Folder calculator

11 printable files

(file list disabled)

calculator/.idea/.gitignore

```
# Default ignored files
/shelf/
/workspace.xml
# Editor-based HTTP Client requests
/httpRequests/
# Datasource local storage ignored files
/dataSources/
/dataSources.local.xml
```

calculator/.idea/misc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="ProjectRootManager" version="2" project-jdk-name="openjdk-23" project-jdk-type="JavaSDK">
    <output url="file://$PROJECT_DIR$/out" />
  </component>
</project>
```

calculator/.idea/modules.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="ProjectModuleManager">
    <modules>
      <module fileurl="file://$PROJECT_DIR$/calculator.iml" filepath="$PROJECT_DIR$/calculator.iml" />
    </modules>
  </component>
</project>
```

calculator/.idea/vcs.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project version="4">
  <component name="VcsDirectoryMappings">
    <mapping directory="$PROJECT_DIR$/.." vcs="Git" />
  </component>
</project>
```

calculator/Calculator.java

```
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
 */

package calculator;

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

public class Calculator {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        State state = State.getState();
        Map<String, Operator> commandMap = new HashMap<>();
        commandMap.put("+", new Addition());
```

```
        commandMap.put("-", new Subtraction());
        commandMap.put("*", new Multiplication());
        commandMap.put("/", new Division());
        commandMap.put("sqrt", new SquareRoot());
        commandMap.put("square", new Square());
        commandMap.put("inv", new Reciprocal());
        commandMap.put("clear", new Clear());

        System.out.println("Calculator (type 'exit' to quit)");
        while (true) {
            System.out.print("> ");
            String input = scanner.nextLine().trim();

            if (input.equalsIgnoreCase("exit")) {
                break;
            }
            if (input.matches("-?\\d+(\\.\\d+)?")) {
                double number = Double.parseDouble(input);
                state.setValue(number);
                state.pushToStack(number);
            }
            else if (commandMap.containsKey(input.toLowerCase())) {
                commandMap.get(input.toLowerCase()).execute();
                state.pushToStack(state.value());
            } else {
                System.out.println("Unknown command: " + input);
                continue;
            }
            System.out.println(state.stackToString());
        }

        scanner.close();
    }
}
```

calculator/CalculatorTests.java

```
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
 */

package calculator;

public class CalculatorTests {

    public static void main(String[] args) {
        // Créer une instance de la classe State pour gérer l'état de la calculatrice
        State state = State.getState();
        System.out.println("=== Tests Améliorés de la Calculatrice ===");

        // Fonction utilitaire pour tester les résultats
        System.out.println("Lancement des tests...");
        try {

            // Étape 1 : Empiler des valeurs et tester
            state.setValue(10.0);
            state.pushToStack(state.value());
            state.setValue(5.0);
            state.pushToStack(state.value());
            testResult(state.stackToString(), "[5.0, 10.0]", "Étape 1 - Empilement de 10 et 5");


            // Étape 2 : Addition (10 + 5)
            new Addition().execute();
```

```
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[15.0]", "Étape 2 - Addition (10 + 5)");

        // Étape 3 : Empiler une nouvelle valeur et multiplier
        state.setValue(3.0);
        state.pushToStack(state.value());
        new Multiplication().execute(); // Résultat * 3
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[45.0]", "Étape 3 - Multiplication par 3");

        // Étape 4 : Diviser par une nouvelle valeur
        state.setValue(9.0);
        state.pushToStack(state.value());
        new Division().execute(); // Résultat / 9
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[5.0]", "Étape 4 - Division par 9");

        // Étape 5 : Racine carrée
        state.setValue(4.0);
        state.pushToStack(state.value());
        new SquareRoot().execute();
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[2.0, 5.0]", "Étape 5 - Racine carrée de 4");

        // Étape 6 : Mise au carré
        new Square().execute();
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[4.0, 5.0]", "Étape 6 - Mise au carré");

        // Étape 7 : Inverse (1/x)
        new Reciprocal().execute();
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[0.25, 5.0]", "Étape 7 - Inverse (1/4)");

        // Étape 8 : Combinaison complexe
        state.setValue(2.0);
        state.pushToStack(state.value());
        new Multiplication().execute(); // Résultat * 2
        state.pushToStack(state.value());
        state.setValue(5.0);
        state.pushToStack(state.value());
        new Subtraction().execute(); // Résultat - 5
        state.pushToStack(state.value());
        testResult(state.stackToString(), "[-4.5, 5.0]", "Étape 8 - Combinaison complexe");

        // Étape 9 : Réinitialisation de la pile
        new Clear().execute();
        testResult(state.stackToString(), "[]", "Étape 9 - Réinitialisation (Clear)");

    } catch (Exception e) {
        System.out.println("Une erreur inattendue s'est produite: " + e.getMessage());
    }

    System.out.println("=== Fin des Tests Améliorés ===");
}

/**
 * Fonction utilitaire pour tester un résultat et afficher un message de succès ou d'échec.
 *
 * @param actual   Le résultat actuel obtenu
 * @param expected Le résultat attendu
 * @param testName Nom du test pour identification
 */
private static void testResult(String actual, String expected, String testName) {
    if (actual.equals(expected)) {
        System.out.println("✅ " + testName + " réussi ! Résultat: " + actual);
```

```
        } else {
            System.out.println("❌ " + testName + " échoué. Attendu: " + expected + ", Obtenu: " + actual);
        }
    }
}
```

calculator/JCalculator.java

```
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
 */

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;

/**
 * @class JCalculator
 * @brief Classe qui implémente l'interface graphique d'une calculatrice.
 *
 * La classe JCalculator permet de créer une interface utilisateur pour une calculatrice
 * avec diverses opérations arithmétiques et gestion de la pile.
 */
public class JCalculator extends JFrame
{
  /**
   * @brief Tableau représentant une pile vide.
   */
  private static final String[] empty = { "< empty stack >" };

  /**
   * @brief Zone de texte contenant la valeur introduite ou le résultat courant.
   */
  private final JTextField jNumber = new JTextField("0");

  /**
   * @brief Composant liste représentant le contenu de la pile.
   */
  private final JList jStack = new JList(empty);

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

  /**
   * @brief Met à jour l'interface graphique après une opération.
   *
   * Cette méthode met à jour la valeur dans la zone de texte jNumber et le contenu de la pile affichée dans
jStack.
   */
  private void update()
  {
```

```
    jNumber.setText(State.getState().getValueString());

    Object[] stackArray = State.getState().stackToArray(); // Récupère les éléments de la pile
    jStack.setListData(stackArray); // Met à jour la JList avec les éléments actuels de la pile

    if(State.getState().isStackEmpty())
      jStack.setListData(empty);
  }

  /**
   * @brief Ajoute un bouton à l'interface et associe une opération.
   *
   * @param name Nom du bouton.
   * @param x Position horizontale dans la grille.
   * @param y Position verticale dans la grille.
   * @param color Couleur du texte du bouton.
   * @param operator Instance de l'opérateur associé au bouton.
   *
   * Cette méthode crée un bouton avec un opérateur donné. Lorsqu'on clique sur le bouton, l'opérateur est
exécuté et l'interface est mise à jour.
   */
  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();
    });
  }

  /**
   * @brief Constructeur de la classe JCalculator.
   *
   * Ce constructeur initialise l'interface utilisateur de la calculatrice en configurant
   * la disposition et en ajoutant les différents boutons nécessaires.
   */
  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 MemoryRecall());

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

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

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

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

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

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

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

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

    // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
    addOperatorButton("1/x", 4, 2, Color.RED, new Reciprocal());
    addOperatorButton("x²", 4, 3, Color.RED, new Square());
    addOperatorButton("Sqrt", 4, 4, Color.RED, new SquareRoot());

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

    // 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);
  }
}
```

calculator/Operator.java

```
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
```

```
 */

package calculator;

/**
 * Classe abstraite représentant un opérateur.
 */
abstract class Operator {
    /**
     * Méthode abstraite d'exécution, à implémenter dans les sous-classes.
     */
    abstract void execute();

    /**
     * Exécute une opération binaire en utilisant les deux opérandes données.
     *
     * @param operand1 Premier opérande.
     * @param operand2 Deuxième opérande.
     * @param state    L'état actuel de la calculatrice.
     * @param operation L'opération binaire à exécuter.
     */
    protected void executeBinaryOperation(double operand1, double operand2, State state, BinaryOperation
operation) {
        try {
            double result = operation.compute(operand1, operand2);
            state.setValue(result);
        } catch (ArithmeticException e) {
            state.setError(e.getMessage());
        }
    }

    /**
     * Calcule une opération binaire entre deux opérandes.
     *
     * @param operand1 Premier opérande.
     * @param operand2 Deuxième opérande.
     * @return Le résultat du calcul.
     */
    double compute(double operand1, double operand2) {
        return 0;
    }
}

/**
 * Classe représentant un chiffre dans la calculatrice.
 */
class Digit extends Operator {
    int digit;

    /**
     * Constructeur de Digit.
     *
     * @param digit Le chiffre à représenter.
     */
    Digit(int digit) {
        this.digit = digit;
    }

    @Override
    void execute() {
        State state = State.getState();

        if (state.isWaitingForNextOperand()) {
            state.setValue(0); // Réinitialise l'affichage
            state.setWaitingForNextOperand(false); // Désactive l'attente
        }
```

```
            state.appendValue(digit); // Ajoute le chiffre
        }
    }

    /**
     * Classe représentant l'opérateur de suppression du dernier chiffre.
     */
    class BackSpace extends Operator {
        @Override
        void execute() {
            State.getState().delLastValue();
        }
    }

    /**
     * Classe représentant l'opérateur de suppression de l'erreur.
     */
    class ClearError extends Operator {
        @Override
        void execute() {
            State.getState().clearError();
        }
    }

    /**
     * Classe représentant l'opérateur de réinitialisation complète.
     */
    class Clear extends Operator {
        @Override
        void execute() {
            State.getState().clear();
        }
    }

    /**
     * Classe représentant l'opérateur de rappel de la mémoire.
     */
    class MemoryRecall extends Operator {
        @Override
        void execute() {
            State.getState().recallValue();
        }
    }

    /**
     * Classe représentant l'opérateur de stockage de la valeur en mémoire.
     */
    class MemoryStore extends Operator {
        @Override
        void execute() {
            State.getState().storeValue();
        }
    }

    /**
     * Classe représentant l'opérateur de changement de signe.
     */
    class ChangeSign extends Operator {
        @Override
        void execute() {
            State.getState().changeSign();
        }
    }

    /**
```

```java
 * Classe représentant l'opérateur d'ajout d'un point décimal.
 */
class AppendDot extends Operator {
    @Override
    void execute() {
        State.getState().appendDot();
    }
}

/**
 * Classe abstraite pour les opérations unaires.
 */
abstract class UnaryOperation extends Operator {
    private final UnaryFunction operation;

    /**
     * Constructeur d'opération unaire.
     *
     * @param operation La fonction unaire à appliquer.
     */
    UnaryOperation(UnaryFunction operation) {
        this.operation = operation;
    }

    @Override
    void execute() {
        State state = State.getState();
        if (state.stackSize() < 1) {
            System.out.println("Not enough operands in the stack.");
            return;
        }

        double operand = state.popFromStack();
        try {
            double result = operation.apply(operand);
            state.setValue(result);
        } catch (ArithmeticException e) {
            state.setError(e.getMessage());
        }
        state.prepareForNextOperand();
    }

    /**
     * Interface fonctionnelle pour les fonctions unaires.
     */
    @FunctionalInterface
    interface UnaryFunction {
        double apply(double operand);
    }
}

/**
 * Classe représentant l'opération de carré.
 */
class Square extends UnaryOperation {
    Square() {
        super(operand -> operand * operand);
    }
}

/**
 * Classe représentant l'opération de racine carrée.
 */
class SquareRoot extends UnaryOperation {
    SquareRoot() {
        super(operand -> {
```

```
            if (operand < 0) throw new ArithmeticException("Cannot compute square root of a negative number.")
;
            return Math.sqrt(operand);
        });
    }
}

/**
 * Classe représentant l'opération de réciproque (1/x).
 */
class Reciprocal extends UnaryOperation {
    Reciprocal() {
        super(operand -> {
            if (operand == 0) throw new ArithmeticException("Cannot compute reciprocal of zero.");
            return 1 / operand;
        });
    }
}

/**
 * Classe abstraite pour les opérations binaires.
 */
abstract class BinaryOperation extends Operator {
    @Override
    void execute() {
        State state = State.getState();
        if (state.stackSize() < 2) {
            System.out.println("Not enough operands in the stack.");
            return;
        }
        double operand2 = state.popFromStack();
        double operand1 = state.popFromStack();
        executeBinaryOperation(operand1, operand2, state, this);
        state.prepareForNextOperand();
    }

    /**
     * Calcule une opération binaire entre deux opérandes.
     *
     * @param operand1 Premier opérande.
     * @param operand2 Deuxième opérande.
     * @return Le résultat du calcul.
     */
    abstract double compute(double operand1, double operand2);
}

/**
 * Classe représentant l'opération d'addition.
 */
class Addition extends BinaryOperation {
    @Override
    double compute(double operand1, double operand2) {
        return operand1 + operand2;
    }
}

/**
 * Classe représentant l'opération de soustraction.
 */
class Subtraction extends BinaryOperation {
    @Override
    double compute(double operand1, double operand2) {
        return operand1 - operand2;
    }
}
```

```java
/**
 * Classe représentant l'opération de multiplication.
 */
class Multiplication extends BinaryOperation {
    @Override
    double compute(double operand1, double operand2) {
        return operand1 * operand2;
    }
}

/**
 * Classe représentant l'opération de division.
 */
class Division extends BinaryOperation {
    @Override
    double compute(double operand1, double operand2) {
        if (operand2 == 0) {
            throw new ArithmeticException("Illegal division");
        }
        return operand1 / operand2;
    }
}

/**
 * Classe représentant l'opérateur Enter.
 */
class Enter extends Operator {
    @Override
    void execute() {
        State state = State.getState();
        double currentValue = state.value();
        state.pushToStack(currentValue);
        state.prepareForNextOperand();
        Operator currentOperator = state.getCurrentOperator();
        if (currentOperator != null) {
            double operand1 = state.popFromStack();
            double operand2 = currentValue;
            try {
                double result = currentOperator.compute(operand1, operand2);
                state.setValue(result);
            } catch (ArithmeticException e) {
                state.setError(e.getMessage());
            }
            state.setCurrentOperator(null);
            state.setWaitingForNextOperand(false);
        }
    }
}
```

calculator/Stack.java

```java
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
 */

package calculator;

import java.util.Arrays;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Classe représentant une pile générique.
 *
```

```java
 * @param <T> Le type d'éléments stockés dans la pile.
 */
public class Stack<T> implements Iterable<T> {
    private T[] elements;
    private int size = 0;
    private static final int INITIAL_CAPACITY = 10;

    /**
     * Constructeur de la classe Stack.
     * Initialise la pile avec une capacité initiale définie.
     */
    @SuppressWarnings("unchecked")
    public Stack() {
        elements = (T[]) new Object[INITIAL_CAPACITY];
    }

    /**
     * Empile un élément sur la pile.
     *
     * @param item L'élément à empiler.
     */
    public void push(T item) {
        if (size == elements.length) {
            resize(2 * elements.length);  // double la capacité si nécessaire
        }
        elements[size++] = item;
    }

    /**
     * Désempile un élément de la pile.
     *
     * @return L'élément désemparé.
     * @throws NoSuchElementException si la pile est vide.
     */
    public T pop() {
        if (size == 0) {
            throw new NoSuchElementException("La pile est vide");
        }
        T item = elements[--size];
        elements[size] = null;  // pour éviter les fuites de mémoire
        return item;
    }

    /**
     * Retourne une représentation sous forme de chaîne de caractères du contenu de la pile.
     *
     * @return Une chaîne représentant les éléments de la pile.
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        for (int i = size - 1; i >= 0; i--) {
            sb.append(elements[i]);
            if (i > 0) sb.append(", ");
        }
        sb.append("]");
        return sb.toString();
    }

    /**
     * Retourne un tableau représentant l'état actuel de la pile.
     *
     * @return Un tableau contenant les éléments de la pile.
     */
    public T[] toArray() {
        return Arrays.copyOfRange(elements, 0, size);
```

```java
}

/**
 * Retourne un itérateur pour parcourir les éléments de la pile.
 *
 * @return Un itérateur sur les éléments de la pile.
 */
@Override
public Iterator<T> iterator() {
    return new StackIterator();
}

/**
 * Vérifie si la pile est vide.
 *
 * @return true si la pile est vide, sinon false.
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Retourne la taille actuelle de la pile.
 *
 * @return Le nombre d'éléments dans la pile.
 */
public int size() {
    return size;
}

/**
 * Classe interne représentant un itérateur pour la pile.
 */
private class StackIterator implements Iterator<T> {
    private int current = size - 1;

    /**
     * Vérifie s'il reste des éléments à parcourir dans la pile.
     *
     * @return true s'il reste des éléments, sinon false.
     */
    @Override
    public boolean hasNext() {
        return current >= 0;
    }

    /**
     * Retourne l'élément suivant dans la pile.
     *
     * @return L'élément suivant.
     * @throws NoSuchElementException si aucun élément n'est disponible.
     */
    @Override
    public T next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return elements[current--];
    }
}

/**
 * Redimensionne la capacité du tableau d'éléments.
 *
 * @param newCapacity La nouvelle capacité du tableau.
 */
```

```
    @SuppressWarnings("unchecked")
    private void resize(int newCapacity) {
        elements = Arrays.copyOf(elements, newCapacity);
    }
}
```

calculator/State.java

```
/**
 * @author Aubry Antoine
 * @author Faria dos Santos Dani Tiago
 */

package calculator;

/**
 * Classe représentant l'état de la calculatrice.
 * Gère la valeur courante, la mémoire, les erreurs, et la pile des opérations.
 */
public class State {
    private static State instance;
    private String value = "";
    private String memory = "0";
    private String error = "";
    private boolean hasError = false;
    private boolean isMutable = true;
    private Operator currentOperator = null;
    private boolean waitingForNextOperand = false;
    private boolean clearedOnNextInput = false;

    /**
     * Constructeur par défaut de la classe State.
     */
    public State() {}

    private Stack<Double> stack = new Stack<>();

    /**
     * Empile une valeur sur la pile.
     *
     * @param value La valeur à empiler.
     */
    public void pushToStack(double value) {
        stack.push(value);
    }

    /**
     * Désempile une valeur de la pile.
     *
     * @return La valeur désemparée.
     */
    public double popFromStack() {
        if (stack.isEmpty()) {
            setError("Stack is empty");
            return 0;
        }
        return stack.pop();
    }

    /**
     * Retourne une représentation sous forme de chaîne de caractères du contenu de la pile.
     *
     * @return Une chaîne représentant les éléments de la pile.
     */
    public String stackToString() {
```

```
    return stack.toString();
}

/**
 * Retourne un tableau représentant l'état actuel de la pile.
 *
 * @return Un tableau contenant les éléments de la pile.
 */
public Object[] stackToArray() {
    return stack.toArray();
}

/**
 * Retourne l'instance unique de l'état.
 *
 * @return L'instance unique de la classe State.
 */
public static State getState() {
    if (instance == null) {
        instance = new State();
    }
    return instance;
}

/**
 * Efface les erreurs actuelles et réinitialise l'état mutable.
 */
public void clearError() {
    value = "";
    error = "";
    hasError = false;
    isMutable = true;
}

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

/**
 * Efface la valeur actuelle et réinitialise la pile.
 */
public void clear() {
    value = "";
    isMutable = true;
    clearStack();
}

/**
 * Vide la pile des valeurs.
 */
private void clearStack() {
    while (!stack.isEmpty()) {
        stack.pop();  // Dépile tous les éléments
    }
}

/**
 * Ajoute une valeur entière au champ de saisie.
 *
 * @param x La valeur à ajouter.
 */
```

```java
public void appendValue(int x) {
    if (clearedOnNextInput) {
        value = "";
        clearedOnNextInput = false;
    }
    value += x;
}

/**
 * Ajoute un point décimal à la valeur actuelle.
 */
public void appendDot() {
    if (value.isEmpty()) value += "0";
    if (!value.contains(".")) value += ".";
}

/**
 * Prépare l'état pour la saisie du prochain opérande.
 */
public void prepareForNextOperand() {
    clearedOnNextInput = true;
    waitingForNextOperand = true;
}

/**
 * Change le signe de la valeur actuelle.
 */
public void changeSign() {
    if (!hasError) {
        double val = value();
        value = val > 0 ? "-" + value : value.substring(1);
    }
}

/**
 * Supprime le dernier chiffre de la valeur actuelle.
 */
public void delLastValue() {
    if (isMutable && !value.isEmpty()) {
        value = value.substring(0, value.length() - 1);
    }
}

/**
 * Stocke la valeur actuelle dans la mémoire.
 */
public void storeValue() {
    if (!hasError) {
        memory = value.isEmpty() ? "0" : value;
    }
}

/**
 * Rappelle la valeur stockée en mémoire.
 */
public void recallValue() {
    value = memory;
    isMutable = false;
}

/**
 * Retourne la valeur actuelle sous forme de chaîne de caractères.
 *
 * @return La valeur actuelle ou l'erreur si elle est présente.
 */
public String getValueString() {
```

```
    return hasError ? error : (value.isEmpty() ? "0" : value);
}

/**
 * Retourne la valeur actuelle sous forme de double.
 *
 * @return La valeur actuelle.
 */
public double value() {
    try {
        return value.isEmpty() ? 0 : Double.parseDouble(value);
    } catch (NumberFormatException e) {
        setError("Invalid number format: " + value);
        return 0;
    }
}

/**
 * Définit la valeur actuelle.
 *
 * @param x La valeur à définir.
 */
public void setValue(double x) {
    value = formatValue(x);
    isMutable = false;
}

/**
 * Formate la valeur pour supprimer la partie décimale si elle est inutile.
 *
 * @param x La valeur à formater.
 * @return La valeur formatée sous forme de chaîne.
 */
private String formatValue(double x) {
    return (x == (long) x) ? String.valueOf((long) x) : Double.toString(x);
}

/**
 * Définit le message d'erreur actuel.
 *
 * @param errorMessage Le message d'erreur.
 */
public void setError(String errorMessage) {
    error = errorMessage;
    hasError = true;
}

/**
 * Définit l'opérateur courant.
 *
 * @param operator L'opérateur à définir.
 */
public void setCurrentOperator(Operator operator) {
    currentOperator = operator;
}

/**
 * Retourne l'opérateur courant.
 *
 * @return L'opérateur courant.
 */
public Operator getCurrentOperator() {
    return currentOperator;
}

/**
```

```java
     * Vérifie si l'état attend la saisie d'un prochain opérande.
     *
     * @return true si l'état attend un prochain opérande, sinon false.
     */
    public boolean isWaitingForNextOperand() {
        return waitingForNextOperand;
    }

    /**
     * Définit si l'état attend la saisie d'un prochain opérande.
     *
     * @param waiting true pour attendre un prochain opérande, sinon false.
     */
    public void setWaitingForNextOperand(boolean waiting) {
        waitingForNextOperand = waiting;
    }

    /**
     * Retourne la taille actuelle de la pile.
     *
     * @return Le nombre d'éléments dans la pile.
     */
    public int stackSize() {
        return stack.size();
    }
}
```

calculator/calculator.iml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module type="JAVA_MODULE" version="4">
  <component name="NewModuleRootManager" inherit-compiler-output="true">
    <exclude-output />
    <content url="file://$MODULE_DIR$">
      <sourceFolder url="file://$MODULE_DIR$" isTestSource="false" packagePrefix="calculator" />
    </content>
    <orderEntry type="inheritedJdk" />
    <orderEntry type="sourceFolder" forTests="false" />
  </component>
</module>
```