Lab Report

EXERCISE 6: REVERSE POLISH NOTATION

	Name	Titel des Kurses	Datum
Juri Wiechmann Bartholomäus Berresheim	571085 568624	Prof. Dr. Weber-Wulff Info 2 Group 2	1.12.2019

Index

Introduction

In this week's lab we worked with the Reverse Polish Notation and Stacks. We had to write our own stack class and also implement multiple functions around the subject of the Reverse Polish Notation.

PRE-LAB

2. **Lukasiewicz was a Polish logician, so his notation for parentheses-free expressions is often called Reverse Polish Notation. To get your brain in gear, convert the following expressions to RPN! What are the values of the expressions?**

a.	1 * 2 + 3	= 12*3 +	= 5
b.	1 + 2 * 3	= 123*+	= 7
c.	1 + 2 – 3 ^ 4	= 12+34^-	= -78
d.	1 ^ 2 -3 * 4	= 12^34*-	= -11
e.	1 + 2 * 3 – 4 ^ 5 + 6	= 123*+45^-6+	= -1011
f.	(1 + 2) * 3 + (4 ^ (5 – 6))	= 12+3*456-^+	= 9.25
g.	1 + 2 + 3 / 4 + 5 + 6 * (7 + 8)	= 12+34/+5+678+*+	= 98.75
h.	9 - 1 - 2 - 3 * 2 – 1	= 91-2-32*-1-	= -1

3. **For the infix expression** `a + b ^ c * d ^ e ^ f - g - h / ( i + j )`**, do the following:**
   a. **Show how to generate the corresponding postfix expression.**

To convert infix to postfix, we have to do it while looking at the precedence of the operators. Parenthesis have the highest precedence, so we start with them.

(i+j) turns into "ij+"

then we look at the operator with the second highest precedence, which is the exponent operator '^'.

So "b^c" and "d^e^f" turn into "bc^" and "def^^".

One precedence lower, we have the multiplication and division operators.

That means "bc^ * def^^" and "h / ij+" turn into "bc^def^^*" and "hij+/"

And finally, we have addition and subtraction left.

So "a + bc^def^^* - g - hij+/" turns into "abc^def^^*+g-hij+/-"

   b. **Show how to evaluate the resulting postfix expression.**

When evaluating a postfix expression, we always take a pair of operands from the left side of the operator to do the calculation. Since we are working with letters and not numbers, we can't calculate the result. Which is why I'll assign a value to each of the letters for demonstration purposes.

a = 6, b = 4, c = 3, d = 8, e = 1, f = 3, g = 2, h = 5, i = 8, j = 7

Which means our postfix “abc^def^^*+g-hij+/-” turns into “643^813^^*+2-587+/-”.

In our new postfix , we have 3 of the previously mentioned pairs: “43^”, “13^” and “87+”. Once evaluated they turn into “64”, “1” and “15” respectively.

After that, the postfix looks like this “6 (64)81^*+2-5 (15)/-”
(I put the multi digit numbers into parenthesis so that we can better differentiate between numbers.

If we continue the same way, we get something that looks like this:

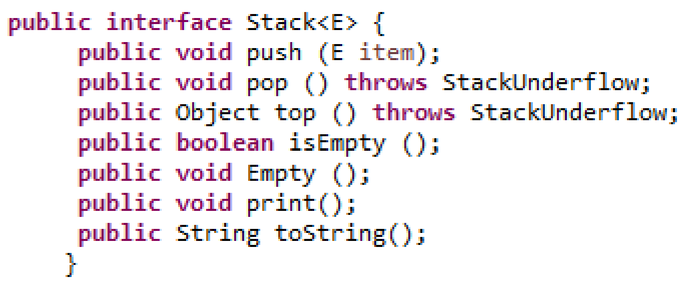
“6 (64)81^*+2-5 (15)/-”	-> “6 (64)8*+2-(0.3)-”	//0.3 = 1/3
“6 (64)8*+2-(0.3)-”	-> “6 (512)+2-(0.3)-”	
“6 (512)+2-(0.3)-”	-> “(518)2-(0.3)-”	
“(518)2-(0.3)-”	-> “(516)(0.3)-”	
“(516)(0.3)-”	-> “515,7	//515 + 2/3

After the evaluation, we get 515,7 as our result.

Assignments

Read through all **of the exercises before starting! This exercise will not be done with pair programming, but with pairs working in parallel. ~~The groups will still be chosen at random~~. Choose your partner yourself. First agree on your interface in exercise 1. Then one person should get exercise 2 to work while the other one starts exercise 3. Then you exchange your code, and voilà, it works! Now you can get back together to do the fourth exercise, but without the gong. Include in your report a reflection on the differences in the tightly regulated pair programming and working together like this.**

1. **First agree on a Java interface for Stack. What methods do you need? What parameters will they take? What will they return? This is an abstract data type, so don't make it too specific to your exact needs, but what do you expect a Stack to offer. You should call the interface** `Stack.java`.



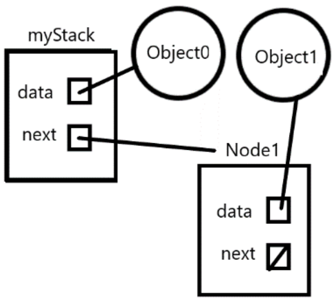
```java
public interface Stack<E> {
     public void push (E item);
     public void pop () throws StackUnderflow;
     public Object top () throws StackUnderflow;
     public boolean isEmpty ();
     public void Empty ();
     public void print();
     public String toString();
    }
```

So here we decided to work with generic types “<E>” to make it fit to all type. Later we worked with Objects but this way we can add different types into the same Stack. Our Print method just prints the `toString()` method.

2. **Implement a class `StackAsList.java` as discussed in the lecture, using a linked list of objects that you implement yourself! Don't use the Stack or LinkedList that is available by default in Java. Try and type it in yourself, not just copy the handout. How will you test this? Your class should include both an exception on stack underflow as well as stack overflow. Will you really need both exceptions? Why or why not?**

   Override the `toString()` method to provide a useful way of printing a stack. Now make it generic, so it can take values of any type.

Because we weren't allowed to use the given Java-Stack we needed to code our own one. A Stack consist of either a Linked List or an Array. For this task, we went with a Linked List, which is a construct of Node-Objects. A Node contains data in a field and points to the next one. And for out Linked List to work, we created an inner class "Node":



```
private class Node {

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Node() {
        data = null;
        next = null;
    }
    Object data;
    Node next;
    public String toString() {
        return data.toString();
    }
}
```

"myStack" is our Top (the top element of the Stack). After creating the Node class, we implemented all the methods from the Stack interface. We had learned in a previous lecture about how a Stack behaves. Let's quickly explain every method:

S With the push() method, we add an Object at the Top of the Stack. To do that, we create a temporary Node in which we save the new Object and point to the current Top. Then we just set the temporary Node to myStack and by this to the new Top.

```
public void push(Object item) {

    Node temp = new Node(item, myStack);
    myStack = temp;
}
```

The Top method just gives us the Data of our Top/Head/myStack.

```
@Override
public Object top() throws StackUnderflow {
    // TODO Auto-generated method stub
    return myStack.data;
}
```

With pop we pop the Top by assigning myStack to the next element in the Stack. The old Top then gets garbage collected since no Node is pointing to it.

```
public void pop() throws StackUnderflow {
    if(!isEmpty())
    myStack = myStack.next;
}
```

"isEmpty()" returns a boolean that indicates whether our Stack is empty or not. Its empty if our Top is null.

```java
public boolean isEmpty() {
    if(myStack == null) {
        return true;
    }
    else
        return false;
}
```

With "Empty()" we set our Top to null and all the other Nodes get garbage collected.

```java
public void Empty() {
    myStack = null;
}
```

Last but no least our "print()" and "toString()" method. Like we said before we just print the "toString()" method in print method.

```java
@Override
public void print() {
    System.out.println(this.toString());
}
```

Now the "toString()" method is more complicated. Because we don't have an index that we can count down we need to create an temporary node. We start at the Top. Each step in the loop we convert the data to a String and then we set our temporary node to the next one. That's go on till the next Node is null.

```java
@Override
public String toString() {
    String S_myString = "";
    Node temp = new Node();
    temp = myStack;
    while (isEmpty()) {
        if(temp.next == null) {
            break;
        }
        temp = temp.next;
        S_myString += temp.data + "\r\n";
    }
    return S_myString;
}
```

3. **Implement a class Postfix.java that has a method** `public int evaluate (String pfx) {...}` **that takes a String representing a postfix expression and determines the value represented by that expression. You will need to access the individual characters of the string and store them in a stack. This is necessary for the evaluation, luckily your partner is currently in the process of making one. Build a test class and check the postfix expressions you did in the finger exercises. If there is a difference between the value computed and the value expected, either you were wrong, or the implementation is wrong or both.**

To start things of, we implemented a class Postfix in which we created a evaluate() method, that takes a String pfx as input. We chose to make the method return a double instead of an int one, since we are working with divisions and want a more precise result. We then created a char field called token, and in the method we added 4 double variables called rhs, lhs, tokenInt and result respectively, followed by the creation of a new StackAsList object named stack.

```java
public class Postfix {

    char token;
```

```
public double evaluate (String pfx) throws StackUnderflow, StackException {


    double rhs= 0;
    double lhs = 0;
    double tokenInt = 0;

    double score = 0;

    StackAsList stack = new StackAsList();
```

We then added a for-loop in which we set the value of token to the char of the pfx String at the current index i. After that we wrote an if-statement that would check if the Ascii value of token is smaller than 58 and bigger than 47. If true, we would assign the numeric value of token to tokenInt using the getNumericValue() method from the Character class. Then we would push tokenInt on the stack using the push() method.

```
for (int i = 0; i <pfx.length(); i++)
{
    token = pfx.charAt(i);

    if (token > 47 && token < 58)
    {
        tokenInt = Character.getNumericValue(token);
        stack.push(tokenInt);
    }
```

If the conditions from the if-statement are not met, we have an else-if statement that would check if token is an operator. To do that, we wrote a method called isOperator() that would check if the inputted char is an operator and return a boolean value depending on the case.

```
public boolean isOperator (char token) {

    if (token == '+' ||
        token == '-' ||
        token == '*' ||
        token == '/' ||
        token == '^')
        return true;

    else
        return false;
}
```

In the else-if statement checking for operators, we first assign the top of the stack as value to rhs and then pop it from the stack, using the top() and push() methods, followed by the same procedure for lhs.

```
else if (isOperator(token))
{

    rhs = (double) stack.top();
    stack.pop();
    lhs =  (double) stack.top();
    stack.pop();
```

After that we have one if- and four else-if statements, each checking for one of the five operators. If one of the conditions is met, it would then be followed by a calculation using rhs, lhs and the corresponding operator. The result would then be assigned as value to the result boolean, which in turn gets pushed on top of the stack.

```
if (token == '+')
    {result = lhs + rhs;
    stack.push(result);
    }
else if (token == '-')
    {result = lhs - rhs;
    stack.push(result);
    }
else if (token == '*')
    {result = lhs * rhs;
    stack.push(result);
    }

else if (token == '/')
    {result =  lhs /rhs;
    stack.push(result);
    }
else if (token == '^')
    {result = (double) Math.pow(lhs, rhs);
    stack.push(result);
    }
```

After the else-if statement checking for operators, we added an else statement that would throw a new StackException with the message “Wrong input”. It only gets triggered if the input String contains something other than operators and numbers.

```
else {
    throw new StackException("Wrong Input");
}
```

For that, we created a class StackException that extends Exception, in which we have a constructor that gets a String as input, that it uses to call the constructor from the Exception class.

```
public class StackException extends Exception
{
    public StackException( String message )
    {
     super( message );
    }
 }
```

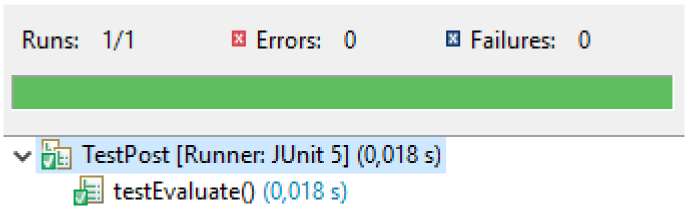
After the for-loop, we return the top of the stack.

```
return (double) stack.top();
```

For the test class, we chose to make a JUnit one, since it has been useful in previous labs. We called it TestPost and created in it a method called testEvaluate() that throws StackUnderflow and StackException. In the method we created an Object of type Postfix named test and a double called output.

We then used each postfix from the prelab as parameters to call the evaluate() method and assigned the result to output. We then used the assertEquals() method to compare output with the expected result.

```
class TestPost {

    @Test
    void testEvaluate() throws StackUnderflow, StackException {

        Postfix test = new Postfix();
        double output = 0;

        output = test.evaluate("12*3+");
        assertEquals(5,output);

        output = test.evaluate("123*+");
        assertEquals(7,output);

        output = test.evaluate("12+34^-");
        assertEquals(-78,output);

        output = test.evaluate("12^34*-");
        assertEquals(-11,output);

        output = test.evaluate("123*+45^-6+");
        assertEquals(-1011,output);

        output = test.evaluate("12+3*456-^+");
        assertEquals(9.25,output);

        output = test.evaluate("12+34/+5+678+*+");
        assertEquals(98.75,output);

        output = test.evaluate("91-2-32*-1-");
        assertEquals(-1,output);

    }
}
```

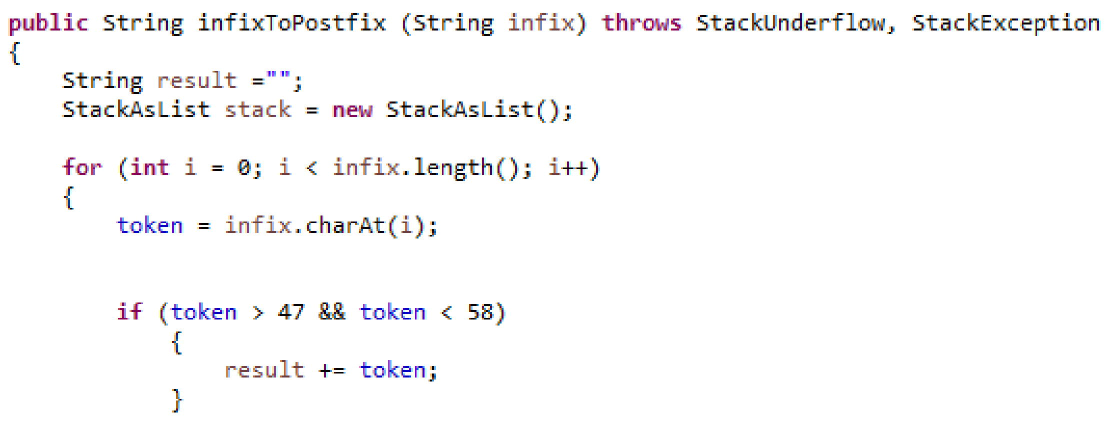


```
Runs: 1/1        Errors: 0        Failures: 0

TestPost [Runner: JUnit 5] (0,018 s)
    testEvaluate() (0,018 s)
```

4. **Now add another method to the** `Postfix.java` **class** `public String infixToPostfix (String ifx){...}`

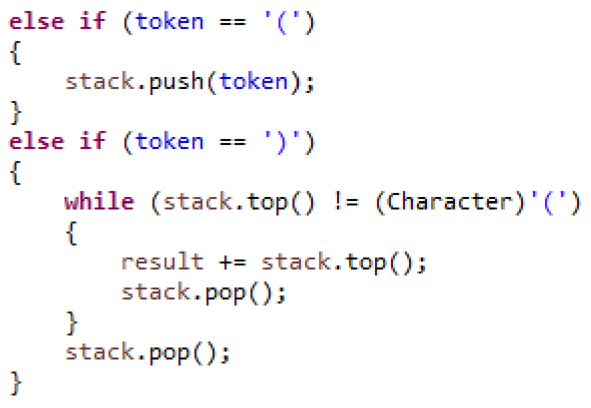
that converts an infix expression which is presented as a `String` to a `String` representing a postfix expression! Throw an exception if your input is not well-formed.

To complete this task, we started by creating a method called infixToPostfix, that gets a String infix as input and returns a String. After that we created a new StackAsList Object named stack, and a String called result. They were followed by a for-loop. In the loop, we first assigned the value of token to the char of the infix String at the current index i. We then have an if-statement that would check if the Ascii value of token is smaller than 58 and bigger than 47. If true, we would add token to result.



```
public String infixToPostfix (String infix) throws StackUnderflow, StackException
{
    String result ="";
    StackAsList stack = new StackAsList();

    for (int i = 0; i < infix.length(); i++)
    {
        token = infix.charAt(i);


        if (token > 47 && token < 58)
            {
                result += token;
            }
```

That if statement was followed by three else-if statements, the first one of which would check if token is equal to and open parenthesis, if true it would push token on the stack using the push() method. The second else-if would check if token is a closed parenthesis, if true it would be followed by a while-loop that would add the top of the stack to result and then pop it, until the top of the stack is equal to an open parenthesis. It would then leave the loop and pop the open parenthesis from the top of the stack.



```
else if (token == '(')
{
    stack.push(token);
}
else if (token == ')')
{
    while (stack.top() != (Character)'(')
    {
        result += stack.top();
        stack.pop();
    }
    stack.pop();
}
```

The last else-if statement would use the previously mentioned isOperator() method to check if token is an operator. If true, it would then lead to a while loop with a lot of conditions. First it would check if the stack is not empty using the isEmpty() method. Then it would check if the following is not true: the top of the stack is of lower precedence than token OR token is right associative and top is of equal precedence.

```
else if (isOperator(token))
{
    while (!stack.isEmpty() &&(!((precedence((char) stack.top()) < precedence(token) )||
            (token == '^' && precedence((char) stack.top()) == precedence(token) ))))
    {
```

To compare the precedences of the different operators, we created a new method called precedence(), it would take a char as input and return the corresponding precedence under the form of an int. If the input is not one of the listed operators, it would return -1.

```
public int precedence (char token)
{
    switch(token) {

    case '+':
    case '-':
        return 0;
    case '*':
    case '/':
        return 1;
    case '^':
        return 2;
    default:
        return -1;

    }
}
```

Since the '^' operator is the only one that is right associative, we didn't write a method for associativity and just checked if token is equal to '^'.

If the conditions from the while loop are met, we would add the top of the stack to result and then pop it. After the loop ended, we would then push token on top of the stack.

```
{
    result += stack.top();
    stack.pop();
}

stack.push(token);
```

After the last else-if statement, we added an else statement that would throw a StackException with "Wrong Input" as parameters. It only gets triggered if the input String contains something other than operators and numbers.

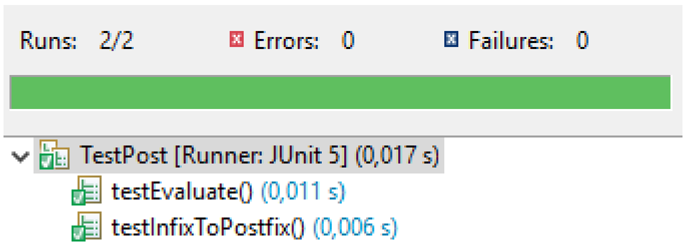
```
}
else {
    throw new StackException("Wrong Input");
}
```

After the for-loop, we created another while loop that would add the top of the stack to result and then pop it, until the stack is empty. And then finally, it would return result.

```java
while (!stack.isEmpty())
{
    result += stack.top();
    stack.pop();
}

return result;
```

To see if it was working as intended, we added another method to the JUnit test class and used the infixes from the prelab to check our infixToPostfix() method.

```java
@Test
void testInfixToPostfix() throws StackUnderflow, StackException {

    Postfix test = new Postfix();
    String output = "";

    output = test.infixToPostfix("1*2+3");
    assertEquals("12*3+",output);

    output = test.infixToPostfix("1+2*3");
    assertEquals("123*+",output);

    output = test.infixToPostfix("1+2-3^4");
    assertEquals("12+34^-",output);

    output = test.infixToPostfix("1^2-3*4");
    assertEquals("12^34*-",output);

    output = test.infixToPostfix("1+2*3-4^5+6");
    assertEquals("123*+45^-6+",output);

    output = test.infixToPostfix("(1+2)*3+(4^(5-6))");
    assertEquals("12+3*456-^+",output);

    output = test.infixToPostfix("1+2+3/4+5+6*(7+8)");
    assertEquals("12+34/+5+678+*+",output);

    output = test.infixToPostfix("9-1-2-3*2-1");
    assertEquals("91-2-32*-1-",output);

}
```


Runs: 2/2   Errors: 0   Failures: 0

TestPost [Runner: JUnit 5] (0,017 s)
  testEvaluate() (0,011 s)
  testInfixToPostfix() (0,006 s)

5. **Now add another method that reads an infix string from the console, evaluates the result and prints the result to the console.**

For this task, we created a new method called readInfix() that throws StackUnderflow, IOException and StackException. In the method we created a new BufferedReader called br, that has a new InputStreamReader as parameters, which in turn hsa the System input as parameter. We then added a String called input and assigned br.readLine() as its value. After which we called the infixToPostfix() method using input as parameter. We then called the evaluate() method with the result of the previously called method as parameter. And finally we output the result over the console using the System.out.println() method.

```
public void readInfix() throws StackUnderflow, IOException, StackException
{
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String input = br.readLine();
    System.out.println(evaluate(infixToPostfix(input)));

}
```

After some deliberation, we chose to create another method that would delete the white spaces contained in Strings, and implemented it in the beginning of the evaluate() and the infixToPostfix() methods.

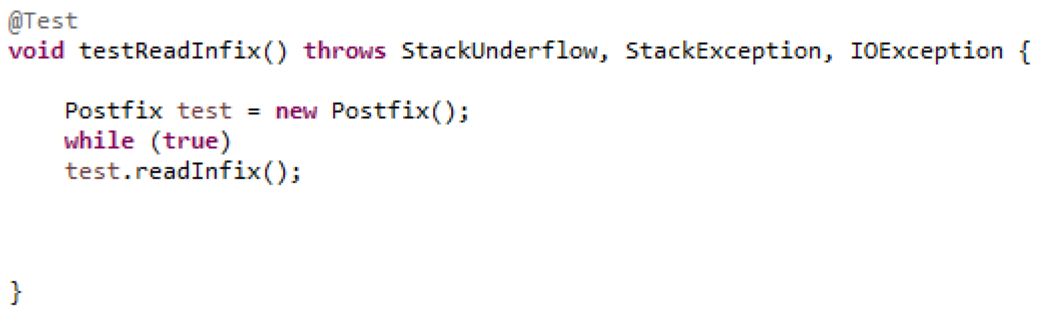
```
infix = Usefull_Methods.deleteSpaces(infix);
```

```
pfx = Usefull_Methods.deleteSpaces(pfx);
```

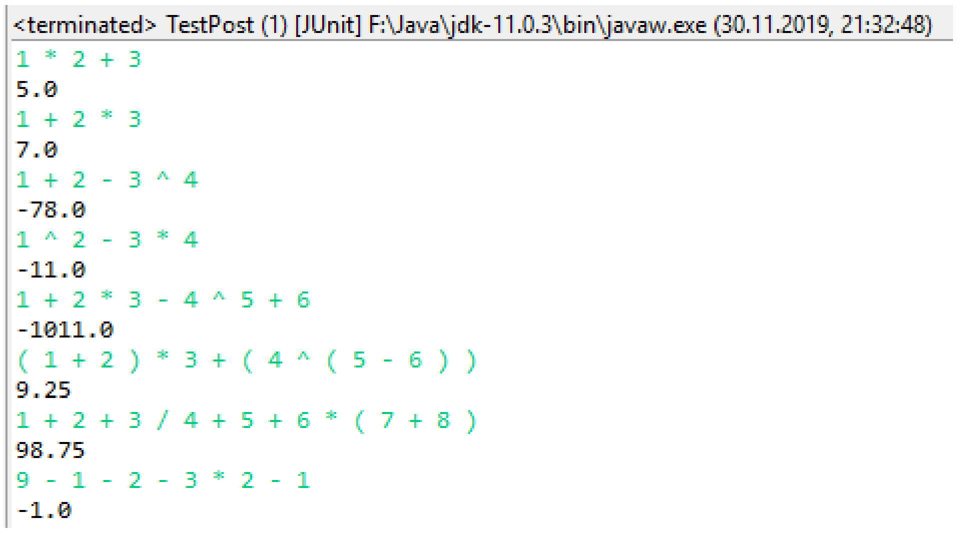
To do that, we first created a new class called Usefull_Methods and added a method deleteSpaces() that gets a String as input. In the method, we converted the inputted String to a char array using the toCharArray() method and assigned it to a newly created char array called cs[] as its value. After that we created a String called output and a for loop that would do a loop for every element in cs. In the loop we have an if-statement that checks if the current char is not a white space- If true, it gets added to output. After the loop, output gets returned.

```
public class Usefull_Methods {

    public static String deleteSpaces(String s) {

        char[] cs = s.toCharArray();
        String output = "";
        for(char e : cs) {
            if(e != ' ') output += e;
        }
        return output;
    }
}
```

To finish things off, we wrote another method in our test class, to check if the readInfix() and deleteSpaces() methods work as intended. Once again, we used the expressions from the prelab. In the method we created a new Postfix Object and an infinite while-loop in which we called the readInfix() method.



```java
@Test
void testReadInfix() throws StackUnderflow, StackException, IOException {

    Postfix test = new Postfix();
    while (true)
    test.readInfix();



}
```



```
<terminated> TestPost (1) [JUnit] F:\Java\jdk-11.0.3\bin\javaw.exe (30.11.2019, 21:32:48)
1 * 2 + 3
5.0
1 + 2 * 3
7.0
1 + 2 - 3 ^ 4
-78.0
1 ^ 2 - 3 * 4
-11.0
1 + 2 * 3 - 4 ^ 5 + 6
-1011.0
( 1 + 2 ) * 3 + ( 4 ^ ( 5 - 6 ) )
9.25
1 + 2 + 3 / 4 + 5 + 6 * ( 7 + 8 )
98.75
9 - 1 - 2 - 3 * 2 - 1
-1.0
```

After comparing the console ouput with our prelab answers, we can confirm that both the readInfix() and the deleteSpaces() methods work.

Reflections

BARTHOLOMÄUS

This week’s lab was quite alright, there weren’t any major issues that took hours of fixing, just a couple smaller ones but that's normal. This was probably due to the Postfix Evaluation Handout, since it corresponds to the pseudo code of what we needed to program. This week was also the first time since the Lab 0, that I’ve worked with someone I had worked with before, which made communication easier. I think that was important since it was the first lab where we had to code independently from our group partner, and a good communication made it easier to fix compatibility issues between our codes.

JURI

This week`s lab was interesting because it was the first time that both group members programmed their own little code simultaneously and at the end put them together, instead of switching every 20 mins. I think that it’s how we’ll work in the companies later. This isn’t very easy if the number of team members increase or the problems gets bigger. In order to make this work properly a good communication and organisation is needed. One problem we had were, that he used the top-method in order to add the data to a String. But my “top()”-method doesn’t give the data first. In our first version it returned the node. Even by saying the return type is “Node” in our interface we got in trouble in the communication. We simply solved this by changing the return type to Object and return the data of the Top-Node.

Source:

S1: https://people.f4.htw-berlin.de/~weberwu/info2/Handouts/Postfix-evaluation.html

Appendix:

```
public interface Stack<E> {
        public void push (E item);
        public void pop () throws StackUnderflow;
        public Object top () throws StackUnderflow;
        public boolean isEmpty ();
        public void Empty ();
        public void print();
        public String toString();

       }

public class StackAsList implements Stack {
       private Node myStack;
       private class Node {

              public Node(Object data, Node next) {
                     this.data = data;
                     this.next = next;
              }
              public Node() {
                     data = null;
                     next = null;
              }
              Object data;
              Node next;
              public String toString() {
                     return data.toString();
              }
       }
       public StackAsList() {
              myStack=null;
       }
       @Override
       public void push(Object item) {

              Node temp = new Node(item, myStack);
              myStack = temp;
       }

       @Override
       public void pop() throws StackUnderflow {
              if(!isEmpty())
              myStack = myStack.next;
       }

       @Override
       public Object top() throws StackUnderflow {
              // TODO Auto-generated method stub
              return myStack.data;
       }

       @Override
       public boolean isEmpty() {
              if(myStack == null) {
                     return true;
              }
              else
                     return false;
       }
       @Override
       public void Empty() {
              myStack = null;
       }

       @Override
       public void print() {
              System.out.println(this.toString());
       }
       @Override
       public String toString() {
              String S_myString = "";
              Node temp = new Node();
              temp = myStack;
              while (isEmpty()) {
                     if(temp.next == null) {
                             break;
                     }
                     temp = temp.next;
                     S_myString += temp.data + "\r\n";
              }
              return S_myString;
       }
}

public class Postfix {

       char token;

       public String infixToPostfix (String infix) throws StackUnderflow,
StackException
       {
              String result ="";
              StackAsList stack = new StackAsList();

              String multidigit = "";


              for (int i = 0; i < infix.length(); i++)
              {
                     token = infix.charAt(i);

                     if (isNumber(token)){

                             result += token;
                     }
                     else if (token == '(')
                     {
                             stack.push(token);
                     }
                     else if (token == ')')
                     {
                             while (stack.top() != (Character)'(')
                             {
                                     result += stack.top();
                                     stack.pop();
```

```
                          }
                          stack.pop();
                    }
                    else if (isOperator(token))

                    {
                          while (!stack.isEmpty() &&(!((precedence((char)
stack.top()) < precedence(token) )||
                                       (token == '^' && precedence((char)
stack.top()) == precedence(token) ))))
                          {
                                 result += stack.top();
                                 stack.pop();
                          }

                          stack.push(token);
                    }
                    else {
                          throw new StackException("Wrong Input");
                    }

             }

             while (!stack.isEmpty())
             {
                    result += stack.top();
                    stack.pop();
             }

             return result;
       }

       public boolean isOperator (char token) {

             if (token == '+' ||
                    token == '-' ||
                    token == '*' ||
                    token == '/' ||
                    token == '^')
                    return true;

             else
                    return false;
       }


       public int precedence (char token)
       {
             switch(token) {

             case '+':
             case '-':
                    return 0;
             case '*':
             case '/':
                    return 1;
             case '^':
                    return 2;
```

```
		default:
			return -1;

		}
	}

	public double evaluate (String pfx) throws StackUnderflow,
StackException {


		double rhs= 0;
		double lhs = 0;
		double tokenInt = 0;

		double result = 0;

		StackAsList stack = new StackAsList();

		pfx = Usefull_Methods.deleteSpaces(pfx);

		for (int i = 0; i <pfx.length(); i++)
		{
			token = pfx.charAt(i);

			if (token > 47 && token < 58)
			{
				tokenInt = Character.getNumericValue(token);
				stack.push(tokenInt);
			}

			else if (isOperator(token))
			{

				rhs = (double) stack.top();
				stack.pop();
				lhs =  (double) stack.top();
				stack.pop();

				if (token == '+')
					{result = lhs + rhs;
					stack.push(result);
					}
				else if (token == '-')
					{result = lhs - rhs;
					stack.push(result);
					}
				else if (token == '*')
					{result = lhs * rhs;
					stack.push(result);
					}

				else if (token == '/')
					{result =  lhs /rhs;
					stack.push(result);
					}
				else if (token == '^')
					{result = (double) Math.pow(lhs, rhs);
					stack.push(result);
```

```
                                }
                        }
                        else {
                                throw new StackException("Wrong Input");
                        }
                }

                return (double) stack.top();

        }

        public void readInfix() throws StackUnderflow, IOException,
StackException
        {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                String input = br.readLine();
                System.out.println(evaluate(infixToPostfix(input)));

        }
        public boolean isNumber(char value) {
        if(token > 47 && token < 58)
                        return true;
                else
                        return false;
     }

}



class TestPost {

        @Test
        void testEvaluate() throws StackUnderflow, StackException {

                Postfix test = new Postfix();
                double output = 0;

                output = test.evaluate("12*3+");
                assertEquals(5,output);

                output = test.evaluate("123*+");
                assertEquals(7,output);

                output = test.evaluate("12+34^-");
                assertEquals(-78,output);

                output = test.evaluate("12^34*-");
                assertEquals(-11,output);

                output = test.evaluate("123*+45^-6+");
                assertEquals(-1011,output);

                output = test.evaluate("12+3*456-^+");
                assertEquals(9.25,output);
```

```java
        output = test.evaluate("12+34/+5+678+*+");
        assertEquals(98.75,output);

        output = test.evaluate("91-2-32*-1-");
        assertEquals(-1,output);

    }

    @Test
    void testInfixToPostfix() throws StackUnderflow, StackException {

        Postfix test = new Postfix();
        String output = "";

        output = test.infixToPostfix("12+3");
        assertEquals("123+",output);

        output = test.infixToPostfix("12+34");
        assertEquals("12 34 +",output);

        output = test.infixToPostfix("1*2+3");
        assertEquals("12*3+",output);

        output = test.infixToPostfix("1+2*3");
        assertEquals("123*+",output);

        output = test.infixToPostfix("1+2-3^4");
        assertEquals("12+34^-",output);

        output = test.infixToPostfix("1^2-3*4");
        assertEquals("12^34*-",output);

        output = test.infixToPostfix("1+2*3-4^5+6");
        assertEquals("123*+45^-6+",output);

        output = test.infixToPostfix("(1+2)*3+(4^(5-6))");
        assertEquals("12+3*456-^+",output);

        output = test.infixToPostfix("1+2+3/4+5+6*(7+8)");
        assertEquals("12+34/+5+678+*+",output);

        output = test.infixToPostfix("9-1-2-3*2-1");
        assertEquals("91-2-32*-1-",output);

    }
}

package ubung6;

public class Usefull_Methods {

    public static String deleteSpaces(String s) {

        char[] cs = s.toCharArray();
        String output = "";
        for(char e : cs) {
            if(e != ' ') output += e;
            }
            return output;
        }

}

package ubung6;

public class StackException extends Exception
{
    public StackException( String message )
    {
        super( message );  // Calls Exception's constructor
    }

 }

package ubung6;

public class StackUnderflow extends Exception {

}
```