```java
package proj4;

/**
 *  Runs the ExpressionEvaluator which evaluates a fully parenthesized expression.
 */
public class Client
{
    public static void main(String args[])
    {
        ExpressionEvaluator eval = new ExpressionEvaluator();
        eval.evaluateString("(10*(6-2))");
        //eval.evaluateFile("proj4/expressions.txt");
    }
}
```

```
package proj4;

/**
 * This class implements a divide token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Divide implements Token
{
    /**
     * The constructor for the divide token.
     */
    public Divide()
    {
        // No initialization needed for this token
    }

    /**
     * Compute the sum of two operands.
     * @param op1
     * @param op2
     * @return the sum of the two operands
     */
    public Operand compute(Operand op1, Operand op2)
    {
        int num1 = Integer.parseInt(op1.toString());
        int num2 = Integer.parseInt(op2.toString());
        return new Operand(num1 / num2);
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        Token third = s.pop();
        Token second = s.pop();
        Token first = s.pop();
        Token result = compute((Operand) first, (Operand) third);
        s.push(result);
    }

    /**
     * The toString method returns the string representation of the divide token.
     * @return the string representation of the divide token
     */
    public String toString()
    {
        return "/";
    }
}
```

```
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Divide class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class DivideTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Divide divide;
    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception
    {
        divide = new Divide();
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception
    {
        divide = null;
        stack = null;
    }

    @Test
    public void testCompute()
    {
        Operand op1 = new Operand(10);
        Operand op2 = new Operand(2);
        Operand result = divide.compute(op1, op2);
        assertEquals("Expected division result", 5, Integer.parseInt(result.toString()));
    }

    @Test
    public void testHandle()
    {
        stack.push(new Operand(10));
        stack.push(new Divide());
        stack.push(new Operand(2));
        divide.handle(stack);
        assertEquals("Expected stack size after handling divide", 1, stack.size());
        assertEquals("Expected top of stack to be the result of division", new Operand(5),
stack.peek());
    }

    @Test
    public void testHandleWithEmptyStack()
    {
        divide.handle(stack);
        assertEquals("Expected stack size after handling empty stack", 0, stack.size());
    }
}
```

```
package proj4;

/**
 * This class implements a exponent token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Exponent implements Token
{
    /**
     * The constructor for the exponent token.
     */
    public Exponent()
    {
        // No initialization needed for this token
    }

    /**
     * Compute the sum of two operands.
     * @param op1
     * @param op2
     * @return the sum of the two operands
     */
    public Operand compute(Operand op1, Operand op2)
    {
        int num1 = Integer.parseInt(op1.toString());
        int num2 = Integer.parseInt(op2.toString());
        return new Operand((int) Math.pow(num1, num2));
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        Token third = s.pop();
        Token second = s.pop();
        Token first = s.pop();
        Token result = compute((Operand) first, (Operand) third);
        s.push(result);
    }

    /**
     * The toString method returns the string representation of the exponent token.
     * @return the string representation of the exponent token
     */
    public String toString()
    {
        return "^";
    }
}
```

```
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Exponent class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class ExponentTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Exponent exponent;
    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception
    {
        exponent = new Exponent();
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception
    {
        exponent = null;
        stack = null;
    }

    @Test
    public void testCompute()
    {
        Operand op1 = new Operand(2);
        Operand op2 = new Operand(3);
        Operand result = exponent.compute(op1, op2);
        assertEquals("Expected exponentiation result", 8, Integer.parseInt(result.toString(
)));
    }

    @Test
    public void testHandle()
    {
        stack.push(new Operand(2));
        stack.push(new Exponent());
        stack.push(new Operand(3));
        exponent.handle(stack);
        assertEquals("Expected stack size after handling exponent", 1, stack.size());
        assertEquals("Expected top of stack to be the result of exponentiation", new Operan
d(8), stack.peek());
    }

    @Test
    public void testHandleWithEmptyStack()
    {
        exponent.handle(stack);
        assertEquals("Expected stack size after handling empty stack", 0, stack.size());
    }
```

```java
}
```

```
package proj4;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * Evaluates a fully-parenthesized expression.
 *
 * evaluateString() demonstrates using the Expression iterator. You need to
 * alter the method body for this project.
 *
 * evaluateFile() works completely and should not be altered. It lets you
 * evaluate multiple expressions in a text file by calling evaluateString()
 * on each line of the file.  Try it by calling it from the main() method
 * with the provided sample file.
 */
public class ExpressionEvaluator
{
    /**
     * Given a fully-parenthesized arithmetic expression, evaluates it
     * and returns the answer
     *
     * @param expression infix expression as a string like "(1-(2+3))"
     * @return answer string, like "-4" for the above example
     */
    public String evaluateString(String expression)
    {
        Expression exp = new Expression(expression);
        Stack<Token> stack = new Stack<Token>();

        // example of using a for-each loop to get each meaningful substring
        for (String substring: exp)
        {
            System.out.println(substring);
            Token token = null;
            if (substring.equals("+"))
            {
                token = new Plus();
                stack.push(token);
            }

            else if (substring.equals("-"))
            {
                token = new Minus();
                stack.push(token);
            }

            else if (substring.equals("*"))
            {
                token = new Multiply();
                stack.push(token);
            }

            else if (substring.equals("/"))
            {
                token = new Divide();
                stack.push(token);
            }

            else if (substring.equals("^"))
            {
                token = new Exponent();
                stack.push(token);
            }

            else if (substring.equals("("))
            {
                // do nothing
            }

            else if (substring.equals(")"))
            {
                token = new RightParen();
                token.handle(stack);
            }

            else
            { // must be an operand
                token = new Operand(substring);
                token.handle(stack);
            }
        }
        // at this point, the stack should have only one item left
        if (stack.size() != 1)
        {
            System.out.println("Error: stack size is " + stack.size());
            System.out.println("Expected stack size is 1");
            System.out.println("Expression: " + expression);
            return "";
        }
        return stack.pop().toString();
    }


    /**
     * Given a filename with one arithmetic expression per line, evaluate each
     * expression and print it.
     *
     * @param filename name of file as a string.  Include the extension
     *                 (like "input.txt") and if you are not supplying a path, then
     *                 this file is assumed to be inside your src folder in IntelliJ,
     *                 but not inside the proj4 package.
     */
    public void evaluateFile(String filename)
    {
        try (Scanner myscan = new Scanner(new File(filename)))
        {
            while (myscan.hasNextLine())
            {
                String line = myscan.nextLine();
                System.out.print(line + " = ");
                System.out.println(evaluateString(line));
            }
        }
        catch (FileNotFoundException e)
        {
            System.out.println("File not found. Make sure it's in src folder but not proj4
folder");
        }
    }



}
```

```java
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the ExpressionEvaluator class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class ExpressionEvaluatorTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private ExpressionEvaluator evaluator;

    @Before
    public void setUp() throws Exception
    {
        evaluator = new ExpressionEvaluator();
    }

    @After
    public void tearDown() throws Exception
    {
        evaluator = null;
    }

    @Test
    public void testEvaluate()
    {
        String expression = "((3+4)*2)";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result", 14, result);
    }

    @Test
    public void testEvaluateWithNegativeNumbers()
    {
        String expression = "((-3+4)*2)";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result with negative numbers", 2, result);
    }

    @Test
    public void testEvaluateWithNestedExpressions()
    {
        String expression = "((3+(4*2))-1)";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result with nested expressions", 10, result);
    }

    @Test
    public void testEvaluateWithDivision()
    {
        String expression = "((8/4)+2)";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result with division", 4, result);
    }
    @Test
    public void testEvaluateWithExponentiation()
    {
        String expression = "(2^3)";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result with exponentiation", 8, result);
    }

    @Test
    public void testEvaluateWithInvalidExpression()
    {
        String expression = "(3+4*2";
        String result = evaluator.evaluateString(expression);
        assertEquals("Expected evaluation result with invalid expression", "", result);
    }

    @Test
    public void testEvaluateWithEmptyExpression()
    {
        String expression = "";
        String result = evaluator.evaluateString(expression);
        assertEquals("Expected evaluation result with empty expression", "", result);
    }

    @Test
    public void testEvaluateWithSingleOperand()
    {
        String expression = "5";
        int result = Integer.parseInt(evaluator.evaluateString(expression));
        assertEquals("Expected evaluation result with single operand", 5, result);
    }
}
```

```
/** DO NOT ALTER THIS CLASS **/
package proj4;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Iterator for the concept of an Expression, which
 * is a string representation of a fully-parenthesized
 * arithmetic expression.
 */
public class Expression implements Iterable<String>{

    private String exp;  // expression to parse
    private int start;   // index at which next meaningful substring will come from

    /**
     * non-default constructor
     *
     * @param s fully-parenthesized infix expression
     */
    public Expression(String s){
        exp = s;
        start = 0;
    }

    /**
     * creates an Iterator to be used by Expression
     * (requirement of Iterable interface)
     *
     * @return Iterator that must have hasNext() and next()
     */
    @Override
    public Iterator<String> iterator() {
        return new Parser();
    }


    /**
     * Inner class used to create the Iterator for an Expression
     */
    private class Parser implements Iterator<String> {

        /**
         * Says if there's another lexeme (substring) left to parse
         * which can be turned into a Token.  We know there
         * is if start is still a valid index and the substring
         * starting at index start isn't all whitespace
         *
         * @return true if there's a tokenable substring still left to read, else false
         */
        @Override
        public boolean hasNext() {
            return start < exp.length() &&
                    !(exp.substring(start).trim().isEmpty());
        }

        /**
         * Return the next substring (lexeme) that can be turned
         * into a token.  Possible substrings include non-negative operands
         * (single or multi-digit), parentheses, or operators
         *
         * @return "(", ")", "+", "-", "*", "/", "^", or a non-negative operand like "125"
         */
        @Override
        public String next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            else {
                // get past any whitespace
                char ch = exp.charAt(start);
                while (Character.isWhitespace(ch)) {
                    start++;
                    ch = exp.charAt(start);
                }

                // ch is now a real tokenable substring: either an operator, paren, or star
t of a number
                String tokenable = "";
                if (Character.isDigit(ch)) {                    // if we've got a digit...
                    do {
                        tokenable = tokenable + String.valueOf(ch);  //...gather any subseq
uent digits
                        start++;
                        ch = exp.charAt(start);
                    }
                    while (Character.isDigit(ch));
                }
                else {
                    start++;
                    tokenable = String.valueOf(ch);
                }
                return tokenable;
            }
        }
    }
}
```

```
package proj4;

/**
 * This class implements a minus token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Minus implements Token
{
    /**
     * The constructor for the minus token.
     */
    public Minus() {
        // No initialization needed for this token
    }

    /**
     * Compute the sum of two operands.
     * @param op1
     * @param op2
     * @return the sum of the two operands
     */
    public Operand compute(Operand op1, Operand op2)
    {
        int num1 = Integer.parseInt(op1.toString());
        int num2 = Integer.parseInt(op2.toString());
        return new Operand(num1 - num2);
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        Token third = s.pop();
        Token second = s.pop();
        Token first = s.pop();
        Token result = compute((Operand) first, (Operand) third);
        s.push(result);
    }

    /**
     * The toString method returns the string representation of the minus token.
     * @return the string representation of the minus token
     */
    public String toString()
    {
        return "-";
    }
}
```

```
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Minus class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class MinusTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Minus minus;
    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception
    {
        minus = new Minus();
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception
    {
        minus = null;
        stack = null;
    }

    @Test
    public void testCompute()
    {
        Operand op1 = new Operand(5);
        Operand op2 = new Operand(3);
        Operand result = minus.compute(op1, op2);
        assertEquals("Expected subtraction result", 2, Integer.parseInt(result.toString()))
;
    }

    @Test
    public void testHandle()
    {
        stack.push(new Operand(5));
        stack.push(new Minus());
        stack.push(new Operand(3));
        minus.handle(stack);
        assertEquals("Expected stack size after handling minus", 1, stack.size());
        assertEquals("Expected top of stack to be the result of subtraction", new Operand(2
), stack.peek());
    }

    @Test
    public void testHandleWithEmptyStack()
    {
        minus.handle(stack);
        assertEquals("Expected stack size after handling empty stack", 0, stack.size());
    }
    @Test
    public void testHandleWithInsufficientTokens()
    {
        stack.push(new Operand(5));
        minus.handle(stack);
        assertEquals("Expected stack size after handling insufficient tokens", 1, stack.siz
e());
    }

}
```

```
package proj4;

/**
 * This class implements a multiply token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Multiply implements Token
{
    /**
     * The constructor for the multiply token.
     */
    public Multiply() {
        // No initialization needed for this token
    }

    /**
     * Compute the sum of two operands.
     * @param op1
     * @param op2
     * @return the sum of the two operands
     */
    public Operand compute(Operand op1, Operand op2)
    {
        int num1 = Integer.parseInt(op1.toString());
        int num2 = Integer.parseInt(op2.toString());
        return new Operand(num1 * num2);
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        Token third = s.pop();
        Token second = s.pop();
        Token first = s.pop();
        Token result = compute((Operand) first, (Operand) third);
        s.push(result);
    }

    /**
     * The toString method returns the string representation of the multiply token.
     * @return the string representation of the multiply token
     */
    public String toString()
    {
        return "*";
    }
}
```

```java
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Multiply class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class MultiplyTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);
    private Multiply multiply;
    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception
    {
        multiply = new Multiply();
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception
    {
        multiply = null;
        stack = null;
    }

    @Test
    public void testCompute()
    {
        Operand op1 = new Operand(5);
        Operand op2 = new Operand(3);
        Operand result = multiply.compute(op1, op2);
        assertEquals("Expected multiplication result", 15, Integer.parseInt(result.toString
()));
    }

    @Test
    public void testHandle()
    {
        stack.push(new Operand(5));
        stack.push(new Multiply());
        stack.push(new Operand(3));
        multiply.handle(stack);
        assertEquals("Expected stack size after handling multiply", 1, stack.size());
        assertEquals("Expected top of stack to be the result of multiplication", new Operan
d(15), stack.peek());
    }

    @Test
    public void testHandleWithEmptyStack()
    {
        multiply.handle(stack);
        assertEquals("Expected stack size after handling empty stack", 0, stack.size());
    }

    @Test
    public void testHandleWithInsufficientTokens()
    {
        stack.push(new Operand(5));
        multiply.handle(stack);
        assertEquals("Expected stack size after handling insufficient tokens", 1, stack.siz
e());
    }
}
```

```
package proj4;

/**
 * This class implements an operand token.
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Operand implements Token
{
    private String value;
    /**
     * non-default constructor
     *
     * @param value number that this Operand represents
     */
    public Operand(String value)
    {
        this.value = value;
    }

    /**
     * non-default constructor
     *
     * @param value number that this Operand represents
     */
    public Operand(int value)
    {
        this.value = Integer.toString(value);
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        s.push(this);
    }

    /**
     * The toString method returns the string representation of the operand.
     * @return the string representation of the operand
     */
    public String toString()
    {
        return value;
    }
}
```

```
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Operand class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class OperandTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Operand operand;
    private Stack<Token> stack;

    @Before
    public void setUp() throws Exception
    {
        operand = new Operand(5);
        stack = new Stack<Token>();
    }

    @After
    public void tearDown() throws Exception
    {
        operand = null;
        stack = null;
    }

    @Test
    public void testHandle()
    {
        operand.handle(stack);
        assertEquals("Expected stack size after handling operand", 1, stack.size());
        assertEquals("Expected top of stack to be the operand", operand, stack.peek());
    }

    @Test
    public void testToString()
    {
        assertEquals("Expected string representation of operand", "5", operand.toString());
    }

    @Test
    public void testNonDefaultConstructorWithInt()
    {
        Operand operand3 = new Operand(15);
        assertEquals("Expected string representation of operand", "15", operand3.toString()
);
    }

    @Test
    public void testNonDefaultConstructorWithString()
    {
        Operand operand4 = new Operand("20");
        assertEquals("Expected string representation of operand", "20", operand4.toString()
);
    }
}
```

```
package proj4;

/**
 * This class implements a plus token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Plus implements Token
{
    /**
     * The constructor for the plus token.
     */
    public Plus()
    {
        // No initialization needed for this token
    }

    /**
     * Compute the sum of two operands.
     * @param op1
     * @param op2
     * @return the sum of the two operands
     */
    public Operand compute(Operand op1, Operand op2)
    {
        int num1 = Integer.parseInt(op1.toString());
        int num2 = Integer.parseInt(op2.toString());
        return new Operand(num1 + num2);
    }

    /**
     * The handle method. Just add it to the stack.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        Token third = s.pop();
        Token second = s.pop();
        Token first = s.pop();
        Token result = compute((Operand) first, (Operand) third);
        s.push(result);
    }

    /**
     * The toString method returns the string representation of the plus token.
     * @return the string representation of the plus token
     */
    public String toString()
    {
        return "+";
    }
}
```

```java
package proj4;
/**
 * JUnit test class.  Use these tests as models for your own.
 */

import java.util.concurrent.TimeUnit;

import junit5.documentation.src.test.java.example.FastTest;

import java.beans.Transient;

public class PlusTest
{
    @Rule // a test will fail if it takes longer than 1/10 of a second to run
        public Timeout timeout = Timeout.millis(100);

    //Test compute method
    // The compute method should return the sum of two operands
    @Test
    public void testCompute()
    {
        Plus plus = new Plus();
        Operand op1 = new Operand(5);
        Operand op2 = new Operand(10);
        Operand result = plus.compute(op1, op2);
        assertEquals(15, Integer.parseInt(result.toString()));
    }

    //Test handle method
    // The handle method should pop three tokens from the stack, compute the sum of the fir
st and third tokens,
    // and push the result back onto the stack
    @Test
    public void testHandle()
    {
        Stack<Token> stack = new Stack<Token>();
        Operand op1 = new Operand(5);
        Operand op2 = new Operand(10);
        Operand op3 = new Operand(15);
        stack.push(op1);
        stack.push(op2);
        stack.push(op3);

        Plus plus = new Plus();
        plus.handle(stack);

        assertEquals(25, Integer.parseInt(stack.pop().toString()));
        Operand result = (Operand) stack.pop();
        assertEquals(25, Integer.parseInt(result.toString()));
        assertEquals(0, stack.size());
    }
}
```

```
package proj4;

/**
 * This class implements a right parenthesis token.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class RightParen implements Token
{
    /**
     * The constructor for the right parenthesis token.
     */
    public RightParen()
    {
        // No initialization needed for this token
    }

    /**
     * Processes the current token.
     * @param s the Stack the token uses when processing itself.
     */
    public void handle(Stack<Token> s)
    {
        // Pop the last 3 tokens from the stack. Check for size first
        if (s.isEmpty() || s.size() < 3)
        {
            System.out.println("Error: Not enough tokens to process.");
            System.out.println("Stack: " + s);
            return;
        }
        Token first = s.pop();
        Token second = s.pop();
        Token third = s.pop();

        // Check if the popped tokens are valid
        boolean firstIsOperand = first instanceof Operand;
        boolean secondIsOperator = !(second instanceof Operand);
        boolean thirdIsOperand = third instanceof Operand;


        // Push the tokens back to the stack
        s.push(third);
        s.push(second);
        s.push(first);

        if (firstIsOperand && secondIsOperator && thirdIsOperand)
        {
            second.handle(s);
            System.out.println("Processed: " + third + " " + second + " " + first);
            System.out.println("Stack: " + s);
        }

        else
        {
            System.out.println("Error: Invalid tokens for processing.");
            System.out.println("Stack: " + s);
            return;
        }
    }

    /**
     * Returns the token as a printable String
     *
     * @return the String version of the token.  For example, ")"
     * for a right parenthesis.
     */
    public String toString()
    {
        return ")";
    }
}
```

```java
package proj4;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the RightParen class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 */

public class RightParenTest
{
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Stack<Token> stack;
    private RightParen rightParen;

    @Before
    public void setUp() throws Exception
    {
        stack = new Stack<Token>();
        rightParen = new RightParen();
    }

    @After
    public void tearDown() throws Exception
    {
        stack = null;
        rightParen = null;
    }

    @Test
    public void testHandleWithValidTokens()
    {
        stack.push(new Operand(5));
        stack.push(new Exponent());
        stack.push(new Operand(2));
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling valid tokens", 1, stack.size());
    }

    @Test
    public void testHandleWithInvalidTokens()
    {
        stack.push(new Operand(5));
        stack.push(new Operand(2));
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling invalid tokens", 3, stack.size());
    }

    @Test
    public void testHandleWithEmptyStack()
    {
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling empty stack", 0, stack.size());
    }

    @Test
    public void testHandleWithInsufficientTokens()
    {
        stack.push(new Operand(5));
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling insufficient tokens", 1, stack.siz
e());
    }

    @Test
    public void testHandleWithNonOperatorToken()
    {
        stack.push(new Operand(5));
        stack.push(new Operand(2));
        stack.push(new Operand(3));
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling non-operator token", 3, stack.size
());
    }

    @Test
    public void testHandleWithInvalidTokenOrder()
    {
        stack.push(new Operand(5));
        stack.push(new Operand(2));
        stack.push(new Exponent());
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling invalid token order", 3, stack.siz
e());
    }

    @Test
    public void testHandleWithValidTokenOrder()
    {
        stack.push(new Operand(5));
        stack.push(new Exponent());
        stack.push(new Operand(2));
        rightParen.handle(stack);
        assertEquals("Expected stack size after handling valid token order", 1, stack.size(
));
    }
}
```

```
package proj4;

import java.util.Deque;

/**
 * This class implements a stack of generic objects.
 * The stack is a last-in-first-out (LIFO) data structure.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 * I confirm that I have read the course academic integrity policy and that I am not in vio
lation.
 */
public class Stack<T>
{
    /**
     * The stack is implemented using a Deque.
     */
    private Deque<T> stack;
    /**
     * Constructor for objects of class Stack
     * @return a new stack
     */
    public Stack()
    {
       stack = new java.util.ArrayDeque<T>();
    }

    /**
     * Returns true if the stack is empty, false otherwise.
     * @return true if the stack is empty, false otherwise.
     */
    public boolean isEmpty()
    {
        return stack.isEmpty();
    }

    /**
     * Pushes a value onto the stack.
     * @param value
     */
    public void push(T value)
    {
        stack.push(value);
    }

    /**
     * Pops a value off the stack.
     * @return the value popped off the stack
     */
    public T pop()
    {
        return stack.pop();
    }

    /**
     * Peeks at the top value of the stack.
     * Looks at the value at the top of the stack without removing it.
     * @return the value at the top of the stack
     */
    public T peek()
    {
        return stack.peek();
    }

    /**
     * Returns the number of elements in the stack.
     * @return the number of elements in the stack
     */
    public int size()
    {
        return stack.size();
    }

    /**
     * return stack as a printable string
     * @return string in the form {>A, B, C} where > denotes the stack top
     */
    public String toString()
    {
        String ans = "{>";
        for (int i = stack.size() - 1; i >= 0; i--) {
            if (i == stack.size() - 1)
            {
                ans += stack.getFirst();
            }

            else
            {
                ans += ", " + stack.getFirst();
            }
            stack.addLast(stack.removeFirst());
        }
        ans += "}";
        return ans;
    }

}
```

```
package proj4;
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

/**
 *
 * This class tests the Stack class.
 *
 * @author Kaito Nguyen
 * @version May 21st 2025, Spring 2025
 *
 */
public class StackTest
{

        @Rule
    public Timeout timeout = Timeout.millis(100);

    private Stack<String> stack;

    @Before
    public void setUp() throws Exception
    {
        stack = new Stack<String>();
    }

    @After
    public void tearDown() throws Exception
    {
        stack = null;
    }

    @Test
    public void testStackConstructor_toString ()
    {
        assertEquals ("An empty stack. (> indicates the top of the stack)", "{>}", stack.to
String());
    }

    @Test
    public void testStackPushOneOntoEmptyStack ()
    {
        stack.push("A");
        assertEquals ("Pushing A onto an empty stack.", "{>A}", stack.toString().replaceAll
("[ ]+", ""));
    }

    @Test
    public void testStackPushTwoOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        assertEquals ("Pushing first A and then B onto an empty stack.", "{>B,A}", stack.to
String().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPushThreeOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        stack.push("C");
        assertEquals ("Pushing first A, then B, then C onto an empty stack.", "{>C,B,A}", s
tack.toString().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPopOneOntoEmptyStack ()
    {
        stack.push("A");
        String popped = stack.pop();
        assertEquals ("Popping A from a stack with one element.", "A", popped);
        assertEquals ("Popping A from a stack with one element.", "{>}", stack.toString().r
eplaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPopTwoOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        String popped = stack.pop();
        assertEquals ("Popping B from a stack with two elements.", "B", popped);
        assertEquals ("Popping B from a stack with two elements.", "{>A}", stack.toString()
.replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPopThreeOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        stack.push("C");
        String popped = stack.pop();
        assertEquals ("Popping C from a stack with three elements.", "C", popped);
        assertEquals ("Popping C from a stack with three elements.", "{>B,A}", stack.toStri
ng().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPopEmptyStack ()
    {
        String popped = stack.pop();
        assertEquals ("Popping from an empty stack.", null, popped);
        assertEquals ("Popping from an empty stack.", "{>}", stack.toString().replaceAll("[
 ]+", ""));
    }

    @Test
    public void testStackPeekOneOntoEmptyStack ()
    {
        stack.push("A");
        String peeked = stack.peek();
        assertEquals ("Peeking A from a stack with one element.", "A", peeked);
        assertEquals ("Peeking A from a stack with one element.", "{>A}", stack.toString().
replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPeekTwoOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        String peeked = stack.peek();
        assertEquals ("Peeking B from a stack with two elements.", "B", peeked);
        assertEquals ("Peeking B from a stack with two elements.", "{>B,A}", stack.toString
().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPeekThreeOntoEmptyStack ()
    {
        stack.push("A");
        stack.push("B");
        stack.push("C");
        String peeked = stack.peek();
        assertEquals ("Peeking C from a stack with three elements.", "C", peeked);
        assertEquals ("Peeking C from a stack with three elements.", "{>C,B,A}", stack.toSt
ring().replaceAll("[ ]+", ""));
    }

    @Test
    public void testStackPeekEmptyStack ()
    {
        String peeked = stack.peek();
        assertEquals ("Peeking from an empty stack.", null, peeked);
        assertEquals ("Peeking from an empty stack.", "{>}", stack.toString().replaceAll("[
 ]+", ""));
    }

    @Test
    public void testStackIsEmpty ()
    {
        assertTrue ("Checking if an empty stack is empty.", stack.isEmpty());
        stack.push("A");
        assertFalse ("Checking if a non-empty stack is empty.", stack.isEmpty());
        stack.pop();
        assertTrue ("Checking if an empty stack is empty.", stack.isEmpty());
    }

    @Test
    public void testStackSize ()
    {
        assertEquals ("Checking the size of an empty stack.", 0, stack.size());
        stack.push("A");
        assertEquals ("Checking the size of a stack with one element.", 1, stack.size());
        stack.push("B");
        assertEquals ("Checking the size of a stack with two elements.", 2, stack.size());
        stack.push("C");
        assertEquals ("Checking the size of a stack with three elements.", 3, stack.size())
;
        stack.pop();
        assertEquals ("Checking the size of a stack with two elements.", 2, stack.size());
    }

    @Test
    public void testStackToString ()
    {
        assertEquals ("Checking the string representation of an empty stack.", "{>}", stack
.toString().replaceAll("[ ]+", ""));
        stack.push("A");
        assertEquals ("Checking the string representation of a stack with one element.", "{
>A}", stack.toString().replaceAll("[ ]+", ""));
        stack.push("B");
        assertEquals ("Checking the string representation of a stack with two elements.", "
{>B,A}", stack.toString().replaceAll("[ ]+", ""));
        stack.push("C");
        assertEquals ("Checking the string representation of a stack with three elements.",
 "{>C,B,A}", stack.toString().replaceAll("[ ]+", ""));
    }
}
```

```java
package proj4;

/**
 * Describes the methods that must be defined in order for an
 * object to be considered a token.  Every token must be able
 * to be processed (handle) and printable (toString).
 *
 * @author Chris Fernandes
 * @version 5/18/25
 */
public interface Token
{
        /** Processes the current token.  Since every token will handle
         *  itself in its own way, handling may involve pushing or
         *  popping from the given stack.
         *
         *  @param s the Stack the token uses when processing itself.
         */
    public void handle(Stack<Token> s);



    /** Returns the token as a printable String
     *
     *  @return the String version of the token.  For example, ")"
     *  for a right parenthesis.
     */
    public String toString();
}
```