Lab 1: Stacks

F28SG – Introduction to Data Structures and Algorithms (6 marks)

The project is organized as follows:

- The src directory contains all the source files
    - Stack.java is the implementation of a stack from the lecture
    - Reverse.java is the file you should work with
- The test directory contains the unit tests for the project.

Plagiarism policy

This lab is not group work, and you are assessed individually.

Therefore, the work you submit for this lab **must be entirely your own**. You must not share your code solution with other students, and you must not copy code solutions from others. The university plagiarism policy is clear:

> *"Plagiarism involves the act of taking the ideas, writings or inventions of another person and using these as if they were one's own, whether intentionally or not."*
> ***Definition 2.1.***

The disciplinary action for plagiarism is an award of an F grade (fail) for the course. Serious instances of plagiarism will result in Compulsory Withdrawal from the university.

Q1) Write unit tests for the Push and Pop operations (2 points)

First write unit tests for the **push** and **pop** operations in the `StackTest.java` class. To start, an empty stack `st` with space for 2 elements has been provided, together with a test method for each operator.

You should write unit tests that:

1. Covers normal behaviour of push() and pop(), for example:
    a. the size increases/decreases as expected
    b. the top element is correct after a push operation
2. Pushes to a full stack throws an exception
3. Pops from an empty stack throws an exception

You need to create separate test methods for 2 and 3, or you can have all tests in the methods provided. Note that for 2 and 3 you need to check that the `StackException` is raised when attempting to pop from an empty stack or pushing to a full stack (see lecture 2 for how to test for exceptions).

Run JUnit. These tests should all succeed!

Q2) Reverse an Array Using a Stack (1 point)

In `Reverse.java` there is a method

```
public static void reverse(String[] arr){
    // your code here
}
```

that should reverse the given array `arr` using only a stack. In `ReverseTest.java` there is a test method for this method called `testReverse()`. First read the test and make sure you understand what it does. Try to run the test (it should fail). Then you should implement the `reverse` method of `Reverse.java`. Note that for checking that two arrays are equal, you should use the `assertArrayEquals` assertion rather than the `assertEquals` assertion.

When you have completed the implementation then re-run the test again and it should now succeed. Note that there will be other tests relating to part 3 that fails.

State the Big-Oh complexity as a comment above the `reverse` method.

Hint! *If you push all elements of `arr` (in the correct order) to a stack, and then pop each element and store them in the correct place in the array, then they will come out in the reversed order.*

Q3) A Polish Notation Calculator [challenging] (2 points)

Arithmetic operations are normally written in an infix form, for example: 5 + 2 or 3 - 5 + 2. However, to avoid ambiguity, we something need to use bracketing. E.g. how should we read 3 - 5 + 2?

1. *3 - 5 + 2 = (3 - 5) + 2 = -2 + 2* ***= 0***
1. *3 - 5 + 2 = 3 - (5 + 2) = 3 -* ***7*** *= -4*

To avoid such bracketing, we can write this in a prefix way (also called *Polish notation*) where we write the operator first. Here you write

<operator> <left side> <right side>

and you can remove all brackets. For example:

1. 5 + 2 → + 5 2
1. (3 - 5) + 2 → + (3 - 5) 2 → + (- 3 5) 2 → + - 3 5 2
1. 3 - (5 + 2) → - 3 (5 + 2) → - 3 (+ 5 2) → - 3 + 5 2

Your task is to use a `Stack` to implement a calculator using polish notation. Empty methods (listed below) for this task are provided in `Calculator.java`, and you should use these.

You can assume that the input is in Polish notation. This should return an integer, which is the result of the calculation:

```
public static int calculate(String[] cmds){
  Stack stack = new Stack();
  <your code>
}
```

such that for example

```
String[] cmds = {"+","-","3","5","2"};

calculate(cmds);
```

returns 0. The method should at least support **addition** and **subtraction** (division and multiplication are optional). The method should then use a stack to perform the computation of `cmds`.

The simplest way to implement `calculate` is to first reverse the list (you can use the method from Part 2 for this). This will turn the list into what is known as *reversed polish notation.* It may be useful to sketch on paper how to do this by stepping throw the example shown above (reversed, i.e. `{"2","5","3","-","+"})`.

To start with, note that you need to iterate through the reversed list, and for each step separate between arguments that are numbers and arguments that are operators (`"+","-","/","*"`).

You also need to be able to convert a string to a number, and the following method could be useful:

```
public static int convert(String s){ <your code> }
```

Moreover, you will need a method to check if a string is an operator (`"+","-","/","*"`) or a number (you can assume that they will always be one or the other). For example, a method that returns true if the given string is a number or false otherwise:

```
public static boolean isNumber(String s){ <your code> }
```

`Hint! Integer.parseInt` method converts a string to an integer and throws a `NumberFormatException` exception if this is not possible. Meaning, if an exception is thrown then you know it is not an integer, so it has to be an operator, and if it is not thrown then you know that it is an integer.

Finally, you will need to be able to perform arithmetic. For this, you can implement a method:

```
public static int applyOp(String fst, String op, String snd){ <your code> }
```

such that for example `applyOp`("5","+","2") returns 7.

The `calculate` method will need to use these operations, and for each iteration you either need

- to push the element to the stack,
- or pop and apply the operations above.

This depends on the type of argument (operator or number).

Unit tests are provided in `CalculatorTest.java` for each of these methods to help you in the development.

Hint! The idea here is that for each iteration, if the element you are working on is a number then you push the number to `stack`. If it is an operator, then you pop the first two elements of `stack` and apply the `applyOp` method with the operator and the two elements popped (in this case the first two elements should always be numbers). Be sure that you get the left- and right-hand side correct when you call the `applyOp` method. You can find out what the operator character is with the equals method on Strings, e.g. `op.equals("+")` which returns `true` or `false`. At the end of the iteration, the stack should contain one element, which you need to convert to an integer and return.

Q4) Code Quality (1 point)

Code quality is vitally important for so many reasons. Not least, for readability and maintainability, not just for yourself but for others too since in industry, software engineering is almost always a group exercise. Real world software engineering is mostly about refactoring and testing code, rather than writing new code (more code means higher maintenance costs!).

An additional mark is awarded if your code is deemed to be of high quality:

- **Big-Oh**
  - Has Big-Oh complexity been documented for all methods where requested?
- **Code simplicity**
  - Is the code as short as it can be?
  - Is the Big-Oh complexity as small as it can possibly be?
  - Is the control flow (loops, while statements, if statements, etc.) simple to follow?
- **Documentation**
  - Are you using comments ***inside your implementation methods and test methods*** to provide an algorithmic commentary about what the code is doing.

  Here is a **good** comment:

  ```
  // creates unidirectional connection from the predecessor node to the new node
  prevNode.nextNode = newNode;
  ```

  Here is a **less useful** comment:

  ```
  // sets preNode.nextNode to newNode
  prevNode.nextNode = newNode;
  ```

  - Is Javadoc syntax used for documenting the code? ***Hint!*** In Eclipse move your cursor to the text definition of a field, method or a class, then use the following keyboard shortcut: ***Alt + Shift + j*** , and then fill in the generated Javadoc template. If you are using MacOS,the shortcut is: ***⌘ + Alt + j***
- **Conventions**
  - Are sensible Java code conventions used, e.g. for code indentation, declarations, statements, etc? See Sections 4, 5, 6, 7 and 9 of *Java Code Conventions*. ***Hint!*** *Use the keyboard shortcut in Eclipse:* ***Control + Shift + f*** , or on MacOS: ***⌘ + Shift + f***

An Additional Challenge (optional)

An alternative way is to use the polish notation directly. As in the previous part, you will need to iterate through the list and for each iteration the behavior depends on the type of string you are working on (operator or number). However, in this case it also depends on what is currently on top of the stack. You should extend the `Lab2` class with another method `calculatePolish` which computes the list directly without reversing the list first. Write unit tests for this method.

Hint ! In this case, for each element in the iteration. If the element is a number and the

first element of `stack` is a number then:

```
first <- pop stack
operator <- pop stack
```

Followed by pushing the result of `applyOp(first,operator,element)`. If this is not the case then push the element on the stack. If at the end of the iteration, there are more than one element on the stack, then you need to repeatedly pop the first three elements, apply the `applyOp` operator, and push the result on the stack until there is only one element left which you return.