Java 8
Introduction aux expressions lambda, stream et interfaces fonctionnelles

Les expressions lambda

Les expressions **lambda** sont apparues avec Java 8.
Les expressions **lambda** sont utilisées essentiellement avec les interfaces ne contenant qu'une seule méthode à définir.
Une interface ne contenant qu'une seule méthode abstraite est appelée une **interface fonctionnelle**.

Exemple

On déclare une interface fonctionnelle **I** avec la seule méthode **int operation(int x, int y)**

```
interface I{
	public int operation(int x, int y);
}
public class Main2 {

	public static void main(String[] args) {

		// objet i1 pour faire des additions
		I i1 = new I() {
			@Override
			public int operation(int x, int y) {
				// TODO Auto-generated method stub
				return x+y;
			}
		};
		System.out.println(i1.operation(10, 12));

		// objet i2 pour faire des soustractions
		I i2 = new I() {
			@Override
			public int operation(int x, int y) {
				// TODO Auto-generated method stub
				return x-y;
			}
		};
		System.out.println(i2.operation(8, 15));
	}
}
```

Résultat :
22
-7
La création de l'objet i1

```
I i1 = new I() {
                @Override
                public int operation(int x, int y) {
                      // TODO Auto-generated method stub
                      return x+y;
                }
          };
```

peut être remplacée par l'expression **lambda** suivante :

```
I i1 = (x,y)->x+y;
```

Le compilateur « sait » que I est une interface fonctionnelle, une seule méthode à définir qui reçoit 2 entiers, le compilateur « sait » que x et y doivent être des entiers, il retournera la somme des 2 paramètres x et y.

Une expression **lambda** est constituée de 3 parties :

Le compilateur déduit le type des paramètres x, y et le type retourné par inférence.

Le code complet de l'application est maintenant le suivant :

```
interface I{
      public int operation(int x, int y);
}
public class Main2 {

      public static void main(String[] args) {
            // objet i1 pour faire des additions
            I i1 = (x,y)->x+y;
            System.out.println(i1.operation(10, 12));

            // objet i2 pour faire des soustractions
            I i2 = (x,y)->x-y;
            System.out.println(i2.operation(12, 20));
      }
}
```

Quelques règles :
-le type des paramètres est optionnel : int param-> et param-> sont équivalents,
-les parenthèses sont optionnelles lorsqu'il y a 1 seul paramètre : (x)-> et x-> sont équivalents,
-les accolades sont nécessaires s'il y a plusieurs instructions à exécuter.

```
I i3 = (x,y)-> {
                System.out.println("1er opérande : "+x);
                System.out.println("2ème opérande: "+y);
                return x+y;
          };
```

Sans paramètre : **() -> System.out.println("Zero parameter lambda");**

1er exemple d'expression lambda avec la méthode forEach()

Soit une liste de noms créée de la façon suivante :

```
List<String> lesamis =Arrays.asList("Jean","Alain","Mouloud","Rémi","Brian","Mohamed",
"Pierre","Robert","Tydian","Michel","Rémas");
```

L'affichage de cette liste s'effectue avec la boucle for :

```
for (String amis : lesamis)
        System.out.print(amis+" ");
```

Les collections proposent la méthode **forEach()** qui accepte un paramètre de type **Consumer**.

L'interface **Consumer<T>** est fonctionnelle, la méthode **accept(T t)** est la seule à définir.
T symbolise le type des objets que la méthode accept() traitera, par exemple accept(Integer t).

Comme son nom l'indique, un objet **Consumer<T>** consomme (utilise puis passe au suivant) chaque élément de la collection en exécutant la méthode **accept(T t)** sur chacun des éléments.

Utilisation d'une classe anonyme :

```
lesamis.forEach(new Consumer<String>() {
        @Override
        public void accept(String t) {
        System.out.print(t+"  ");
        }
});
```

La méthode **accept()** est exécutée pour chaque élément de la liste.

L'interface **Consumer<T>** est fonctionnelle, la méthode **accept(T t)** est la seule à définir: on peut alors utiliser l'expression **Lambda**:

```
lesamis.forEach( (amis)->System.out.print(amis+"  ") );
```

ou

```
lesamis.forEach(amis->System.out.print(amis+"  ") );
```

Comment ça marche ?

Le compilateur "sait" que **lesamis.forEach()** reçoit comme argument un objet de type interface fonctionnelle **Consumer<T>**.
L'interface fonctionnelle ne propose obligatoirement qu'une seule méthode, ici la méthode **accept(T t)** qui reçoit un seul argument, identifié par amis dans l'expression Lambda.
Le compilateur implémente automatiquement le code de la méthode **accept(T t)** avec le code qui suit la flèche → , ici System.out.println(amis+ " ").
La liste est de type **String**, à chaque parcours de la boucle, l'élément "concerné" de la liste est affecté par déduction automatiquement à la variable *amis* de type String.

La méthode **forEach()** exécute la méthode de l'interface fonctionnelle pour chaque objet de la liste.

2ème exemple d'expression lambda avec la méthode forEach()

L'interface **Predicate** est fonctionnelle, elle propose une méthode abstraite **test(T t)** à définir.

Par exemple :
Predicate<String> predicate = s -> s.length() == 3;
La méthode **test()** implicitement définie ici retourne **true** si la taille du String traité est de **3**.

Utilisation :

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

public class Test {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Predicate<String> predicate = s -> s.length() == 3;
		List<String> liste=Arrays.asList("Jean","Luc","Pauline","Marc","Jan");
		//affichage de tous les prénoms à 3 lettres : 1ère solution
		for (int i=0;i<liste.size();i++) {
			if (predicate.test(liste.get(i)))
				System.out.println(liste.get(i));
		}
		//affichage de tous les prénoms à 3 lettres : 2ème solution
		Consumer<String> c = s-> {
				if (predicate.test(s))
					System.out.println(s);
		};

		System.out.println();
		liste.forEach(c);
	}
}
```

Affichage après exécution :
Luc
Jean

Luc
Jean

3ème exemple d'expression lambda

Génération de nombres aléatoires compris sans l'intervalle [0,10[en utilisant l'interface fonctionnelle **Supplier**.

L'interface fonctionnelle **Supplier** propose une méthode **get()** à définir.

```
import java.util.Random;
import java.util.function.Supplier;

public class Test {

	public static void main(String[] args) {
		Random random = new Random();
		Supplier<Integer> newRandom = () -> random.nextInt(10);

		for (int index = 0; index < 5; index++) {
		   System.out.println(newRandom.get() + " ");
		}
	}
}
```

4ème exemple d'expression lambda

Les collections proposent la méthode **removeIf()** qui accepte un paramètre de type **Predicat**.

L'interface **Predicat<T>** est fonctionnelle, la méthode **test(T t)** est la seule à définir.

On veut supprimer toutes les chaines de caractères d'une liste qui ont une longueur de 5 caractères.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Test {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		List<String> liste=Arrays.asList("Jean","Luc","Pauline","Marc","Jan");

		ArrayList<String> listeBis = new ArrayList<String>();

		listeBis.removeIf(s->s.length()==4);
		listeBis.forEach(s->System.out.println(s));
	}
}
```

Les stream

Soit une liste de noms créée de la façon suivante :

```
List<String> lesamis =Arrays.asList("Jean","Alain","Mouloud","Rémi""Brian","Mohamed",
"Pierre","Robert","Tydian","Michel","Rémas");
```

L'affichage de cette liste s'effectue avec la boucle forEach :

```
lesamis.forEach(amis->System.out.print(amis+"  ") );
```

Transformation de la liste en caractères majuscules

On se propose de créer une nouvelle liste qui contiendra les mêmes éléments que la précédente mais en lettres majuscules.

▷Création de la liste vide

```
List<String> lesamisEnMaj = new ArrayList<String>() ;
```

▷Remplissage de la nouvelle liste

```
for (String amis : lesamis)
        lesamisEnMaj.add(amis.toUpperCase());
```

▷On peut également utiliser la méthode **forEach()** avec une expression Lambda

```
lesamis.forEach(amis ->
        lesamisEnMaj.add(amis.toUpperCase()));
```

Utilisation des stream

Les stream sont un nouveau concept apparu avec Java 8.
L'API stream est désigné pour améliorer le traitement des données du point de vue des volumes, des temps de réponse et des algorithmes.

Un stream peut être vu comme un flux de données construit à partir d'une source de données (une collection, un tableau).

- Un **stream** est un objet obtenu à partir d'une source de données. Il peut être vu comme un flux de données séquentiel obtenu à partir d'une source de données (une collection, un tableau…).
- Un **stream** ne modifie pas les données de la source à partir de laquelle il est construit.
- Un **stream** ne stocke pas les données qu'il traite.
- Un **stream** peut ensuite subir des opérations comme la translation (mapping) et/ou le filtrage (filter), c'est à dire que chaque objet du flux initial subit l'opération choisie. Ces opérations reconstruisent généralement un autre **stream**.
- Un **stream** doit à la fin subir une opération terminale comme *forEach*, *reduce* ou *collect* pour exploiter le résultat des opérations. Le **stream** est dit alors «consommé».

Toutes les collections Java 8 possèdent une méthode stream() qui retourne un flux séquentiel contenant les objets de la collection. On peut ensuite appliquer des opérations de la classe Stream à ce flux, c'est à dire appliquer des opérations à chaque objet du flux créé par la méthode stream().

1er exemple:

```
List<String> lesamis =Arrays.asList("Jean","Alain","Mouloud","Rémi""Brian","Mohamed",
"Pierre","Robert","Tydian","Michel","Rémas");
```

```
List<String> lesamisEnMaj = lesamis.stream()        //1*)
        .map(nom -> nom.toUpperCase())              //2*)
        .collect(Collectors.toList());              //3*)
```

1*) La méthode **stream()** de la liste *lesamis* retourne un stream.
2*) A partir du flux précédent retourné par stream(), la méthode **map()** construit un nouveau stream où chaque élément est le résultat de l'application de la méthode indiquée. Ici chaque élément de la liste est mis en majuscule. La méthode map() reçoit comme argument un objet Function de type interface fonctionnelle où la seule méthode à définir est apply(T t), t étant l'objet sur lequel la méthode apply() est appliquée. On utilise ici une expression Lambda pour définir la méthode apply(). La méthode toUpperCase() convertit chaque nom de la liste en lettres majuscules.
3*) La méthode **collect()** récupére les données du stream créé par map() et enregistre les résultats dans la collection lesamisEnMaj. La méthode collect() est une opération terminale.

La méthode collect() de la classe Stream reçoit comme argument un objet Collector. Cet objet Collector est obtenu avec la méthode statique Collectors.toList() qui accumule les éléments du Stream dans une List, ici la liste *lesamisEnmaj*.

2ème exemple: affichage de chaque nom en majuscule (sans stockage)

```
lesamis.stream()
        .map(nom -> nom.toUpperCase())
        .forEach(amis->System.out.print(amis+" "));
```

La méthode forEach() est une opération terminale.

3ème exemple: comptage et affichage du nombre de lettres de chaque nom,

```
lesamis.stream()
        .map(nom -> nom.length())
        .forEach(taille -> System.out.print(taille+" "));
```

La méthode map crée un stream où chaque élément est le nombre de caractères du nom correspondant de la liste initiale.

4ème exemple: affichage de la liste triée par ordre alphabétique ;

```
lesamis.stream().sorted().forEach(nom->System.out.print(nom+" "));
```

La méthode sorted() trie par défaut selon l'ordre naturel.

5ème exemple: affichage de tous les noms qui commencent par M

```
lesamis.stream()
        .filter(nom->nom.startsWith("M"))
        .forEach(nom -> System.out.print(nom+" "));
System.out.println() ;
```

La méthode filter() reçoit comme argument un prédicat; elle retourne un Stream avec tous les

objets de la liste qui vérifie le prédicat.
Predicat est une interface fonctionnelle dont la méthode à définir est boolean test(T t). Cette méthode doit retourner true pour que le prédicat soit vrai, ici l'instruction nom.startsWith("M") ; qui retourne true si le nom commence par M.

6ème exemple: création d'une collection avec tous les noms qui commencent par M

```
List<String> debuteAvecM =
        lesamis.stream()
                .filter(nom->nom.startsWith("M"))
                .collect(Collectors.toList());
```

7ème exemple: recherche du plus grand nombre d'une liste

```
List<Integer> numbers = Arrays.asList(5, 10, 3, 8, 15, 7, 1, 4);

OptionalInt opt = numbers.stream().mapToInt(n->n).max();

System.out.println("MaxOpt = "+opt);
int maximum = opt.orElse(0);
System.out.println("Valeur max= "+maximum);
```

La méthode **mapToInt()** de la classe Stream reçoit comme argument une interface dont la seule méthode à définir doit retourner un entier.
La méthode **mapToInt()** retourne un stream d'entiers **IntStream** dont la méthode max() retourne un objet de type **OptionalInt**.
Cet objet **OptionalInt** contient la valeur maximale trouvée ou indique qu'il est vide si par exemple la liste numbers était vide.
L'instruction **int maximum = opt.orElse(0);** affecte à maximum soit la valeur maximale trouvée, soit 0 si la liste était vide.

Exemple avec une liste vide :

```
List<Integer> numbers1 = new ArrayList<Integer>();
OptionalInt opt = numbers1.stream().mapToInt(n->n).max();
System.out.println("MaxOpt = "+opt);
int maximum = opt.orElse(0);
System.out.println("Valeur max= "+maximum);
```

Affichage :

```
MaxOpt = OptionalInt.empty
Valeur max= 0
```

8ème exemple: moyenne d'une liste de nombres

```
OptionalDouble optMoy = numbers.stream().mapToDouble(n->n).average();

System.out.println("MaxOpt = "+optMoy);
double moyenne = optMoy.orElse(0);
System.out.println("Valeur moyenne = "+moyenne);
```

9ème exemple: tri d'une liste de nombres.

On crée ici une nouvelle liste de nombres triée par ordre croissant.

```
List<Integer> numbers = Arrays.asList(5, 10, 3, 8, 15, 7, 1, 4);

List<Integer> triee = numbers.stream().sorted().collect(Collectors.toList());

for (int i: triee)
System.out.print(i+" ");    System.out.println();
```

La classe Stream propose la méthode sorted() qui retourne un Stream constitué des éléments triés dans l'ordre naturel.
La méthode collect() de la classe Stream reçoit comme argument un objet Collector. Cet objet Collector est obtenu avec la méthode statique Collectors.toList() qui accumule les éléments du Stream dans une List, ici la liste *triee*.

11ème exemple : somme des nombres d'une liste

On fait appel ici à la méthode **reduce** de la classe Stream.

```
List<Integer> numbers = Arrays.asList(5, 10, 3, 8, 15, 7, 1, 4);

int somme = numbers.stream().reduce(0,(n1,n2)->n1+n2) ;

System.out.println("somme= "+somme);
```

La méthode **reduce()** réalise une réduction des éléments du stream.
Le 1er argument - 0 ici - est appelé ***identity***: c'est à la fois la valeur initiale et la valeur du résultat si le stream et vide.
Le 2ème argument est appelée la fonction *d'accumulation*: elle prend 2 paramètres: le résultat partiel et la prochaine valeur du flux. Elle retourne le résultat quand tout le flux a été traité.
La fonction d'accumulation est donnée sous la forme de l'expression lambda (n1,n2)->n1+n2 , cette fonction somme toutes les éléments du stream avec comme valeur initiale 0.

Les interfaces fonctionnelles

Une interface ne contenant qu'une seule méthode abstraite est appelée une **interface fonctionnelle**.

Une interface fonctionnelle peut contenir

- des méthodes **par défaut**, c'est-à-dire des méthodes notées **default** et déjà implémentées,
- des méthodes **static**.

```
interface I{
     public int operation(int x, int y);
     default void afficher(int x, int y) {
          System.out.println("1er opérande : "+x);
          System.out.println("2ème opérande: "+y);
     }
     default int f(int x ) { return x;}
```

```
	static int carre(int x) {
		return x*x;
	}
}
```

Dans le main :

```
	I i1b = (x,y)->x+y;
	System.out.println(i1b.f(5));
	System.out.println(I.carre(10));
```

Il est possible de préfixer la déclaration d'une interface fonctionnelle par l'annotation **@FunctionalInterface**. Le compilateur vérifiera ainsi la cohérence de la déclaration de l'interface.