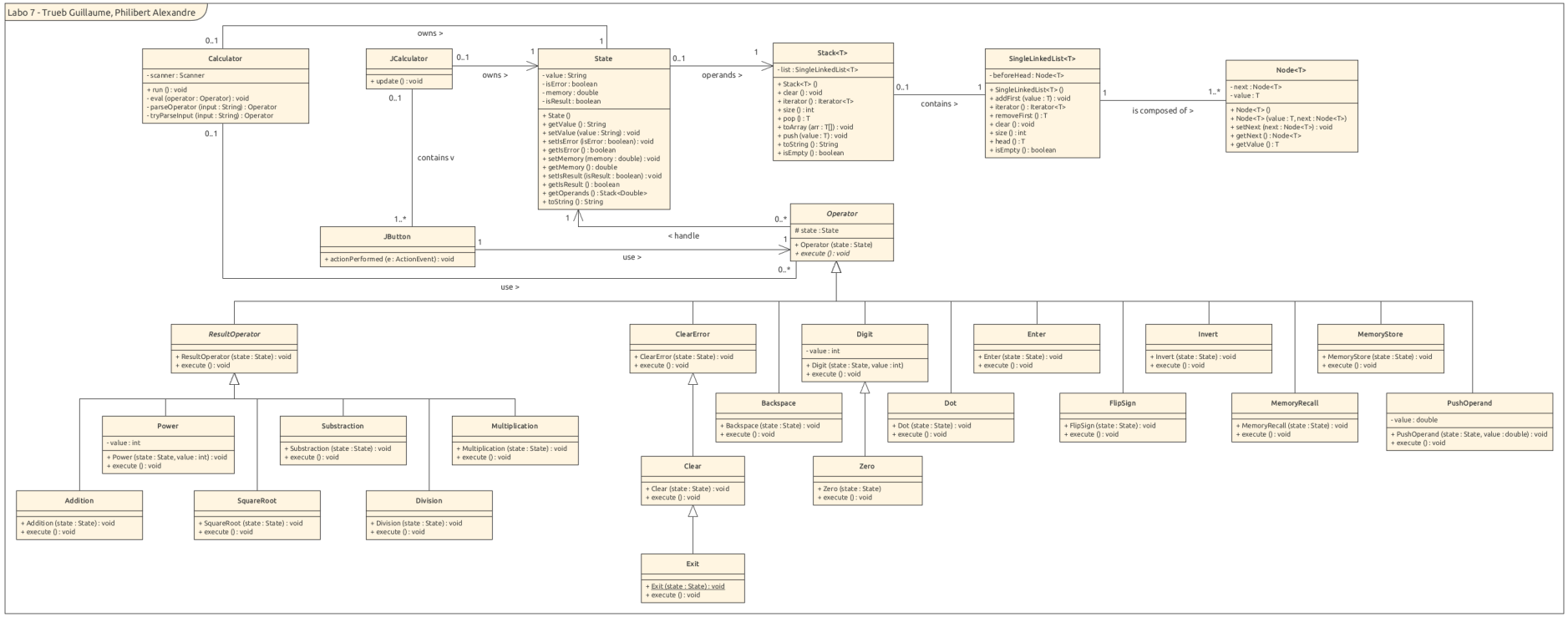
POO - Labo 7 Calculator

Alexandre Philibert **Guillaume Trüeb**

1 Diagramme


Labo 7 - Trueb Guillaume, Philibert Alexandre
Calculator
- scanner : Scanner
+ run () : void
- eval (operator : Operator) : void
- parseOperator (input : String) : Operator
- tryParseInput (input : String) : Operator
JCalculator
+ update () : void
State
- value : String
- isError : boolean
- memory : double
- isResult : boolean
+ State ()
+ getValue () : String
+ setValue (value : String) : void
+ setIsError (isError : boolean) : void
+ getIsError () : boolean
+ setMemory (memory : double) : void
+ getMemory () : double
+ setIsResult (isResult : boolean) : void
+ getIsResult () : boolean
+ getOperands () : Stack<Double>
+ toString () : String
Stack<T>
- list : SingleLinkedList<T>
+ Stack<T> ()
+ clear () : void
+ iterator () : Iterator<T>
+ size () : int
+ pop () : T
+ toArray (arr : T[]) : void
+ push (value : T) : void
+ toString () : String
+ isEmpty () : boolean
SingleLinkedList<T>
- beforeHead : Node<T>
+ SingleLinkedList<T> ()
+ addFirst (value : T) : void
+ iterator () : Iterator<T>
+ removeFirst () : T
+ clear () : void
+ size () : int
+ head () : T
+ isEmpty () : boolean
Node<T>
- next : Node<T>
- value : T
+ Node<T> ()
+ Node<T> (value : T, next : Node<T>)
+ setNext (next : Node<T>) : void
+ getNext () : Node<T>
+ getValue () : T
JButton
+ actionPerformed (e : ActionEvent) : void
Operator
state :State
+ Operator (state : State)
+ execute () : void
owns >
owns >
operands >
contains >
is composed of >
contains v
< handle
use >
use >
ResultOperator
+ ResultOperator (state : State) : void
+ execute () : void
Power
- value : int
+ Power (state : State, value : int) : void
+ execute () : void
Substraction
+ Substraction (state : State) : void
+ execute () : void
Multiplication
+ Multiplication (state : State) : void
+ execute () : void
Addition
+ Addition (state : State) : void
+ execute () : void
SquareRoot
+ SquareRoot (state : State) : void
+ execute () : void
Division
+ Division (state : State) : void
+ execute () : void
ClearError
+ ClearError (state : State) : void
+ execute () : void
Digit
- value : int
+ Digit (state : State, value : int)
+ execute () : void
Enter
+ Enter (state : State) : void
+ execute () : void
Invert
+ Invert (state : State) : void
+ execute () : void
MemoryStore
+ MemoryStore (state : State) : void
+ execute () : void
Backspace
+ Backspace (state : State) : void
+ execute () : void
Dot
+ Dot (state : State) : void
+ execute () : void
FlipSign
+ FlipSign (state : State) : void
+ execute () : void
MemoryRecall
+ MemoryRecall (state : State) : void
+ execute () : void
PushOperand
- value : double
+ PushOperand (state : State, value : double) : void
+ execute () : void
Clear
+ Clear (state : State) : void
+ execute () : void
Zero
+ Zero (state : State)
+ execute () : void
Exit
+ Exit (state : State) : void
+ execute () : void

2 Choix et hypothèses de travail

- La classe `Calculator` défini une classe interne `Exit` qui hérite de `Clear`, cela permet, lors de l'extension de la calculette terminal, d'exécuter du code avant la fermeture de l'application.

3 Rapport de tests

Classe	Nom	Description	Résultat
AdditionTest	shouldAddValues()	Valide qu'Addition additionne correctement la valeur est la première opérande	OK
AdditionTest	shouldNotAddWhenOperandEmpty()	Valide qu'Addition n'effectue pas l'addition si les opérandes sont vides	OK
AdditionTest	shouldNotAddWhenIsError()	Valide qu'Addition n'effectue pas l'addition si le state est en erreur	OK
BackspaceTest	shouldRemoveValue()	Valide que Backspace enlève le dernier	OK
BackspaceTest	shouldReplaceWithZeroIfValueBecomesEmpty()	Valide qu'un zéro est inséré dans la valeur courrant lorsque celle-ci devient vide lors de l'exécution de Backspace	OK
ClearErrorTest	shouldClearError()	Valide que l'erreur est supprimée du state lorsque ClearError est exécuté	OK
ClearErrorTest	shouldClearValue()	Valide que la valeur courante est réinitialisé à 0 lorsque ClearError est exécuté	OK
ClearTest	shouldClearError()	Valide que l'erreur est supprimée du state lorsque Clear est exécuté	OK
ClearTest	shouldClearOperands()	Valide que la pile des opérandes est vidée lorsque Clear est appelé	OK
DigitTest	shouldAppendDigitToValue()	Valide que Digit ajoute un chiffre à la valeur courante	OK
DigitTest	shouldPushOperandWhenIsResult()	Valide que la valeur courante est poussée sur la pile d'opérandes lorsque le state est en mode résultat	OK
DigitTest	shouldReplaceZeroWhenZeroIsNegative()	Valide que lorsque la valeur courante vaut « −0 », elle est remplacée par « -{value} »	OK
DivisionTest	shouldDivideValues()	Valide que Division effectue une division entre la valeur courante et la première opérande	OK
DivisionTest	shouldNotDivideWhenOperandEmpty()	Valide que Division n'effectue pas de division lorsque la pile d'opérandes est vide	OK

DivisionTest	shouldNotDivideWhenIsError()	Valide que Division n'effectue pas de division lorqsue	OK
DivisionTest	shouldErrorWhenOperandsEmpty()	Valide que Division mets le state en erreur lorsque la pile d'opérandes est vide	OK
DivisionTest	shouldErrorWhenDividingByZero()	Valide que Division mets le state en erreur lorsqu'une division par 0 est effectuée	OK
DotTest	shouldPutDot()	Valide que Dot ajoute un « . »	OK
DotTest	shouldNotPutDotIfValueAlreadyHasOne()	Valide que Dot n'ajoute pas un deuxième point à la valeur courante	OK
EnterTest	shouldPushValueToOperands()	Valide que Enter ajoute la valeur sur la pile d'opérandes	OK
FlipSignTest	shouldFlipSignPositiveToNegative()	Valide que FlipSign inverse le signe de la valeur courante de positif vers négatif	OK
FlipSignTest	shouldFlipSignNegativeToPositive()	Valide que FlipSign inverse le signe de la valeur courante de néfatif vers positif	OK
FlipSignTest	shouldFlipSignWhenValueIsZero()	Valide que « 0 » est inversé en « −0 »	OK
InvertTest	shouldInvertValues()	Valide que Invert inverse la valeur courante	OK
InvertTest	shouldErrorWhenValueIsZero()	Valide que Invert mets le state en erreur lorsque la valeur courante vaut « 0 »	OK
MemoryRecallTest	shouldRecallStoredValue()	Valide que RecallStored mets la valeur courante à la valeur de la mémoire du state	OK
MemoryRecallTest	shouldNotRecallWhenIsError()	Valide que RecallStored ne change pas la valeur courante lorsque le state est en erreur	OK
MemoryStoreTest	shouldStoreValue()	Valide que MemoryStore mets la valeur courante dans la mémoire du state	OK
MultiplicationTest	shouldMultiplyValues()	Valide que la multiplication effectue la multiplication de la valeur courante et de la première opérande	OK
MultiplicationTest	shouldNotMultiplyWhenOperandEmpty()	Valide que Multiplication n'effectue pas la multiplication lorsque la pile d'opérandes est vide	OK

MultiplicationTest	shouldNotMultiplyWhenIsError()	Valide que Multipilcation n'effectue pas la multiplication lorsque le state est en erreur	OK
PowerTest	shouldSquare()	Valide que Power effectue la puissance de la valeur courante en fonction du paramètre value	OK
SquareRootTest	shouldSqrtValues()	Valide que SquareRoot effectue la racine carée de la valeur courante	OK
SquareRootTest	shouldNotSqrtWhenIsError()	Valide que SquareRoot mets le state en erreur lorsque la valeur courante est un nombre négatif	OK
SubtractionTest	shouldSubtractValues()	Valide que Subtraction effectue la soustraction de la valeur courante et de la première opérande	OK
SubtractionTest	shouldNotSubtractWhenOperandEmpty()	Valide que Subtraction n'effectue pas la soustraction si la pile d'opérandes est vide	OK
SubtractionTest	shouldNotSubtractWhenIsError()	Valide que Substraction n'effectue pas la soustraction lorsque le state est en erreur	OK
ZeroTest	shouldNotAddZeroIfAlreadyZero()	Valide que Zero n'ajoute pas un deuxième zéro à la valeur courante	OK
SingleLinkedListTest	shouldAddAtBeginning()	Valide que addFirst() ajoute un élément au début de la liste	OK
SingleLinkedListTest	listShouldBeEmpty()	Valide que isEmpty() retourne `true` lorsque la liste est vide	OK
SingleLinkedListTest	listShouldNotBeEmpty()	Valide que isEmpty() retourne `false` lorsque la liste contient un élément	OK
SingleLinkedListTest	listShouldHaveSizeZero()	Valide size() est « 0 » lorsque la liste ne continent pas de valeurs	OK
SingleLinkedListTest	listShouldHaveSize()	Valide que size() retourne la bonne taille de la liste	OK
SingleLinkedListTest	listShouldRemoveFirst()	Valide que removeFirst() enlève le premier élément de la liste	OK
SingleLinkedListTest	iteratorShouldIterate()	Valide que l'itérateur de la liste itère sur les éléments	OK
StackTest	shouldHaveCorrectLength()	Valide que le stack à la bonne taille selon le nombre d'éléments contenu	OK

StackTest	shouldHaveCorrectPopValue()	Valide que pop() retourne la valeur de l'élément du haut de la pile	OK
StackTest	shouldBeEmpty()	Valide que isEmpty() retourne `true` sur un stack vide	OK
StackTest	shouldEmptyStackIteratorHasNextBeEmpty()	Valide que hasNext() d'une liste vide retourne `false`	OK
StackTest	shouldIteratorIterate()	Valide que l'itérateur du stack itère sur les éléments	OK
StackTest	shouldFillArrayWithStackValues()	Valide que toArray() rempli le tableau passé en argument	OK
StackTest	shouldClearStack()	Valide que clear() vide le stack	OK

File - Main.java

```java
import calculator.JCalculator;

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

File - Calculator.java

```java
package calculator;

import calculator.operator.*;

import java.util.Scanner;

/**
 * @author Philibert Alexandre, Trüeb Guillaume
 *
 * Provides a basic calculator on the standard output.
 */
public class Calculator {

    private final State state;

    private final Scanner scanner;

    public Calculator() {
        state = new State();
        scanner = new Scanner(System.in);
    }

    /**
     * Starts the calculator
     */
    public void run() {
        Operator operator;

        do {
            System.out.print("> ");

            operator = parseOperator(scanner.nextLine().strip());

            if (operator != null) {
                eval(operator);
            } else {
                System.out.println("# unknown operation #");
            }

            if (state.getIsError()) {
                System.out.println("# error #");
            }

            System.out.println(state.getOperands());
        } while (!(operator instanceof Exit));
    }

    /**
     * Eval part of the REPL
     * @param operator The operator to execute
     */
    private void eval(Operator operator) {
        if (!(operator instanceof PushOperand)) {
            state.setValue(Double.toString(state.getOperands().pop()));
        }

        operator.execute();

        if (state.getIsResult()) {
            state.getOperands().push(Double.parseDouble(state.getValue()));
            state.setIsResult(false);
        }
    }

    /**
     * Returns an Operator to execute given an input string.
     *
     * @param input The string from which to derive the operator to execute
     * @return The operator derived from the input string
     */
```

File - Calculator.java

```
71      private Operator parseOperator(String input) {
72          return switch(input) {
73              case "+" -> new Addition(state);
74              case "-" -> new Subtraction(state);
75              case "*" -> new Multiplication(state);
76              case "/" -> new Division(state);
77              case "square" -> new Power(state, 2);
78              case "sqrt" -> new SquareRoot(state);
79              case "clear" -> new Clear(state);
80              case "clearerr" -> new ClearError(state);
81              case "ms" -> new MemoryStore(state);
82              case "mr" -> new MemoryRecall(state);
83              case "flip" -> new FlipSign(state);
84              case "invert" -> new Invert(state);
85              case "exit" -> new Exit(state);
86              default -> tryParseInput(input);
87          };
88      }
89
90      /**
91       * Tries to parse the input as a double, if it fails return null.
92       */
93      private Operator tryParseInput(String input) {
94          try {
95              return new PushOperand(state, Double.parseDouble(input));
96          } catch (NumberFormatException e) {
97              return null;
98          }
99      }
100
101     /**
102      * This operator should be called when exiting the calculator, it provides a place to handle
    logic when exiting
103      * the calculator.
104      */
105     private static class Exit extends Clear {
106         public Exit(State state) {
107             super(state);
108         }
109
110         @Override
111         public void execute() {}
112     }
113 }
114
```

File - JCalculator.java

```java
package calculator;

import calculator.operator.*;

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

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

//import java.awt.event.*;

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

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

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

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

    private final State state;

    // Mise a jour de l'interface apres une operation (jList et jStack)
    private void update()
    {
        if (state.getIsError()) {
            jNumber.setText("# error #");
        } else {
            jNumber.setText(state.toString());
        }

        Double[] operands = new Double[state.getOperands().size()];
        state.getOperands().toArray(operands);

        String[] stringOperands = new String[state.getOperands().size()];
        for (int i = 0; i < state.getOperands().size(); i++) {
            stringOperands[i] = Double.toString(operands[i]);
        }

        jStack.setListData(stringOperands.length == 0 ? empty : stringOperands);
    }

    // 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();
        });
    }
```

```
 71
 72      public JCalculator()
 73      {
 74          super("JCalculator");
 75          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 76          getContentPane().setLayout(new GridBagLayout());
 77
 78          state = new State();
 79
 80          // Contraintes des composants graphiques
 81          constraints.insets = new Insets(3, 3, 3, 3);
 82          constraints.fill = GridBagConstraints.HORIZONTAL;
 83
 84          // Nombre courant
 85          jNumber.setEditable(false);
 86          jNumber.setBackground(Color.WHITE);
 87          jNumber.setHorizontalAlignment(JTextField.RIGHT);
 88          constraints.gridx = 0;
 89          constraints.gridy = 0;
 90          constraints.gridwidth = 5;
 91          getContentPane().add(jNumber, constraints);
 92          constraints.gridwidth = 1; // reset width
 93
 94          // Rappel de la valeur en memoire
 95          addOperatorButton("MR", 0, 1, Color.RED, new MemoryRecall(state));
 96
 97          // Stockage d'une valeur en memoire
 98          addOperatorButton("MS", 1, 1, Color.RED, new MemoryStore(state));
 99
100          // Backspace
101          addOperatorButton("<=", 2, 1, Color.RED, new Backspace(state));
102
103          // Mise a zero de la valeur courante + suppression des erreurs
104          addOperatorButton("CE", 3, 1, Color.RED, new ClearError(state));
105
106          // Comme CE + vide la pile
107          addOperatorButton("C",  4, 1, Color.RED, new Clear(state));
108
109          // Boutons 1-9
110          for (int i = 1; i < 10; i++)
111              addOperatorButton(String.valueOf(i), (i - 1) % 3, 4 - (i - 1) / 3,
112                      Color.BLUE, new Digit(state, i));
113          // Bouton 0
114          addOperatorButton("0", 0, 5, Color.BLUE, new Zero(state));
115
116          // Changement de signe de la valeur courante
117          addOperatorButton("+/-", 1, 5, Color.BLUE, new FlipSign(state));
118
119          // Operateur point (chiffres apres la virgule ensuite)
120          addOperatorButton(".", 2, 5, Color.BLUE, new Dot(state));
121
122          // Operateurs arithmetiques a deux operandes: /, *, -, +
123          addOperatorButton("/", 3, 2, Color.RED, new Division(state));
124          addOperatorButton("*", 3, 3, Color.RED, new Multiplication(state));
125          addOperatorButton("-", 3, 4, Color.RED, new Subtraction(state));
126          addOperatorButton("+", 3, 5, Color.RED, new Addition(state));
127
128          // Operateurs arithmetiques a un operande: 1/x, x^2, Sqrt
129          addOperatorButton("1/x", 4, 2, Color.RED, new Invert(state));
130          addOperatorButton("x^2", 4, 3, Color.RED, new Power(state, 2));
131          addOperatorButton("Sqrt", 4, 4, Color.RED, new SquareRoot(state));
132
133          // Entree: met la valeur courante sur le sommet de la pile
134          addOperatorButton("Ent", 4, 5, Color.RED, new Enter(state));
135
136          // Affichage de la pile
137          JLabel jLabel = new JLabel("Stack");
138          jLabel.setFont(new Font("Dialog", 0, 12));
139          jLabel.setHorizontalAlignment(JLabel.CENTER);
140          constraints.gridx = 5;
```

```
141         constraints.gridy = 0;
142         getContentPane().add(jLabel, constraints);
143 
144         jStack.setFont(new Font("Dialog", 0, 12));
145         jStack.setVisibleRowCount(8);
146         JScrollPane scrollPane = new JScrollPane(jStack);
147         constraints.gridx = 5;
148         constraints.gridy = 1;
149         constraints.gridheight = 5;
150         getContentPane().add(scrollPane, constraints);
151         constraints.gridheight = 1; // reset height
152 
153         setResizable(false);
154         pack();
155         setVisible(true);
156     }
157 }
```

File - State.java

```java
package calculator;

import util.Stack;


/**
 * @author Philibert Alexandre, Trüeb Guillaume
 *
 * Represents the state of a calculator and provides ways to interact with it.
 */
public class State {

    /**
     * The current value of the calculator
     */
    private String value;

    private boolean isError;

    private final Stack<Double> operands;

    private double memory;

    private boolean isResult;

    public State() {
        operands = new Stack<>();
        value = "0";
        isResult = false;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public void setIsError(boolean isError) {
        this.isError = isError;
    }

    public boolean getIsError() {
        return isError;
    }

    public void setMemory(double memory) {
        this.memory = memory;
    }

    public double getMemory() {
        return memory;
    }

    public boolean getIsResult() {
        return isResult;
    }

    public void setIsResult(boolean isResult) {
        this.isResult = isResult;
    }

    public Stack<Double> getOperands() {
        return operands;
    }

    @Override
    public String toString() {
        return value.isEmpty() ? "0" : value;
```

File - State.java

```
71      }
72  }
73
```

File - SingleLinkedList.java

```java
package util.SingleLinkedList;

import java.util.Iterator;

public class SingleLinkedList<T> implements Iterable<T> {
    private final Node<T> beforeHead;

    public SingleLinkedList() {
        beforeHead = new Node<>(null);
    }

    public void addFirst(T value) {
        Node<T> currentHead = beforeHead.getNext();
        beforeHead.setNext(new Node<T>(value, currentHead));
    }

    public boolean isEmpty() {
        return beforeHead.getNext() == null;
    }

    public void clear() {
        beforeHead.setNext(null);
    }

    public T head() {
        Node<T> head = beforeHead.getNext();

        return head == null ? null : head.getValue();
    }

    public T removeFirst() {
        Node<T> head = beforeHead.getNext();
        T value = head.getValue();
        beforeHead.setNext(head.getNext());

        return value;
    }

    public int size() {
        int size = 0;

        Node<T> curr = beforeHead;
        while ((curr = curr.getNext()) != null) {
            size++;
        }

        return size;
    }

    @Override
    public Iterator<T> iterator() {
        return new SingleLinkedListIterator<>(beforeHead);
    }

    private static class Node<T> {
        private final T value;

        private Node<T> next;

        public Node(T value) {
            this.value = value;
        }

        public Node(T value, Node<T> next) {
            this(value);
            this.next = next;
        }

        public void setNext(Node<T> next) {
            this.next = next;
```

```java
        }

        public Node<T> getNext() {
            return next;
        }

        public T getValue() {
            return value;
        }
    }

    public static class SingleLinkedListIterator<T> implements Iterator<T> {

        private Node<T> curr;

        public SingleLinkedListIterator(Node<T> beforeHead) {
            this.curr = beforeHead;
        }

        @Override
        public boolean hasNext() {
            return curr.getNext() != null;
        }

        @Override
        public T next() {
            curr = curr.getNext();

            return curr.getValue();
        }
    }
}

```

File - Stack.java

```java
package util;

import util.SingleLinkedList.SingleLinkedList;

import java.util.Iterator;
import java.util.StringJoiner;

public class Stack<T> implements Iterable<T> {

    private final SingleLinkedList<T> list;

    public Stack() {
        list = new SingleLinkedList<>();
    }

    public void push(T value) {
        list.addFirst(value);
    }

    public T pop() {
        return list.removeFirst();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public void clear() {
        list.clear();
    }

    public void toArray(T[] arr) {
        if (arr.length < list.size()) {
            throw new RuntimeException("Array is smaller than Stack");
        }

        Iterator<T> it = iterator();
        for (int i = 0; i < arr.length; ++i) {
            arr[i] = it.next();
        }
    }

    public int size() {
        return list.size();
    }

    @Override
    public String toString() {
        StringJoiner s = new StringJoiner(" ");
        Iterator<T> it = iterator();

        while (it.hasNext()) {
            s.add(it.next().toString());
        }

        return s.toString();
    }

    @Override
    public Iterator<T> iterator() {
        return list.iterator();
    }
}

```

File - Addition.java

```java
package calculator.operator;

import calculator.State;
import util.Stack;

public class Addition extends ResultOperator {
    public Addition(State state) {
        super(state);
    }

    public void execute() {
        if (state.getIsError() || state.getOperands().isEmpty()) {
            return;
        }

        super.execute();

        double value = Double.parseDouble(state.getValue());
        Stack<Double> operands = state.getOperands();

        state.setValue(Double.toString(value + operands.pop()));
    }
}

```

File - Backspace.java

```java
package calculator.operator;

import calculator.State;

public class Backspace extends Operator {
    public Backspace(State state) {
        super(state);
    }

    @Override
    public void execute() {
        String value = state.getValue();

        if (!value.isEmpty()) {
            String newValue = value.substring(0, value.length() - 1);

            if (newValue.isEmpty()) {
                newValue = "0";
            }

            state.setValue(newValue);
        }
    }
}

```

File - Clear.java

```java
package calculator.operator;

import calculator.State;

public class Clear extends ClearError {
    public Clear(State state) {
        super(state);
    }

    @Override
    public void execute() {
        super.execute();

        state.getOperands().clear();
    }
}

```

File - ClearError.java

```java
package calculator.operator;

import calculator.State;

public class ClearError extends Operator {
    public ClearError(State state) {
        super(state);
    }

    @Override
    public void execute() {
        state.setIsError(false);
        state.setValue("0");
    }
}

```

File - Digit.java

```java
package calculator.operator;

import calculator.State;

public class Digit extends Operator {

    private final int value;

    public Digit(State state, int value) {
        super(state);
        this.value = value;
    }

    @Override
    public void execute() {
        String value = state.getValue();

        if (value.equals("0")) {
            value = "";
        } else if (value.equals("-0")) {
            value = "-";
        }

        if (state.getIsResult()) {
            state.getOperands().push(Double.parseDouble(state.getValue()));
            state.setIsResult(false);
            value = Integer.toString(this.value);
        } else {
            value += this.value;
        }


        state.setValue(value);
    }
}

```

File - Division.java

```java
package calculator.operator;

import calculator.State;
import util.Stack;

public class Division extends ResultOperator {
    public Division(State state) {
        super(state);
    }

    public void execute() {
        if (state.getIsError()) {
            return;
        }

        if (state.getOperands().isEmpty()) {
            state.setIsError(true);
            return;
        }

        super.execute();

        double value = Double.parseDouble(state.getValue());
        Stack<Double> operands = state.getOperands();

        state.setValue(Double.toString(operands.pop() / value));
    }
}

```

File - Dot.java

```java
package calculator.operator;

import calculator.State;

/**
 * Adds a dot to the state value only if none is present
 */
public class Dot extends Operator {
    public Dot(State state) {
        super(state);
    }

    @Override
    public void execute() {
        String value = state.getValue();

        if (!value.contains(".")) {
            // Double.parseDouble needs a "0" in front of the ".", this helps make the parsing easier
            if (value.isEmpty()) {
                value += "0";
            }

            state.setValue(value + '.');
        }
    }
}
```

File - Enter.java

```java
package calculator.operator;

import calculator.State;

/**
 * Push the value onto the operands stack then sets the current value to 0
 */
public class Enter extends Operator {
    public Enter(State state) {
        super(state);
    }

    @Override
    public void execute() {
        if (state.getIsError()) {
          return;
        }

        state.getOperands().push(Double.parseDouble(state.getValue()));
        state.setValue("0");
    }
}

```

File - FlipSign.java

```java
package calculator.operator;

import calculator.State;

/**
 * Flips the sign of the current value
 */
public class FlipSign extends Operator {
    public FlipSign(State state) {
        super(state);
    }

    @Override
    public void execute() {
        String value = state.getValue();

        if (!value.isEmpty() && value.charAt(0) == '-') {
            state.setValue(value.substring(1));
        } else {
           if (value.isEmpty()) {
               value += 0;
           }
            state.setValue("-" + value);
        }
    }
}

```

File - Invert.java

```java
package calculator.operator;

import calculator.State;

/**
 * Inverts the current value such that x becomes 1/x
 */
public class Invert extends Operator {
    public Invert(State state) {
        super(state);
    }

    @Override
    public void execute() {
        double value = Double.parseDouble(state.getValue());

        if (value == 0) {
            state.setIsError(true);
            return;
        }

        state.setValue(Double.toString(1/value));
    }
}

```

File - MemoryRecall.java

```java
package calculator.operator;

import calculator.State;

public class MemoryRecall extends Operator {
    public MemoryRecall(State state) {
        super(state);
    }

    @Override
    public void execute() {
        state.setValue(Double.toString(state.getMemory()));
    }
}

```

File - MemoryStore.java

```java
package calculator.operator;

import calculator.State;

public class MemoryStore extends Operator {
    public MemoryStore(State state) {
        super(state);
    }

    @Override
    public void execute() {
        if (state.getIsError()) {
            return;
        }

        state.setMemory(Double.parseDouble(state.getValue()));
    }
}

```

File - Multiplication.java

```java
package calculator.operator;

import calculator.State;
import util.Stack;

public class Multiplication extends ResultOperator {
    public Multiplication(State state) {
        super(state);
    }

    public void execute() {
        if (state.getIsError() || state.getOperands().isEmpty()) {
            return;
        }

        super.execute();

        double value = Double.parseDouble(state.getValue());
        Stack<Double> operands = state.getOperands();

        state.setValue(Double.toString(value * operands.pop()));
    }
}

```

File - Operator.java

```java
package calculator.operator;

import calculator.State;

/**
 * The Operator class provides a way to interact with a calculator state
 */
public abstract class Operator
{
    protected final State state;

    public Operator(State state) {
        this.state = state;
    }

    public abstract void execute();
}
```

File - Power.java

```java
package calculator.operator;

import calculator.State;

public class Power extends ResultOperator {

    private final int value;

    public Power(State state, int value) {
        super(state);
        this.value = value;
    }

    @Override
    public void execute() {
        if (state.getIsError()) {
            return;
        }

        super.execute();

        double value = Double.parseDouble(state.getValue());
        state.setValue(Double.toString(Math.pow(value, this.value)));
    }
}

```

File - PushOperand.java

```java
package calculator.operator;

import calculator.State;

public class PushOperand extends Operator {
    private final double value;

    public PushOperand(State state, double value) {
        super(state);
        this.value = value;
    }

    @Override
    public void execute() {
        state.getOperands().push(value);
    }
}

```

File - ResultOperator.java

```java
package calculator.operator;

import calculator.State;

/**
 * The result operator puts the state in the result mode.
 * This class should be used for all operators that are expected to push the computed value into the stack once an
 * operand is executed.
 */
public abstract class ResultOperator extends Operator {
    public ResultOperator(State state) {
        super(state);
    }

    @Override
    public void execute() {
        state.setIsResult(true);
    }
}

```

File - SquareRoot.java

```java
package calculator.operator;

import calculator.State;

public class SquareRoot extends ResultOperator {
    public SquareRoot(State state) {
        super(state);
    }

    @Override
    public void execute() {
        if (state.getIsError()) {
            return;
        }

        double value = Double.parseDouble(state.getValue());

        if (value < 0) {
            state.setIsError(true);
            return;
        }

        super.execute();

        state.setValue(Double.toString(Math.sqrt(value)));
    }
}

```

File - Subtraction.java

```java
package calculator.operator;

import calculator.State;
import util.Stack;

public class Subtraction extends ResultOperator {
    public Subtraction(State state) {
        super(state);
    }

    public void execute() {
        if (state.getIsError() || state.getOperands().isEmpty()) {
            return;
        }

        super.execute();

        double value = Double.parseDouble(state.getValue());
        Stack<Double> operands = state.getOperands();

        state.setValue(Double.toString(value - operands.pop()));
    }
}

```

File - Zero.java

```java
package calculator.operator;

import calculator.State;

import java.util.Objects;


public class Zero extends Digit {
    public Zero(State state) {
        super(state, 0);
    }

    @Override
    public void execute() {
        String value = state.getValue();

        if (!Objects.equals(value, "") && Double.parseDouble(value) != 0.0) {
            super.execute();
        }
    }
}

```

File - AdditionTest.java

```java
package calculator.operator;


import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class AdditionTest {
    @Test
    public void shouldAddValues() {
        State state = new State();
        state.setValue("3.5");
        state.getOperands().push(4.0);

        new Addition(state).execute();

        assertEquals("7.5", state.getValue());
        assertTrue(state.getIsResult());
    }

    @Test
    public void shouldNotAddWhenOperandEmpty() {
        State state = new State();
        state.setValue("5");

        new Addition(state).execute();

        assertEquals("5", state.getValue());
    }

    @Test
    public void shouldNotAddWhenIsError() {
        State state = new State();
        state.setIsError(true);
        state.setValue("5");
        state.getOperands().push(3.0);

        new Addition(state).execute();

        assertEquals("5", state.getValue());
    }
}

```

File - BackspaceTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class BackspaceTest {
    @Test
    public void shouldRemoveValue() {
        State state = new State();
        state.setValue("23.5");

        new Backspace(state).execute();

        assertEquals("23.", state.getValue());
    }

    @Test
    public void shouldReplaceWithZeroIfValueBecomesEmpty() {
        State state = new State();
        state.setValue("2");

        new Backspace(state).execute();

        assertEquals("0", state.getValue());
    }
}

```

File - ClearErrorTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class ClearErrorTest {
    @Test
    public void shouldClearError() {
        State state = new State();
        state.setIsError(true);

        new ClearError(state).execute();

        assertFalse(state.getIsError());
    }

    @Test
    public void shouldClearValue() {
        State state = new State();
        state.setIsError(true);
        state.setValue("3.5");

        new ClearError(state).execute();

        assertEquals("0", state.getValue());
    }
}

```

File - ClearTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class ClearTest {
    @Test
    public void shouldClearError() {
        State state = new State();
        state.setIsError(true);

        new Clear(state).execute();

        assertFalse(state.getIsError());
    }

    @Test
    public void shouldClearOperands() {
        State state = new State();
        state.getOperands().push(3.0);
        state.getOperands().push(2.5);

        new Clear(state).execute();

        assertEquals(0, state.getOperands().size());
    }
}

```

File - DivisionTest.java

```java
package calculator.operator;


import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class DivisionTest {
    @Test
    public void shouldDivideValues() {
        State state = new State();
        state.setValue("2.0");
        state.getOperands().push(6.0);

        new Division(state).execute();

        assertEquals("3.0", state.getValue());
        assertTrue(state.getIsResult());
    }

    @Test
    public void shouldNotDivideWhenOperandEmpty() {
        State state = new State();
        state.setValue("5");

        new Division(state).execute();

        assertEquals("5", state.getValue());
    }

    @Test
    public void shouldNotDivideWhenIsError() {
        State state = new State();
        state.setIsError(true);
        state.setValue("5");
        state.getOperands().push(3.0);

        new Division(state).execute();

        assertEquals("5", state.getValue());
    }

    @Test
    public void shouldErrorWhenOperandsEmpty() {
        State state = new State();
        state.setValue("6");

        new Division(state).execute();

        assertTrue(state.getIsError());
    }

    @Test
    public void shouldErrorWhenDividingByZero() {
        State state = new State();
        state.getOperands().push(3.0);
        state.setValue("0.0");

        new Division(state).execute();

        assertEquals("Infinity", state.getValue());
    }
}

```

File - EnterTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class EnterTest {
    @Test
    public void shouldPushValueToOperands() {
        State state = new State();
        state.setValue("3.75");

        new Enter(state).execute();

        assertEquals(3.75, state.getOperands().pop());
        assertEquals("0", state.getValue());
    }
}

```

File - DigitTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class DigitTest {
    @Test
    public void shouldAppendDigitToValue() {
        State state = new State();
        state.setValue("3");

        new Digit(state, 5).execute();

        assertEquals("35", state.getValue());
    }

    @Test
    public void shouldPushOperandWhenIsResult() {
        State state = new State();
        state.setValue("12.5");
        state.setIsResult(true);

        new Digit(state, 4).execute();

        assertEquals("4", state.getValue());
        assertEquals(12.5, state.getOperands().pop());
    }

    @Test
    public void shouldReplaceZeroWhenZeroIsNegative() {
        State state = new State();
        state.setValue("-0");

        new Digit(state, 3).execute();

        assertEquals("-3", state.getValue());
    }
}

```

File - DotTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class DotTest {
    @Test
    public void shouldPutDot() {
        State state = new State();
        state.setValue("32");

        new Dot(state).execute();

        assertEquals("32.", state.getValue());
    }

    @Test
    public void shouldNotPutDotIfValueAlreadyHasOne() {
        State state = new State();
        state.setValue("45.03");

        new Dot(state).execute();

        assertEquals("45.03", state.getValue());
    }
}

```

File - FlipSignTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class FlipSignTest {
    @Test
    public void shouldFlipSignPositiveToNegative() {
        State state = new State();
        state.setValue("3.15");

        new FlipSign(state).execute();

        assertEquals("-3.15", state.getValue());
    }

    @Test
    public void shouldFlipSignNegativeToPositive() {
        State state = new State();
        state.setValue("-3.15");

        new FlipSign(state).execute();

        assertEquals("3.15", state.getValue());
    }

    @Test
    public void shouldFlipSignWhenValueIsZero() {
        State state = new State();
        state.setValue("0");

        new FlipSign(state).execute();

        assertEquals("-0", state.getValue());
    }
}

```

File - InvertTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class InvertTest {
    @Test
    public void shouldInvertValues() {
        State state = new State();
        state.setValue("10.0");

        new Invert(state).execute();

        assertEquals("0.1", state.getValue());
    }

    @Test
    public void shouldErrorWhenValueIsZero() {
        State state = new State();
        state.setValue("0.0");

        new Invert(state).execute();

        assertTrue(state.getIsError());
    }
}

```

File - MemoryRecallTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class MemoryRecallTest {
    @Test
    public void shouldRecallStoredValue() {
        State state = new State();
        state.setMemory(4.75);

        new MemoryRecall(state).execute();

        assertEquals("4.75", state.getValue());
    }

    @Test
    public void shouldNotRecallWhenIsError() {
        State state = new State();
        state.setMemory(4.75);
        state.setIsError(true);

        new MemoryRecall(state).execute();

        assertTrue(state.getIsError());
    }
}

```

File - MultiplicationTest.java

```java
package calculator.operator;


import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class MultiplicationTest {
    @Test
    public void shouldMultiplyValues() {
        State state = new State();
        state.setValue("2");
        state.getOperands().push(3.0);

        new Multiplication(state).execute();

        assertEquals("6.0", state.getValue());
        assertTrue(state.getIsResult());
    }

    @Test
    public void shouldNotMultiplyWhenOperandEmpty() {
        State state = new State();
        state.setValue("5");

        new Multiplication(state).execute();

        assertEquals("5", state.getValue());
    }

    @Test
    public void shouldNotMultiplyWhenIsError() {
        State state = new State();
        state.setIsError(true);
        state.setValue("5");
        state.getOperands().push(3.0);

        new Multiplication(state).execute();

        assertEquals("5", state.getValue());
    }
}

```

File - PowerTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class PowerTest {
    @Test
    public void shouldSquare() {
        State state = new State();
        state.setValue("3.0");

        new Power(state, 2).execute();

        assertEquals("9.0", state.getValue());
    }
}

```

File - SquareRootTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class SquareRootTest {
    @Test
    public void shouldSqrtValues() {
        State state = new State();
        state.setValue("4");

        new SquareRoot(state).execute();

        assertEquals("2.0", state.getValue());
    }

    @Test
    public void shouldNotSqrtWhenIsError() {
        State state = new State();
        state.setIsError(true);
        state.setValue("4.0");

        new SquareRoot(state).execute();

        assertEquals("4.0", state.getValue());
    }
}

```

File - SubtractionTest.java

```java
package calculator.operator;


import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class SubtractionTest {
    @Test
    public void shouldSubtractValues() {
        State state = new State();
        state.setValue("7.5");
        state.getOperands().push(4.0);

        new Subtraction(state).execute();

        assertEquals("3.5", state.getValue());
        assertTrue(state.getIsResult());
    }

    @Test
    public void shouldNotSubtractWhenOperandEmpty() {
        State state = new State();
        state.setValue("5");

        new Subtraction(state).execute();

        assertEquals("5", state.getValue());
    }

    @Test
    public void shouldNotSubtractWhenIsError() {
        State state = new State();
        state.setIsError(true);
        state.setValue("5");
        state.getOperands().push(3.0);

        new Subtraction(state).execute();

        assertEquals("5", state.getValue());
    }
}

```

File - ZeroTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class ZeroTest {
    @Test
    public void shouldNotAddZeroIfAlreadyZero() {
        State state = new State();
        state.setValue("0");

        new Zero(state).execute();

        assertEquals("0", state.getValue());
    }
}

```

File - MemoryStoreTest.java

```java
package calculator.operator;

import calculator.State;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

public class MemoryStoreTest {
    @Test
    public void shouldStoreValue() {
        State state = new State();
        state.setValue("4.25");

        new MemoryStore(state).execute();

        assertEquals(4.25, state.getMemory());
    }
}

```

File - SingleLinkedListTest.java

```java
package util;

import org.junit.jupiter.api.Test;
import util.SingleLinkedList.SingleLinkedList;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;


public class SingleLinkedListTest {

    @Test
    void shouldAddAtBeginning() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        list.addFirst(6);
        list.addFirst(3);

        assertEquals(3, list.head());
    }

    @Test
    void listShouldBeEmpty() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        assertTrue(list.isEmpty());
    }

    @Test
    void listShouldNotBeEmpty() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        list.addFirst(7);

        assertFalse(list.isEmpty());
    }

    @Test
    void listShouldHaveSizeZero() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        assertEquals(0, list.size());
    }

    @Test
    void listShouldHaveSize() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        list.addFirst(2);
        list.addFirst(2);
        list.addFirst(2);

        assertEquals(3, list.size());
    }

    @Test
    void listShouldRemoveFirst() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();

        list.addFirst(3);
        list.addFirst(7);
        list.removeFirst();

        assertEquals(1, list.size());
    }

    @Test
    void iteratorShouldIterate() {
        SingleLinkedList<Integer> list = new SingleLinkedList<>();
```

```
71
72          list.addFirst(3);
73          list.addFirst(2);
74          list.addFirst(1);
75
76          assertIterableEquals(List.of(1,2,3), list);
77      }
78  }
79
```

File - StackTest.java

```java
package util;

import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

public class StackTest {
    @Test
    public void shouldHaveCorrectLength() {
        Stack<Integer> stack = new Stack<>();

        stack.push(34);
        stack.push(5);
        stack.push(1);

        assertEquals(3, stack.size());
    }

    @Test
    public void shouldHaveCorrectPopValue() {
        Stack<Integer> stack = new Stack<>();

        stack.push(34);
        stack.push(5);

        assertEquals(5, stack.pop());
    }

    @Test
    public void shouldBeEmpty() {
        Stack<Integer> stack = new Stack<>();

        stack.push(3);
        stack.pop();

        assertTrue(stack.isEmpty());
    }

    @Test
    public void shouldEmptyStackIteratorHasNextBeEmpty() {
        Stack<Integer> stack = new Stack<>();

        assertFalse(stack.iterator().hasNext());
    }

    @Test
    public void shouldIteratorIterate() {
        Stack<Integer> stack = new Stack<>();

        stack.push(45);
        stack.push(8);
        stack.push(6);

        assertIterableEquals(List.of(6, 8, 45), stack);
    }

    @Test
    public void shouldFillArrayWithStackValues() {
        Stack<Integer> stack = new Stack<>();

        stack.push(5);
        stack.push(6);
        stack.push(7);

        Integer[] arr = new Integer[3];
        stack.toArray(arr);

        assertArrayEquals(new Integer[]{7, 6, 5}, arr);
```

File - StackTest.java

```
71      }
72
73      @Test
74      public void shouldClearStack() {
75          Stack<Integer> stack = new Stack<>();
76
77          stack.push(1);
78          stack.push(2);
79          stack.push(3);
80
81          stack.clear();
82
83          assertTrue(stack.isEmpty());
84      }
85  }
86
```