

**Java 8**

***Non ce n’est pas que les streams***



**Lucile Thienot, Jérôme Tama**

Fiche de suivi de document

Identification

|  |  |
| --- | --- |
| **Titre du document :** | Article Java 8 |
| **Version du document :** | 1.0 |
| **Unité :** | ONEPOINT |
| **Client :** |  |

Historique

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Nature | Auteur | Date |
| 1.0 | Version initiale | j.tama/l.thienot | 16/11/2021 |

SOMMAIRE

Table des matières

[1. INTRODUCTION 4](#_Toc88036140)

[2. INNER CLASSES TO LAMBDA 4](#_Toc88036141)

[2.1. Mais c’est quoi une lambda ? 5](#_Toc88036142)

[2.2. Utilisons-les pour mieux les comprendre 5](#_Toc88036143)

[3. CONSUMER, SUPPLIER, FUNCTION AND MANY OTHERS 6](#_Toc88036144)

[3.1. L’essentiel 6](#_Toc88036145)

[3.2. Plus en détail 7](#_Toc88036146)

[3.2.1. Des basiques … 7](#_Toc88036147)

[3.2.2. … Aux composées 8](#_Toc88036148)

[4. STREAMS 10](#_Toc88036149)

[4.1. Créer un Stream : générateurs de flux 11](#_Toc88036150)

[4.2. Appliquer des opérations : méthodes intermédiaires 11](#_Toc88036151)

[4.3. Obtenir un résultat : méthodes terminales 13](#_Toc88036152)

[4.3.1. Dénombrer 13](#_Toc88036153)

[4.3.2. Réduire 13](#_Toc88036154)

[4.3.3. Rechercher 15](#_Toc88036155)

[4.4. Découpage 15](#_Toc88036156)

[5. OUI MAIS ALORS ON A PERDU LES NOMS ? 15](#_Toc88036157)

[5.1. Référence de méthode 16](#_Toc88036158)

[5.1. Lambdas nommées 16](#_Toc88036159)

[6. CAS REEL 18](#_Toc88036160)

[6.1. Cas 1 18](#_Toc88036161)

[6.2. Cas 2 18](#_Toc88036162)

[6.2.1. Mieux avec lambda 19](#_Toc88036163)

[6.2.2. Mieux avec lambda 19](#_Toc88036164)

[6.2.3. Mieux legacy 20](#_Toc88036165)

[7. CONCLUSION 20](#_Toc88036166)

1. INTRODUCTION

Java 8 est sortie en 2014. À ce jour, les nombreuses nouveautés que cette version a apportées ne semblent toujours pas connues de la grande majorité des développeurs.

Nous aimerions tenter ici d’éclaircir ce que sont les lambdas et les streams. D’après ce que nous entendons lors de nombreuses qualifications techniques, le sujet reste nébuleux.

Nous essaierons aussi, modestement, d’y ajouter un petit retour d’expérience sur leur utilisation ...

1. INNER CLASSES TO LAMBDA

Plongeons tout de suite dans les lambdas.

Pour nous servir d’exemple nous utiliserons l’interface org.springframework.jdbc.core.RowMapper :

public interface RowMapper<T> {

T mapRow(ResultSet rs, int rowNum) throws SQLException;

}

Les classes internes allègent le code en nous permettant de ne pas créer des classes à tour de bras pour finalement ne redéfinir ou n’implémenter qu’un nombre réduit de méthodes.

Seulement ces classes internes restent très verbeuses : dans les classes internes de l’exemple ci-dessous, combien de lignes nous donnent réellement de l’information ? De combien de lignes devons-nous faire abstraction pour nous concentrer sur l’essentiel ?

public List<DataRecord> getDataRecords(String jobId){

return jdbcTemplate.query("select \* from DATARECORD ",

new RowMapper<DataRecord>(){ ⓵

@Override

public DataRecord mapRow(ResultSet rs, int rowNum) throws SQLException { ⓶

return new DataRecord(rs.getString("job\_id"), rs.getString("analysis\_name"));

}

}

);

}

⓵ Déclaration et utilisation de l’inner classe

⓶ Implémentation de la méthode.

Dans l’exemple du RowMapper, utilisé pour extraire le retour d’une requête sql jdbc, l’information que l’on retire de la définition de la classe interne est :

L'instance de la classe `DataRecord` est créée en exploitant les colonnes `job\_id` et `analysis\_name`.

Le reste n’est que syntaxe et bruit pour le lecteur. Et c’est ce que proposent de simplifier les expressions lambdas, introduites en java 8.

Le raisonnement est le suivant :

* le RowMapper n’a qu’une unique méthode à redéfinir. On pourrait donc se passer de préciser la méthode redéfinie lors de l’implémentation.
* le type du paramètre d’entrée de la méthode peut se déduire de la définition de la méthode abstraite.
* le type paramétré du RowMapper se déduit du type de l’objet retourné par la méthode mapRow.

Les expressions lambdas permettent d’appliquer ces simplifications.

* 1. Mais c’est quoi une lambda ?

Les expressions lambdas sont un sucre syntaxique simplifiant l’implémentation de classe/interface.

Elles permettent de définir des fonctions sans les nommer. Elles peuvent être manipulées et exécutées dans un programme sans avoir un nommage figé.

Les lambdas s’écrivent de la façon suivante :

(Type1 param1, Type2 param2, .., TypeN paramN) -> { traitement }

* 1. Utilisons-les pour mieux les comprendre

Appliquons les simplifications énoncées ci-dessus à notre exemple :

public List<DataRecord> getRecords(String jobId) {

return jdbcTemplate.query("select \* from DATARECORD ",

(ResultSet rs, int rowNum) -> {

return new DataRecord(rs.getString("job\_id"), rs.getString("analysis\_name"));

});

}

Il ne reste que l’essentiel : la définition du corps de la méthode en fonction des paramètres d’entrée.

La donnée essentielle s’écrit maintenant:

(ResultSet rs, int rowNum) -> {

return new DataRecord(rs.getString("job\_id"), rs.getString("analysis\_name"));

}

où rs est le résultat de la requête, rowNum est le numéro de la ligne en cours.  
Dans notre exemple, le compilateur comprend de lui-même qu’on définit un RowMapper via l’implémentation

de son unique méthode.

Notons que dans une expression lambda :

* Si le type des paramètres peut être inféré, il peut être omis.
* S’il y a un et un seul paramètre, les parenthèses peuvent être omises.
* S’il n’y a qu’une seule instruction, les accolades autour du traitement, peuvent être omises. Dans ce cas, le mot clé return et le point-virgule ; de fin d’instruction peuvent eux aussi être omis.
* Le nom des paramètres est indépendant de celui défini dans la méthode implémentée.

Si on applique toutes ces règles on peut encore simplifier la lambda utilisée pour définir le RowMapper :

ResultSet rs, int rowNum) -> {

return new DataRecord(rs.getString("job\_id"), rs.getString("analysis\_name"));

}

Puis:

(resulSet, index) -> {

return new DataRecord(resulSet.getString("job\_id"), resulSet.getString(

"analysis\_name"));

}

Puis:

(rs, rowNum) -> new DataRecord(rs.getString("job\_id"), rs.getString("analysis\_name")))

ATTENTION : S’il est vrai que les lambdas peuvent améliorer la lisibilité du code. Ce n’est pas toujours le cas. En effet une lambda de 40 lignes posée comme une verrue au milieu de votre code ne facilitera la lecture du code pour personne.

Par ailleurs, il est strictement interdit de laisser sortir une *checked exception* d’une lambda. Si vous avez l’habitude d’en utiliser, c’est le moment ou jamais d’arrêter avec cette hérésie.

1. CONSUMER, SUPPLIER, FUNCTION AND MANY OTHERS

Les interfaces qui peuvent être implémentées sous la forme de lambdas ne possèdent qu’une unique méthode non implémentée. Elles sont appelées interfaces fonctionnelles et peuvent être annotées @FunctionalInterface. Cette annotation permet au compilateur de vérifier que l’interface est bien fonctionnelle, mais elle n’est en aucun cas obligatoire.

La classe Comparator<T> est un exemple d’interface fonctionnelle commune aux développeurs : sa méthode compare(param1,param2) renvoie le résultat de la comparaison entre deux objets de type T. Elle existait certes avant java 8, mais cette dernière version l’a dotée de l’annotation @FunctionalInterface pour la désigner comme telle.

Nous avons jusqu’ici utilisé des expressions lambdas pour implémenter des interfaces courantes comme RowMapper, Comparator ou Runnable. Mais les lambdas sont plus largement utilisées pour définir des méthodes à l’endroit même où elles sont utilisées, sans porter d’intérêt particulier à l’interface fonctionnelle sous-jacente. Il est possible de vouloir déclarer de telles méthodes lorsqu’elles ne sont utilisées qu’une fois par exemple, ou encore pour passer un traitement en paramètre d’un autre traitement. La méthode que l’on souhaite définir via une lambda doit ici encore implémenter une interface fonctionnelle. Java 8 fournit une bibliothèque d’interfaces fonctionnelles standard appelée java.util.function, qui permet au développeur d’avoir accès aux interfaces fonctionnelles les plus communes sans les définir lui-même.

* 1. L’essentiel

Les interfaces du package java.util.function (naissance avec Java 8) sont des fonctions génériques définissant les cas d’usages les plus courants:

* Une *function* prendra un ou plusieurs paramètres en entrée et retournera un résultat.
* Un *supplier* ne prendra pas de paramètre d’entrée et fournira un résultat.
* Un *consumer* prendra un ou plusieurs paramètres d’entrée et ne renverra aucun résultat.
* Un *operator* prendra un ou plusieurs paramètres du même type et fournira un retour de ce type.
* Un *predicate* prendra un ou plusieurs paramètres en entrée et retournera un booléen.
  1. Plus en détail
     1. Des basiques …

• Function<T,R> : désigne une fonction prenant un paramètre d’entrée de type T et retournant un objet de type R.

Function<Integer, String> getNumber = entier -> "Number " + entier;

Elle s’applique en utilisant apply :

Elle se décline en :

* UnaryOperator<T> : fonction dont les types d’entrée et de retour sont identiques. C’est donc une Function<T,T>.
* BiFunction<T,U,R> : fonction qui prend en entrée deux paramètres, le premier de T et le second de type U.
* BinaryOperator<T> : fonction qui prend en entrée deux paramètres de type T. C’est donc une BiFunction<T,T,T>. Par exemple :

BinaryOperator<Integer> add = (a,b) -> a+b;

• Consumer<T> : désigne une fonction prenant un paramètre d’entrée de type T et de type de retour void

Consumer<Integer> display = entier -> System.out.println("Number " + entier);

Elle s’applique en utilisant apply :

>> display.apply(1) ; // afficher Number 1

Il se décline également en BiConsumer<T,U> : prend en entrée deux paramètres, le premier de type T et le second de type U.

• Supplier<R> : désigne une fonction ne prenant aucun paramètre d’entrée et retournant un objet de type R.

Suppler<Double> randomFrom0To100 = () -> Math.random()\* 100 ;

Elle s’applique en utilisant get :

>> randomFrom0To100.get(); // affiche un nombre aléatoire en 0 et 100

• Predicate<T> : désigne une fonction prenant un paramètre d’entrée de type T et renvoyant un booléen. Exemple :

Predicate<String> isNull = (str) -> str == null;

Elle s’applique en utilisant test :

>> isNull.test("HelloJava8"); // renvoie faux

Les fonctions présentées ci-dessous ne permettent pas de manipuler des types primitifs. Des fonctions spécifiques existent pour ceux-ci :

* Paramètre d’entrée de type primitif :
  + IntFunction<R>, IntConsumer dont le type du paramètre d’entrée est int.
  + Et les autres dérivés sur le même modèle : DoubleFunction<R>, DoubleConsumer, DoubleUnaryOperator, LongFunction<R>, etc.
* Retour de type primitif :
  + IntSupplier dont le type de retour est int, DoubleSupplier dont le type de retour est double, etc.
  + ToIntFunction<T> dont le type de retour est int, et sur le même modèle : ToIntBiFunction<T,U>, ToLongFunction<T>, ToLongBiFunction<T,U>, ToDoubleFunction<T>, etc.
    1. … Aux composées

**Composition de prédicats**

La composition de prédicats permet de créer un prédicat par la combinaison logique de plusieurs prédicats.

Prenons l’exemple suivant :

Predicate<Sock> isRed;

Predicate<Sock> isBlue;

Predicate<Sock> isHole ;

Pour déterminer si une chaussette est rouge ou bleue et non trouée, on définit le prédicat ci-dessous :

Predicate<Sock> isRedOrBlueWithoutHoles = sock -> (isRed.test(sock) || isBlue.test(sock)) &&

!isHoled.test(sock);

L’interface Predicate propose des méthodes permettant une réécriture concise et naturelle de ce prédicat :

Predicate<Sock> isRedWithoutHoles = isRed.or(isBlue).and(not(isHoled));

**Composition de fonctions**

En mathématiques, la composition consiste à créer une fonction par l’application d’une fonction au résultat d’une autre fonction. Par exemple :

f(x) = x + 1  
g(x) = x2  
f(g(x)) = (x2) + 1   
g(f(x)) = (x + 1)2

où f(g(x)) et g(f(x)) sont des compositions de f et g.

En utilisant les fonctions du package java.util.function, on peut créer ces mêmes fonctions composées :

Function<Integer,Integer> f = x -> x + 1;

Function<Integer,Integer> g = x -> x^2;

Function<Integer,Integer> composition1 = x -> f.apply(g.apply(x)); // f(g(x))

Function<Integer,Integer> composition2 = x -> g.apply(f.apply(x)); // g(f(x))

Vous conviendrez aisément que ces apply successifs ne sont pas d’une lisibilité à toute épreuve. Pour clarifier ces compositions, Java 8 propose pour les *function* une interface plus commode :

Function<Integer,Integer> composition1 = f.compose(g); ⓵   
Function<Integer,Integer> composition2 = f.andThen(g); ⓶

La différence entre compose et andThen réside dans l’ordre d’évaluation des fonction :

⓵ équivalent à g.andThen(f)

⓶ équivalent à g.compose(f)

1. STREAMS

L’API Stream<T> a été introduite par Java 8. Le Stream est un objet java qui permet de définir via une API une série de traitements à réaliser sur une collection ou tableau d’objets. Cette API nécessite largement l’utilisation de lambdas et de références de méthodes (que nous verrons plus tard). En revanche, aucune structure de boucle n’est nécessaire pour appliquer ces opérations.

Avec les Streams, la programmation java peut devenir déclarative au lieu d’être impérative : déclarer ce que doit faire un traitement et non comment il doit le faire. Dans ce cadre déclaratif, les fonctions sont manipulées comme n’importe quel objet java pour être passées en paramètre ou retournées.

Prenons un exemple :

String[] tableau = {"toto", "titi", "tata", "toto", ""};   
long count = Stream.of(tableau) ⓵   
 .filter(item -> item.isBlank()) ⓶   
 .distinct() ⓷  
 .count(); ⓸

⓵ Création d’un Stream à partir du tableau.

⓶ Filtrage des éléments vide.  
⓷ Suppression des doublons.  
⓸ Décompte des éléments restants.

Ce programme ne déclare que les opérations à effectuer. À aucun moment la façon d’exécuter ces traitements n’est définie : pas de boucle, pas de variable locale pour stocker les résultats temporaires, etc. Le Stream est maître des algorithmes à utiliser, de l’optimisation et même de l’ordonnancement des opérations. La documentation de chacune des méthodes du Stream précise les garanties qu’elles offrent, charge au développeur de les prendre en compte.

Stream signifie flux : c’est à dire qu’une fois qu’un Stream a été utilisé pour appliquer une succession d’opération et a permis d’obtenir un résultat, il n’est plus réutilisable.

L’exemple suivant ne fonctionnera donc pas :

String[] tableau = {"toto", "titi", "tata", "toto", ""};

Stream<String> stream = Stream.of(tableau);

long count = stream

.filter(item -> item.isBlank())

.distinct()

.count();

long totosNumber = stream  
.filter(str -> "toto".equalsIgnoreCase(str)) .count(); ⓵

⓵ Stream déjà épuisé!

* 1. Créer un Stream : générateurs de flux

Toute séquence d’éléments peut être transformée en Stream :

• Un tableau :

String[] helloWorld = {"Hello", "stream", "world", "!"};

Stream<String> helloStream = Arrays.stream(helloWorld);

Stream<String> otherHelloStream = Stream.of(helloWorld)

•Une Collection :

List<String> helloWorld = Arrays.asList("Hello", "stream", "world", "!");

Stream<String> helloStream = helloWorld.stream();

• Une suite numérique :

IntStream zeroToHundred = IntStream.range(0, 100);

DoubleStream squaresOfTwo = DoubleStream.iterate(2, i -> i < 1000000, i -> i \* 2);

• Un autre flux :

Stream<String> fewWords = Stream.<String>builder()

.add("words")

.add("to")

.add("add")

.build();

Stream<String> filesLines = Files.lines(Path.of("/c/documents/file-sample.txt"));

Stream<String> linesStartingWithAddedWords = Stream.concat(fewWords, filesLines);

• Ou même rien du tout :

Stream<Object> empty = Stream.empty();

* 1. Appliquer des opérations : méthodes intermédiaires

Les méthodes intermédiaires sont l’ensemble des opérations applicables à un Stream et qui renvoient un Stream.

Puisqu’elles renvoient un Stream, elles peuvent être chaînées pour appliquer plusieurs méthodes intermédiaires successivement.

Ces méthodes utilisent une approche builder, c’est-à-dire que leur invocation permet de créer le pipeline de traitement qui sera *ou pas* invoqué dans le futur.

❗️Les méthodes intermédiaires ne déclenchent aucune exécution. Elles ne font que configurer une future utilisation.

• distinct : ne conserve que les éléments non égaux du Stream initial.

Stream<Character> letters = Stream.of('a', 'b', 'j', 'z', 'b');

Stream<Character> distinctLetters = letters.distinct(); // contains only 'a', 'b', 'j', 'z'

• sorted : Elle prend en paramètre un Comparator et trie les éléments selon leur ordre naturel. Il est possible de préciser l’ordre dans lequel les éléments doivent être triés en fournissant un Comparator en entrée.

Stream<Character> letters = Stream.of('a', 'b', 'j', 'z', 'b');  
letters.sorted();  
// equivalent à  
letters.sorted((someLetter, someOtherLetter) -> someLetter.compareTo(someOtherLetter));   
// equivalent à   
letters.sorted(Comparator.naturalOrder());

• limit / skip : limit(x) tronque le Stream à x éléments, tandis que skip(x) retire du Stream les x premiers éléments.

• filter : Prend en paramètre un Predicate et retire du Stream tous les éléments ne le respectant pas.

Stream<String> namesContainingToto = Stream.of("Pierre-Toto", "Jean-Toto", null, "Tutu",

"Toto")

.filter(item -> Objects.nonNull(item))

.filter(item -> item.contains("Toto"));

• map : Prend en paramètre une Function et l’applique sur chacun des éléments du Stream. Permet de passer d’un Stream<K> à un Stream<V>.

PairOfSocks[] socks = {

new PairOfSocks("blanc", 38),

new PairOfSocks("bordeaux", 42),

new PairOfSocks("bleu", 39)

};

Stream<PairOfSocks> socksStream = Arrays.stream(socks);

Stream<Integer> socksSizes = Arrays.stream(socks).map(pair -> pair.size);

Stream<String> socksColors = Arrays.stream(socks).map(pair -> pair.color);

Stream<PairOfSocks> biggerSocks = Arrays.stream(socks).map(pair -> new PairOfSocks(pair.

color, pair.size + 1));

mapToInt, mapToDouble et mapToLong sont des spécialisations de map qui imposent que le Stream résultant de l’application du map soit respectivement un IntStream, DoubleStream, LongStream. Ces Streams permettent de manipuler les types primitifs int, double et long.

• flatMap : transforme chaque élément du Stream en un autre Stream via la fonction passée en paramètre et retourne la concaténation de chacun des Streams obtenus.

Stream<String> names = Stream.of("Pierre-Toto", "Jean-Toto", "Tutu", "Toto");

Stream<String> letters = names.flatMap(name -> Arrays.stream(name.split(""))); // splits into

letters

Le Stream d’origine n’a donc pas forcément la même cardinalité que le Stream résultant du flatmap.

• peek : Il permet de "jeter un coup d’oeil" sur les éléments du Stream, sans les transformer, et sans être final non plus. Dans la grande majorité des cas, si vous apercevez un peek, fuyez. La doc indique clairement qu’il s’agit d’une méthode dédiée au debug. Par ailleurs le peex ne fonctionne que si une opération intermédiaire ou l’opération terminal a besoin de parcourir l’intégralité des éléments.

List<String> l = Arrays.asList("A", "B", "C", "D");  
long count = l.stream().peek(item -> System.out.println(item)).count();⓵

⓵ Le count n’a pas besoin de parcourir la liste donc pas de parcours de liste, et donc le peek n’est pas déclenché

* 1. Obtenir un résultat : méthodes terminales

Les méthodes terminales sont les méthodes de l’API Stream qui ne renvoient pas un Stream : elles terminent donc les enchaînements d’opérations sur un Stream.

❗️Les méthodes terminales déclenchent l’exécution des traitements configurés via les méthodes intermédiaires, provoquant ainsi l’épuisement du Stream.

* + 1. Dénombrer

• count :

◦ dénombre les éléments présents dans le Stream.

◦ type de retour : long

PairOfSocks[] socks = {

new PairOfSocks("blanc", 38),

new PairOfSocks("bordeaux", 42),

new PairOfSocks("bleu", 39)

};

long socksUnderSize40 = Stream.of(socks)

.filter(sock -> sock.size < 40)

.count(); // returns 2

* + 1. Réduire

Les méthodes de réduction permettent de passer d’un Stream à un unique résultat.

* allMatch :
  + renvoie vrai si tous les éléments du Stream respectent le prédicat fourni en paramètre. Faux sinon.
  + type de retour : boolean
* anyMatch :
  + renvoie vrai si au moins un élément du Stream respecte le prédicat fourni en paramètre. Faux sinon.
  + type de retour : boolean

PairOfSocks[] socks = {

new PairOfSocks("blanc", 38),

new PairOfSocks("bordeaux", 42),

new PairOfSocks("bleu", 39)

};

boolean existsSockSizeOver38 = Stream.of(socks)

.anyMatch(sock -> sock.size > 38); // returns true

* noneMatch :
  + renvoie vrai si aucun élément du Stream ne respecte le prédicat fourni en paramètre. Faux sinon.
  + type de retour : boolean
* reduce :
  + renvoie un objet qui résulte de l’accumulation de tous les éléments du Stream via le BiOperator donné en entrée d
  + type de retour : boolean

PairOfSocks[] socks = {

new PairOfSocks("blanc", 38),

new PairOfSocks("bordeaux", 42),

new PairOfSocks("bleu", 39)

};

PairOfSocks neutralSock = new PairOfSocks("gris", 40);

BinaryOperator<PairOfSocks> makePatchworkSocks = (someSocks, someOtherSocks) -> new

PairOfSocks(someSocks.color + "," + someOtherSocks.color, someSocks.size);

PairOfSocks patchworkSocks = Arrays.stream(socks).reduce(neutralSock, makePatchworkSocks); // returns a PairOfSocks("gris,blanc,bordeaux,bleu", 40)

* collect : rassemble tous les éléments du Stream dans un nouvel objet, tel qu’une List, une Map ou une String par exemple.
  + prend en paramètre un collecteur de type Collector. Collectors propose implémentations usuelles telles que tolist().

PairOfSocks[] socks = {

new PairOfSocks("blanc", 38),

new PairOfSocks("bordeaux", 42),

new PairOfSocks("bleu", 39)

};

List<String> availableColors = Stream.of(socks)  
 .map(sock -> sock.color) // only keep colors  
 .collect(Collectors.toList()); // make a list of them  
Map<String,Integer> availableSizesByColor = Stream.of(socks)  
 .collect(Collectors.toMap(  
 sock -> sock.color, // take sock colors as keys  
 sock -> sock.size // take sock sizes as values  
)); // Only works if there are no color duplicates

* + 1. Rechercher

Les méthodes de recherche d’éléments renvoient des Optional<T>. Un Optional encapsule un objet java (ou du vide) et fournit une API qui permet mettre en place des traitements ne dépendants pas de la nullité de l’objet, sans mettre en place de structure conditionnelle.

Pour un stream de type Stream<T> :

* findAny :
  + renvoie n’importe quel élément du Stream.
  + type de retour : Optional<T>
* findFirst:
  + renvoie le premier élément du Stream
  + type de retour : Optional<T>
* max:
  + renvoie l’élément le plus grand du Stream, selon le comparateur passé en paramètre
  + type de retour : Optional<T>
* min:
  + renvoie l’élément le plus petit du Stream, selon le comparateur passé en paramètre
  + type de retour : Optional<T>
  1. Découpage

Rappelons que les méthodes intermédiaires ne font effectivement aucun traitement. Si l’on doit effectuer de nombreuses transformations, il peut être plus lisible de scinder le code :

Stream<PairOfSocks> socksStream = socks.stream();

Stream<PairOfSocks> usedSocksStream = socksStream

.filter(item -> item.used);

Stream<PairOfSocks> smallUsedSocksStream = usedSocksStream

.filter(item -> item.size < 35);

Map<Integer, List<PairOfSocks>> pairOfSocksBySize = smallUsedSocksStream

.collect(Collectors.groupingBy(item -> item.size));

1. OUI MAIS ALORS ON A PERDU LES NOMS ?

Comme chacun le sait, après le choix entre espace et tabulation, un nommage correct reste souvent un des meilleurs moyens d’avoir du code lisible.

Or a priori les merveilleuses lambdas nous ont fait perdre nos noms !

* 1. Référence de méthode

Imaginons que nous ayons une liste de chaînes de caractères myList de type List<String>, dont nous souhaitons afficher chacun des éléments sur la sortie standard. L’implémentation naïve serait la suivante :

for (String element : myList) {

System.out.println(element);

}

Mais maintenant que nous connaissons les lambdas, passons à une version plus concise :

myList.forEach(element -> System.out.println(element));  
// pour chaque élément que l'on appelera "element" de myList,  
// appliquer la méthode System.out.println avec comme paramètre d'entrée "element"

La méthode forEach applique à chaque élément de la liste le Consumer<String> fourni en entrée : fonction qui prend une String en entrée et ne renvoie rien.

On pourrait amplement se satisfaire de cette version. Mais poussons encore légèrement le curseur de la concision. En effet, le forEach itère sur une simple liste de chaînes de caractères : le Consumer prendra forcément un élément de cette liste en paramètre d’entrée. Seule la méthode à appliquer aux éléments nous donne de l’information :

myList.forEach(System.out::println); // à chaque élément de myList, appliquer la méthode println issue de la classe System.out

On vient alors d’utiliser une référence de méthode. De manière générale, les références de méthodes s’effectuent ainsi :

<nom de la classe ou de l'instance>::<nom de la méthode>

On peut les utiliser si notre lambda comporte pour seule instruction un appel de méthode et une seule variable.

myList.forEach(item -> item.toString()); // équivaut à :   
myList.forEach(String::toString);

myList.forEach(element -> System.out.println(element)); // équivaut à : myList.forEach(System.out::println);

* 1. Lambdas nommées

Imaginons désormais que nous souhaitions concaténer notre élément à une autre chaîne de caractères lors de l’affichage :

myList.forEach(element -> System.out.println("This is one element of my list: " + element));

Il n’est pas possible de faire référence à la méthode *println*, puisque son paramètre d’entrée ne peut plus être implicite.

En revanche, si nous avons plusieurs listes sur lesquelles appliquer ce traitement, il est possible de mutualiser la déclaration de notre expression lambda. Il faudra pour cela, créer une variable correctement typée et l’initialiser avec une lambda.

myList.forEach(element -> System.out.println("This is one element of my list: " + element));

theirList.forEach(element -> System.out.println("This is one element of my list: " + element));

peut devenir :

Consumer<String> elementPrinter = element -> System.out.println("This is one element of my

list: " + element);

myList.forEach(elementPrinter);

theirList.forEach(elementPrinter);

On a pu extraire l’expression lambda dans une variable de type Consumer<String> et ainsi la nommer et la réutiliser.

Une version qu’on voit moins souvent -et c’est bien dommage- nous permet d’utiliser les méthodes de sa propre classe.

Je peux écrire :

public void printElements(List<String> myList, List<String> theirList) {

myList.forEach(item -> this.printElement(item));

theirList.forEach(item -> this.printElement(item));  
}

private void printElement(String element) {

System.out.println("This is one element of my list: " + element);  
}

Or comme on l’a vu, si la lambda contient pour seule instruction un appel de méthode, on peut utiliser les méthodes références :

public void printElements(List<String> myList, List<String> theirList) {

myList.forEach(this::printElement);

theirList.forEach(this::printElement);  
}

private void printElement(String element) {

System.out.println("This is one element of my list: " + element);  
}

1. CAS REEL
   1. Cas 1

Ainsi lors d’une revue, nous sommes tombés sur le code suivant :

public Map<String, List<Record>> getMap(List<Record> records, boolean sortByAnalysis) {  
 Map<String, List<Record>> groupedRecords = new HashMap<>();⓵  
 for (Record record : records) { ⓶   
 String keyword = sortByAnalysis ? record.getAnalysisName() : record.getJobId(); ⓷   
 if (!groupedRecords.containsKey(keyword)) { ⓸   
 groupedRecords.put(keyword, new ArrayList<>()); ⓹   
 }   
 groupedRecords.get(keyword).add(record); ⓺  
 }  
 return groupedRecords;  
}

⓵ On instancie la map que l’on va retourner   
⓶ On boucle sur la liste  
⓷ On extraie la clef de regroupement  
⓸ Si la map ne contient pas encore la clef   
⓹ On l’ajoute avec une liste vide   
⓺ On ajoute l’élément à la liste présente à cette clef

En utilisant les apis à notre disposition, nous avons effectué la réécriture suivante :

public Map<String, List<Record>> getMap(List<Record> records, boolean sortByAnalysis) {

Function<Record, String> classifier = sortByAnalysis ? Record::getAnalysisName : Record::getJobId; ⓵  
 return records.stream().collect(Collectors.groupingBy(classifier)); ⓶   
}

⓵ On initialise le classifier  
⓶ On regarde la plateforme travailler.

* 1. Cas 2

Regardons un autre exemple de code également réel, dont le jargon métier a été modifié :

public boolean hasRedSocks(Home home) {

return home.getRooms().stream()

.filter(room -> room.getName().equals("bedroom"))

.findAny()

.flatMap(room -> room.getFurnitures().stream()

.filter(furniture -> furniture.getName().equals("sock drawer"))

.findAny().flatMap(furniture -> furniture.getClothes().stream()

.filter(clothe -> clothe instanceof Sock)

.map(clothe -> ((Sock) clothe).getMaterial())

.filter(material -> material instanceof Cotton && ((Cotton) material).getColor().equals("red"))  
 .findAny()  
 .map(o -> true)))

.orElse(false) ;

Alors, sceptique ? Pourtant tout cela nous semble très clair ! En effet, l’api Stream utilisée à mauvais escient devient tout à fait indigeste. Considérez les trois réécritures suivantes :

* + 1. Mieux avec lambda

public boolean hasRedSock1(Home home) { ⓵   
 return home.getRooms()   
 .stream()  
 .anyMatch(room -> isBedroom(room) && containsRedSock1(room));  
}

private boolean containsRedSock1(Room room) {

return room.getFurnitures()

.stream()

.anyMatch(furniture -> isSockDrawer(furniture) && containsRedSock1(furniture));  
}

private boolean containsRedSock1(Furniture furniture) {

return furniture.getClothes()

.stream()

.anyMatch(this::isARedSock);  
}

* + 1. Mieux sans lambda

public boolean hasRedSock2(Home home) { ⓶   
 for (Room room : home.getRooms()) {   
 if (isBedroom(room) && containsRedSock2(room)) {  
 return true;  
 }  
 }  
 return false;  
}

private boolean containsRedSock2(Room room) {

for (Furniture furniture : room.getFurnitures()) {

if (isSockDrawer(furniture) && containsRedSock2(furniture)) {

return true;  
 }  
 }

return false;

}

private boolean containsRedSock2(Furniture furniture) {

for (Clothe clothe : furniture.getClothes()) {

if (isARedSock(clothe)) {

return true;  
 }  
 }

return false;  
}

* + 1. Mieux legacy

public boolean hasRedSock3(Home home) { ⓷   
 Stream<Room> bedRooms = home.getRooms().stream()   
 .filter(this::isBedroom); // Retains only bedrooms  
 Stream<Furniture> sockDrawers = bedRooms  
 .map(Room::getFurnitures) // Gets the list of furnitures  
 .flatMap(List::stream) // Turns them to stream  
 .filter(this::isSockDrawer); // Filter on sock drawer  
 Stream<Clothe> clothes = sockDrawers  
 .map(Furniture::getClothes) // Get clothes  
 .flatMap(List::stream);  
 return clothes.anyMatch(this::isARedSock); // Has at least one red sock  
}

Les réécritures ⓵ et ⓷ utilisent l’api Stream tandis que la ⓶ utilise des boucles *for* et conditions basiques du langage.

Bien que moins concise, la réécriture ⓶ n’en est pas moins lisible que les autres.

Le gain en lisibilité obtenu via ces réécritures ne découle que de la réorganisation et de l’extraction de la logique métier dans des méthodes simples, et non pas de l’utilisation ou non des fonctionalités de Java 8.

Les nouveautés ne sont pas forcément mieux, ni forcément mauvaises. Et c’est la force d’un bon développeur que de savoir quand et où les utiliser.

1. CONCLUSION

Avec c’est article qui ne fait que gratter la surface, nous esperons avant tout donner des clefs pour comprendre les mécanismes à l’œuvre dans les streams, et les APIs fonctionnelles.

Nous pourrions encore explorer la programmation fonctionnelle (ou ce qu’il est possible de faire depuis Java8), nous pencher sur les exécutions de stream en parallèle, ou sur les optimisations faites par le compilateur java.

Mais avant tout ce qui nous intéresse ici, c’est de lever le voile sur la *magie noire* et de vous donner envie d’aller creuser vous mêmes les merveilleuses arcanes suprêmes de la connaissance du java.

Bon voyage.  
Lucile Thienot & Jérôme Tama