09.06.2021

HTW Internationale Medieninformatik
B7.1 Informatik 2 Gruppe 2
with Jonas Fichtmüller and
Cay Horstmann

Leander Kammermeier(577530)
Jonas Blome(579030)
Audrey Julie-Claire Gambia (566356)

Exercise 8: Reverse Polish Notation

PreLab:

2)

1 * 2 + 3	1 2 * 3 +	5
1 + 2 * 3	1 2 3 * +	7
1 + 2 - 3 ^ 4	1 2 + 3 4 ^ -	-78
1 ^ 2 - 3 * 4	1 2 ^ 3 4 * -	-11
1 + 2 * 3 - 4 ^ 5 + 6	1 2 3 * + 4 5 ^ - 6 +	-1011
(1 + 2) * 3 + (4 ^ (5 - 6))	1 2 + 3 * 4 5 6 - ^ +	9,25
1 + 2 + 3 / 4 + 5 + 6 * (7 + 8)	1 2 + 3 4 / + 5 + 6 7 8 + * +	98,75
9 - 1 - 2 - 3 * 2 - 1	9 1 - 2 - 3 2 * - 1 -	-1

3)

a)

The upper expression is split into single calculations and then added together as shown through the different colored markings.

a+b^c*d^e^f-g-h/(i+j)

ab+c^d*e^f^ghij+/--

b)

The evaluation of the upper expression happens through multiple steps shown in the pseudo code.

The operands are pushed into a stack and as soon as an operator is met, it is applied to / between the upper two operands in the stack. The result is then pushed into the stack again. After all of the operators are used up, the top of the stack remains as the result.

Lab Exercises:

1)

We agreed on using Eclipse and decided we would need the following methods:

Stack.pop(), which would return the popped element as a String

09.06.2021

Stack.push(T elem), which would take the element to be pushed as a parameter

Stack.top(), which would return the data of the element at the top of the stack

2)

After agreeing on the Stack interface, we created a new class for StackAsList. We first implemented the fields top and size to store the first node in the stack and the size of the stack. Then we had to create a new private class for nodes which we called Node. This method had the fields data and next that could store data of any type and another Node to link to. Using this we could implement push, pop and top. For push we used an element as a parameter that would be added to the stack. If the stack was empty, it would be put into the top field of the class and the next field of the new node would be set to null. If it wasn't empty, it would be added to the top and the former top would be put into the next field of the new node. Now, for the pop method we used three if/else statements. The first checks if the top.next field was not empty and if so, set the top field to top.next and return the data of the old top element. The second if else statement checks if the top is not null. If so it sets top to null and returns the data of the old top. The last else just throws a new exception that says "Stack Underflow". We don't need a stack overflow exception because the stack is handled by the java api and takes care of such errors itself. That's why we only implemented the stack underflow exception.

```
public class StackAsList<T> implements Stack<T> {
    private Node top;
    private int size = 0;

    @Override
    public T pop() {
        if (top.next != null) {
            T topString = top.data;
            top = top.next;
            size--;
            return topString;
        }
        else if (top != null) {
            T topString = top.data;
            top = null;
            size--;
            return topString;
        }
        else {
            throw new RuntimeException("Stack Underflow");
        }
    }

    public int getSize() {
        return size;
    }

    @Override
    public void push(T elem) {
        if (top == null) {
            Node newNode = new Node(elem, null);
            top = newNode;
            size++;
        }
        else {
            Node formerTop = top;
            Node newNode = new Node(elem, formerTop);
            top = newNode;
```

Then we implemented the top method that if the top is not null returns the data of top and in any other case returns null. We tested this method after also implementing the postfix class by simple using all the methods and seeing if there was no error. We also took care of every possible case that

could come up with the if statements in the methods of StackAsList. Now we overwrote the toString method. With a while loop we went through every node in the stack, until the last one, and added the data to a string that was initialized as “”. Finally we added a new method to check the size of the stack and that would be increased or decreased with every successful pop or push operation. Finally we changed the Stack and StackAsList class to be able to handle every kind of type as node data. We did this by using <T> and return this type when using pop, push and top.

```
public static void main(String[] args) {

}


@Override
public String toString() {
    String toString = "";
    Node currentNode = top;
    while(currentNode != null) {
        toString = toString + currentNode.data;
        currentNode = currentNode.next;
    }
    return null;
}

private class Node {
    T data;
    Node next;

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

@Override
public T top() {
    if (top != null) {
        return top.data;
    }
    else {
        return null;
    }
}
```

3)

After implementing the Postfix.java class, we added the wanted evaluate() Method with a String as its parameter. Following the pseudocode given, we translated the steps into real code. For every character in the string, we check if it is an operand or operator. If it is an operand, we push it. If the Character is an Operator, we get the top two numbers from the stack and apply the operator to them. We translate the Characters to an int by using the Character.getNumericValue() method. After checking every character, we print the top of the stack as it represents the answer of the calculation.

```
public void evaluate (String pfx) {
    StackAsList<Character> stack = new StackAsList<Character>();

    String r = "";
    char t = ' ';

    //while(pfx != "") {

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

        t = pfx.charAt(i);


        if(t == '1' || t == '2' || t == '3' || t == '4' || t == '5' || t == '6' ||
            stack.push(t);
        }

        if(t == '*' || t == '+' || t == '-' || t=='/') {
           int rhs = Character.getNumericValue(stack.top());
           stack.pop();

           int lhs = Character.getNumericValue(stack.top());
           stack.pop();


           int a = 0;
           if(t == '*') {
               a = lhs * rhs;
           }
           if(t == '+') {
               a = lhs + rhs;
           }
           if(t == '-') {
               a = lhs - rhs;
           }
           if(t == '/') {
               a = lhs / rhs;
           }
           stack.push((char) (a + '0'));

        }

    }
    System.out.println("result is:" + Character.getNumericValue(stack.top()));

//}

}
```

We tested this method with different single digit calculations. They all seemed to work, however our method is confined to single digit results, as we are working with Characters. If we would want to be able to calculate larger numbers, we would have to change the architecture of our method to handle Integers instead of chars, which would make it harder to pull apart the input string.

```
Postfix p = new Postfix();
p.evaluate("35+");
p.evaluate("91-2-32*-1-");
p.evaluate("12+34-");
p.evaluate("12*3+");
p.evaluate("335++");
p.evaluate("35+");
```

```
result is:8
result is:-1
result is:-1
result is:5
result is:-1
result is:8
```

4)

To write a method that can convert an infix expression into a postfix expression we tried to follow the pseudocode given on https://people.f4.htw-berlin.de/~weberwu/info2/Handouts/Postfix-evaluation.html First of all we created a new Stack called stack using static StackAsList<Character> stack = new StackAsList<Character>();
In our method public String InfixToPost(String infix) we first created a char variable t and a String variable s. With a while-loop we iterated through the given String infix and according to that t would be the character at the current index position of this String (t will be next token). If t is a number between 0 and 9 (an operand) String s will be updated to s + t. If t is an open parenthesis, it will be pushed. If t is a closed parenthesis, while the element on top of the stack is an open parenthesis the String s will be updated to s + stack.pop(). Otherwise we just call stack.pop(). If t is an operator, while the stack has a size larger of 0 and the element on top isn't an open parenthesis and the precedence of the top is lower than the precedence of t, String s will be updated to s + stack.top() and t will be pushed. Last but not least while the stack has a size larger than 0, String s will be updated to s + stack.top() and it we call stack.pop(). To check the precedence of a character we created another method: public static int precedence(char t). If t is a '+' or '-' -operator the precedence will be 1, if it is a '*', '/' or a '%' - operator the precedence will be 2 and for everything else the precedence will be 0.

```
public static int precidence(char t) {
    if(t == '+' || t == '-') {
        return 1;
    }
    if(t == '*' || t == '/' || t == '%') {
        return 2;
    }
    return 0;
}
```

To test out our method we rewrote the main method, that would now create a new InfixToPostfix element called p. For that element we then call the method InfixToPost with the given String "3+5".

```
public class InfixToPostfix {
    static StackAsList<Character> stack = new StackAsList<Character>();
    public static void main(String[] args) {
        InfixToPostfix p = new InfixToPostfix();
        p.InfixToPost("35+");
        }
    public String InfixToPost(String infix) {
```

Unfortunately, the method doesn't work correctly. When the method is executed a NullPointerException occurs.

```
Exception in thread "main" java.lang.NullPointerException
        at InfixToPostfix.InfixToPost(InfixToPostfix.java:31)
        at InfixToPostfix.main(InfixToPostfix.java:9)
```

We couldn't quite figure out the mistake so we decided to leave it as is to show our ideas and efforts.

09.06.2021

5)

Even though our infixToPostfix() Method wasn't working, we implemented the technicality of reading a string from the console via a Scanner as we had used in previous labs. By calling scanner.Next() we get the typed line as a String, with which we then would call the infixToPostfix() Method and call then the evaluate() method with its answer. At last the result of the evaluate() Method is printed.

Refections:

Leander:

I struggled with understanding the general concept of what we were actually trying to achieve throughout the lab, which got clearer, but not perfectly understandable, over time. Other than that working with given pseudo-code was really interesting and also bouncing ideas back and forth with Audrey and Jonas was fun. Even though we didn't manage to finish the implementation, I think we are on the right path.

Jonas:

In this session I learned to think about data structures in a very ground level way. I learned how to throw new exceptions and how to work with stacks properly. It was fun and easy to work with Leander and Audrey.

Audrey:

This Lab was quite enjoyable and getting to know all the possible mathematical expressions was really interesting. I also liked how we had to work together this time and that is was a little more like it would also be in reality. As to the programming, I found it a very helpful exercise to get to know stacks and learn how to implement and handle them in Java while having some orientation with the given pseudocode. Writing and using an interface was also new to me but I feel, due to this exercise I understand its way of working. Working in a group of three with Leander and Jonas was pleasant.

Time spend:

To complete this Lab we needed about three hours in addition to the actual lab.

Code:

Postfix Class:

```
import java.util.Scanner;


public class Postfix{


public static void main(String[] args) {
```

```
Postfix p = new Postfix();
p.evaluate("35+");
p.evaluate("91-2-32*-1-");
p.evaluate("12+34-");
p.evaluate("12*3+");
p.evaluate("335++");
p.evaluate("35+");
Scanner scanner = new Scanner( System.in );
String s = scanner.next();
   System.out.println("Input string is:" + s);
System.out.println(p.infixToPostfix(s));

}

public String infixToPostfix (String ifx) {
 StackAsList<Character> stack = new StackAsList<Character>();
 String r = "";
 char t = ' ';
 int i = 0;
 while(i < ifx.length()) {
 t = ifx.charAt(i);
 if(t == '0' || t == '1' || t == '2' || t == '3' || t == '4' || t == '5' || t == '6' || t == '7' || t == '8' || t == '9') {
 r = r + t;
 }
 else if(t == '(') {
 stack.push(t);
 }
 else if(t == ')') {
 while(stack.top() != '(') {
 r = r + stack.top();
 stack.pop();
 }
 stack.pop();
 }
```

```
 else if(t == '-' || t == '+' || t == '/' || t == '*' || t == '%' ) {

 while(precedence(stack.top()) < precedence(t)) {
 r = r + stack.top();
 stack.pop();
 if(stack.top() == null) {
 break;
 }
 }

 stack.push(t);
 }
 while(stack.top() != null) {
 r = r + stack.top();
 stack.pop();
 }
 i++;
 }
    return r;
}

public int precedence(char t) {

if(t == '+' || t == '-') {
return 1;
}
if(t == '*' || t == '/' || t == '%') {
return 2;
}
return 0;
}



public void evaluate (String pfx) {
   StackAsList<Character> stack = new StackAsList<Character>();
```

09.06.2021

```
String r = "";
char t = ' ';

//while(pfx != "") {

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

t = pfx.charAt(i);



if(t == '1' || t == '2' || t == '3' || t == '4' || t == '5' || t == '6' || t == '7' || t == '8' || t == '9') {
stack.push(t);
}

if(t == '*' || t == '+' || t == '-' || t=='/') {
  int rhs = Character.getNumericValue(stack.top());
  stack.pop();

  int lhs = Character.getNumericValue(stack.top());
  stack.pop();



  int a = 0;
  if(t == '*') {
  a = lhs * rhs;
  }
  if(t == '+') {
  a = lhs + rhs;
  }
  if(t == '-') {
  a = lhs - rhs;
  }
  if(t == '/') {
  a = lhs / rhs;
```

```
  }
  stack.push((char) (a + '0'));

}

}
System.out.println("result is:" + Character.getNumericValue(stack.top()));

//}

}
}
```

Stack Class

```
public interface Stack<T> {

abstract T pop();

abstract T top();

   abstract void push(T elem);



}
```

StackAsList Class

```
public class StackAsList<T> implements Stack<T> {
private Node top;
private int size;

@Override
public T pop() {
size++;
if (top.next != null) {
```

```
T topString = top.data;
top = top.next;
return topString;
}
else if (top != null) {
T topString = top.data;
top = null;
return topString;
}
else {
return null;
}
}

public int getSize() {
   return size;
}

@Override
public void push(T elem) {
size++;
if (top == null) {
Node newNode = new Node(elem, null);
top = newNode;
}
else {
Node formerTop = top;
Node newNode = new Node(elem, formerTop);
top = newNode;
}
}

public static void main(String[] args) {
```

```
}



@Override
public String toString() {
String toString = "";
Node currentNode = top;
while(currentNode != null) {
toString = toString + currentNode.data;
currentNode = currentNode.next;
}
return null;
}

private class Node {
T data;
Node next;

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

@Override
public T top() {
if (top != null) {
return top.data;
}
else {
return null;
}
}
}
```

09.06.2021