Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

B. Sc. Internationale Medieninformatik
Prof. Dr. Weber-Wulff

Group 1

Tuesday, 26.11.2019

Group members:

Timo Schmidt	571252	timo.schmidt@student.htw-berlin.de
Pavel Tsvyatkov	559632	pavel.tsvyatkov@student.htw-berlin.de

Post-Lab Report

Lab 6: Reverse Polish Notation

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.

After reading through all the exercises, we discussed together what methods we are going to need in our Stack interface. First we thought that we will need to be able to push things on top of the Stack, so we included a push() method which has a void return type. Since we want to be able to work with different data types, it takes something of type Object as parameter.

We wrote that the method can throw a StackOverflow exception, even if we thought that it won't be necessary. We will need to remove things from the stack so we also wrote a pop() method which also has a void return type. It takes no parameters, but now it was important to include that it can throw a StackUnderflow exception. We want to be able to see what's the top element without making any changes to the Stack, so we wrote the method top() which returns something of type Object. It takes no parameters and it can also throw a StackUnderflow exception.

We thought that it would be necessary to check if the Stack is empty so we included the isEmpty() method that takes no parameters and returns a boolean. We also wrote a method makeEmpty() which has a void return type and takes no parameters. Since we already knew what we will be dealing with in the following exercises we decided to include the toString method, which returns something of type String and doesn't take any parameters.

After writing down the methods our Stack interface looked like this:

```
1  public interface Stack {
2
3      void push (Object obj) throws StackOverflow;
4      void pop() throws StackUnderflow;
5      Object top() throws StackUnderflow;
6      boolean isEmpty();
7      void makeEmpty();
8      String toString();
9
10 }
```

Now that we have agreed on a Stack interface, we started working in parallel, with Timo working on exercise 2 and Pavel working on exercise 3.

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. (Timo)

First I created a new class called StackAsList. I implemented the interface by adding "implements Stack" after the class name and IntelliJ filled the class with all the method signatures. Since our StackAsList class should act like a LinkedList I also needed a class List and a class Node. I started by creating the Node class: It has two fields data and next. I wanted it to be generic so I made the data field to be of type Object. The next field is a Node since it's pointing to the next Node. I created two constructors although I later only used the latter one. This constructor takes an Object and Node as parameter and assigns them to the respective field.

```
public class Node {

    Object data;
    Node next;

    Node (Object obj) {
        data = obj;
    }

    Node (Object obj, Node next) {
        data = obj;
        this.next = next;
    }

}
```

Now I created a new class called List which I will use later for the StackAsList class. I first thought about the methods that are necessary and also took a look at the one from J. M. Bishop to make sure I didn't forget anything.

I went on and created three fields of type Node: head, curr and prev for pointing to the respective field. For the sake of completeness I also included a field of type int called size and created a getter method for this, although I didn't use it later on.

```
public class List {

    private Node head, curr, prev;
    private int size;

    public List() {
        head = null;
        curr = null;
        prev = null;
        size = 0;
    }
```

I then wrote the method bodies for addFirst(), remove(), isEmpty(), current(), reset(), eol() and succ(). I used our fields head, curr and prev to implement them properly. Last but not least I created another method print() which returns a String representing the Stack from Top to Bottom.

```
public String print() {
    String out = "TOP: ";
    for(this.reset(); !this.eol(); this.succ())
        out += curr.data + ((curr.next != null) ? ", " : "");
    return out + " :BOTTOM";
}
```

Now I was finally able to continue with the StackAsList() class. I created a new field list of my List class and a constructor which calls its method makeEmpty() (which creates a new list).

```
public class StackAsList implements Stack {

    private List list;

    // Constructor
    public StackAsList() {
        this.makeEmpty();
    }
```

The rest was simple. I used my List class and filled the bodies of our interface methods to make our stack work like a list. The first method push which takes an Object as parameter only needed one line with calling the addFirst() method from the list with the Object as parameter.

```
public void push(Object obj) throws StackOverflow {
    list.addFirst(obj);
}
```

This method could potentially throw a StackOverflow. IntelliJ suggested me to create new class for it and I filled its constructor with a System.err.println to indicate that the stack is full.

```
public class StackOverflow extends Exception {
    private static final long serialVersionUID = 1L;
    StackOverflow() { System.err.println("StackOverflow Exception. Stack is Full."); }
}
```

Now this is very unlikely since we have almost infinite storage nowadays so we don't really need this. It could be useful for smaller devices like smart watches though. I decided to leave it in for the sake of completeness.

The next two methods pop() and top() both could potentially throw a StackUnderflow on the other hand when the stack is empty. So I also created a new class for it and made its body similar to the Overflow method except the error message is different. (Stack is empty)

```
public class StackUnderflow extends Exception {
    private static final long serialVersionUID = 1L;
    StackUnderflow() {
        System.err.println("StackUnderflow Exception. Stack is Empty.");
    }
}
```

This is much more likely to happen so it definitely makes sense to include it in the StackAsList class. I added an if-statement to both of the methods to check if the list is empty using the isEmpty() method. This method simply returns the isEmpty() boolean method from my List.

```
public boolean isEmpty() {
    return list.isEmpty();
}
```

If the list is empty I throw a new StackUnderflow, otherwise I reset the pointer to top position. In the pop() method I then called the method remove() from my List whereas in top() I returned the object on top of our stack by asking the List for it's current Node.

```
public void pop() throws StackUnderflow {
    if (list.isEmpty())
        throw new StackUnderflow();
    else {
        list.reset();
        list.remove();
    }
}

public Object top() throws StackUnderflow {
    if (list.isEmpty())
        throw new StackUnderflow();
    else {
        list.reset();
        return list.current();
    }
}
```

Now the only thing left is the toString() method. I overwrote the method from the interface and called the print() method of my List:

```
public String toString(){
    return list.print();
}
```

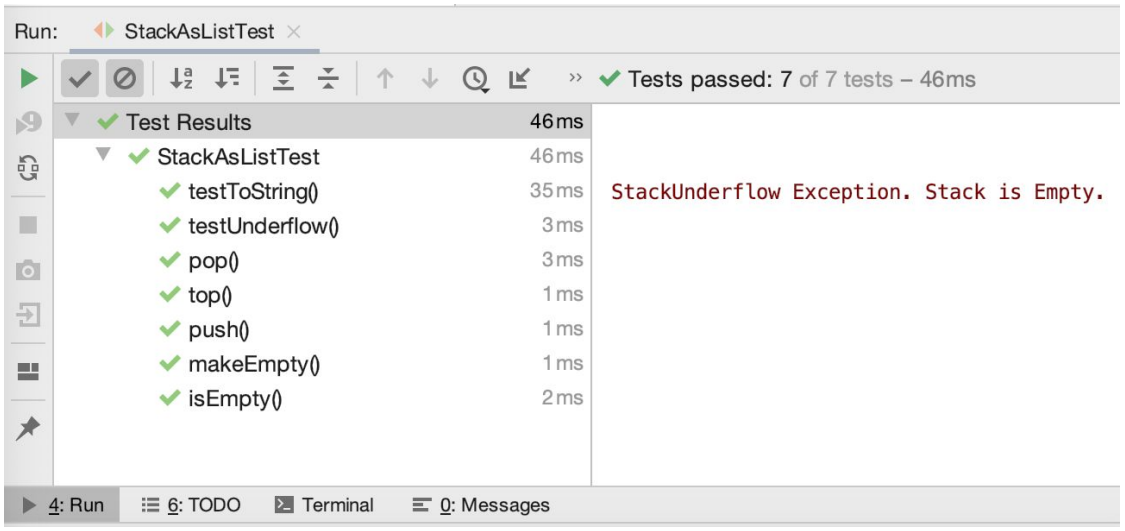
To test my StackAsList class I created a new TestClass called StackAsListTest. IntelliJ once again suggested me the method signatures from the StackAsList class and I filled them with a few use cases. I used the methods assertEquals and assertTrue/assertFalse from JUnit to make sure there are no mistakes in my code.

For example I tested the push() method by pushing different objects to the stack such as Integers, String and Characters. I then called the top() method to compare if the just added object is now on the stack.

```
class StackAsListTest {

    @Test
    void push() throws StackOverflow, StackUnderflow {
        StackAsList s = new StackAsList();

        // Push Integer
        s.push( obj: 123456789);
        assertEquals( expected: 123456789, s.top());

        // Push String
        s.push( obj: "Hello World!");
        assertEquals( expected: "Hello World!", s.top());

        // Push Character
        s.push( obj: 'C');
        assertEquals( expected: 'C', s.top());
    }
```

After finishing all the test cases I also wanted to test if an Exception gets thrown. Since I didn't know how to do that I searched the internet for a way to implement it. I learned about assertEquals which is also in JUnit (Source: https://howtodoinjava.com/junit5/expected-exception-example/). My first parameter is the StackUnderflow.class and the second parameter is an Executable. For the latter I could use a lambda expression to call my method pop() on an empty stack which should throw an Underflow exception.

All of my test cases were successful.



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. (Pavel)

To solve this exercise I first created a new class called Postfix. I then wrote the method *public int evaluate (String pfx)* and included both StackOverflow and StackUnderflow exceptions. After that I created a new StackAsList called s, the implementation of which Timo was working on at this time. Now I could use my notes from the prelab to write the method.
(Source: https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/)

```
7⊖    public int evaluate (String pfx) throws StackOverflow, StackUnderflow {
8
9         // Create a new stack
10        StackAsList s = new StackAsList();
```

After that I created a for loop which iterates through the postfix String. Since I need to access the individual characters of the postfix String one by one, I call the charAt(i) (character at position i) method on the postfix String and store it in a variable of type char which I called c.

```
11
12        // Iterate through the postfix String
13        for (int i = 0; i < pfx.length(); i++) {
14
15            // Scan characters
16            char c = pfx.charAt(i);
17
```

In case there are spaces in the postfix expression, there was a nice way to ignore and skip them.

```
17
18            // Skip spaces
19            if (c == ' ')
20                continue;
21
```

After that I included an if statement to check if the scanned character is a letter or a digit, for which there is a method called isLetterOrDigit() in Java. I also had to use the Character wrapper class instead of char. In case the scanned character is a letter or a digit, we push it into the Stack. There was a very interesting way to get an int out of a char by subtracting '0' from it, about which I didn't know up until now. It was very useful, because in this way if the character scanned is for example '5', a 5 is going to get pushed into the Stack.
(Source: https://programming.guide/java/converting-char-to-int.html)

```
21
22            // Push operands to stack
23            if (Character.isLetterOrDigit(c))
24                s.push(c - '0');
25
```

After that I included an else statement and I was referring to the algorithm to evaluate a postfix from the first exercise in the prelab. I took the top of the Stack by calling s.top(), which I also cast to an int, and assigned it to a new int variable that I called op1. After that I removed the element by calling s.pop(). Similar to the last step, I created a new int variable called op2, in which I assigned the top of the Stack and then popped it again.

```
26              else {
27                  int op1 = (int) s.top();
28                  s.pop();
29                  int op2 = (int) s.top();
30                  s.pop();
```

Inside the else statement I decided to use a switch statement to deal with different operators. The cases are almost the same. Only the case when the scanned character is a '^' and it's needed to raise to the power of a number, I used the Math.pow method. Since Math.pow returns a double and our method returns an int, I made sure to cast it to an int.

```
32                  switch(c) {
33                      case '+':
34                          s.push(op2 + op1);
35                          break;
36                      case '-':
37                          s.push(op2 - op1);
38                          break;
39                      case '*':
40                          s.push(op2 * op1);
41                          break;
42                      case '/':
43                          s.push(op2 / op1);
44                          break;
45                      case '^':
46                          s.push((int) Math.pow(op2, op1));
47                  }
```

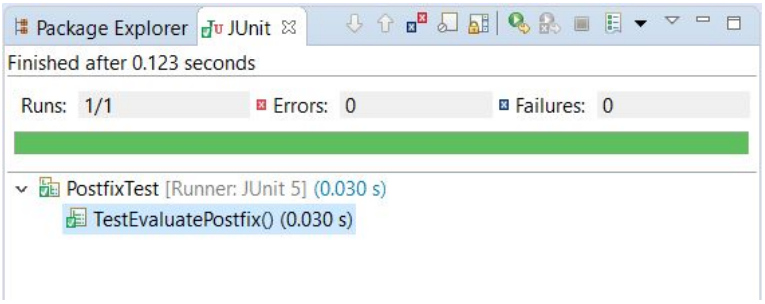
The very last thing to do in the evaluate method was to return the top of the Stack (again casted to an int) which is the determined value.

```
50
51          return (int) s.top();
52
```

Now that the method was finished I wanted to test if it was working properly. To do so I asked Timo for his code of exercise 2 and we exchanged our exercises. I created a test class called PostfixTest and a test case called TestEvaluatePostfix. I added assertions for all the prelab exercises and an additional one to test if our way of skipping spaces also works.

```java
class PostfixTest {

    @Test
    void TestEvaluatePostfix() throws StackOverflow, StackUnderflow {
        Postfix pfx = new Postfix();

        // Checking all finger exercises from prelab
        assertEquals(5, pfx.evaluate("12*3+"));
        assertEquals(7, pfx.evaluate("123*+"));
        assertEquals(-78, pfx.evaluate("12+34^-"));
        assertEquals(-11, pfx.evaluate("12^34*-"));
        assertEquals(-1011, pfx.evaluate("123*+45^-6+"));
        assertEquals(9, pfx.evaluate("12+3*456-^+"));
        assertEquals(98, pfx.evaluate("12+34/+5+678+*+"));
        assertEquals(-1, pfx.evaluate("91-2-32*-1-"));

        // Checking with spaces
        assertEquals(-1, pfx.evaluate("9 1 - 2 - 3 2 * - 1 -"));

    }
```

Our test method passed succesfully.



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 do this we used our notes from the prelab which we got from the website geeksforgeeks.org. We created the method signature and added both Underflow and Overflow since our StackAsList is using them. We once again made a new stack by creating a new instance of the StackAsList class called s. To keep track of our output we added a String result (which is empty at the beginning). Similar to the evaluation method we were using a for loop to iterate through our infix String. We also ignored all the spaces by adding a corresponding if statement.

```
public String infixToPostfix(String ifx) throws StackOverflow, StackUnderflow {

    // Create a new stack
    StackAsList s = new StackAsList();

    // Create a new String which represents the result
    String result = "";

    // Iterate through the infix String
    for (int i = 0; i < ifx.length(); i++) {

        // Scan characters
        char c = ifx.charAt(i);

        // Skip spaces
        if (c == ' ')
            continue;
```

Now we simply went through the steps from the prelab and added several if/ else if statements for the respective character. If the character is a letter/digit we concatenate the result String with it. If the character is a "(" (opening parenthesis) we add it to our stack by pushing it. If it's a closing parenthesis we pop all the characters from the stack until the open parenthesis is encountered. To pop them we are concatenating the result String with the character at the top (s.top) and delete it afterwards by using s.pop. With another if statement we popped the opening parenthesis too.

```
// If character is Letter/Digit, output it
else if (Character.isLetterOrDigit(c))
    result += c;

// If character is '(' push it to the stack
else if (c == '(')
    s.push(c);

// If character is ')' pop and output until '('
else if (c == ')') {
    while (!s.isEmpty() && (Character) s.top() != '(') {
        result += s.top();
        s.pop();
    }
    // then delete '(' from stack
    if (!s.isEmpty() && (Character) s.top() == '(')
        s.pop();
}
```

Now to check the precedence we made another method called getPrecedence() which takes a char as parameter. The method returns an Integer so we can compare the numbers later on. If the character is a '+' or '-' the method returns 1, for multiplication/division ('*' and '/') a 2 and for "to the power of" (^) we return a 3. Everything else leads to a -1.

```
private int getPrecedence(char c) {
    if (c == '+' || c == '-')
        return 1;
    else if (c == '*' || c == '/')
        return 2;
    else if (c == '^')
        return 3;
    else
        return -1;
}
```

We can use this method for our next step: checking if a character has a higher precedence than the one on the stack. If so we push the character to our stack. We needed to include !s.isEmpty() in the conditional since the stack could be empty and throw a StackUnderflow Exception.

```
// If character has higher precedence than the one in the stack
else if (s.isEmpty() || getPrecedence(c) > getPrecedence((Character) s.top())) {
    s.push(c);
}
```

For any other case (no letter, digit, parenthesis or a lower precedence than the one on the stack) we pop all the operators with equal/greater precendence from the stack after concatenating them with our result String. The character then gets pushed to the stack.

```
else {
    while (!s.isEmpty() && getPrecedence((Character) s.top()) >= getPrecedence(c) &&
            ((Character) s.top() != '(' || (Character) s.top() != ')')) {
        result += s.top();
        s.pop();
    }
    s.push(c);
}
```

Now we closed the for loop and wrote another while loop to check all characters that could be left on the stack. To do so we checked first if the stack is not empty and output the remaining characters as long as it's a valid character. To determine if a character is valid we once again used our getPrecedence method. As long as the character is not equal to -1 (which would mean we are dealing with a character other than +,-,*,/ or ^).. We also started at this point to think about the Exception that should get thrown when the input was not valid.

```
while (!s.isEmpty()) {
    if (getPrecedence((Character) s.top()) != -1) {
        result += s.top();
        s.pop();
    }
}
```

We then added a new String called exc at the top to include a message to tell the user that the input was not valid:.

```
String exc = "Input expression is not valid!";
```

We thought about when an expression is not valid and started with the closing parenthesis. We added an else-statement to return our error message. This would be the case for instance when there was no opening parenthesis before. We added the same else statement to our last while-loop to check the remaining stack characters.

```
else
    return exc;
```

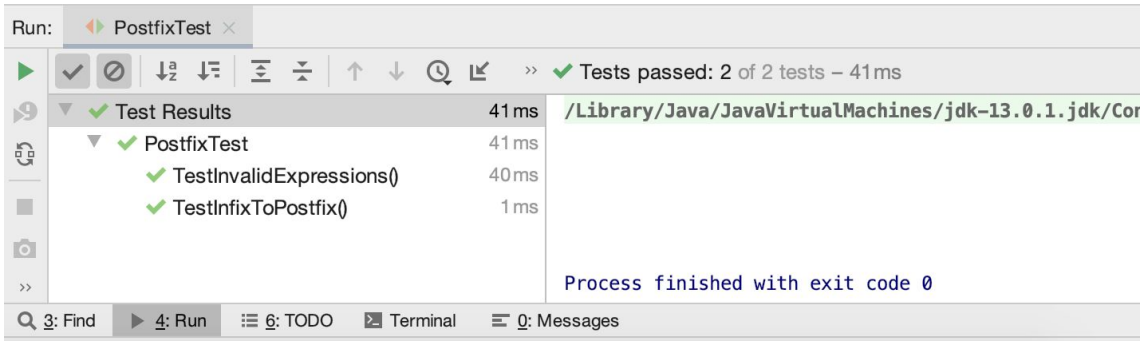
To test our new method we wrote a new test case for our PostFixTest class. We first tested all the finger exercises (this time with spaces between the characters) and used assertEquals with our expected output.

```
@Test
void TestInfixToPostfix() throws StackOverflow, StackUnderflow {
    Postfix pfx = new Postfix();

    // Checking all finger exercises from prelab
    assertEquals( expected: "12*3+", pfx.infixToPostfix( ifx: "1 * 2 + 3"));
    assertEquals( expected: "123*+", pfx.infixToPostfix( ifx: "1 + 2 * 3 "));
    assertEquals( expected: "12+34^-", pfx.infixToPostfix( ifx: "1 + 2 - 3 ^ 4"));
    assertEquals( expected: "12^34*-", pfx.infixToPostfix( ifx: "1 ^ 2 - 3 * 4 "));
    assertEquals( expected: "123*+45^-6+", pfx.infixToPostfix( ifx: "1 + 2 * 3 - 4 ^ 5 + 6"));
    assertEquals( expected: "12+3*456-^+", pfx.infixToPostfix( ifx: "( 1 + 2 ) * 3 + ( 4 ^ ( 5 - 6 ) )"));
    assertEquals( expected: "12+34/+5+678+*+", pfx.infixToPostfix( ifx: "1 + 2 + 3 / 4 + 5 + 6 * ( 7 + 8 )"));
    assertEquals( expected: "91-2-32*-1-", pfx.infixToPostfix( ifx: "9 - 1 - 2 - 3 * 2 - 1"));
}
```

We also wrote a second method to test invalid expressions. To do so we copied the String exc in the body of the test case and used it as our expected output. We tested both parentheses in the wrong place (or too many) and invalid characters.

```
@Test
void TestInvalidExpressions() throws StackOverflow, StackUnderflow {
    Postfix pfx = new Postfix();

    String exc = "Input expression is not valid!";

    // Wrong parentheses
    assertEquals(exc, pfx.infixToPostfix( ifx: "((5+3)+2"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "5+)3"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "5+(3"));
    assertEquals(exc, pfx.infixToPostfix( ifx: ")3+4("));
    assertEquals(exc, pfx.infixToPostfix( ifx: "3+4+5))"));

    // Invalid characters
    assertEquals(exc, pfx.infixToPostfix( ifx: "!"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "$"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "?"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "&"));
    assertEquals(exc, pfx.infixToPostfix( ifx: "#"));
}
```

All tests were successful.



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

We created a new method with return type void and called it readAndEvaluateInfix(). This method also could potentially throw an Under/Overflow Exception. To read a String from the console we created a new Scanner (as a field).

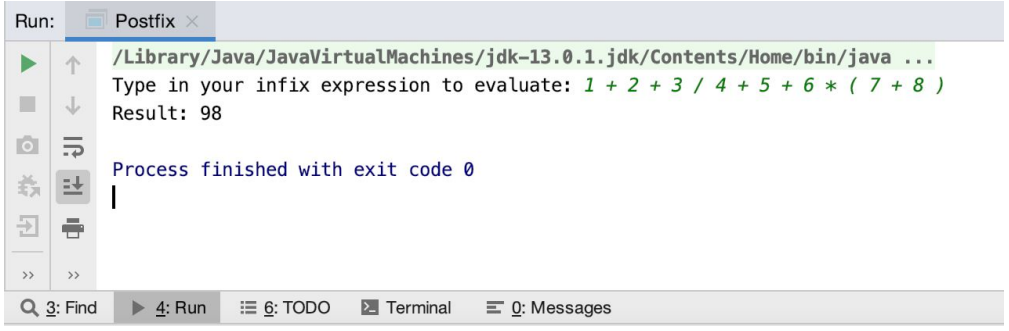
```
private Scanner scanner = new Scanner(System.in);
```

We then request the user to type in their infix String by adding a System.out.print (no "ln" to make sure the user directly types his input behind our message). To read the input we created a new String called ifx which is assigned to the method nextLine() of Scanner.

We then print our result with a System.out.println. Its content is the String "Result: " concatenated with evaluate(infixToPostfix(ifx)));. This nested method call first uses the infixToPostfix() method to transform the infix String into a postfix String and then the evaluate() method to calculate the result.

```
public void readAndEvaluateInfix() throws StackOverflow, StackUnderflow {

    System.out.print("Type in your infix expression to evaluate: ");
    String ifx = scanner.nextLine();

    System.out.println("Result: " + evaluate(infixToPostfix(ifx)));

}
```

To test it we also created a main method, a new instance of our Postfix class and called the method from there. We tested it with a infix from the prelab exercises and it worked.



What did we learn?

Timo Schmidt

In this week's exercises I could take a look behind the scenes by creating a StackAsList class without using one that is available by default from Java. When I first learned about RPN in the lecture I wasn't quite sure why I would need it. But during these exercises I've learned what's it good for and how to transform an infix expression to a postfix one. I also gained a better understanding of what a StackOverflow/Underflow is and how to throw an respective Exception.

I really liked that we could split up exercises this week. It was making things faster and I was able to focus on my part. After exchanging our code we read through the code of each other and checked if there are any significant things that we should change.

Pavel Tsvyatkov

After working on this exercise I learned about postfix notation and how to convert an infix to a postfix and evaluate it. I learned about the interesting way to get an int from a char by subtracting '0' and about the method isLetterOrDigit which is available in Java. While testing the methods I saw how the Overflow and Underflow exceptions work. I think this way of working together in parallel has advantages, because each of us can focus on a single task.

Appendix

Node

```
public class Node {

   Object data;
   Node next;

   Node (Object obj) {
       data = obj;
   }

   Node (Object obj, Node next) {
       data = obj;
       this.next = next;
   }

}
```

List

```
public class List {

   private Node head, curr, prev;
   private int size;

   public List() {
       head = null;
       curr = null;
       prev = null;
       size = 0;
   }

/*    public void add(Object obj) {
       if (head == null) {
           head = new Node(obj, null);
           curr = head;
       } else {
           if (curr == null){
               curr = new Node(obj, curr);
               prev.next = curr;
               prev = curr;
           } else {
               Node tmp = new Node (obj, curr.next);
               curr.next = tmp;
               prev      = curr;
               curr      = tmp;
           }
       }
       size += 1;
   }*/

   public void addFirst(Object obj) {
       Node tmp = new Node (obj, head);
       head = tmp;
       prev = null;
       curr = head;
   }
```

```
    public void remove() {
        if (this.isEmpty() || curr == null) {
            return;
        } else {
            if (prev == null) {
                head = curr.next;
                curr = head;
            } else {
                prev.next = curr.next;
                curr = curr.next;
            }
            size -= 1;
        }
    }

    public boolean isEmpty() {
        return head == null;
    }

    public Object current() {
        return curr.data;
    }

    public int getSize(){
        return this.size;
    }

    public void reset() {
        curr = head;
        prev = null;
    }

    public boolean eol() {
        return (curr == null);
    }

    public void succ() {
        curr = curr.next;
        if (prev == null)
            prev = head;
        else
            prev = prev.next;
    }

    public String print() {
        String out = "TOP: ";
        for(this.reset(); !this.eol(); this.succ())
            out += curr.data + ((curr.next != null) ? ", " : "");
        return out + " :BOTTOM";
    }

}
```

Stack (Interface)

```
public interface Stack {

   void push (Object obj) throws StackOverflow;
   void pop() throws StackUnderflow;
   Object top() throws StackUnderflow;
   boolean isEmpty();
   void makeEmpty();
   String toString();

}
```

StackAsList

```
public class StackAsList implements Stack {

   private List list;

   // Constructor
   public StackAsList() {
       this.makeEmpty();
   }

   public void push(Object obj) throws StackOverflow {
       list.addFirst(obj);
   }

   public void pop() throws StackUnderflow {
       if (list.isEmpty())
           throw new StackUnderflow();
       else {
           list.reset();
           list.remove();
       }
   }

   public Object top() throws StackUnderflow {
       if (list.isEmpty())
           throw new StackUnderflow();
       else {
           list.reset();
           return list.current();
       }
   }

   public boolean isEmpty() {
       return list.isEmpty();
   }

   public void makeEmpty() {
       list = new List();
   }

   public String toString(){
       return list.print();
   }

}
```

StackOverflow

```
public class StackOverflow extends Exception {
   private static final long serialVersionUID = 1L;
   StackOverflow() {
       System.err.println("StackOverflow Exception. Stack is Full.");
   }
}
```

StackUnderflow

```
public class StackUnderflow extends Exception {
   private static final long serialVersionUID = 1L;
   StackUnderflow() {
       System.err.println("StackUnderflow Exception. Stack is Empty.");
   }
}
```

StackAsListTest

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class StackAsListTest {

   @Test
   void push() throws StackOverflow, StackUnderflow {
       StackAsList s = new StackAsList();

       // Push Integer
       s.push(123456789);
       assertEquals(123456789, s.top());

       // Push String
       s.push("Hello World!");
       assertEquals("Hello World!", s.top());

       // Push Character
       s.push('C');
       assertEquals('C', s.top());
   }

   @Test
   void pop() throws StackUnderflow, StackOverflow {
       StackAsList s = new StackAsList();

       // Push and Pop
       s.makeEmpty();
       s.push(123);
       s.pop();
       assertTrue(s.isEmpty());

   }

   @Test
   void top() throws StackOverflow, StackUnderflow {
```

```
        StackAsList s = new StackAsList();
        s.push("Test");
        assertEquals("Test", s.top());

    }

    @Test
    void isEmpty() throws StackUnderflow, StackOverflow {
        StackAsList s = new StackAsList();

        // Check if empty at beginning
        assertTrue(s.isEmpty());

        // Push and pop and check again
        s.push(123);
        s.pop();
        assertTrue(s.isEmpty());

    }

    @Test
    void makeEmpty() throws StackOverflow {
        StackAsList s = new StackAsList();

        // Fill List
        s.push(1);
        s.push(2);
        s.push(3);

        // List is now expected to not be empty
        assertFalse(s.isEmpty());

        // Remove all objects by using makeEmpty
        s.makeEmpty();

        // List is now expected to be empty
        assertTrue(s.isEmpty());
    }

    @Test
    void testToString() throws StackOverflow {
        StackAsList s = new StackAsList();

        // Fill List
        s.push(1);
        s.push(2);
        s.push(3);

          // Expected output: "TOP: <pushed values separated by comma and space>
:BOTTOM"
        assertEquals("TOP: 3, 2, 1 :BOTTOM", s.toString());
    }

    @Test
    void testUnderflow() throws StackUnderflow {
        StackAsList s = new StackAsList();
        assertThrows(StackUnderflow.class, () -> s.pop());
    }

}
```

Postfix

```
import java.util.Scanner;

public class Postfix {

   private Scanner scanner = new Scanner(System.in);

   public int evaluate (String pfx) throws StackOverflow, StackUnderflow {

       // Create a new stack
       StackAsList s = new StackAsList();

       // Iterate through the postfix String
       for (int i = 0; i < pfx.length(); i++) {

           // Scan characters
           char c = pfx.charAt(i);

           // Skip spaces
           if (c == ' ')
               continue;

           // Push operands to stack
           if (Character.isLetterOrDigit(c))
               s.push(c - '0');

           else {
               int op1 = (int) s.top();
               s.pop();
               int op2 = (int) s.top();
               s.pop();

               switch(c) {
                   case '+':
                       s.push(op2 + op1);
                       break;
                   case '-':
                       s.push(op2 - op1);
                       break;
                   case '*':
                       s.push(op2 * op1);
                       break;
                   case '/':
                       s.push(op2 / op1);
                       break;
                   case '^':
                       s.push((int) Math.pow(op2, op1));
               }
           }
       }
       return (int) s.top();
   }

   public String infixToPostfix(String ifx) throws StackOverflow, StackUnderflow {

       // Create a new stack
       StackAsList s = new StackAsList();

       // Create a new String which represents the result
```

```
        String result = "";

        String exc = "Input expression is not valid!";

        // Iterate through the infix String
        for (int i = 0; i < ifx.length(); i++) {

            // Scan characters
            char c = ifx.charAt(i);

            // Skip spaces
            if (c == ' ')
                continue;

            // If character is Letter/Digit, output it
            else if (Character.isLetterOrDigit(c))
                result += c;

            // If character is '(' push it to the stack
            else if (c == '(')
                s.push(c);

            // If character is ')' pop and output until '('
            else if (c == ')') {
                while (!s.isEmpty() && (Character) s.top() != '(') {
                    result += s.top();
                    s.pop();
                }
                // pop '(' from stack
                if (!s.isEmpty() && (Character) s.top() == '(')
                    s.pop();
                else
                    return exc;

            }

            // If character has higher precedence than the one in the stack
              else if (s.isEmpty() || getPrecedence(c) > getPrecedence((Character)
s.top())) {
                s.push(c);
            }

            // If character is an operator
            // Pop all operators from stack with equal/greater precedence
            else {
                    while (!s.isEmpty() && getPrecedence((Character) s.top()) >=
getPrecedence(c) &&
                        ((Character) s.top() != '(' || (Character) s.top() != ')'))
{
                    result += s.top();
                    s.pop();
                }
                s.push(c);
            }
        }

        // Check the stack for any characters left
        while (!s.isEmpty()) {
            if (getPrecedence((Character) s.top()) != -1) {
                result += s.top();
```

```
                s.pop();
            }
            else
                return exc;
        }

        return result;
    }

      /** Method to get the precedence (as a value from 1 to 3) of the
corresponding operator
     * any other character returns a -1
     **/
    private int getPrecedence(char c) {
        if (c == '+' || c == '-')
            return 1;
        else if (c == '*' || c == '/')
            return 2;
        else if (c == '^')
            return 3;
        else
            return -1;
    }

    public static void main(String[] args) throws StackOverflow, StackUnderflow {
        Postfix pfx = new Postfix();
        pfx.readAndEvaluateInfix();
    }

    /**
     * Method to read an infix String from the console, evaluate it and print it
to the console.
     * @throws StackOverflow
     * @throws StackUnderflow
     */
    public void readAndEvaluateInfix() throws StackOverflow, StackUnderflow {

        System.out.print("Type in your infix expression to evaluate: ");
        String ifx = scanner.nextLine();

        System.out.println("Result: " + evaluate(infixToPostfix(ifx)));

    }
}
```

PostfixTest

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PostfixTest {

   @Test
   void TestEvaluatePostfix() throws StackOverflow, StackUnderflow {
       Postfix pfx = new Postfix();

       // Checking all finger exercises from prelab
       assertEquals(5, pfx.evaluate("12*3+"));
       assertEquals(7, pfx.evaluate("123*+"));
       assertEquals(-78, pfx.evaluate("12+34^-"));
       assertEquals(-11, pfx.evaluate("12^34*-"));
       assertEquals(-1011, pfx.evaluate("123*+45^-6+"));
       assertEquals(9, pfx.evaluate("12+3*456-^+"));
       assertEquals(98, pfx.evaluate("12+34/+5+678+*+"));
       assertEquals(-1, pfx.evaluate("91-2-32*-1-"));

       // Checking with spaces
       assertEquals(-1, pfx.evaluate("9 1 - 2 - 3 2 * - 1 -"));

   }

   @Test
   void TestInfixToPostfix() throws StackOverflow, StackUnderflow {
       Postfix pfx = new Postfix();

       // Checking all finger exercises from prelab
       assertEquals("12*3+", pfx.infixToPostfix("1 * 2 + 3"));
       assertEquals("123*+", pfx.infixToPostfix("1 + 2 * 3 "));
       assertEquals("12+34^-", pfx.infixToPostfix("1 + 2 - 3 ^ 4"));
       assertEquals("12^34*-", pfx.infixToPostfix("1 ^ 2 - 3 * 4 "));
       assertEquals("123*+45^-6+", pfx.infixToPostfix("1 + 2 * 3 - 4 ^ 5 + 6"));
        assertEquals("12+3*456-^+", pfx.infixToPostfix("( 1 + 2 ) * 3 + ( 4 ^ ( 5 - 6 )
)"));
        assertEquals("12+34/+5+678+*+", pfx.infixToPostfix("1 + 2 + 3 / 4 + 5 + 6 * ( 7 +
8 )"));
       assertEquals("91-2-32*-1-", pfx.infixToPostfix("9 - 1 - 2 - 3 * 2 - 1"));
   }

   @Test
   void TestInvalidExpressions() throws StackOverflow, StackUnderflow {
       Postfix pfx = new Postfix();

       String exc = "Input expression is not valid!";

       // Wrong parentheses
       assertEquals(exc, pfx.infixToPostfix("((5+3)+2"));
       assertEquals(exc, pfx.infixToPostfix("5+)3"));
       assertEquals(exc, pfx.infixToPostfix("5+(3"));
       assertEquals(exc, pfx.infixToPostfix(")3+4("));
       assertEquals(exc, pfx.infixToPostfix("3+4+5))"));

       // Invalid characters
       assertEquals(exc, pfx.infixToPostfix("!"));
       assertEquals(exc, pfx.infixToPostfix("$"));
       assertEquals(exc, pfx.infixToPostfix("?"));
       assertEquals(exc, pfx.infixToPostfix("&"));
       assertEquals(exc, pfx.infixToPostfix("#"));
   }
}
```

Prelab - Timo Schmidt

P1
Make sure that you understand postfix evaluation.

P2
Łukasiewicz 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?

*1 * 2 + 3 =*	*1 2 * 3 +*
*1 + 2 * 3 =*	*1 2 3 * +*
1 + 2 - 3 ^ 4 =	*1 2 + 3 4 ^ -*
*1 ^ 2 - 3 * 4 =*	*1 2 ^ 3 4 * -*
*1 + 2 * 3 - 4 ^ 5 + 6 =*	*1 2 3 * + 4 5 ^ - 6 +*
*(1 + 2) * 3 + (4 ^ (5 - 6)) =*	*1 2 + 3 * 4 5 6 - ^ +*
*1 + 2 + 3 / 4 + 5 + 6 * (7 + 8) =*	*1 2 + 3 4 / + 5 + 6 7 8 + * +*
*9 - 1 - 2 - 3 * 2 - 1 =*	*9 1 - 2 - 3 2 * - 1 -*

P3
For the infix expression a + b ^ c * d ^ e ^ f - g - h / (i + j), do the following:
Result: a b c ^ + d e f ^ ^ * g - h i j + / -

Show how to generate the corresponding postfix expression.

We go through the infix expression from left to right and work with a stack.

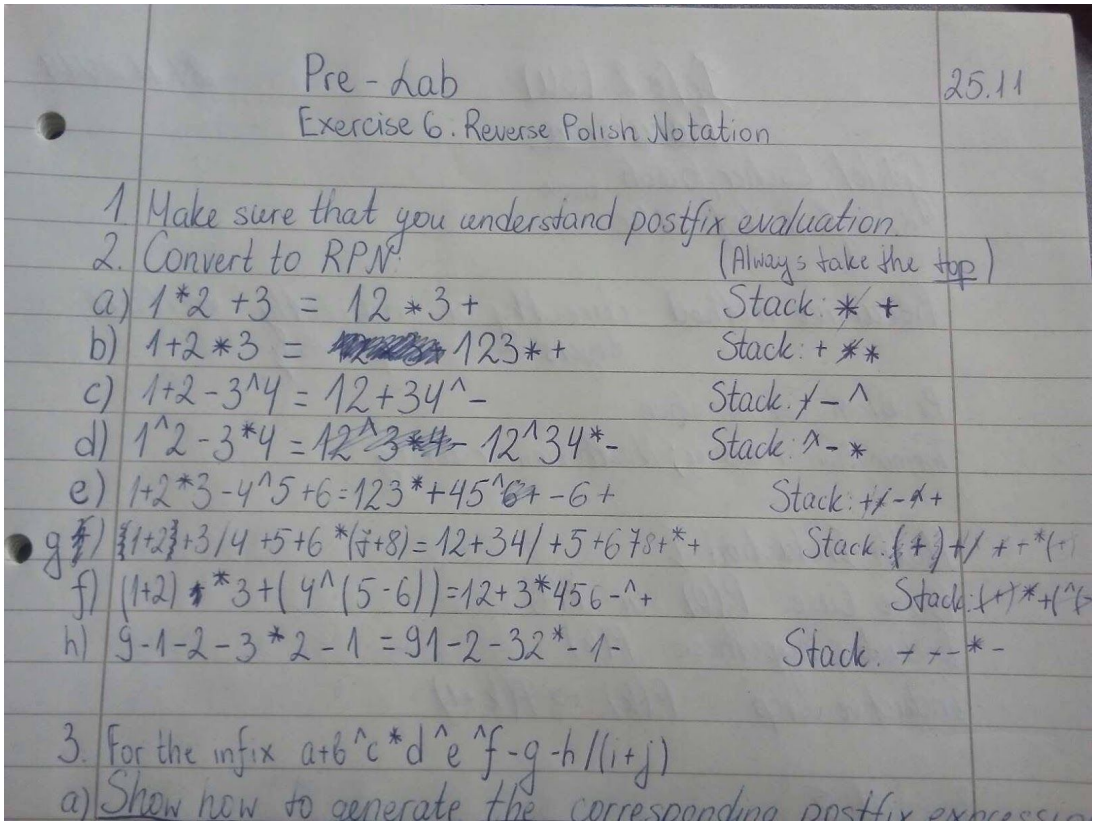
- If the next character is an operand -> output
- Else if the next character is an operator -> push on the stack
- if the next character is also an operator,
    - check if its precedence is greater than the one of the stack,
      if so, output it (also when „(„ or if empty)
    - if not output all operators from the stack which are greater/equal in precendence
      afterwards push the scanned operator on the stack
- if the next character is a „(„ -> push on stack
- if the next character is a „)“ -> pop the stack + output, until „(„ in encountered
  both parentheses get discarded and NOT OUTPUTTED

Show how to evaluate the resulting postfix expression.

- Create a stack to store operands
- if next character is a number -> push it on the stack
- if next character is a operator -> pop the operands of the operator from stack
  evaluate operator and push result back to stack
- if expression is finished -> Number on the stack = Result

Source: geeksforgeeks.org

Prelab - Pavel Tsvyatkov



Pre-Lab 25.11

Exercise 6. Reverse Polish Notation

1. Make sure that you understand postfix evaluation
2. Convert to RPN (Always take the top)

a) 1*2+3 = 12*3+ Stack: * +

b) 1+2*3 = 123*+ Stack: + **

c) 1+2-3^4 = 12+34^- Stack: + - ^

d) 1^2-3*4 = 12^34*- Stack: ^ - *

e) 1+2*3-4^5+6 = 123*+45^-6+ Stack: + - ^ +

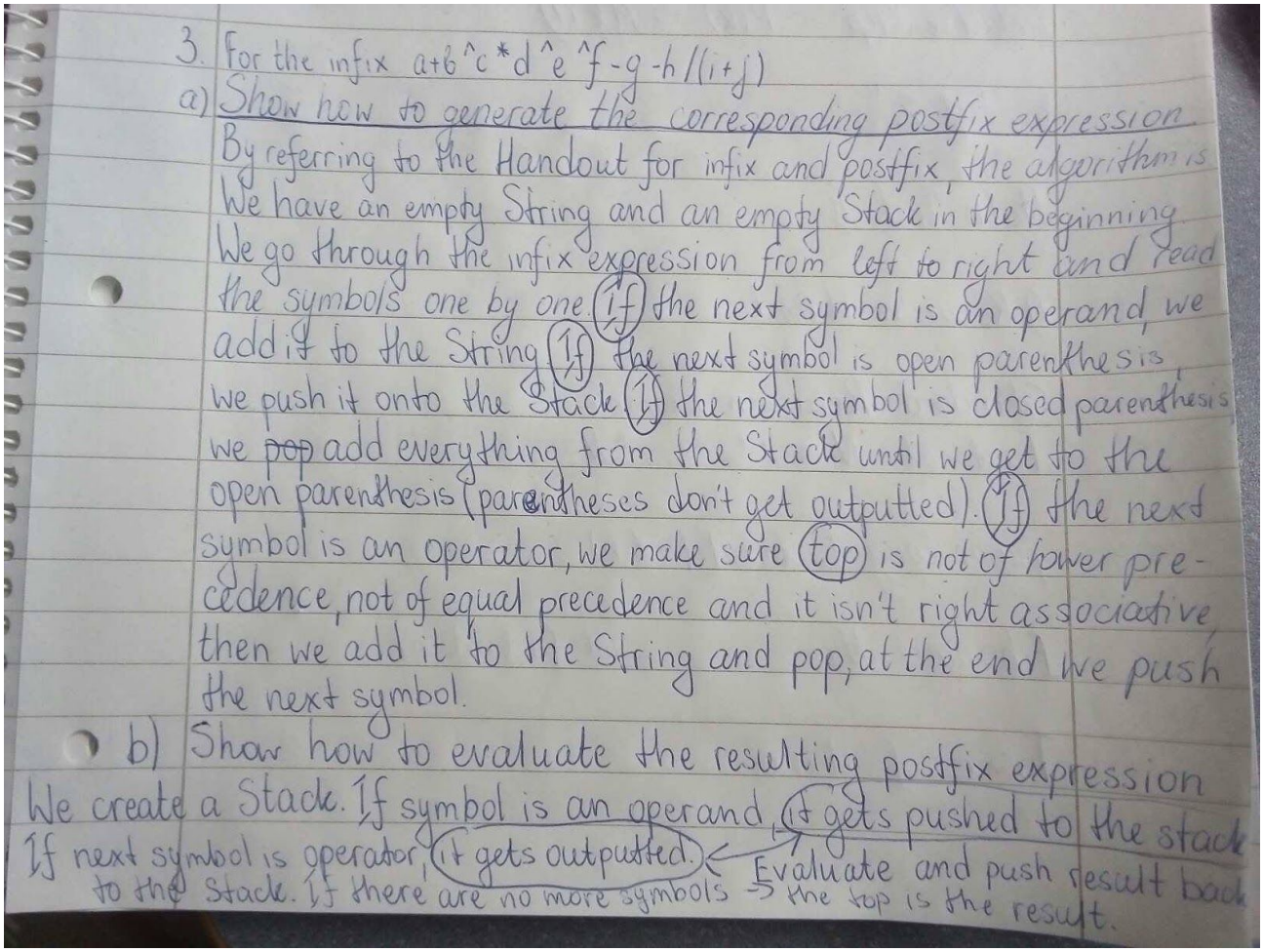
g) (1+2)+3/4+5+6*(7+8) = 12+34/+5+678+*+ Stack: (+) + / + + * (+

f) (1+2)*3+(4^(5-6)) = 12+3*456-^+ Stack: (+) * + (^ -

h) 9-1-2-3*2-1 = 91-2-32*-1- Stack: + + - - * -

3. For the infix a+b^c*d^e^f-g-h/(i+j)

a) Show how to generate the corresponding postfix expression



3. For the infix a+b^c*d^e^f-g-h/(i+j)

a) Show how to generate the corresponding postfix expression

By referring to the Handout for infix and postfix, the algorithm is: We have an empty String and an empty Stack in the beginning. We go through the infix expression from left to right and read the symbols one by one. (1) If the next symbol is an operand, we add it to the String. (2) If the next symbol is open parenthesis we push it onto the Stack. (3) If the next symbol is closed parenthesis we pop add everything from the Stack until we get to the open parenthesis (parentheses don't get outputted). (4) If the next symbol is an operator, we make sure (top) is not of lower precedence, not of equal precedence and it isn't right associative then we add it to the String and pop, at the end we push the next symbol.

b) Show how to evaluate the resulting postfix expression

We create a Stack. If symbol is an operand, it gets pushed to the stack. If next symbol is operator, it gets outputted. Evaluate and push result back to the Stack. If there are no more symbols -> the top is the result.