```java
package proj4;
/**
 * Client class that converts test proj4 input file from infix to postfix
 *
 * @author Neil Daterao
 * @version 2/24/2024
 * @Note I affirm that I have carried out the attached academic endeavors with full academic honesty, in
 * accordance with the Union College Honor Code and the course syllabus.
 */

public class Client
{
    /**
     * Main method
     * @param args
     */
    public static void main(String[] args)
    {
        Converter converter = new Converter("src/resources/converter_test.txt");
        converter.convert();

    }


}

```

```java
package proj4;

/**
 * Converter class which contains a void method convert, that converts an infix expression to a postfix expression.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Converter {
    private FileReader reader;
    private Stack<Token> stack;
    private final String BUFFER = " --> ";

    /**
     * Non-default constructor; Gradescope needs this to run tests
     *
     * @param infile path to the input file
     */
    public Converter(String infile) {
        stack = new Stack<Token>();
        reader = new FileReader(infile);
    }

    /**
     * Converts infix expression to postfix expression, prints to System.out
     */
    public void convert() {
        StringBuilder infixExpression = new StringBuilder();
        StringBuilder postfixExpression = new StringBuilder();

        String token;
        while (!(token = reader.nextToken()).equals("EOF")) {
            Token tokenObj = createToken(token);
            if (tokenObj != null){
```

```
                if (!(tokenObj instanceof Semicolon)) { infixExpression.append(token); }
                postfixExpression.append(tokenObj.handle(stack));

                if (tokenObj instanceof Semicolon) {
                    printInfixAndPostfix(infixExpression.toString(), postfixExpression.toString());
                    postfixExpression.setLength(0); //reset the infix and postfix stringbuilders to handle a new line
                    infixExpression.setLength(0);
                }
            }
            else {
                infixExpression.append(token);
                postfixExpression.append(token);
            }
        }
    }

    /**
     * Creates a Token object based on the input string
     *
     * @param tokenStr input string representing a token
     * @return Token object created from the input string
     * returns null if an Operand
     */
    private Token createToken(String tokenStr) {
        switch (tokenStr) {
            case "+":
                return new Plus();
            case "-":
                return new Minus();
            case "*":
                return new Multiply();
```

```
            case "/":
                return new Divide();
            case "^":
                return new Exponent();
            case "(":
                return new LeftParen();
            case ")":
                return new RightParen();
            case ";":
                return new Semicolon();
            default:
                return null;
        }
    }

    /**
     * Prints both the infix and postfix string with a buffer, " --> "
     * @param infix Infix String
     * @param postfix Postfix String
     */
    private void printInfixAndPostfix(String infix, String postfix) {
        String toPrint = infix + BUFFER + postfix;
        System.out.println(toPrint);
    }
}
```

```java
package proj4;

/**
 * Class for Divide token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Divide implements Token {


    public String toString() {
        return "/";
    }

    public String handle(Stack<Token> s)
    {
        StringBuilder output = new StringBuilder();
        //check all 3 contingencies for rule 3 of algorithm
        while (!s.isEmpty() && !(s.peek() instanceof LeftParen) && getTokenPrecedence(this) <= getTokenPrecedence(s.peek())) {
            output.append(s.pop().toString());
        }
        s.push(this);
        return output.toString();
    }

    /**
     * Returns precedence of a token
     * @param token
     * @return 3 if exponent, 2 if * or /, 1 if + or -
     */
    private int getTokenPrecedence(Token token) {
        if (token instanceof Exponent) {
            return EXPONENTPRECEDENCE;
        } else if (token instanceof Multiply || token instanceof Divide) {
```

```
            return MULTIPLYDIVIDEPRECEDENCE;
        } else if (token instanceof Plus || token instanceof Minus) {
            return PLUSMINUSPRECEDENCE;
        } else {
            return 0; // Default precedence for unknown tokens
        }
    }

}

```

```java
package proj4;

/**
 * Class for Exponent token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Exponent implements Token {


    public String toString() {
        return "^";
    }

    public String handle(Stack<Token> s)
    {
        StringBuilder output = new StringBuilder();
        //check all 3 contingencies for rule 3 of alg
        while (!s.isEmpty() && !(s.peek() instanceof LeftParen) && getTokenPrecedence(this) <= getTokenPrecedence(s.peek())) {
            output.append(s.pop().toString());
        }
        s.push(this);
        return output.toString();
    }

    /**
     * Returns precedence of a token
     * @param token
     * @return 3 if exponent, 2 if * or /, 1 if + or -
     */
    private int getTokenPrecedence(Token token) {
        if (token instanceof Exponent) {
            return EXPONENTPRECEDENCE;
        } else if (token instanceof Multiply || token instanceof Divide) {
```

```
            return MULTIPLYDIVIDEPRECEDENCE;
        } else if (token instanceof Plus || token instanceof Minus) {
            return PLUSMINUSPRECEDENCE;
        } else {
            return 0; // Default precedence for unknown tokens
        }
    }

}

```

```java
package proj4;
import java.io.*;
/**
 * A FileReader object will read tokens from an input file.  The name of
 * the input file is given when the constructor is run. The lone method,
 * nextToken(), will return the next token in the file as a String.
 *
 * @author Chris Fernandes
 * @version 2/20/17
 */
public class FileReader
{
    private StreamTokenizer st;     //file descriptor

    /** non-default constructor.
     *
     *  @param fileName path to input file. Can either be a full
     *  path like "C:/project4/proj4_input.txt" or a relative one like
     *  "src/proj4_input.txt" where file searching begins inside
     *  the project folder.
     *  Be sure to use forward slashes to specify folders.  And
     *  don't forget the quotes (it's a String!)
     */
    public FileReader(String fileName)
    {
        try {
        st = new StreamTokenizer(
                new BufferedReader(
                    new InputStreamReader(
                        new FileInputStream(fileName))));
        } catch (IOException e) {}
        st.resetSyntax();
```

```
// remove default rules
        st.ordinaryChars(0, Character.MAX_VALUE); // turn on all chars
        st.whitespaceChars(0, 39);                    // ignore unprintables &
        st.whitespaceChars(95, Character.MAX_VALUE); // everything after ^
    }

    /** Returns the next valid token from the input file.
     *  Possible tokens are "+", "-", "*", "/", "^", "(", ")",
     *  ";", operands (capital letters), and the special
     *  token "EOF" which is returned when the
     *  end of the input file is reached.
     *
     *  @return next valid token or "EOF"
     */
    public String nextToken()
    {
        try
        {
        while (st.nextToken() != StreamTokenizer.TT_EOF) {
            if (st.ttype < 0)
            {
                return st.sval;
            }
            else
            {
                return String.valueOf((char)st.ttype);
            }
        }
        return "EOF";
        } catch (IOException e) {}
        return "error on token read";
    }
}
```

```java
package proj4;

/**
 * Class for Left Parenthesis Token
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class LeftParen implements Token {

    public String toString() { return "("; }

    public String handle(Stack<Token> s) {
        s.push(this);
        return "";
    }

}

```

```java
package proj4;

/**
 * Class for Minus token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Minus implements Token {

    public String toString() {
        return "-";
    }

    public String handle(Stack<Token> s)
    {
        StringBuilder output = new StringBuilder();
        //check all 3 contingencies for rule 3 of algorithm
        while (!s.isEmpty() && !(s.peek() instanceof LeftParen) && getTokenPrecedence(this) <= getTokenPrecedence(s.peek())) {
            output.append(s.pop().toString());
        }
        s.push(this);
        return output.toString();
    }

     /**
     * Returns precedence of a token
     * @param token
     * @return 3 if exponent, 2 if * or /, 1 if + or -
     */
    private int getTokenPrecedence(Token token) {
        if (token instanceof Exponent) {
            return EXPONENTPRECEDENCE;
        } else if (token instanceof Multiply || token instanceof Divide) {
            return MULTIPLYDIVIDEPRECEDENCE;
```

```
        } else if (token instanceof Plus || token instanceof Minus) {
            return PLUSMINUSPRECEDENCE;
        } else {
            return 0; // Default precedence for unknown tokens
        }
    }

}

```

```java
package proj4;

/**
 * Class for Multiply token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Multiply implements Token {

    public String toString() {
        return "*";
    }

    public String handle(Stack<Token> s)
    {
        StringBuilder output = new StringBuilder();
        //check all 3 contingencies for rule 3 of algorithm
        while (!s.isEmpty() && !(s.peek() instanceof LeftParen) && getTokenPrecedence(this) <= getTokenPrecedence(s.peek())) {
            output.append(s.pop().toString());
        }
        s.push(this);
        return output.toString();
    }

     /**
     * Returns precedence of a token
     * @param token
     * @return 3 if exponent, 2 if * or /, 1 if + or -
     */
    private int getTokenPrecedence(Token token) {
        if (token instanceof Exponent) {
            return EXPONENTPRECEDENCE;
        } else if (token instanceof Multiply || token instanceof Divide) {
            return MULTIPLYDIVIDEPRECEDENCE;
```

```
        } else if (token instanceof Plus || token instanceof Minus) {
            return PLUSMINUSPRECEDENCE;
        } else {
            return 0; // Default precedence for unknown tokens
        }
    }

}

```

```java
package proj4;

/**
 * Class for plus token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Plus implements Token {

    public String toString() {
        return "+";
    }

    public String handle(Stack<Token> s)
    {
        StringBuilder output = new StringBuilder();
        //check all 3 contingencies for rule 3 of algorithm
        while (!s.isEmpty() && !(s.peek() instanceof LeftParen) && getTokenPrecedence(this) <= getTokenPrecedence(s.peek())) {
            output.append(s.pop().toString());
        }
        s.push(this);
        return output.toString();
    }

     /**
     * Returns precedence of a token
     * @param token
     * @return 3 if exponent, 2 if * or /, 1 if + or -
     */
    private int getTokenPrecedence(Token token) {
        if (token instanceof Exponent) {
            return EXPONENTPRECEDENCE;
        } else if (token instanceof Multiply || token instanceof Divide) {
            return MULTIPLYDIVIDEPRECEDENCE;
```

```
        } else if (token instanceof Plus || token instanceof Minus) {
            return PLUSMINUSPRECEDENCE;
        } else {
            return 0; // Default precedence for unknown tokens
        }
    }

}

```

```java
package proj4;

/**
 * Class for Right Parenthesis Token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class RightParen implements Token {

    public String toString() { return ")"; }

    public String handle(Stack<Token> s) {
        StringBuilder output = new StringBuilder();
        while (!(s.peek() instanceof LeftParen)) {
            output.append(s.pop().toString());
        }
        s.pop(); // Discard the left parenthesis
        return output.toString();
    }

}
```

```java
package proj4;

/**
 * Class for Right Parenthesis Token.
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Semicolon implements Token {

    public String toString() { return ";"; }

    public String handle(Stack<Token> s) {
        StringBuilder output = new StringBuilder();
        while (!s.isEmpty()) { output.append(s.pop().toString()); }
        return output.toString();
    }

}
```

```java
package proj4;

/**
 * Generic Stack Class. Able to create a stack of any data type.
 * Invariant:
 * 1. Number of elements in the Stack is stored in the instance variable count
 * 2. For an empty sequence, we don't care what is stored in data. For a nonempty sequence, the elements of the sequence
 *    are stored from the front to the end in contents[0] to contents[count-1], we don't care what's stored in the rest of
 *    data
 * 3. Capacity of the stack is incremented using predetermined capacityIncrement
 * 4. Popping from the stack decreases count by 1
 * 5. Pushing to the stack increases count by 1
 *
 * @author Neil Daterao
 * @version 2/24/2024
 */
public class Stack<T>
{

    private int count;
    private int capacity;
    private int capacityIncrement;
    private T[] contents;

    private final int DEFAULTCAPACITY = 10;
    private final int DEFAULTCAPACITYINCREMENT = 5;


    /**
     * Default constructor for Stack, initializes an empty stack.
     */
    public Stack() {
```

```
        count = 0;
        capacity = DEFAULTCAPACITY;
        capacityIncrement = DEFAULTCAPACITYINCREMENT;
        contents = (T[]) new Object[capacity];

    }

    /**
     *
     * @return True if stack is empty, false if not
     */
    public boolean isEmpty() {
        return (size() == 0);  //erase this line
    }

    /**
     * Pushes an element onto the top of this stack.
     *
     * @param toPush the element to be pushed onto this stack
     * @throws NullPointerException if the specified element is null and this stack does not permit null elements
     */
    public void push(T toPush) {
        if (toPush == null) {
            throw new NullPointerException("Cannot push null elements onto the stack");
        }
        ensureCapacity();
        contents[count++] = toPush;

    }

    /**
     * Removes the element at the top of this stack and returns it.
     *
     * @return the element removed from the top of
```

```
the stack
     * returns null if empty
     */
    public T pop() {
        if (count == 0 ) { return null; }
        else { return contents[--count]; }
    }

     /**
     * Retrieves, but does not remove, the element at the top of this stack.
     *
     * @return the element at the top of the stack
     * returns null if empty
     */
    public T peek() {
        if (count == 0 ) { return null; }
        else { return contents[count-1]; }
    }

    /**
     *
     * @return the size of the stack
     */
    public int size() {
        return count;
    }

    /**
     * Returns a string representation of this stack. The string representation
     * consists of the elements in the stack, enclosed in curly braces ("{}"),
    * with the top of the stack indicated by ">" and separated by commas.
    * The top element is listed first, followed by the subsequent elements
    * in the order they would be popped off the stack.
    *
    * @return a string representation of this
```

```
stack
    */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("{>");
        for (int i = size()-1; i >= 0; i--) {
            sb.append(contents[i]);
            if (i > 0) {
                sb.append(", ");
            }
        }
        sb.append("}");
        return sb.toString();
    }

    /**
     * Private helper to ensure capacity of contents array for stack
     */
    private void ensureCapacity() {
        if (count == capacity) {
            capacity += capacityIncrement;
            T[] tempArray = (T[]) new Object[capacity];
            for (int i=0; i < size(); i++) {
                tempArray[i] = contents[i];
            }
            contents = tempArray;
        }
    }

}
```

```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 10/26/08
 *
 */
public interface Token
{

    public static final int PLUSMINUSPRECEDENCE = 1;
    public static final int MULTIPLYDIVIDEPRECEDENCE = 2;
    public static final int EXPONENTPRECEDENCE = 3;

    /** Processes the current token.  Since every token will handle
     *  itself in its own way, handling may involve pushing or
     *  popping from the given stack and/or appending more tokens
     *  to the output string.
     *
     *  @param s the Stack the token uses, if necessary, when processing itself.
     *  @return String to be appended to the output
     */
    public String 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();


}

```

```java
package test;
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;

import proj4.Stack;

/**
 *
 * Tests for generic stack class. Has tests for a string and integer stack, ensuring stack can be created regardless of type
 *
 * @author Neil Daterao
 * @version 2/22/2024
 *
 */
public class StackTest {

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

    private Stack<String> stringStack;
    private Stack<Integer> intStack;

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

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

```

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

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

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

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

    @Test
    public void testIsEmptyOnEmptyIntStack() {
        assertTrue("Expected the stack to be empty", intStack.isEmpty());
```

```
    }

    @Test
    public void testIsEmptyOnNonEmptyIntStack() {
        intStack.push(10);
        assertFalse("Expected the stack to be non-empty", intStack.isEmpty());
    }

    @Test
    public void testIsEmptyAfterPushAndPopOnIntStack() {
        intStack.push(10);
        assertFalse("Expected the stack to be non-empty after push", intStack.isEmpty());
        intStack.pop();
        assertTrue("Expected the stack to be empty after pop", intStack.isEmpty());
    }

    @Test
    public void testIsEmptyAfterPushAndPeekOnIntStack() {
        intStack.push(10);
        assertFalse("Expected the stack to be non-empty after push", intStack.isEmpty());
        intStack.peek();
        assertFalse("Expected the stack to be non-empty after peek", intStack.isEmpty());
    }

    @Test
    public void testPushElementOnIntStack() {
        intStack.push(10);
        assertEquals("Stack size should be 1 after pushing one element", 1, intStack.size());
        assertFalse("Stack should not be empty after pushing one element", intStack.isEmpty());
        assertEquals("The top element should be the one pushed", Integer.valueOf(10), intStack.peek());
```

```
    }

    @Test(expected = NullPointerException.class)
    public void testPushNullElementOnIntStack() {
        intStack.push(null);
    }

    @Test
    public void testPushMultipleElementsOnIntStack() {
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);
        assertEquals("Stack size should be 3 after pushing multiple elements", 3, intStack.size());
        assertEquals("The top element should be the last one pushed", Integer.valueOf(30), intStack.peek());
    }


    @Test
    public void testPopFromEmptyIntStack() {
        assertNull("Pop should return null when the stack is empty", intStack.pop());
    }

    @Test
    public void testPopFromNonEmptyIntStack() {
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);

        assertEquals("Pop should return the top element when the stack is non-empty", Integer.valueOf(30), intStack.pop());
        assertEquals("Stack size should decrease after pop", 2, intStack.size());
        assertEquals("The new top element should be the one pushed before the popped element", Integer.valueOf(20), intStack.peek());
```

```
    }

    @Test
    public void testPopUntilEmptyOnIntStack() {
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);

        assertEquals("Pop should return the top element when the stack is non-empty", Integer.valueOf(30), intStack.pop());
        assertEquals("Pop should return the top element when the stack is non-empty", Integer.valueOf(20), intStack.pop());
        assertEquals("Pop should return the top element when the stack is non-empty", Integer.valueOf(10), intStack.pop());
        assertTrue("The stack should be empty after popping all elements", intStack.isEmpty());
    }


    @Test
    public void testPeekOnEmptyIntStack() {
        assertNull("Peek should return null when the stack is empty", intStack.peek());
    }

    @Test
    public void testPeekOnNonEmptyIntStack() {
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);

        assertEquals("Peek should return the top element without removing it", Integer.valueOf(30), intStack.peek());
        assertEquals("Stack size should remain unchanged after peek", 3, intStack.size());
    }

```

```
    @Test
    public void testPeekDoesNotAffectIntStack() {
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);

        intStack.peek();

        assertEquals("Stack size should remain unchanged after peek", 3, intStack.size());
        assertFalse("Stack should not be empty after peek", intStack.isEmpty());
    }

    @Test
    public void testPushElementOnStringStack() {
        stringStack.push("one");
        assertEquals("Stack size should be 1 after pushing one element", 1, stringStack.size());
        assertFalse("Stack should not be empty after pushing one element", stringStack.isEmpty());
        assertEquals("The top element should be the one pushed", "one", stringStack.peek());
    }

    @Test(expected = NullPointerException.class)
    public void testPushNullElementOnStringStack() {
        stringStack.push(null);
    }

    @Test
    public void testPushMultipleElementsOnStringStack() {
        stringStack.push("one");
        stringStack.push("two");
        stringStack.push("three");
        assertEquals("Stack size should be 3 after pushing multiple elements", 3, stringStack.size());
```

```
        assertEquals("The top element should be the last one pushed", "three", stringStack.peek());
    }

    @Test
    public void testPopFromEmptyStringStack() {
        assertNull("Pop should return null when the stack is empty", stringStack.pop());
    }

    @Test
    public void testPopFromNonEmptyStringStack() {
        stringStack.push("one");
        stringStack.push("two");
        stringStack.push("three");

        assertEquals("Pop should return the top element when the stack is non-empty", "three", stringStack.pop());
        assertEquals("Stack size should decrease after pop", 2, stringStack.size());
        assertEquals("The new top element should be the one pushed before the popped element", "two", stringStack.peek());
    }

    @Test
    public void testPopUntilEmptyOnStringStack() {
        stringStack.push("one");
        stringStack.push("two");
        stringStack.push("three");

        assertEquals("Pop should return the top element when the stack is non-empty", "three", stringStack.pop());
        assertEquals("Pop should return the top element when the stack is non-empty", "two", stringStack.pop());
        assertEquals("Pop should return the top element when the stack is non-empty", "one",
```

```
stringStack.pop());
        assertTrue("The stack should be empty after popping all elements", stringStack.isEmpty());
    }

    @Test
    public void testPeekOnEmptyStringStack() {
        assertNull("Peek should return null when the stack is empty", stringStack.peek());
    }

    @Test
    public void testPeekOnNonEmptyStringStack() {
        stringStack.push("one");
        stringStack.push("two");
        stringStack.push("three");

        assertEquals("Peek should return the top element without removing it", "three", stringStack.peek());
        assertEquals("Stack size should remain unchanged after peek", 3, stringStack.size());
    }

    @Test
    public void testPeekDoesNotAffectStringStack() {
        stringStack.push("one");
        stringStack.push("two");
        stringStack.push("three");

        stringStack.peek();

        assertEquals("Stack size should remain unchanged after peek", 3, stringStack.size());
        assertFalse("Stack should not be empty after peek", stringStack.isEmpty());
    }


```

```

}
```

```java
package test;

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;

import proj4.*;

/**
 *
 * Tests for Tokens. Has 35 tests for various types of tokens and their respective handle methods
 *
 * @author Neil Daterao
 * @version 2/22/2024
 *
 */
public class TokenTester {
    @Rule
    public Timeout timeout = Timeout.millis(100);

    private Stack<Token> stack;

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

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


    @Test
      public void testRightParenToString() {
          Token rightParen = new RightParen();
```

```
            assertEquals(")", rightParen.toString());
        }


    @Test
        public void testLeftParenToString() {
            Token leftParen = new LeftParen();
            assertEquals("(", leftParen.toString());
        }


    @Test
        public void testExponentToString() {
            Token exponent = new Exponent();
            assertEquals("^", exponent.toString());
        }

    @Test
        public void testPlusToString() {
            Token plus = new Plus();
            assertEquals("+", plus.toString());
        }

    @Test
        public void testMinusToString() {
            Token minus = new Minus();
            assertEquals("-", minus.toString());
        }

    @Test
        public void testMultiplyToString() {
            Token multiply = new Multiply();
            assertEquals("*", multiply.toString());
        }

    @Test
        public void testDivideToString() {
            Token divide = new Divide();
            assertEquals("/", divide.toString());
        }

```

```
    @Test
    public void testSemicolonToString() {
        Semicolon sc = new Semicolon();
        assertEquals(";", sc.toString());
    }

    @Test
    public void testPlusHandleEmptyStack() {
        Plus plus = new Plus();

        assertEquals("Handle should return an empty string", "", plus.handle(stack));
        assertEquals("Handle an empty stack should push the Plus token onto the stack", "+", stack.peek().toString());
    }

    @Test
    public void testPlusHandleHigherPrecedence() {
        Plus plus = new Plus();

        stack.push(new Multiply()); // Higher precedence than Plus
        assertEquals("Handle when the top of stack contains elements of higher precedence should pop Multiply and add plus to stack", "*", plus.handle(stack));
        assertEquals("+", stack.peek().toString());
    }

    @Test
    public void testPlusHandleEqualPrecedence() {
        Plus plus = new Plus();

        stack.push(new Minus()); // Lower precedence than Plus
        assertEquals("Handle when the stack contains elements of equal precedence should pop minus and add plus to stack", "-", plus.handle(stack));
```

```
        assertEquals("+", stack.peek().toString());
    }

    @Test
    public void testPlussHandleLeftParenStopAtParen() {
        Plus plus = new Plus();

        stack.push(new LeftParen());
        stack.push(new Minus());
        assertEquals("Handle should not pop elements beyond LeftParen", "-", plus.handle(stack));
        assertEquals("Minus should be pushed onto the stack after LeftParen", "{>+, (}", stack.toString());
    }



    @Test
    public void testMinusHandleEmptyStack() {
        Minus minus = new Minus();

        assertEquals("Handle should return an empty string", "", minus.handle(stack));
        assertEquals("Handle an empty stack should push the Minus token onto the stack", "-", stack.peek().toString());
    }

    @Test
    public void testMinusHandleHigherPrecedence() {
        Minus minus = new Minus();

        stack.push(new Multiply()); // Higher precedence than Minus
        assertEquals("Handle when the top of stack contains elements of higher precedence should pop
```

```
 Multiply and add Minus to stack", "*", minus.handle(stack));
        assertEquals("-", stack.peek().toString());
    }


    @Test
    public void testMinusHandleEqualPrecedence() {
        Minus minus = new Minus();

        stack.push(new Plus()); // Lower precedence than Minus
        assertEquals("Handle when the stack contains elements of equal precedence should pop Plus and add Minus to stack", "+", minus.handle(stack));
        assertEquals("-", stack.peek().toString());
    }

    @Test
    public void testMinusHandleLeftParenStopAtParen() {
        Minus minus = new Minus();

        stack.push(new LeftParen());
        stack.push(new Plus());
        assertEquals("Handle should not pop elements beyond LeftParen", "+", minus.handle(stack));
        assertEquals("Minus should be pushed onto the stack after LeftParen", "{>-, (}", stack.toString());
    }

    @Test
    public void testMultiplyHandleEndUpOnStack() {
        Multiply multiply = new Multiply();

        assertEquals("Handle should return an
```

```
empty string", "", multiply.handle(stack));
        assertEquals("Multiply object should end up on stack", "*", stack.peek().toString());
    }

    @Test
    public void testDivideHandleEndUpOnStack() {
        Divide divide = new Divide();

        assertEquals("Handle should return an empty string", "", divide.handle(stack));
        assertEquals("Divide object should end up on stack", "/", stack.peek().toString());
    }

    @Test
    public void testMultiplyHandleHigherPrecedence() {
        Multiply multiply = new Multiply();

        stack.push(new Exponent()); // Higher precedence than Multiply
        assertEquals("Handle when the top of stack contains elements of higher precedence should pop Exponent and add Multiply to stack", "^", multiply.handle(stack));
        assertEquals("*", stack.peek().toString());
    }

    @Test
    public void testDivideHandleHigherPrecedence() {
        Divide divide = new Divide();

        stack.push(new Exponent()); // Higher precedence than Divide
        assertEquals("Handle when the top of stack contains elements of higher precedence should pop Exponent and add Divide to stack", "^", divide.handle(stack));
```

```
        assertEquals("/", stack.peek().toString());
    }

    @Test
    public void testMultiplyHandleLowerPrecedence() {
        Multiply multiply = new Multiply();

        stack.push(new Plus()); //Lower precedence than Multiply
        assertEquals("Handle when the top of stack contains elements of lower precedence should add multiply to stack", "", multiply.handle(stack));
        assertEquals("*", stack.peek().toString());
        assertEquals("{>*, +}", stack.toString());
    }

    @Test
    public void testDivideHandleLowerPrecedence() {
        Divide divide = new Divide();

        stack.push(new Plus()); //Lower precedence than Divide
        assertEquals("Handle when the top of stack contains elements of lower precedence should add divide to stack", "", divide.handle(stack));
        assertEquals("/", stack.peek().toString());
        assertEquals("{>/, +}", stack.toString());
    }


    @Test
    public void testMultiplyHandleEqualPrecedence() {
        Multiply multiply = new Multiply();

        stack.push(new Divide()); //Lower
```

```
precedence than Minus
        assertEquals("Handle when the top of stack contains elements of equal precedence should pop Divide and add multiply to stack", "/", multiply.handle(stack));
        assertEquals("*", stack.peek().toString());
        assertEquals("{>*}", stack.toString());
    }

    @Test
    public void testDivideHandleEqualPrecedence() {
        Divide divide = new Divide();

        stack.push(new Multiply()); //Lower precedence than Minus
        assertEquals("Handle when the top of stack contains elements of equal precedence should pop Multiply and add divide to stack", "*", divide.handle(stack));
        assertEquals("/", stack.peek().toString());
        assertEquals("{>/}", stack.toString());
    }

    @Test
    public void testMultiplyHandleStopAtLeftParen() {
        Multiply multiply = new Multiply();

        stack.push(new LeftParen());
        stack.push(new Exponent());
        assertEquals("Handle should not pop elements beyond LeftParen", "^", multiply.handle(stack));
        assertEquals("Plus should be pushed onto the stack after LeftParen", "{>*, (}", stack.toString());
    }

```

```java
    @Test
    public void testDivideHandleStopAtLeftParen() {
        Divide divide = new Divide();

        stack.push(new LeftParen());
        stack.push(new Exponent());
        assertEquals("Handle should not pop elements beyond LeftParen", "^", divide.handle(stack));
        assertEquals("Plus should be pushed onto the stack after LeftParen", "{>/, (}", stack.toString());
    }

    @Test
    public void testExponentHandleEndUpOnStack() {
        Exponent exponent = new Exponent();

        assertEquals("Handle should return an empty string", "", exponent.handle(stack));
        assertEquals("Exponent object should end up on stack", "^", stack.peek().toString());
    }

    @Test
    public void testExponentHandleEqualPrecedence() {
        Exponent exponent = new Exponent();

        stack.push(new Exponent());
        assertEquals("Handle when the stack contains elements of equal precedence should pop exponent and add exponent to stack", "^", exponent.handle(stack));
        assertEquals("^", stack.peek().toString());
    }

    @Test
    public void testExponentHandleLowerPrecedence
```

```
() {
        Exponent exponent = new Exponent();

        stack.push(new Multiply()); //Lower precedence than Divide
        assertEquals("Handle when the top of stack contains elements of lower precedence should add exponent to stack", "", exponent.handle(stack));
        assertEquals("^", stack.peek().toString());
        assertEquals("{>^, *}", stack.toString());
    }

    @Test
    public void testExponentsHandleLeftParenStopAtParen() {
        Exponent exponent = new Exponent();

        stack.push(new LeftParen());
        stack.push(new Exponent());
        assertEquals("Handle should not pop elements beyond LeftParen", "^", exponent.handle(stack));
        assertEquals("Exponent should be pushed onto the stack after LeftParen", "{>^, (}", stack.toString());
    }

    @Test
    public void testLeftParenHandleEmpty() {
        LeftParen lp = new LeftParen();

        assertEquals("Handle should return empty string", "", lp.handle(stack));
        assertEquals("Left paren should be added to stack", "(", stack.peek().toString());
    }

    @Test
    public void testLeftParenHandleAfterFewOperations() {
```

```
        LeftParen lp = new LeftParen();

        stack.push(new Exponent());
        stack.push(new LeftParen());
        stack.push(new Plus());
        assertEquals("Handle should return empty string", "", lp.handle(stack));
        assertEquals("Left paren should be added to stack", "(", stack.peek().toString());
        assertEquals("{>(, +, (, ^}", stack.toString());
    }

    @Test
    public void testRightParenHandle() {
        RightParen rightParen = new RightParen();

        stack.push(new LeftParen());
        stack.push(new Multiply());
        stack.push(new Plus());

        assertEquals("Returned string should be all items on stack down to LeftParen", "+*", rightParen.handle(stack));
        assertEquals("Stack should have all items down to LeftParen popped off", "{>}", stack.toString());
    }

    @Test
    public void testRightParenHandleTwoSetsOfLeftParen() {
        RightParen rightParen = new RightParen();

        stack.push(new LeftParen());
        stack.push(new Multiply());
        stack.push(new Plus());
        stack.push(new LeftParen());
        stack.push(new Multiply());
        stack.push(new Plus());

```

```
        assertEquals("Returned string should be all items on stack down to LeftParen", "+*", rightParen.handle(stack));
        assertEquals("Stack should have all items down to LeftParen popped off", "{>+, *, (}", stack.toString());
    }

    @Test
    public void testSemicolonHandle() {
        Semicolon semicolon = new Semicolon();

        stack.push(new Multiply());
        stack.push(new Plus());
        stack.push(new LeftParen());
        assertEquals("Semicolon should pop everything off the stack", "(+*", semicolon.handle(stack));
        assertEquals("Stack should be empty after semicolon handle", true, stack.isEmpty());
    }



}
```

```java
package test;
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;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;


import proj4.Converter;


/**
 * Testing class for Converter. Compares outputs based on a testing file and its solutions.
 */
public class ConverterTester {

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

    private Converter converter;
    private final String TESTFILEPATH = "/Users/neil/Desktop/CSC-151/Project 4 - The Infix to Postfix Converter/src/resources/converter_test.txt";
    private final String TESTFILEPATHSOLS = "/Users/neil/Desktop/CSC-151/Project 4 - The Infix to Postfix Converter/src/resources/converter_test_sols.txt";

    private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    private final PrintStream originalOut = System.out;
```

```java



    @Before
    public void setUp() throws Exception {
        converter = new Converter(TESTFILEPATH);
        System.setOut(new PrintStream(outContent));
    }

    @After
    public void tearDown() throws Exception {
        converter = null;
        System.setOut(originalOut);
    }


    @Test
    public void testConvert() {
        converter.convert();
        String expString = getSolsAsString(TESTFILEPATHSOLS);
        assertEquals("Output printed to console should match expected output", expString.trim(), outContent.toString().trim());

    }

    private String getSolsAsString(String FilePath) {
        try (BufferedReader reader = new BufferedReader(new FileReader(FilePath))) {
            StringBuilder expectedOutputBuilder = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                if (line.equals("EOF")) {
                    break;
                }
                expectedOutputBuilder.append(line + "\n");
```

```
            }
            return expectedOutputBuilder.toString();
        } catch (IOException e) {
            e.printStackTrace();
            return "";
        }
    }
}

```