Java 8

FIP ING 39

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2016-2021

But du cours

- montrer la grande nouveauté de java 8 : les lambda-expressions ;
- compléter par quelques autres nouveautés qui améliorent la vie.

Méthodes par défaut dans les interfaces

- une interface ne contient que des en-têtes de méthodes ;
- souvent, il y a cependant une certaine redondance entre celles-ci :

```
1  /* interface pour les trucs qui ressemblent
2     à des tableaux */
3  public interface Indexable<T> {
4      int size();
5
6      T get(int i);
7
8      void set(int i, T val);
9
10     void inverser();
11 }
```

Ici, il serait possible d'écrire la méthode inverser sans faire référence à une implémentation.

Solution usuelle en java 7

On crée une classe abstraite pour ça :

```
public abstract class AbstractIndexable<T>
        implements Indexable<T>{
    public void inverser() {
        int i= 0; int j= size() -1 ;
        while (i < j) {
            T tmp= get(i); set(i, get(j));
            set(j, tmp);
            i++; j--;
        }
    }
}
```

les classes concrètes peuvent alors, au choix

- implémenter toutes les méthodes d'`Indexable` ;
- étendre `AbstractIndexable` et profiter de l'implémentation d'`inverser`.

Méthodes par défaut

Pour diminuer cette nécessité d'utilisation d'une classe abstraite, java 8 introduit la notion de « méthode par défaut » dans une interface.

- une interface peut contenir le corps d'une méthode (précédé de `default`) ;
- elle ne peut toujours pas contenir de variable d'instance (sinon, ce serait une classe) ;
- quand on implémente une telle interface, on n'est pas obligé de réécrire les méthodes par défaut.
- une classe peut implémenter *plusieurs interfaces* avec des méthodes par défaut
- s'il y a conflit (deux méthodes par défaut de même signature sont héritées), elle doit les redéfinir ou c'est une erreur.

Méthodes par défaut

```
public interface Indexable<T> {
    int size();

    T get(int i);

    void set(int i, T val);

    default void inverser() {
        int i= 0; int j= size() -1 ;
        while (i < j) {
            T tmp= get(i);
            set(i, get(j));
            set(j, tmp);
            i++;
            j--;
        }
    }
}
```

Les fonctions comme données

```
class FiltreJava implements FilenameFilter {
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(".java");
    }
}

public class ListerJava {
 public static void main(String[] args) {
   File d= new File("src/main/java/demo/introfonctions");
    for (File f: d.listFiles(new FiltreJava())) {
      System.out.println(f.getAbsolutePath());
    }
 }
}
```

- `FiltreJava` sert uniquement à « enrober » la fonction accept qui fait `return name.endsWith(".java")`
- long et souvent peu lisible.

Autre exemple

```
class MaTache implements ActionListener {
 @Override
 public void actionPerformed(ActionEvent e) {
   System.out.println("On travaille !!");
 }
}
public class DemoTimer {
 public static void main(String[] args) {
   Timer timer= new Timer(1000, new MaTache());
   timer.start();
 }
}
```

Les fonctions comme données

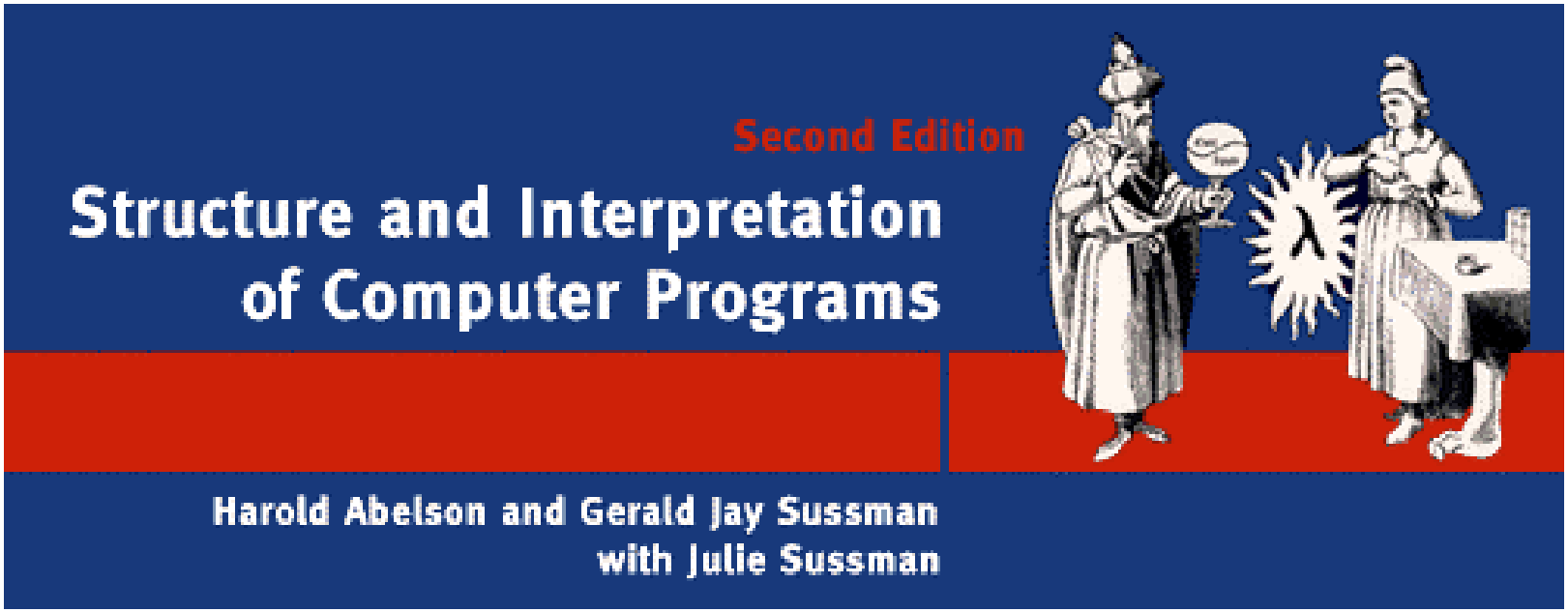
```
class FiltreJava implements FilenameFilter {
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(".java");
    }
}

public class ListerJava {
 public static void main(String[] args) {
   File d= new File("src/main/java/demo/introfonctions");
    for (File f: d.listFiles(new FiltreJava())) {
      System.out.println(f.getAbsolutePath());
    }
 }
}
```

Le même en java 8 ! ! !

```
public class ListerJava2 {
 public static void main(String[] args) {
   File dir= new File("src/main/java/demo/introfonctions");
   for (File f:
    dir.listFiles((d,n) -> n.endsWith(".java"))) {
    System.out.println(f.getAbsolutePath());
   }
 }
}
```

- `(d,n) -> n.endsWith(".java")` définit à la volée la fonction `accept`;
- c'est une lambda expression.

Une page de pub : les langages fonctionnels



- lisp, scheme, scala, clojure, ocaml, haskell, ruby ? python ?
- depuis 1958 ;
- gros regain d'intérêt à cause du parallélisme.

Quel est le truc ?

- sucre syntaxique : on a toujours besoin d'une classe et d'un objet de cette classe ;
- mais le compilateur les crée à la volée à votre place ;
- création d'une *classe interne anonyme* (on en reparle plus tard, promis)
- pas forcément de déclaration : *inférence de types* ;
- une lambda peut remplacer toute valeur dont le type est *une interface fonctionnelle.*

Interface fonctionnelle

- l'idée : une interface qui définit une et une seule fonction.

```
1 public interface FilenameFilter {
2  boolean accept(File dir, String name);
3 }
```

- On peut écrire :

```
1  FilenameFilter filter=
2    (File d, String n) -> n.endsWith(".c");
```

- inférence de type : si on a

```
1  FilenameFilter filter=
2    (d, n) -> n.endsWith(".c");
```

  On sait que d est de type `File` et `n` de type `String`.

- **définition :** interface qui a exactement une méthode abstraite ;
- elle peut avoir une ou plusieurs méthodes par défaut.

Exemple

```
@FunctionalInterface
public interface FonctionReelle {
 double eval(double x);

 default double derivee(double x, double delta) {
   return (eval(x+delta) - eval(x-delta))/ (2*delta);
 }
}
```

Lambda et collections : le filtre

idée : ne conserver que les éléments d'une collection qui vérifient un filtre booléen.

Exemple : ne conserver que les messages de Toto :

```
List<Message> res= new ArrayList<>();
for (Message m: lesMessages) {
  if (m.getAuteur().equals("toto"))
    res.add(m);
}
```

type de traitement très courant... du coup :

```
List<Message> r= lesMessages.stream()
        .filter(m -> m.getAuteur().equals("toto"))
        .collect(Collectors.toList());
```

Lambda et collections : les streams

- vue d'une collection comme une suite d'éléments auquels on applique des fonctions ;
- créés par la méthode `stream()`
- parfois parallélisables : méthode `parallelStream()` de `Collection` ;

Petite démo parallèle

```
public class DemoTri {
 private static void time(Runnable r) {
   long start = System.nanoTime();
   r.run();
   long end = System.nanoTime();
   System.out.println("Temps mis " + (end - start) / 1.0e9)
 }

 public static void main(String[] args) {
   ArrayList<Double> l = new ArrayList<>();
   for (int i = 0; i < 1_000_000; i++) {
    l.add(Math.random());
   }
   time(() -> l.stream().sorted().toArray());
   time(() -> l.parallelStream().sorted().toArray());
   time(() -> l.stream().sorted().toArray());
   time(() -> l.parallelStream().sorted().toArray());
 }
}
```

collect

permet de « revenir » des streams aux collections (ou à d'autres types).

- Rassemble les résultats d'un traitement à l'aide d'un `Collector`
- En pratique, on utilise les collectors définis dans la classe `Collectors` :
- toList()
- toSet()
- counting
- groupingBy
- Collectors.joining : concaténation
- ...

map

applique une fonction à tous les éléments du stream, et renvoie un autre stream :

```
List<String> titres=
  messages.stream()
    .map(m -> m.getTitle())
    .collect(Collectors.toList());
```

Combinaison des opérations

```
Set<String> titresByToto=
  messages.stream()
    .filter(m-> m.getAuthors().equals("toto"))
    .map(m -> m.getTitle())
    .collect(Collectors.toSet());
```

reduce

- alternative à collect ;
- permet de combiner les éléments du stream en un seul à l'aide d'un opérateur ;
- soit T le type des éléments du stream, et une opération f : $T \times T \rightarrow T$;
- alors reduce applique cette opération à tous les éléments du stream.

Exemple

```
1 List<Integer> l= Arrays.asList(3,10, 7,15,20);
2 System.out.println(l.stream().reduce(1, (a,b)-> a*b));
```

Pointeurs de méthodes

Il arrive très souvent de vouloir définir une lambda qui soit :

- ou une méthode statique d'une classe :

```
1  ArrayList<String> l= ...;
2  List<String> l1= l.stream()
3      .map(s -> StringUtils.inverser(s))
4      .collect(Collectors.toList());
```

- ou une méthode des objets contenus dans le Stream :

```
1  ArrayList<String> l= ...;
2  List<String> l1= l.stream()
3      .map(s -> s.toUppercase())
4      .collect(Collectors.toList());
```

- pour ces deux cas, on dispose d'une notation allégée :

Pointeurs de méthodes

Il arrive très souvent de vouloir définir une lambda qui soit :

- ou une méthode statique d'une classe :

```
1  ArrayList<String> l= ...;
2  List<String> l1= l.stream()
3      .map(StringUtils::inverser)
4      .collect(Collectors.toList());
```

- ou une méthode des *objets* contenus dans le Stream :

```
1  ArrayList<String> l= ...;
2  List<String> l1= l.stream()
3      .map(String::toUppercase)
4      .collect(Collectors.toList());
```

Tony Hoare's « Null References : The Billion Dollar Mistake »

À propos de `null` :

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Optional

- une méthode retourne une valeur ou reçoit un argument ;
- dans certains cas, on veut dire que cette valeur peut être absente ;
- solution usuelle : renvoyer (ou passer) `null`.

Problème

Ça n'est pas explicite. Du coup, le programmeur se méfie de tout argument ou toute valeur retournée de type objet.

- Optional permet de le rendre explicite ;
- ça ne résoud que partiellement le problème (il est au niveau du langage lui-même) ;

Optional

Construction

```
Optional<Integer> a= Optional.of(3);
Optional<Integer> b= Optional.empty();
```

méthodes

- isPresent() : renvoie vrai si on a une valeur ;
- filter/map : comme pour un stream ;
- orElse(autreValeur) : la valeur (si elle est là), sinon une valeur par défaut ;
- ifPresent(Consumer consumer) : prend comme argument une fonction qui fera quelque chose avec notre élément.
- get() : renvoie la valeur (ou lève une exception).

Interfaces générales

Pour faciliter la vie, java définit un certain nombre d'interfaces très générales dans le package `java.util.function` :

- Function<T,R> : une fonction qui prend un argument T et retourne R ;
- DoubleUnaryOperator : prend un double et retourne un double ;
- Consumer<R> : prend comme argument un objet de type R et retourne void.
- ...

λ et Comparator

En java 8, l'interface `Comparator` est dotée de méthodes statiques utiles pour créer des comparateurs à la volée :

```
List<Personne> l= ...;
Collections.sort(l,
  Comparator
    .comparing(Personne::getNom)
    .thenComparingInt(Personne::getAge));
```