

TP 1: Découverte d'expressions Lambda

Prérequis

Développeur Java comprenant notions d'interface, classe anonyme.

Objectifs

Comprendre les enjeux associés aux expressions lambda

Montrer l'impact de l'expression lambda sur le code compilé

Gérer avec les expressions lambda les paramètres et types de retour

Comprendre l'inférence de méthode avec les expressions lambda

Contexte et enjeux

INFO 1 : jusqu'à Java 7, il n'existait que deux types de références : des références vers des valeurs primitives (`int i = 37`), et des références vers des instances d'objets (`Employe employe1 = new Employe()`).

INFO 2 : dans d'autres langages (Groovy, Scala,...) pourtant, il était déjà possible d'établir des références vers des '**closures**', c'est-à-dire des blocs de code anonymes.

INFO 3 : en Java, qui ne disposait pas de cette facilité, la technique qui s'en rapproche le plus consiste à définir une interface décrivant la fonctionnalité souhaitée, puis à instancier une classe (souvent anonyme) implémentant la fonctionnalité. L'instance obtenue peut alors être affectée à une référence et/ou être passée en paramètre d'une méthode... **cette façon de faire est très verbeuse** comme vous le constaterez plus loin dans ce tutoriel.

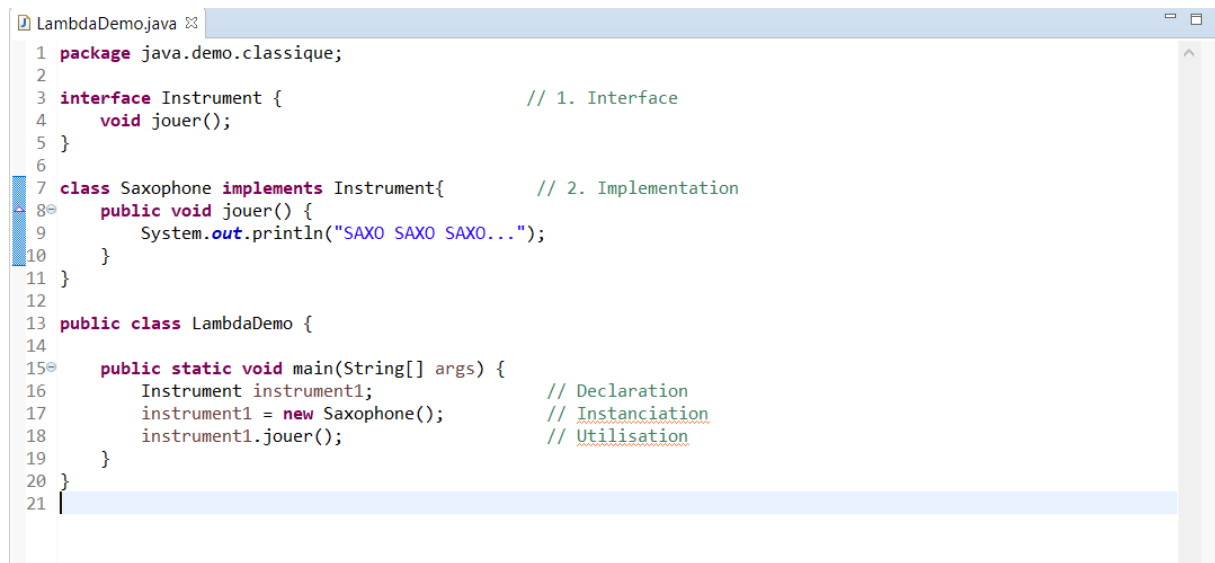
INFO 4 : de ce fait, Oracle s'est décidé à intégrer les **closures** dans le langage. Ainsi à partir de Java 8, il est possible d'avoir une référence vers...une méthode !

DEFINITION : une expression Lambda est l'implémentation d'une closure en java.

SYNTAXE : une closure se conçoit comme une méthode anonyme. A ce titre, elle peut accepter des paramètres et retourner un résultat.

Partie 1 : Approche classique de mise en œuvre d'une interface

Analysez le code suivant (ici classes et interfaces codées dans le même fichier pour faciliter la compréhension) :



```
1 package java.demos.classique;
2
3 interface Instrument {           // 1. Interface
4     void jouer();
5 }
6
7 class Saxophone implements Instrument { // 2. Implementation
8     public void jouer() {
9         System.out.println("SAXO SAXO SAXO...");
10    }
11 }
12
13 public class LambdaDemo {
14
15     public static void main(String[] args) {
16         Instrument instrument1; // Declaration
17         instrument1 = new Saxophone(); // Instanciation
18         instrument1.jouer(); // Utilisation
19     }
20 }
21 }
```

Partie 2 : Classe anonyme

Le codage de la classe Saxophone a été nécessaire pour pouvoir utiliser l'interface Instrument. Or il arrive souvent, qu'une implémentation est trop spécifique pour être généraliser dans une classe. Du coup il apparait que le code doit être écrit directement à l'emplacement où il est nécessaire. La solution classiquement (avant java 8) utilisée est l'implémentation d'une classe anonyme.

Question : Que donne ce code si nous souhaitons utiliser juste une classe anonyme en évitant l'implémentation de la classe ? (implémenter dans un nouveau package)

Partie 3 : Expression Lambda

En utilisant ce que nous venons de voir sur les expressions lambdas écrire le même code en utilisant une expression lambda. (implémenter dans un nouveau package).

- Ajouter des paramètres à la méthode *jouer* de l'interface Instrument et réécrire l'expression lambda (Par exemple : `void jouer(String titre)`)
- Ajouter un type de retour (Par exemple : `String jouer(String titre)`)

Discussions :

Analysez les binaires générés par les 3 projets et commentez les.

TP 3 : Tri sur des Strings

Soit la liste suivante :

```
List<String> listStrings =  
Arrays.asList("Orange", "Grape", "Apple", "Lemon", "Banana");
```

En utilisant les lambdas les trier par :

- La longueur de chaque élément (length : de la plus courte à la plus longue et/ou inversement)
- L'ordre alphabétique
- Les chaînes de caractère contenant "e" en premier et les autres ensuite.
- Refaire le même code en utilisant : `Arrays.sort(words, (s1,s2) -> Utils.yourMethod(s1,s2))`

TP 2: Mise en œuvre de lambdas pour le tris d'objets

Soit le projet *TPExpressionsLambdaTris* .

- Réécrire le tri en utilisant les expressions lambdas.

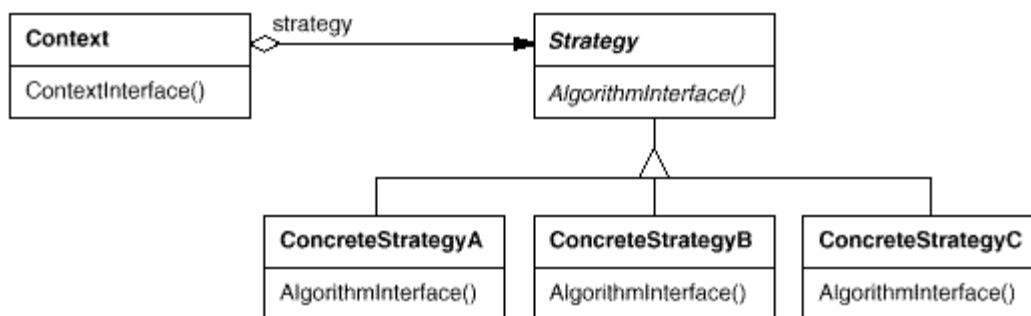
TP 4 : Réécriture d'une implémentation des Design Pattern Strategy et Observer en utilisant les interfaces fonctionnelles et des expressions Lambdas

1. Objectif

L'objectif de billet est de montrer un cas pratique de réécriture d'un pattern bien connu et déjà implémenté dans les versions antérieures en utilisant des expressions lambdas. L'idée étant que cette réécriture soit plus concise et plus simple à lire.

1.1. Le pattern Strategy

L'idée générale dans ce patron est la suivante : une famille d'algorithmes est encapsulée de manière qu'ils soient interchangeables. Les algorithmes peuvent changer indépendamment de l'application qui s'en sert. Il comporte trois rôles : le contexte, la stratégie et les implémentations. La stratégie est l'interface commune aux différentes implémentations - typiquement une classe abstraite. Le contexte est l'objet qui va associer un algorithme avec un processus.



1.1.1. Implémentation

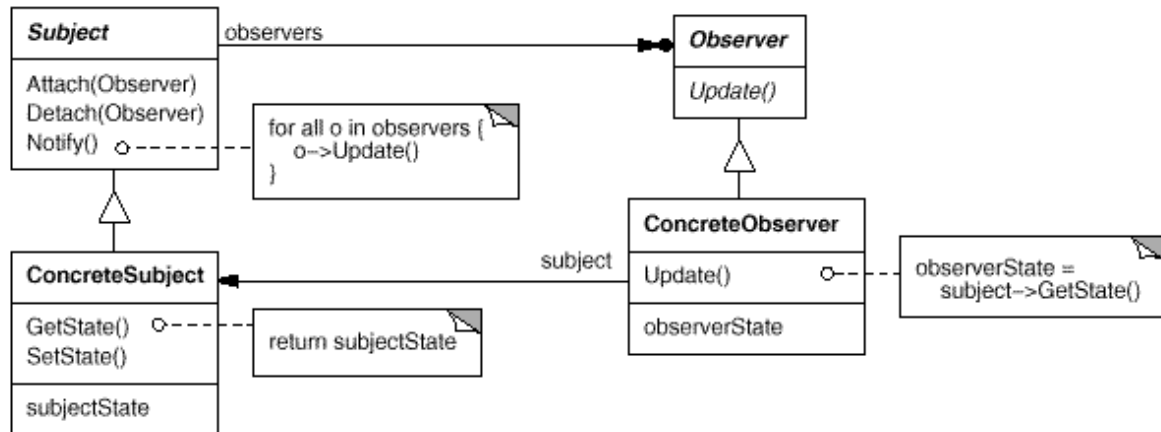
Télécharger le projet *StrategyByLambda*

Analyser et commenter la réécriture en lambda proposée.

1.2. Le Pattern Observer

Le patron **observateur** est un [patron de conception](#) de la famille des patrons comportementaux. Dans ce patron, le sujet observable se voit attribuer une collection d'observateurs qu'il notifie lors de changements d'états. Chaque observateur concret est chargé de faire les mises à jour adéquates en fonction des changements notifiés.

Ainsi, l'observé n'est pas responsable des changements qu'il entraîne sur les observateurs.



- Télécharger le projet *ObserverByLambda*, qui est une implémentation de ce pattern.
- Réécrire ce pattern (Comme on l'a fait pour le pattern strategy) en utilisant les Lambdas