```java
package proj4;
/**
 * the code that runs the converter using an input file
 * it creates a new converter and for each input file used and
 * prints the converted equations
 *
 * @author Matthew Caulfield
 * @version 10/23/17
 */

public class Client
{
    public static void main(String[] args)
    {
    String inputFile = "C:/Users/Matt/eclipse-
workspace/Caulfieldproj4/src/proj4/proj4_input.txt";
    Converter aConverter = new Converter(inputFile);
    aConverter.convert();
    }
}
```

```java
package proj4;

/**
 * Converts a series of prefix equation into postfix equations
 * that are in a given file. It uses the FileReader class to read the file
 * and each token used has its own handle method for how tokens should
 * be added to the postfix
 *
 * @author Matthew Caulfield
 * @version 10/22/2017
 *
 */
public class Converter {
	//stack that will hold tokens that are not operands
	//it is changed by the handle methods for tokens
	private Stack<Token> operationStack;
	//the file reader for file that is being converted
	private FileReader aFile;
	// the prefix string of the current equation
	//it is added to for every token parsed by the
	//prefix it is reset for every equation
	private String prefix;
	//the postfix string of the current equation
	//it is added to when ever deemed necessary by the
	//handle method of a token
	//postfix it is reset for every equation
	private String postfix;

	/**
	 * non-default constructor; Gradescope needs this to run tests
	 * @param infile path to the input file
	 */
```

```
public Converter(String infile){
    operationStack = new Stack<Token>();
    aFile = new FileReader(infile);
    postfix = "";
    prefix = "";
}

/**
 *convert converts equations in aFile instance variable
 *to postfix equations. It allows each token to handle its
 *what is added to the postfix string and what is added and removed to
 *the operationStack the postfix and prefix strings are reset for
 *every equation
 */
public void convert(){
   String currentSymbol = aFile.nextToken();
   while(currentSymbol != "EOF") {
          if(!currentSymbol.equals(";")) {
                prefix += currentSymbol;
          }
          if(currentSymbol.equals("(")) {
                LeftParen aLeftParen = new LeftParen();
                aLeftParen.handle(operationStack);
          }
          else if(currentSymbol.equals(")")){
                RightParen aRightParen = new RightParen();
                postfix += aRightParen.handle(operationStack);

          }
          else if(currentSymbol.equals(";")) {
                Semicolon aSemicolon = new Semicolon();
                postfix += aSemicolon.handle(operationStack);
                this.printPrefixAndPostfix();
                postfix = "";
                prefix = "";
          }
          else if(currentSymbol.equals("^")) {
                Exponent aExponent = new Exponent();
                postfix += aExponent.handle(operationStack);
          }
          else if(currentSymbol.equals("/")) {
                Divide aDivide = new Divide();
                postfix += aDivide.handle(operationStack);
          }
          else if(currentSymbol.equals("*")) {
                Multiply aMultiply = new Multiply();
                postfix += aMultiply.handle(operationStack);
          }
          else if(currentSymbol.equals("+")) {
                Plus aPlus = new Plus();
                postfix += aPlus.handle(operationStack);
          }
          else if(currentSymbol.equals("-")) {
                Minus aMinus = new Minus();
                postfix += aMinus.handle(operationStack);
```

```
            }
            else {
                  postfix += currentSymbol;
            }
            currentSymbol = aFile.nextToken();
      }
   }

   /**
    * prints the current postfix and prefix equation
    * in the proper format
    */
   private void printPrefixAndPostfix(){
      System.out.println(prefix + " --> " + postfix);
   }
}
package proj4;

/**
 * The divider class is both a token and an operator
 * it contains information on how to process the divider
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Divide implements Token, Operator{
      //the precedence of the divide operator and token
      //it should never be changed by any method
      private final int precedence = 2;

      /**
       * returns the precedence of divide
       * @return the precedence of the divide class
       */
      public int getPrecedence() {
            return precedence;
      }

      /**
       * @return divide as a string "/"
       */
      public String toString() {
            return "/";
      }

      /**
       * processes the divide token by popping any operator with a
       * greater than or equal to precedence until the stack is
       * empty or you the top of the stack is a left parenthesis
       * @param the stack of values to handle
       * @return the string containing all of the tokens that were popped
       */
      public String handle(Stack<Token> aStack) {
            String toReturn = "";
            boolean stillPop = true;
```

```
			while(stillPop) {
				if(aStack.isEmpty()) {
					stillPop = false;
				}
				else if(aStack.peek().toString().equals("(")){
					stillPop = false;
				}
				else if(((Operator)aStack.peek()).getPrecedence() <
this.getPrecedence()){
					stillPop = false;
				}
				else{
					toReturn += aStack.pop();
				}
			}
			aStack.push(this);
			return toReturn;
		}
}
package proj4;

/**
 * The exponent class is both a token and an operator
 * it contains information on how to process the exponent
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Exponent implements Token, Operator{
		//the precedence of the exponent operator and token
		//it should never be changed by any method
		private final int precedence = 3;

		/**
		 * returns the precedence of exponent
		 * @return the precedence of the exponent class
		 */
		public int getPrecedence() {
			return precedence;
		}

		/**
		 * @return exponent as a string "^"
		 */
		public String toString() {
			return "^";
		}

		/**
		 * processes the exponent token by popping any operator with a
		 * greater than or equal to precedence until the stack is
		 * empty or you the top of the stack is a left parenthesis
		 * @param the stack of values to handle
		 * @return the string containing all of the tokens that were popped
		 */
```

```
	public String handle(Stack<Token> aStack) {
		String toReturn = "";
		boolean stillPop = true;
		while(stillPop) {
			if(aStack.isEmpty()) {
				stillPop = false;
			}
			else if(aStack.peek().toString().equals("(")){
				stillPop = false;
			}
			else if(((Operator)aStack.peek()).getPrecedence() <
this.getPrecedence()) {
				stillPop = false;
			}
			else{
				toReturn += aStack.pop();
			}
		}
		aStack.push(this);
		return toReturn;
	}

}


package proj4;

/**
 * The left paren class is a token for the left parenthesis
 * it contains information on how to process the left parenthesis
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class LeftParen implements Token{

	/**
	 * returns the left parenthesis as a string
	 * @return returns a string of left Parenthesis
	 */
	public String toString() {
		return "(";
	}

	/**
	 * process the left parenthesis token
	 * it returns null and it pushes the
	 * left parenthesis to the top of the stack
	 * @param the stack of values to handle
	 * @return  null because nothing should be popped
	 */
	public String handle(Stack<Token> aStack) {
		aStack.push(this);
		return null;
	}
```

```
}
```

```
package proj4;
import proj4.ListNode;

/**
 *This is the Linked List class it acts like a list
 *but its data is stored in different nodes
 *@author Matthew Caulfield
 *@version 10/13/17
 *
 * 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 LinkedList<T>
{
    private int length;          // number of nodes
    private ListNode firstNode;  // pointer to first node

    public LinkedList()
    {
        length=0;
        firstNode=null;
    }

    /** insert new String at linked list's head
     *
     * @param toPush the String to be inserted
     */
    public void insertAtHead(T toPush)
    {
      ListNode newnode = new ListNode(toPush);
        if (isEmpty())
        {
            firstNode=newnode;
        }
        else
        {
            newnode.next=firstNode;
            firstNode=newnode;
        }
        length++;
    }

    /** remove and return data at the head of the list
     *
     *  @return the String the deleted node contains.  Returns null if list empty.
     */
    public T removeHead()
    {
      if(this.isEmpty()) {
            return null;
      }
```

```
   else {
          T firstNodeT = (T) firstNode.data;
          firstNode = firstNode.next;
          length --;
          return firstNodeT;
   }

}

/** insert data at end of list
 *
 * @param newData new String to be inserted
 */
public void insertAtTail(T newData)
{
   ListNode newNode = new ListNode(newData);
   ListNode currentNode = firstNode;
   if(this.isEmpty() == true) {
          firstNode = newNode;
   }
   else {
          for(int i = 0; i < this.getLength()-1; i++) {
                currentNode = currentNode.next;
          }
          currentNode.next = newNode;
   }
   length++;
}



/**
 * Inserts a node into the LL at the index given the following nodes
 * get pushed back an index number. If the number of nodes is less than the index
 * than the data is added at the tail
 * @param index number of where the new string should be added
 */
public void insertAtIndex(T newData, int index) {
   ListNode newNode = new ListNode(newData);
   ListNode currentNode = firstNode;
   if(index == 0) {
          this.insertAtHead(newData);
   }
   else if(this.getLength() > index) {
          for(int i = 0; i < index - 1; i++) {
                currentNode = currentNode.next;
          }
          newNode.next = currentNode.next;
          currentNode.next = newNode;
          length++;
   }
   else {
          this.insertAtTail(newData);
   }
}
```

```
    /**
     * returns the data at a given index
     * if the index does not exist it returns null
     * @param the index that holds the returned data
     * @return data at the given index
     */
    public T returnByIndex(int index) {
      if(index < 0) {
            return null;
      }
      else if(index < this.getLength()) {
            ListNode currentNode = firstNode;
                  for(int i = 0; i < index; i++) {
                        currentNode = currentNode.next;
                  }
                  return (T) currentNode.data;
      }
      else {
            return null;
      }
    }

    /**
     * changes the data of the node at the given index
     * @param the new data for the node
     * @param the index at which the data is changed
     */
    public void changeAtIndex(T newData, int index) {
      if(index >= 0 && index < this.getLength()) {
            ListNode currentNode = firstNode;
            for(int i = 0; i < index; i++) {
                  currentNode = currentNode.next;
            }
            currentNode.data = newData;
      }
    }

    /**
     * search for first occurrence of value and return index where found
     *
     * @param value string to search for
     * @return index where string occurs (first node is index 0).  Return -1 if value
not found.
     */
    public int indexOf(String value)
    {
      boolean found = false;
      int i = 0;
      ListNode currentNode = firstNode;
      while(found == false && i < this.getLength()) {
            if(currentNode.data.equals(value)) {
                  found = true;
                  return i;
            }
```

```
            i++;
            currentNode = currentNode.next;
        }
        return -1;
    }


    /**
     *  @return return linked list as printable string
     */
    public String toString()
    {
        String toReturn="(";
        ListNode runner=firstNode;
        while (runner!=null)
        {
            toReturn = toReturn + runner;  //call node's toString automatically
            runner=runner.next;
            if (runner!=null)
            {
                toReturn = toReturn + ",";
            }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }

    /**
     * @return returns a copy of the Linked List
     */
    public LinkedList clone(){
        LinkedList aClone = new LinkedList();
        for(int i = 0; i < this.getLength(); i++) {
            aClone.insertAtTail(this.returnByIndex(i));
        }
        return aClone;
    }

    /**
     *
     * @return length of LL
     */
    public int getLength() {return length;}

    /**
     *
     * @return true if LL empty or false if not
     */
    public boolean isEmpty() {return getLength()==0;}
}


package proj4;

/**
```

```java
 * The ListNode class is more data-specific than the LinkedList class.  It
 * details what a single node looks like.  This node has one data field,
 * holding a pointer to an object.
 *
 * This is the only class where I'll let you use public instance variables.
 *
 */
public class ListNode<T>
{
       //data is stored here and can not be changed
    public T data;
    //points to the next list node
    public ListNode next;

    /**
     * default constructor takes an object that is
     * the data to be stored in the node
     * @param the data to be stored
     */
    public ListNode(T new_data)
    {
        data = new_data;
        next = null;
    }

    /**
     * returns the data as a string
     */
    public String toString(){
      return data.toString();
    }

}


package proj4;

/**
 * The minus class is both a token and an operator
 * it contains information on how to process the plus
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Minus implements Token, Operator{
       //the precedence of the minus operator and token
       //it should never be changed by any method
       private int precedence = 1;

       /**
        * returns the precedence of minus
        * @return the precedence of the minus class
        */
       public int getPrecedence() {
              return precedence;
```

```
	}

	/**
	 * @return minus as a string "-"
	 */
	public String toString() {
		return "-";
	}

	/**
	 * processes the minus token by popping any operator until the stack is
	 * empty or you the top of the stack is a left parenthesis
	 * @param the stack of values to handle
	 * @return the string containing all of the tokens that were popped
	 */
	public String handle(Stack<Token> aStack) {
		String toReturn = "";
		boolean stillPop = true;
		while(stillPop) {
			if(aStack.isEmpty()) {
				stillPop = false;
			}
			else if(aStack.peek().toString().equals("(")){
				stillPop = false;
			}
			else{
				toReturn += aStack.pop();
			}
		}
		aStack.push(this);
		return toReturn;
	}
}


package proj4;

/**
 * The multiply class is both a token and an operator
 * it contains information on how to process the multiplier
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Multiply implements Token, Operator{
	//the precedence of the multiply operator and token
	//it should never be changed by any method
	private final int precedence = 2;

	/**
	 * returns the precedence of multiply
	 * @return the precedence of the multiply class
	 */
	public int getPrecedence() {
		return precedence;
```

```
	}

	/**
	 * @return multiply as a string "*"
	 */
	public String toString() {
		return "*";
	}

	/**
	 * processes the multiply token by popping any operator with a
	 * greater than or equal to precedence until the stack is
	 * empty or you the top of the stack is a left paranthesis
	 * @param the stack of values to handle
	 * @return the string containing all of the tokens that were popped
	 */
	public String handle(Stack<Token> aStack) {
		String toReturn = "";
		boolean stillPop = true;
		while(stillPop) {
			if(aStack.isEmpty()) {
				stillPop = false;
			}
			else if(aStack.peek().toString().equals("(")){
				stillPop = false;
			}
			else if(((Operator)aStack.peek()).getPrecedence() <
this.getPrecedence()) {
				stillPop = false;
			}
			else{
				toReturn += aStack.pop();
			}
		}
		aStack.push(this);
		return toReturn;
	}
}

package proj4;

/**
 * describes the method that is needed to be an operator
 * every operator is a token
 * the method is getPrecedence
 *
 * @author Matthew Caulfield
 * @version 10/22/17
 *
 */
public interface Operator extends Token{
	/**
	 * returns the precedence of the operator
	 * @return the precedence
	 */
```

```
	public int getPrecedence();
}


package proj4;

/**
 * The plus class is both a token and an operator
 * it contains information on how to process the plus
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Plus implements Token, Operator{
	//the precedence of the plus operator and token
	//it should never be changed by any method
	private int precedence = 1;

	/**
	 * returns the precedence of plus
	 * @return the precedence of the plus class
	 */
	public int getPrecedence() {
		return precedence;
	}

	/**
	 * @return plus as a string "+"
	 */
	public String toString() {
		return "+";
	}

	/**
	 * processes the plus token by popping any operator until the stack is
	 * empty or you the top of the stack is a left parenthesis
	 * @param the stack of values to handle
	 * @return the string containing all of the tokens that were popped
	 */
	public String handle(Stack<Token> aStack) {
		String toReturn = "";
		boolean stillPop = true;
		while(stillPop) {
			if(aStack.isEmpty()) {
				stillPop = false;
			}
			else if(aStack.peek().toString().equals("(")){
				stillPop = false;
			}
			else{
				toReturn += aStack.pop();
			}
		}
		aStack.push(this);
		return toReturn;
```

```
	}
}
```

```
package proj4;

/**
 * The right paren class is a token for the right parenthesis
 * it contains information on how to process the right parenthesis
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class RightParen implements Token{

	/**
	 * returns the left parenthesis as a string
	 * @return returns a string of right Parenthesis
	 */
	public String toString() {
		return ")";
	}

	/**
	 * process the right parenthesis token and returns
	 * all tokens between this parenthesis and the next
	 * left parenthesis
	 * @param the stack of values to handle
	 * @return returns all tokens between this parenthesis
	 * and the next left parenthesis
	 */
	public String handle(Stack<Token> aStack) {
		String toReturn = "";
		while(!aStack.peek().toString().equals("(")) {
			toReturn += aStack.pop();
		}
		aStack.pop();
		return toReturn;
	}
}
```

```
package proj4;

/**
 * The semicolon class is a token for the semicolon
 * it contains information on how to process the semicolon
 * when called by the converter
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Semicolon implements Token{
	/**
	 * returns the semicolon as a string
```

```
         * @return returns a string of right semicolon
         */
        public String toString() {
               return ";";
        }

        /**
         * process the semicolon token and returns
         * all tokens left in the stack as a string
         * @param the stack of values to handle
         * @return returns all tokens left in stack as a
         * string
         */
        public String handle(Stack<Token> aStack) {
               String toReturn = "";
               while(!aStack.isEmpty()) {
                      toReturn += aStack.pop().toString();
               }
               return toReturn;
        }
}


package proj4;

/**
 * this class is an ADT that can hold objects and operates
 * on a last in first out basis
 * the last object added is the next object to be able to leave
 *
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class Stack<T>
{
        //contents holds all objects in the LinkedList the
        //only way to add objects to contents is through push
        //and the only way to remove an object is through pop
        private LinkedList contents;

        /**
         * default constructor for the stack method
         * initializes the stack with an empty LinkedList
         */
    public Stack() {
       contents = new LinkedList();

    }

    /**
     * returns whether or not the stack is empty
     * @return if the stack is empty
     */
    public boolean isEmpty() {
       return contents.isEmpty();
```

```
    }

    /**
     * adds a value to the top of the stack
     * @param toPush the value to push on top of the stack
     */
    public void push(T toPush) {
       contents.insertAtHead(toPush);
    }

    /**
     * removes and returns the value from the top of the stack
     * @return the removed first value in the stack
     */
    public T pop() {
       return (T) contents.removeHead();
    }

    /**
     * returns the first value in the stack without removing it
     * @return the first value in the stack
     */
    public T peek() {
       return (T) contents.returnByIndex(0);
    }

    /**
     * @return the size of the stack
     */
    public int size() {
       return contents.getLength();
    }

    /**
     * @return the stack as a string with a pointer to the top value
     * of the stack
     */
    public String toString() {
        String toReturn = "{>";
        int stackSize = this.size();
        for(int i = 0; i < stackSize; i++) {
             toReturn += contents.returnByIndex(i).toString();
             if(i<stackSize-1) {
                   toReturn += ",";
             }
        }
        toReturn+="}";
        return toReturn;
    }

}

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;

/**
 *
 * test the stack class
 *
 * @author Matthew Caulfield
 * @version 10/22/17
 *
 */
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.toString());
    }

    @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.toString().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}", stack.toString().replaceAll("[ ]+", ""));
    }

    @Test
    public void testPeek() {
       stack.push("A");
        stack.push("B");
        stack.push("C");
        assertEquals ("Pushing first A, then B, then C onto an empty stack.", "C",
stack.peek());
        assertEquals ("stack should not be altered", "{>C,B,A}",
stack.toString().replaceAll("[ ]+", ""));

    }

    @Test
    public void testPopEmptyStack() {
       assertEquals("popping an empty string should return null", null, stack.pop());

    }

    @Test
    public void testPopOneStack() {
       stack.push("A");
       assertEquals("popping a stack should return A", "A", stack.pop());
       assertEquals("stack should now be empty", "{>}", stack.toString());
    }

    @Test
    public void testPopTwoStack() {
       stack.push("A");
       stack.push("B");
       assertEquals("popping a stack should return B", "B", stack.pop());
       assertEquals("stack should now contain just A", "{>A}",
stack.toString().replaceAll("[ ]+", ""));
       assertEquals("popping a stack should return A", "A", stack.pop());
       assertEquals("stack should now be empty", "{>}", stack.toString());
    }

    @Test
    public void testPopThreeStack() {
       stack.push("A");
       stack.push("B");
       stack.push("C");
       assertEquals("popping a stack should return C", "C", stack.pop());
       assertEquals("stack should now contain just A and B", "{>B,A}",
stack.toString().replaceAll("[ ]+", ""));
       assertEquals("popping a stack should return B", "B", stack.pop());
       assertEquals("stack should now contain just A", "{>A}",
stack.toString().replaceAll("[ ]+", ""));
       assertEquals("popping a stack should return A", "A", stack.pop());
       assertEquals("stack should now be empty", "{>}", stack.toString());
```

```
    }
}
```

```
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
{
       /** 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();

}
```

```
package proj4;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

/**
 *
 * test all tokens and operators
 * only needs to test one token of precedent 2 or 3 and
 * one token of precedent 1 because 2 or 3 share the same
 * logic and all precedent 1 share the same logic
 *
 * @author Matthew Caulfield
 * @version 10/22/17
 */
public class TokenTest {
       private Divide divide;
       private Exponent exponent;
```

```
   private LeftParen leftParen;
   private Minus minus;
   private Multiply multiply;
   private Plus plus;
   private RightParen rightParen;
   private Semicolon semicolon;
   private Stack <Token> aStack;

   @Before
public void setUp() throws Exception {
    divide = new Divide();
    exponent = new Exponent();
    leftParen = new LeftParen();
    minus = new Minus();
    multiply = new Multiply();
    plus = new Plus();
    rightParen = new RightParen();
    semicolon = new Semicolon();
    aStack = new <Token>Stack();
}

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

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

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

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

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

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

   @Test
   public void testRightParenToString() {
          assertEquals(")", rightParen.toString());
   }
```

```
	@Test
	public void testSemicolonToString() {
		assertEquals(";", semicolon.toString());
	}

	@Test
	public void testDivideGetPrecedence() {
		assertEquals(2, divide.getPrecedence());
	}

	@Test
	public void testExponentGetPrecedence() {
		assertEquals(3, exponent.getPrecedence());
	}

	@Test
	public void testMinusGetPrecedence() {
		assertEquals(1, minus.getPrecedence());
	}

	@Test
	public void testMultiplyGetPrecedence() {
		assertEquals(2, multiply.getPrecedence());
	}

	@Test
	public void testPlusGetPrecedence() {
		assertEquals(1, plus.getPrecedence());
	}

	@Test
	public void testPrecedentTwoOrThree() {
		aStack.push(divide);
		aStack.push(exponent);
		aStack.push(leftParen);
		aStack.push(plus);
		aStack.push(multiply);
		assertEquals("*", divide.handle(aStack));
	}

	@Test
	public void testPrecedentTOne() {
		aStack.push(divide);
		aStack.push(exponent);
		aStack.push(leftParen);
		aStack.push(plus);
		aStack.push(minus);
		aStack.push(multiply);
		assertEquals("*-+", plus.handle(aStack));
	}

}
```