variables*.