Lambdas et Stream

Lambdas

Avant de commencer le cours, on va revenir sur un concept de Java 8 qui n'a qu'un rapport lointain avec les lambdas, mais qui est important ne fut-ce que parce que la majorité des tutoriels y font référence sans bien expliquer ce que c'est.

Methods par défaut des interfaces

Jusqu'à présent on avait vu que les interfaces étaient des contrats, donc des définitions de fonctions que des classes étaient chargées d'implémenter.

En Java 8, on s'est dit qu'il serait sympa que les interfaces définissent des fonctions communes à toutes les classes susceptibles de les implémenter.

Imaginons que l'interface IMachin déclare une fonction getRandom() qui renvoie un nombre aléatoire. Si on le faisait de manière classique il faudrait que chaque classe implémentant IMachin fournisse une fonction getRandom().

Maintenant il est possible de déclarer cette fonction dans l'interface :

```
public interface IMachin {
	public default int getRandom() {
		return new Random().nextInt(10);
	}
}
```

Si on implémente une classe de IMachin, disons Machin, la fonction sera automatiquement implementée :

int a = new Machin().getRandom() ;

A noter qu'on peut surcharger la fonction dans une classe implémentant l'interface en question.

Exercice : Créer IMachin, Machin et une autre classe qui surcharge la method default. Vérifier que ça marche.

Interface fonctionnelle

Une interface fonctionnelle (IF) est une interface qui ne possède qu'une seule method et qui est annotée avec `@FunctionalInterface`

```
@FunctionalInterface
public interface Truc {
public void myFunc() ;
}
```

A noter qu'une IF peut posséder des method default en plus de son unique method.

Vu comme ça, cela ne présente pas grand intérêt. Si on voulait se servir de cette interface sans implémenter une classe, il faudrait passer par une classe anonyme et ce n'est pas forcément très simple.

```
Truc t = new Truc() {
	public void myFunc() {
		System.out.println("Coucou");
```

```
        }
};
t.myFunc() ;
```

Ici on afficherait *Coucou.*

C'est là qu'arrive les lambdas qui permettent d'écrire de façon beaucoup plus simple.

Soit l'interface fonctionnelle :

```
@FunctionalInterface
public interface Monologue {
        public void parle();
}
```

Si on veut utiliser cette interface sans créer de classe qui l'implémente, on peut passer par une lambda.

```
Monologue m = () -> System.out.println("Coucou");
m.parle();
```

C'est quand même plus simple qu'une classe anonyme.

La structure d'une lambda est la suivante :

- A gauche ce qui décrit la signature d'une méthode
    - () = pas de paramètre
    - (a) = un paramètre. Dans ce cas, on peut omettre les (). (a) -> est identique à a ->
    - (a,b) = 2 parametres
    - etc
- A droite le traitement

Comment se fait-il qu'on n'a besoin que de la signature de la method et pas de son nom ? Eh bien, il s'agit d'une IF, qui n'a donc qu'une method. Pas la peine de préciser son nom. C'est cela le fondement des lambdas : il s'agit de « donner » un traitement à une method définie dans une interface fonctionnelle. Et ce, de manière concise.

Pour ce qui est du traitement, trois choses :

- S'il n'y a qu'une ligne de traitement, pas la peine de mettre des {}. Mais il faut mettre un ; en fin de ligne
- S'il n'y a qu'une ligne, inutile de mettre un return (si la method renvoie qchose).
  `Truc t = () -> 1223` renvoie toujours 1223 ;
- Sinon, il faut mettre des {} (n'oubliez pas le ; après le }) :
  `Monologue m = () -> {System.out.println("Coucou "); System.out.println("toi");};`

Si on a des paramètres, on peut évidemment faire des traitement plus sophistiqués :

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

Exercice

Soit l'interface fonctionnelle :

```
@FunctionalInterface
public interface Parler {
        public void parler(String s);
```

```
}
```

Ecrivez une classe anonyme utilisant *Parler.* Puis écrivez une lambda.

Exercice

Soit l'interface fonctionnelle :

```
@FunctionalInterface
public interface Operation {
        public int opere(int a, int b);
}
```

Ecrivez 4 lambdas qui font la somme, la différence, le produit et la division des valeurs passées en paramètre.

Vous allez me dire : mais quel est l'intérêt ? Certes, ça permet une écriture plus concise, mais d'un autre côté, c'est pas très compréhensible (du moins au début).

Et bien … Dans d'autres langages, il y a une possibilité que n'offre pas le Java : passer une method en paramètre à une method. Croyez-moi sur parole, ça permet de faire des choses très puissantes.

Or, avec les interfaces fonctionnelles, on peut le faire : passer une IF en paramètre, c'est la même chose que de passer une method.

Imaginons (si Truc est une IF déclarant la method dummy(String))

```
Truc t = () -> System.out.println("Value is : ") ;
```

On peut déclarer une method aFunc comme ça.

```
public void aFunc(Truc truc) {
    t.dummy("une chaine") ;
}
```

et l'appel serait

```
aFunc(t) ;
```

Ou mieux

```
aFunc(a -> System.out.println("Value is : "+a)) ;
```

Mais quand même … L'intérêt est limité puisque pour chaque lambda il faut déclarer une IF.
C'est là qu'entre en jeux les IF fournies par Java (dans `java.util.function.*`).

Il n'y a plus à déclarer les IF puisqu'elles sont fournies. Ca donne une autre ampleur aux lambdas (et, comme on le verra, c'est le fondement des Streams).

Voyons les plus simples de ces IF « par défaut » :

- `java.util.function.Function<T,R>` : sa méthode fonctionnelle a la signature `R apply(T t)` . Elle permet donc de traiter un paramètre T et de renvoyer un type R.
- `java.util.function.Predicate<T>` : sa méthode `boolean test(T t)` permet, comme vous vous en doutez, de faire un test sur le paramètre et de retourner un `boolean` en fonction du résultat.
- `java.util.function.Consumer<T>` : Cette interface fonctionnelle est un peu particulière car c'est la seule qui a pour vocation de modifier les données qu'elle reçoit. Sa méthode fonctionnelle `void`

`accept(T t)` est faite pour appliquer des traitements au paramètre passer et ne retourne rien.

- `java.util.function.Supplier<T>` : Celle-ci permet de renvoyer un élément de type T sans prendre de paramètre via la méthode fonctionnelle `T get()`.

Prenons l'exemple de l'IF *Function.*

```
Function<Integer, Integer> carre = (a) -> a*a;
System.out.println(carre.apply(10));
```

La lambda renvoie donc le carré de ce qui lui est passé en paramètre. Ce qui est intéressant, en plus, avec ces IF toutes prêtes, c'est qu'elles ont des method default très intéressantes. *Function* possède par exemple *addThen()* qui permet d'enchainer des lambdas.

Si on ajoute

```
Function<Integer, Integer> foisDix = (a) -> a*10;
System.out.println(carre.addThen(foisDix).apply(10));
```

On obtiendra 1000 au final.

Exercices

- Avec une *Function*, faites une lambda renvoie un entier sous forme chaine.
- En ajoutant une autre *Function* et en utilisant un addThen(), faites en sorte qu'en partant d'un int N, on ait en sortie *La valeur est = N*
- Avec un *Predicate*, déterminer si un entier est pair.
- Créer une Class Person (nom et age comme champs) et modifiez le nom et l'age via un *Consumer*.
- Créer une lambda qui renvoie un nombre aléatoire entre 0 et 100 de type Integer avec un *Supplier*.

Référence de method

Dans une lambda, si on utilise une method (statique ou non) qui utilise le même nombre et le même type de paramètres déterminés pour la lambda (ou l'IF), on peut utiliser l'opérateur ::

```
Parler par = s -> System.out.println(s) ;
```

Est équivalent à

```
Parler par = System.out::println() ;
```

Puisque la method parler() n'accepte qu'un paramètre de type String, et println() aussi .

Autre exemple plus complexe. Soit la classe

```
public class Somme {
	public static int staticSomme(int a,int b) {
		return a+b;
	}
	public int somme(int a,int b) {
		return a+b;
	}
}
```

Si on se réfère à l'IF Operation, vu plus haut :

```
Somme som = new Somme();

Operation op = (a,b) -> Somme.staticSomme(a, b);
System.out.println(op.opere(10, 5));

op = (a,b) -> som.somme(a, b);
```

```
System.out.println(op.opere(10, 5));

op = Somme::staticSomme;
System.out.println(op.opere(10, 5));

op = som::somme;
System.out.println(op.opere(10, 5));
```

Les lignes surlignées sont équivalentes puisque opere() et somme() ont le même nombre et le même type de paramètre.

C'est un peu abstrait, certes, mais ça permet de coder encore plus concis.

Les Streams

Autant les Lambdas peuvent sembler nébuleuses et avoir un intérêt mal défini, autant les Streams qui les utilisent sont un des gros apports de Java 8.

Derrière ce nom se cache en fait un tout nouveau pattern de manipulation de données (remplaçant du pattern `Iterator` entre autre), qu'elles proviennent d'un tableau, d'une collection, d'un flux de fichier (avec `java.nio` par exemple), d'un flux réseau et j'en passe.
Avec les streams nous allons pouvoir parcourir, filtrer, manipuler, transformer nos données.

Les streams se trouvent dans le package `java.util.stream` et ils utilisent massivement les interfaces fonctionnelles (et donc les lambdas) pour appliquer des traitements.
On pourrait faire une analogie avec le langage SQL où vous pouvez spécifier des contraintes aux données que vous recherchez dans votre base, du genre « je veux tous les membres dont l'âge est inférieur à 18 ans et qui ont posté plus de 50 messages sur le forum » : vous pouvez faire ce genre de choses facilement avec les streams sans avoir à faire tout un tas de boucles, de contrôle de traitements...

Avant de commencer

Quelques informations sont toutefois nécessaires avant de jouer avec les streams. Les streams ressemblent beaucoup aux collections mais en divergent en plusieurs points :

- Un stream ne stocke aucune donnée, il se contente de les transférer vers une suite d'instructions;
- Un stream ne modifie pas les données qu'il reçoit de sa source (flux, collection, tableau, …).
- Un stream est à usage unique : une fois utilisé complètement, impossible de l'utiliser une seconde fois. Si nous devons réutiliser les données d'une source une seconde fois, nous devrons recréer un second stream.
- Les traitements fait sur un stream peuvent être de deux natures :
    - **Intermédiaire** : ce genre d'opération conserve le stream ouvert ce qui permet d'effectuer d'autre opérations dessus. Nous pourrons voir ceci lors de l'utilisation des méthodes map() ou filter() .
    - **Terminale** : c'est l'opération finale du stream, c'est ce qui lance la « consommation » du stream. La méthode **reduce()** en est un exemple.

Créer un stream

Imaginons qu'on crée une List de 100 entiers de 0 à 99.

Pour créer le Stream associé, on fait :

```
Stream<Integer> st = list.stream();
```

C'est tout ! Le Stream est créé et on peut commencer à faire des opérations dessus.

Parcourir un Stream

Avant, pour parcourir une liste, il fallait faire une boucle. Si on veut afficher la liste vue au-dessus, il fallait faire :

```
for (Integer i :liste) {
        System.out.println(i) ;
}
```

Avec un Stream, on fait :

```
st.forEach(s -> System.out.println(s));
```

ou mieux,

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

En pratique, *forEach* reçoit en paramètre un *Consumer*, c.a.d une IF qui ne reçoit un paramètre et ne retourne rien.

ATTENTION : si vous faites :

```
Stream<Integer> st = list.stream();
st.forEach(System.out::println);
st.forEach(System.out::println);
```

La 3e ligne va lever une Exception : après la seconde ligne, le Stream a été utilisé et est donc fermé. Il faut donc en rouvrir un.

Filtrage

C'est une fonction intermédiaire qui s'applique avant le *forEach*. Elle reçoit un *Predicate*, c.a.d quelque chose qui reçoit un paramètre et retourne true ou false ;

Si je veux filtrer et n'afficher que les éléments de la liste dont la valeur est > 50, ça donne :

```
st.filter(x -> x > 50).forEach(System.out::println);
```

On remarque que l'on enchaine les fonctions du Stream. C'est même un de ses intérêts : en une seule ligne, on peut faire des opérations complexes.

On peut évidemment chaîner les *filter()*

Map

C'est une opération très puissante mais un peu difficile à appréhender. Elle sert à ne garder qu'une partie de ce qui a servi à construire le Stream (la List dans notre cas).
Pour mieux comprendre, imaginons qu'on ait une List de Person et qu'on décide

- De filtrer pour ne garder que les gens dont l'âge est > 50
- Et au final de n'afficher qu'une liste de nom (et plus une liste de Person).

```
sp.filter(p -> p.getAge() > 50).map(p -> p.getNom()).forEach(System.out ::println);
```

Ici, le *map* décide à partir de Person, de ne renvoyer que le nom.

Collect

Jusqu'à présent, on en faisait que travailler dans le Stream sans rien pouvoir récupérer en sortie.

collect permet de récupérer le résultat des opérations successives sous une certaine forme. Cette forme est définie par un objet `Collectors` (implémentant l'interface `Collector`).

Reprenons l'exemple précédent : si on veut récupérer une liste après les opérations de filtre et de map, il faut faire :

```
List<String> lp2 = sp.filter(p -> p.getAge() > 50).map(p -> p.getNom()).collect(Collectors.toList());
```

A noter que le type de la List est automatiquement inféré de ce que renvoie le *map()*.

Count

Comme son nom l'indique, renvoie le nombre d'éléments dans le Stream.

```
sp.filter(p -> p.getAge() > 50).count();
```

Reduce

Là, c'est un peu tricky. Cette fonction permet d'agréger le contenu d'un Stream. De faire la somme de tous les éléments, d'en faire la moyenne, de déterminer le max, etc … En substance, de réduire une série de valeurs à une valeur unique (d'où son nom).

Si on veut faire la somme des poids de toutes les Person, on fait :

```
System.out.println("somme = " + sp.map(p -> p.getAge()).reduce((x,y) -> x+y).get());
```

Dans *reduce()*, x represente la valeur issue de l'opération de reduce (ici une somme) et y la nouvelle valeur à insérer dans l'opération de reduce.

De surcroit *reduce()* renvoie un *Optional* (comme avec les *Entity*) et il faut donc faire un *get()* à la fin. D'ailleurs si j'avais codé de manière vraiment propre, j'aurais du tester le retour avec un *isPresent()*.

Voilà, c'est la fin. Il existe d'autres fonctions et même celles présentées ont des fonctionnalités supplémentaires et je vous laisse les découvrir.

Un tutorial pas trop mal dont je me suis inspiré : https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java/5012411-creez-des-classes-anonymes-des-interfaces-fonctionnelles-des-lambdas-et-des-references-de-methode (lambdas) et https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java/5013326-manipulez-vos-donnees-avec-les-streams (streams)