Chapitre 6

Les Streams

L'objectif de ce cours est de découvrir les **Streams** et des **Lambda Expression** qui sont une nouveauté de la version Java 8.0.

1. Les lambda expression

1.1. Introduction

Les "lambda expression" est un terme qui a été créé lors de l'apparition des langages fonctionnels dans les années 70 comme lisp et plus récemment scheme.

Dans un langage fonctionnel et interprété, la "lambda expression" est le moyen naturel de créer, à tout moment du programme, un traitement (ou fonction) pour pouvoir ensuite exécuter ce traitement.
Ces traitements sont nommés ou anonyme.

Exemple :

cas d'une lambda expression anonyme :
(setq x ((lambda (x y) (* x y)) 20 30)) ; x a la valeur 600

cas d'une lambda expression nommée
(defun func1 (lambda (x y) (* x y)))
(func1 20 30) ;retourne 600

(defun func2 (lambda (x y) (+ x y)))

(func 20 30) ;retourne 50

Ainsi, il est facile de passer en paramètre d'un traitement une lambda expression.

(defun generique (lambda f g x y) (f x y) / (g x y)))

(generique func1 func2 20 30) ; retourne 12 (600/50)

On appelle cela **écrire du code générique** : passer en paramètre d'un traitement un autre traitement.

On voit ici que la lamba expression est composée de deux parties :
- les paramètres formels
- un corps d'instruction.

Ainsi la lambda expression est tout simplement le moyen de créer des sous-programmes (ou fonction, procédure) au sein d'un langage fonctionnel.

Les langages objets qui sont issus des langages procéduraux (non fonctionnels et ne connaissant pas les lambda expression) ne contenaient pas non plus les lambda expression.
Or grâce à l'héritage (et la notion de classe abstraite), dans un langage objet, il est possible de créer des traitements génériques. Il suffit pour cela de passer en paramètre d'une méthode M de type T abstrait, un objet de sous-type réel de T' et d'utiliser dans la méthode M une méthode de T qui a été surchargé par T'.

Par contre, la contrainte est que le traitement que l'on veut passer en paramètre soit encapsulé dans un objet qui, lui est passé en paramètre.

C'est cette contrainte que les lambda expression permettent de lever. Rendant ainsi une écriture du code plus facile et plus courte.

1.2. Les classes anonymes

Ainsi, en JAVA, passer un traitement en paramètre consiste, à passer un objet. Et qui dit objet dit classe. Il faut donc créer une classe.

Cela pouvait donc devenir assez lourd d'où la possibilité de "créer" des classes anonyme.

Remarque : Ce n'est pas nous qui créons une classe anonyme. Nous on crée un objet à partir d'une classe (ou interface) abstraite et c'est la JVM qui crée, une classe qui est donc anonyme (génération d'un fichier xxxx$.class), classe qui hérite de la classe abstraite (ou implémente l'interface).

Exemple sur le tri d'une collection :

```
Personne p1 = new Personne("LAFONT","Paul",35);
Personne p2 = new Personne("DUPONT","Andre",50);
Personne p3 = new Personne("ZOERIA","Sylvie",25);
Personne p4 = new Personne("DUPONT","Paul",20);

ArrayList<Personne> l = new ArrayList<Personne>();
l.add(p1);
l.add(p2);
l.add(p3);
l.add(p4);

Collections.sort(l,new Comparator<Personne>()
                { public int compare(Personne p1,
                                     Personne p2)
                    {
                       return p1.getNom().compareTo(p2.getNom());
                     }});
for(Personne p:l)System.out.println(p.toString());
```

L'objectif est donc d'aller encore plus loin dans la simplification du code et donc de passer, des classes anonymes au lambda expression.

Pour preuve :

```
Collections.sort(l,(e1, e2)->(e1.getNom().compareTo(e2.getNom())));
for(Personne p:l)System.out.println(p.toString());
```

1.3. La lambda expression en JAVA

La syntaxe de l'écriture d'une lambda expression est

```
(paramètres) -> expression

(paramètres) -> { instruction1; instruction2; ….. return expression;}
```

```
Où paramètres est une liste de paramètres espacés par des virgules
Les paramètres sont typés ou non (facultatif).
```

En JAVA, une lambda expression est un objet qui est une instance d'une class anonyme qui implémente une interface contenant une seule méthode abstraite

Exemple :

```
interface Func2Int { public int f(int x,int y); }
```

```
public static void generique(Func2Int func1,Func2Int func2,int x,int y)
    {
        System.out.println(func1.f(x,y) / func2.f(x,y));
    }
```

```
Func2Int f1 = (x,y)->(x * y);
Func2Int f2 = (x,y)->(x + y);

generique(f1,f2,20,30);  // Affiche 12
```

Syntaxe simplifiée d'une lambda expression :

Pour pouvoir rendre encore plus simple le code, il existe la syntaxe du **::** (double deux-points)

```
Classe :: Methode
```

Avec les conditions suivantes :
- *Classe :: Methode* est passé en paramètre d'une interface I
- L'interface I contient une méthode abstraite dont la signature est compatible avec la méthode *Methode* définie dans la classe *Classe*.

(Voir l'exemple ci-dessous pour cette notation)

1.4. Exemple

Les exemples précédents sont codés dans l'exemple 36.

Voir sur le site http://jacques.laforgue.free.fr cours NFA 032
l'exemple **Exemple36_LambdaExpressionSyntaxe**

Commentaire en cours

1.5. Exemple de calcul générique

On se propose de faire le code générique qui permet de calculer la convergence d'une fonction mathématique :

Xn-1 = f(Xn)
Avec |Xn-Xn-1| tend vers 0

On applique notre code générique sur le problème de calcul de la racine carrée qui se calcule suivant la méthode dite de Newton.
Pour plus d'information voire l'exercice de NFA 031 : Exercice05_RacineCarre.
Rappel de la méthode de Newton :
calculer la racine carrée de a consiste à réaliser la fonction:

$x_{k+1} = ½(x_k+a/x_k)$ avec $x_0 = 1$

Voir sur le site http://jacques.laforgue.free.fr cours NFA 032
l'exemple **Exemple37_LambdaExpressionSerie**

Commentaire en cours

1.6. Conclusion

L'utilisation des lambda expressions est donc un vrai progrès pour la simplification du code.
Certains pourraient dire que l'on perd en lisibilité, mais cela est subjectif.

Comme le montre l'exemple 36, l'utilisation des lambda expressions associées aux Stream permet une utilisation des collections radicalement différentes.
Le modèle map/filter/reduce permet un nouveau style de programmation (voir plus loin).

Beaucoup de classe de l'API JAVA, depuis la version 8, sont impactées et ont donc été enrichies pour pouvoir appliquer les lambda expressions.

La conséquence est l'injection de la notion de méthode par défaut (**default**) dans les interfaces, permettant ainsi de ramener une interface composée de plusieurs méthodes à une seule méthode abstraite et donc son utilisation à travers les lambda expressions.

2. Les Streams

Le package : java.util.stream

Dans ce package on retrouve notamment :

- les interfaces
  - BaseStream : iterator parallel
  - Stream : filter map reduce distinct concat count findFirst forEach sorted …
  - IntStream LongStream DoubleStream
- les classes
  - Collectors

(source http://blog.ippon.fr/2014/03/17/api-stream-une-nouvelle-facon-de-gerer-les-collections-en-java-8/)

2.1. Introduction

Jusqu'à présent, effectuer des traitements sur des Collections ou des tableaux en Java passait essentiellement par l'utilisation du pattern Iterator.

Java 8 nous propose l'API Stream pour simplifier ces traitements en introduisant un nouvel objet, Stream.

Un stream se construit à partir d'une source de données (une collection, un tableau ou des sources I/O par exemple), et possède un certain nombre de propriétés spécifiques :

- Un stream ne stocke pas de données, contrairement à une collection. Il se contente de les transférer d'une source vers une suite d'opérations.
- Un stream ne modifie pas les données de la source sur laquelle il est construit. S'il doit modifier des données pour les réutiliser, il va construire un nouveau stream à partir du stream initial. Ce point est très important pour garder une cohérence lors de la parallélisation du traitement.
- Le chargement des données pour des opérations sur un stream s'effectue de façon lazy. Cela permet d'optimiser les performances de nos applications. Par exemple, si l'on recherche dans un stream de chaînes de caractères une chaîne correspondant à un certain pattern, cela nous permettra de ne charger que les éléments nécessaires pour trouver une chaîne qui conviendrait, et le reste des données n'aura alors pas à être chargé.
- Un stream peut ne pas être borné, contrairement aux collections. Il faudra cependant veiller à ce que nos opérations se terminent en un temps fini – par exemple avec des méthodes comme limit(n) ou findFirst( ).
- Enfin, un stream n'est pas réutilisable. Une fois qu'il a été parcouru, si l'on veut réutiliser les données de la source sur laquelle il avait été construit, nous serons obligés de reconstruire un nouveau stream sur cette même source.

Il existe deux types d'opérations que l'on peut effectuer sur un stream : les opérations intermédiaires et les opérations terminales.

Les opérations intermédiaires (Stream.map ou Stream.filter par exemple) sont effectuées de façon lazy et renvoient un nouveau stream, ce qui crée une succession de streams que l'on appelle stream pipelines. Tant qu'aucune opération terminale n'aura été appelée sur un stream pipelines, les opérations intermédiaires ne seront pas réellement effectuées.

Quand une opération terminale sera appelée (Stream.reduce ou Stream.collect par exemple), on va alors traverser tous les streams créés par les opérations intermédiaires, appliquer les différentes opérations aux données puis ajouter l'opération terminale. Dès lors, tous les streams seront dit consommés, ils seront détruits et ne pourront plus être utilisés.

2.2. Création d'un stream

On peut créer un stream de plusieurs façons. La plus simple consiste à appeler la méthode stream() ou parallelStream() sur une collection, mais un certain nombre de méthodes ont été ajoutées aux classes déjà existantes.

Notons ainsi la méthode chars() de la classe String, qui renvoie un IntStream construit sur les différents caractères de la chaîne de caractères, ou encore la méthode lines() de la classe BufferedReader qui crée un stream de chaînes de caractères à partir des lignes du fichier ouvert. À la classe Random s'ajoute aussi une méthode intéressante, ints(), qui renvoie un stream d'entiers pseudo aléatoires.

L'API propose également des méthodes statiques au sein de la classe Stream. Par exemple, le code suivant : "Stream.iterate(1, x -> x*2)" renverra un stream infini d'entiers contenant la suite des puissances de 2. Le premier argument contient la valeur initiale du stream, et le deuxième la fonction permettant de passer de l'élément n à l'élément n+1 dans le stream.

2.3. Parallélisation

L'un des points forts de cette nouvelle API est de nous permettre de paralléliser nos traitements de façon particulièrement aisée. En effet, n'importe quel stream peut être parallélisé en appelant sa méthode parallel() héritée de l'interface BaseStream – de la même façon, un stream peut être rendu séquentiel en invoquant la méthode sequential(). On peut également construire un stream parallèle sur une collection directement en appelant la méthode parallelStream() sur cette collection.

Ces méthodes nous permettent de masquer la répartition du travail, mais ne doivent pas être prises à la légère : en essayant de gagner en performance en parallélisant n'importe quel traitement, on prend le risque de produire l'effet inverse (nous y reviendrons plus tard).

2.4. Opérations intermédiaires

Les opérations intermédiaires peuvent être stateful ou stateless. Les opérations stateless effectuent un traitement sur les éléments du stream un à un sans avoir à prendre en compte les autres éléments du stream.

```
List<Commande> mesCommandes = … ;

List<Client> mesClients = mesCommandes.stream()
     .map( c -> c.getClient() )
     .collect( Collectors.toList() );
```

collect permet ici simplement de stocker le résultat dans une liste

Les opérations stateful quant à elles, ont généralement besoin de connaître l'ensemble du stream pour donner un résultat (par exemple Stream.distinct ou Stream.sorted). Par conséquent, paralléliser un tel traitement risque bien souvent de baisser nos performances au lieu de les améliorer.

```
List<Commande> mesCommandes = … ;
```

```
List<Client> mesClients = mesCommandes.stream()
     .map( c -> c.getClient() )
     .distinct()
     .collect( Collectors.toList() );
```

On renvoie la liste de nos clients, sans doublons, grâce à l'opération intermédiaire stateful distinct()

2.5. Opérations terminales

Nous disposons de deux types de réductions dans l'API Stream. Les opérations de réductions simples et les réductions mutables.

Les réductions simples sont celles auxquelles on pourrait penser en premier lieu : La somme d'éléments (Stream.sum), le maximum (Stream.max), ou le nombre d'éléments (Stream.count) sont des réductions simples. Dans sa forme générale, elle se défini de la façon suivante :

```
<U> U reduce(U identity,

     BiFunction<U, ? super T, U> accumulator,

     BinaryOperator<U> combiner);
```

L'élément identité est l'élément initial pour la réduction (et l'élément renvoyé si le stream est vide). L'accumulator crée un nouveau résultat partiel à partir d'un résultat partiel et d'un nouvel élément, et le combiner crée un nouveau résultat partiel à partir de deux résultats partiels.

Deux points sont à noter dans cette méthode :

- Tout d'abord, l'identité doit être une identité au sens mathématique du terme pour la fonction combiner : combiner.apply(u, identity) doit être égal à u quel que soit u.
- La fonction combiner doit être associative. Cela est une nécessité pour ne pas obtenir de résultat aléatoire lors d'une parallélisation du travail.
-

La méthode sum peut donc être réécrite en utilisant la méthode reduce :

```
List<Commande> mesCommandes = …;

int chiffreAffaire = mesCommandes.stream()
     .reduce( 0,
          (result, commande) -> result + commande.getPrice(),
          (resultA, resultB) -> resultA + resultB );
```

On peut réécrire la méthode sum() en utilisant la méthode reduce

Les réductions mutables généralisent ce concept en accumulant les éléments d'un stream dans un container. Ce dernier peut être une Collection, un StringBuilder, ou même un simple entier (auquel cas nous aurions affaire à une réduction simple).

```
<R> R collect(Supplier<R> supplier,

     BiConsumer<R, ? super T> accumulator,

     BiConsumer<R, R> combiner);
```

Nous retrouvons une syntaxe relativement similaire à la syntaxe de la méthode reduce. Cette fois-ci nous devons cependant initialiser un container (supplier), puis définir

la méthode accumulator qui ajoutera un élément à un container, et enfin la méthode combiner qui créera un nouveau container à partir de deux container temporaires.

Afin de simplifier notre code, l'API Stream nous propose également une autre classe, **Collectors**, qui encapsule les trois arguments nécessaire à une réduction pour certaines opérations classiques (récupération des données dans une liste, une map ou un set, concaténer des chaînes de caractères…). On pourrait par exemple modifier notre code précédent pour obtenir le même résultat :

```
List<Commande> mesCommandes = …;

int chiffreAffaire = mesCommandes.stream()
     .collect( Collectors.summingInt( Commande::getPrice ) );
```

2.6. Exemple

Voir sur le site http://jacques.laforgue.free.fr cours NFA 032
l'exemple **Exemple38_LambdaExpressionEtStream**

Commentaire en cours

2.7. Conclusion

Cette nouvelle API fournie par le JDK 8 va donc modifier fondamentalement notre façon de traiter les Collections en nous proposant une alternative au pattern Iterator relativement lourd à mettre en place. Celle-ci tire profit de la nouvelle syntaxe des lambdas expressions pour réduire notre code au maximum tout en améliorant nos performances. De plus, la classe Collectors présentée succinctement ici nous offre de nombreux patterns qui remplaceront dans de nombreux cas le pattern Iterator.