Due: Friday, April 21st at 11:59pm  
Worth: 75 points (65 functionality, 10 design/style)

Goals  
- to use the built-in stack class (import java.util.Stack)  
- to catch and handle exceptions (import java.util.EmptyStackException and java.util.NoSuchElementException)  
- to use (not build) an applet to exercise your code in addition to using JUnit  
- to focus on the parts of a project you need to change and just use the rest  
- to build the class file from scratch but meeting the required specifications

Overview  
For this assignment, you'll build a postfix expression evaluator class (PostfixEvaluator). This will use a stack to hold the numbers. You can interact with your code through an applet that calls it. There are also JUnit tests which your program needs to pass.

You do not write the stack, the token scanner, the applet, or the JUnit tests. You write the expression evaluator class from scratch. There are several supporting files here to make your job easier. Get them from the AutoLab handout.

Specifications  
Create a PostfixEvaluator class that has a public static evaluate method. That method takes in a String as input and returns the String result of evaluating the input as a postfix expression. Note that the string is returned so that the error messages can be naturally returned. Use the algorithm we've discussed in class and in the book.

For example, an expression of 15 4 * 24 - 0.5 * 100 3 7 + / * should result in 180.0

You must use a Stack of Double in your evaluation.

Input numbers can be real numbers (normal notation) but may not be negative. The result may be negative.

Allowed operations are addition(+), subtraction(-), multiplication(*), and division(/). Any other operator should cause an error message to be printed.

The evaluate method returns a string. That string is the value of the postfix expression if it is well-formed and leaves the stack empty. If there are extra values on the stack, the method returns an error message which may include the result but must note that it is not valid.
If an exception occurs, your code should catch it and return a string describing the problem. Your program should not throw any exceptions. Your messages should be very similar to the following.

<table>
<thead>
<tr>
<th>Unusual or exceptional condition</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>input string is empty or all spaces or tabs</td>
<td>No input.</td>
</tr>
<tr>
<td>EmptyStackException</td>
<td>Stack Underflow. Not enough operands on stack.</td>
</tr>
<tr>
<td>ArithmeticException</td>
<td>Infinity</td>
</tr>
<tr>
<td>values remain on stack after end of input</td>
<td>Computed answer, but values remain on stack.</td>
</tr>
<tr>
<td>evaluation ended before end of input</td>
<td>Computed answer, but not all input used.</td>
</tr>
<tr>
<td>parentheses in input</td>
<td>( has no meaning here</td>
</tr>
</tbody>
</table>

**Implementation Details**

The PostfixEvaluator class does not have a constructor and does not create any instances. It does not have instance variables. It has a public static `evaluate` method that takes in a String and returns a String as noted above.

Hints: Your solution could use a private static method (not visible outside the class) that takes a stack and the Character operator and performs the operation (similar to the one in the arithmetic evaluation example we did in class). Any exceptions raised in that method could be handled in the `evaluate` method. A switch statement could come in handy; remember the breaks.

You are provided `TokenScanner` and `Token` classes you must use. Do not modify them. They have Javadoc comments. The TokenScanner class breaks a string into parts called Tokens. You can ask the Tokens whether they are numbers or operators and handle them appropriately. These classes have Javadoc comments and pages. You may also read the code.

Construct a TokenScanner by passing in a String. The String argument passed in to the evaluate method is what you need to tokenize. Write a loop using `hasNextToken` and `nextToken` much as you would for a Scanner.

The Token class is documented and has methods `isOperator()`, `isNumber()`, `numberValue()`, and `operatorCharValue()`. Note that Token recognizes left and right parentheses although they are not part of valid postfix expressions. They are valid for infix expressions. There is also information about operator precedence in Token. Ignore that.

Your evaluate method should catch a variety of exceptions. For some of those (NoSuchElementException and ArithmeticException), the action can be `return e.getMessage();`. Your code should not let any reasonable exceptions escape; the JUnit tests attempt to raise the
exceptions we expect your program to handle. If you pass those tests, the exception handling is okay.

Here is sample code for catching and handling multiple exceptions:

```java
try {
    // code that we are attempting
} catch (<exception type> e) {
    // code that handles the exception
} catch (<exception type> e) {
    // code that handles the exception
}
```

For example:

```java
try {
    // some code
} catch (ArithmeticException e) {
    // returns the message associated with the exception
    return e.getMessage();
} catch (EmptyStackException e) {
    return "Stack Underflow. Not enough operands on stack.";
}
```

**Ideas for Getting Started**

1. Understand the problem and the many resources you have.
2. Download the files and create a project.
3. Create a PostfixEvaluator class with an evaluate method. Have it return its input as its result.
4. Run the PostfixApplet.
5. Write the loop that uses TokenScanner to break the input into Tokens. Look at the Tokens either by printing them to System.out or building a String and returning that.
6. Keep enhancing until you're done. Be sure to run the JUnit tests to try to uncover bugs.

Submit your PostfixEvaluator.java file as the postfix lab to Autolab. Points are 65 points from AutoLab testing and 10 points style/design.