**CSC 151 Assignment #5**

1. **Honor Code**

*For group assignments (when allowed):*

*We affirm that we have carried out our academic endeavors with full academic honesty.*

*[Signed, Manav Bilakhia, Jason D’Amico, Saeed AlSuwaidi ]*

1. Resources/References
2. **Java files and outputs**
3. Java files

Class: StackInterface.java

/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
package assignment;  
  
import java.util.EmptyStackException;  
  
*/\*\*  
 \* An interface for the ADT stack.  
 \*  
 \** ***@author*** *Frank M. Carrano  
 \** ***@author*** *Timothy M. Henry  
 \** ***@version*** *5.0  
 \*/*public interface StackInterface<T> {  
 */\*\*  
 \* Adds a new entry to the top of this stack.  
 \*  
 \** ***@param*** *newEntry An object to be added to the stack.  
 \*/* public void push(T newEntry);  
 */\*\*  
 \* Removes and returns this stack's top entry.  
 \*  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *EmptyStackException if the stack is empty before the operation.  
 \*/* public T pop()throws EmptyStackException;  
 */\*\*  
 \* Retrieves this stack's top entry.  
 \*  
 \** ***@return*** *The object at the top of the stack.  
 \** ***@throws*** *EmptyStackException if the stack is empty.  
 \*/* public T peek()throws EmptyStackException;  
 */\*\*  
 \* Detects whether this stack is empty.  
 \*  
 \** ***@return*** *True if the stack is empty.  
 \*/* public boolean isEmpty();  
 */\*\* Removes all entries from this stack. \*/* public void clear();  
}

Class: OurStack.java

package assignment;  
import java.util.Stack;  
*/\*\*  
\* A class of stacks.  
\*  
\** ***@author*** *Frank M. Carrano  
\** ***@version*** *5.0  
\*/*/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class OurStack<T> implements StackInterface<T> {  
 private Stack<T> theStack;  
 // Implement the constructor with no parameter, push, peek, pop, isEmpty and clear  
 // by using the private Stack theStack of type java.util.Stack  
  
 public OurStack() {  
 this.theStack = new Stack<T>();  
 }  
  
 public void push(T newEntry) {  
 this.theStack.push(newEntry);  
 }  
  
 public T pop() {  
 return this.theStack.pop();  
 }  
  
 public T peek() {  
 return this.theStack.peek();  
 }  
  
 public boolean isEmpty() {  
 return this.theStack.isEmpty();  
 }  
  
 public void clear() {  
 this.theStack.clear();  
 }  
  
 public static void main(String[] args) {  
 Stack<String> testStack = new Stack<>();  
 System.*out*.println("New stack is empty: " + testStack.isEmpty());  
   
 System.*out*.println("Adding four entries to the stack...");  
  
 for (int i = 0; i < 4; i++) {  
 System.*out*.println("Item added: " + i);  
 testStack.add("entry " + i);  
 }  
  
   
 }  
}

Class LispToken.java

package assignment;  
*/\*\*  
\* This class represents either an operand or an operator for an arithmetic  
\* expressions in Lisp.  
\*  
\** ***@author*** *Charles Hoot  
\** ***@version*** *5.0  
\*/*/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class LispToken {  
 private Character operator;  
 private Double operand;  
 private boolean isOperator;  
  
 */\*\*  
 \* Constructor for objects of class LispToken for operators.  
 \* isOperator is true and operand is 0.0, operator is anOperator  
 \*  
 \** ***@param*** *anOperator of type Character  
 \*/* public LispToken(Character anOperator) {  
 this.operator = anOperator;  
 this.operand = 0.0;  
 this.isOperator = true;  
 }  
  
 */\*\*  
 \* Constructor for objects of class LispToken for operands.  
 \* isOperator is false and operand is the value, operator is ' '  
 \*  
 \** ***@param*** *value of type Double  
 \*/* public LispToken(Double value) {  
 this.operator = ' ';  
 this.operand = value;  
 this.isOperator = false;  
 }  
  
 */\*\*  
 \* applyOperator: Applies this operator to two given operand values.  
 \*  
 \** ***@param*** *value1 The value of the first operand.  
 \** ***@param*** *value2 The value of the second operand.  
 \** ***@return*** *The real result of the operation.  
 \*/* public Double applyOperator(Double value1, Double value2) {  
 if (this.operator == '\*') {  
 return value1 \* value2;  
 } else if (this.operator == '/') {  
 return value1 / value2;  
 } else if (this.operator == '+') {  
 return value1 + value2;  
 } else {  
 // Assertion: operator is '-'  
 return value1 - value2;  
 }  
 }  
  
 */\*\*  
 \* getIdentity: Gets the identity value of this operator. For example, x + 0 = x, so 0 is the  
 \* identity for + and will be the value associated with the expression (+).  
 \*  
 \** ***@return*** *The identity value of the operator as Double.  
 \*/* public Double getIdentity() {  
 if (this.operator == '+' || this.operator == '-') {  
 return 0.0;  
 } else {  
 // Assertion: operator is '/' or '\*', which both have an identity value of 1  
 return 1.0;  
 }  
 }  
  
 */\*\*  
 \* takesZeroOperands: Detects whether this operator returns a value when it has no operands.  
 \*  
 \** ***@return*** *True if the operator returns a value when it has no operands, or  
 \* false if not.  
 \*/* public boolean takesZeroOperands() {  
 // Only two operators that take zero operands are \* and +  
 return this.operator == '\*' || this.operator == '+';  
 }  
  
 */\*\*  
 \* getValue: Gets the value of this operand.  
 \*  
 \** ***@return*** *The real value of the operand.  
 \*/* public Double getValue() {  
 return this.operand;  
 }  
  
 */\*\*  
 \* isOperator: Returns true if the object is an operator.  
 \*  
 \** ***@return*** *True is this object is an operator.  
 \*/* public boolean isOperator() {  
 return this.isOperator;  
 }  
  
 */\*\*  
 \* toString: Returns a string representation of the operator or operand  
 \*  
 \** ***@return*** *String  
 \*/* public String toString() {  
 if (this.isOperator()) {  
 return this.operator.toString();  
 } else {  
 return this.operand.toString();  
 }  
 }  
}

Class: LispExpressionEvaluator.java

package assignment;  
import java.util.ArrayList;  
import java.util.EmptyStackException;  
import java.util.Scanner;  
*/\*\*  
 \* This class evaluates a simple arithmetic Lisp expression of numeric values.  
 \*  
 \** ***@author*** *Charles Hoot  
 \** ***@author*** *Jesse Grabowski  
 \** ***@author*** *Joseph Erickson  
 \** ***@author*** *Zeynep Orhan modified  
 \** ***@version*** *5.0  
 \*/*/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class LispExpressionEvaluator {  
 */\*\*  
 \* Evaluates a Lisp expression.  
 \*  
 \* The algorithm: Scan the tokens in the string.  
 \* If you see "(", push the next operator onto the stack.  
 \* If you see an operand, push it onto the stack.  
 \* If you see ")", Pop operands and push them onto a second stack until you find an  
 \* operator. Apply the operator to the operands on the second stack. Push the  
 \* result on the stack.  
 \*  
 \* What may occur? (Samples only)  
 \* If you run out of tokens, the value on the top  
 \* of the stack is the value of the expression.  
 \* OR  
 \* How to detect illegal expressions:  
 \* If you read numeric values and the  
 \* expression stack is empty  
 \* Error message: Bad Expression: needs an operator for the data  
 \*  
 \* If there is a ) and the expression stack is empty  
 \* Error message: mismatched )  
 \*  
 \* If there is a ) and operands needed but the expression stack is empty  
 \* Error message: mismatched )  
 \*  
 \* If the top operator requires at least one operand but it is not in the expression stack  
 \* Error message:operator nameOfTheOperand + " requires at least one operand"  
 \*  
 \* If the string does not have any more characters but the expression stack is not empty  
 \* Error message:incomplete expression / multiple expressions  
 \*  
 \* If the operator is not one of the +, -, \*, ?  
 \* Error message: found an operator when we should not  
 \*  
 \* If the expression is legal  
 \* Message:" and evaluates to " + whateverTheResultIs  
 \*  
 \*  
 \*  
 \*  
 \* Format of sample messages:  
 \* Message for a legal expression  
 \*  
 \* The expression '(+ (- 1) (\* 3 3 4) (/ 3 2 3) (\* 4 4))' is legal in Lisp:  
 \* and evaluates to 51.5  
 \*  
 \* Message for an illegal expression  
 \*  
 \* The expression '(+ (-) (\* 3 3 4) (/ 3 2 3) (\* 4 4))' is not legal in Lisp:  
 \* operator - requires at least one operand  
 \*  
 \** ***@param*** *lispExp A string that is a valid lisp expression.  
 \** ***@param*** *mes An ArrayList of strings that stores the messages generated.  
 \** ***@return*** *A double that is the value of the expression.  
 \*/* @SuppressWarnings("resource")  
 public static double evaluate(String lispExp, ArrayList<String> mes) {  
 StackInterface<LispToken> expressionStack = new OurStack<>();  
 StackInterface<LispToken> secondStack = new OurStack<>();  
 boolean nextIsOperator = false;  
 Scanner lispExpScanner = new Scanner(lispExp);  
 // Use zero or more white spaces as delimiter  
 // that breaks the string into single characters  
 lispExpScanner = lispExpScanner.useDelimiter("\\s\*");  
 boolean legal = true;  
 String errorMessage = "";  
 // Hint: use  
 // lispExpScanner.hasNext() to test if there are more tokens  
 // lispExpScanner.hasNextInt() to test if there is an integer  
 // lispExpScanner.next() to get the next String  
  
 while (lispExpScanner.hasNext() && legal) {  
 // Handles next value in scanner  
  
 if (lispExpScanner.hasNextInt()) {  
 double expression = lispExpScanner.nextDouble();  
 expressionStack.push(new LispToken(expression));  
 System.*out*.println("pushing int" + expression);  
 } else {  
 // Assertion: next value in scanner is a character (i.e., non-numeric value)  
 char next = lispExpScanner.next().charAt(0);  
 System.*out*.println(next + "read");  
 if (next == '(') {  
 // Action: push next operator onto the stack  
  
 if (!lispExpScanner.hasNextInt()) {  
 // Assertion: next value in scanner is an operator  
 char Operator = lispExpScanner.next().charAt(0);  
 expressionStack.push(new LispToken(Operator));  
 System.*out*.println("pushing" + Operator);  
 } else {  
 // Assertion: error; next number is not an operator  
 legal = false;  
 errorMessage = "unknown operator";  
 }  
 } else if (next == ')') {  
 if (expressionStack.isEmpty()) {  
 // Assertion: expression stack is empty, thus no operation can be performed  
 legal = false;  
 errorMessage = "mismatched )";  
 } else {  
 int counter = 0;  
  
 // Action: pop operands until an operator is found  
  
 while (!expressionStack.isEmpty() && !expressionStack.peek().isOperator()) {  
 // Assertion: current top item of stack is an operand  
 LispToken operand = expressionStack.pop();  
 secondStack.push(operand);  
 counter++;  
 }  
  
 if (expressionStack.isEmpty()) {  
 // Since the stack is empty, there is no operator at the end of the list of operands (thus causing a mismatched ')' error)  
 legal = false;  
 errorMessage = "mismatched )";  
 } else {  
 // Action: Performing operation  
  
 LispToken operator = expressionStack.pop();  
 Double result = 0.0;  
  
 if (counter >= 1) {  
 result = secondStack.pop().getValue();  
  
 if (counter == 1) {  
 // *TODO: Need a comment here explaining why identity is used* result = operator.applyOperator(operator.getIdentity(), result);  
 } else {  
 while(!secondStack.isEmpty()) {  
 result = operator.applyOperator(result, secondStack.pop().getValue());  
 }  
 }  
 } else if (counter == 0) {  
 if (operator.takesZeroOperands()) {  
 result = operator.getIdentity();  
 } else {  
 legal = false;  
 errorMessage = "operator " + operator + " requires at least one operand";  
 }  
 }  
  
 // Result of operation pushed to expression stack  
 expressionStack.push(new LispToken(result));  
 }  
 }  
 } else {  
 // Assertion: character is an operator (should only be added after parentheses)  
 legal = false;  
 errorMessage = "found an operator when we should not";  
 }  
 }  
 }  
  
 // Message creation  
  
 String message = "";  
 double value = 0.0;  
  
 // Message handling for legal lisp  
 if (legal) {  
 value = expressionStack.pop().getValue();  
  
 if (!expressionStack.isEmpty()) {  
 // Assertion: the lisp was processed without any errors, but there are multiple items in the expression stack (implying an incomplete expression / multiple expressions error)  
 errorMessage = "incomplete expression / multiple expressions";  
 legal = false;  
 }  
  
 message = "The expression '" + lispExp + "'\nis legal in Lisp:\nand evaluates to " + value + "\n";  
 }  
  
 // Message handling for illegal lisp  
 if (!legal) {  
 message = "The expression '" + lispExp + "'\nis not legal in Lisp:\n" + errorMessage + "\n";  
  
 value = -1.0;  
 }  
  
 mes.add(message);  
 return value;  
 }  
  
 public static void main(String args[]) {  
 String tests[] = {  
 "(+ 1 3)",  
 "(- 1)",  
 "(-)",  
 "(+)",  
 "(\*)",  
 "(/)",  
 "(- 1 2)",  
 "(+ (- 1) (\* 3 3 4) (/ 3 2 3) (\* 4 4))",  
 "(+ (-) (\* 3 3 4) (/ 3 2 3) (\* 4 4))",  
 "(+ (- 1) (\* 3 3 4) ) 5 (\* (/ 3 2 3) (\* 4 4))",  
 "(+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))",  
 "+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))",  
 };  
 ArrayList<String> mes = new ArrayList<>();  
 for (int i = 0; i < tests.length; i++) {  
 *evaluate*(tests[i], mes);  
 System.*out*.println(mes.get(i));  
 }  
 System.*out*.println("Done.");  
 }  
}

Class: LispExpressionEvaluatorTests.java

package assignment;  
import java.util.ArrayList;  
import java.util.EmptyStackException;  
import java.util.Scanner;  
*/\*\*  
 \* This class evaluates a simple arithmetic Lisp expression of numeric values.  
 \*  
 \** ***@author*** *Charles Hoot  
 \** ***@author*** *Jesse Grabowski  
 \** ***@author*** *Joseph Erickson  
 \** ***@author*** *Zeynep Orhan modified  
 \** ***@version*** *5.0  
 \*/*/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class LispExpressionEvaluator {  
 */\*\*  
 \* Evaluates a Lisp expression.  
 \*  
 \* The algorithm: Scan the tokens in the string.  
 \* If you see "(", push the next operator onto the stack.  
 \* If you see an operand, push it onto the stack.  
 \* If you see ")", Pop operands and push them onto a second stack until you find an  
 \* operator. Apply the operator to the operands on the second stack. Push the  
 \* result on the stack.  
 \*  
 \* What may occur? (Samples only)  
 \* If you run out of tokens, the value on the top  
 \* of the stack is the value of the expression.  
 \* OR  
 \* How to detect illegal expressions:  
 \* If you read numeric values and the  
 \* expression stack is empty  
 \* Error message: Bad Expression: needs an operator for the data  
 \*  
 \* If there is a ) and the expression stack is empty  
 \* Error message: mismatched )  
 \*  
 \* If there is a ) and operands needed but the expression stack is empty  
 \* Error message: mismatched )  
 \*  
 \* If the top operator requires at least one operand but it is not in the expression stack  
 \* Error message:operator nameOfTheOperand + " requires at least one operand"  
 \*  
 \* If the string does not have any more characters but the expression stack is not empty  
 \* Error message:incomplete expression / multiple expressions  
 \*  
 \* If the operator is not one of the +, -, \*, ?  
 \* Error message: found an operator when we should not  
 \*  
 \* If the expression is legal  
 \* Message:" and evaluates to " + whateverTheResultIs  
 \*  
 \*  
 \*  
 \*  
 \* Format of sample messages:  
 \* Message for a legal expression  
 \*  
 \* The expression '(+ (- 1) (\* 3 3 4) (/ 3 2 3) (\* 4 4))' is legal in Lisp:  
 \* and evaluates to 51.5  
 \*  
 \* Message for an illegal expression  
 \*  
 \* The expression '(+ (-) (\* 3 3 4) (/ 3 2 3) (\* 4 4))' is not legal in Lisp:  
 \* operator - requires at least one operand  
 \*  
 \** ***@param*** *lispExp A string that is a valid lisp expression.  
 \** ***@param*** *mes An ArrayList of strings that stores the messages generated.  
 \** ***@return*** *A double that is the value of the expression.  
 \*/* @SuppressWarnings("resource")  
 public static double evaluate(String lispExp, ArrayList<String> mes) {  
 StackInterface<LispToken> expressionStack = new OurStack<>();  
 StackInterface<LispToken> secondStack = new OurStack<>();  
 boolean nextIsOperator = false;  
 Scanner lispExpScanner = new Scanner(lispExp);  
 // Use zero or more white spaces as delimiter  
 // that breaks the string into single characters  
 lispExpScanner = lispExpScanner.useDelimiter("\\s\*");  
 boolean legal = true;  
 String errorMessage = "";  
 // Hint: use  
 // lispExpScanner.hasNext() to test if there are more tokens  
 // lispExpScanner.hasNextInt() to test if there is an integer  
 // lispExpScanner.next() to get the next String  
  
 while (lispExpScanner.hasNext() && legal) {  
 // Handles next value in scanner  
  
 if (lispExpScanner.hasNextInt()) {  
 double expression = lispExpScanner.nextDouble();  
 expressionStack.push(new LispToken(expression));  
 System.*out*.println("pushing int" + expression);  
 } else {  
 // Assertion: next value in scanner is a character (i.e., non-numeric value)  
 char next = lispExpScanner.next().charAt(0);  
 System.*out*.println(next + "read");  
 if (next == '(') {  
 // Action: push next operator onto the stack  
  
 if (!lispExpScanner.hasNextInt()) {  
 // Assertion: next value in scanner is an operator  
 char Operator = lispExpScanner.next().charAt(0);  
 expressionStack.push(new LispToken(Operator));  
 System.*out*.println("pushing" + Operator);  
 } else {  
 // Assertion: error; next number is not an operator  
 legal = false;  
 errorMessage = "unknown operator";  
 }  
 } else if (next == ')') {  
 if (expressionStack.isEmpty()) {  
 // Assertion: expression stack is empty, thus no operation can be performed  
 legal = false;  
 errorMessage = "mismatched )";  
 } else {  
 int counter = 0;  
  
 // Action: pop operands until an operator is found  
  
 while (!expressionStack.isEmpty() && !expressionStack.peek().isOperator()) {  
 // Assertion: current top item of stack is an operand  
 LispToken operand = expressionStack.pop();  
 secondStack.push(operand);  
 counter++;  
 }  
  
 if (expressionStack.isEmpty()) {  
 // Since the stack is empty, there is no operator at the end of the list of operands (thus causing a mismatched ')' error)  
 legal = false;  
 errorMessage = "mismatched )";  
 } else {  
 // Action: Performing operation  
  
 LispToken operator = expressionStack.pop();  
 Double result = 0.0;  
  
 if (counter >= 1) {  
 result = secondStack.pop().getValue();  
  
 if (counter == 1) {  
 // *TODO: Need a comment here explaining why identity is used* result = operator.applyOperator(operator.getIdentity(), result);  
 } else {  
 while(!secondStack.isEmpty()) {  
 result = operator.applyOperator(result, secondStack.pop().getValue());  
 }  
 }  
 } else if (counter == 0) {  
 if (operator.takesZeroOperands()) {  
 result = operator.getIdentity();  
 } else {  
 legal = false;  
 errorMessage = "operator " + operator + " requires at least one operand";  
 }  
 }  
  
 // Result of operation pushed to expression stack  
 expressionStack.push(new LispToken(result));  
 }  
 }  
 } else {  
 // Assertion: character is an operator (should only be added after parentheses)  
 legal = false;  
 errorMessage = "found an operator when we should not";  
 }  
 }  
 }  
  
 // Message creation  
  
 String message = "";  
 double value = 0.0;  
  
 // Message handling for legal lisp  
 if (legal) {  
 value = expressionStack.pop().getValue();  
  
 if (!expressionStack.isEmpty()) {  
 // Assertion: the lisp was processed without any errors, but there are multiple items in the expression stack (implying an incomplete expression / multiple expressions error)  
 errorMessage = "incomplete expression / multiple expressions";  
 legal = false;  
 }  
  
 message = "The expression '" + lispExp + "'\nis legal in Lisp:\nand evaluates to " + value + "\n";  
 }  
  
 // Message handling for illegal lisp  
 if (!legal) {  
 message = "The expression '" + lispExp + "'\nis not legal in Lisp:\n" + errorMessage + "\n";  
  
 value = -1.0;  
 }  
  
 mes.add(message);  
 return value;  
 }  
  
 public static void main(String args[]) {  
 String tests[] = {  
 "(+ 1 3)",  
 "(- 1)",  
 "(-)",  
 "(+)",  
 "(\*)",  
 "(/)",  
 "(- 1 2)",  
 "(+ (- 1) (\* 3 3 4) (/ 3 2 3) (\* 4 4))",  
 "(+ (-) (\* 3 3 4) (/ 3 2 3) (\* 4 4))",  
 "(+ (- 1) (\* 3 3 4) ) 5 (\* (/ 3 2 3) (\* 4 4))",  
 "(+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))",  
 "+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))",  
 };  
 ArrayList<String> mes = new ArrayList<>();  
 for (int i = 0; i < tests.length; i++) {  
 *evaluate*(tests[i], mes);  
 System.*out*.println(mes.get(i));  
 }  
 System.*out*.println("Done.");  
 }  
}

class: LispTokenTests.java

package assignment;  
  
import static org.junit.Assert.\*;  
  
import org.junit.After;  
import org.junit.AfterClass;  
import org.junit.Before;  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
import assignment.\*;  
/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class LispTokenTests {  
  
 LispToken plus = new LispToken('+');  
 LispToken minus = new LispToken('-');  
 LispToken mult = new LispToken('\*');  
 LispToken div = new LispToken('/');  
  
 @BeforeClass  
 public static void setUpBeforeClass() throws Exception {  
 }  
  
 @AfterClass  
 public static void tearDownAfterClass() throws Exception {  
 }  
  
 @Before  
 public void setUp() throws Exception {  
 }  
  
 @After  
 public void tearDown() throws Exception {  
 }  
  
 @Test  
 public void plusTests() {  
 *assertTrue*(plus.isOperator());  
 *assertEquals*(plus.getIdentity(), (Double) 0.0);  
 *assertTrue*(plus.takesZeroOperands());  
 }  
  
 @Test  
 public void minusTests() {  
 *assertTrue*(minus.isOperator());  
 *assertEquals*(minus.getIdentity(), (Double) 0.0);  
 *assertFalse*(minus.takesZeroOperands());  
 }  
  
 @Test  
 public void multTests() {  
 *assertTrue*(mult.isOperator());  
 *assertEquals*(mult.getIdentity(), (Double) 1.0);  
 *assertTrue*(mult.takesZeroOperands());  
 }  
  
 @Test  
 public void divTests() {  
 *assertTrue*(div.isOperator());  
 *assertEquals*(div.getIdentity(), (Double) 1.0);  
 *assertFalse*(div.takesZeroOperands());  
 }  
  
 @Test  
 public void plusOperations() {  
 *assertEquals*(plus.applyOperator(1.0, 2.0), (Double) 3.0);  
 *assertEquals*(plus.applyOperator(1.0, -13.0), (Double) (-12.0));  
 }  
  
 @Test  
 public void minusOperations() {  
 *assertEquals*(minus.applyOperator(1.0, 2.0), (Double) (-1.0));  
 *assertEquals*(minus.applyOperator(1.0, -13.0), (Double) (14.0));  
 }  
  
 @Test  
 public void multOperations() {  
 *assertEquals*(mult.applyOperator(3.0, 2.0), (Double) (6.0));  
 *assertEquals*(mult.applyOperator(-3.0, 2.0), (Double) (-6.0));  
 *assertEquals*(mult.applyOperator(-3.0, -2.0), (Double) (6.0));  
 }  
  
 @Test  
 public void divOperations() {  
 *assertEquals*(div.applyOperator(3.0, 2.0), (Double) (1.5));  
 *assertEquals*(div.applyOperator(12.0, 2.0), (Double) (6.0));  
 *assertEquals*(div.applyOperator(13.0, -5.0), (Double) (-2.6));  
 }  
}

Class: OurStackTests.java

package assignment;  
  
import static org.junit.Assert.\*;  
  
import org.junit.After;  
import org.junit.AfterClass;  
import org.junit.Before;  
import org.junit.BeforeClass;  
import org.junit.Test;  
  
import assignment.\*;  
/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* @author Manav Bilakhia (MB)  
 \* @author Saeed AlSuwaidi  
 \* @author Jason D'Amico  
 \*/  
public class OurStackTests {  
  
 LispToken plus = new LispToken('+');  
 LispToken minus = new LispToken('-');  
 LispToken mult = new LispToken('\*');  
 LispToken div = new LispToken('/');  
  
 @BeforeClass  
 public static void setUpBeforeClass() throws Exception {  
 }  
  
 @AfterClass  
 public static void tearDownAfterClass() throws Exception {  
 }  
  
 @Before  
 public void setUp() throws Exception {  
 }  
  
 @After  
 public void tearDown() throws Exception {  
 }  
  
 @Test  
 public void addEntries() {  
 StackInterface<String> testStack = new OurStack<>();  
  
 for (int i = 0; i < 4; i++) {  
 String newEntry = "entry " + i;  
 testStack.push(newEntry);  
 *assertEquals*(testStack.peek(), newEntry);  
 }  
 }  
  
 @Test  
 public void clearEntries() {  
 StackInterface<String> testStack = new OurStack<>();  
  
 for (int i = 0; i < 4; i++) {  
 String newEntry = "entry " + i;  
 testStack.push(newEntry);  
 }  
  
 testStack.clear();  
 *assertTrue*(testStack.isEmpty());  
 }  
  
 @Test  
 public void popEntries() {  
 StackInterface<String> testStack = new OurStack<>();  
  
 String[] entries = {"entry 1", "entry 2", "entry 3", "entry 4"};  
  
 for (int i = 0; i < entries.length; i++) {  
 String newEntry = entries[i];  
 testStack.push(newEntry);  
 }  
  
 while(testStack.isEmpty()) {  
 String popped = testStack.pop();  
  
 boolean isInEntries = false;  
  
 for (int i = 0; i < entries.length; i++) {  
 isInEntries = isInEntries || entries[i].equals(popped);  
 }  
  
 *assertTrue*(isInEntries);  
 }  
 }  
}

1. Sample output 1
2. Describe your test 1: Checking if no operand conditions work as expected for all operators
3. Text output 1:

The expression '(-)'

is not legal in Lisp:

operator - requires at least one operand

The expression '(+)'

is legal in Lisp:

and evaluates to 0.0

The expression '(\*)'

is legal in Lisp:

and evaluates to 1.0

The expression '(/)'

is not legal in Lisp:

operator / requires at least one operand

1. Screenshot 1:

Text

Description automatically generated

1. Sample output 2
2. Describe your test 2: calculation with one operand must work as expected
3. Text output 2:

The expression '(- 1)'

is legal in Lisp:

and evaluates to -1.0

1. Screenshot 2:
2. Shape

   Description automatically generated with medium confidence
3. Sample output 3
4. Describe your test 3: Checking if error handling works properly
5. Text output 3:

The expression '(+ (-) (\* 3 3 4) (/ 3 2 3) (\* 4 4))'

is not legal in Lisp:

operator - requires at least one operand

The expression '(+ (- 1) (\* 3 3 4) ) 5 (\* (/ 3 2 3) (\* 4 4))'

is not legal in Lisp:

incomplete expression / multiple expressions

The expression '(+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))'

is not legal in Lisp:

mismatched )

The expression '+ (- 1) (\* 3 3 4) (/ 3 2 3)) (\* 4 4))'

is not legal in Lisp:

found an operator when we should not

1. Screenshot 3:

Text

Description automatically generated