

## Lab report X4 – Junit Testing and Linked Lists

---

### Exercise description:

[http://www.tfh-berlin.de/~ischmied/Inf2/Exercises/X4\\_JUnit\\_Testing.html](http://www.tfh-berlin.de/~ischmied/Inf2/Exercises/X4_JUnit_Testing.html)

### Work description:

Learn to test single units and their compositions systematically. Implement Linked Lists.

### 1. Preperation Questions

#### 1. *Shunting Yard Algorithm*

Before starting to work on lab tasks we searched for some definitions on the mentioned Shunting yard algoritmn and the Reverse Polish Notation.

Both famous types have good explanations on the web, so it was not hard to get information and working instructions on that. A detailed explanation with examples of transformed expressions can be found on the wikipedia page.

#### 2. *Data types for Queue and Stack*

In order to deal with digits and operators we decided to invent the respective queue and stack class which takes linked lists of characters as the respective data types.

Later on coming to the voluntary part we extended our program to deal with multi-digit numbers while data types keep the same.

Obviously it is also possible to set the mentioned data types to string as we are dealing with strings all over here.

### 2. Lab experiments

#### 1. *Implementing Queue and Stack*

First task was to implement a queue and a stack class by using the datatype we have chosen in the prep questions above.

Therefor we implemented first an interface with the given specification:

```
public void add (<element> e);  
public <element> remove() throws EmptyExeception;
```

Figure 1: Interface to be implemented by queue and stack class

We chosed `Character` as the generic datatype for our interface as we wanted to deal with single characters from an input string for the first solution.

After that we simply used eclipse to create two classes both implementing our generated interface. The implementation was nearly the same for both classes except from the internal work of the add

method. For the queue class the add method has to add a value at the end of our linked list while the add method of the stack class has to add a new value at the beginning of the list.

Both classes got additionally some more methods for removing or getting the first or last element as helper methods.

### 1.1. Testing

Testing our self developed data types was a new thing for us also I did it few times on work. We tried to be as close as we can at the “XP” (*eXtreme Programming*) development model which offers the „test-before-code“ paradigm which means that test classes have to be implemented before real classes are implemented. So we only created the empty classes `RPNQueue.java` and `RPNStack.java` for existence reason and then started alternately writing a test method and writing the real method. A little bit hard at the beginning but with ongoing activities we became familiar with this kind of work.

### 2. RPN class

Having necessary datatypes created we started to implement the main class for the rpn calculation. This class uses both classes, `RPNQueue.java` and `RPNStack.java` as data elements for recalculating an infix formula into a rpn formula.

The method signature for the `toRPN` method was giving. This method should transform the infix to a rpn formula returning it as a string. As mentioned above the wiki page for the *shunting-yard-algorithm* was a great help for understanding how the transformation can be done. So we followed these instructions to iterate through a given formula string and decided on each character what should be done. Here we needed another method `isNumber()` to simply decide if the current character is a number or not as this is important for dealing with each character.

The second helper method `isHigher()` was needed to define which operator symbol has which precedence over another one. This was more complicated later when we dealt with brackets and powers of numbers as well as here the order is much more important.

```
protected static boolean isNumber(char c) {  
  
    try {  
        int i = Integer.parseInt(String.valueOf(c));  
        return true;  
    }  
    catch (Exception e) {  
        return false;  
    }  
}
```

Figure 2: helper method `isNumber` to determine characters as numbers

```
protected static boolean isHigher(char c1, char c2) {  
  
    // all are higher than open parenthesis  
    if ( (c1 == '-' || c1 == '+' || c1 == '*' || c1 == '/') && (c2 == '(') ) {  
        return true;  
    }  
  
    if ( (c1 == '*' || c1 == '/') && (c2 == '+' || c2 == '-') ) {  
        return true;  
    }  
}
```

```

// power is always higher, also if c2 is power too
if (c1 == '^') {
    return true;
}

return false;
}

```

Figure 3: helper method isHigher to determine operator precedences

Another problem which occurred during testing our transformation was the problem that the given method signature defines an integer type as the return type. This is ok for addition and subtraction calculations but not for divisions. So we decided to change its signature to return values with the type double to be more flexible.

Again we tested not only the mentioned `toRPN()` method but our new invented methods as well.

### 3. *evaluateRPN()* method

The third task was to arithmetically calculate the result of the given formula using the prior transformed `rpnFormula`. Here we had to research a possible algorithm for dealing with this new notation.

It was not so hard as you always have to search for an operator and take the two numbers in front of it from the formula, calculating it and storing the result back at this position.

An error that occurred later was a problem with brackets. As it can be seen in the screenshots we started from beginning to deal with brackets as well as power calculations in our expressions. But if an open bracket is typed in as the very first character our program stopped. This was due to the fact that the program wants to read out all elements from the stack which is at this moment of course empty.

Solving the problem was than easy as we first extended our test cases and afterwards implemented the necessary check for an empty stack.

As mentioned above we changed the method signature to return double values as the arithmetically calculated result will be of type double as well.

### 4. *Applet*

Finally, to display our results and be able to input a formula we created an applet allowing the user to input an infix formula as well as an `rpn` formula. By submitting it the transformed `rpn` formula as well as the calculated result are displayed in the applet.

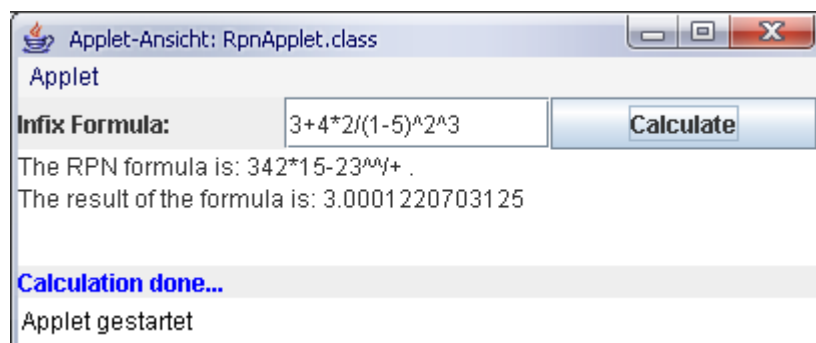


Figure 4: Applet displaying input and calculated values

If things went wrong during transformation or calculation a textfield indicates this by displaying a

respective error message.

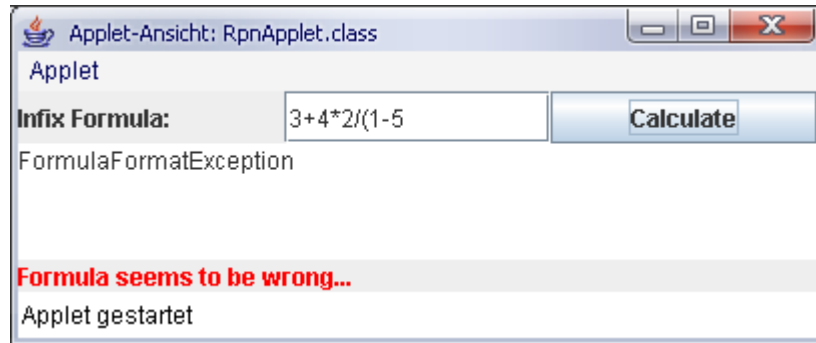


Figure 5: Error message indicating calculation error

## Voluntary work tasks:

### 6. Dealing with multi-digit-numbers

Up to here our RPN class only allowed input formulas with single-digit numbers in order to be able to calculate the result. Now for dealing with multi-digit numbers we had to refactor our program by changing several parts of our class.

We decided to extend our prior RPN class in order to use our helper methods also in the new class without rewriting them. The `toRPN()` and `evaluateRPN()` methods will be overwritten as we had to change some steps here. The major place is of course at first the transformation method `toRPN()` to enable the usage of multi-digit numbers.

To be able to differentiate between multi-digit numbers in our transformed rpn formula we decided to add a separator after each (multi-digit) number. We've chosen '#' as a separator as it does not appear in a normal arithmetical formula.

Here our test cases helped us out finding errors occurred in our method as we easily had seen them. A typical problem was the error that the last digit was not followed by a separator sign. This was due to the problem that it is added last and afterwards no loop run is done.

We solved it by adding this separator when combining the queue and the stack.

The `evaluateRPN()` method did not required a lot of changes. We only had to assemble each multi-digit by hand by finding the separator sign. The calculation itself is done exactly the same way as with single-digit numbers.

**Time usage:**

Unfortunately this lab takes again a lot of time for research, implementation and combination of subtopics.

The lab time in class was used to research information about mentioned algorithm and starting to implement respective the interface and classes for stack and queue. The rpn as well as the applet then takes much more time as expected.

Additionally we tried to implement our program with test programming as mentioned in the XP development model which takes even more time as we had to become familiar with that software development approach. But finally this solved a lot of problems as we had to write our tests before started to implement the respective methods. Also for overall tests in later development steps it was very helpful and should be included as lab task in future topics.

As learned from my work experiences we tried to work as a team with a software repository management.

**Source codes:**

This time we created a website with the described applet and links to our source code files. It can be found here: [http://www.f4.fhtw-berlin.de/~s0517903/Info/lab4/x4\\_applet.html](http://www.f4.fhtw-berlin.de/~s0517903/Info/lab4/x4_applet.html)

## **Appendix:**

### **Attention:**

Please have in mind that the appended source codes are only for completeness reasons. These can also be downloaded from the mentioned website.

#### **1.Link to website which includes applet**

[http://www.f4.fhtw-berlin.de/~s0517903/Info/lab4/x4\\_applet.html](http://www.f4.fhtw-berlin.de/~s0517903/Info/lab4/x4_applet.html)

#### **2.Link to JavaDoc location**

<http://www.f4.fhtw-berlin.de/~s0517903/Info/lab4/doc/index.html>