TD interface : POO.

Exercice 2 : Transformation de textes

On s'intéresse à des « transformations » de texte. Une opération de transformation consiste, à partir d'une chaîne de caractères initiale, à produire une nouvelle chaîne de caractères, résultat de la transformation de la chaîne initiale.

Pour représenter ces transformations on fournit l'interface suivante :

```
package text;
/**
 * Define an opration to transform a string text into another string text.
 */
public interface Transformation {
  /** apply a transformation to the input String and return the resulting transformed String
   * @param input the String to transform
   * @return the transformed String
   */
  public String transform(String input);
  /** return a description of what the transformation does
   * @return a description of what the transformation does
   */
  public String getDescription();
}
```

Q1. On souhaite disposer d'une transformation qui remplace tous les caractères majuscules de la chaîne initiale par leur minuscule et laisse les autres caractères inchangés.

Donnez le code java d'une classe `ToLowerCaseTransformation` qui permet de modéliser de telles transformations.

NB : il existe une méthode `toLowerCase()` dans la classe `String`.

```
public class ToLowerCaseTransformation implements Transformation {
    public String transform(String input) {
        String Res = "";
        for(char c : input){
            Res += c.toLowerCase();
        }
        return Res;
    public String getDescription() {
        Return "desc";
    }
}
```

Q 2 . Dans l'expression :

```
T ref = new ToLowerCaseTransformation();
```

quelles sont toutes les valeurs possibles pour le type *T* telles que ce code puisse compiler ?

T -> Object, ToLowerCaseTransformation, Transformation

Q 3 . On appelle `MultiStringTransformer` une classe qui dispose d'une méthode `multiTransform` qui prend en paramètres :

- un tableau de chaînes de caractères
- une transformation (au sens décrit ci-dessus)

et dont le résultat est le tableau des chaînes de caractères transformées par la transformation fournie.

Donnez le code java de la méthode `multiTransform`.

```
public String[] multiTransform(String[] input, Transform t){
    String[] Res = new String[input.length];
    for(int i=0; i<input.length ;i+1 ){
        Res[i] = t.transform(input[i]);
    }
    return Res;
```

Q 4 . On donne la variable suivante :

```
String[] data = new String[] { "Timoleon", "jaVa", "JRR Tolkien" };
```

Donnez une séquence de lignes de code qui :

1. crée un objet `MultiStringTransformer`,
2. définit une variable `lowers` avec pour valeur le tableau des chaînes de caractères issues de `data` transformées en minuscules,
3. définit une variable `rot13s` avec pour valeur le tableau des chaînes de catactères issues de `data` codées en *Rot-13*.

```
MultiStringTransformer t = new MultiStringTransformer();
String[] lowers = t.multiTransform(data, new ToLowerTransformation());
String[] rot13s = t.multiTransform(data, new Rot13Transformation())
```

Exercice 3 :

On s'intéresse aux expressions arithmétiques binaires telles que $1+2$, 6×7, etc.
Ces expressions sont caractérisées par deux entiers correspondant aux opérandes et un opérateur (addition et multiplication dans les exemples précédents).
L'interface `Operator` du paquetage `expression.operator` permet de définir le type des opérateurs arithmétiques :

```
package expression.operator;
public interface Operator {
  /** performs the operation of this operator with given two operands
  * @param x the left operand
  * @param y the right operand
  * @return the result of the operation
  */
  public int compute(int x, int y);
  /** get the char used to display this operator
  * @return the char used to display this operator
  */
  public char getChar();
}
```

Q1. Donnez le code d'une classe `Addition` du paquetage `expression.operator` permettant de modéliser les opérateurs correspondant à l'addition.

Le caractère d'affichage est alors «+».

On suppose des classes similaires définies pour les autres opérateurs (`Multiplication`, etc.).

```
public class Addition implements Operator {

    public int compute(int x, int y){
        return x+y;
    }

    public char getChar(){
        Return '+';
    }
}
```

Le type `BinaryExpression`, du paquetage `expression`, permet de définir des expressions binaires à partir de ses deux opérandes et de son opérateur.
Ce type dispose :

- d'une méthode `public int getValue()` qui permet d'obtenir la valeur résultat du calcul de l'expression
- d'une méthode `public String toString()` qui a pour résultat une chaîne de caractères correspondant à l'expression. Par exemple `"1+2"` ou `"6x7"` pour les expressions données précédemment comme exemple.

Q2. Donnez le code de la classe `BinaryExpression`.

Le code des méthodes accesseurs n'est pas demandé.

```
public class BinaryExpression {
    private int x;
    private int y;
    private Operator op;

    public BinaryExpression(int x, int y, Operator op) {
        this.x=x;
        this.y=y;
        this.op=op;
    }
```

```
public int getValue(){
    return this.op.compute(this.x, this.y);
}

public String toString(){
  return String.format("%d %c %d",
          this.x, this.op.getChar(), this.y);
}
```

Q 3. Donnez le code d'une méthode `main` de la classe `ExpressionMain` qui :

- crée un objet `BinaryExpression` correspondant à l'expression binaire $1 + 2$,
- crée un objet `BinaryExpression` correspondant à l'expression binaire 6×7,
- affiche le texte de chacune de ces expressions binaires et son résultat.

```
public class Main {
    public static void main(String[] args) {
        BinaryExpression b1 = new BinaryExpression(1,2, new Addition());
        BinaryExpression b2 = new BinaryExpression(6, 7, new Multiplication());
        System.out.println(String.format("%s = %d",
                b1.toString(), b1.getValue()));
        System.out.println(String.format("%s = %d",
                b2.toString(), b2.getValue()));
    }
}
```