Haute Ecole d'Ingénierie et de Gestion

POO Laboratoire 7

Calculatrice

Sebastian Diaz & Dunant Guillaume
04/12/2023

Introduction

Le laboratoire vise à élaborer une calculatrice fonctionnant selon la notation polonaise inverse (RPN) en utilisant des classes Java distinctes. Le fichier JCalculator.java, fourni par l'assistant et l'enseignant, établit l'interface graphique d'une calculatrice encore incomplète. Pour achever sa mise en œuvre, il est nécessaire d'implémenter la classe Operator et ses sous-classes. L'accent sera mis sur la maximisation de la factorisation des opérations au sein de la hiérarchie de classes, éliminant ainsi les constructions conditionnelles explicites. La notation polonaise inverse sera basée sur une pile (Stack) que nous devrons créer. Cette approche permet des calculs sans avoir recours aux parenthèses, et la pile sera conçue de manière modulaire pour assurer une structure flexible. Enfin, la classe State gérera les états internes de la calculatrice, la représentation de la pile, ainsi que la synchronisation avec l'interface graphique.

Classe Stack

La classe Stack représente une implémentation de la structure de données de pile. Elle offre les fonctionnalités classiques d'une pile, permettant l'ajout (push) et la suppression (pop) d'éléments selon le principe LIFO. La pile est générique, ce qui signifie qu'elle peut stocker des éléments de n'importe quel type. La classe propose également des méthodes pour obtenir la taille de la pile (size), vider la pile (emptyStack), et récupérer ses éléments sous forme de tableau ou de chaîne de caractères. Un itérateur est également disponible pour parcourir les éléments de la pile.

La classe Stack est conçu de manière modulaire avec un nœud (Node) interne qui représente chaque élément de la pile. Les opérations sur la pile sont effectuées en manipulant les liens entre ces nœuds.

Classe Node

La classe Node représente les éléments individuels d'une pile. Chaque nœud contient une valeur de type générique et une référence vers le nœud précédent dans la pile. Les nœuds sont utilisés pour construire la structure de la pile, permettant ainsi une gestion efficace des éléments selon le principe Last In, First Out.

Il y a deux constructeurs qui permet de créer le nœud. Le premier en spécifiant la valeur uniquement, et l'autre en indiquant également son précédant. La classe offre des méthodes pour accéder et modifier la valeur du nœud (getValue, setValue), ainsi que pour obtenir et définir le nœud précédent (getPrevious, setPrevious).

Classe Iterator

La classe Iterator implémente un itérateur simple permettant de parcourir les nœuds d'une pile. L'itérateur est initialisé avec un nœud de départ, généralement le sommet de la pile. La méthode next permet de récupérer la prochaine valeur non encore retournée dans le parcours, en se déplaçant vers le nœud précédent. La méthode hasNext indique si une valeur suivante est disponible en vérifiant si le nœud précédent existe.

Cet itérateur offre une manière pratique de traverser les éléments de la pile sans avoir à manipuler directement la structure interne des nœuds.

Classe Operator

La classe Operator, en tant que classe abstraite centrale, offre une structure modulaire pour diverses opérations spécifiques. On distingue deux catégories principales : UnaryOperation, regroupant les opérations à une seule opérande telles que la racine carrée, la puissance et l'inverse. Puis nous avons DoubleOperation, dédiée aux opérations à deux opérandes (Addition, Subtraction, Multiplication, Division). En plus de ces catégories, d'autres sous-classes d'Operator sont présentes pour des opérations spécifiques comme le stockage en mémoire, l'effacement de caractères ou la réinitialisation de la pile

de mémoire. La classe Operator a accès à l'état interne de la calculatrice (State), ce qui lui permet d'appliquer les opérations de manière cohérente et de maintenir la synchronisation avec les données en cours de calcul.

Classe State

La classe State assume le rôle central de la gestion des états internes, intervenant à chaque sollicitation d'opération. Elle est conçue pour maintenir une représentation précise de l'état courant de la calculatrice. À l'intérieur de cette classe, on observe la présence de constantes telles que "ERROR" pour indiquer un état d'erreur, "DEFAULTVAL" pour la valeur par défaut, "NEGATE" pour le signe négatif, et "DOT" pour le point décimal. Ces constantes contribuent à une gestion uniforme des états et des opérations au sein de la classe.

La pile est l'élément fondamental pour stocker temporairement les valeurs pendant les calculs. Les opérations de la pile, telles que l'ajout et la récupération de valeurs, sont orchestrées par des méthodes telles que "pushCurrent" et "getStackValue".

L'utilisation de la mémoire offre la possibilité de stocker une valeur pour une utilisation ultérieure, illustrée par les fonctions MS (Memory Store) et MR (Memory Recall) de la JCalculator. Enfin, la classe est également équipée de méthodes pour gérer le mode d'erreur, l'entrée utilisateur, la modification de la valeur courante, et d'autres opérations spécifiques. Ces méthodes incluent "setError", "setUserInput", "appendToCurrent", et "negateCurrent".

Classe JCalculator

La classe JCalculator constitue l'interface graphique d'une calculatrice fonctionnant en notation polonaise inverse. Héritant de la classe JFrame, elle offre une mise en page structurée comprenant un champ de texte pour afficher la valeur courante et une liste pour représenter la pile de calcul. Les composants graphiques sont soigneusement positionnés grâce à la gestion des contraintes GridBagConstraints. La méthode « update » assure une mise à jour cohérente de l'interface à chaque opération, en utilisant les données provenant de la classe State. Les boutons numériques et opérateurs sont liés à des instances spécifiques d'Operator, mettant en œuvre des fonctionnalités telles que l'ajout de chiffres à la valeur courante, les opérations arithmétiques, et le changement de signe. En somme, la classe JCalculator agit comme l'interface visuelle intuitive et interactive qui permet à l'utilisateur d'interagir avec la calculatrice RPN, orchestrant les opérations à travers les mécanismes élaborés dans les autres classes.

Classe Calculator

La classe Calculator contient l'implémentation de la calculatrice en mode console. Pour l'utiliser, on peut soit entrer un nombre (décimal ou entier) pour l'ajouter sur la pile ou bien une opération (liste des opérations supportées obtenue grâce à la commande *HELP*). Pour faire le lien entre l'entrée de l'utilisateur et l'opération à exécuter, nous avons utilisé une HashMap qui permet de facilement récupérer le bon Operator grâce à un String sans utiliser un switch – case ou des if – else à la suite. Après chaque entrée de l'utilisateur (sauf *HELP*), le contenu de la pile puis la valeur courante sont affichés.

Malheureusement, une petite erreur se produit lors de l'appel des fonctions *C* ou *CE*. En effet, après avoir réinitialisé la valeur courante, elle y mette la valeur par défaut, qui est 0. Ceci est tout à fait adapter pour la version graphique de la calculatrice car l'utilisateur peut changer la valeur courante mais ce n'est pas le cas en console, ce qui a pour conséquence qu'un 0.0 apparaît sur la pile. Il faudrait modifier le comportement des Operator Clear et ClearError pour réparer cela mais nous manquons malheureusement de temps.

Classe TestStack

Le fichier “TestStack.java” est une classe de test visant à évaluer notre implémentation de notre pile. Les tests comprennent le push, le pop, si la pile est vide et l’utilisation d’un itérateur pour parcourir les éléments. Le code utilise des méthodes pour comparer le résultat attendu avec celui obtenu, affichant les succès en vert et les cas d’échec en rouge dans la console.

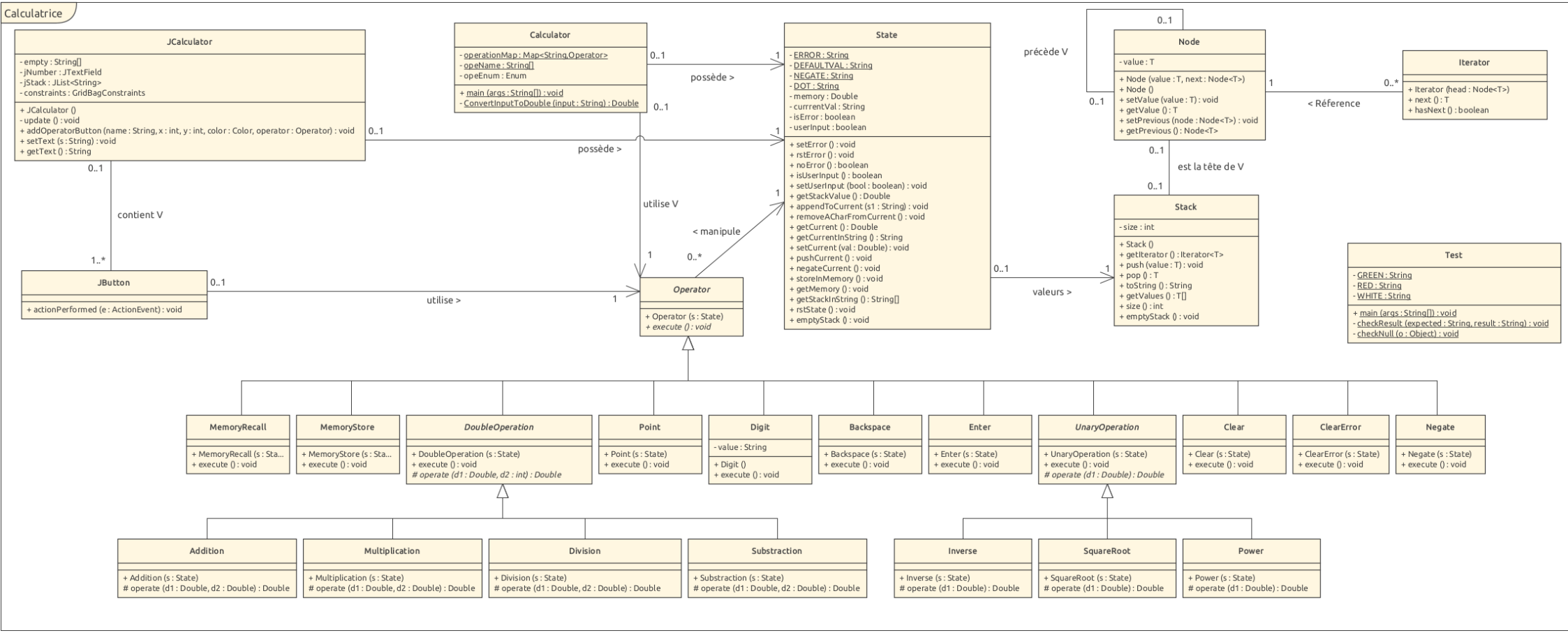
Test des programmes

Pour tester le bon fonctionnement de la calculatrice, en mode graphique ou en mode console, nous avons essayé chacune des fonctions possibles en lançant les programmes afin de vérifier que leur comportement est bien celui attendu.

Pour tester le fonctionnement de la pile, le fichier TestStack.java possède sa propre méthode main qui lancera différent test de la pile et de son itérateur.

Annexes

- Diagramme de classe
- Code source


Calculatrice
JCalculator
- empty : String[]
- jNumber : JTextField
- jStack : JList<String>
- constraints : GridBagConstraints
+ JCalculator ()
- update () : void
+ addOperatorButton (name : String, x : int, y : int, color : Color, operator : Operator) : void
+ setText (s : String) : void
+ getText () : String
0..1
contient V
1..*
JButton
+ actionPerformed (e : ActionEvent) : void
0..1
utilise >
1
Calculator
- operationMap : Map<String,Operator>
- opeName : String[]
- opeEnum : Enum
+ main (args : String[]) : void
- ConvertInputToDouble (input : String) : Double
0..1
possède >
1
0..1
possède >
1
utilise V
1
0..*
< manipule
1
State
- ERROR : String
- DEFAULTVAL : String
- NEGATE : String
- DOT : String
- memory : Double
- currrentVal : String
- isError : boolean
- userInput : boolean
+ setError () : void
+ rstError () : void
+ noError () : boolean
+ isUserInput () : boolean
+ setUserInput (bool : boolean) : void
+ getStackValue () : Double
+ appendToCurrent (s1 : String) : void
+ removeACharFromCurrent () : void
+ getCurrent () : Double
+ getCurrentInString () : String
+ setCurrent (val : Double) : void
+ pushCurrent () : void
+ negateCurrent () : void
+ storeInMemory () : void
+ getMemory () : void
+ getStackInString () : String[]
+ rstState () : void
+ emptyStack () : void
Operator
+ Operator (s : State)
+ execute () : void
0..1
valeurs >
1
précède V
0..1
0..1
Node
- value : T
+ Node (value : T, next : Node<T>)
+ Node ()
+ setValue (value : T) : void
+ getValue () : T
+ setPrevious (node : Node<T>) : void
+ getPrevious () : Node<T>
1
0..*
< Réference
Iterator
+ Iterator (head : Node<T>)
+ next () : T
+ hasNext () : boolean
0..1
est la tête de V
0..1
Stack
- size : int
+ Stack ()
+ getIterator () : Iterator<T>
+ push (value : T) : void
+ pop () : T
+ toString () : String
+ getValues () : T[]
+ size () : int
+ emptyStack () : void
Test
- GREEN : String
- RED : String
- WHITE : String
+ main (args : String[]) : void
- checkResult (expected : String, result : String) : void
- checkNull (o : Object) : void
MemoryRecall
+ MemoryRecall (s : Sta...
+ execute () : void
MemoryStore
+ MemoryStore (s : Sta...
+ execute () : void
DoubleOperation
+ DoubleOperation (s : State)
+ execute () : void
# operate (d1 : Double, d2 : int) : Double
Point
+ Point (s : State)
+ execute () : void
Digit
- value : String
+ Digit ()
+ execute () : void
Backspace
+ Backspace (s : State)
+ execute () : void
Enter
+ Enter (s : State)
+ execute () : void
UnaryOperation
+ UnaryOperation (s : State)
+ execute () : void
# operate (d1 : Double) : Double
Clear
+ Clear (s : State)
+ execute () : void
ClearError
+ ClearError (s : State)
+ execute () : void
Negate
+ Negate (s : State)
+ execute () : void
Addition
+ Addition (s : State)
# operate (d1 : Double, d2 : Double) : Double
Multiplication
+ Multiplication (s : State)
# operate (d1 : Double, d2 : Double) : Double
Division
+ Division (s : State)
# operate (d1 : Double, d2 : Double) : Double
Substraction
+ Substraction (s : State)
# operate (d1 : Double, d2 : Double) : Double
Inverse
+ Inverse (s : State)
# operate (d1 : Double) : Double
SquareRoot
+ SquareRoot (s : State)
# operate (d1 : Double) : Double
Power
+ Power (s : State)
# operate (d1 : Double) : Double

Main.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date    : 16.11.2023
 * Fichier: Main.java
 */
import calculator.JCalculator;

/**
 * Programme principale
 */
public class Main
{
  /**
   * Démarre l'interface graphique
   * @param args Arguments du programme
   */
  public static void main(String ... args) {
    new JCalculator();
  }
}

```

calculator\JCalculator.java

```java
package calculator;

import java.awt.Color;
import java.awt.Font;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.JTextField;

//import java.awt.event.*;

public class JCalculator extends JFrame
{
  // Tableau representant une pile vide
  private static final String[] empty = { "< empty stack >" };

  // Zone de texte contenant la valeur introduite ou resultat courant
  private final JTextField jNumber = new JTextField("0");

  // Composant liste representant le contenu de la pile
  private final JList<String> jStack = new JList<>(empty);

  // Contraintes pour le placement des composants graphiques
  private final GridBagConstraints constraints = new GridBagConstraints();

  private State state = new State();

  public void setText(String s){
    jNumber.setText(s);
  }

  public String getText(){
    return jNumber.getText();
  }


  // Mise a jour de l'interface apres une operation (jList et jStack)
  private void update()
  {
    jNumber.setText(state.getCurrentInString());

    String[] values = state.getStackInString();
    if(values != null){
      jStack.setListData(values);
    }
    else{
      jStack.setListData(empty);
    }
  }

```

```
57    // Ajout d'un bouton dans l'interface et de l'operation associee,
58    // instance de la classe Operation, possedeant une methode execute()
59    private void addOperatorButton(String name, int x, int y, Color color,
60                    final Operator operator)
61    {
62      JButton b = new JButton(name);
63      b.setForeground(color);
64      constraints.gridx = x;
65      constraints.gridy = y;
66      getContentPane().add(b, constraints);
67      b.addActionListener((e) -> {
68      operator.execute();
69      update();
70        });
71    }
72
73    public JCalculator()
74    {
75      super("JCalculator");
76      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
77      getContentPane().setLayout(new GridBagLayout());
78
79      // Contraintes des composants graphiques
80      constraints.insets = new Insets(3, 3, 3, 3);
81      constraints.fill = GridBagConstraints.HORIZONTAL;
82
83      // Nombre courant
84      jNumber.setEditable(false);
85      jNumber.setBackground(Color.WHITE);
86      jNumber.setHorizontalAlignment(JTextField.RIGHT);
87      constraints.gridx = 0;
88      constraints.gridy = 0;
89      constraints.gridwidth = 5;
90      getContentPane().add(jNumber, constraints);
91      constraints.gridwidth = 1; // reset width
92
93      // Rappel de la valeur en memoire
94      addOperatorButton("MR", 0, 1, Color.RED, new MemoryRecall(state));
95
96      // Stockage d'une valeur en memoire
97      addOperatorButton("MS", 1, 1, Color.RED, new MemoryStore(state));
98
99      // Backspace
100     addOperatorButton("<=", 2, 1, Color.RED, new Backspace(state));
101
102     // Mise a zero de la valeur courante + suppression des erreurs
103     addOperatorButton("CE", 3, 1, Color.RED, new ClearError(state));
104
105     // Comme CE + vide la pile
106     addOperatorButton("C",  4, 1, Color.RED, new Clear(state));
107
108     // Boutons 1-9
109     for (int i = 1; i < 10; i++)
110       addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
111             Color.BLUE, new Digit(state, i));
112     // Bouton 0
113     addOperatorButton("0", 0, 5, Color.BLUE, new Digit(state, 0) );
114
115     // Changement de signe de la valeur courante
```

```
116     addOperatorButton("+/-", 1, 5, Color.BLUE, new Negate(state));
117
118     // Operateur point (chiffres apres la virgule ensuite)
119     addOperatorButton(".", 2, 5, Color.BLUE, new Point(state));
120
121     // Operateurs arithmetiques a deux operandes: /, *, -, +
122     addOperatorButton("/", 3, 2, Color.RED, new Division(state));
123     addOperatorButton("*", 3, 3, Color.RED, new Multiplication(state));
124     addOperatorButton("-", 3, 4, Color.RED, new Subtraction(state));
125     addOperatorButton("+", 3, 5, Color.RED, new Addition(state));
126
127     // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
128     addOperatorButton("1/x", 4, 2, Color.RED, new Inverse(state));
129     addOperatorButton("x^2", 4, 3, Color.RED, new Power(state));
130     addOperatorButton("Sqrt", 4, 4, Color.RED, new SquareRoot(state));
131
132     // Entree: met la valeur courante sur le sommet de la pile
133     addOperatorButton("Ent", 4, 5, Color.RED, new Enter(state));
134
135     // Affichage de la pile
136     JLabel jLabel = new JLabel("Stack");
137     jLabel.setFont(new Font("Dialog", 0, 12));
138     jLabel.setHorizontalAlignment(JLabel.CENTER);
139     constraints.gridx = 5;
140     constraints.gridy = 0;
141     getContentPane().add(jLabel, constraints);
142
143     jStack.setFont(new Font("Dialog", 0, 12));
144     jStack.setVisibleRowCount(8);
145     JScrollPane scrollPane = new JScrollPane(jStack);
146     constraints.gridx = 5;
147     constraints.gridy = 1;
148     constraints.gridheight = 5;
149     getContentPane().add(scrollPane, constraints);
150     constraints.gridheight = 1; // reset height
151
152     setResizable(false);
153     pack();
154     setVisible(true);
155   }
156 }
157
```

calculator\Operator.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date   : 16.11.2023
 * Fichier: Operator.java
 */
package calculator;

/**
 * Classe abstraite permettant de représenter
 *  une opération de la caluclatrice à effectuer
 */
abstract class Operator {
    protected State state;
    /**
     * Méthode qui sera appelée pour effectuer l'opération
     */
    abstract void execute();

    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Operator(State s){
        state = s;
    }
}

/**
 * Opération rajoutant un chiffre à la valeur courante
 */
class Digit extends Operator {
    private String value;
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     * @param val Valeur qui sera ajoutée à la valeur courante
     */
    public Digit(State s, int val){
        super(s);
        value = "" + val;
    }

    public void execute() {
        if(state.noError()){
            if(!state.isUserInput()) {
                state.pushCurrent();
                state.setUserInput(true);
            }
            state.appendToCurrent(value);
        }
    }
}

/**
 * Opération ajouant un point à la valeur courante
 */
```

```java
class Point extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Point(State s) {
        super(s);
    }

    public void execute() {
        if (state.noError() && state.isUserInput()) {
            state.appendToCurrent(".");
        }
    }
}

/**
 * Opération permettant d'enlevé un caractère de la valeur courante
 */
class Backspace extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Backspace(State s) {
        super(s);
    }

    public void execute() {
        if(state.noError() && state.isUserInput()){
            state.removeACharFromCurrent();
        }
    }
}

/**
 * Classe abstraite représentant les opérations nécessitant deux opérandes
 */
abstract class DoubleOperation extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public DoubleOperation(State s) {
        super(s);
    }

    /**
     * Opération qui sera effectuée entre les deux valeurs
     * @param d1 Valeur 1
     * @param d2 Valeur 2
     * @return Résultat de l'opération
     */
    abstract protected Double operate(Double d1, Double d2);

    public void execute() {
        if(state.noError()){
            Double d1 = state.getCurrent();
            Double d2 = state.getStackValue();
```

```
116
117             if(d1 != null && d2 != null){
118                 state.setCurrent(operate(d1, d2));
119                 state.setUserInput(false);
120             }
121             else{
122                 state.setError();
123             }
124         }
125     }
126 }
127
128 /**
129  * Opération d'addition
130  */
131 class Addition extends DoubleOperation {
132     /**
133      * Constructeur
134      * @param s Etat interne de la calculatrice
135      */
136     public Addition(State s) {
137         super(s);
138     }
139
140     protected Double operate(Double d1, Double d2) {
141         return d1 + d2;
142     }
143 }
144
145 /**
146  * Opération de soustraction
147  */
148 class Subtraction extends DoubleOperation {
149     /**
150      * Constructeur
151      * @param s Etat interne de la calculatrice
152      */
153     public Subtraction(State s) {
154         super(s);
155     }
156
157     protected Double operate(Double d1, Double d2) {
158         return d1 - d2;
159     }
160
161 }
162
163 /**
164  * Opération de multiplication
165  */
166 class Multiplication extends DoubleOperation {
167     /**
168      * Constructeur
169      * @param s Etat interne de la calculatrice
170      */
171     public Multiplication(State s) {
172         super(s);
173     }
174
```

```
175     protected Double operate(Double d1, Double d2) {
176         return d1 * d2;
177     }
178 }
179 
180 /**
181  * Opération de division
182  */
183 class Division extends DoubleOperation {
184     /**
185      * Constructeur
186      * @param s Etat interne de la calculatrice
187      */
188     public Division(State s) {
189         super(s);
190     }
191 
192     protected Double operate(Double d1, Double d2) {
193         return d2 / d1;
194     }
195 }
196 
197 /**
198  * Classe abstraite représentant les opérations nécessitant une seule opérande
199  */
200 abstract class UnaryOperation extends Operator {
201     /**
202      * Constructeur
203      * @param s Etat interne de la calculatrice
204      */
205     public UnaryOperation(State s) {
206         super(s);
207     }
208 
209     /**
210      * Operation qui sera effectuée sur la valeur courante
211      * @param d1 Valeur qui sera modifiée
212      * @return Nouvelle valeur
213      */
214     abstract protected Double operate(Double d1);
215 
216     public void execute() {
217         if(state.noError()){
218             Double d1 = state.getCurrent();
219             if(d1 != null){
220                 state.setCurrent(operate(d1));
221                 state.setUserInput(false);
222             }
223             else{
224                 state.setError();
225             }
226         }
227     }
228 }
229 
230 /**
231  * Opération pour retourner la racine carrée de la valeur courante
232  */
233 class SquareRoot extends UnaryOperation {
```

```
234     /**
235      * Constructeur
236      * @param s Etat interne de la calculatrice
237      */
238     public SquareRoot(State s) {
239         super(s);
240     }
241 
242     protected Double operate(Double d1) {
243         return Math.sqrt(d1);
244     }
245 }
246 
247 /**
248  * Opération pour retourner le carré de la valeur courante
249  */
250 class Power extends UnaryOperation {
251     /**
252      * Constructeur
253      * @param s Etat interne de la calculatrice
254      */
255     public Power(State s) {
256         super(s);
257     }
258 
259     protected Double operate(Double d1) {
260         return d1 * d1;
261     }
262 }
263 
264 /**
265  * Opération pour changer le signe de la valeur courante
266  */
267 class Negate extends Operator{
268     /**
269      * Constructeur
270      * @param s Etat interne de la calculatrice
271      */
272     public Negate(State s) {
273         super(s);
274     }
275 
276     public void execute(){
277         if(state.noError()){
278             if(state.isUserInput()){
279                 state.negateCurrent();
280             }
281             else{
282                 state.setCurrent(-state.getCurrent());
283             }
284         }
285     }
286 }
287 
288 /**
289  * Opération pour obtenir l'inverse de la valeur courante
290  */
291 class Inverse extends UnaryOperation {
292     /**
```

```
293      * Constructeur
294      * @param s Etat interne de la calculatrice
295      */
296     public Inverse(State s) {
297         super(s);
298     }
299 
300     protected Double operate(Double d1) {
301         return 1 / d1;
302     }
303 }
304 
305 /**
306  * Opération pour réinitialiser la valeur courante et la stack
307  */
308 class Clear extends Operator {
309     /**
310      * Constructeur
311      * @param s Etat interne de la calculatrice
312      */
313     public Clear(State s) {
314         super(s);
315     }
316 
317     public void execute() {
318         state.emptyStack();
319         state.rstState();
320     }
321 }
322 
323 /**
324  * Opération pour réinitialiser la valeur courante
325  */
326 class ClearError extends Operator {
327     /**
328      * Constructeur
329      * @param s Etat interne de la calculatrice
330      */
331     public ClearError(State s) {
332         super(s);
333     }
334 
335     public void execute() {
336         state.rstState();
337     }
338 }
339 
340 /**
341  * Opération pour stocker une valeur en mémoire
342  */
343 class MemoryStore extends Operator {
344    /**
345      * Constructeur
346      * @param s Etat interne de la calculatrice
347      */
348     public MemoryStore(State s) {
349         super(s);
350     }
351 
```

```
352     public void execute() {
353         if(state.noError()){
354             state.storeInMemory();
355             state.setUserInput(true);
356         }
357     }
358 }
359 
360 /**
361  * Opération pour récupérer la valeur stockée en mémoire
362  */
363 class MemoryRecall extends Operator {
364     /**
365      * Constructeur
366      * @param s Etat interne de la calculatrice
367      */
368     public MemoryRecall(State s) {
369         super(s);
370     }
371 
372     public void execute() {
373         if(state.noError()){
374             state.getMemory();
375             state.setUserInput(false);
376         }
377 
378     }
379 }
380 
381 /**
382  * Opération pour ajouter la valeur courante sur la stack
383  */
384 class Enter extends Operator {
385     /**
386      * Constructeur
387      * @param s Etat interne de la calculatrice
388      */
389     public Enter(State s) {
390         super(s);
391     }
392 
393     public void execute() {
394         if(state.noError()){
395             state.pushCurrent();
396             state.setUserInput(true);
397         }
398     }
399 }
```

calculator\State.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date    : 16.11.2023
 * Fichier: State.java
 */
package calculator;

import util.Iterator;
import util.Stack;

/**
 * Classe servant à représenter l'état interne
 * de la calculatrice
 */
public class State {
    private static final String ERROR = "# error #";
    private static final String DEFAULTVAL = "0";
    private static final String NEGATE = "-";
    private static final String DOT = ".";

    private Stack<Double> stack = new Stack<>();
    private Double memory = 0.;

    private String currentVal = DEFAULTVAL;
    private boolean isError = false;
    private boolean userInput = true;

    /**
     * Passe la calculatrice en mode erreur
     */
    public void setError() {
        isError = true;
        currentVal = ERROR;
    }

    /**
     * Enlève le mode erreur de la calculatrice
     */
    public void rstError() {
        isError = false;
        currentVal = DEFAULTVAL;
    }

    /**
     * Obtient l'état du erreur
     * @return true s'il la calculatrice
     * n'est pas en mode erreur
     */
    public boolean noError() {
        return !isError;
    }

    /**
     * Obtient si la valeur courante est une entrée de l'utilisateur
     * ou le résultat d'une opération
     * @return true si c'est ue entrée de l'utilisateur
```

```
57          */
58         public boolean isUserInput(){
59             return userInput;
60         }
61 
62         /**
63          * Change l'état de userInput
64          * @param bool Nouvel état
65          */
66         public void setUserInput(boolean bool){
67             userInput = bool;
68         }
69 
70         /**
71          * Obtient la valeur du dessus de la stack
72          * @return Double
73          */
74         public Double getStackValue(){
75             return stack.pop();
76         }
77 
78         /**
79          * Ajoute un String à la fin de la valeur courante
80          * @param s1 String à ajouter
81          */
82         public void appendToCurrent(String s1) {
83             if (currentVal == null || currentVal.equals(DEFAULTVAL)) {
84                 currentVal = s1;
85             } else if(s1.equals(DOT)) {
86                 if (!currentVal.contains(DOT)) {
87                     currentVal += s1;
88                 }
89             }
90             else {
91                 currentVal += s1;
92             }
93         }
94 
95         /**
96          * Enlève le dernier caractère de la valeur courante
97          */
98         public void removeACharFromCurrent(){
99             currentVal = currentVal.substring(0, currentVal.length() - 1);
100        }
101 
102        /**
103         * Obtient la valeur courante en Double
104         * @return La valeur courante ou null si l'état
105         * est en mode erreur
106         */
107        public Double getCurrent(){
108            if(currentVal.equals(ERROR)){
109                return null;
110            }
111            else{
112                return Double.parseDouble(currentVal);
113            }
114        }
115 
```

```java
    /**
     * Obtient la valeur courante en String
     * @return String
     */
    public String getCurrentInString(){
        return currentVal;
    }

    /**
     * Défini une nouvelle valeur courante
     * @param val Nouvelle valeur
     */
    public void setCurrent(Double val){
        currentVal = val.toString();
    }

    /**
     * Déplace la valeur courante sur la stack
     */
    public void pushCurrent(){
        stack.push(Double.parseDouble(currentVal));
        currentVal = DEFAULTVAL;
    }

    /**
     * Inverse le signe de la valeur courante
     */
    public void negateCurrent(){
        if(currentVal.contains(NEGATE)){
            currentVal = currentVal.substring(1, currentVal.length());
        }
        else{
            currentVal = NEGATE + currentVal;
        }
    }

    /**
     * Stock la valeur courante dans la mémoire
     */
    public void storeInMemory(){
        memory = getCurrent();
        currentVal = DEFAULTVAL;
    }

    /**
     * Modifie la valeur courante par la valeur en mémoire
     */
    public void getMemory(){
        setCurrent(memory);
    }

    /**
     * Retourne les valeurs dans la stack au format String
     * @return String[] contenant les valeurs ou null si la stack est vide
     */
    public String[] getStackInString(){
        if(stack.size() == 0){
            return null;
        }
```

```
175
176         String[] stringStack = new String[stack.size()];
177         Iterator<Double> i = stack.getIterator();
178         int counter = stack.size();
179
180         while(i.hasNext()){
181             stringStack[--counter] = i.next().toString();
182         }
183
184         return stringStack;
185     }
186
187     /**
188      * Réinitialise l'état interne
189      */
190     public void rstState(){
191         rstError();
192         currentVal = DEFAULTVAL;
193         setUserInput(true);
194     }
195
196     /**
197      * Vide la stack
198      */
199     public void emptyStack(){
200         stack.emptyStack();
201     }
202 }
203
```

util\Stack.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date    : 16.11.2023
 * Fichier: Stack.java
 */
package util;

import java.lang.reflect.Array;
import java.util.Arrays;

/**
 * Implémentation de la pile
 */
public class Stack<T> {

    Node<T> head;
    int size = 0;

    /**
     * Constructeur
     */
    public Stack(){
        head = null;
    }

    /**
     * Retourne un itérateur pour parcourir la stack
     * @return Iterator<T>
     */
    public Iterator<T> getIterator(){
        Iterator<T> ite = new Iterator<>(head);
        return ite;
    }

    /**
     * Ajoute une valeur sur la pile
     * @param value Valeur à ajouter
     */
    public void push(T value){
        if (head == null) {
            head = new Node<T>(value);
        }
        else{
            head = new Node<T>(value, head);
        }
        ++size;
    }

    /**
     * Retourne et retire la dernière valeur de la stack
     * @return La valeur retirée
     */
    public T pop(){

        if(head == null){
            return null;
```

```
57          }
58          Node<T> top = head;
59          head = head.getPrevious();
60          --size;
61          return top.getValue();
62      }
63
64      /**
65       * Retourne les valeurs de la stacks en String
66       */
67      public String toString(){
68          T[] values = getValues();
69
70          if (values == null) {
71              return "[]";
72          }
73          else{
74              return Arrays.toString(getValues());
75          }
76      }
77
78      /**
79       * Retourne un tableau contenant les valeurs de la stacks
80       * @return T[] ou null si la stack est vide
81       */
82      public T[] getValues(){
83
84          if(head == null){
85              return null;
86          }
87
88          T[] values = (T[]) Array.newInstance(head.getValue().getClass(), size);
89          int counter = 0;
90          Iterator<T> ite = getIterator();
91          while(ite.hasNext()){
92              values[counter++] = ite.next();
93          }
94          return values;
95      }
96
97      /**
98       * Retourne le nombre d'élément de la stack
99       * @return int
100      */
101     public int size(){
102         return size;
103     }
104
105     /**
106      * Retire tous les éléments de la stack
107      */
108     public void emptyStack(){
109         head = null;
110         size = 0;
111     }
112 }
113
```

util\Node.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date    : 16.11.2023
 * Fichier: Node.java
 */
package util;

/**
 * Classe représentant les éléments d'une stack
 */
class Node<T> {
    private T value;
    private Node<T> previous;

    /**
     * Constructeur avec le noeud précédent null
     * @param value Valeur stockée dans le noeud
     */
    public Node(T value){
        this.value = value;
        previous = null;
    }

    /**
     * Constucteur
     * @param value Valeur stockée dans le noeud
     * @param next Noeud précédent
     */
    public Node(T value, Node<T> next){
        this(value);
        this.previous = next;
    }

    /**
     * Modifie la valeur du noeud
     * @param value Nouvelle valeur
     */
    public void setValue(T value){
        this.value = value;
    }

    /**
     * Retourne la valeur contenue dans le noeud
     * @return T
     */
    public T getValue() {
        return value;
    }

    /**
     * Modifie le noeud précédent référencé
     * @param node Nouveau noeud
     */
    public void setPrevious(Node<T> node){
        previous = node;
    }
```

```java

    /**
     * Retourne le noeud précédant
     * @return Node<T>
     */
    public Node<T> getPrevious(){
        return previous;
    }
}

```

util\Iterator.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date    : 16.11.2023
 * Fichier: Iterator.java
 */
package util;

/**
 * Iterateur simple permettant de parcourir une suite de noeud
 */
public class Iterator<T>{
    private Node<T> value;

    /**
     * Constructeur
     * @param head Neud à partir du quel commencer
     */
    public Iterator(Node<T> head){
        value = new Node<T>(null, head);
    }

    /**
     * Retourne la prochaine valeur non retournée
     * @return T
     */
    public T next(){

        if(!hasNext()){
            return null;
        }
        else{
            value = value.getPrevious();
            return value.getValue();
        }
    }

    /**
     * Définit si une aleur suivante est disponible
     * @return true si une valeur est disponible
     */
    public boolean hasNext(){
        return (value.getPrevious() != null);
    }
}
```

calculator\Calculator.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date   : 04.12.2023
 * Fichier: Calculator.java
 */

package calculator;

import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

 /**
  * Calculatrice en mode console
  */
public class Calculator {
    private static State state;
    private static Map<String,Operator> operationsMap;
    private static final String[] opeName =
        {"+", "-", "*", "/", "POW", "SQRT", "NEG",
        "INV", "MS", "MR", "C", "CE", "HELP", "EXIT"};

    private enum opeEnum {ADD, SUB, MULT, DIV, POW, SQRT,
         NEG, INV, MS, MR, C, CE, HELP, EXIT};

    /**
     * Converti un string en Double
     * @param input String à convertir
     * @return Double ou null si la conversion a échoué
     */
    private static Double convertInputToDouble(String input){
        try{
            return Double.parseDouble(input);
        }
        catch(NumberFormatException e){
            return null;
        }
    }

    /**
     * Programme principale pour la calculatrice en mode console
     * @param args Arguments de lancement
     */
    public static void main(String[] args) {
        //Affichage titre
        System.out.println("****************************\n" +
                           "*        Calculator        *\n" +
                           "****************************\n");

        System.out.printf("%s pour afficher les opérations ou %s pour quitter\n\n",
opeName[12], opeName[13]);

        String input;
        Double val;
        Scanner scanner = new Scanner(System.in);
        boolean firstVal = true;
```

```
56
57          while (true) {
58
59              //Récupère l'entrée de l'utilisateur
60              System.out.print("> ");
61              input = scanner.nextLine();
62
63              //Si l'input est un nombre
64              if((val = convertInputToDouble(input)) != null){
65                  if(firstVal){
66                      firstVal = false;
67                  }
68                  else{
69                      state.pushCurrent();
70                  }
71                  state.setCurrent(val);
72              }
73              else{
74                  input = input.toUpperCase();
75                  Operator ope = operationsMap.get(input);
76
77                  //Si l'input est une opération
78                  if (ope != null) {
79                      ope.execute();
80                  }
81                  //Si EXIT
82                  else if(input.equals(opeName[opeEnum.EXIT.ordinal()])){
83                      break;
84                  }
85                  //Si HELP
86                  else if(input.equals(opeName[opeEnum.HELP.ordinal()])){
87                      for(String s : opeName){
88                          System.out.println(s);
89                      }
90                      continue;
91                  }
92                  //Input non reconnu
93                  else{
94                      System.out.println("Opération inconnue");
95                  }
96              }
97
98              //Affiche les valeurs contenues dans la stack et la valeur courante
99              String[] stackStrings = state.getStackInString();
100             if(stackStrings != null){
101                 for(String s : stackStrings){
102                     System.out.print(s + " ");
103                 }
104             }
105             System.out.println(state.getCurrentInString());
106         }
107
108         scanner.close();
109     }
110
111     static{
112         state = new State();
113         operationsMap = new HashMap<>();
114         operationsMap.put(opeName[opeEnum.ADD.ordinal()], new Addition(state));
```

```
        operationsMap.put(opeName[opeEnum.SUB.ordinal()], new Subtraction(state));
        operationsMap.put(opeName[opeEnum.MULT.ordinal()], new Multiplication(state));
        operationsMap.put(opeName[opeEnum.DIV.ordinal()], new Division(state));
        operationsMap.put(opeName[opeEnum.POW.ordinal()], new Power(state));
        operationsMap.put(opeName[opeEnum.SQRT.ordinal()], new SquareRoot(state));
        operationsMap.put(opeName[opeEnum.NEG.ordinal()], new Negate(state));
        operationsMap.put(opeName[opeEnum.INV.ordinal()], new Inverse(state));
        operationsMap.put(opeName[opeEnum.MS.ordinal()], new MemoryStore(state));
        operationsMap.put(opeName[opeEnum.MR.ordinal()], new MemoryRecall(state));
        operationsMap.put(opeName[opeEnum.C.ordinal()], new Clear(state));
        operationsMap.put(opeName[opeEnum.CE.ordinal()], new ClearError(state));
    }
}

```

TestStack.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date   : 04.12.2023
 * Fichier: TestStack.java
 */

import java.util.ArrayList;
import util.Iterator;
import util.Stack;

/**
 * Classe pour tester le bon fonctionnement de l'implémentation de la stack
 */
public class TestStack {

    //Code ANSI pour afficher en couleur dans la console
    private static final String GREEN = "\u001B[32m";
    private static final String RED   = "\u001B[31m";
    private static final String WHITE = "\u001B[37m";

    /**
     * Vérifie si la valeur attendue et la valeur obtenue sont égales
     * @param expected Valeur attendue
     * @param result Valeur obtenue
     */
    private static void checkResult(String expected, String result){
        System.out.println("Valeur attendue: " + expected);
        System.out.println("Valeur obtenue : " + result);
        System.out.println("Résultat: " +
            (expected.equals(result)? (GREEN + "Réussi") : (RED + "Echoué"))
            + WHITE + "\n");
    }

    /**
     * Vérifie si l'objet passé en paramètre est null
     * @param o Objet à vérifier
     */
    private static void checkNull(Object o){
        System.out.println("Valeur attendue: null");
        System.out.println("Valeur obtenue : " + (o == null? "null" : o.toString()));
        System.out.println("Résultat: " +
            (o == null? (GREEN + "Réussi") : (RED + "Echoué"))
            + WHITE + "\n");
    }

    /**
     * Programme principal de test
     * @param args Arguments du programme
     */
    public static void main(String[] args) {
        System.out.println("**Programme de test de la stack**\n");

        //Test ajout d'éléments
        System.out.println("Test Stack.push()");

        Stack<Integer> stck = new Stack<>();
```

```java
        String expectedResult = "[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]";
        String result = "";

        for(int i = 1; i <= 10; ++i){
            stck.push(i);
        }
        result = stck.toString();

        checkResult(expectedResult, result);

        //Test récupération des élémemts
        System.out.println("Test Stack.pop()");
        ArrayList<Integer> resultArray = new ArrayList<>();
        for(int i = 1; i <= 10; ++i){
            resultArray.add(stck.pop());
        }

        checkResult(expectedResult, resultArray.toString());

        //Test que la stack soit bien vide
        System.out.println("Vérifictation que la stack est vide");
        expectedResult = "[]";

        checkResult(expectedResult, stck.toString());

        //Test de Stack.pop() sur une stack vide
        System.out.println("Test de Stack.pop() sur une stack vide");

        checkNull(stck.pop());

        //Test de l'itérateur
        System.out.println("Test de l'itérateur");
        resultArray = new ArrayList<>();
        expectedResult = "[9, -32, 45, 21]";
        stck.push(21);
        stck.push(45);
        stck.push(-32);
        stck.push(9);

        Iterator<Integer> ite = stck.getIterator();
        while (ite.hasNext()) {
            resultArray.add(ite.next());
        }

        checkResult(expectedResult, resultArray.toString());

        //Test itérateur sur une stack vide
        System.out.println("Test itérateur sur une stack vide");
        stck = new Stack<>();
        ite = stck.getIterator();

        System.out.println("Iterator.hasNext(): " + ite.hasNext());
        checkNull(ite.next());
    }
}

```