POO - Laboratoire 7

Calculatrice

Lestiboudois Maxime & Parisod Nathan

27/11/2024

Contents

Introduction

Dans ce laboratoire, nous avons implémenté la logique d'une calculatrice utilisant la notation polonaise inversée. L'interface nous étant déjà fournie, nous nous sommes concentrés sur l'aspect fonctionnel de la calculatrice.
Nous avons également implémenter une extension de notre calculatrice afin de pouvoir l'utiliser directement dans un terminal.

Cahier des charges

Objectif du Projet

Créer une calculatrice fonctionnant en notation polonaise inverse (Reverse Polish Notation - RPN). Elle doit inclure une interface graphique (GUI) et un mode console, réutilisant les mêmes classes et principes. Le projet doit suivre une architecture Modèle-Vue-Contrôleur (MVC).

Spécifications Fonctionnelles

Interface Graphique (GUI)

- Implémenter une interface utilisateur dans la classe JCalculator.
- Boutons requis :
  - Chiffres : 0-9.
  - Opérations unaires : sqrt, 1/x, `x^2`.
  - Opérations binaires : +, -, *, /.
  - Autres boutons :
    - MR : récupérer une valeur en mémoire.
    - MS : stocker une valeur en mémoire.
    - `<=` : backspace pour supprimer le dernier caractère.
    - CE : réinitialiser l'affichage.
    - C : réinitialiser l'affichage et vider la pile.
    - Ent : placer la valeur courante sur la pile.
    - +/- : changer le signe de la valeur courante.
    - . : pour les nombres décimaux.
- Affichage :
  - Zone de texte (JTextField) pour la valeur courante.
  - Liste (JList) pour visualiser la pile.
- Mise à jour de l'affichage après chaque opération via une méthode update().

Mode Console

- Développer une classe Calculator permettant une interaction textuelle.
- Commandes utilisateur :
  - Saisir un nombre pour l'ajouter à la pile.
  - Entrer une opération (+, sqrt, etc.).
  - exit : quitter la calculatrice.
- Fonctionnalités similaires à la version graphique :
  - Gestion de la pile.
  - Support des opérations unaires, binaires et de mémoire.

Gestion des Données et Opérations

- Pile personnalisée (Stack) :
  - Empiler une valeur.
  - Désempiler une valeur.
  - Obtenir l'état actuel de la pile sous forme de tableau.
  - Fournir un itérateur pour parcourir la pile.
  - Implémentée avec une liste chaînée sans structures Java préconstruites.
- État interne (State) :
  - Stocker :
    - Valeur courante.
    - Pile des valeurs.
    - État d'erreur.
  - Fournir des méthodes pour gérer et manipuler ces données.

Spécifications Techniques

Architecture MVC

- Modèle (State) :
  - Indépendant de l'interface graphique.
  - Stocke les données et implémente la logique de calcul.
- Vue (JCalculator) :
  - Interface utilisateur graphique basée sur Swing.
  - Réagit aux changements dans le modèle.

- Contrôleurs (Operator) :
  - Boutons dans l'interface graphique agissant comme des contrôleurs.
  - Appel de la méthode execute() d'un objet Operator.

Hiérarchie des Classes

- Classe générique Stack pour représenter une pile.
- Classe State pour l'état interne.
- Hiérarchie Operator :
  - Classe de base Operator :
    - Méthode execute().
  - Sous-classes spécialisées :
    - NumberOperator, AdditionOperator, SqrtOperator, etc.
- Classe JCalculator pour l'interface graphique.
- Classe Calculator pour le mode console.

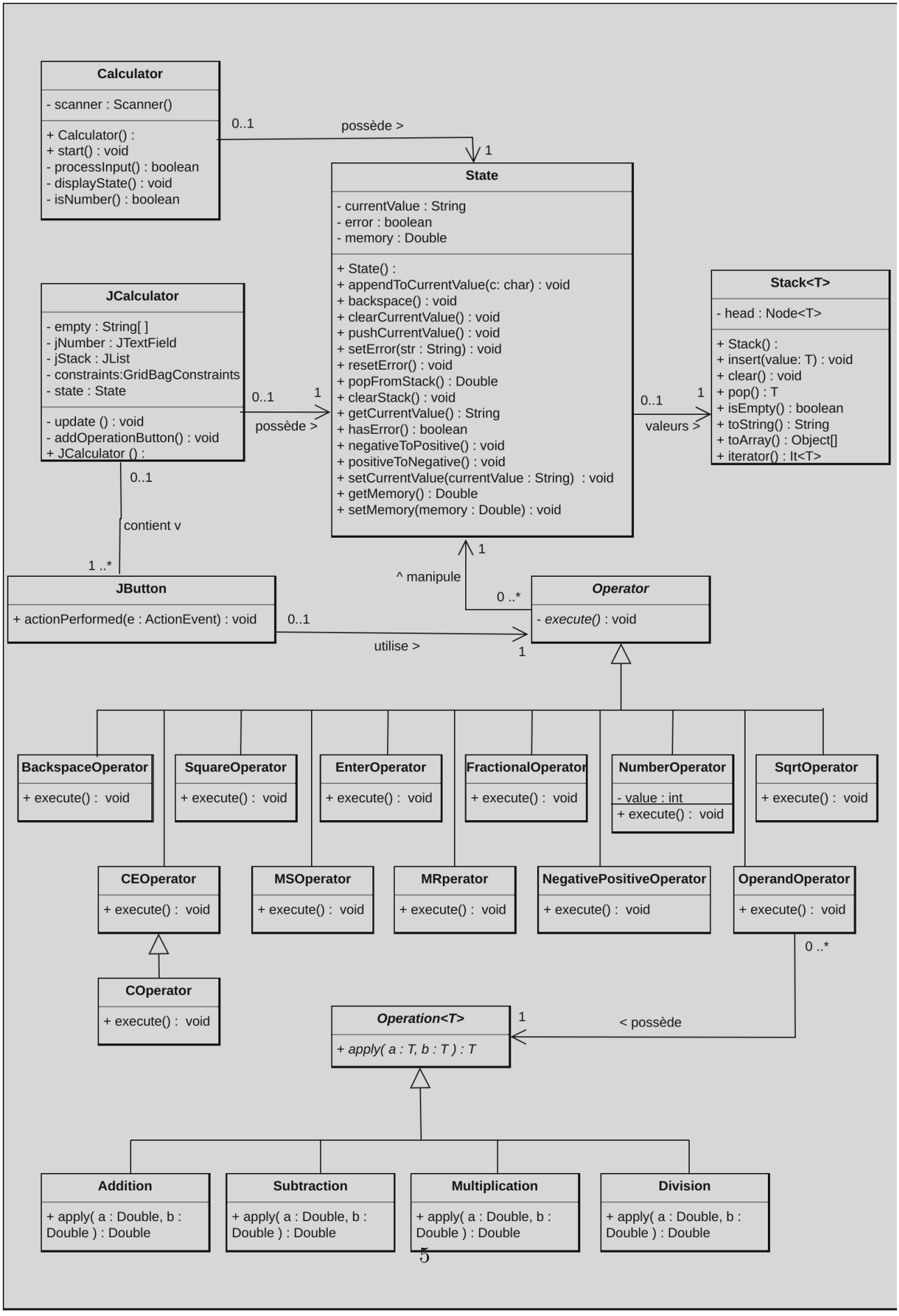
Contraintes

- Aucune utilisation de switch ou de if pour sélectionner l'opération dans Operator.
- Pas de propriétés statiques pour le stockage des données.
- Le code doit être modulaire et réutilisable.
- Respect des principes de conception objet.

Tests

- Élaborer une grille de tests couvrant :
  - Les opérations unaires et binaires.
  - Les erreurs (ex. : division par zéro, pile vide).
  - Le stockage et rappel de mémoire.
- Inclure des cas limites et des tests d'intégration.

Schéma UML


Calculator
- scanner : Scanner()
+ Calculator() :
+ start() : void
- processInput() : boolean
- displayState() : void
- isNumber() : boolean
0..1
possède >
1
State
- currentValue : String
- error : boolean
- memory : Double
+ State() :
+ appendToCurrentValue(c: char) : void
+ backspace() : void
+ clearCurrentValue() : void
+ pushCurrentValue() : void
+ setError(str : String) : void
+ resetError() : void
+ popFromStack() : Double
+ clearStack() : void
+ getCurrentValue() : String
+ hasError() : boolean
+ negativeToPositive() : void
+ positiveToNegative() : void
+ setCurrentValue(currentValue : String) : void
+ getMemory() : Double
+ setMemory(memory : Double) : void
JCalculator
- empty : String[ ]
- jNumber : JTextField
- jStack : JList
- constraints:GridBagConstraints
- state : State
- update () : void
- addOperationButton() : void
+ JCalculator () :
0..1
1
possède >
Stack<T>
- head : Node<T>
+ Stack() :
+ insert(value: T) : void
+ clear() : void
+ pop() : T
+ isEmpty() : boolean
+ toString() : String
+ toArray() : Object[]
+ iterator() : It<T>
0..1
1
valeurs >
0..1
contient v
1 ..*
^ manipule
1
0 ..*
JButton
+ actionPerformed(e : ActionEvent) : void
0..1
utilise >
1
Operator
- execute() : void
BackspaceOperator
+ execute() : void
SquareOperator
+ execute() : void
EnterOperator
+ execute() : void
FractionalOperator
+ execute() : void
NumberOperator
- value : int
+ execute() : void
SqrtOperator
+ execute() : void
CEOperator
+ execute() : void
MSOperator
+ execute() : void
MRperator
+ execute() : void
NegativePositiveOperator
+ execute() : void
OperandOperator
+ execute() : void
0 ..*
COperator
+ execute() : void
Operation<T>
+ apply(a : T, b : T) : T
1
< possède
Addition
+ apply(a : Double, b : Double) : Double
Subtraction
+ apply(a : Double, b : Double) : Double
Multiplication
+ apply(a : Double, b : Double) : Double
Division
+ apply(a : Double, b : Double) : Double

Listing du code

Folder src/calculator

21 printable files

(file list disabled)

src/calculator/Addition.java

```java
/**
 * Représente une opération d'addition dans une calculatrice.
 * Hérite de la classe abstraite {@code Operation} paramétrée par le type {@code Double}.
 *
 * @param <Double> Le type des opérandes et du résultat de l'opération.
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;


public class Addition extends Operation<Double> {

    /**
     * Applique l'opération d'addition sur deux nombres de type {@code Double}.
     *
     * @param a Le premier opérande.
     * @param b Le second opérande.
     * @return La somme des deux opérandes {@code a + b}.
     */
    @Override
    public Double apply(Double a, Double b) {
        return a + b;
    }
}
```

src/calculator/BackspaceOperator.java

```java
/**
 * Représente un opérateur de suppression dans une calculatrice.
 * Cet opérateur permet de supprimer le dernier caractère saisi dans l'état actuel.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class BackspaceOperator extends Operator {

    /**
     * Crée un nouvel opérateur de suppression associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
    public BackspaceOperator(State state) {
        super(state);
    }
    /**
     * Exécute l'opération de suppression du dernier caractère dans l'état actuel.
     * Appelle la méthode {@code backspace()} sur l'objet {@code state}.
     */
    @Override
    public void execute() {
```

```
        state.backspace();
    }
}
```

src/calculator/CEOperator.java

```
/**
 * Représente un opérateur "CE" (Clear Entry) dans une calculatrice.
 * Cet opérateur permet de réinitialiser la valeur courante et de gérer les erreurs.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;


public class CEOperator extends Operator {

    /**
     * Crée un opérateur CE associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
    public CEOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération CE. Si une erreur est présente, elle est réinitialisée.
     * Réinitialise également la valeur courante.
     */
    @Override
    public void execute() {
        if(state.hasError()){
            state.resetError();
        }
        state.clearCurrentValue();
    }
}
```

src/calculator/COperator.java

```
/**
 * Représente un opérateur "C" (Clear) dans une calculatrice.
 * Hérite de l'opérateur CE, avec une fonctionnalité supplémentaire de vidage de la pile.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class COperator extends CEOperator {

    /**
     * Crée un opérateur C associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
    public COperator(State state) {
        super(state);
    }

    /**
```

```
     * Exécute l'opération C. Réinitialise la valeur courante, corrige les erreurs,
     * et vide la pile.
     */
    @Override
    public void execute() {
        super.execute();
        state.clearStack();
    }
}
```

src/calculator/Division.java

```
/**
 * Représente une opération de division dans une calculatrice.
 * Hérite de la classe abstraite {@code Operation} paramétrée par le type {@code Double}.
 *
 * @param <Double> Le type des opérandes et du résultat de l'opération.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */

package calculator;

public class Division extends Operation<Double>{

    /**
     * Applique l'opération de division sur deux nombres de type {@code Double}.
     *
     * @param a Le numérateur.
     * @param b Le dénominateur.
     * @return Le quotient de la division {@code a / b}.
     * @throws ArithmeticException Si le dénominateur est égal à zéro.
     */
    @Override
    public Double apply(Double a, Double b) {
        if(b!=0) {
            return a / b;
        }
        else {
            throw new ArithmeticException();
        }
    }
}
```

src/calculator/EnterOperator.java

```
/**
 * Représente un opérateur "Enter" dans une calculatrice.
 * Cet opérateur permet d'ajouter la valeur courante dans la pile.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class EnterOperator extends Operator {

    /**
     * Crée un opérateur Enter associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
```

```
    public EnterOperator(State state) {
        super(state);
    }

     /**
      * Exécute l'opération Enter, en ajoutant la valeur courante dans la pile.
      */
     @Override
     public void execute() {
         state.pushCurrentValue();
     }
 }
```

src/calculator/FractionnalOperator.java

```
 /**
  * Représente un opérateur de conversion en fraction inverse (1/x) dans une calculatrice.
  * Si aucune erreur n'est présente, il calcule l'inverse de la valeur courante.
  *
  * @Author :  Maxime Lestiboudois
  * @Author :  Nathan Parisod
  * @date : 27/11/2024
  */
 package calculator;

 public class FractionnalOperator extends Operator {

     /**
      * Crée un opérateur de fraction associé à un état donné.
      *
      * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
      */
     public FractionnalOperator(State state) {
         super(state);
     }

     /**
      * Exécute l'opération fraction inverse (1/x).
      * Si la valeur courante est différente de zéro, calcule et met à jour l'état.
      * Gère les erreurs en cas de division par zéro ou de pile vide.
      */
     @Override
     public void execute() {
         if (state.hasError()) return;
         if(!state.getCurrentValue().equals("0")) {
             state.pushCurrentValue();
         }
         Double a = state.popFromStack();
         if(a != null){
             Double b = 1/a;
             state.setCurrentValue(b.toString());
             state.pushCurrentValue();
         }
         else {
             state.setError("Erreur d'addition");
         }
     }
 }
```

src/calculator/JCalculator.java

```
 /**
  * @modified by :  Maxime Lestiboudois
  * @modified by :  Nathan Parisod
  * @date of modification : 27/11/2024
```

```
 */

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 jStack = new JList(empty);

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

    // Instance de l'état de la calculatrice
    private final State state = new State();



    // Mise a jour de l'interface apres une operation (jList et jStack)
    private void update() {
        // Modifier une zone de texte, JTextField.setText(string nom)
        // Modifier un composant liste, JList.setListData(Object[] tableau)
      System.out.println(state.getCurrentValue());
        jNumber.setText(state.getCurrentValue());
        Object[] stackData = state.getStack().toArray();
        if (stackData.length == 0) {
            jStack.setListData(empty);
        } else {
            jStack.setListData(stackData);
        }
    }

    // Ajout d'un bouton dans l'interface et de l'operation associee,
    // instance de la classe Operation, possedeant une methode execute()
    private void addOperatorButton(String name, int x, int y, Color color,
                                   final Operator operator) {
        JButton b = new JButton(name);
        b.setForeground(color);
        constraints.gridx = x;
        constraints.gridy = y;
        getContentPane().add(b, constraints);
        b.addActionListener(e -> {
            operator.execute();
            update();
        });
    }
```

```
public JCalculator() {
    super("JCalculator");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().setLayout(new GridBagLayout());

    // Contraintes des composants graphiques
    constraints.insets = new Insets(3, 3, 3, 3);
    constraints.fill = GridBagConstraints.HORIZONTAL;

    // Nombre courant
    jNumber.setEditable(false);
    jNumber.setBackground(Color.WHITE);
    jNumber.setHorizontalAlignment(JTextField.RIGHT);
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.gridwidth = 5;
    getContentPane().add(jNumber, constraints);
    constraints.gridwidth = 1; // reset width

    // Rappel de la valeur en memoire
    addOperatorButton("MR", 0, 1, Color.RED, new MROperator(state));

    // Stockage d'une valeur en memoire
    addOperatorButton("MS", 1, 1, Color.RED, new MSOperator(state));

    // Backspace
    addOperatorButton("<=", 2, 1, Color.RED, new BackspaceOperator(state));

    // Mise a zero de la valeur courante + suppression des erreurs
    addOperatorButton("CE", 3, 1, Color.RED, new CEOperator(state));

    // Comme CE + vide la pile
    addOperatorButton("C", 4, 1, Color.RED, new COperator(state));

    // Boutons 1-9
    for (int i = 1; i < 10; i++)
        addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
                Color.BLUE, new NumberOperator(i, state));
    // Bouton 0
    addOperatorButton("0", 0, 5, Color.BLUE, new NumberOperator(0, state));

    // Changement de signe de la valeur courante
    addOperatorButton("+/-", 1, 5, Color.BLUE, new PositiveNegativeOperator(state));

    // Operateur point (chiffres apres la virgule ensuite)
    addOperatorButton(".", 2, 5, Color.BLUE, new PointOperator(state));

    // Operateurs arithmetiques a deux operandes: /, *, -, +
    addOperatorButton("/", 3, 2, Color.RED, new OperandOperator(new Division(), state));
    addOperatorButton("*", 3, 3, Color.RED, new OperandOperator(new Multiplication(), state));
    addOperatorButton("-", 3, 4, Color.RED, new OperandOperator(new Substraction(),state));
    addOperatorButton("+", 3, 5, Color.RED, new OperandOperator(new Addition(),state));

    // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
    addOperatorButton("1/x", 4, 2, Color.RED, new FractionnalOperator(state));
    addOperatorButton("x^2", 4, 3, Color.RED, new SquareOperator(state));
    addOperatorButton("Sqrt", 4, 4, Color.RED, new SqrtOperator(state));

    // Entree: met la valeur courante sur le sommet de la pile
    addOperatorButton("Ent", 4, 5, Color.RED, new EnterOperator(state));

    // Affichage de la pile
    JLabel jLabel = new JLabel("Stack");
    jLabel.setFont(new Font("Dialog", 0, 12));
    jLabel.setHorizontalAlignment(JLabel.CENTER);
    constraints.gridx = 5;
```

```
        constraints.gridy = 0;
        getContentPane().add(jLabel, constraints);

        jStack.setFont(new Font("Dialog", 0, 12));
        jStack.setVisibleRowCount(8);
        JScrollPane scrollPane = new JScrollPane(jStack);
        constraints.gridx = 5;
        constraints.gridy = 1;
        constraints.gridheight = 5;
        getContentPane().add(scrollPane, constraints);
        constraints.gridheight = 1; // reset height

        setResizable(false);
        pack();
        setVisible(true);
    }
}
```

src/calculator/MROperator.java

```
/**
 * Représente un opérateur "MR" (Memory Recall) dans une calculatrice.
 * Permet de rappeler une valeur stockée en mémoire dans la valeur courante.
 *
 * @Author :  Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class MROperator extends Operator {

    /**
     * Crée un opérateur MR associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
    public MROperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération MR, en remplaçant la valeur courante par la valeur en mémoire.
     * Si aucune mémoire n'est définie, l'opération est ignorée.
     */
    @Override
    public void execute() {
        System.out.println("currentValue in MR" + state.getCurrentValue() + " memory="+ (state.getMemory() == null));
        if(state.getMemory() == null) {
            return;
        }
        state.setCurrentValue(state.getMemory().toString());
        System.out.println("currentValue in MR" + state.getCurrentValue());
        //est-ce qu'il faut reset la mémoire? non
        //est-ce que la valeur est directement push dans la stack? ???
    }
}
```

src/calculator/MSOperator.java

```
/**
 * Représente un opérateur "MS" (Memory Store) dans une calculatrice.
 * Permet de sauvegarder la valeur courante dans la mémoire.
 *
 * @Author :  Maxime Lestiboudois
```

```java
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */

package calculator;

public class MSOperator extends Operator {

    /**
     * Crée un opérateur MS associé à un état donné.
     *
     * @param state L'état de la calculatrice sur lequel l'opérateur doit agir.
     */
    public MSOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération MS, en sauvegardant la valeur courante dans la mémoire.
     */
    @Override
    public void execute() {
        state.setMemory(Double.parseDouble(state.getCurrentValue()));
        System.out.println("enregistré: "+ state.getMemory());

    }
}
```

src/calculator/Multiplication.java

```java
/**
 * Représente une opération de multiplication dans une calculatrice.
 * Hérite de la classe abstraite {@code Operation} paramétrée par le type {@code Double}.
 *
 * @param <Double> Le type des opérandes et du résultat de l'opération.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class Multiplication  extends Operation<Double>{

    /**
     * Applique l'opération de multiplication sur deux nombres de type {@code Double}.
     *
     * @param a Le premier opérande.
     * @param b Le second opérande.
     * @return Le produit des deux opérandes {@code a * b}.
     */
    @Override
    public Double apply(Double a, Double b) {
        return a*b;
    }
}
```

src/calculator/NumberOperator.java

```java
/**
 * La classe {@code NumberOperator} représente un opérateur qui ajoute une valeur numérique à
 * l'état actuel de l'application.
 *
 * @Author : Maxime Lestiboudois
```

```java
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

import java.sql.SQLOutput;

public class NumberOperator extends Operator {

    /**
     * La valeur numérique associée à cet opérateur.
     */
    private final int value;

    /**
     * Constructeur de la classe {@code NumberOperator}.
     *
     * @param value La valeur numérique que cet opérateur représente.
     * @param state L'état dans lequel l'opérateur sera appliqué.
     */
    public NumberOperator(int value, State state) {
        super(state);
        this.value = value;
    }

    /**
     * Exécute l'opération en ajoutant la valeur numérique à la chaîne actuelle
     * dans l'état en cours. La valeur est convertie en caractère et ajoutée
     * à l'état via la méthode {@code appendToCurrentValue}.
     */
    @Override
    public void execute() {
        state.appendToCurrentValue((char) (value + 48));    // 48 est le code ASCII pour '0'
    }
}
```

src/calculator/OperandOperator.java

```java
/**
 * La classe {@code OperandOperator} représente un opérateur qui effectue une opération
 * entre deux opérandes, extraits de la pile de l'état actuel.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class OperandOperator extends Operator {

    /**
     * L'opération à appliquer sur les deux opérandes (de type {@code Double}).
     */
    private final Operation<Double> operand;

    /**
     * Constructeur de la classe {@code OperandOperator}.
     *
     * @param operand L'opération à effectuer sur les deux opérandes.
     * @param state L'état dans lequel l'opération sera exécutée.
     */
    public OperandOperator(Operation<Double> operand, State state) {
        super(state);
        this.operand = operand;
    }
```

```
    /**
     * Exécute l'opération entre les deux derniers opérandes extraits de la pile.
     * Si les opérandes sont valides, l'opération est appliquée et le résultat est
     * stocké dans l'état. Si une erreur survient (par exemple si l'un des opérandes est nul),
     * un message d'erreur est défini dans l'état.
     */
    @Override
    public void execute() {
        if (state.hasError()) return;
        if(!state.getCurrentValue().equals("0")) {
            state.pushCurrentValue();
        }
        Double b = state.popFromStack();
        Double a = state.popFromStack();
        if (a != null && b != null) {
            state.setCurrentValue(operand.apply(a, b).toString());
            state.pushCurrentValue();
        } else {
            state.setError("Erreur");
        }
    }
}
```

src/calculator/Operation.java

```
/**
 * La classe abstraite {@code Operation} définit une opération générique qui peut être
 * appliquée sur deux valeurs de type {@code T}.
 *
 * @param <T> Le type des valeurs sur lesquelles l'opération sera effectuée.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public abstract class Operation<T> {

    /**
     * Applique l'opération sur deux valeurs de type {@code T}.
     *
     * @param a La première valeur de l'opération.
     * @param b La deuxième valeur de l'opération.
     * @return Le résultat de l'opération, de type {@code T}.
     */
    public abstract T apply(T a, T b);
}
```

src/calculator/Operator.java

```
/**
 * La classe abstraite {@code Operator} représente un opérateur qui interagit avec un état
 * donné pour effectuer une opération spécifique. Elle est destinée à être étendue par des
 * sous-classes qui définiront des opérations particulières.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

abstract class Operator {
```

```
    /**
     * L'état dans lequel l'opérateur va être exécuté.
     */
    protected State state;

    /**
     * Constructeur de la classe {@code Operator}.
     *
     * @param state L'état dans lequel l'opérateur sera exécuté.
     */
    public Operator(State state) {
        this.state = state;
    }

    /**
     * Méthode abstraite qui doit être implémentée dans les sous-classes pour exécuter
     * l'opération spécifique de l'opérateur.
     */
    abstract void execute();
}
```

src/calculator/PointOperator.java

```
/**
 * La classe {@code PointOperator} représente un opérateur qui ajoute un point ('.')
 * à la valeur actuelle de l'état, permettant ainsi de saisir des nombres décimaux.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class PointOperator extends Operator {

    /**
     * Constructeur de la classe {@code PointOperator}.
     *
     * @param state L'état dans lequel le point sera ajouté à la valeur actuelle.
     */
    public PointOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération en ajoutant un point ('.') à la valeur actuelle dans l'état.
     */
    @Override
    public void execute() {
        state.appendToCurrentValue('.');
    }
}
```

src/calculator/PositiveNegativeOperator.java

```
/**
 * La classe {@code PositiveNegativeOperator} représente un opérateur qui permet de
 * basculer la valeur actuelle entre positive et négative dans l'état. Si la valeur est
 * positive, elle devient négative, et si elle est négative, elle devient positive.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;
```

```java
public class PositiveNegativeOperator extends Operator {

    /**
     * Constructeur de la classe {@code PositiveNegativeOperator}.
     *
     * @param state L'état dans lequel la valeur sera convertie entre positive et négative.
     */
    public PositiveNegativeOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération en basculant la valeur actuelle entre positive et négative.
     * Si la valeur actuelle est négative, elle devient positive, et vice versa.
     */
    @Override
    public void execute() {
        if(state.getCurrentValue().charAt(0) == '-'){
            state.negativeToPositive();
        }
        else{
            state.positiveToNegative();
        }
    }
}
```

src/calculator/SqrtOperator.java

```java
/**
 * La classe {@code SqrtOperator} représente un opérateur qui calcule la racine carrée
 * de la valeur actuelle dans l'état. Si la valeur est valide, la racine carrée est
 * calculée et le résultat est stocké dans l'état.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class SqrtOperator extends Operator {

    /**
     * Constructeur de la classe {@code SqrtOperator}.
     *
     * @param state L'état dans lequel la racine carrée sera calculée.
     */
    public SqrtOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération en calculant la racine carrée de la valeur actuelle dans l'état.
     * Si la valeur est valide, la racine carrée est calculée et le résultat est stocké.
     * En cas d'erreur, un message d'erreur est défini dans l'état.
     */
    @Override
    public void execute() {
        if (state.hasError()) return;
        if(!state.getCurrentValue().equals("0")) {
            state.pushCurrentValue();
        }
        Double a = state.popFromStack();
        if(a != null){
            Double b = Math.sqrt(a);
```

```
            state.setCurrentValue(b.toString());
            state.pushCurrentValue();
        }
        else {
            state.setError("Erreur");
        }
    }
}
```

src/calculator/SquareOperator.java

```
/**
 * La classe {@code SquareOperator} représente un opérateur qui calcule le carré de la
 * valeur actuelle dans l'état. Si la valeur est valide, son carré est calculé et le
 * résultat est stocké dans l'état.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package calculator;

public class SquareOperator extends Operator {

    /**
     * Constructeur de la classe {@code SquareOperator}.
     *
     * @param state L'état dans lequel le carré sera calculé.
     */
    public SquareOperator(State state) {
        super(state);
    }

    /**
     * Exécute l'opération en calculant le carré de la valeur actuelle dans l'état.
     * Si la valeur est valide, son carré est calculé et le résultat est stocké.
     * En cas d'erreur, un message d'erreur est défini dans l'état.
     */
    @Override
    public void execute() {
        if (state.hasError()) return;
        if(!state.getCurrentValue().equals("0")) {
            state.pushCurrentValue();
        }
        Double a = state.popFromStack();
        if(a != null){
            Double b = a * a;
            state.setCurrentValue(b.toString());
            state.pushCurrentValue();
        }
        else {
            state.setError("Erreur");
        }
    }
}
```

src/calculator/State.java

```
/**
 * La classe {@code State} représente l'état d'une calculatrice, incluant la gestion de la valeur
 * actuelle, de la pile de valeurs, de la mémoire et des erreurs. Elle permet de manipuler et
 * d'effectuer des opérations sur ces éléments tout au long de l'exécution d'une série d'opérations.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
```

```java
 * @date : 27/11/2024
 */
package calculator;

import util.Stack;

public class State {

    /**
     * La pile contenant les valeurs précédemment calculées.
     */
    private final Stack<Double> stack = new Stack<>();

    /**
     * La valeur actuelle de l'affichage sous forme de chaîne de caractères.
     */
    private String currentValue = "0";

    /**
     * Indicateur d'erreur. Si {@code true}, une erreur a eu lieu et le calcul est interrompu.
     */
    private boolean error = false;

    /**
     * La mémoire pour stocker une valeur temporaire.
     */
    private Double memory;

    /**
     * Constructeur de la classe {@code State}.
     * Initialise l'état avec une pile vide, une valeur actuelle à "0" et aucune erreur.
     */
    public State() {
    }

    /**
     * Ajoute un caractère à la valeur actuelle, en le convertissant en chaîne.
     * Si la valeur actuelle est "0", elle est remplacée par le caractère.
     *
     * @param c Le caractère à ajouter à la valeur actuelle.
     */
    public void appendToCurrentValue(char c) {
        if (error) {
            resetError();
        }
        if (currentValue.equals("0")) {
            currentValue = Character.toString(c);
        } else {
            currentValue += c;
        }
    }

    /**
     * Supprime le dernier caractère de la valeur actuelle.
     * Si la valeur actuelle a plus d'un caractère, elle est tronquée. Sinon, elle devient "0".
     */
    public void backspace() {
        if (!error && currentValue.length() > 1) {
            currentValue = currentValue.substring(0, currentValue.length() - 1);
        } else {
            currentValue = "0";
        }
    }
```



```java
    /**
     * Réinitialise la valeur actuelle à "0".
```

```
 */
public void clearCurrentValue() {
    currentValue = "0";
}

/**
 * Pousse la valeur actuelle sur la pile après l'avoir convertie en {@code double}.
 * Réinitialise ensuite la valeur actuelle à "0".
 */
public void pushCurrentValue() {
    try {
        double value = Double.parseDouble(currentValue);
        stack.insert(value);
        clearCurrentValue();
    } catch (NumberFormatException e) {
        System.out.println("Error in pushCurrentValue");
    }
}

/**
 * Définit un message d'erreur et modifie la valeur actuelle pour afficher "Erreur".
 * L'indicateur d'erreur est mis à {@code true}.
 *
 * @param message Le message d'erreur à afficher.
 */
public void setError(String message) {
    error = true;
    currentValue = "Erreur";
}

/**
 * Réinitialise l'erreur et rétablit la valeur actuelle à "0".
 */
public void resetError() {
    error = false;
    clearCurrentValue();
}

/**
 * Pop une valeur de la pile. Si la pile est vide, une erreur est signalée.
 *
 * @return La valeur popée de la pile, ou {@code null} en cas d'erreur.
 */
public Double popFromStack() {
    if (!stack.isEmpty()) {
        return stack.pop();
    } else {
        System.out.println("Error in popFromStack");
        return null;
    }
}

/**
 * Vide la pile de toutes ses valeurs.
 */
public void clearStack() {
    stack.clear();
}

/**
 * Obtient la valeur actuelle de l'affichage.
 *
 * @return La valeur actuelle sous forme de chaîne de caractères.
 */
public String getCurrentValue() {
    return currentValue;
```

```
    }

    /**
     * Vérifie si une erreur a eu lieu.
     *
     * @return {@code true} si une erreur a eu lieu, sinon {@code false}.
     */
    public boolean hasError() {
        return error;
    }

    /**
     * Convertit la valeur actuelle de négative à positive, si elle est négative.
     * Si la valeur est déjà positive, rien n'est fait.
     */
    public void negativeToPositive() {
        currentValue = currentValue.substring(1, currentValue.length());
    }

    /**
     * Convertit la valeur actuelle de positive à négative, si elle est positive.
     * Si la valeur est déjà négative, rien n'est fait.
     */
    public void positiveToNegative() {
        currentValue = "-" + currentValue;
    }

    /**
     * Définit une nouvelle valeur actuelle.
     *
     * @param currentValue La nouvelle valeur actuelle sous forme de chaîne.
     */
    public void setCurrentValue(String currentValue) {
        this.currentValue = currentValue;
    }

    /**
     * Obtient la pile contenant les valeurs précédemment calculées.
     *
     * @return La pile de valeurs.
     */
    public Stack<Double> getStack() {
        return stack;
    }

    /**
     * Obtient la valeur stockée en mémoire.
     *
     * @return La valeur stockée en mémoire.
     */
    public Double getMemory() {
        return memory;
    }

    /**
     * Définit la valeur à stocker en mémoire.
     *
     * @param memory La valeur à stocker en mémoire.
     */
    public void setMemory(Double memory) {
        this.memory = memory;
    }
}
```


src/calculator/Substraction.java

```java
/**
 * Représente une opération de soustraction dans une calculatrice.
 * Hérite de la classe abstraite {@code Operation} paramétrée par le type {@code Double}.
 *
 * @param <Double> Le type des opérandes et du résultat de l'opération.
 *
 * @Author :  Maxime Lestiboudois
 * @Author :  Nathan Parisod
 * @date : 27/11/2024
 *  */

package calculator;

public class Substraction extends Operation<Double>{

    /**
     * Applique l'opération de soustraction sur deux nombres de type {@code Double}.
     *
     * @param a Le premier opérande.
     * @param b Le second opérande.
     * @return La différence entre les deux opérandes {@code a - b}.
     */
    @Override
    public Double apply(Double a, Double b) {
        return a-b;
    }
}
```

Folder src/util

3 printable files

(file list disabled)

src/util/It.java

```java
/**
 * Classe It représentant un itérateur générique pour une structure de données chaînée.
 * Permet de parcourir les éléments d'une liste chaînée de manière séquentielle.
 *
 * @param <T> le type des éléments à parcourir.
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package util;

import java.util.NoSuchElementException;

public class It<T> {
    /**
     * Le nœud courant de la liste chaînée.
     */
    Node<T> current;

    /**
     * Constructeur de l'itérateur.
     * Initialise l'itérateur à partir d'un nœud de départ.
     *
     * @param start le nœud de départ de l'itérateur.
     */
    public It(Node<T> start) {
        this.current = start;
    }

    /**
     * Vérifie si l'itérateur a un élément suivant.
     *
     * @return true si un élément suivant existe, false sinon.
     */
    public boolean hasNext() {
        return current != null;
    }

    /**
     * Renvoie l'élément suivant dans la liste chaînée et avance l'itérateur.
     *
     * @return la valeur de l'élément courant.
     * @throws NoSuchElementException si aucun élément suivant n'existe.
     */
    public T next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        T value = current.data;
        current = current.next;
        return value;
    }
}
```


src/util/Node.java

```java
/**
 * Classe Node représentant un nœud d'une liste chaînée.
 * Chaque nœud contient une donnée et une référence vers le nœud suivant dans la liste.
 *
 * @param <T> le type de la donnée contenue dans le nœud.
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package util;

class Node<T> {
    /**
     * La donnée stockée dans ce nœud.
     */
    T data;

    /**
     * La référence vers le nœud suivant dans la liste.
     */
    Node<T> next;

    /**
     * Constructeur de la classe Node.
     * Initialise le nœud avec une donnée et sans nœud suivant.
     *
     * @param data la donnée à stocker dans le nœud.
     */
    Node(T data) {
        this.data = data;
        this.next = null;
    }
}
```

src/util/Stack.java

```java
/**
 * Classe Stack représentant une pile générique basée sur une structure de liste chaînée.
 * Permet d'insérer, retirer et manipuler des éléments selon le principe LIFO (Last In, First Out).
 *
 * @param <T> le type des éléments stockés dans la pile.
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package util;

import java.util.NoSuchElementException;

public class Stack<T> {
    /**
     * Référence au sommet de la pile.
     */
    private Node<T> head;

    /**
     * Constructeur de la pile.
     * Initialise une pile vide.
     */
    public Stack() {
        this.head = null;
    }

    /**
     * Insère un nouvel élément au sommet de la pile.
```

```
     *
     * @param value la valeur à insérer.
     */
    public void insert(T value) {
        Node<T> newNode = new Node<>(value);
        newNode.next = head;
        head = newNode;
    }

    /**
     * Vide la pile en supprimant tous ses éléments.
     */
    public void clear() {
        head = null;
    }

    /**
     * Retire et renvoie l'élément au sommet de la pile.
     *
     * @return la valeur de l'élément retiré.
     * @throws NoSuchElementException si la pile est vide.
     */
    public T pop() {
        if (isEmpty()) {
            throw new NoSuchElementException("Stack is empty");
        }
        T value = head.data;
        head = head.next;
        return value;
    }

    /**
     * Vérifie si la pile est vide.
     *
     * @return true si la pile est vide, false sinon.
     */
    public boolean isEmpty() {
        return head == null;
    }

    /**
     * Renvoie une représentation sous forme de chaîne de caractères des éléments de la pile.
     * Les éléments sont listés dans l'ordre de leur apparition, séparés par des espaces.
     *
     * @return une chaîne de caractères représentant la pile.
     */
    @Override
    public String toString() {
        String result = "";
        Node<T> current = head;
        while (current != null) {
            result += current.data;
            if (current.next != null) {
                result += " ";
            }
            current = current.next;
        }
        return result;
    }

    /**
     * Convertit la pile en un tableau d'objets.
     * Les éléments sont ordonnés du sommet vers le bas de la pile.
     *
     * @return un tableau contenant les éléments de la pile.
     */
```

```
    public Object[] toArray() {
        int size = 0;
        Node<T> current = head;
        while (current != null) {
            size++;
            current = current.next;
        }

        Object[] array = new Object[size];
        current = head;
        for (int i = 0; i < size; i++) {
            array[i] = current.data;
            current = current.next;
        }
        return array;
    }

    /**
     * Crée un itérateur pour parcourir les éléments de la pile.
     *
     * @return un itérateur positionné au sommet de la pile.
     */
    public It<T> iterator() {
        return new It<>(head);
    }
}
```

Folder src/test

5 printable files

(file list disabled)

src/test/CalculatorTest.java

```java
/**
 * Classe de test pour les fonctionnalités d'une calculatrice.
 * Contient des tests pour vérifier le bon fonctionnement des opérations arithmétiques
 * et des opérateurs associés.
 *
 * @autor Maxime Lestiboudois
 * @autor Nathan Parisod
 * @date 27/11/2024
 */
package test;

import calculator.*;

public class CalculatorTest {

    /**
     * Point d'entrée principal pour exécuter tous les tests de la calculatrice.
     * @param args arguments de la ligne de commande (non utilisés).
     */
    public static void main(String[] args) {

        // Test d'Addition
        /**
         * Teste l'opération d'addition.
         * Ajoute deux valeurs stockées dans l'état et vérifie que le résultat est correct.
         */
        Addition addition = new Addition();
        State state = new State();
        state.appendToCurrentValue('2');
        state.appendToCurrentValue('3');
        state.pushCurrentValue();
        state.appendToCurrentValue('4');
        state.pushCurrentValue();
        Double value1 = state.popFromStack();
        Double value2 = state.popFromStack();
        Double result = addition.apply(value1, value2);
        state.setCurrentValue(result.toString());
        state.pushCurrentValue();
        assert state.getCurrentValue().equals("7.0") : "execute() a échoué";

        // Test de BackspaceOperator
        /**
         * Teste l'opérateur de suppression (BackspaceOperator).
         * Supprime le dernier caractère de la valeur courante et vérifie le résultat.
         */
        BackspaceOperator backspaceOperator = new BackspaceOperator(state);
        state.setCurrentValue("123");
        backspaceOperator.execute();
        assert state.getCurrentValue().equals("12") : "execute() a échoué";

        // Test de CEOperator
        /**
         * Teste l'opérateur CE (Clear Entry).
         * Remet la valeur courante à zéro et vérifie le résultat.
         */
        CEOperator ceOperator = new CEOperator(state);
```

```
state.setCurrentValue("123");
ceOperator.execute();
assert state.getCurrentValue().equals("0") : "execute() a échoué";

// Test de COperator
/**
 * Teste l'opérateur C (Clear).
 * Remet la valeur courante et l'état entier à zéro, et vérifie le résultat.
 */
COperator cOperator = new COperator(state);
state.setCurrentValue("123");
cOperator.execute();
assert state.getCurrentValue().equals("0") : "execute() a échoué";

// Test de Division
/**
 * Teste l'opération de division.
 * Divise deux valeurs et vérifie le résultat pour des cas valides et un cas de division par zéro.
 */
Division divide = new Division();
state.setCurrentValue("10");
state.pushCurrentValue();
state.setCurrentValue("2");
state.pushCurrentValue();
value1 = state.popFromStack();
value2 = state.popFromStack();
result = divide.apply(value1, value2);
state.setCurrentValue(result.toString());
state.pushCurrentValue();
assert state.getCurrentValue().equals("5.0") : "execute() a échoué";

state.setCurrentValue("10");
state.pushCurrentValue();
state.setCurrentValue("0");
state.pushCurrentValue();
value1 = state.popFromStack();
value2 = state.popFromStack();
result = divide.apply(value1, value2);
state.setCurrentValue(result.toString());
assert state.getCurrentValue().equals("Infinity") : "execute() a échoué";

// Test d'EnterOperator
/**
 * Teste l'opérateur Enter.
 * Ajoute la valeur courante à la pile et vérifie que la valeur reste inchangée.
 */
EnterOperator enterOperator = new EnterOperator(state);
state.setCurrentValue("123");
enterOperator.execute();
assert state.getCurrentValue().equals("123") : "execute() a échoué";

// Test de FractionOperator
/**
 * Teste l'opérateur de fraction (FractionOperator).
 * Transforme une valeur en sa fraction (1/x) et vérifie le résultat.
 */
FractionnalOperator fractionOperator = new FractionnalOperator(state);
state.setCurrentValue("2");
fractionOperator.execute();
assert state.getCurrentValue().equals("0.5") : "execute() a échoué";

// Test de MROperator
/**
 * Teste l'opérateur MR (Memory Recall).
 * Rappelle une valeur précédemment mémorisée et vérifie le résultat.
 */
```

```
MROperator mrOperator = new MROperator(state);
state.setCurrentValue("123");
mrOperator.execute();
assert state.getCurrentValue().equals("123") : "execute() a échoué";

// Test de MSOperator
/**
 * Teste l'opérateur MS (Memory Store).
 * Mémorise la valeur courante et vérifie qu'elle est correctement stockée.
 */
MSOperator msOperator = new MSOperator(state);
state.setCurrentValue("123");
msOperator.execute();
assert state.getCurrentValue().equals("123") : "execute() a échoué";

// Test de Multiplication
/**
 * Teste l'opération de multiplication.
 * Multiplie deux valeurs stockées dans l'état et vérifie le résultat.
 */
Multiplication multiplication = new Multiplication();
state.setCurrentValue("10");
state.pushCurrentValue();
state.setCurrentValue("2");
state.pushCurrentValue();
value1 = state.popFromStack();
value2 = state.popFromStack();
result = multiplication.apply(value1, value2);
state.setCurrentValue(result.toString());
state.pushCurrentValue();
assert state.getCurrentValue().equals("20.0") : "execute() a échoué";

// Test de NumberOperator
/**
 * Teste l'ajout d'un chiffre via NumberOperator.
 * Ajoute un chiffre à la fin de la valeur courante et vérifie le résultat.
 */
NumberOperator numberOperator = new NumberOperator(5, state);
state.setCurrentValue("123");
numberOperator.execute();
assert state.getCurrentValue().equals("1235") : "execute() a échoué";

// Test de OperandOperator
/**
 * Teste un opérateur générique avec un opérande (OperandOperator).
 * Effectue une opération arithmétique (addition) et vérifie le résultat.
 */
OperandOperator operandOperator = new OperandOperator(new Addition(), state);
state.setCurrentValue("10");
state.pushCurrentValue();
state.setCurrentValue("2");
state.pushCurrentValue();
operandOperator.execute();
assert state.getCurrentValue().equals("12.0") : "execute() a échoué";

// Test de PointOperator
/**
 * Teste l'opérateur Point.
 * Ajoute un point décimal à la valeur courante et vérifie le résultat.
 */
PointOperator pointOperator = new PointOperator(state);
state.setCurrentValue("123");
pointOperator.execute();
assert state.getCurrentValue().equals("123.") : "execute() a échoué";

// Test de PositiveNegativeOperator
```

```
        /**
         * Teste l'opérateur de changement de signe.
         * Change le signe de la valeur courante et vérifie le résultat.
         */
        PositiveNegativeOperator positiveNegativeOperator = new PositiveNegativeOperator(state);
        state.setCurrentValue("123");
        positiveNegativeOperator.execute();
        assert state.getCurrentValue().equals("-123") : "execute() a échoué";
        state.setCurrentValue("-123");
        positiveNegativeOperator.execute();
        assert state.getCurrentValue().equals("123") : "execute() a échoué";

        // Test de SqrtOperator
        /**
         * Teste l'opérateur de racine carrée.
         * Calcule la racine carrée d'une valeur et vérifie les résultats pour des cas positifs, nuls et négatifs.
         */
        SqrtOperator sqrtOperator = new SqrtOperator(state);
        state.setCurrentValue("4");
        sqrtOperator.execute();
        assert state.getCurrentValue().equals("2.0") : "execute() a échoué";

        state.setCurrentValue("0");
        sqrtOperator.execute();
        assert state.getCurrentValue().equals("0.0") : "execute() a échoué";

        state.setCurrentValue("-4");
        sqrtOperator.execute();
        assert state.getCurrentValue().equals("Erreur") : "execute() a échoué";

        // Test de SquareOperator
        /**
         * Teste l'opérateur de mise au carré (SquareOperator).
         * Calcule le carré d'une valeur et vérifie le résultat.
         */
        SquareOperator squareOperator = new SquareOperator(state);
        state.setCurrentValue("4");
        squareOperator.execute();
        assert state.getCurrentValue().equals("16.0") : "execute() a échoué";

        state.setCurrentValue("-4");
        squareOperator.execute();
        assert state.getCurrentValue().equals("16.0") : "execute() a échoué";

        // Test de Substraction
        /**
         * Teste l'opération de soustraction.
         * Soustrait deux valeurs stockées dans l'état et vérifie le résultat.
         */
        Substraction substraction = new Substraction();
        state.setCurrentValue("10");
        state.pushCurrentValue();
        state.setCurrentValue("2");
        state.pushCurrentValue();
        value1 = state.popFromStack();
        value2 = state.popFromStack();
        result = substraction.apply(value1, value2);
        state.setCurrentValue(result.toString());
        state.pushCurrentValue();
        assert state.getCurrentValue().equals("8.0") : "execute() a échoué";

        System.out.println("Tous les tests de Calculator réussis !");
    }
}
```

src/test/JCalculatorTest.java

```java
/**
 * Classe de test pour les fonctionnalités de la calculatrice graphique JCalculator.
 * Cette classe utilise des méthodes utilitaires pour simuler les clics sur les boutons
 * et vérifier les résultats affichés.
 *
 * @author Maxime Lestiboudois
 * @author Nathan Parisod
 * @date 27/11/2024
 */
package test;

import calculator.*;

public class JCalculatorTest {

    /**
     * Point d'entrée principal pour exécuter tous les tests de JCalculator.
     * @param args arguments de la ligne de commande (non utilisés).
     */
    public static void main(String[] args) {
        JCalculator calculator = new JCalculator();

        // Test d'Addition
        /**
         * Teste l'opération d'addition.
         * Simule l'ajout de deux nombres via les boutons et vérifie que le résultat est correct.
         */
        System.out.println("Test d'Addition...");
        JCalculatorTestUtils.clickButton(calculator, "2");
        JCalculatorTestUtils.clickButton(calculator, "Ent");
        JCalculatorTestUtils.clickButton(calculator, "3");
        JCalculatorTestUtils.clickButton(calculator, "+");
        assert JCalculatorTestUtils.getDisplayValue(calculator).equals("5") : "Addition échouée";

        // Test de Soustraction
        /**
         * Teste l'opération de soustraction.
         * Simule la soustraction de deux nombres via les boutons et vérifie le résultat.
         */
        System.out.println("Test de Soustraction...");
        JCalculatorTestUtils.clickButton(calculator, "1");
        JCalculatorTestUtils.clickButton(calculator, "0");
        JCalculatorTestUtils.clickButton(calculator, "Ent");
        JCalculatorTestUtils.clickButton(calculator, "4");
        JCalculatorTestUtils.clickButton(calculator, "-");
        assert JCalculatorTestUtils.getDisplayValue(calculator).equals("6") : "Soustraction échouée";

        // Test de Multiplication
        /**
         * Teste l'opération de multiplication.
         * Multiplie deux nombres via les boutons et vérifie le résultat.
         */
        System.out.println("Test de Multiplication...");
        JCalculatorTestUtils.clickButton(calculator, "3");
        JCalculatorTestUtils.clickButton(calculator, "Ent");
        JCalculatorTestUtils.clickButton(calculator, "5");
        JCalculatorTestUtils.clickButton(calculator, "*");
        assert JCalculatorTestUtils.getDisplayValue(calculator).equals("15") : "Multiplication échouée";

        // Test de Division
        /**
         * Teste l'opération de division.
         * Divise deux nombres via les boutons et vérifie le résultat.
         */
```

```
System.out.println("Test de Division...");
JCalculatorTestUtils.clickButton(calculator, "8");
JCalculatorTestUtils.clickButton(calculator, "Ent");
JCalculatorTestUtils.clickButton(calculator, "4");
JCalculatorTestUtils.clickButton(calculator, "/");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("2") : "Division échouée";

// Test de Backspace
/**
 * Teste le bouton de suppression (Backspace).
 * Supprime le dernier chiffre entré et vérifie le résultat.
 */
System.out.println("Test de Backspace...");
JCalculatorTestUtils.clickButton(calculator, "1");
JCalculatorTestUtils.clickButton(calculator, "2");
JCalculatorTestUtils.clickButton(calculator, "<=");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("1") : "Backspace échoué";

// Test de CEOperator
/**
 * Teste le bouton CE (Clear Entry).
 * Réinitialise la valeur affichée et vérifie le résultat.
 */
System.out.println("Test de CEOperator...");
JCalculatorTestUtils.clickButton(calculator, "1");
JCalculatorTestUtils.clickButton(calculator, "2");
JCalculatorTestUtils.clickButton(calculator, "3");
JCalculatorTestUtils.clickButton(calculator, "CE");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("0") : "CEOperator échoué";

// Test de COperator
/**
 * Teste le bouton C (Clear).
 * Réinitialise l'affichage et vide la pile, puis vérifie le résultat.
 */
System.out.println("Test de COperator...");
JCalculatorTestUtils.clickButton(calculator, "5");
JCalculatorTestUtils.clickButton(calculator, "Ent");
JCalculatorTestUtils.clickButton(calculator, "C");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("0") : "COperator échoué";
assert JCalculatorTestUtils.getStackContents(calculator).length == 0 : "La pile devrait être vide";

// Test de Memory Store et Recall
/**
 * Teste les boutons MS (Memory Store) et MR (Memory Recall).
 * Mémorise une valeur puis la rappelle, en vérifiant le résultat.
 */
System.out.println("Test de Memory Store et Recall...");
JCalculatorTestUtils.clickButton(calculator, "5");
JCalculatorTestUtils.clickButton(calculator, "MS");
JCalculatorTestUtils.clickButton(calculator, "CE");
JCalculatorTestUtils.clickButton(calculator, "MR");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("5") : "Memory Store/Recall échoué";

// Test de Square
/**
 * Teste le bouton carré (x²).
 * Calcule le carré d'un nombre et vérifie le résultat.
 */
System.out.println("Test de Square...");
JCalculatorTestUtils.clickButton(calculator, "3");
JCalculatorTestUtils.clickButton(calculator, "x^2");
assert JCalculatorTestUtils.getDisplayValue(calculator).equals("9") : "Square échoué";
```



```
// Test de Sqrt
/**
```

```
         * Teste le bouton racine carrée (Sqrt).
         * Calcule la racine carrée d'un nombre et vérifie le résultat.
         */
        System.out.println("Test de Sqrt...");
        JCalculatorTestUtils.clickButton(calculator, "1");
        JCalculatorTestUtils.clickButton(calculator, "6");
        JCalculatorTestUtils.clickButton(calculator, "Sqrt");
        assert JCalculatorTestUtils.getDisplayValue(calculator).equals("4") : "Sqrt échoué";

        // Test de Negate
        /**
         * Teste le bouton de changement de signe (+/-).
         * Change le signe du nombre affiché et vérifie le résultat.
         */
        System.out.println("Test de Negate...");
        JCalculatorTestUtils.clickButton(calculator, "6");
        JCalculatorTestUtils.clickButton(calculator, "+/-");
        assert JCalculatorTestUtils.getDisplayValue(calculator).equals("-6") : "Negate échoué";

        System.out.println("Tous les tests de JCalculator réussis !");
    }
}
```

src/test/JCalculatorTestUtils.java

```
/**
 * Classe utilitaire pour tester la calculatrice graphique JCalculator.
 * Contient des méthodes pour simuler des clics sur les boutons, récupérer des composants
 * et obtenir les informations affichées ou stockées dans la pile.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package test;

import calculator.JCalculator;

import javax.swing.*;
import java.awt.*;

public class JCalculatorTestUtils {

    /**
     * Récupère un bouton par son étiquette dans le conteneur donné.
     * @param container le conteneur où chercher le bouton.
     * @param label l'étiquette du bouton recherché.
     * @return le bouton correspondant, ou null si aucun bouton n'est trouvé.
     */
    public static JButton getButtonByLabel(Container container, String label) {
        for (Component component : container.getComponents()) {
            if (component instanceof JButton button) {
                if (button.getText().equals(label)) {
                    return button;
                }
            } else if (component instanceof Container) {
                JButton button = getButtonByLabel((Container) component, label);
                if (button != null) {
                    return button;
                }
            }
        }
        return null;
    }
```

```java
    /**
     * Récupère un composant par son nom dans le conteneur donné.
     * @param container le conteneur où chercher le composant.
     * @param name le nom du composant recherché.
     * @return le composant correspondant, ou null si aucun composant n'est trouvé.
     */
    public static Component getComponentByName(Container container, String name) {
        for (Component component : container.getComponents()) {
            if (name.equals(component.getName())) {
                return component;
            } else if (component instanceof Container) {
                Component child = getComponentByName((Container) component, name);
                if (child != null) {
                    return child;
                }
            }
        }
        return null;
    }

    /**
     * Simule un clic sur un bouton d'une calculatrice.
     * @param calculator l'instance de la calculatrice.
     * @param label l'étiquette du bouton à cliquer.
     * @throws IllegalArgumentException si le bouton n'est pas trouvé.
     */
    public static void clickButton(JCalculator calculator, String label) {
        JButton button = getButtonByLabel(calculator, label);
        if (button == null) throw new IllegalArgumentException("Bouton non trouvé : " + label);
        button.doClick();
    }

    /**
     * Récupère la valeur affichée sur l'écran de la calculatrice.
     * @param calculator l'instance de la calculatrice.
     * @return la valeur affichée sous forme de chaîne de caractères.
     * @throws IllegalStateException si le champ d'affichage n'est pas trouvé.
     */
    public static String getDisplayValue(JCalculator calculator) {
        JTextField displayField = (JTextField) getComponentByName(calculator, "jNumber");
        if (displayField == null) throw new IllegalStateException("Champ d'affichage introuvable");
        return displayField.getText();
    }

    /**
     * Récupère le contenu de la pile de la calculatrice.
     * @param calculator l'instance de la calculatrice.
     * @return un tableau de chaînes contenant les éléments de la pile.
     * @throws IllegalStateException si la pile n'est pas trouvée.
     */
    public static String[] getStackContents(JCalculator calculator) {
        JList<String> stackList = (JList<String>) getComponentByName(calculator, "jStack");
        if (stackList == null) throw new IllegalStateException("Pile introuvable");
        ListModel<String> model = stackList.getModel();
        String[] stackContents = new String[model.getSize()];
        for (int i = 0; i < model.getSize(); i++) {
            stackContents[i] = model.getElementAt(i);
        }
        return stackContents;
    }
}
```

src/test/StackTest.java

```java
/**
```

```java
 * Classe de test pour la pile (Stack).
 * Cette classe teste les principales fonctionnalités de la pile, y compris l'insertion,
 * la suppression, le vidage, et la conversion en tableau ou en chaîne de caractères.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package test;

import util.Stack;

public class StackTest {

    /**
     * Point d'entrée principal pour exécuter les tests de la classe Stack.
     * @param args arguments de la ligne de commande (non utilisés).
     */
    public static void main(String[] args) {
        Stack<Double> stack = new Stack<>();

        // Test de la méthode insert
        /**
         * Teste l'insertion d'éléments dans la pile.
         * Vérifie que les éléments sont correctement placés au sommet de la pile.
         */
        stack.insert(1.0);
        stack.insert(2.0);
        assert stack.toArray()[0].equals(2.0) : "Erreur : l'élément au sommet devrait être 2.0";
        assert stack.toArray()[1].equals(1.0) : "Erreur : l'élément suivant devrait être 1.0";

        // Test de la méthode pop
        /**
         * Teste le retrait d'éléments de la pile.
         * Vérifie que les éléments sont retirés dans l'ordre LIFO (Last In, First Out).
         */
        assert stack.pop().equals(2.0) : "Erreur : l'élément retiré devrait être 2.0";
        assert stack.pop().equals(1.0) : "Erreur : l'élément retiré devrait être 1.0";

        // Test de la méthode isEmpty
        /**
         * Teste la méthode isEmpty.
         * Vérifie que la pile est considérée comme vide après le retrait de tous les éléments.
         */
        assert stack.isEmpty() : "Erreur : la pile devrait être vide";

        // Test de la méthode toString
        /**
         * Teste la conversion de la pile en chaîne de caractères.
         * Vérifie que la représentation en chaîne est correcte.
         */
        stack.insert(3.0);
        stack.insert(4.0);
        assert stack.toString().equals("[4.0, 3.0]") : "Erreur : la pile devrait afficher [4.0, 3.0]";

        // Test de la méthode clear
        /**
         * Teste le vidage de la pile.
         * Vérifie que la pile est vide après l'appel à la méthode clear.
         */
        stack.insert(5.0);
        stack.clear();
        assert stack.isEmpty() : "Erreur : la pile devrait être vide";
```



```java
        // Test de la méthode toArray
        /**
```

```
         * Teste la conversion de la pile en tableau.
         * Vérifie que les éléments du tableau sont ordonnés comme dans la pile.
         */
        stack.insert(5.0);
        stack.insert(6.0);
        Object[] array = stack.toArray();
        assert array[0].equals(6.0) : "Erreur : l'élément au sommet devrait être 6.0";
        assert array[1].equals(5.0) : "Erreur : l'élément suivant devrait être 5.0";

        System.out.println("Tous les tests pour Stack ont réussi !");
    }
}
```

src/test/StateTest.java

```
/**
 * Classe de test pour la classe State.
 * Teste les différentes fonctionnalités offertes par State, y compris la gestion de la valeur courante,
 * des erreurs, de la pile et de la mémoire.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
package test;

import calculator.State;

public class StateTest {

    /**
     * Point d'entrée principal pour exécuter les tests de la classe State.
     * @param args arguments de la ligne de commande (non utilisés).
     */
    public static void main(String[] args) {
        State state = new State();

        // Test de la méthode appendToCurrentValue
        /**
         * Teste l'ajout de caractères à la valeur courante.
         * Vérifie que la valeur courante est correctement mise à jour après chaque ajout.
         */
        state.appendToCurrentValue('1');
        assert state.getCurrentValue().equals("1") : "appendToCurrentValue('1') a échoué";
        state.appendToCurrentValue('2');
        assert state.getCurrentValue().equals("12") : "appendToCurrentValue('2') a échoué";
        state.appendToCurrentValue('3');
        assert state.getCurrentValue().equals("123") : "appendToCurrentValue('3') a échoué";

        // Test de la méthode backspace
        /**
         * Teste la suppression du dernier caractère de la valeur courante.
         * Vérifie que le caractère est correctement supprimé.
         */
        state.backspace();
        assert state.getCurrentValue().equals("12") : "backspace() a échoué";

        // Test de la méthode clearCurrentValue
        /**
         * Teste la réinitialisation de la valeur courante à zéro.
         * Vérifie que la valeur courante devient "0".
         */
        state.clearCurrentValue();
        assert state.getCurrentValue().equals("0") : "clearCurrentValue() a échoué";
```

```
// Test de la méthode pushCurrentValue
/**
 * Teste l'ajout de la valeur courante à la pile.
 * Vérifie que la valeur courante est réinitialisée après l'ajout.
 */
state.appendToCurrentValue('1');
state.appendToCurrentValue('2');
state.appendToCurrentValue('3');
state.pushCurrentValue();
assert state.getCurrentValue().equals("0") : "pushCurrentValue() a échoué";

// Test de la méthode setError
/**
 * Teste la définition d'une erreur.
 * Vérifie que la valeur courante est remplacée par le message d'erreur.
 */
state.setError("Erreur");
assert state.getCurrentValue().equals("Erreur") : "setError() a échoué";

// Test de la méthode resetError
/**
 * Teste la réinitialisation d'une erreur.
 * Vérifie que la valeur courante redevient "0".
 */
state.resetError();
assert state.getCurrentValue().equals("0") : "resetError() a échoué";

// Test de la méthode popFromStack
/**
 * Teste le retrait d'une valeur de la pile.
 * Vérifie que la valeur retirée devient la valeur courante.
 */
state.appendToCurrentValue('1');
state.appendToCurrentValue('2');
state.appendToCurrentValue('3');
state.popFromStack();
assert state.getCurrentValue().equals("123") : "popFromStack() a échoué";

// Test de la méthode clearStack
/**
 * Teste le vidage de la pile.
 * Vérifie que la pile est complètement vidée.
 */
state.clearStack();
assert state.getCurrentValue().equals("0") : "clearStack() a échoué";

// Test de la méthode getCurrentValue
/**
 * Teste la récupération de la valeur courante.
 * Vérifie que la valeur courante est correcte.
 */
assert state.getCurrentValue().equals("0") : "getCurrentValue() a échoué";

// Test de la méthode hasError
/**
 * Teste la vérification de la présence d'une erreur.
 * Vérifie qu'aucune erreur n'est détectée lorsque la valeur courante est normale.
 */
assert !state.hasError() : "hasError() a échoué";

// Test de la méthode negativeToPositive
/**
 * Teste la conversion d'une valeur négative en valeur positive.
 * Vérifie que le signe de la valeur courante est correctement changé.
 */
state.appendToCurrentValue('-');
```

```
        state.appendToCurrentValue('1');
        state.negativeToPositive();
        assert state.getCurrentValue().equals("1") : "negativeToPositive() a échoué";
        state.negativeToPositive();
        assert state.getCurrentValue().equals("1") : "negativeToPositive() a échoué";

        // Test de la méthode positiveToNegative
        /**
         * Teste la conversion d'une valeur positive en valeur négative.
         * Vérifie que le signe de la valeur courante est correctement changé.
         */
        state.positiveToNegative();
        assert state.getCurrentValue().equals("-1") : "positiveToNegative() a échoué";
        state.positiveToNegative();
        assert state.getCurrentValue().equals("-1") : "positiveToNegative() a échoué";

        // Test de la méthode setCurrentValue
        /**
         * Teste la définition explicite de la valeur courante.
         * Vérifie que la valeur courante est correctement mise à jour.
         */
        state.clearStack();
        state.setCurrentValue("123");
        assert state.getCurrentValue().equals("123") : "setCurrentValue() a échoué";

        // Test de la méthode setMemory
        /**
         * Teste la définition de la mémoire.
         * Vérifie que la valeur est correctement stockée en mémoire.
         */
        state.setMemory(123.0);
        assert state.getMemory().equals(123.0) : "setMemory() a échoué";

        // Test de la méthode getMemory
        /**
         * Teste la récupération de la valeur en mémoire.
         * Vérifie que la valeur stockée est correcte.
         */
        assert state.getMemory().equals(123.0) : "getMemory() a échoué";

        System.out.println("Tous les tests pour State ont réussi !");
    }
}
```

src/Calculator.java

```java
/**
 * Représente une calculatrice interactive fonctionnant en console.
 * Elle permet d'exécuter des opérations arithmétiques et des manipulations d'état
 * en saisissant des commandes textuelles.
 *
 * @Author : Maxime Lestiboudois
 * @Author : Nathan Parisod
 * @date : 27/11/2024
 */
import calculator.*;
import java.util.Scanner;
public class Calculator {
/** Représente l'état interne de la calculatrice, incluant la pile et la mémoire.*/
    private final State state;

/** Permet de lire les entrées de l'utilisateur via la console.*/
    private final Scanner scanner;

    /**
     * Constructeur par défaut qui initialise la calculatrice.
     * Crée un état vierge et configure le scanner pour les entrées utilisateur.
     */
    public Calculator() {
        this.state = new State();
        this.scanner = new Scanner(System.in);
    }

    /**
     * Démarre la boucle principale de la calculatrice.
     * Permet à l'utilisateur de saisir des commandes ou des opérandes.
     */
    public void start(){
        boolean stay = true;
        while(stay){
            System.out.print("> ");
            String input = scanner.nextLine().trim();

            if(input.equals("exit")){
                break;
            }

            stay = processInput(input);

            displayState();
        }
    }

    /**
     * Traite une entrée utilisateur.
     * Si l'entrée est un nombre, elle est ajoutée à la pile.
     * Si l'entrée correspond à un opérateur, celui-ci est exécuté.
     *
     * @param input La chaîne saisie par l'utilisateur.
     * @return {@code true} si la calculatrice doit continuer à fonctionner,
     *         {@code false} sinon (lorsqu'un opérateur "exit" est saisi ou une erreur survient).
     */
    private boolean processInput(String input){
        if(isNumber(input)){

            for(char c : input.toCharArray()){
                new NumberOperator(c - '0', state).execute();
            }
            new EnterOperator(state).execute();
```

```
    }
    else {
        // Si l'entrée est un opérateur, trouver et exécuter l'opération correspondante
        switch (input) {
            case "+":
                new OperandOperator(new Addition(),state).execute();
                break;
            case "-":
                new OperandOperator(new Substraction(), state).execute();
                break;
            case "*":
                new OperandOperator(new Multiplication(), state).execute();
                break;
            case "/":
                new OperandOperator(new Division(), state).execute();
                break;
            case "x^2":
                new SquareOperator(state).execute();
                break;
            case "sqrt":
                new SqrtOperator(state).execute();
                break;
            case "1/x":
                new FractionnalOperator(state).execute();
                break;
            case "c":
                new COperator(state).execute();
                break;
            case "exit":
                return false;
            default:
                System.out.println("Opérateur inconnu");
                return false;
        }
    }
    return true;
}

/**
 * Affiche l'état actuel de la calculatrice.
 * Si une erreur est présente, affiche "Error". Sinon, affiche la pile actuelle.
 */
private void displayState(){
    if
    (state.hasError()){
        System.out.println("Error");
    }
    else{
        System.out.println(state.getStack().toString());
    }
}
/**
 * Vérifie si une chaîne correspond à un nombre entier valide.
 *
 * @param input La chaîne à vérifier.
 * @return {@code true} si la chaîne est un nombre entier, {@code false} sinon.
 */
private boolean isNumber(String input){
    try {
        Double.parseDouble(input);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}
```

```java
    /**
     * Point d'entrée de l'application.
     * Initialise et démarre une nouvelle instance de la calculatrice.
     *
     * @param args Les arguments de ligne de commande (non utilisés).
     */
    public static void main(String[] args){
        new Calculator().start();
    }
}
```

src/Main.java

```java
/**
  * @modified by :  Maxime Lestiboudois
  * @modified by :  Nathan Parisod
  * @date of modification : 27/11/2024
 */

import calculator.JCalculator;

public class Main
{
    public static void main(String ... args) {
        new JCalculator();
    }
}
```

Choix de conception

Les opérateurs

Nous avons reçu un dossier "starter" fourni par le corps professoral. Dans ce dossier, nous avons trouvé le fichier Operator, contenant la classe abstraite Operator, qui nous a servit de base pour construire tous les opérateurs (fonctionnalités des boutons) de notre calculatrice.
Nos opérateurs sont tous, sans exception, des classes enfants de la classe abstraite Operator, elles redéfinissent donc toutes la méthode abstraite execute() de la super-classe. Nous avons choisi que chaque opérateur, excepté les opérateurs d'opérations arithmétiques, serait un enfant direct de la super-classe Operator.
Par ailleurs, chaque classe-enfant ne comporte aucun attribut, excepté les classe NumberOperator et OperantOperator. NumberOperator possède un attribut privé et final. Ce choix nous permet de ne pas redéfinir une nouvelle classe pour chaque chiffre, ces derniers étant définis lors de l'instanciation de l'objet via le constructeur.
Nous discuterons de la classe Operand Operator dans le point suivant.

Opérations arithmétiques

Les opérations arithmétiques entre les différentes valeurs de la stack se font via la classe OperandOperaor.
Cette classe prend comme attribut une instance de la classe Operation<T> (ici T est un objet de type Double). La classe Operation¡T¿ est une classe publique abstraite qui implémente la méthode (abstraite également) apply(T a, T b). L'appel à cette fonction depuis OperandOperator permet de traiter les opérations arithmétiques proposées par les classes enfants de Operation.
Parmi les classes enfants de Operation, nous avons choisi de n'implémenter que les classes suivantes:

- Addition
- Subtraction
- Multiplication
- Division

Nous n'avons pas besoin de plus pour notre calculatrice, cependant nous gardons ainsi la possibilité d'implémenter d'autres opérations utilisant deux opérandes de manière simple et optimisée.

Gestion d'écriture et de mémoire

Comme demandé dans le cahier des charges, notre calculatrice comprend un bouton "C", permettant d'effacer ce qui était en train d'être écrit (classe CEOperator), ainsi qu'un bouton reprenant cette même fonctionnalité et supprimant par la même occasion le contenu de la stack (pile) de la calculatrice (classe COperator). Afin d'éviter toute redondance de code, nous avons décidé que la classe COperator serait une sous-classe de la classe

CEOperator. Ainsi, un simple appel à la super méthode nous permet d'accéder aux fonctionnalités de la super-classe CEOperator.
Notre calculatrice implémente également une fonctionnalité mémoire avec les boutons MR et MS. MS permet de sauver dans l'attribut memory de la classe State, un objet de type Double. Le bouton MR permet de récupérer la valeur sauvée dans la mémoire. Ces deux fonctionnalités fonctionnement respectivement grâce aux classes MSOperator et MROperator.
Nous avons choisi que lorsque la mémoire serait récupérée, la valeur contenue en mémoire ne serait pas affecter et resterait la même. De même lorsque les boutons C ou CE seraient sélectionnés, la mémoire n'est pas affectée par l'exécution des autres fonctionnalités.

Stack

La classe Stack<T> est indispensable au bon fonctionnnement de la calculatrice. C'est en effet l'élément qui sert à stocker toutes les valeurs entrées par l'utilisateur et l'endroit d'où les valeurs sont tirées pour être par la suite traitées (calculées).
Ayant nous-même créé la stack, nous avons décidé de créer une classe Node, représentant les éléments présents dans la Stack. La Stack en elle-même représente alors une chaîne, permettant de contrôler l'ordre d'arrivée et de sortie des éléments.

State et Operator

La classe State est celle qui centralise toutes les informations liées à la calculatrice. Elle prend comme seul attribut une instance de la classe Stack. La classe State renvoie les données qui seront affichées par la calculatrice, données qui sont modifiées par les instances des objets de type Operator.

Tests effectués

Pour les tests effectués, nous avons dans un premier temps tester les méthodes présentes dans State.java et Stack.java. Ceci inclus les toutes opérations pour ajouter, retirer, récupérer et afficher les éléments de la pile.
Une fois que tout fonctionnait dans State et dans la Stack, nous avons utilisé la calculatrice en mode console et en mode graphique. C'est à dire que nous avons testé les mêmes opérations dans les deux modes pour vérifier que les résultats étaient les mêmes, une fois en appelant les méthodes de la classe Calculator et une fois en simulant des cliques sur les boutons de la calculatrice graphique. Pour la partie graphique, nous avons implémenté une classe JCalculatorTestUtils qui possède les méthodes nécessaire à la simulation de cliques sur les boutons de la calculatrice.
Tous nos tests fonctionnent correctement et nous n'avons pas rencontré de problèmes majeurs lors de l'implémentation.

Conclusion

Nos deux programmes/calculatrices marchent tel que nous le souhaitons vu que les tests effectués se comportent comme prévus. Nous avons grâce à ce laboratoire pu apprendre comment fonctionne une architecture MVC et avons pu mettre en pratique notre code dans deux contextes différents, une fois sur l'interface graphique et la seconde fois dans l'extension, en mode console.
Nous avons ainsi appris que tout code bien construit peut être réutiliser dans plusieurs contextes différents.