TD 1

Exercice 1 : Propriétés fonctionnelles

Soit la classe `C1`.

```java
public class C1 {

    public int a = 1;
    public static int b = 1;
    public static final int C = 1;
    private int d = 1;

    void m1(int val) { this.a = val; }

    int m2() { return this.a; }

    void m3() {
        System.out.println("a = " + a);
    }

    int m4(int val) { return val+val; }

    int m5(int val) { return val + a; }

    int m6(int val) { return val + b; }

    int m7(int val) { return val + C; }

    int m8(int val) { return val + d; }

    int m9(int val) { return d/val; }

    int m10(int val) { return (int) (d/(float)val); }

    int m11(int val) { val++; return val; }

}
```

Question 1 : discuter des propriétés fonctionnelles des méthodes suivantes :

- `String::lowercase`
- `Math::max`, `Math::sin`
- `List::add`, `List::size`, `List::contains`
- `Integer::valueOf`
- `File::length`
- les méthodes `m1`à `m11` de `C1`

Exercice 2 : Paires

On désire réaliser un type générique de paire de valeurs. Les méthodes possibles sont :

- `fst` pour avoir le premier élément de la paire
- `snd` pour avoir le second élément de la paire
- `changeFst` pour retourner une nouvelle paire où le premier élément a été changé par la valeur passée en paramètre (potentiellement d'un autre type que celui précédent)
- `changeSnd` pour retourner une nouvelle paire où le second élément a été changé par la valeur passée en paramètre (potentiellement d'un autre type que celui précédent)

Question 1 : écrire la classe `Paire` générique permettant cela. Vous écrirez aussi une méthode `toString` qui pour une paire `(x,y)` affiche `(x, y) :: Paire[A,B]` où `A` et `B` sont respectivement les types de `x` et de `y`.

Question 2 : écrire le programme qui permet d'obtenir la sortie suivante (utilisation des méthodes `changeFst` et `changeSnd`) :

```
(1, un) :: Paire[Integer,String]
(1.0, un) :: Paire[Double,String]
(1.0, (1, un) :: Paire[Integer,String]) :: Paire[Double,Paire]
```

Exercice 3 : Arbres généralisés

Arbres simples On désire pouvoir représenter des arbres dont les feuilles contiennent des entiers (`Integer` afin de retourner `null` dans certains cas) et dont les noeuds peuvent avoir un nombre quelconque de fils (sous-arbres). L'interface correspondante est la suivante :

```java
public interface Arbre {
    int taille(); // nombre de valeurs
    boolean contient(final Integer val); // vrai si val est contenue dans l'arbre, faux sino
    Set<Integer> valeurs(); // ensemble des valeurs différentes dans l'arbre
    Integer somme(); // somme des valeurs
    Integer min(); // valeur minimale
    Integer max(); // valeur maximale
    boolean estTrie(); // vrai si l'arbre est trié, faux sinon
}
```

Question 1 : écrire une classe `Feuille` pour les feuilles

Question 2 : écrire une classe `Noeud` pour les noeuds

Des tests vous seront fournis pour tester vos classes.

Généralisation On souhaite généraliser ces arbres à des arbres contenant n'importe quel type de valeurs (mais un seul type donné pour un arbre donné). Pour commencer on va se limiter aux méthodes suivantes :

```
public interface Arbre { // à généraliser
    int taille();
    boolean contient(final Integer val);
    Set<Integer> valeurs();
}
```

Question 1 : généraliser les classes `Arbre`, `Feuille` et `Noeud` afin de pouvoir les utiliser avec n'importe quel type de valeurs. Illustrer cela avec des valeurs de type `Integer` (exemple l'arbre `[[1,2],3]`) puis de type `String` (exemple l'arbre `[["1","2"],"3"]`).

On souhaite maintenant prendre en compte et généraliser la méthode permettant de calculer la somme des valeurs dans l'arbre:

```
public interface Arbre { // à généraliser
...
Integer somme() // à généraliser
}
```

Pour cela il est nécessaire de faire une hypothèse sur le type des valeurs : il doit être possible de sommer ces dernières (en effet, que voudrait dire sommer des voitures ou des personnes dans un arbre les contenant ?). Pour cela on va dans un premier temps définir une interface qui caractérise les objets pouvant être sommés.

Question 2 : définir une interface `Sommable` caractérisant les objets pouvant être sommés : ils ont une opération `sommer` qui prend un objet du même type qu'eux et renvoit leur somme, qui est un objet du même type.

Question 3 : définir deux classes, `Entier` et `Chaine`, encapsulant respectivement un entier et une chaine de caractères, et qui implémentent `Sommable` (pour 1, 2 et 3 on obtient 6, et pour "a", "b", et "c", on obtient "abc").

Question 4 : modifier votre généralisation des arbres afin de pouvoir intégrer la méthode généralisée pour `somme` (astuce : cela n'est possible que pour les arbres dont les valeurs sont des `Sommable`). Illustrer cela avec les arbres utilisés dans la question 1, modifiés pour prendre en compte `Entier` au lieu d'`Integer` et `Chaine` au lieu de `String`.

On souhaite maintenant prendre en compte et généraliser les méthodes permettant de calculer le minimum et le maximum des valeurs dans l'arbre et de savoir si l'arbre est trié :

```
public interface Arbre { // à généraliser
...
Integer min() // à généraliser
Integer max() // à généraliser
```

```
boolean estTrie(); // à généraliser
}
```

Question 5 : modifier votre code afin de pouvoir faire cela (astuce : Java dispose d'une interface `Comparable` qui propose une méthode `compareTo`). Illustrer cela avec les arbres utilisés dans la question 4.

Exercice 4 : Utilisation de lambdas

On dispose d'une librairie permettant de gérer des commandes (elle vous sera fournie). Un DAO est disponible pour simuler l'accès à une base de données. Pour l'utiliser :

```
public class Question4 {
    public static void main(final String[] args) {
        DAO data = DAO.instance();
        // afficher les commandes (non normalisées)
        for (Commande c : data.commandes()) {
            System.out.println(c);
        }
        // afficher les commandes (normalisées)
        for (Commande c : data.commandes()) {
            System.out.println(c.normaliser());
        }
        ...
    }
}
```

En utilisant les méthodes disponibles, réalisez les points suivants :

1. affichage des produits commandés à TVA réduite (utiliser `selectionProduit`). Vous ferez trois versions différentes (lambda, lambda avec écriture et appel de méthode, référence de méthode). Résultat attendu :

   ```
   [Masques, Gel]
   ```

2. affichage des produits commandés à TVA réduite et coûtant plus de 5€ (utiliser `selectionProduit`). Résultat attendu :

   ```
   [Masques]
   ```

3. affichage des commandes (non normalisées) de plus de 2 items (utiliser `selectionCommande`). Résultat attendu :

   ```
   [Commande
       Masques x2
       Gel x10
       Camembert x2
   ```

```
    Masques x3
]
```

4. affichage des commandes (non normalisées) contenant au moins un produit à TVA réduite commandé en plus de 2 exemplaires (utiliser `selectionCommandeSurExistanceLigne`). Résultat attendu :

```
[Commande
    Masques x2
    Gel x10
    Camembert x2
    Masques x3
]
```

5. affichage des commandes en utilisant la règle suivante pour les lignes de la commande : prix TTC = prix unitaire * (1+TVA) * quantité (utiliser `affiche`, qui permet de choisir la règle de calcul et normalise avant calcul, et non `toString`). Résultat attendu :

```
Commande
+------------+------------+-----+------------+--------+------------+
+ nom        + prix       + qté + prix ht    + tva    + prix ttc   +
+------------+------------+-----+------------+--------+------------+
+    Yaourts +       2,50 +   6 +      15,00 + 10,00% +      16,50 +
+  Camembert +       4,00 +   1 +       4,00 + 20,00% +       4,80 +
+------------+------------+-----+------------+--------+------------+
Total :      21,30

Commande
+------------+------------+-----+------------+--------+------------+
+ nom        + prix       + qté + prix ht    + tva    + prix ttc   +
+------------+------------+-----+------------+--------+------------+
+  Camembert +       4,00 +   2 +       8,00 + 20,00% +       9,60 +
+    Masques +      25,00 +   5 +     125,00 +  5,50% +     131,88 +
+        Gel +       5,00 +  10 +      50,00 +  5,50% +      52,75 +
+------------+------------+-----+------------+--------+------------+
Total :     194,23
```

6. affichage des commandes en utilisant la règle suivante : même règle que la précédente plus réduction de la valeur du prix unitaire pour chaque ligne dont la quantité est supérieure à 2 (utiliser `affiche` à nouveau). Résultat attendu :

```
Commande
+------------+------------+-----+------------+--------+------------+
+ nom        + prix       + qté + prix ht    + tva    + prix ttc   +
+------------+------------+-----+------------+--------+------------+
+    Yaourts +       2,50 +   6 +      15,00 + 10,00% +      14,00 +
+  Camembert +       4,00 +   1 +       4,00 + 20,00% +       4,80 +
```

```
+------------+------------+-----+------------+--------+------------+
Total :      18,80

Commande
+------------+------------+-----+------------+--------+------------+
+ nom        + prix       + qté + prix ht    + tva    + prix ttc   +
+------------+------------+-----+------------+--------+------------+
+  Camembert +       4,00 +   2 +       8,00 + 20,00% +       9,60 +
+    Masques +      25,00 +   5 +     125,00 +  5,50% +     106,88 +
+        Gel +       5,00 +  10 +      50,00 +  5,50% +      47,75 +
+------------+------------+-----+------------+--------+------------+
Total :     164,23
```