Rapport TP Java

TP réalisé par SCHIRVANIAN Léo & M'BIDA Otmane

Version du compilateur utilisé : Eclipse 4.9.0

Nom de la classe contenant le programme principal : main

Explication du code :

La calculette est répartie en 3 classes répartie en 3 package pour correspondre au modèle MVC : « Modèle », « Contrôleur » et « Vue ». Même si nous avons compris le principe de ce modèle, nous ne sommes pas tout à fait sûr si telle méthode doit aller dans cette classe plutôt que dans une autre, nous pensons donc qu'il subsiste encore des erreurs dans l'architecture MVC.

Nous avons en intégralité répondu au diagramme UML demandé (swap, push, pop, accumulateur, calcul en polonaise inversée, …). Nous avons même rajouté quelques améliorations puisque notre calculette est capable de gérer et de calculer des nombres à taille infini et de précision infini (pas de limite de chiffre après la virgule) grâce à la librairie BigDecimal de java.

```
public class Modele {


    //ATTRIBUTS

    private Stack<BigDecimal> operandes = new Stack<BigDecimal>();
    private BigDecimal accumulateur;
    private Stack<Integer> operandesCompteur = new Stack<Integer>();
    private int compteur;
    private boolean decimal;
```

Le modèle de notre calculette se base sur les attributs suivants

- operandes est une pile de BigDecimal qui correspond aux nombres rentrés par l'utilisateur.

- accumulateur correspond aux nombres en train d'être rentré par l'utilisateur.

- compteur est le nombre de chiffre après la virgule que possède l'accumulateur.

- operandesCompteur est une pile d'entier qui correspond aux nombres de chiffres après la virgule de tous les éléments de operandes. (operandes et operandesCompteur sont synchronisés).

- decimal est un boolean qui indique si on est en « mode décimal » ou non, c'est-à-dire si l'utilisateur écrit à droite ou à gauche de la virgule.

L'utilité des attributs compteur, decimal et operandesCompteur qui n'était pas forcément demandés se retrouve dans l'application de plusieurs méthodes charnières comme ci-dessous :

```
public void actualiserAccumulateurDecimal(String str) {
    if(str == "0") {
        modele.setCompteur(modele.getCompteur()+1);
        String s = "0.";
        for (int i = 0; i < modele.getCompteur(); i++) {
            s = s.concat("0");
        }
        modele.setAccumulateur(new BigDecimal(s));

    }
    else {
        if(modele.getAccumulateur().compareTo(BigDecimal.ZERO) >= 0) {
            modele.setCompteur(modele.getCompteur()+1);
            BigDecimal d = new BigDecimal(str);
            BigDecimal bd = new BigDecimal("1");
            for(int i = 0; i<modele.getCompteur(); i++) {
                  bd = bd.divide(new BigDecimal("10"));
                }
            modele.setAccumulateur(modele.getAccumulateur().add(d.multiply(bd)));
        }
        else {
            modele.setCompteur(modele.getCompteur()+1);
            BigDecimal d = new BigDecimal(str);
            BigDecimal bd = new BigDecimal("1");
            for(int i = 0; i<modele.getCompteur(); i++) {
                  bd = bd.divide(new BigDecimal("10"));
                }
            modele.setAccumulateur(modele.getAccumulateur().subtract(d.multiply(bd)));
        }
    }
}
```

```
//traite le problème des 0.00000.. pour faire 0.05 par exemple

//si l'accumulateur est supérieur à 0

//on calcule 10^(-compteur) en BigDecimal pour avoir une très grande précision
//on fait le calcul approprié
//si l'accumulateur est supérieur à 0

//on calcule 10^(-compteur) en BigDecimal pour avoir une très grande précision
//on fait le calcul approprié
```

Compteur permet de réaliser les puissances de 10 négatives pour écrire des nombres décimaux, la méthode actualiserAccumulateurDecimal est appelé quand l'utilisateur clique sur un bouton « Chiffre » et que decimal est true (elle permet d'écrire après la virgule)

```
public void compteurResult() {
    modele.getOperandesCompteur().pop();
    String s = modele.getAccumulateur().toString();
    int i = s.indexOf(".");
    if(i>0) {
        int t = s.substring(i,s.length()-1).length();
        modele.setCompteur(t);
    }
    modele.testDecimal();      //si le compteur vaut 0, on synchronise decimal sur false sinon on synchronise sur true
}
```

Par ailleurs, operandesCompteur est très utile pour calculer le compteur d'un calcul, c'est-à-dire compter le nombre de décimal d'un résultat après calcul. Ce qui permet par exemple à l'utilisateur de pouvoir réécrire à la suite du résultat obtenu s'il possède des décimales vu que notre méthode actualiserAccumulateurDecimal décide de mettre à jour l'accumulateur avec des BigDecimal et non avec des chaînes de caractères.

```
public void swap() {
    BigDecimal chiffre1 = operandes.pop();
    operandes.push(accumulateur);
    accumulateur = chiffre1;
    int chiffre2 = operandesCompteur.pop();
    operandesCompteur.push(compteur);
    compteur = chiffre2;
    testDecimal();
}
```

```
public void testDecimal() {
    if(compteur == 0) {
        setDecimal(false);
    }
    else {
        setDecimal(true);
    }
}
```

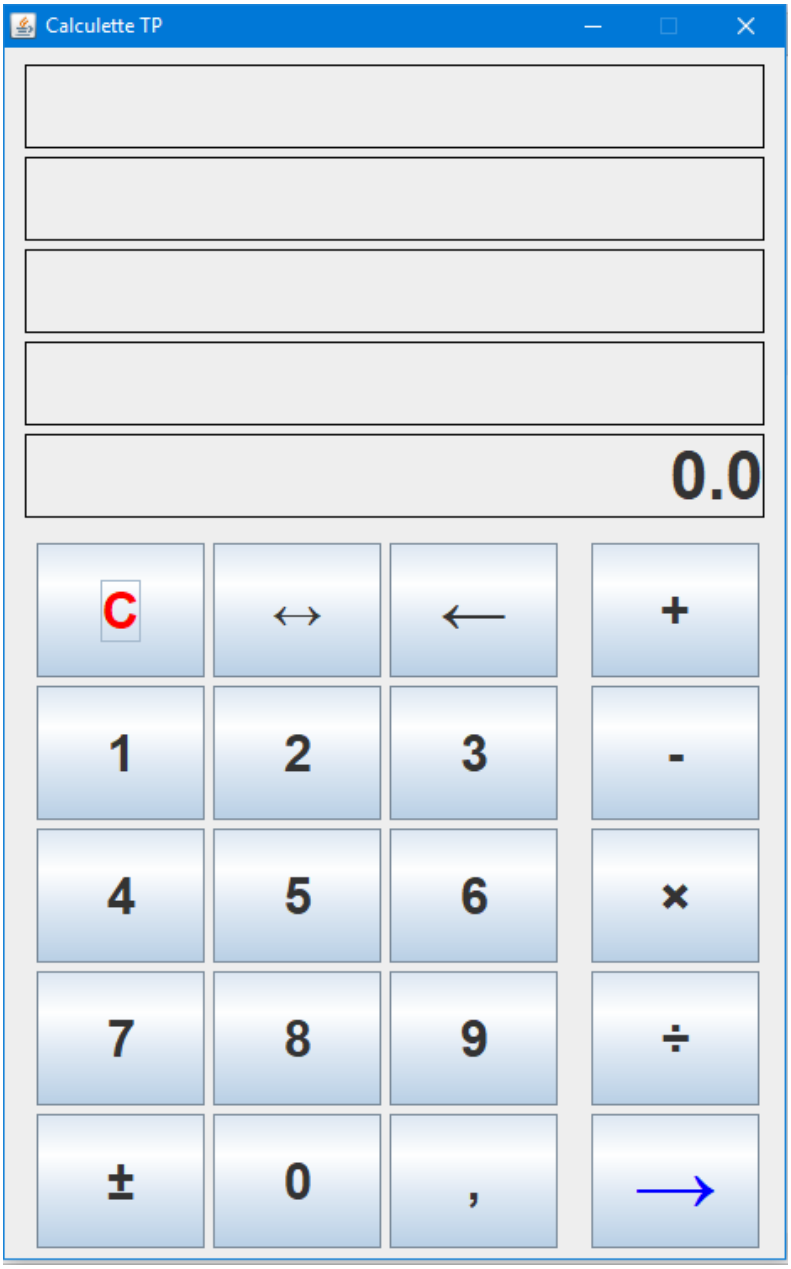
De plus, grâce à cela la méthode swap permet de réécrire sans erreur après la virgule en inversant non seulement l'accumulateur et le premier élément de la pile operandes mais aussi le compteur et le compteur du premier élément de la pile. Cela combiné à notre méthode actualiserAccumulateurDecimal évite de nombreuses erreurs qui gêneraient l'utilisateur et qui nous ont parus indispensable de corriger en rajoutant des attributs.

```
public void calcul(String str) {
    BigDecimal chiffre1 = BigDecimal.ZERO;
    BigDecimal chiffre2 = BigDecimal.ZERO;
    switch(str) {
    case "+" :
        chiffre1 = modele.getOperandes().pop();         //premier de la pile operandes
        chiffre2 = modele.getAccumulateur();            //accumulateur
        modele.setAccumulateur(chiffre1.add(chiffre2));
        modele.setDecimal(false);
        break;
    case "-" :
        chiffre1 = modele.getOperandes().pop();         //premier de la pile operandes
        chiffre2 = modele.getAccumulateur();            //accumulateur
        modele.setAccumulateur(chiffre1.subtract(chiffre2));
        modele.setDecimal(false);
        break;
    case "x" :
        chiffre1 = modele.getOperandes().pop();         //premier de la pile operandes
        chiffre2 = modele.getAccumulateur();            //accumulateur
        modele.setAccumulateur(chiffre1.multiply(chiffre2));
        modele.setDecimal(false);
        break;
    case "÷" :
        chiffre1 = modele.getOperandes().pop();         //premier de la pile operandes
        chiffre2 = modele.getAccumulateur();            //accumulateur
        modele.setAccumulateur(chiffre1.divide(chiffre2,10, BigDecimal.ROUND_HALF_UP));
        modele.setDecimal(false);
        break;
    default :
        break;
    }
}
```

La méthode de calcul ci-dessous est le centre de notre calculette, elle prend en entrée un string, la méthode reconnait d'abord l'opérateur et associe le bon calcul. Ici, on utilise les méthodes de la libraire BigDecimal pour avoir une précision optimale sur les résultats.

Nous pensons que notre calculette est plutôt fiable même si, malgré le fait qu'elle ait été soumise à une batterie de test, nous ne pouvons affirmer qu'elle soit exempte d'erreurs. Néanmoins, nous avons corrigé et même amélioré notre calculette pour qu'elle soit la plus fiable et intelligente possible.

Nous sommes aussi conscients que la librairie BigDecimal ne doit pas être la plus adaptée pour des calculs simples mais c'est le seul moyen que nous avons trouvé pour résoudre le problème de précision des nombres flottants en informatique. En effet, il est bon de rappeler que nous « calculons » notre accumulateur et que nous ne passons pas par des chaînes de caractère, ce qui posait de gros problèmes. On s'est donc basé sur la règle d'or que ce que l'utilisateur veut, le programme donne et ne propose pas autre chose, cette erreur de précision était donc inenvisageable et il a été une priorité absolue que de le corriger avec des BigDecimal.



Concernant le visuel de la calculette, nous avons essayé au maximum d'obéir au diagramme UML et au visuel montré dans le TP1 et en s'inspirant aussi de la calculatrice de base de Windows. En revanche, l'aspect technique est lui complétement respecté, il y a addition, multiplication, soustraction et division, un bouton swap, un bouton push et pop, un reset, un bouton plus-moins, une virgule pour les nombres décimaux, les 10 chiffres et évidemment elle adopte une notation polonaise inversée. Aucunes autres fonctions n'ont été rajoutés en plus de celles-ci.

Nous avons donc opté pour une calculatrice en notation polonaise inversée qui soit très fiable (avec une infinité de nombres calculables et sans limite de précision). Comme nous l'avons dit nous avons au maximum banni toutes les potentielles erreurs qu'on peut produire avec n'importe quelle combinaison de boutons mais nous ne pouvons jamais être sûr que notre calculette est parfaite. Néanmoins, nous l'avons codé et amélioré dans le but qu'elle le soit.