Lambdas

CS-108

Michel Schinz

2024-03-19

1 Introduction

En programmation, il est fréquemment utile de pouvoir passer du code en paramètre à une fonction ou méthode.

Par exemple, pour qu'une méthode de tri de liste soit aussi générale que possible, il faut qu'elle prenne un argument lui permettant de déterminer l'ordre dans lequel trier les éléments de la liste. Or la représentation la plus naturelle pour cet ordre de tri est un morceau de code qui, étant donné deux éléments de la liste à trier, détermine si le premier est plus petit que, égal à, ou plus grand que le second.

Il est dès lors utile qu'un langage de programmation offre une syntaxe concise pour représenter de tels morceaux de code. C'est le cas de la plupart des langages de programmation modernes, dont Java, qui offre la notion de **lambda** dans ce but.

Pour illustrer l'utilité de ce concept, admettons que nous devions écrire une classe `Sorter` dotée d'une méthode statique `sortDescending`, triant par ordre *décroissant* les éléments d'une liste d'entiers qu'on lui passe en argument. Le corps de la méthode `sortDescending` consiste en un simple appel à la méthode `sort` de `List`, qui trie la liste à laquelle on l'applique étant donné un **comparateur** sachant comparer deux éléments. Ce comparateur n'est rien d'autre que le morceau de code représentant l'ordre de tri mentionné ci-dessus.

```
public final class Sorter {
  public static void sortDescending(List<Integer> l) {
    l.sort(/* ... comparateur (à faire) */);
  }
}
```

Reste à trouver quel comparateur passer à `sort` pour obtenir une liste triée par ordre *décroissant*, sachant qu'elle trie la liste par ordre *croissant*. Avant cela, il convient toutefois d'examiner la notion de comparateur plus en détail.

2 Comparateurs

En Java, un comparateur n'est rien d'autre qu'une instance d'une classe qui implémente l'interface `java.util.Comparator`, dont la définition (légèrement simplifiée) est :

```
public interface Comparator<T> {
  public abstract 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`).

À 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.

3 Comparateur d'entier inverse

Pour compléter le corps de la méthode `sortDescending`, il nous 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.

Voyons comment définir ce comparateur d'entiers inversé.

3.1 Classe imbriquée statiquement

Une première manière de définir le comparateur consiste à écrire une classe le représentant, nommée p.ex. `IIC` (pour *inverse integer comparator*). Cette classe peut naturellement être imbriquée statiquement dans la classe `Sorter` et rendue privée, car elle est avant tout destinée à être utilisée par `sortDescending`. On obtient alors[1] :

[1] Attention : pour tester si les deux entiers reçus sont égaux, il faut impérativement utiliser la méthode `equals`, et pas l'opérateur d'égalité ==. La raison en est que les arguments de `compare` sont des *objets* (de

```
public final class Sorter {
  public static void sortDescending(List<Integer> l) {
    l.sort(new IIC());
  }

  private static class IIC 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;
    }
  }
}
```

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(List<Integer> l) {
    l.sort(new IIC());
  }

  private static class IIC 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.

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`.

3.2 Classe 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 `IIC` 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(List<Integer> l) {
    l.sort(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. Toutefois, 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.

3.3 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 syntaxe, on peut définir la méthode `sortDescending` simplement ainsi :

```
public final class Sorter {
  public static void sortDescending(List<Integer> l) {
    l.sort((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` — sans `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**.[2]

4 Lambdas

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

4.1 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`. Il se trouve qu'elle possède également d'autre méthodes, aussi bien statiques que par défaut, qui n'ont pas été présentées à la §2 dans un soucis de simplicité. Toutefois, comme il s'agit de méthodes concrètes et non abstraites, `Comparator` reste une interface fonctionnelle.

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.

[2] 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*.

4.2 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 la méthode abstraite de l'interface fonctionnelle. Ceux-ci sont séparés par une flèche symbolisée par les deux caractères → :

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 Java puisse le 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 unique 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 on ne peut donc savoir à quelle méthode correspond la lambda.

4.2.1 Arguments abrégés

Dans leur forme générale, les arguments 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 Java,
2. lorsque la fonction ne prend qu'un seul argument, les parenthèses peuvent être omises.

4.2.2 Corps abrégé

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` de `String`) :

```
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());
}
```

4.2.3 Accès à l'environnement

Une des raisons pour lesquelles les lambdas sont puissantes est qu'elles ont accès à ce que l'on appelle leur **environnement**, c-à-d 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[3].

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(List<Integer> l, Order o) {
    l.sort((i1, i2) ->
```

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

```
            o == Order.ASCENDING
            ? Integer.compare(i1, i2)
            : Integer.compare(i2, i1));
  }
}
```

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 de la liste 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.

5 Utilisation des lambdas

Pour que les lambdas soient utilisables, il faut bien entendu qu'il existe des méthodes prenant en argument des valeurs dont le type est une interface fonctionnelle. La bibliothèque Java offre de nombreuses méthodes de ce type, qui utilisent généralement des interfaces fonctionnelles provenant du paquetage `java.util.function`.

Les sections ci-dessous présentent quelques exemples d'utilisation de certaines de ces méthodes, afin de donner un aperçu de ce que les lambdas permettent.

5.1 Parcours des collections « itérables »

Pour mémoire, les collections Java peuvent en général être parcourues au moyen d'un itérateur, que l'on obtient grâce à la méthode `iterator`. Cette méthode est définie dans l'interface `Iterable`, destinée à être implémentée par toute classe « itérable », c-à-d dont le contenu peut être parcouru de cette manière.

En plus de la méthode abstraite `iterator`, l'interface `Iterable` offre la méthode par défaut `forEach`, qui permet de parcourir le contenu de la collection sans devoir créer explicitement un itérateur. Attention, cette méthode ne doit pas être confondue avec la *boucle for-each* déjà examinée !

La méthode `forEach` est définie ainsi dans `Iterable` :

```
public interface Iterable<T> {
  default void forEach(Consumer<T> action) { /* ... */ }
}
```

Son argument est ce que la bibliothèque Java nomme un **consommateur** (*consumer*), décrit par l'interface fonctionnelle `Consumer` du paquetage `java.util.function`, qui ressemble à ceci :

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

La méthode `forEach` de `Iterable` appelle simplement la méthode `accept` du consommateur pour chacun des éléments de la collection, dans l'ordre d'itération. Elle pourrait donc se définir très facilement, par exemple au moyen d'une boucle *for-each* :

```
public interface Iterable<T> {
  default void forEach(Consumer<T> action) {
    for (T element: this)
      action.accept(element);
  }
}
```

Étant donné que l'interface `Consumer` est une interface fonctionnelle, l'argument passé à la méthode `forEach` peut être une lambda. Ainsi, pour afficher à l'écran tous les éléments d'une liste de chaînes, on peut écrire :

```
List<String> list = /* ... */;
list.forEach(e -> System.out.println(e));
```

5.2 Parcours des tables associatives

Comme nous l'avons vu dans la leçon sur les collections, les tables associatives ne sont malheureusement pas itérables : elles n'implémentent pas l'interface `Iterable`, et leur contenu ne peut donc pas être parcouru au moyen d'un itérateur ou de la boucle *for-each*.

La seule solution que nous connaissons à ce stade pour parcourir une table associative consiste donc à parcourir l'ensemble de ses paires clef/valeur, sur lequel on peut obtenir une vue au moyen de la méthode `entrySet`. Par exemple, pour afficher toutes les paires clef/valeur d'une table associative `map`, on peut écrire :

```
Map<String, Integer> map = /* ... */;
for (Map.Entry<String, Integer> e : map.entrySet()) {
  String k = e.getKey();
  int v = e.getValue();
  System.out.println(k + " : " + v);
}
```

Le fait de devoir manipuler les paires clef/valeur rend ce code relativement lourd. Heureusement, l'interface `Map` offre une méthode `forEach` similaire à celle offerte par les listes mais prenant en argument une lambda à *deux* arguments : le premier est la clef, le second la valeur qui lui est associée. Au moyen de cette méthode, le code ci-dessus se récrit beaucoup plus agréablement ainsi :

```
Map<String, Integer> map = /* ... */;
map.forEach((k, v) -> System.out.println(k + "␣:␣" + v));
```

Le type de l'argument de `forEach` est `BiConsumer`, une interface fonctionnelle qui n'est rien d'autre qu'une version à deux arguments (d'où le préfixe *bi*) d'un consommateur :

```
public interface BiConsumer<T, U> {
  void accept(T t, U u);
}
```

5.3 Ajout dans une table associative

En plus de la méthode `forEach`, l'interface `Map` offre plusieurs méthodes destinées à être utilisées avec des lambdas et qui simplifient souvent la manipulation des tables associatives.

Par exemple, pour compter le nombre d'occurrences de chaque mot dans une liste `words`, on peut écrire la boucle suivante :

```
List<String> words = List.of("to","be","or","not","to","be");
Map<String, Integer> count = new HashMap<>();
for (String w: words) {
  if (! count.containsKey(w))
    count.put(w, 1);
  else
    count.put(w, count.get(w) + 1);
}
```

Le corps de la boucle distingue deux cas : si le mot n'a pas encore été rencontré, et ne fait donc pas partie de la table, il y est ajouté avec un nombre d'occurrences de 1 ; sinon, le nombre d'occurrences actuel est extrait et incrémenté.

Il est très fréquent d'écrire ce genre de code lorsqu'on utilise une table associative, et pour cette raison l'interface `Map` offre la méthode `merge` permettant de le simplifier. En l'utilisant, la boucle ci-dessus peut se récrire ainsi :

```
Map<String, Integer> count = new HashMap<>();
for (String w: words)
  count.merge(w, 1, (c, v) -> c + 1);
```

5.4 Construction de comparateur

Il est fréquent de devoir trier des listes en utilisant un critère de tri différent de celui par défaut. Par exemple, on peut vouloir trier une liste de chaînes par longueur plutôt que par ordre alphabétique.

Pour ce faire, on peut utiliser la méthode `sort` de `List` en lui passant en argument un comparateur comparant les chaînes par longueur. Ce comparateur peut bien entendu s'écrire au moyen d'une lambda :

```
List<String> l = Arrays.asList("bas","bras","as","a","sabre");
l.sort((s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Il est toutefois possible de simplifier encore la définition de ce comparateur, en utilisant la méthode statique `comparing` de `Comparator`. Cette méthode prend en argument une lambda qui reçoit en argument un des éléments à comparer et retourne la valeur à utiliser pour la comparaison. Ainsi, l'appel à `sort` ci-dessus peut se récrire :

```
l.sort(Comparator.comparing(s -> s.length())
```

En utilisant une référence de méthode, concept décrit à la section suivante, ce code peut être rendu encore plus concis :

```
l.sort(Comparator.comparing(String::length))
```

6 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 lambdas, 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 courantes, 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.

6.1 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);
```

6.2 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<>();
```

6.3 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;
```

6.4 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);
```

7 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*.