Homework 1: Having fun with stacks and queues

Task 1: Build a stack using queues

Stack and queue data structures are very versatile and can be used in a variety of situations. But did you know that you can also **build one using the other**?

In this task, you will implement a working stack **using 2 queues**. The algorithms for pop() and push() are fairly simple. Let us say you are keeping track of 2 queues inside your stack class: **q1** and **q2**.
To **add** a new element to the stack → push():

- Add the element to q2.
- One by one, remove all elements from q1 and add them to q2.
- Swap the queues q1 and q2.

To **remove** an element from the stack → pop():

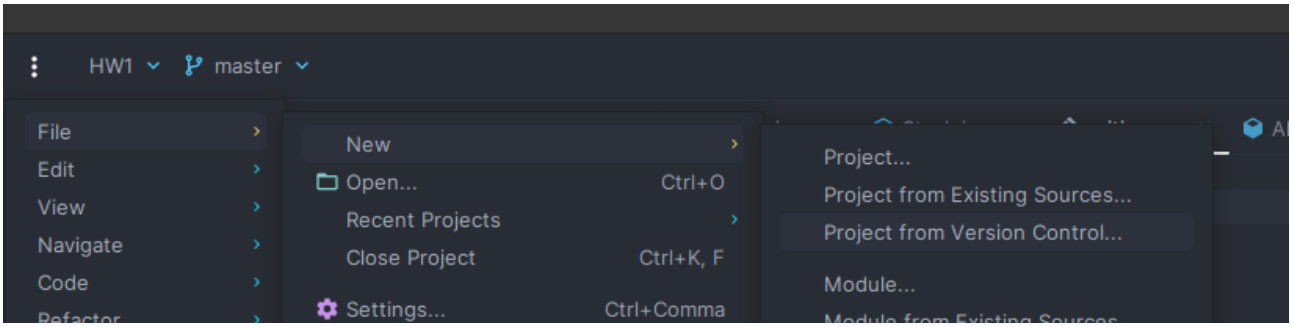
- Remove an element from q1 and return it.

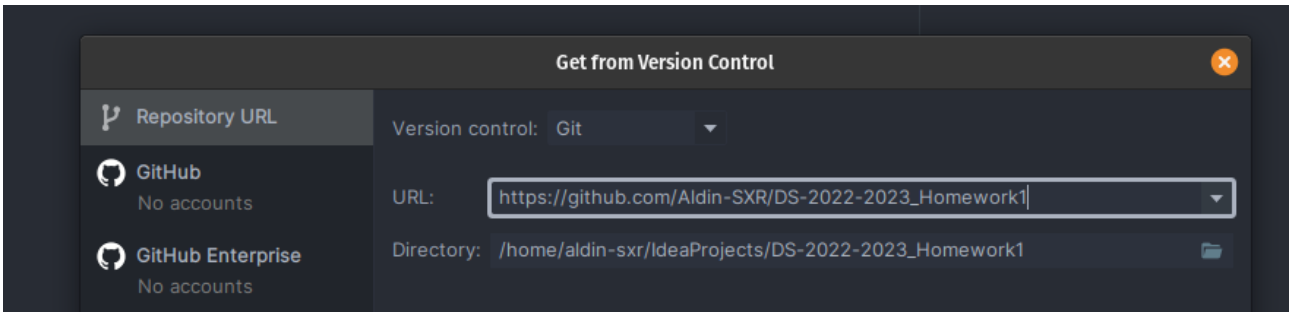
Using the logic above, implement a **Stack** class which supports the following methods:

- **public void push(Item data)** → add an element of *any type* to the stack
- **public Item pop()** → remove and return the top-most element from the stack
- **public Item peek()** → return the top-most element from the stack, *without* removing it.
- **public int size()** → return the number of elements on the stack
- **public boolean isEmpty()** → return whether the stack is empty or not

To help you get started, with the homework, you have the **homework skeleton** code:
https://github.com/Aldin-SXR/DS-2022-2023_Homework1

To import this existing code, you can use Intellij IDEA's **File > New > Project From Version Control…**, and provide the URL.





Alternatively, you can download the code, and use File > New > Project From Existing Sources…, or just copy and paste all code (although not recommended 🙂).

When it comes to Task 1, there is a **Stack.java** file with method signatures provided, in *src/main/java/task1*. You also have a **Queue.java** file which contains an already implemented Queue.

You must not:

- remove any of the methods in Stack.java, rename them nor change their *signatures*
- modify the Queue.java class in *any way* (no removing or adding methods to it).

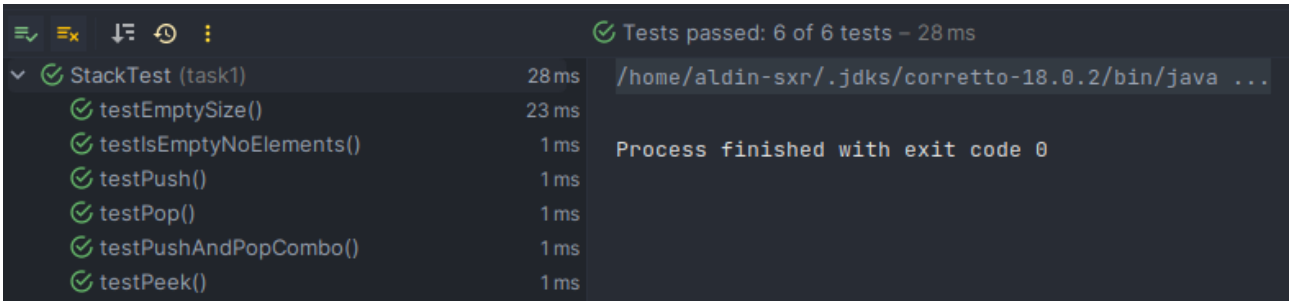
You should:

- implement the missing method bodies for the required functionalities, and make sure they return proper output (if any).

You may:

- add additional helper methods / variables to the Stack class *if you need them* for the solution (on the condition that you do not modify any existing method signatures, and keep the main logic in those existing methods).

In the **test** package, there is also a **StackTest.java** test class which you can run to check if your stack is working properly. All tests should be **passed** if it is.



Task 2: Arithmetics using stacks

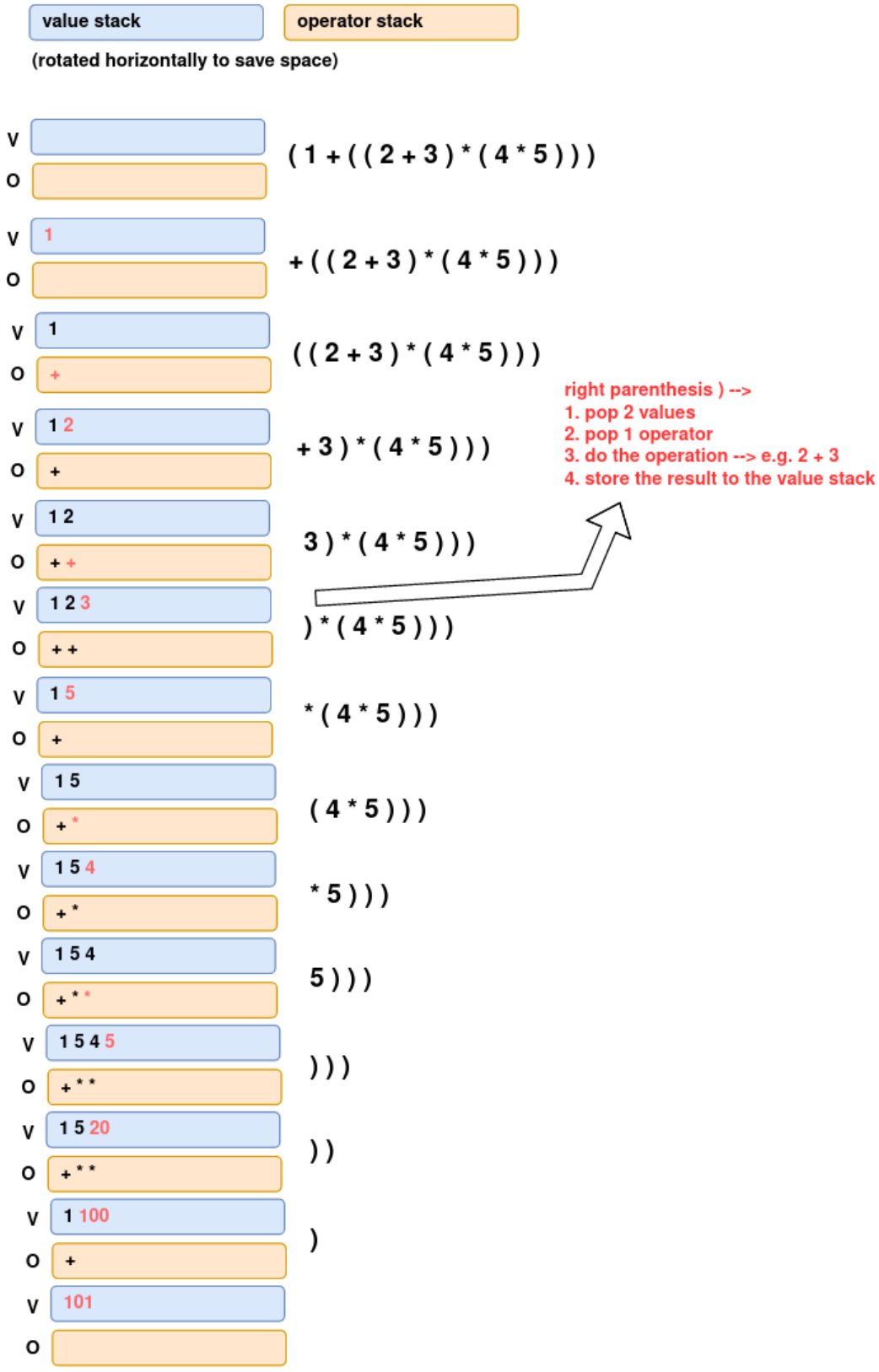
Another fun thing you can do with stacks is **calculate arithmetic expressions**.

This can be achieved using *two stacks*. The first stack is called the “value (operand) stack” and it is used to store numbers (operands). The second stack is the “operator stack”, used to store

arithmetical operations (operators). In order to use the algorithm, every individual expression **must be** enclosed in parentheses. The algorithm works in the following way:

1. take an arithmetic expression, with **properly balanced** parentheses
    a. example: **((5 + (3 * 8)) - (2 * 7))**
2. read the expression symbol by symbol
3. if *number*, push it onto the value stack
4. if *operator*, push it onto the operator stack
5. ignore *left parentheses*
6. when *right parentheses* are encountered:
    a. pop an operator and two values (from their respective stacks)
    b. perform an operation with those values (e.g. 3 * 8)
    c. push the result onto the value stack
7. Steps 3 - 6 are repeated until the end of the expression is reached, at which point the solution to the expression is the only item left in the value stack.

Here is an illustration which demonstrates how this algorithm works on one expression:
(1 + ((2 + 3) * (4 * 5)))


value stack
operator stack
(rotated horizontally to save space)
v
o
(1 + ((2 + 3) * (4 * 5)))
v 1
o
+ ((2 + 3) * (4 * 5)))
v 1
o +
((2 + 3) * (4 * 5)))
v 1 2
o +
+ 3) * (4 * 5)))
right parenthesis) -->
1. pop 2 values
2. pop 1 operator
3. do the operation --> e.g. 2 + 3
4. store the result to the value stack
v 1 2
o + +
3) * (4 * 5)))
v 1 2 3
o + +
) * (4 * 5)))
v 1 5
o +
* (4 * 5)))
v 1 5
o + *
(4 * 5)))
v 1 5 4
o + *
* 5)))
v 1 5 4
o + * *
5)))
v 1 5 4 5
o + * *
)))
v 1 5 20
o + * *
))
v 1 100
o +
)
v 101
o

In this homework assignment, your task will be to implement this algorithm **using the stack from Task 1** (no built-in Stack allowed). The algorithm has to support the following arithmetic operations:

- **addition** (2 + 4)
- **subtraction** (4 - 9)
- **multiplication** (5 * 3)
- **division** (10 / 5)
- **modulo** (15 % 4)
- **exponentiation - power** (3 ^ 5)
- **square root** - √ (√ 16)

To get started, you have a few class skeletons available in the homework project, in *src/main/java/task2*.
In the **Algorithm.java** file, you will need to implement the following methods:

- **public static Double calculate(String expression)** → takes in the entire expression as a parameter and returns the result
    - The algorithm should work with **real numbers**, not just integers.
- **public static ArrayList<Double> calculateFromFile(String filePath)** → given a TXT file path which contains a *list of expressions* (each expression in a new line), return an array list of results for each expression.
    - There is a file in **src/test/resources** called *expressions.txt*

In the **FileUtils.java** file you will need to implement:

- **public static ArrayList<String> readFile(String filePath)** → given a file path, *read all lines* from the file and return an array list containing the lines.
    - You need to use this method in calculateFromFile().

In the **App.java** file, you will need to implement:

- A simple **main()** method that lets the user choose how to interact with the algorithm.
    - A user can manually enter the expression they want into the program, via console input (using the Scanner input), or provide a path to the file containing the expressions.
    - In any case, the user should be able to then **see the result** of their provided expression(s).
    - It is up to you how to implement this method (how complex it will be, will it repeatedly ask the user, how “nice” the “UI” will look like, etc)

You must not:

- remove any of the methods in Algorithm.java or FileUtils.java, rename them nor change their signatures.

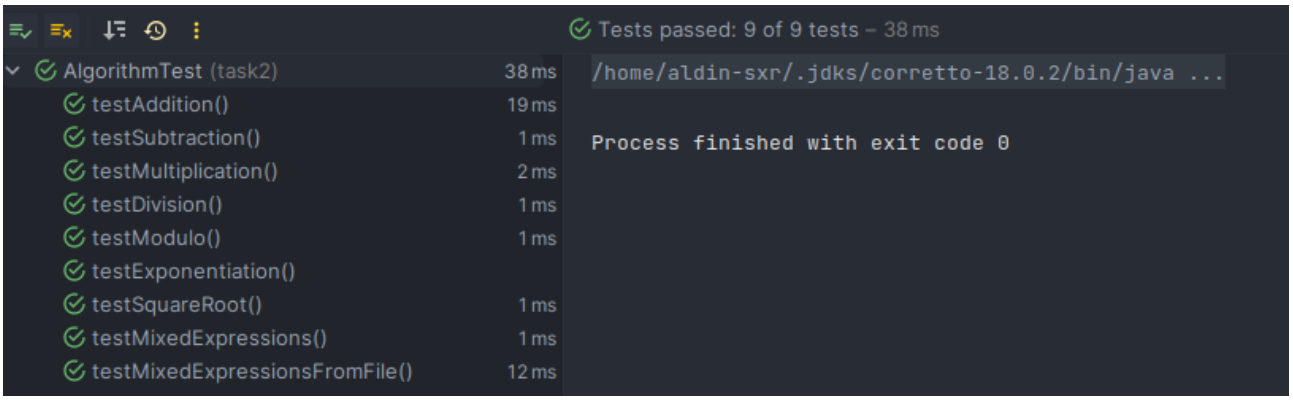
You should:

- implement the missing method bodies for the required functionalities, and make sure they return proper output (if any).

You may:

- add additional helper methods / variables *if you need them* for the solution (on the condition that you do not modify any existing method signatures, and keep the main logic in those existing methods).

In the **test** package, there is also an **AlgorithmTest.java** test class which you can run to check if your algorithm implementation is working properly. All tests should be **passed** if it is.
NOTE: All tests round off results at 2 decimals.



Some hints

Hint: In order to simplify your implementation of the algorithm, **you can** assume that each *token* (parenthesis, number or operator) is separated by a whitespace.

- Example: (2+3) → *invalid*, but (2 + 3) → *valid*

Moreover, assume that each expression is entered within parentheses and that the parentheses are *perfectly balanced*.

- example: (3 + 4) * 5 → *invalid*, but ((3 + 4) * 5) → *valid*

If you want to and know how to, you **can** try to code the algorithm so it works without whitespaces in the expressions, but sticking to the hints above is fine too.

Deadline

The assignment will be open on LMS until **April 16th, 23:55**.

Please send the professor an email if you need any further explanations, help or hints.
Good luck.