Assignment 4

Publication date: 29.12
Submission date: 14.01 23:59
People in-charge: Yonatan Alexander and Tal Beja.

הוראות מקדימות:

1. העבודה כתובה בלשון זכר (מטעמי נוחיות), אך פונה לשני המינים.
2. ניתן ומומלץ לבצע את העבודה בזוגות בהתאם להוראות המופיעות בסילבוס הקורס.
3. כל איחור בהגשת התרגיל יגרום להורדת ציון, כפי שהוגדר בנהלים שבאתר הקורס.
4. תרגיל אותו לא ניתן להדר(לקמפל) או להריץ, יקבל ציון נכשל.
5. לעבודה מצורפים שבעה קבצים:
   BinaryOp.java, CalcToken.java, ExpTokenizer.java, Tester.java, Stack.java, StackAsArray.java, PeekableStack.java.
6. יש להגיש את כל הקבצים המצורפים בנוסף לקבצים שאתם יוצרים במהלך העבודה מקובצים בקובץ zip. **לא ייבדקו קבצים שיוגשו בפורמט שונה מzip.**
7. את הקובץ יש להגיש בSubmission System.
8. העבודה תיבדק **באופן אוטומטי**. לכן, יש להקפיד על ההוראות ולבצע אותן **במדויק**. כל חוסר מעקב אחר ההוראות **יגרור פגיעה משמעותית בציון**.
9. סגנון כתיבת הקוד ייבדק באופן ידני. יש להקפיד על כתיבת קוד ברור, מתן שמות משמעותיים למשתנים, הזחות (אינדנטציה) **והוספת הערות בקוד** המסבירות את תפקידם של מקטעי קוד שונים. אין צורך למלא את הקוד בהערות סתמיות אך חשוב לכתוב הערות בנקודות קריטיות המסבירות קטעים חשובים בקוד. כתיבת קוד אשר אינו עומד בסטנדרטים אלו תגרור הפחתה בציון העבודה. בנוסף, הערות יש לרשום אך ורק ב**אנגלית** (כתיבת הערות בכל שפה אחרת שקולה לאי כתיבת הערות).
10. בכדי לוודא שהגשתם את התרגיל באופן תקין – הורידו את קובץ הzip שהגשתם ב Submission System למחשב שלכם, חלצו אותו ונסו להדר ולהריץ את התוכנית. הליך זה הוא חשוב, עקב מקרים שקרו בעבר בהם הקובץ שהועלה למערכת ההגשות היה פגום.
11. במידה ואינכם בטוחים מהו הפירוש המדויק להוראה מסוימת, או בכל שאלה אחרת הקשורה **בתוכן** העבודה, אנא היעזרו בפורום או בשעות הקבלה של האחראים על העבודה (פרוט שעות הקבלה מופיע באתר הקורס). בכל בעיה **אישית** הקשורה בעבודה (מילואים ,אשפוז וכו') אנא צרו את הפניה המתאימה במערכת הגשת העבודות כפי שמוסבר בסילבוס שבאתר הקורס.
12. שימו לב כי בעבודה ניתנו 5 נקודות "תמריץ" עבור מעקב אחר הוראות העבודה. במידה וההוראות לא מולאו (שכחתם להוסיף שותף במידה ובחרתם להגיש בזוגות, חוסר בדיקה של פורמט הקבצים, שמות לא נכונים וכדומה) הדבר יגרור הורדה של נקודות אלו.
13. בעבודה זו ניתן לצבור עד 115 נקודות:
    a. 5 נקודות עבור מעקב אחר הוראות (סעיף 12).
    b. 15 נקודות עבור הבדיקה הידנית: כתיבת הערות 7 נק', הזחה 4 נק' ושמות משתנים 4 נק'.
    c. 80 נקודות על כתיבת הקוד בהתאם לדרישות התרגיל.
    d. 15 נקודות בונוס על החלק הבונוס.

Introduction

The objectives of this assignment are to exercise a few advanced object oriented programming and basic data structures concepts. The first mini-goal is to understand that objects can be something which is not physical, like a chair or a table, but also something more nonconcrete, like 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 receive classes and interfaces from some third party, you should understand how to use them and how to extend them (if needed) in the most efficient way (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).
If you have any questions, please ask them at the assignment's forum.
In your code, all fields and all methods that you may add to the classes and not explicitly stated in this text, must be either private or protected to ensure encapsulation.
We suggest you document your methods using Javadoc (http://www.cs.bgu.ac.il/~intro141/Practical_Sessions/Javadoc). In addition you need to add comments to your code, fields, and variables as was done in previous assignments.
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.

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

Prefix and Infix notations for arithmetic expressions:

Arithmetic expressions are made of operators (+, -, etc.) and operands (either 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: prefix and infix without variables.

Prefix notation is a notation in which the operator comes before its operands in the expression (e.g. “+ 2 4**”**).

More formally, in this assignment a prefix expression is recursively defined as follows:

1. Any number is a prefix expression.
2. If P1 is a prefix expression, P2 is a prefix expression, and Op is an operator, then “Op P1 P2” is a prefix expression.

Infix notation is the common arithmetic and logical formula notation, in which an operator is written between the operands it acts on (e.g. “2 + 4”).

More formally, in this assignment 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 prefix expression representation does not require the use of parentheses- the order of operations is clear (no ambiguity) from the order of the operands and operators. However, such 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 prefix 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 prefix expression “* + 3 5 2”.

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

Infix expression	Prefix 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 prefix 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 115 points.
The work is divided as follows.
Part 1: 50 points
Part 2: 40 points
Testing: 10 points
Bonus: 15 points

Part 1 (Prefix Calculator) – 50 Points

In this part of the assignment you are asked to write a program that simulates a prefix calculator. More specifically, the program you write should take, as input, an expression in prefix 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 prefix 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. Stack, as will be thought 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 prefix evaluation algorithm. Note that, in this part of the assignment, it is ok to assume that the input expression is syntactically correct (i.e., the input is a valid prefix expression string).

A Prefix Expression Evaluation Algorithm

There is a very simple algorithm for evaluating *syntactically correct* prefix expressions, which reads the expression **right to left** 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: “3”, “6” and “+” from left to right. The pseudo code for the prefix expression evaluation algorithm is given below.

```
while tokens remain:

        read the next token.

        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 prefix 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 also 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 (again) 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. And again, you are encouraged to look at the code and try to understand it. We deliberately made this an abstract class and not an interface in order to make sure that the classes that extend it will override the `toString()` method.

Class *BinaryOp*

The abstract class BinaryOp (in the BinaryOp.java file) is provided and is used to represent a binary operation, that is an operation that is executed on two operands. You are encouraged to look at the code and try to understand it. You should also think why 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 val)` where val is the number the token should represent.
    - `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 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 operand order can matter, depending on 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 (again) encouraged to look at the code and try to understand it.
This 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 leftToRight in the constractor. For example in this part you need a right to left tokenizer. So to initialize it we will call the constractor in that way: ExpTokenizer(exp, **false**);. In the next part we need the left to right one. So to initialize it we will call the constractor in that 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 CalcToken nextElement() {
        CalcToken resultToken = null;

        String token = nextString();
        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;
    }
```

Note: you may assume that all the tokens in the input expression are either real numbers, or one of the five operators: "+", "-","*"," ^" and "/". In the future we may add more types of tokens, but then this code will be modified.
You will need to support both real positive and negative numbers (e.g. -4 or 4.2).

Class Calculator

This will be an abstract class that you will need to create in a file Calculator.java. This class will be used to define a calculator's features, and both the InfixCalculator and PrefixCalculator 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 current result of the last expression that was evaluated. Think what the returned value should be in case no expression was parsed.

You should think which of the methods above should be abstract and which should be implemented here.

Class PrefixCalculator

The primary class in the code, which you will implement for Part 1 of the assignment, is Class PrefixCalculator. This class will extend the Calculator class. Note that the method `evaluate(String expr)`, receives *expr* which is a String representing a valid prefix 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 prefix expressions, as well as the expected output value, are given below.

Examples of Prefix Expressions and their Corresponding Output

```
PrefixCalculator c1 = new PrefixCalculator();

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
```

Note: You **may** assume, in Part 1 of the assignment (if you do not submit the bonus), that the input expression is a valid prefix expression

Part 2 (Infix Calculator) – 40 Points

For the second part of the assignment, you will write classes to evaluate expressions in regular *infix notation*. Note that even though we are more familiar with this format of writing expressions, these expressions are more difficult to evaluate. In this part of the assignment, you will need to take into account the predefined operator precedence definitions, as well as the bracketing used to override precedence (e.g. "(4 + 5) * 6").

An Infix Expression Evaluation Algorithm

The algorithm we will use for evaluating infix expressions (which, we should note, is *not* the algorithm used by compilers) is similar to the algorithm from Part 1 in the sense that it reads the expression left to right in chunks, or *tokens*. However, in addition to the previous tokens which were restricted to numbers and operators, the infix expression evaluator is extended to support bracket tokens as well, and to take into account operator precedence.

The main operation used by the algorithm is called *collapse* because it replaces three items on the stack with one simpler but equivalent item.

The pseudo-code for the *collapse* operation, which takes as input a new token T and then iteratively "peeks" at the top three elements in the stack before deciding whether to "collapse" them or not is given below:

Collapse(T)

```
While not end collapse:

  Consider the new token T, in addition to E0, E1, and E2 (if there
  are at least 3 elements), which are the first three elements in the
  stack, in top-down order, and decide whether to apply Case 1, Case
  2 or Case 3 below.

    • Case 1: E0 and E2 are Value Tokens and E1 is a Binary
            Operation and
      (
            (If T is a Binary Operation with lower or equal
             precedence than E1)
         or
           (T is not a Binary Operation)):
                pop E0, pop E1, pop E2
                apply the binary operator E1 to operands  E0 and E2
                push the result

    • Case 2: E0 and E2 are, correspondingly, close and open
            brackets, and E1 is a value:
                pop E0, pop E1, pop E2
                push E1

    • Case 3: neither Case 1 nor Case 2 above apply, then end collapse.
```

Note that the collapse operation **does not** push the new token (i.e. T, above) onto the stack (it will be pushed in the evaluation stage). Also note that **multiple collapses are possible (the iterative while loop),** if a collapse creates the criteria for a subsequent collapse (See the Infix Evaluation Example below).

The table below summarizes the decisions taken by the collapse operation, given E0,E1,E2 and the new token T. (The pseudo code for this operation will follow.)

E0	**E1**	**E2**	**Additional criteria**	**Action**
ValueToken	BinaryOp	ValueToken	- T is a BinaryOp and E1 has equal to or higher precedence than T **or** - T is not a BinaryOp	- pop E0, E1, E2 - push the value of E1 with operands E2 and E0
CloseBracket	Value	OpenBracket		- pop E0, E1, E2 - push E1

The pseudo code for the infix expression evaluation algorithm, which employs the collapse operation, is given below:

```
while tokens remain:

   read the next token T.

   Collapse(T).

   push the token T onto the stack.

when no more tokens:

   while stack has more than one element:

      Collapse(null).
```

When the above pseudo code completes, and there are no tokens left, the stack only contains one item: the final value of the given infix expression. As can be seen from the above pseudo-code, the Collapse method must also address the case where a null argument is passed. In this case, it should work as described in the Collapse pseudo-code (where T is null).
In the next section we give an example to demonstrate the above algorithm for infix expression evaluation.

An Infix Expression Evaluation Example

Consider the string "(1 + 2) + 4 * 5". The steps of evaluation are as follows:

Stack Contents	Next Token	Action Taken
empty	(	• collapse: does nothing • push (
(	1	• collapse: does nothing • push 1
1 (	+	• collapse: does nothing • push +
+ 1 (	2	• collapse: does nothing • push 2
2 + 1 (	)	• collapse: ○ pop 2, pop +, pop 1 ○ push 3 [= 1 + 2] • push)
) 3 (	+	• collapse: ○ pop), pop 3, pop (○ push 3 • push +
+ 3	4	• collapse: does nothing • push 4
4 + 3	*	• collapse: does nothing (since * has higher precedence than +) • push *
* 4 + 3	5	• collapse: does nothing • push 5
5 *	None	• collapse: ○ pop 5, pop *, pop 4 ○ push 20 [= 4 * 5]

4 + 3		
20 + 3	None	• collapse: ○ pop 20, pop +, pop 3 ○ push 23 [= 3 + 20]
23	None	none - finished!

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

The classes you need to implement or modify

Class *PeekableStackAsArray*

Because the task of evaluating infix expressions is more complex than that of evaluating prefix expressions, we require an *augmented* stack with some additional functionality. More specifically, we need to be able to "peek" into the stack and examine some of its elements without removing them from the stack. You are provided with the PeekableStack interface (in the PeekableStack.java file), which includes the following methods:

- since PeekableStack extends Stack, it includes all of the Stack methods (push(), pop() and isEmpty()).
- `Object peek(int i),` which returns the i-th stack element from the top of the stack. There is no need to return a deep copy of the object. Remember that in computer science we start counting from 0. If the stack contains less than i elements, an `EmptyStackException` should be thrown. (You will need to import `java.util.EmptyStackException`).
- `int size(),` which returns the current size of the stack.
- `void clear(),` which empties the stack.
- `String stackContents()`, which returns a string representing the stack's contents. The purpose of this method is to help you in the process of debugging your code.

This interface must be implemented in a class called PeekableStackAsArray, which (surprisingly) uses an array to hold the stack items. You may not copy code from your StackAsArray class (think how you can reuse the code you already were given in other classes).

Token classes

For this part of the assignment, you will be modifying the token classes which you previously created in Part 1 to support precedence. To this end, you will need to override the getPrecedence() method in each Binary-Operation class as will be described in the next section.

In addition, the expression tokenizer requires the following types of tokens:
Tokens which represent additional allowed symbols, such as brackets "(", ")". You must provide classes OpenBracket and CloseBracket, which are subclasses of CalcToken, and have no additional methods.

Operator Precedence

This part of the assignment also uses the getPrecedence() method that exists for every Binary-Operation. You must ensure that for any two BinaryOp objects A and B,

1. A.getPrecedence() == B.getPrecedence() if the respective operations are of equal precedence.
2. A.getPrecedence() > B.getPrecedence() if A's operation occurs before B's operation (i.e. it is of higher precedence).

Notes:

- Power ("^") precedence is higher than Multiplication ("*") precedence.
- Multiplication("*") precedence is higher than Addition("+") precedence.
- Multiplication("*") has the same precedence as Division ("/").
- Addition("+") has the same precedence as Subtraction("-").

Class ExpTokenizer

In this part you will need to add to modify the class ExpTokenizer that you implemented in part 1. Update the method nextElement() in ExpTokenizer.java so it will also handle the bracket tokens: "(", and ")".

Class *InfixCalculator*

The primary class of this part of the assignment is InfixCalculator. This class should extend the Calculator class (you implemented in Part 1). It should implement the `void evaluate(String expr)`, which processes *expr* using the above algorithm and a PeekableStackAsArray. This method evaluates the infix expression *expr*, and stores its value which can be retrieved by using the `getCurrentResult()` method.
You should also provide the following private method:

- `void collapse(CalcToken token)`, which collapses the stack until a collapse rule cannot be applied, given the next token.

Examples for some input infix expressions, as well as their expected output values, are given below.

Examples of Infix Expressions and their Corresponding Output

```
InfixCalculator c2 = new InfixCalculator();

c2.evaluate("5 + 2");
c2.getCurrentResult(); //returns 7.0

c2.evaluate("4 ^ 2 * 3 / 8 + 2 – 10");
c2.getCurrentResult();  //returns -2.0

c2.evaluate("0.5 + 2 * 3 ^ 2");
c2.getCurrentResult();  //returns 18.5

c2.evaluate("( 4 )");
c2.getCurrentResult();  //returns 4.0

c2.evaluate("( ( 10 ^ 2 ) * ( 10 – 10.02 ) )");
c2.getCurrentResult();  //returns -2.0

c2.evaluate("( ( 10 / ( 2 ^ 2 ) / 3.75 ) * ( 6 ) ) ^ ( 0.5 * ( ( (
10.5 - 6.5 ) ) ) )");
c2.getCurrentResult();  //returns 16.0
```

Note: You **may** assume, in Part 2 of the assignment (if you do not submit the bonus), that the input expression is a valid infix expression.

Bonus Part – 15 Points (optional)

In the bonus part of the assignment you will need to add support for valid and invalid infix expressions. A valid infix expression was described in part 2. An invalid expression is any other string.

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 the invalid token.

Class *InfixCalculator*

You need to modify the `double evaluate(String expr)` method. As before it will process and evaluate *expr* using the algorithm in part 2, and a PeekableStackAsArray, and return its value. If the given expr is invalid, a ParseException exception should be thrown.

Examples of Invalid Infix Expressions and their Errors

```
InfixCalculator c3 = new InfixCalculator();

c3.evaluate(“4 $ 5”);  // SYNTAX ERROR: $
c3.evaluate(“( 7.0.1 )”);  // SYNTAX ERROR: 7.0.1
c3.evaluate(“5 + 6 * ( 4.0 ) +”);  // SYNTAX ERROR: cannot perform
operation +
c3.evaluate(“*”);  // SYNTAX ERROR: cannot perform operation *
c3.evaluate(“) 5 (”);  // SYNTAX ERROR: invalid parentheses format
```

What is NOT allowed

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

Testing – 10 Points (mandatory)

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/~intro141/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 `PrefixCalculator.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.

In case you decide to submit the bonus part, please provide tests that test both valid and invalid expressions.

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 Prefix Calculator, and checks if the expression “1 2 +” evaluates to 3.0:

```
PrefixCalculator pc = new PrefixCalculator();
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 both Part 1 and Part 2 of the assignment, so the output of running the Tester, will be:

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

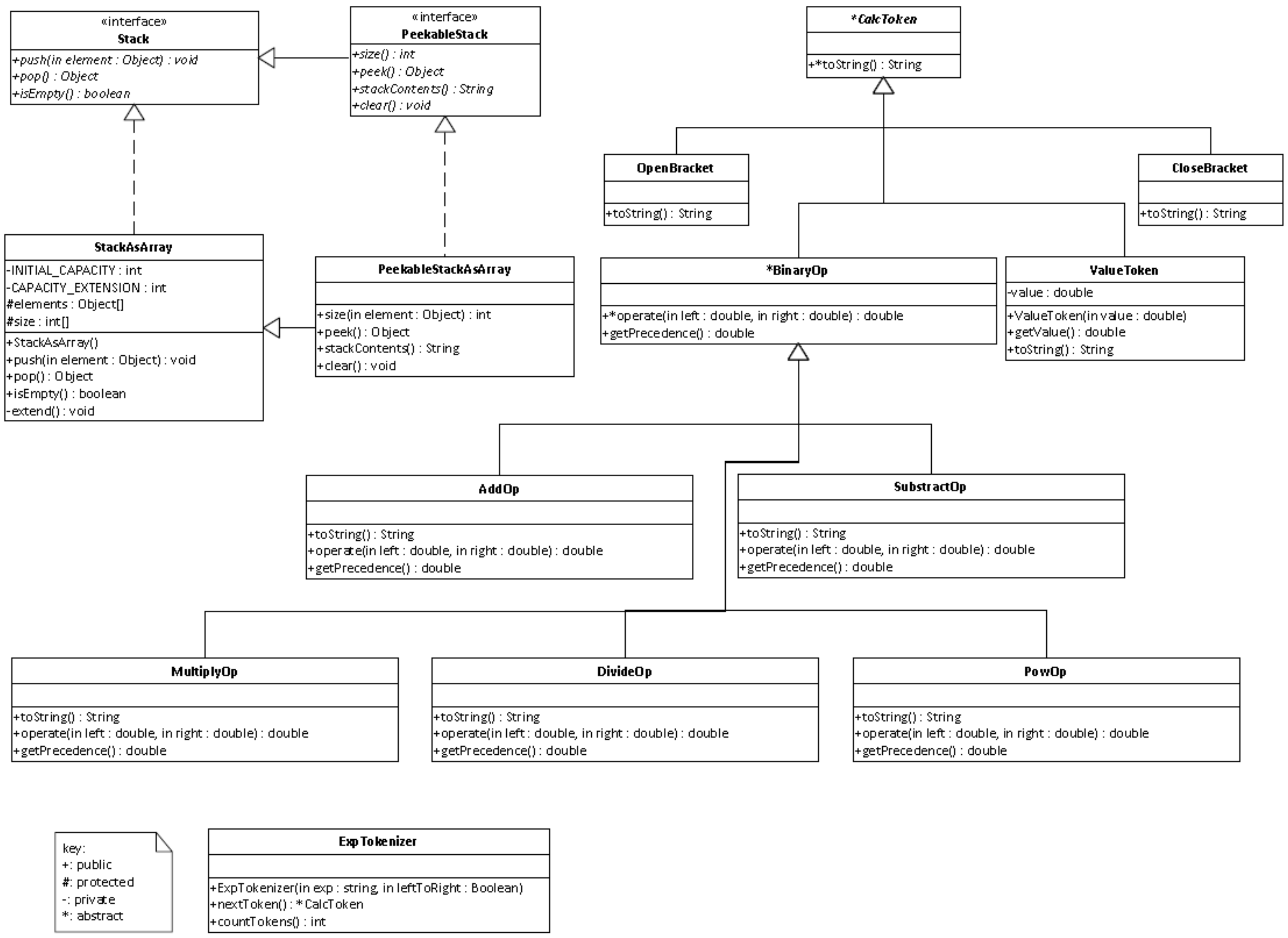
Where *i* (the total number of tests) should be at least **50**.

Style hints

The expectations 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 Parts 1 and 2, that describes the classes you will use and their interactions (extensions and implementations). The Calculator, PrefixCalculator and InfixCalculator classes are missing from this diagram.



«interface»
Stack
+push(in element : Object) : void
+pop() : Object
+isEmpty() : boolean
«interface»
PeekableStack
+size() : int
+peek() : Object
+stackContents() : String
+clear() : void
*CalcToken
+*toString() : String
OpenBracket
+toString() : String
CloseBracket
+toString() : String
StackAsArray
-INITIAL_CAPACITY : int
-CAPACITY_EXTENSION : int
#elements : Object[]
#size : int[]
+StackAsArray()
+push(in element : Object) : void
+pop() : Object
+isEmpty() : boolean
-extend() : void
PeekableStackAsArray
+size(in element : Object) : int
+peek() : Object
+stackContents() : String
+clear() : void
*BinaryOp
+*operate(in left : double, in right : double) : double
+getPrecedence() : double
ValueToken
-value : double
+ValueToken(in value : double)
+getValue() : double
+toString() : String
AddOp
+toString() : String
+operate(in left : double, in right : double) : double
+getPrecedence() : double
SubstractOp
+toString() : String
+operate(in left : double, in right : double) : double
+getPrecedence() : double
MultiplyOp
+toString() : String
+operate(in left : double, in right : double) : double
+getPrecedence() : double
DivideOp
+toString() : String
+operate(in left : double, in right : double) : double
+getPrecedence() : double
PowOp
+toString() : String
+operate(in left : double, in right : double) : double
+getPrecedence() : double
key:
+: public
#: protected
-: private
*: abstract
ExpTokenizer
+ExpTokenizer(in exp : string, in leftToRight : Boolean)
+nextToken() : *CalcToken
+countTokens() : int

What to submit for assignment 4

You need to submit both Part 1 and Part 2 (and the bonus if you decided to do it) is a single archive (zip) file.

Part 1 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, PrefixCalculator.java**.
- **Stack.java**, **CalcToken.java** and **BinaryOp.java**.

Part 2 java files:

- **StackAsArray.java**, **PeekableStackAsArray.java**.
- The token classes, in files **ValueToken.java**, **OpenBracket.java**, **CloseBracket.java**, **AddOp.java**, **SubtractOp.java**, **MultiplyOp.java**, **DivideOp.java**, and **PowOp.java**.
- **ExpTokenizer.java**.
- **Calculator.java, InfixCalculator.java**.
- **Stack.java**, **PeekableStack.java**, **CalcToken.java,** and **BinaryOp.java**.

Bonus Files:

- **ParseException.java.**

Note: You also need to submit the java files we provided which should remain complete and unmodified (such as Stack.java).

GOOD LUCK