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";
        }
    }

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

```java
 */
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;
    }
}
```