Lambda calcul dans Java 1.8 – adaptation du cours de Jean Michel Doudoux

Une interface fonctionnelle (functional interface en anglais)
est basiquement une interface dans laquelle une seule méthode abstraite est définie. Elle doit respecter certaines contraintes :

- elle ne doit avoir qu'une seule méthode déclarée abstraite
- les méthodes définies dans la classe Object ne sont pas prises en compte comme étant des méthodes abstraites
- toutes les méthodes doivent être publiques
- elle peut avoir des méthodes par défaut, ainsi que des méthode de classe

Avant Java 8 un certain nombre d'interfaces ne définissaient qu'une seule méthode : ces interfaces sont désignées par le terme Single Abstract Method (SAM).
Avant Java 8, le JDK contenait déjà de nombreuses interfaces qui respectaient ces règles et sont donc des interfaces fonctionnelles, par exemple :

- `Comparator<T>` qui définit la méthode `int compare(T o1, T o2)`
- `Callable<V>` qui définit la méthode `V call() throws exception`
- `Runnable` qui définit la méthode `void run()`
- `ActionListener` qui définit la méthode `void actionPerformed(ActionEvent)`
- ...

Elles se prêtent bien à l'utilisation d'une classe anonyme interne.

Il est donc possible d'utiliser une expression lambda à la place d'une classe anonyme.

L'annotation `@FunctionalInterface` permet d'indiquer explicitement au compilateur que l'interface est une interface fonctionnelle : le compilateur pourra alors vérifier que les contraintes d'une interface fonctionnelles sont respectées. Son utilisation est facultative.

Les expressions Lambda
Une expression lambda est une fonction anonyme : sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom. C'est un raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée.

Une expression lambda est donc un raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre. Elle est particulièrement utile notamment lorsque le traitement n'est utile qu'une seule fois : elle évite d'avoir à écrire une méthode dans une classe.

Une expression lambda permet d'encapsuler un traitement pour être passé à d'autres traitements. C'est un raccourci syntaxique aux classes anonymes internes pour une interface qui ne possède qu'une seule méthode abstraite. Ce type d'interface est nommé interface fonctionnelle.

Lorsque l'expression lambda est évaluée par le compilateur, celui-ci infère le type vers l'interface fonctionnelle. Cela lui permet d'obtenir des informations sur les paramètres utilisés, le type de la valeur de retour, les exceptions qui peuvent être levées.

Elles permettent d'écrire du code plus compact et plus lisible. Elles ne réduisent pas l'aspect orienté objet du langage qui a toujours été une force mais au contraire, rendent celui-ci plus riche et plus élégant pour certaines fonctionnalités.

La syntaxe des expressions lambda

Les avantages des expressions lambda est d'avoir une syntaxe très simple.
La syntaxe d'une expression lambda est composée de trois parties :

- un ensemble de paramètres, d'aucun à plusieurs
- l'opérateur `->`
- le corps de la fonction

Elle peut prendre deux formes principales :

```
(paramètres) -> expression;
(paramètres) -> { traitements; }
```

L'écriture d'une expression lambda doit respecter plusieurs règles générales :

- zéro, un ou plusieurs paramètres
- les paramètres sont entourés par des parenthèses et séparés par des virgules. Des parenthèses vides indiquent qu'il n'y a pas de paramètre
- le corps de l'expression peut contenir zéro, une ou plusieurs instructions. Si le corps ne contient d'une seule instruction, les accolades ne sont pas obligatoires et le type de retour correspond à celui de l'instruction. Lorsqu'il y a plusieurs instructions alors elles doivent être entourées avec des accolades.

Exemple

`comparatorLength` est un objet implémentant l'interface fonctionnelle `Comparator<String>`.

La technique classique de création pour créer l'objet `comparatorLength` :

```
Comparator<String> comparatorLength = new Comparator<String>{
 public int compare(String chaine1, String chaine2) {
     Integer.compare(chaine1.length(),chaine2.length()) }}
```

Utilisation d'une lambda expression pour créer l'objet `comparatorLength` :

```
      Comparator<String> comparatorLength = (String chaine1, String chaine2)
            -> Integer.compare(chaine1.length(),chaine2.length());
```

l'usage d'une lambda expression anonyme :

```
monBouton.addActionListener((ActionEvent event) -> System.out.println("clic"));
```

encore plus concis - le type du paramétre est inféré (calculé par le compilateur) :

```
      monBouton.addActionListener(event -> System.out.println("clic"));
```

exemple d'une lambda expression équivalente à la construction d'objet anonyme d'une classe anonyme. Le code suivant

```
//lambda expression
      button.setOnAction (
            (ActionEvent event) ->
            System.out.println(Thread.currentThread().getName()+" : Hello World!")
      );
```

remplace :

```
// objet anonyme d'une classe anonyme
button.setOnAction(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
            System.out.println(Thread.currentThread().getName()+" : Hello World!");
            }}      );
```

exemple d'une lambda expression équivalente à la construction d'un objet. Le code suivant :

```
EventHandler<ActionEvent> handler2 =
                    (event) -> {
                          String st = ((Button) event.getSource()).getText()+"\n";
                          righTextArea.appendText(st);  };
```

remplace :

```
EventHandler<ActionEvent> handler2 = new EventHandler<ActionEvent>() {
                    @Override
                    public void handle(ActionEvent event) {
                          String st = ((Button) event.getSource()).getText()+"\n";
                          righTextArea.appendText(st);
             }};
```