Rappel Séance 2 (week-end 27-28/11/2021)

1. Créer un objet à partir des interfaces fonctionnelles suivante en utilisant les expressions Lambda

```
@FunctionalInterface
interface Testable{
   boolean isCorrect(String email);
}
@FunctionalInterface
interface Operation{
   int addition(int a,int b);
}
@FunctionalInterface
interface Logger{
   void display(String log);
}
```

2. Choisir les expressions Lambda correcte pour la méthode suivante:

   Integer add(Integer a,Integer b);

```
1 -    a,b-> a+b;
2 -    (a,b)-> a+b;
3 -    (a,b)-> return a+b;
4 -    (a,b)-> {return a+b;};
5 -    (a,b)-> {return a+b};
```

Séance 3 (week-end 11-12/12/2021)

Les objectifs de la séance d'aujourd'hui:

Objectif 5.1 : Création des **annotations** en Java

Objectif 5.2 : Utilisation des **collections** en Java

Complement Youtube de cette séance : https://youtu.be/KsO3Uqf_oxw

I. Création et utilisation des annotations en java

1. Dans votre projets "Mes TPs Java", créer un package "ma.education.tp5.annotations"
2. Créer une annotation Programmer, les annotations sont créées en utilisant le mot réservé @interface . Cette annotation contient les deux signatures: int id(); et String name();

```
package ma.education.tp5.annotations;
public @interface Programmer {
      abstract int id();
      String name();
}
```

3. Cette annotation sera appliquée seulement aux classes et aux interfaces. Pour le dire, l'annotation Programmer doit être annoté par l'annotation @Target

```
@Target(ElementType.TYPE)
public @interface Programmer {
      abstract int id();
      String name();
}
```

ElementType.TYPE = Class, interface (including annotation type), or enum declaration

4. Dans ma.education.tp5.annotations, créer la classe Calculatrice, en utilisant l'annotation précédente @Programmer mentionner le programmeur qui a développé la classe Calculatrice.

```
@Programmer(id=10,name="Said ALAMI")
public class Calculatrice {
}
```

5. Créer une classe TestReflectionAnnotation pour afficher les valeurs de l'annotation @Programmer utilisées dans la classe Calculatrice

```
public class TestReflectionAnnotation {
      public static void main(String[] args) {
            Class c = Calculatrice.class;
            Programmer programmer = (Programmer)
c.getDeclaredAnnotation(Programmer.class);

System.out.println(programmer.id()+":"+programmer.name());
      }
}
```

Exécuter cette classe et remarquer que l'annotation n'existe pas au moment de l'exécution

Exception in thread "main" java.lang.NullPointerException

Une annotation est définie par sa rétention, c'est-à-dire la façon dont elle sera conservée. La rétention est définie grâce à la méta-annotation @Retention. Les différentes rétentions d'annotation possibles sont :

***SOURCE :** L'annotation est accessible durant la compilation mais n'est pas intégrée dans le fichier .class généré.*

***CLASS :** L'annotation est accessible durant la compilation, elle est intégrée dans le fichier .class généré mais elle n'est pas chargée dans la JVM à l'exécution.*

***RUNTIME :** L'annotation est accessible durant la compilation, elle est intégrée dans le fichier class généré et elle est chargée dans la JVM à l'exécution. Elle est accessible par introspection (la reflection).*

6. Ajouter alors @Retention(RetentionPolicy.RUNTIME) à l'annotation @Programmer et exécuter encore une fois la classe TestReflectionAnnotation

```
package ma.education.tp5.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Programmer {
      abstract int id();
      String name();
}
```

7. Créer une classe fille de Calculatrice et appeler la CalculatriceMath. Est ce que la classe fille va hériter l'annotation @Programmer de sa classe mère Calculatrice.

```
public class CalculatriceMath extends Calculatrice{
}
```

8. Changer la classe TestReflectionAnnotation pour vérifier si CalcutriceMath est aussi annotée par @Programmer

```
public class TestReflectionAnnotation {
      public static void main(String[] args) {
            Class c = CalculatriceMath.class;
            Programmer programmer = (Programmer)
c.getAnnotation(Programmer.class);

System.out.println(programmer.id()+":"+programmer.name());
      }
}
```

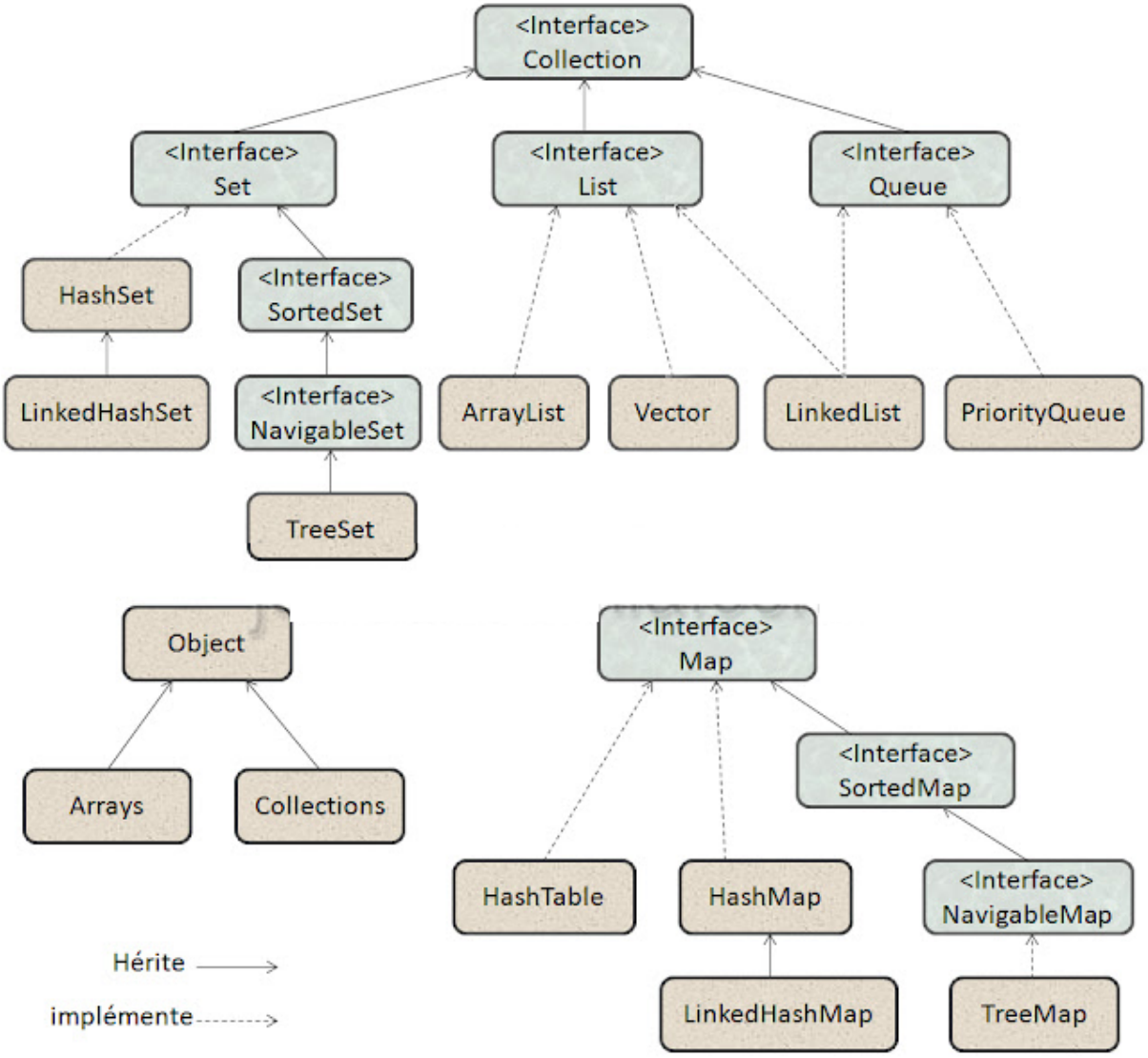
N'oublier pas de changer c.**getDeclaredAnnotation**(Programmer.class) par c.**getAnnotation**(Programmer.class)

Exécuter cette classe et remarquer l'exception : java.lang.NullPointerException

9. Annoter l'annotation @Programmer par l'annotation @Inherited et refaire l'exécution de la classe TestReflectionAnnotation. C'est quoi votre remarque?

Annoter le package "ma.education.tp5.annotations" par l'annotation @Programmer et afficher les valeurs de l'annotation en utilisant la réflection.

II. Utilisation des collections en java



A. List : est une collection qui accepte des éléments dupliqués et qui respecte l'ordre d'insertion.

1. Créer un package "ma.education.tp5.collections". Créer une classe TestList avec la méthode main comme suivant

```
package ma.education.tp5.collections;

import java.util.ArrayList;
import java.util.List;

public class TestList {
   public static void main(String[] args) {
       List<Integer> list1= new ArrayList<Integer>();
       list1.add(12);
       list1.add(23);
       list1.add(23);
       list1.add(12);
```

```
    }
}
```

2. Afficher les éléments de la liste en utilisant :
    a. La boucle for classique

```
// Affichage en utilisant la boucle for classique
for(int i = 0;i<list1.size();i++){
   Integer e = (Integer) list1.get(i);
   System.out.println(" element "+i +" : "+e);
}
```

    b. La boucle for avancée

```
// Affichage en utilisant la boucle for avancée
for(Integer e :list1){
   System.out.println(" element "+e);
}
```

    c. Un iterateur

```
// Affichage en utilisant un iterator
Iterator<Integer> iter = list1.iterator();
while(iter.hasNext()){
   Integer e = iter.next();
   System.out.println("element : "+e);
}
```

    d. La méthode forEach de la List

```
// Affichage en utilisant un iterator
list1.forEach(i-> System.out.println(" element "+i));
```

 Quelle est la meilleure façon pour itérer sur une collection?

3. Remarquer que les listes acceptent les doublons (12,12) et (23,23) et aussi qu'ils respectent l'ordre d'insertion.

4. Maintenant créer une liste list2 avec les nombres suivants : 1,5,6,9,16
    a. Boucler sur la liste et afficher le carré des élément de la liste s'il est supérieur à 37

```
List<Integer> list2= Arrays.asList(1,5,6,9,16);
list2.stream().map(i->i*i).filter(i->i>37).forEach(i->
System.out.println(i));
```

5. Donner le resultat de l'execution de la classe suivante:

```
public class TestList {
   public static void main(String[] args) {
        List<Integer> list3= new ArrayList<>();
        list3.add(5);
```

```
        list3.add(10);
        list3.add(15);
        list3.add(20);
        list3.add(2,10);
        list3.forEach(i-> System.out.println(i));
    }
}
```

B. <u>Set : est une collection qui n'accepte pas des éléments dupliqués. Il représente le modèle mathématique des ensembles. Les sets ne respecte pas l'ordre d'insertion</u>

Les Sets ajoutent à l'interface Collection des méthodes pour l'interdiction des éléments en double. Ils utilisent une table de hachage pour le stockage de ses éléments.

6. Dans le package "ma.education.tp5.collections". Créer une classe TestSet avec la méthode main comme suivant:

```
package ma.education.tp5.collections;

import java.util.HashSet;
import java.util.Set;

public class TestSet {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<String>();
        set1.add("ABC1");
        set1.add("ABC2");
        set1.add("ABC3");
        set1.add("ABC1");
        set1.add("ABC4");
        set1.add("ABC5");
        set1.forEach(i-> System.out.println(i));
    }
}
```

7. Exécuter TestSet et remarquer que les doublons n'existent pas dans la Set set1. Remarquer que l'ordre d'insertion n'est pas respecté.
8. Créer la classe Client avec le code (Integer) et le name (String) comme attributs. Générer le constructeur et la méthode toString() pour la classe Client

```
class Client{
    Integer code;
    String name;
    public Client(Integer code, String name) {
        this.code = code;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Client{" +
                "code=" + code +
                ", name='" + name + '\'' +
                '}';
    }
```

```
}
```

9. Dans la méthode main de la classe TestSet, créer une Set set2 qui contient des objets de la classe Client.

```
public class TestSet {
   public static void main(String[] args) {
        Set<Client> set2 = new HashSet<Client>();
        set2.add(new Client(1,"ALAMI"));
        set2.add(new Client(1,"ALAMI"));
        set2.add(new Client(2,"SOUHAIL"));
        set2.forEach(i-> System.out.println(i));
   }
}
```

10. Etant donné que les Sets n'acceptent pas les doublons, combien d'objets on aura dans la Set2. L'exécution de TestSet donnera

```
Client{code=1, name='ALAMI'}
Client{code=2, name='SOUHAIL'}
Client{code=1, name='ALAMI'}
```

Il faut alors définir la notion de doublons pour les Sets:

Deux objets sont considérés doublons s'ils ont le même hashCode et si leur méthode equals retourne true;

Exemple : Deux clients sont considérons doublons s'ils ont le même code et le même nom sera représenté en java par la redéfinition de la méthode hashCode et la méthode equals comme suivants:

```
@Override
public boolean equals(Object o) {
   Client client = (Client) o;
   return this.code==client.code && this.name.equals(client.name);
}

@Override
public int hashCode() {
   return code;
}
```

11. Refaire l'exécution de la classe TestSet suite à l'ajout de la méthode hashCode() et la méthode equals(). Quelle votre remarque.

L'implémentation HashSet se base sur la méthode hashCode() et la méthode equals() pour la définition des doublons.

12. Maintenant créer une Set "set3" avec l'implémentation TreeSet.

```java
public class TestSet {
    public static void main(String[] args) {
        Set<Integer> set3 = new TreeSet<>();
        set3.add(22);
        set3.add(11);
        set3.add(15);
        set3.add(9);
        set3.forEach(i-> System.out.println(i));
    }
}
```

13. Faire l'exécution de TestSet et remarquer que l'implémentation TreeSet fait le tri.

```
9
11
15
22
```

14. Créer une Set “set4” qui contient les clients suivant:

```java
public class TestSet {
    public static void main(String[] args) {
        Set<Client> set4 = new TreeSet<>();
        set4.add(new Client(1,"OMAR"));
        set4.add(new Client(3,"SAID"));
        set4.add(new Client(2,"HASSAN"));
        set4.forEach(i-> System.out.println(i));
    }
}
```

15. Remarquer que si les objets ne sont pas comparable (aucun critère de comparaison), une erreur d'exécution sera levée:

```
Exception in thread "main" java.lang.ClassCastException:
ma.education.tp5.collections.Client cannot be cast to java.lang.Comparable
```

16. Pour donner un critère de comparaison, il faut Implémenter l’interface Comparable au niveau de la classe client comme suivant:

```java
class Client implements Comparable{
    Integer code;
    String name;
    public Client(Integer code, String name) {
        this.code = code;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Client{" +
                "code=" + code +
                ", name='" + name + '\'' +
                '}';
    }
    @Override
    public int compareTo(Object o) {
        Client client = (Client) o;
```

```
        return -client.code+this.code;
    }
}
```

17. Suite à l'ajout de la méthode compareTo, refaire de l'exécution de la classe TestSet. Remarquer que les clients sont ordonnés par leur code.

Créer un code java pour trier les clients par leur nom décroissant.

18. Il est possible de créer un comparateur en dehors de la classe Client via l'implémentation de l'interface Comparator. Cela laissera la classe Client propre

La classe Client :

```
class Client{
    Integer code;
    String name;
    public Client(Integer code, String name) {
        this.code = code;
        this.name = name;
    }
    @Override
    public String toString() {
        return "Client{" +
                "code=" + code +
                ", name='" + name + '\'' +
                '}';
    }
}
```

Le comparateur CodeComparator:

```
package ma.education.tp5.collections;

import java.util.Comparator;

public class CodeComparator implements Comparator<Client> {
    @Override
    public int compare(Client o1, Client o2) {
        return o1.code-o2.code;
    }
}
```

La classe TestSet avec l'utilisation du comparateur:

```
public class TestSet {
    public static void main(String[] args) {
        Comparator<Client> c = new CodeComparator();
        Set<Client> set4 = new TreeSet<>(c);
        set4.add(new Client(1,"OMAR"));
        set4.add(new Client(3,"SAID"));
        set4.add(new Client(2,"HASSAN"));
        set4.forEach(i-> System.out.println(i));
    }
```

```
}
```

19. Exécuter la classe TestSet et remarque que les clients sont triés par leur code
20. Améliorer le code de la classe TestSet en utilisant les expressions Lambda et sans créer la classe CodeComparator.

```
Set<Client> set4 = new TreeSet<>((c1,c2) -> c1.code-c2.code);
set4.add(new Client(1,"OMAR"));
set4.add(new Client(3,"SAID"));
set4.add(new Client(2,"HASSAN"));
set4.forEach(i-> System.out.println(i));
```

21. Trier les clients par leur nom en utilisant les expressions Lambda.

```
Set<Client> set4 = new TreeSet<>((c1,c2) -> c1.code-c2.code);
set4.add(new Client(1,"OMAR"));
set4.add(new Client(3,"SAID"));
set4.add(new Client(2,"HASSAN"));
set4.forEach(i-> System.out.println(i));
```