Expressions lambda et streams

Université Française d'Égypte
Richard Grin
Version O 1.6 – 23/11/19

Plan du support

Introduction

- Expressions lambda et streams permettent d'introduire de la programmation fonctionnelle dans le langage Java
- Simplifient et rendent plus lisibles des traitements

Présentation des expressions lambda (EL)

- Définissent des fonctions qui peuvent être utilisées dans d'autres parties du programme

Trier une liste sans EL

```
List<Ville> villes = new ArrayList<>();
villes.add(new Ville("Nice", 300_000));
// Ajouter d'autres villes
...
// Trier la liste par le nombre d'habitants
...
```

Comment fait-on ? (Révision)

Le tri sans EL

L'essentiel du tri : la valeur qui sert à comparer 2 villes

```
villes.sort(
  new Comparator<Ville>() {
    public int compare(Ville v1, Ville v2) {
      return v1.getNbH() – v2.getNbH();
    }
  });
```

`sort(Comparator<? super E> c)`

L'essentiel du tri

```
villes.sort(
  new Comparator<Ville>() {
    public int compare(Ville v1, Ville v2) {
      return v1.getNbH() – v2.getNbH();
    }
  });
```

`sort(Comparator<? super E> c)`

Avec une expression lambda

```
villes.sort(
  (Ville v1, Ville v2) -> v1.getNbH() – v2.getNbH());
```


Avec une expression lambda

```
villes.sort(
  (v1, v2) -> v1.getNbH() – v2.getNbH());
```

Le type de `v1` et `v2` peut être déduit du type de `villes`

Utilité des expressions lambda

- Plus simple et plus lisible
  - on ne garde que l'essentiel : on passe une fonction dont on donne les paramètres et le code
  - on enlève le reste : le nom de l'interface, le nom de la seule méthode de l'interface, les types des paramètres si le contexte permet de les deviner

Syntaxe des expressions lambda

Paramètres

- Paramètres entre parenthèses :
  `(Ville v1, Ville v2) ->` …
- Si le contexte permet d'inférer les types :
  `(v1, v2) ->` …
- Parenthèses optionnelles si un seul paramètre avec un type inféré
  `v1 ->` …
- Parenthèses si pas de paramètres
  `() ->` …

Corps de l'expression

- Une seule expression,
  ou bloc délimité par des accolades
- S'il est composé d'une seule expression, la valeur de l'expression est retournée (si l'expression retourne quelque chose)
- `return` peut aussi être utilisé pour retourner une valeur (à l'intérieur d'un bloc)
- Peut utiliser des variables (déclarées avec un type ou avec `var`)

Exemples

- `(Ville v1, Ville v2) -> v1.getNbH() – v2.getNbH()`
- `(v1, v2) -> v1.getNbH() – v2.getNbH()`
- `ville -> ville.getNom()`
- ```
  container -> {
      int s = container.size(); // ou var s = ...
      container.clear();
      return s;
  }
  ```


Où utiliser une expression lambda

- Utiliser pour définir une action, un comportement et le transmettre à un autre endroit du code
- Assigner à une variable :
  ```
  Comparator<Ville> comparator =
    (v1, v2) -> v1.getNbH() - v2.getNbH();
  villes.sort(comparator.reversed());
  ```
- Passer en paramètre :
  ```
  villes.sort(
    (v1, v2) -> v1.getNbH() – v2.getNbH());
  ```


Contexte d'exécution

- Une expression lambda peut utiliser les variables d'état (d'instance ou `static`) de la classe dans laquelle elle est définie
- Elle peut aussi utiliser les variables locales ou les paramètres de la méthode dans laquelle elle est définie s'ils sont `final` ou effectivement `final`

Exemple

```
static double meilleureNotePromo(List<Etudiant> liste,
                                 int annee) {
    return liste.stream()
           .filter(e -> e.getAnPromo() == annee)
           .mapToDouble(e -> e.getMoyenne())
           .max().getAsDouble();
  }
```

Que fait cette méthode ?
```

Interface fonctionnelle

Type des expressions lambda

- Java est un langage typé
- Quel est le type d'une expression lambda ?

Signature de la méthode sort

- `villes.sort(`
  `(v1, v2) -> v1.getNbH() – v2.getNbH())`
- Dans l'interface `List<E>` :
  `sort(Comparator<? super E> c)`
- Comment
  `(v1, v2) -> v1.getNbH() – v2.getNbH()`
  peut convenir pour le type
  `Comparator<? super E>` ?

Correspondance

- `Comparator` a une seule méthode abstraite
- Cette méthode a le même nombre de paramètres et le même type retour que l'expression lambda
- `(v1, v2) -> v1.getNbH() – v2.getNbH()`
  est mis pour

```
new Comparator<Ville>() {
  public int compare(Ville v1, Ville v2) {
    return v1.getNbH() – v2.getNbH();
  }
}
```

Donc cette expression lambda convient pour le type `Comparator<Ville>`

Interface fonctionnelle

- Interface avec une seule méthode abstraite (peut aussi avoir des méthodes `default` ou `static`)
- Une EL est une instance d'une interface fonctionnelle
- Interface peut être annotée par `@FunctionalInterface` (`java.lang`)
- Annotation optionnelle mais conseillée
  - pour informer le lecteur
  - pour que le compilateur vérifie qu'il y a bien une seule méthode abstraite

Exemple d'interface fonctionnelle

- L'interface `Predicate<T>` (`java.util.function`) contient une seule méthode abstraite qui prend un paramètre de type `T` et qui retourne un `boolean`
- Une expression lambda de ce type sert à filtrer des éléments de type `T` pour ne retenir que ceux qui retournent `true`
- `Predicate` contient aussi 3 méthodes `default` (`and`, `or` et `negate`)

Autres interfaces fonctionnelles

- Le paquetage `java.util.function` contient des interfaces fonctionnelles ; voici les principales (en plus de `Predicate<T>`) :
  - `Consumer<T>` : consomme un `T` (sans résultat)
  - `Supplier<T>` : fournit un `T` (pas de paramètre)
  - `Function<T,R>` : fonction qui prend un `T` en paramètre et retourne un `R`
  - Variantes « Bi » (sauf pour `Supplier`) avec 2 paramètres ; par exemple `BiFunction<T,U,R>`

Exemples

```
Predicate<T> ?
Consumer<T> ?
Supplier<T> ?
Function<T,R> ?
```

- `liste.stream()` `Predicate<Etudiant>`
  `.filter(e -> e.getAnPromo() == 2013)`
  `.map(e -> e.getMoyenne())` `Function<Etudiant, Double>`
- `IntStream.range(1, 100)`
  `.filter(x -> x % 7 == 0)` `Predicate<Integer>`
  `.forEach(x -> System.out.println(x));`
  `Consumer<Integer>`

Variantes pour type primitif

- Interfaces avec des méthodes qui prennent les types primitifs (`boolean`, `double`, `int` ou `long`) comme types des paramètres ou type retour
- Exemple :
  `ToDoubleFunction<T>` fonction qui retourne un `double` (et prend un `T` en paramètre)

Exemple

- `liste.stream()`
  `.filter(e -> e.getAnPromo() == 2013)`
  `.mapToDouble(e -> e.getMoyenne())`
  `ToDoubleFunction<Etudiant>`

Anciennes interfaces

- D'anciennes interfaces du JDK, comme `Comparator<T>`, sont maintenant des interfaces fonctionnelles

Référence de méthode ou de constructeur

Référence de méthode

- « `::` » sert à désigner une méthode (`static` ou non) d'une classe
- Peut être utilisé à la place d'une EL quand l'EL correspond à une méthode qui existe déjà
- Exemple :

```
listeEmp.sort(
  (e1, e2) -> Employe.compareParSalaire(e1, e2);
```

peut être remplacé par

```
listeEmp.sort(Employe::compareParSalaire);
```

`compareParSalaire` est définie dans quelle classe ?

Son type retour ?

`static` ou non ?

3 types de référence de méthode

- Classe::méthode-static ; `Math::pow`
  équivalent à `(x, y) -> Math.pow(x, y)`
- objet::méthode-d'instance ; `System.out::println`
  équivalent à `x -> System.out.println(x)`
- Classe::méthode-d'instance ; `String::compareTo`
  équivalent à (le 1er paramètre est l'objet à qui on envoie le message) `(x, y) -> x.compareTo(y)`

Utilisation référence de méthode

```
class IntPredicates {
   public static boolean isPair(Integer n) {
     return n % 2 == 0;
   }
}
List<Integer> nombres =
   Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> pairs =
   nombres.stream().filter(IntPredicates::isPair)
   .collect(Collectors.toList());
```

Quel cas ?

Classe::méthode-static

Correspond à ?

`x -> IntPredicates.isPair(x)`

Référence de constructeur

- « *`UneClasse`*`::new` » sert à désigner un constructeur de la classe *`UneClasse`*
- Équivaut à une EL dont les paramètres sont les paramètres du constructeur
- Exemple :

```
List<String> chiffres =
  Arrays.asList("1", "2", "3", "4", "5");
List<Integer> nombres =
  chiffres.stream()
  .map(Integer::new)
  .collect(Collectors.toList());
```

Qu'est-ce que ça fait ?

Correspond à ?

`x -> new Integer(x)`

Classes « Optional »

Présentation

- Classe `java.util.Optional<T>` pour faciliter la gestion des valeurs `null`
- Une instance de `Optional<T>` est un container qui peut contenir *ou non* une valeur de type `T` (la valeur peut être `null`)
- `OptionalDouble`, `OptionalInt` et `OptionalLong` pour les type primitifs
- Classes utilisées par les streams ; par exemple le max d'une liste est un Optional qui ne contient aucune valeur si la liste est vide

Exemple basique

Que fait ce code ?

```
class Math2 {
  public static OptionalDouble sqrt(Double x) {
    return x < 0 ? OptionalDouble.empty()
                 : OptionalDouble.of(Math.sqrt(x));
  }
  ...
}
...
OptionalDouble r = Math2.sqrt(x);
if (r.isPresent()) {
  double racine = r.getAsDouble();
  ...
}
```

Crée un `OptionalDouble` qui contient la valeur passée en paramètre

Méthode « `orElse` »

- `orElse` peut remplacer un « if then else » par une simple instruction
- `T orElse(T autre)`
  retourne l'objet s'il est présent, sinon retourne l'`autre` valeur passée en paramètre
- Exemple :

```
Taxe taxe = findTaxe(id).orElse(Taxe.DEFAUT);
```

au lieu de (si on n'utilise pas `Optional<Taxe>`) :

```
Taxe taxe = findTaxe(id);
if (taxe == null) {
  taxe = Taxe.Defaut;
}
```

`findTaxe` retourne un `Optional<Taxe>`

Présentation des streams

Boucle explicite

```
List<Etudiant> etudiants = …
double max = 0;
for (Etudiant e : etudiants) {
  if (e.getAnPromo() == 2013) {
    if (e.getMoyenne() > max) {
      max = e.getMoyenne();
    }
  }
}
```

Que contiendra `max` ?

Boucle implicite – Lambdas

```
List<Etudiant> etudiants = …
double max = etudiants
  .filter(e -> e.getAnPromo() == 2013)
  .map(e -> e.getMoyenne())
  .max();
```

Ça aurait pu être comme cela mais ça ne marche pas !

Boucle implicite – Lambdas

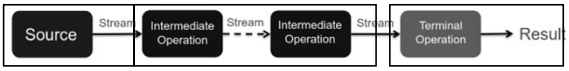
```
List<Etudiant> etudiants = …
double max = etudiants.stream()
  .filter(e -> e.getAnPromo() == 2013)
  .mapToDouble(e -> e.getMoyenne())
  .max().getAsDouble();
```

Pourquoi des streams ?

- L'utilisation des streams permet, sans avoir à écrire du code complexe, de
  - paralléliser des opérations
  - retarder le plus possible l'exécution des certaines opérations (« *lazyness* »)
  - stopper le traitement dès que le résultat attendu est obtenu, sans nécessairement traiter tous les éléments

Pipeline de streams

1. Création du stream à partir d'une source
2. Appliquer des opérations intermédiaires (0 ou plusieurs) pour obtenir de nouveaux streams
3. Appliquer une opération terminale pour avoir le résultat ; le stream ne peut plus être utilisé ensuite



Remarque importante : une opération prend un stream en entrée et retourne un **autre** stream ; ni le stream en entrée ni la source ne sont modifiés par l'opération

Retarder les opérations (1/1)

`anyMatch` retourne `true` si un des éléments vérifie le prédicat

```
List<Etudiant> etudiants = …
boolean uneMentionTB =
  etudiants.stream()
  .filter(e -> e.getAnPromo() == 2013)
  .mapToDouble(e -> e.getMoyenne())
  .anyMatch(m -> m >= 16);
```

Que fait ce code ?

- Inefficace de lancer `filter` sur tous les éléments puis `mapToDouble`, et ensuite de lancer `anyMatch`, alors que, peut-être, le 1er étudiant a une moyenne de 17

Retarder les opérations (2/2)

- Les streams accumulent *automatiquement* les opérations intermédiaires et ne lancent les opérations que lorsqu'une opération terminale est rencontrée
- `anyMatch` est une opération terminale ; elle déclenche les opérations sur le 1er élément du stream :
  - le retenir s'il appartient à la promotion 2013
  - s'il convient, récupérer sa moyenne (sinon passer au 2ème étudiant)
  - finalement voir si cette moyenne est >= 16
  - si c'est le cas le pipeline se termine en retournant `true`, sinon, le 2ème élément est traité, et ainsi de suite

Stream parallèle

- Il suffit de remplacer `stream()` par `parallelStream()` pour obtenir un stream qui effectuera des opérations en parallèle sur les éléments du stream :

```
listeNombres.parallelStream().map(x -> 2*x)
```

Que fait `map` ?

- On peut aussi paralléliser un stream déjà créé, avec la méthode `parallel()` :

```
stream.parallel(). …
```


Paquetage `java.util.stream`

- Contient les classes et interfaces liées aux streams
- En particulier l'interface `Stream<T>` correspond à une suite d'éléments de type `T`
- `Stream` contient des méthodes pour
  - créer un stream (méthodes `static`)
  - appliquer des opérations intermédiaires ou terminales aux éléments du stream

Sources de streams

Création d'un stream avec une collection

- Les méthodes de `Collection<E>`
  `default Stream<E> stream()`
  `default Stream<E> parallelStream()`
  créent un `Stream` avec les éléments d'une collection

Modification de la collection

- Si un stream s'appuie sur une collection, il ne faut pas modifier la collection tant que le traitement du stream n'est pas terminé

Ça vous rappelle quelque chose ?

Création d'un stream avec un tableau

- `static <T> Stream<T> of(T... valeurs)`
  (méthode de `Stream`) crée un `Stream<T>` à partir d'un tableau ou d'une suite d'éléments de type `T`

Création d'un stream de nombres

- `static IntStream range(`
  `int débutInclus, int finExclue)`
  (méthode de `IntStream`) crée un `IntStream` composé des nombres entiers compris entre les nombres donnés en paramètre ; la méthode `rangeClosed` inclut la fin
- Exemple :

```
IntStream.range(1, 100)
  .filter(x -> x % 7 == 0)
  .forEach(System.out::println);
```


Création d'un stream infini

- Ces méthodes `static` de `Stream` créent des streams « infinis » :
- `generate` convient pour créer un stream dont les éléments sont fournis par un `Supplier`
- `iterate` convient pour créer une « suite récurrente » (éléments fournis par un `UnaryOperator`)

Exemple

- Stream pour suite définie par $u_{n+1} = 3\,u_n + 2$

```
Stream<BigInteger> s =
  Stream.iterate(
    BigInteger.ONE,
    n -> n.multiply(new BigInteger("3"))
                  .add(new BigInteger("2")));
```

- Afficher les 5 premiers éléments de ce stream : `s.limit(5).forEachOrdered(System.out::println);`
- Afficher le 100ème : `System.out.println(s.skip(99).findFirst().get());`

A quoi sert le `get()` final ? Pensez qu'un stream peut être vide.

Opérations intermédiaires

Présentation

- Un stream peut être traité par des opérations
- Une opération représente le type de traitement qui sera fait sur le stream : sélection (filter), transformation (map), tri (sort), …
- Une opération prend le plus souvent en paramètre une expression lambda utilisée par l'opération

Exemple avec `filter`

- La méthode `filter` représente une sélection qu'on va faire sur les éléments
- Pour indiquer le critère qui servira à sélectionner les éléments, une expression lambda de type `Predicate` est passée à la méthode :

```
etudiants.stream()
         .filter(e -> e.getMoyenne() > 10)
```

Que fait cette opération ?

Des opérations intermédiaires

- Filtre pour sélectionner des éléments du stream (`filter`)
- Transformer les éléments du stream pour les mettre dans un autre stream (`map` et `flatMap`)
- Opérations avec état : trier (`sorted`), extraire un « sous-stream » (`limit` et `skip`), enlever les doublons (`distinct`)

`map` de `Stream<T>`

- `<R> Stream<R> map(Function<? super T,? extends R> f)` retourne un stream constitué des éléments formés en appliquant une fonction f : `T` → `R` aux éléments de ce stream :

```
mots.stream()
    .map(x -> x.length())
```

Que fait cette opération ?

-
```
mots.stream()
    .mapToInt(x -> x.length())
```

Plus performant si on veut ensuite travailler sur des entiers

flatMap de Stream<T>

- `<R> Stream<R> flatMap(`
  `Function<? super T,`
  `? extends Stream <? extends R> f)`

  différence avec `map` : la fonction `f` retourne un stream de `R` et pas un `R` ;
  mais ce sont les éléments de ce stream qui sont mis dans le nouveau stream et pas le stream lui-même

Exemple

`Files.lines` retourne un stream de toutes les lignes du fichier identifié par le chemin

```
Stream<String> lignes =
  Files.lines(chemin, StandardCharsets.UTF_8);
Stream<String> mots =
  lignes.flatMap(
    ligne -> Stream.of(ligne.split(" +")));
```

`ligne.split(" +")` retourne un tableau composé des mots de la ligne ; `Stream.of` crée un stream à partir des éléments du tableau

Le stream des mots d'une ligne est « aplati » pour mettre tous les mots de toutes les lignes dans un seul stream (référencé par `mots`)

Exemple

```
Stream<String> lignes =
  Files.lines(chemin, StandardCharsets.UTF_8);
long n =
  lignes
  .flatMap(ligne -> Stream.of(ligne.split(" +")))
  .filter(mot -> mot.length() > 2)
  .map(String::toLowerCase)
  .distinct()
  .count();
```

Quelle sera la valeur de `n` ?

Tri de stream

- `Stream<T> sorted()`
  trie le stream suivant l'ordre naturel
  (`ClassCastException` si pas d'ordre naturel sur `T`)
- `Stream<T> sorted(`
  `Comparator<? super T> comparateur)`
  trie le stream suivant l'ordre de `comparateur`

Opérations terminales

- Une opération terminale traite les données transmises par le stream et fournit un résultat qui n'est pas un stream, terminant ainsi le pipeline des traitements
- Une opération terminale déclenche le début de l'exécution du pipeline (ce qui permet des traitements « lazy »)

Des opérations terminales

- Exécuter une action pour chaque élément du stream (`forEach`, `forEachOrdered`)
- `max`, `min` (prennent un `Comparator` en paramètre et retournent un `Optional<T>`), `count`
- `anyMatch`, `allMatch`, `noneMatch` (un `Predicate` est passé en paramètre)
- Réduction (`reduce`)
- Collecteur (`collect`)
- Retourner les éléments du stream dans un tableau (`toArray`)

`forEach`, `forEachOrdered`

- `void forEach(Consumer<? super T> action)`
- `liste.stream().forEach(System.out::println);`
- Si on veut que l'ordre d'un stream ordonné soit respecté il faut utiliser plutôt `forEachOrdered`

Réduction

- Une « réduction » fournit une valeur après un traitement sur un stream
- Le JDK fournit plusieurs réductions « classiques » comme le calcul de la somme des éléments d'un stream d'entiers
- `reduce` et `collect` permettent aussi des réductions à usage général (pas étudiés dans ce cours d'introduction)

Exemple avec `max`

```
Files.lines(path)
  .max(Comparator.comparingInt(String::length))
  .get();
```

A quoi sert ce `get()` ?

Quel sera le résultat final ?

Collector

- Permet d'accumuler des éléments dans un container

`java.util.stream.Collectors`

- Classe utilitaire qui contient de nombreuses méthodes `static` qui retournent les collecteurs les plus fréquemment utilisés
- Les méthodes retournent un `Collector` utilisable avec une des méthodes `collect` de `Stream<T>`

Exemples d'utilisation des collecteurs

- ```
  List<String> noms =
    employes.stream()
    .filter(p -> p.getGenre().equals("M")
    .map(p -> p.getNom())
    .collect(Collectors.toList());
  ```
  Qu'est-ce que ça fait ?
- ```
  Set<String> noms =
    employes.stream()
    .filter(p -> p.getGenre().equals("M")
    .map(p -> p.getNom())
    .collect(Collectors.toSet());
  ```
  Qu'est-ce que ça fait ?

  Quelle différence avec le 1er exemple ?

Exemple d'utilisation des collecteurs

- ```
  Map<String, List<Employe>> parGenre =
    employes.stream()
    .collect(
       Collectors.groupingBy(Employe::getGenre));
  ```
  Qu'est-ce que ça fait ?

Aide : Quel est le type de `parGenre` ?

Streams de type primitif

Utilité

- Des streams dont les éléments ont un type primitif permettent d'améliorer les performances en évitant les opération de boxing/unboxing :
  - `DoubleStream`
  - `IntStream`
  - `LongStream`
- Ces interfaces héritent de `Stream` et contiennent des méthodes supplémentaires ; par exemple pour calculer la somme des éléments du stream avec la méthode `sum()`

Exemple

```
int somme =
  IntStream.of(1, 2, 3, 4).parallel()
  .map(x -> 2 * x)
  .sum();
```

Valeur de somme ?

Méthodes pour ordre naturel

- Les méthodes `min()` et `max()` (sans paramètre de type `Comparator`) utilisent l'ordre naturel sur les types primitifs
```

Stream<T> ↔ stream de type primitif

- Des méthodes permettent de passer des Stream<T> aux streams de type primitif et vice versa

Exemple

- Pour passer d'un `LongStream` à un `Stream<String>` :

```
Stream<String> s2 = s1.mapToObj(Long::toString);
```