Java 8: Expressions Lambda et API Stream

Ingénierie des Systèmes d'Information 3

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2018 - 2019

Plan

Plan

Paramétrage de comportements

Objectifs

Pouvoir modifier dynamiquement les comportements (ou algorithmes) utilisés par une classe.

Solution

Encapsulation du comportement (qui peut varier) en dehors de la classe qui l'utilise (élément stable).

Exemple avec la Calculatrice :

```
public interface Operation
{       public double calcul (double a, double b);};
public class Calculette
{
        private double val1;
        private double val2;
        ...
        public double calcul (Operation op)
        {
                return op.calcul(val1,val2);
        }
}
```

Paramétrage de comportements

Solution 1 : Classe anonyme :

```
Calculette calc = new Calculette( 2.5,3);
Operation add = new Operation() {
  public double calcul(double a, double b)
  {
        return a+b;
  }
};
double result = calc.calcul(add);
```

Solution 2 : Pattern Strategie : différentes stratégies implémentent l'interface `Operation`, `Calculette` a un attribut de type `Operation`, ...

Solution 3 : Expression lambda :

```
Calculette calc = new Calculette( 2.5, 3);
Operation add = (a,b) -> a+b;
double result = calc.calcul(add);
//ou directement sans la variable add
double result = calc.calcul((a,b) -> a+b);
```

Plan

Interface fonctionnelle

Interface fonctionnelle

Une Interface fonctionnelle est une interface avec **une seule méthode abstraite** (la méthode fonctionnelle). Elle peut avoir une ou plusieurs méthodes par défaut.

Exemple d'une interface fonctionnelle

```
@FunctionalInterface
public interface Operation
{
        public double calcul (double a, double b);

        default void affiche(double a,double b) {
                System.out.println("resultat:"+calcul(a,b));
        };
};
```

Plan

Expression lambda

Expression lambda

Fonction anonyme ayant une liste de paramètre, un corps, et un type de retour.
Notation d'une expression Lambda :

- (paramètre 1, paramètre 2, ...) → expression ;
- (paramètre1, paramètre2, ...) → {statements ; } ;
- paramètre1 → expression ;

Inférence de types : le type des variables de l'expression lambda peut ne pas être déclaré.

Exemples d'expressions Lambda

- () → {} ;
- () → () ;
- () → { return "Mario" ; } ;
- (Integer i) → {return 2*i ;} ;
- i → 2*i ;
- (i,r)→ i*r ;

Expression lambda

Utilisation : une lambda peut remplacer toute valeur dont le type est une interface fonctionnelle.

- Exemple d'assignation de variable :

```
Operation add = (a,b) -> (a+b);
```

- Exemple en argument d'une méthode qui attend une interface fonctionelle (l'expression lambda doit avoir la même signature que l'interface fonctionelle attendue).

```
double result = calc.calcul((a,b) -> a+b);
```

```
Operation add = (double a, double b) -> a+b;
Operation sub = ( a, b) -> a-b;
Operation mul = ( a, b) -> {return a*b;};
Operation[] tabOp = {add, sub, mul};
int no = (int)(Math.random()*3);
Operation o = tabOp [ no ] ;
o.affiche(5.2, 2.5);
```

Interfaces fonctionnelles existantes

Java8 définit plusieurs interfaces fonctionnelles basiques dans `java.util.function` `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`

Functional interface	Descriptor	Method name
Predicate<T>	T → boolean	test()
BiPredicate<T, U>	(T, U) → boolean	test()
Consumer<T>	T → void	accept()
BiConsumer<T, U>	(T, U) → void	accept()
Supplier<T>	() → T	get()
Function<T, R>	T → R	apply()
BiFunction<T, U, R>	(T, U) → R	apply()
UnaryOperator<T>	T → T	identity()
BinaryOperator<T>	(T, T) → T	apply()

Quelques interfaces fonctionnelles existantes

Interfaces fonctionnelles existantes

L'interface `Operation` peut être remplacée par l'interface fonctionnelle existante `BinaryOperator` :

```
public class Calculette
{
        ...
        public double calcul (BinaryOperator<Double> op)
        {
                return op.apply(val1,val2);
        }
}

Double result = calc.calcul((a,b)->a+b);
```

Exemple d'utilisation de l'interface fonctionelle `Runnable` :

```
Runnable r = () -> System.out.println("I'm running");
r.run();
```

Plan

Référence de méthodes

Si une expression lambda appelle une seule méthode directement, on peut faire référence à la méthode par son nom au lieu d'utiliser l'expression lambda.

Les expressions lambda suivantes :

```
Function<String, Integer> lengthCalculator = s -> s.length();
Consumer<String> l = s -> System.out.println(s);
```

peuvent être réécrites avec des références de méthode :

```
Function<String, Integer> lengthCalculator = String::length;
Consumer<String> l = System.out::println;
```

Plan

Expression lambda et Liste/Collections

Certaines méthodes de `Collections` et `List` utilisent des interfaces fonctionnelles.

Tri de liste

- La fonction `void sort(Comparator c)` trie les éléments de la liste.
- L'interface fonctionnelle `Comparator<T>` propose la méthode fonctionelle `int compare(T o1, T o2)`.

On peut donc passer une expression lambda en paramètre de `sort` :

```
ArrayList<String> name = new ArrayList<String>(Arrays.asList("toto",
    "anna","cedric"));
name.sort((a,b) -> a.compareTo(b));
```

Expression lambda et Liste/Collections

Parcours de liste

- La fonction `void forEach(Consumer action)` applique une fonction (dont le retour est `void`) à chaque élément de la liste.
- L'interface fonctionnelle `Consumer<T>` propose la méthode fonctionnelle `void accept(T t)`.

```
name.forEach(System.out::println);
```

Modification d'éléments

- La fonction `void replaceAll(UnaryOperator operator)` remplace chaque élément de la liste.
- La fonction `boolean removeIf(Predicate filter))` retire les éléments de la liste répondant à la condition.

```
name.removeIf(a -> (a.length() > 4));
name.replaceAll(a -> a.toUpperCase());
```

Plan

Plan

Les Flux (API Stream)

Stream ou Flux de données

- Séquence d'éléments sur laquelle on peut effectuer un groupe d'opérations de manière séquentielle ou parallèle.
- Un Stream permet de manipuler simplement un flux de données quelconque.
- Interface `Stream<T>` (`https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html`)

Exemple

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
liste.stream().filter(s->s.contains("to")).distinct()
        .forEach(s->System.out.println(s.length()));
```

Les Flux (API Stream)

Flux vs. Collection

- Les Collections concernent les données (stockage, accès aux éléments), les Flux concernent les calculs sur ces données (filtre, tri, recherche, réduction, ...).
- Les Flux permettent de manipuler des collections de données de manière déclarative : on exprime une requête plutôt que de spécifier une implémentation.
- Manipulation d'un ensemble de données de manière optimisée : un Flux ne stocke pas les données mais les transfère d'une source vers une suite d'opérations.
- Le chargement des données pour des opérations sur un Flux s'effectue de façon paresseuse.

Les Flux (API Stream)

L'utilisation d'un flux se compose de trois étapes :

1. La création du Stream à partir d'une source de données
2. La déclaration d'opérations intermédiaires. Chacune de ces opérations retourne un nouveau flux, mais ces derniers sont créés de manière lazy (paresseuse) : aucun traitement n'est effectué lors de leur déclaration.
3. Un appel d'une opération terminale qui va fournir un résultat. C'est elle qui va déclencher le parcours des données et exécuter les différentes opérations intermédiaires.

Les Flux (API Stream)

1- Source de données

- Un Stream se construit à partir d'une source fournissant les données comme une collection, un fichier, ...
  - appel de la méthode `Stream<E> stream()` sur une collection.

```
List<String> liste = Arrays.asList("toto","tata","anna","
    tom");
Stream<String> flux = liste.stream();
```

  - appel de la méthode `Stream<String> lines()` de la classe `BufferedReader`

Les Flux (API Stream)

1- Source de données

- On peut remplir automatiquement un flux avec la méthode statique `Stream.iterate(T debut, UnaryOperator<T> iteration)`

```
//creation d'un flux contenant les entiers de 0 a 9 :
Stream<Integer> s = Stream.iterate(0, i->i+1).limit(10) ;
```

- On peut créer un flux en spécifiant ses valeurs avec la méthode statique `Stream.of(T... values)`

```
//creation d'un flux contenant 4 chaines
Stream<String> s = Stream.of("toto","tata","anna","tom");
```

- Traitements en parallèle du flux : méthode `parallelStream()` sur une `Collection`.

Les Flux (API Stream)

2- Pipelining et Opérations intermédiaires

- Un Stream ne modifie pas les données de la source : s'il doit les modifier, il va construire un nouveau Stream.
- Une opération intermédiaire sur un Stream retourne un Stream.

Exemple d'opérations intermédiaires

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
//filter exclut les elements qui ne correspondent pas au predicat
Stream<String> fluxFilter = liste.stream().filter(s->s.contains("to"
    ));
//distinct  filtre les doublons (selon equal)
Stream<String> fluxDistinct = fluxFilter.distinct();
```

Les Flux (API Stream)

2- Pipelining et Opérations intermédiaires

- On peut donc **chainer** des opérations intermédiaires (pipeline de flux).

Exemple de chainage d'opérations intermédiaires sur les Flux

Cet exemple est équivalent à l'exemple précédent.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
Stream<String> fluxDistinct = liste.stream().filter(s->s.contains("
    to")).distinct();
```

Les Flux (API Stream)

3- Opérations terminales

- Les Stream issus des opérations intermédiaires sont créés de manière paresseuse : aucun traitement n'est effectué lors de leur déclaration.
- Une opération terminale lance l'exécution du pipeline de flux et produit un résultat.

Exemple d'opération terminale

L'opération `forEach` applique une expression lambda (de type `Consumer`) à chaque élément du flux, puis renvoie void.

```
liste.stream().filter(s->s.contains("to")).distinct()
        .forEach(s->System.out.println(s.length()));
```

Les Flux (API Stream)

4- Consommation unique

- Un Stream ne peut être parcouru (consommé) qu'une seule fois.

Exemple : Un seul parcours possible pour un flux

```
List<String> liste = Arrays.asList("toto","tata","anna","tom");
Stream<String> flux = liste.stream();
flux.forEach(System.out::println);
//lance exception: second parcours impossible
System.out.println("count:"+flux.count());
```

```
List<String> liste = Arrays.asList("toto","tata","anna","tom");
Stream<String> flux = liste.stream();
Stream<String> s3 = flux.filter(s->s.contains("to"));
//lance exception: second parcours impossible
Stream<String> s4 = flux.distinct();
```

Déclenche une erreur `Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed`

Plan

Opérations intermédiaires

Filtrage

- `filter(Predicate<T> p)` filtre les éléments du flux pour ne conserver que ceux correspondant au prédicat
- `distinct()` filtre les doublons (selon `equals`)
- `limit(long n)` tronque le flux en ne gardant que les `n` premiers éléments
- `skip(long n)` tronque le flux en enlevant les `n` premiers éléments

Exemple

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto");
Stream<String> fluxDistinct = liste.stream().filter(s->s.contains("to")).distinct();
```

Le nouveau Flux contient "toto" et "tom".

Opérations intermédiaires

Tri

- `sorted(Comparator<T> comp)` tri les éléments du flux selon le comparateur.

Exemple : tri dans l'ordre alphabétique

```
List<String> liste = Arrays.asList("toto","tata","anna","tom");
Stream<String> listeTrie = liste.stream().sorted((string1,string2)->
    string1.compareTo(string2));
```

Opérations intermédiaires

Mapping

- `map(Function<T,R> mapper)` applique une transformation/fonction à chaque élément du flux.
- `flatmap(Function<T,Stream<R>> mapper)` (aplatissement) : similaire à `map` mais la fonction appliquée à chaque élément renvoie un Flux ; `flatmap` se charge alors de concaténer tous ces flux en un seul.

Exemple

```
List<String> liste = Arrays.asList("toto","tata","anna","tom");
Stream<Integer> fluxInt = liste.stream().map(s->s.length());
//lines() renvoie un Stream<String> pour chaque element
Stream<Stream<String>> fluxStream1 =  liste3.stream().map(s->s.lines
    ());
//il faut utiliser flatMap pour concatener tous les flux en un seul
Stream<String> fluxStream2 =  liste.stream().flatMap(s->s.lines());
```

Le nouveau flux d'entiers contient 4,4,4,3.

Plan

Opérations terminales

ForEach

L'opération `forEach` applique une expression lambda (de type `Consumer`) à chaque élément du flux, puis renvoie void.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
liste.stream().filter(s->s.contains("to")).distinct()
        .forEach(s->System.out.println(s.length()));
liste.stream().forEach(System.out::println);
```

Count

L'opération `count` retourne le nombre d'éléments dans le Flux.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
double d = liste.stream().count();
```

Opérations terminales

Trouver

- L'opération `allMatch(Predicat p)` retourne `vrai` si tous les éléments répondent au prédicat.
- L'opération `anyMatch(Predicat p)` retourne `vrai` si un des éléments répond au prédicat.
- L'opération `noneMatch(Predicat p)` retourne `vrai` si aucun des éléments ne répond au prédicat.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
System.out.println(liste.stream().allMatch(s->s.length()<5));//true
System.out.println(liste.stream().allMatch(s->s.length()==4));//
    false
System.out.println(liste.stream().anyMatch(s->s.length()==4));//true
```

Opérations terminales

Trouver

- L'opération `findAny()` retourne un des éléments du Flux.
- L'opération `findFirst()` retourne le premier élément du Flux.
- L'opération `max(Comparator<? super T> comparator)` renvoie l'élément maximum selon le Comparateur.
- L'opération `min(Comparator<? super T> comparator)` renvoie l'élément minimum selon le Comparateur.

Ces opérations renvoient un `Optional` (cf. §4).

```
System.out.println("max length:"+liste.stream().map(s->s.length()).
    max((a,b)->a-b).get());
```

Opérations terminales : réduction

Opérations de réduction simple

- Une réduction combine les éléments du flux en un seul résultat, en appliquant une ou plusieurs opérations sur l'ensemble des éléments.
- L'opération `reduce(T identity, BinaryOperator<T> accumulator)` :
  - `identity` est l'élément initial (et l'élément renvoyé si le stream est vide).
  - `accumulator` crée un nouveau résultat partiel à partir d'un résultat partiel et d'un nouvel élément
- Exemple de réduction : somme, min, max, moyenne, ...

```
Stream<Integer> integers = Stream.of(1,2,3,4);
Integer sum = integers.reduce(0, (a, b) -> a+b);
```

Opérations terminales : réduction

Conversion d'un flux en une structure de données

La méthode `collect(Collector col)` permet de repasser des Streams aux Collections. On utilise les méthodes statiques de `Collectors`.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto"
    );
List<String> result = liste.stream().filter(s->s.contains("to")).
    distinct().collect(Collectors.toList());
```

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
                    .collect(Collectors.groupingBy(Employee::
                        getDepartment));
```

Opérations terminales : réduction

Classe Collectors

La classe `Collectors` permet d'effectuer des opérations terminales sur les streams. cf. `https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html`

```
// Compute sum of salaries of employee
int total = employees.stream()
        .collect(Collectors.summingInt(Employee::getSalary)));
```

Method	Return type
toList()	List<T>
toSet()	Set<t>
toCollection()	Collection<T>
counting()	Long
summingInt()	Long
averagingInt()	Double
joining()	String
maxBy()	Optional<T>
minBy()	Optional<T>
reducing()	...
groupingBy()	Map<K, List<T>>
partioningBy()	Map<Boolean, List<T>>

Quelques méthodes de la classe `Collectors`

Plan

Classe Optional<T>

Classe Optional<T>

- Permet de représenter l'existence ou l'absence de valeur (évite de renvoyer `null`).
- Conteneur d'objet pouvant être vide.
- S'il n'est pas vide, `isPresent()` retourne `vrai` et `get()` retourne l'objet.
- S'il est vide, `isEmpty()` retourne `vrai` et `get()` renvoie `NoSuchElementException`.

```
List<String> liste = Arrays.asList("toto","tata","anna","tom","toto");
Optional<Integer> optMax = liste.stream().map(s->s.length()).max((a,b)->a-b);
if (optMax.isPresent())
                        System.out.println("max :"+optMax.get());
Optional<String> optFirst = liste.stream().findFirst();
```

Plan

Flux de types primitifs

Exemple d'utilisation de `reduce` pour sommer les éléments d'un flux. Mais ici, cout d'autoboxing (convertir chaque `Integer` en type primitif `int`).

```
Stream<Integer> integers = Stream.of(1,2,3,4);
int sum = integers.reduce(0, Integer::sum);
```

Flux de types primitifs

- Interfaces pour les flux avec des éléments de types primitifs : `IntStream`, `DoubleStream` et `LongStream`
- Nouvelles méthodes de réduction sur ces flux (sum, max, min, ...)

```
List<String> liste = Arrays.asList("toto","tata","anna","tom");
IntStream intstream = liste.stream().mapToInt(s->s.length());
int s = intstream.sum();
```

Plan

Opération Statefull vs. stateless

Opération Stateless

Une opération `stateless` (sans état) est un type d'opérations intermédiaires qui ne gère aucun état pendant le traitement du flux, ce qui permet de paralléliser les traitements sans trop de surcouts. Ces opérations se basent uniquement sur l'élément courant du flux, indépendamment des autres éléments.

- Exemple : `map`, `filter`

Opération Statefull vs. stateless

Opération Statefull

Une opération `statefull` (avec état) est un type d'opérations intermédiaires qui doivent gérer un état pour effectuer correctement leur traitement. Ces opérations peuvent devenir très coûteuses si on manipule un flux de données à la fois ordonné et en parallèle. Dans ce cas-là, il peut être préférable d'utiliser un traitement séquentiel (via l'opération `sequential()`) ou d'ignorer l'ordre des éléments (via l'opération `unordered()`).
Exemple :

- `reduce`, `sum` nécessitent un état interne de taille limitée (*stateful-bounded*).
- `sorted` ou `distinct` nécessitent un état interne de taille potentiellement infinie ! (*stateful-unbounded*).