

1. LAMBDA

Objectif : Mise en œuvre de Lambda, manipulation d'interfaces fonctionnelles.

Récupérez le projet Lab9 se trouvant dans labs\lab9\before, si vous n'avez pas ce projet, nous avons fourni le code dans l'exercice.

Ce projet recherche **les personnes** âgées de plus de 20 ans.

Le but est de réécrire la méthode de recherche en utilisant les lambdas, cette réécriture va rendre cette méthode plus générique.

En effet, au lieu de limiter la méthode à une recherche de personnes plus âgées qu'un certain âge, elle va afficher les personnes qui répondent à un certain critère.

Ce critère aura l'avantage de pouvoir changer à chaque appel de la méthode (par exemple afficher les personnes plus jeunes qu'un certain âge ...). Pour cela on va utiliser l'interface **Predicate<T>**, où <T> est notre classe Personne.

Notre classe Person reste inchangée, seul le test est à modifier.

La classe Person

```
package before.Lab9.src.com.gk;
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;    }

    public void printPerson() {
        System.out.println(name + " a " + age + " ans");
    }

    public String getName() { return name; }

    public void setName(String name) {this.name = name; }

    public int getAge() {return age;}

    public void setAge(int age) {    this.age = age;    }
}
```

Le test – Avant la manipulation de lambdas.

```
package before.Lab9.src.com.gk.test;

import java.util.ArrayList;
import java.util.List;
import before.Lab9.src.com.gk.*;

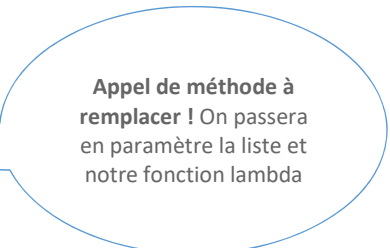
public class OldWay {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Person p1 = new Person("Thomas", 19);
        Person p2 = new Person("Paul", 25);
        Person p3 = new Person("Max", 22);
        Person p4 = new Person("Celine", 27);
        Person p5 = new Person("Jennifer", 18);

        List<Person> persons = new ArrayList<Person>();

        persons.add(p1);
        persons.add(p2);
        persons.add(p3);
        persons.add(p4);
        persons.add(p5);
        printPersonOlderThan(persons, 20);
    }

    //Méthode de recherche
    public static void printPersonOlderThan(List<Person> persons, int age) {
        for (Person p : persons) {
            if (p.getAge() >= age) {
                p.printPerson();
            }
        }
    }
}
```



Appel de méthode à
remplacer ! On passera
en paramètre la liste et
notre fonction lambda

8.1. Interface fonctionnelle Predicate<T>

Le test doit utiliser une fonction qu'on se propose de nommer **plusVieuxQue** qui retournera, les personnes ayant un âge supérieur à 20. On utilise l'interface fonctionnelle **Predicate** car elle détient une méthode **test(T t)** qui retourne un booléen. Cette interface est donc candidate à l'utilisation de lambda expression.

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t) ;

}
```

Cette méthode test(T) effectue un test sur l'objet T, puis retourne vrai ou faux.

- C'est exactement ce que l'on souhaite : il suffit que la méthode test() se charge de vérifier que la personne est plus âgée qu'un certain âge, ou à l'inverse qu'elle est moins âgée.
- Pour cela, il suffit de changer l'implémentation de la méthode test() en fonction du traitement que l'on souhaite effectuer, et de passer l'objet Predicate associé en paramètre de la méthode printPersonsWithPredicate() que l'on écrira en place de la méthode printPersonOlderThan.

Dans un premier temps, il faut créer la fonction **plusVieuxQue**, avec Predicate.
Puis écrire la méthode dans notre test :

```
static printPersonsWithPredicate(List<Person> persons, Predicate<Person> tester).
```

8.2. Interface fonctionnelle Predicate<T>, autre fonction

Dès que votre test fonctionne, ajouter une nouvelle fonction de méthode **plusJeuneQue**, toujours en manipulant Predicate et qui retourne des objets Person plus jeunes que 20 ans. Utiliser la même méthode static *printPersonsWithPredicate* pour afficher les personnes correspondant aux critères.

Utiliser une variable pour définir et initialiser l'âge à 20.

8.3. Interface fonctionnelle `Consommer<T>`, pour l'affichage des personnes

Ajoutons une fonction de méthode, pour afficher les personnes répondant au critère, cette méthode se contente d'effectuer une action générique. Cette fois, c'est l'interface **`Consumer<T>`** qui nous intéresse.

C'est également une interface fonctionnelle, donc ne contenant qu'une seule méthode : `void accept(T t)`. Cette méthode effectue un traitement sur un objet `T` sans rien renvoyer, ce qui est idéal pour un appel à la méthode `printPerson()` !

```
Consumer<Person> afficherPerson = (p) -> {  
    p.printPerson();  
};
```

Créer une nouvelle méthode static qui reçoit la liste, le prédicat et le consommateur (*c'est-à-dire la fonction*) pour gérer la recherche et l'affichage :

```
public static void processPersons(List<Person> persons, Predicate<Person> tester,  
Consumer<Person> block) {
```

```
    ...  
}
```

S'il vous reste du temps, créer une nouvelle fonction de méthode où l'action consiste à ajouter 1 an à chaque personne.

8.4. Gestion bancaire - Lambda et Stream

Objectif : Manipuler des lambdas et des Streams dans notre gestion bancaire.

Reprendre le TestCompte, nous allons manipuler l'historique des mouvements bancaire. A ce stade, vous devez avoir plusieurs Mouvement dans l'historique de type Depot, Retrait.

Afficher l'historique

Dans un premier temps, afficher le contenu de l'historique en utilisant, une expression lambda dans la méthode `forEach()` via une méthode de référence.

Par la suite, manipuler les streams

Calculer la somme des dépôts

Avec la manipulation d'un Stream sur l'historique et les fonctions `map` et `sum`, calculer la somme des dépôts.

Indications, il existe une méthode `mapToInt` qui attend une lambda et retourne un entier, nous pourrions donc l'utiliser ainsi pour obtenir les sommes déposées :

`mapToInt(Mouvement::getPlus)`

Par la suite, on applique la méthode `sum()` au stream obtenu.

Compter les objets Depot ayant un montant supérieur à 60

Créer une fonction de type `IntPredicate` permettant de ne conserver que les montants supérieurs à un certain montant que l'on fixe à 60 ;

Récupérer les montants des dépôts (toujours avec `mapToInt`), filtrer ceux qui sont supérieurs à 60 via le prédicat, compter.

