Assignment 4

Publication date: *27/12/2015*
Submission date: *17/01/2016* *23:59*
People in-charge: R. Mairon and Y. Twitto

Introduction

The objectives of this assignment are to exercise a few advanced object oriented programming and basic concepts of data structures. The first mini-goal is to understand that objects may be physical, like a chair or a table, but they may also be nonconcrete, such as a number or a mathematical operation on two operands.

The second mini-goal is to learn to work with code that some third-party organization provides, which is usually done in large projects. When you are provided with classes and interfaces by some third party, you should understand how to use them and how to extend them (if needed) in the most efficient way (e.g., reusing the code as much as possible, and not copy-pasting it).

For this assignment, you will write classes that evaluate arithmetic expressions represented as text. For example, the string "1 + 2 + (3 * 4)" would evaluate to 15. This process will be similar to how Java, and other programming languages, compile human-written code into machine instructions. Compilation takes syntactic information (your code, essentially a long string of characters) and turns it into semantic information (that is, the operations you want the machine to perform).

General Guidelines:

1. If you have any questions, you are welcome to ask them at the forum page dedicated to the assignment.

2. In the code you submit, any field or methods that you might add to the classes you write, but were not explicitly required in this text, must be either private or protected to ensure encapsulation.

3. Keep in mind that your design and implementation should be flexible. Your code needs to take into account reasonable future enhancements that may be made, and should be easily extended.

4. We advise to fully read the assignment and only then start writing the code.

Postfix and Infix notations for arithmetic expressions

Arithmetic expressions are made of operators (+, -, etc.) and operands. The operands may be numbers, variables or (recursively) smaller arithmetic expressions. The expressions can be written in a variety of notations. In this assignment, we will focus on two standard arithmetic expression notations: postfix and infix. We will get deeper in the first notation and will implement a Postfix Calculator (but we won't implement an Infix Calculator).

Postfix notation is a notation in which the operator follows its operands in the expression (e.g. "2 4 +").

More formally, a postfix expression is recursively defined as follows:

1. Any number is a postfix expression.
2. If P1 and P2 are postfix expressions and Op is an operator, then "P1 P2 Op" is a postfix expression.

Infix notation is the common arithmetic and logical formula notation, in which an operator is written between the operands to which it is applied (e.g. "2 + 4").

More formally, an infix expression is recursively defined as follows:

1. Any number or variable is an infix expression.
2. If I1 is an infix expression, then "(I1)" is an infix expression.

3. If I1 and I2 are infix expressions, and Op is an operator, then "I1 Op I2" is an infix expression.

Using parentheses to resolve ambiguity:
Note that the **postfix** expression representation does not require the use of parentheses. That is because the order of operations is clear (there is no ambiguity) from the order of the operands and operators. However, this is not the case in **infix** notations, which naturally give rise to ambiguous interpretation. Due to this ambiguity, parentheses surrounding groups of operands and operators are necessary to indicate the intended order in which operations are to be performed.

In the absence of parentheses, certain precedence rules determine the order of operations. For example, it is customary that the multiplication operation (*) has higher precedence than the addition operation (+), and thus the infix expression "3+5*2" is interpreted as: "first compute 5*2 and then add 3 to the resulting product". This is equivalent to the postfix expression "3 5 2 * +". Parentheses can be applied to "override" the default (operator-precedence based) order of operations: "(3 + 5) * 2" is interpreted as: "first compute 3+5 and then multiply the obtained sum by 2". This is equivalent to the postfix expression "3 5 + 2 *".

The following table demonstrates some infix expressions and their equivalent prefix expressions.

Infix expression	Postfix expression
1 + 2 + 3	1 2 + 3 +
6 * 5 + 4	6 5 * 4 +
(7 + 2) * 4	7 2 + 4 *
(4 + 3) - (2 * 7)	4 3 + 2 7 * -

Note that, both in infix and postfix notation, the results of the application of some operations become the operands of other operations. For example, in the second example in the table above: “6 * 5 + 4”, "6 * 5" is the first operand for the "+" operator.

The Tasks in Assignment 4 and their Scoring

The total score for this assignment is 100 points.
The work is divided as follows.
Part 1: Implementation of postfix calculator: **50** points
Part 2: Exceptions handling: **20** points
Part 2: Testing: **20** points
Part 3: Documentation using JavaDoc: **10** points

Part 1 (Postfix Calculator) – 50 Points

In this part of the assignment you are asked to write a program that simulates a postfix calculator. More specifically, the program you write should take, as input, an expression in postfix notation and return, as output, the computed value of the given expression. This involves various tasks such as parsing the input text into tokens (i.e. small substrings, delimited by “spaces”), using a stack data-structure class (as will be explained shortly), and then implementing an algorithm which uses the stack and the token parser to evaluate a given postfix expression.

The *Stack* Interface

A Stack holds elements in a Last-In-First-Out (LIFO) fashion (Stack - Wikipedia). This means that the last element that was inserted to the stack is the first one to come out, just like the functions call stack (as we learned in class) or a weapon magazine. The Stack interface supports the following operations:

- `void push(Object element)` – inserts a given element to the top of the stack.

- `Object pop()` – removes the first element from the top of the stack, and returns it.

- `boolean isEmpty()` – returns true if-and-only-if there are no elements in the stack.

The Stack interface is provided in the file Stack.java, which is part of the assignment4.zip file supplied to you with the exercise. The stack, as will be taught in future lessons, is one of the most basic data-structures in computer science. You are encouraged to look at the code and try to understand it.

In the next section we give the suggested postfix evaluation algorithm.

A Postfix Expression Evaluation Algorithm

There is a very simple algorithm for evaluating syntactically correct postfix expressions, which reads the expression left to right in chunks, or tokens. In this part of the assignment, the tokens are substrings delimited by “spaces”, and consist of numbers and/or operators. For example, the expression “6 3 +” will generate the series of three tokens: “6”, “3” and “+”. The pseudo code for the postfix expression evaluation algorithm is given below.

```
while tokens remain:

        read the next token (from left to right).

        if the token is an operator:
            pop the top two elements of the stack.
            perform the operation on the elements.
            push the result of the operation onto the stack.
        else:
            push the token (which must be a number) onto the stack
```

When there are no tokens left, the stack must contain only one item: the final value of the expression. For an example with stack diagrams, see the Wikipedia page on postfix notation.

In the next section we list and describe the classes we provide, and the classes you need to implement for this part of the assignment.

The classes we provide

Class *StackAsArray*

We provide the **StackAsArray** class (in the StackAsArray.java file) which implements the Stack interface using a dynamic array with an increasing capacity that can be expanded as needed. You are encouraged to look at the code and try to understand it.

Class *CalcToken*

The abstract class **CalcToken** (in the CalcToken.java file) is provided with basic common capabilities that will assist you in parsing the tokens of the string.

Class *BinaryOp*

The abstract class **BinaryOp** (in the BinaryOp.java file) is provided and is used to represent a binary operation, an operation that is applied on two operands. Take a look at the code and think of the reason this class is abstract.

The classes you need to implement or modify

Token Classes

You will create the token classes, which are all subclasses of the abstract class CalcToken (CalcToken.java is provided). For this part of the assignment, there are two types of tokens:

1. **Tokens which represent a numerical value**. You must create a class **ValueToken** with the following methods/constructors:
    - `ValueToken(double value)` where "value" is the number represented by the token.
    - `double getValue()` returns the value of the token.
    - `String toString()` which is inherited from the abstract parent class CalcToken.

2. **Tokens which represent operators**. These are described by the abstract class BinaryOp (BinaryOp.java is provided). You must implement the classes **AddOp**, **SubtractOp**, **MultiplyOp**, **DivideOp**, and **PowOp**, which represent addition, subtraction, multiplication, division, and power respectively. All subclasses of BinaryOp must have the following methods:
    - `double operate(double left, double right)` returns the result of the operation using its left and right operands (note that the order of operands can matter, depending the operator).
    - `String toString()`, which is inherited from CalcToken and represents the mathematical symbol of the operation.

Class ExpTokenizer

In order to convert a string into a series of tokens, you will need to have the class ExpTokenizer. For this purpose we created most of the class for you. You are encouraged to look at the code and try to understand it.

The tokenizer can go over the expression from left to right or from right to left. To set the orientation of the tokenizer, use the boolean parameter ***direction*** in the constructor. For example in this part you need a left to right tokenizer. So, to initialize

the tokenizer, we will call the constructor in the following way: ExpTokenizer(exp, **true**).

Note: you may assume that all the tokens are separated by the space character.

You will need to modify the method `nextElement()`, as exemplified below.

```
public Object nextElement() {
        CalcToken resultToken = null;
        String token =  nextToken();
        if (token.equals("+"))
                resultToken =  new AddOp();
        else if (token.equals("*"))
                resultToken =  new MultiplyOp();

        // Fill the rest of the token cases by yourself

        else
            resultToken = new ValueToken(Double.parseDouble(token));

        return resultToken;
}
```

You will need to support both real positive and negative numbers (e.g. -4 or 4.2).

The Calculator Class

The **Calculator** class is an abstract class that you are required to create in a file name Calculator.java. This class will be used to define the features of a calculator. The **PostfixCalculator** class (see next) will extend it. The class must have the following public methods:

- `void evaluate(String expr),` where *expr* is a String representing some expression. This method evaluates (i.e. computes the numerical value of) *expr.*
- `double getCurrentResult()`, returns the result of the last expression that was evaluated. Think about what should be the appropriate returned value in case that no expression was parsed.

You should figure out which of the methods above should be abstract and which should be implemented.

Class PostfixCalculator

The primary class in the code, which you will implement for Part 1 is Class **PostfixCalculator**. This class will extend the Calculator class. Note that the method **`evaluate`**`(String expr),` receives *expr* which is a String representing a postfix expression. This method evaluates (i.e. computes the numerical value of) *expr*, and stores its value. The value can be restored by using the **`getCurrentResult`**`()` method. This method should use the algorithm described above, and should be implemented by using a StackAsArray object.

Examples for some input postfix expressions, as well as the expected output value, are given below.

Examples of Postfix Expressions and their Corresponding Output

```
PostfixCalculator c1 = new PostfixCalculator();

c1.evaluate("2 3 +");
c1.getCurrentResult(); //returns 5.0

c1.evaluate("3 5 -");
c1.getCurrentResult(); // returns -2.0

c1.evaluate("6 2 *");
c1.getCurrentResult(); // returns 12.0

c1.evaluate("10 4 /");
c1.getCurrentResult(); // returns 2.5

c1.evaluate("2 4 ^");
c1.getCurrentResult(); // returns 16.0

c1.evaluate("2 3 + 4 2 - *");
c1.getCurrentResult();  // returns 10.0

c1.evaluate("2 3 ^ 4 2 * / 7 -");
c1.getCurrentResult(); // returns -6.0

c1.evaluate("2 3 ^ 4 2 * / -7 -");
c1.getCurrentResult(); // returns 8.0
```

What is NOT allowed

You may **not** use any classes from `java.util` or anywhere else (Including the Java Collections such as List).

Part 2 (exceptions handling) – 20 Points

In this part of the assignment you will need to add support for invalid postfix expressions. A valid postfix expression was described previously. An invalid expression is any other string. This part is an add-on over the previous part. We are **upgrading** our calculator to handle invalid expressions!

Class ParseException

Create a **ParseException** class that inherits from the RuntimeException class. This class should have a constructor **`ParseException`**`(String message)`, that receives the error message that caused the parsing exception as a parameter.

Class ExpTokenizer

Modify the class ExpTokenizer so it will be able to detect invalid tokens, and **throw a ParseException** in case **the token is invalid**. The thrown ParseException message should be “SYNTAX ERROR: blah” where “blah” is whatever description you choose to give (to the user).

Class *PostfixCalculator*

You need to modify the `double evaluate(String expr)` method. As before it will process and evaluate *expr* using the previously stated algorithm, and return its value. But now, if **the given expr is invalid**, a ParseException exception should be thrown.

Examples of Invalid Postfix Expressions and their Errors

```
PostfixCalculator c3 = new PostfixCalculator();

c3.evaluate(“4 5 $”);  // SYNTAX ERROR: invalid token $
c3.evaluate(“7.0.1”);  // SYNTAX ERROR: invalid number 7.0.1
c3.evaluate(“5 6 7 4”);  // SYNTAX ERROR: write here whatever
description you want
c3.evaluate(“*”);  // SYNTAX ERROR: cannot perform operation * (or
whatever description you choose to give here)
c3.evaluate(“5 6 * 4 2 * 1 2 *”);  // SYNTAX ERROR: invalid
expression
```

Part 3: Testing – 20 Points

Testing is a very important aspect of writing good and correct code. You can read about one possible framework for code testing here:

http://www.cs.bgu.ac.il/~intro161/Tutorial/Testing

We, however, will use a different approach. You should test every public method, using extreme ("boundary") correct and incorrect values, where allowed by preconditions, as well as the obvious "middle of the road" values. Remember to choose **efficient** test cases. For example, testing `PostfixCalculator.evaluate(String expr)` with strings "1 2 +", "3 7 +", and "21 4 +" is unnecessary, since they test the exact same thing - in this case, the addition of two numbers. We suggest the following approach:

- Test basic methods before complicated ones.
- Test simple objects before objects that include or contain the simple objects.

- Don't test too many things in a single test case.

We also provide the file **Tester.java**. This file includes a main function that will be in-charge of running all the tests for your code. A helper function that you will use is the following:

```
private static void test(boolean exp, String msg)
```

This function receives a Boolean expression, and an error message. If the Boolean expression is evaluated to false, the function will print the error message to the screen. The function will also "count" for you the number of successful tests that you executed.

For example: The following test creates a Postfix Calculator, and checks if the expression "1 2 +" evaluates to 3.0:

```
PostfixCalculator pc = new PostfixCalculator();
test(pc.evaluate("1 2 +") == 3.0, "The output of \"1 2 +\" should be 3.0");
```

Please read carefully the code in this file. As you can see, we already provided a few tests to your code. You are required to add your own tests to check Part 1 and Part 2 of the assignment, so the output of running the Tester, will be:

```
All <i> tests passed!
```

Where *i* (the total number of tests) should be at least **50**.

Part 4: Documentation using Javadoc – 10 Points

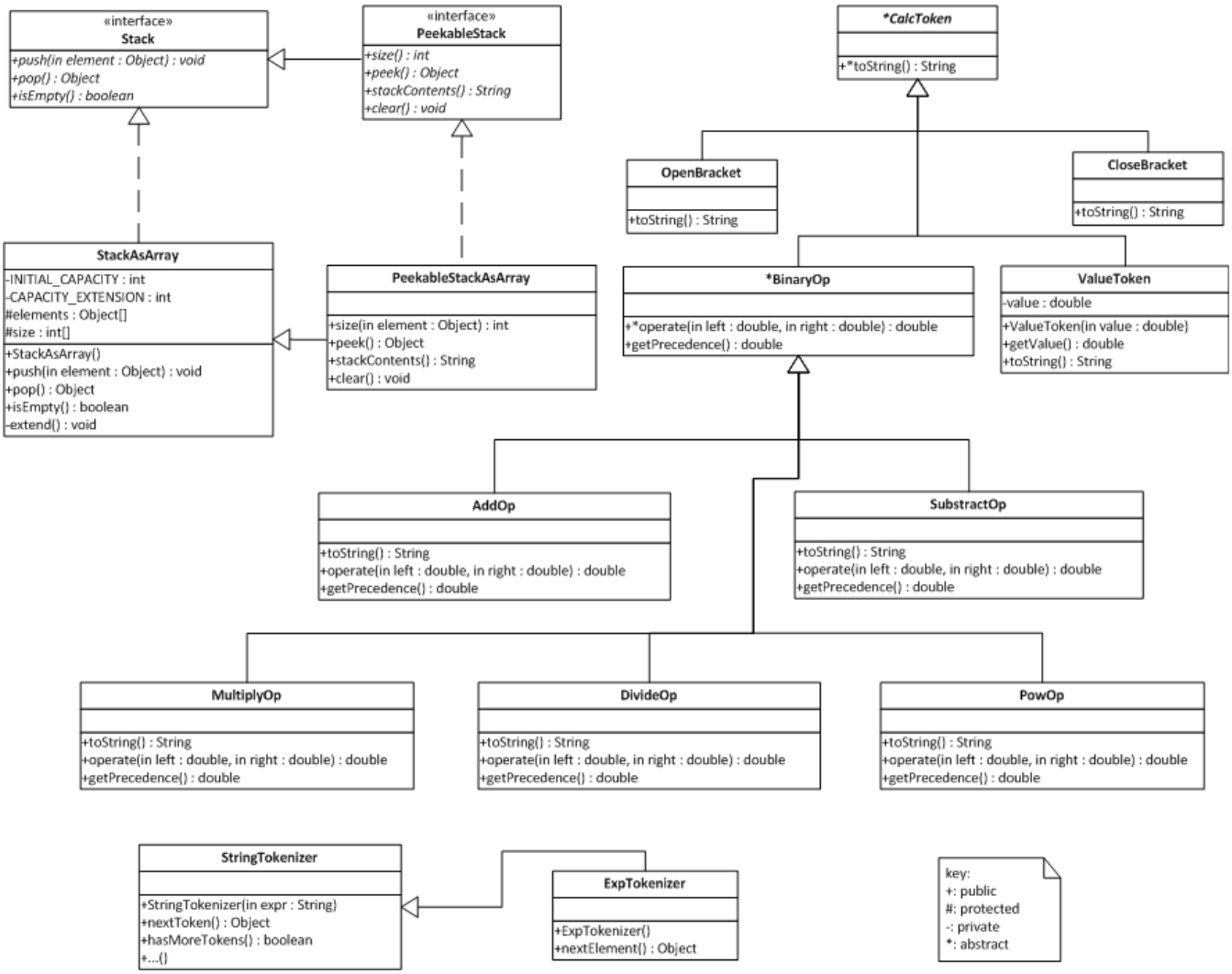
In this homework, you are required to document the headers of all the classes and the headers of all the methods using **Javadoc**. This **includes** the classes and methods we provided you – you should perfect their documentation as well, and adhere it to the Javadoc format wherever it is not such. A penalty of up to 10 points may apply in the case of missing Javadoc in the headers of classes and methods.

We do not require comments that give perfect descriptions – a reasonable description of the classes and methods in the Javadoc will suffice. We do not require that you document your variable and other entities of the code using Javadoc, you can documents them using regular comment (or in Javadoc, whatever you choose). Please read this link regarding the basics of Javadoc. It will help you to get started.

Besides of the requirement to use the Javadoc format for your classes and methods documentation, the style expectation are much the same as in the previous assignments: comments explaining complicated code, well-chosen names for variables and methods, clear indentation and spacing, etc. In particular, we expect you to follow the capitalization conventions: variableName, methodName, ClassName, PUBLIC_STATIC_FINAL_NAME.

Class Diagram

For your convenience we also provide a partial class diagram for Part 1, that describes the classes you will use and their interactions (extensions and implementations). The Calculator, PostfixCalculator, is missing from this diagram.



What to submit for assignment 4

You need to submit your files in a single archive (zip) file. The archive should not contain any directories; it should contain only the files themselves. The java files:

- **StackAsArray.java**.
- The token classes, in files **ValueToken.java**, **AddOp.java**, **SubtractOp.java**, **MultiplyOp.java**, **DivideOp.java**, and **PowOp.java**.
- **ExpTokenizer.java**.
- **Calculator.java, PostfixCalculator.java**.
- **Stack.java**, **CalcToken.java** and **BinaryOp.java**.

- **ParseException.java.**

Note: You also need to submit the java files we provided which should remain complete and unmodified (such as Stack.java). This means that in those files you should only update and enrich the documentation, and align it to Javadoc format.

GOOD LUCK