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";

        // 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 !");
    }
}
```