TEAM:

1. CHONTAS PANAGIOTIS
2. MARICIUC OVIDIU
3. PREPELIȚĂ ANDREI
4. TERCHEA DIANA

Phase 1 - application development

Goal

Develop an application which allows carrying out operations with very big numbers (positive integers).

Specifications

- The numbers are stored as arrays; a way of controlling the number size must be provided.
- Basic operations to be implemented: addition, subtraction, multiplication, division (integer), power, square root.
- User interface:
    - The user requests the computation of an arithmetic expression (e.g., (a + b) * a) and provides the values of the variables.
    - The program displays the results of each step in the computation. Any incorrect outcome (negative result of a subtraction, division by 0) must be signaled explicitly, which results in immediately terminating the computation.
    - Interactive mode: the expression and the values of the variables are entered through a user dialog, which allows creating the expression as a composition of basic operations. The results are also displayed in the same dialog window.
    - Automatic mode: the data input is read from an XML file. Also, the results are written into an XML file.

The programming language used for this project is Java 11.
(https://github.com/OvidiuMariciuc/SoftwareQualityProj)

(Ovidiu Mariciuc)

- **BigNumber type** - each BigNumber has a property called **value** which is a String and represents the way we decided to store the numbers.

  There are 3 ways of declaring a new BigNumber through 3 different constructors:
  BigNumber() - a constructor which takes no arguments and sets the value of the number to *0*.
  BigNumber(String value) - a constructor which takes a given String as an argument and sets the value of the number to the value of the argument. Before setting the value, there are also 2 checks: first it is checked if the given String doesn't contain illegal characters (letters, special characters, etc.); second it is checked if the given String is not null.
  BigNumber(int value) - a constructor which takes a given Integer as an argument and sets the value of the number to the value of the argument after casting it to String. It is also checked if the number provided is positive.

  I added the read/write operations over the BigNumber type - represented by the *getter* (getValue() - for reading the value) and *setter* (for writing the values) methods. There are two setters, one which can take a String as an argument and the other which can take an Integer.

  I provided 2 methods (*length()* and *chars()*) that will be helpful for the later operations between numbers: *length()* which returns the length of the String value and *chars()* which returns the array of characters from the String value.

  The BigNumber class implements the *Comparable* interface which contains only one method called *compareTo(Object)*. This method is utilized for comparing two BigNumber objects (x1, x2) and it returns 0 if the numbers are equal, 1 if the first x1 > x2 and -1 if x1 < x2.

  Finally, the *equalLengths(BigNumber x, BigNumber y)* was used to return two BigNumbers with the same length by adding zeros in front of the shorter number (*equalLengths(9998, 68) -> {9998, 0068}*).

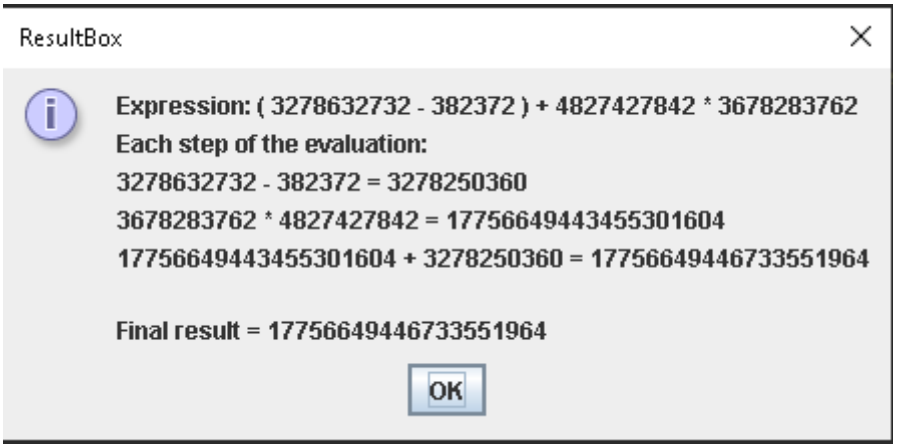
  (https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/dc78bea9c60d14971ad60c0a9cde63d0d990fe24)

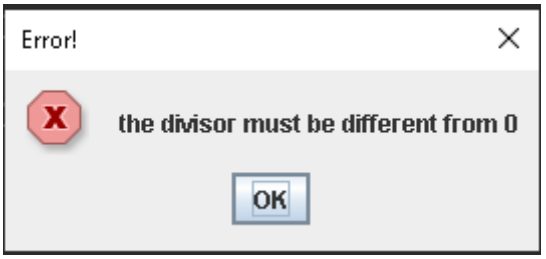
- **BigNumber operations**: addition (x1+x2), subtraction (x1-x2)

  Addition - for this basic operation, I obtained the result by adding the numbers digit by digit, from right to left, while carrying over for any sum that is larger than 10 (sum = current digit from number x1 + current digit from number x2).. The *overflow* (or the number being carried over) can only be 0 or 1. The resulting number is eventually inverted, and if the last digit added has an overflow, "1" is attached to the number.
  Subtraction - this operation is done almost the same as addition. The distinction is that if the numbers aren't large enough to be subtracted from, the overflow is taken away. This time, the *overflow* can only be -1 or 0. Because this method can leave some trailing zeroes, it is used a while loop to eliminate them.

  (https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/dc78bea9c60d14971ad60c0a9cde63d0d990fe24)

- **User interface - interactive mode**: I created a DialogBox which allows introducing the expression as a composition of basic operations. Once you press the "*Evaluate expression*" button, the intermediate and final results will be displayed in the same dialog window. Any improper result (negative subtraction result, division by 0) will result in getting an error dialog (that contains the specific error message) and the computation will be terminated immediately. Pressing the "*Cancel*" button will also result in an immediate termination of the computation.





(https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/685cfe335efea57564f1e2c28000491a4e25c39e
https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/9d9b936568928ec171c1d78733f637eecb7256ea
https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/e0d02dc00c007dca1c75c9149841f12826dad518)

Prepelita Andrei's contribution

- **The multiplication operation**: For this operation I was doing exactly how to multiply 2 numbers on the paper. In the mult method I write the second number as an array of characters and then I initialize the first number with 0 and store the final result there. After that I go through the digits of the second number and take the digits from right to left and basically at each step we add the result of multiplying the digit in position i of the second number and the number x. For this multiplication I made a recursive function multByInt, which received as a parameter the digit in position i of the second number, the first number as a string, powerof10 represents how many zeros I append at the end, the overflow of each multiplication and in the variable sb I will build the result of the multiplication, but without those zeros, which I will append at the end.

- **The division operation** : In the div method, I check if the divisor was 0 and if so it throws an exception, if not, I create a string builder object where I will have the final result. After that I write the number I want to divide as an array of characters. I initialize the overflow with 0. I browse the array, and in the variable digit I assign the overflow multiplied by 10, that is how many digits I have to take for the division to make sense + the digit in position i. In the variable quotient I will have the final result. At the end I remove the leading zeros from the final result.

- **The power operation**: For the power operation, in the pow method, I check if the exponent is 0. If it was 0, we return the result 1. Otherwise, we go from 1 to the exponent and use the multiply function to simulate the power operation.

  https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/6af58b5a632339fa3bc6ea7c8542bad1a9e2e170

- **Parsing the XML document.**
  The value attribute in the expr tag should be the actual expression. And below that variables can be specified, their names being represented by tags a to z, and to associate variables with values add the value tag attribute for each tag a to z. So basically we can only have 26 variables available.

A valid xml file with input data should look like this (input.xml)

In the parseFileXML method of the ParseXMLDocument class, we read the data from the file, retrieve the expression from the value attribute of the expr tag using Java objects, and also iterate through the tags, and retrieve the variables and associated values from the variables and store them in a String-String hashMap.

In the constructExprWithValue method of the ConstructExprWithValue class we receive as parameters the expression and the hashMap containing the variables and their associated values. And the method aims to replace the variables in the expression with the associated values in the hashMap.

https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/37a1e4a8ad742d8cac0011972809ce288fc270df

- **Putting the final result in an XML file.**

  To put the final results in the file we created the methods createXMLFileWithSteps and createXMLFileWithError .
  The createXMLFileWithSteps method is called when the expression receives valid input and has as parameters the initial expression, the steps that were performed to reach the final result, and the value representing the final result. And we build the xml tags step by step where we initially put the expression in a tag, the steps associated with

the result in several tags, and finally associate a tag for the final result. To create tags from xml we used objects from Java.

output.xml with steps result:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<expr value="9 ^ 2 / ( 2 + 7 ) + 5 * 4">
    <steps>
        <step0>9 ^ 2 = 81</step0>
        <step1>7 + 2 = 9</step1>
        <step2>81 / 9 = 9</step2>
        <step3>4 * 5 = 20</step3>
        <step4>20 + 9 = 29</step4>
    </steps>
    <result>29</result>
</expr>
```

The **createXMLFileWithError method** is used when the input is not valid, either we have an operation with a negative number as output, or we do a division to 0. And instead of the final result and the steps associated with the final result, we display the returned exception message in an xml tag.

output.xml with error message:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<expr value="9 ^ 2 / ( 2 - 7 ) + 5 * 4">
    <error>the difference must be positive</error>
</expr>
```

https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/e908cbcd17ef800d8eb0af9efc158bc683d9084a

Terchea Diana's contribution

- Transform expression to postfix notation: În polish notation the operators are written after their operands. We are using this notation to easily parse the mathematical expression. The precendece function returns for an operator his precedence in the initial expression so we can have a clear order for each operation. Transformation of the expresion goes like this: we iterate over each character and we check whether it is an operator or operand. Using the precedence function

from above, we add the operands and operators in our stack. At the end, în our stack we have the postfix notation for the initial expression.
- Evaluate expression: This class with static methods is used to evaluate the postfix expression and return the value of the initial mathematical expression. We iterate over each element of the stack previously created and whenever we find an operator we compute the result using the two operands popped from out stack and we keep the current result. At the end, we will obtain the result of the initial expression.

Chontas Panagiotis's contribution

- I've created the AutomaticMode class where we called the parseFileXML methods to read the data from the file and retrieve the expression, the variables and the values associated with the variables in the variables associated with the ParseXMLDocument class. Then we call the constructExprWithValue method of the ConstructExprWithValue class to replace the variables with their associated values from the expression. Then we call the infixToPostFix method of the InfixToPostFix class to transform the expression from infix form to postfix form. And to get the final result we call the evaluatePostfix method of the PostfixEvaluation class.
- Finally if we don't have any error we call the createXMLFileWithSteps method of the ResultInXMLFile class to put the results in an xml file, including the steps done to get the final result. And in case the input was not valid and we caught an exception, we wrote the error message returned, in the file using the createXMLFileWithError method.

Phase 2 - Unit testing

Specifications

- *Use of unit testing tools to test the code developed during phase 1.*
- *Reminder: unit testing is about discovering if the module being tested can handle incorrect input data (provided by other modules or by application I/O). Error fixing is NOT required.*
- *Testing must provide code coverage as complete as possible. For indications regarding the conditions to be tested, read the courses.*
- *Each module must be tested independently. For simulating the interactions with other modules, where necessary, mocking will be used.*

(Ovidiu Mariciuc)

For the unit-testing framework in the Java ecosystem, I decided to use JUnit 5.

The modules that I tested during this phase were *compareTo*, *equalLengths*, *add* and *sub*. The test class additionally includes an init (set-up) method that runs before each individual test and is responsible for initializing the values used in testing.

The *compareTo* module includes 3 different unit tests, 1 test for each possible result of the comparison: 0 (compareToEqualTest), 1 (compareToGreaterTest), and -1 (compareToLowerTest).

There are two unit tests in the *sub* module: one for the computation of the subtraction operation without error and one for the situation where there is an error (generated by negative subtraction result). The functionality of the *add* and *equalLengths* methods was tested directly, without having to worry about particular scenarios or errors being thrown out.

(https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/7e5d0aee67df3496bef596c59ef710fa63fe9964)

Prepelita Andrei's contribution

In the **BigNumberTest class** I have created test cases for **multiplication, division and power operations**.

- For **multiplication**, I initialized 2 BigNumber numbers and checked if the result of multiplying the 2 numbers using the multiply method, got the correct result using the assertEquals method.
- For **division** I tested 2 cases. In the first case I tested exactly as for the multiply operation, if the result of dividing 2 numbers of type gives the correct result. And in the second case we tested the case where we divide a number to 0. For this one, I used the assertThrows method to check that the divide method throws the IllegalArgumentException when the divisor was 0.
- For the **power** operation we also have 2 cases. The first case, where the power exponent was 0, and the result of the power method was supposed to be 1, was checked with the assertEquals method. And I also tested on 2 other valid inputs, where I also checked the output with the assertEquals method.

In the **ConstructExprWithValueTest class** I have created test cases for **constructExprWithValue** methods from **ConstructExprWithValue class**.

- Basically, to build the test case for the constructExprWithValue method, I initialized an expression and a HashMap with predefined values. And I checked with the assertEquals method, if the constructExprWithValue method with the previous parameters, produces the correct result, in other words replaces in the expression, the characters, with the characters from the initial hashmap.

https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/1815d072693b1e32704107008ee0bdff24e1305b

In the **ParseXMLDocumentTest class** I have created test cases for **parseXML method** from **ParseXMLDocument class**. For testing the **parseFileXML method** I created 3 test cases.

- In the test class I initialized 3 strings related to file names, an expected expression, respectively a HashMap containing as key the variables, and as values, the values assigned to the variables.
- The first test case aims to check if with valid input (existing file and xml extension) the function returns the desired expression and hashmap.
- The second test case aims to check if the method handles the case where a file with extension different from xml is given.
- And the last case is to check if the method handles the case where a file name is given as input that does not exist in the system.

https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/b2721d289a883fe043db6213847f4fd433c3fce3

Terchea Diana's contribution

In the PostfixEvaluationTest class I have created test cases for multiple scenarios:

- The first two test check if a character is evaluate as operator or not.
- The next test checks if an expression is evaluated correctly by the eval function. Given parameters were the expected value and the postfix expression as string.
- The rest of the test cases aims to check if proper error messages are thrown if the provided input for the eval function is incorrect or malformed.

Chontas Panagiotis's contribution

In the InfixToPostFixTest class I have created test cases for multiple scenarios:

- The first check tries to see if the operation with an infix gives the same result as one with a postfix operation.
- The second check tries to see if a random exception is given while giving a wrong input, if it gives the same exception means that the input is wrong.

Phase 3 - use of assertions

Specifications

- *Assertions will be inserted in the application code, in order to check the preconditions, postconditions, and invariants for the*

operations implemented during phase 1. The assertions are inserted directly into the application code, so this phase has nothing to do with unit testing.

- *If the programming used for development has no built-in assertions, a function/method must be written to provide similar behaviour.*

(Ovidiu Mariciuc)

For this phase, I have used the built-in assertions from Java.

The assertions were directly inserted in the application code for the following methods: *compareTo*, *equalLengths, add,* and *sub* in order to check preconditions, postconditions, and invariants (depending on the method).

There are some assertions for checking preconditions that are shared by multiple methods. For example, in each method mentioned above, it is checked if the numbers we are working with are not null. Another common assertion can be found when we are doing number processing, which is checking if the digits are valid (>=0 && <=9).

The *equalLengths* method also includes a check for the postcondition, which is to see if the lengths of the resulting numbers are indeed equal.

The *add* method contains a check for the invariant, which is to examine at each step *i* of the loop that the length of the generated number (after appending a new digit) is correct. In this method, the assertions linked to the postcondition have also been altered, with the length of the resulting number (after the addition of x1 and x2) now being verified if it is bigger than the lengths of x1 and x2.

The *sub* method contains 2 assertions related to the postconditions. First it is checked if the length of the resulting number is equal to the initial length without the removed zeros. Finally, I verified that the length of the generated number after the subtraction is <= the length of x1 (the number we subtracted from).

(https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/ece7960dfc7498cbed956b77dfcf354f14fe699b
https://github.com/OvidiuMariciuc/SoftwareQualityProj/commit/61e7719f04d5e90797128dd5a93656b0076ae771)

Prepelita Andrei's contribution

I have dealt with the **div** method in the **BigNumber** class. I tested the precondition that the input is non-null. I also evaluated that the number returned from the getNumericValue method be in the range 0-9, and also

evaluated that the divisor be greater than the overflow, the overflow being the rest of the division
Also, I checked the precondition in the parsefileXML method of the ParseXMLDocument class that the filename is not null. And I also added assertions in the createXMLFileWithSteps method to check that the inputs, namely expression, stepsResult and resultExpr are different from null.

Chontas Panagiotis's contribution

I dealt with methods from multiply and power. I tested the precondition and also I evaluated that the current element is an operand.

Terchea Diana's contribution

I have dealt with methods from the **PostfixEvaluation** class. I tested the precondition that the input is non-null. I also evaluated for each iteration that the current element is an operand or an operator.