Youssouf IBRAHIM ELMI
26 ans
ETUDES ET DEVELOPPEMENT, spécialisation JAVA J2EE
Tél: +33751135344
yous-souf93@hotmail.com

Java 8

Langage de programmation :

Lambda Expressions : une nouvelle fonctionnalité linguistique, a été introduite dans cette version. Ils nous permettent de traiter la fonctionnalité comme un argument de méthode.

- Les méthodes par défaut permettent d'ajouter de nouvelles fonctionnalités aux interfaces des bibliothèques.

Exemple : on a une interface *Personne* qui possède *getAge()* comme une méthode.
On déclare la manière classique avec java 7 :

```
public interface Personne {
    public int getAge();
    public String getNom();

}
```

On a une classe nommée *PersonneMorale* par exemple. Cette classe va implémenter l'interface Personne mais logiquement la classe *PersonneMorale* ne va pas avoir la méthode *getAge().* Pour cela, on revient sur l'interface *Personne* pour ajouter la nouvelle fonctionnalité de java 8 aux interfaces.
on obtient un résultat ainsi :

```
public interface Personne {
    default int getAge() {return - 1;}
    public String getNom();

}
```

- Les références de méthodes fournissent des expressions lambda faciles à lire pour les méthodes déjà nommées.

Exemple : dans la classe *Principale,* on fait appel a la méthode par :: ;

```
Arrays.sort(tab, Personne :: compare) ;
```

Les annotations répétées permettent d'appliquer le même type d'annotation plusieurs fois à la même déclaration ou au même type d'utilisation.
Exemple avec les annotations en java 8 :

```
@Modifie(auteur="youssouf")
@Modifie(auteur = "Ahmed")
@Modifie(auteur = "Ahmed")
@Modifie(auteur = "Hanane")
```

- Amélioration de l'inférence de type : Par exemple la déclaration de *List* , on a plus besoin de répéter les types du tableau dans *ArrayList*.

```
List<Personne> maL = new ArrayList<>();
```

- **Les collections :**

Les classes flux :
Les classes du nouveau *java.util.stream* package fournissent une API de flux permettant de prendre en charge des opérations de style fonctionnel sur des flux d'éléments.

Youssouf IBRAHIM ELMI
26 ans
ETUDES ET DEVELOPPEMENT, spécialisation JAVA J2EE
Tél: +33751135344
yous-souf93@hotmail.com

```
//collection avec type Objet
List<Personne> maL = new ArrayList<>();
maL.add(new Personne(52,"IBRAHIM"));
maL.add(new Personne(80,"LUCAS"));
maL.add(new Personne(15,"TERRY"));
```

Ou en peux déclarer directement avec *Stream*;

```
//declaration d'un objet Personne avec stream
Stream.of(new Personne(52,"IBRAHIM"), new Personne(80,"LUCAS"), new Personne(15,"TERRY"))
.sorted((x,y)->{return x.compareA(y);})
.forEach(s->System.out.println("Personne: "+s.getAge())); // OU System.out::println
```

Si on utilise des collections classiques avec une chaine des caractères ;

```
List<String> liste = new ArrayList<>();
liste.add("youssouf");
liste.add("ibrahim");
liste.add("elmi");
```

Maintenant, si on veut qu'on affiche nos données avec un *forEach,* voici la syntaxe :

```
//forEach pour parcourir et lambda pour afficher
            System.out.println("-----------affichage classic -----------");
maL.stream().forEach(s->System.out.println("Personne: " + s));
```

Ou afficher de cette manière ;

```
//:: dans le system.out.println est le meme si on disait s->S.O.T(s)
liste.stream().forEach(System.out::println);
```

L'API Stream est intégrée à l'API Collections, permet de réaliser plusieurs fonctionnalités :

- **sorted** (Triés) par exemple :

```
//si on veut trier par age
          System.out.println("-----------affichage par tri Age -----------");
maL.stream().sorted((x,y)->x.getAge() - y.getAge()).forEach(s->System.out.println("Personne: " + s));
```

- **filter**(Filtrer), qui possède plusieurs fonctions. On verra quelque utilisation avec *filter*

```
//si on veut filtrer sur un nom
System.out.println("-----------filtrer les Noms qui commence L -----------");
maL.stream().filter(e->e.getNom().startsWith("L")).forEach(s->System.out.println("Peresonne: "+s));
//maL.stream().filter(e->e.starsWith("L")).forEach(s->System.out.println("Personne: " + s));
```

```
//si on veut avoir que les ages sup 30
        System.out.println("-----------filtrer avec condition Age (>30ans)-----------");
maL.stream().filter(p->p.getAge() > 30 ).forEach(s->System.out.println("Personne "+s));
```

- **collect**, pour collecter un ensemble des calcules et de les affichés ensuite ;

```
System.out.println("-----------compter les nombres d'age-----------");
double comptage = maL.stream().collect(Collectors.counting());
System.out.println("Age Moyen=>" +comptage);
```

- **reduce** (réduire), nous devons effectuer des opérations où un flux réduit à une seule valeur résultante, par exemple, maximum, minimum, somme, produit, etc. La réduction est le processus répété de combinaison de tous les éléments.

```
System.out.println("-----------Sommes d'age-----------");
Integer totalAge = maL.stream().reduce(0, (somme,p)->somme+=p.getAge(), (s1,s2)->s1+s2);
System.out.println("La somme=>" +totalAge);
```

Youssouf IBRAHIM ELMI
26 ans
ETUDES ET DEVELOPPEMENT, spécialisation JAVA J2EE
Tél: +33751135344
yous-souf93@hotmail.com

- Les interface Fonctionnelle : c'est une interface qui représente une fonction et elle contient
  - **Predicate** : qui est un type d'une interface fonctionnelle. Le prédicat prend au paramètre un type et retourne rien ou au pire retourne une booléen.

Voici un exemple :

```
Predicate<Ressource> rDispo  = (Ressource r)->r.estDispo();
Stream.of(new Ressource(true), new Ressource(false))
.filter(rDispo).forEach(System.out::println);
```

- **Function** : elles prennent un paramètre et retourne quelque chose.

Exemple d'une fonction qui prend une température Celsius et retourne en fahrenheit.

```
Function<Integer, Double>cf=x->new Double(x * 9/5 + 32);
System.out.println(cf.apply(35)+" F");
```

- **BiPredicate** : c'est comme la fonction prédicat mais elle prenne deux paramètres en entrer.

Exemple de comparaison de deux chaines des caractères :

```
BiPredicate<String, String> loi = (String s1, String s2)->{return s1.length() > s2.length();};
System.out.println(loi.test("youssouf", "lucas"));
```

- **BiFunction** : c'est comme la fonction *Function*, elle prenne trois paramètres pour retourne un résulte en utilisant une lambda expression comme les autres aussi.

Exemple :la somme de deux chaine des caractères.

```
BiFunction<String, String, Integer>moi = (String s1, String s2)->{return s1.length() + s2.length();};
System.out.println(moi.apply("youssouf", "ahmedi"));
```

- Les dates et les Temps :

```
LocalDate : qui nous donne la date ;
LocalTime : qui nous donne l'heure ;
```

```
LocalTime ttt= LocalTime.now();
System.out.println(ttt);

LocalDateTime tt = LocalDateTime.now();
System.out.println(tt);
```

Si on veut voire l'heure dans d'autre pays, rien n'est plus simple on précise seulement la zone comme ceci :

```
LocalTime europe = LocalTime.now(ZoneId.of("Asia/Tokyo"));
System.out.println(europe);
```