AP assignment 1: Calculator

Sebastian Österlund

1 Assignment: Calculator

Your task is to write a calculator. The input should be in the form of:

```
INPUT: 1 + ( 2 - 3 )
OUTPUT: 0.0

INPUT: ( ( 1 + ( 2 - 3 ) + 5 ) / 2 ) ^ 2
OUTPUT: 6.25

INPUT: 5 + ( ( 1 + 2 ) * 4 ) - 3
OUTPUT: 14.0
```

Writing such a program seems like a very daunting task at first. Where to start? What about parentheses? Etc.

We will simplify the task by splitting the problem into three sub-problems. We start by (1) *tokenizing* the input. We then create a Reverse-Polish Notation (RPN) calculator (2), which is a relatively simple task. Finally, we (3) transform the "infix" input into RPN using the Shunting-yard algorithm, enabling us to use the calculator created in step 2, thus giving us a fully working calculator.

1.1 Tokenization

Tokenization is the act of breaking up a text stream into smaller pieces that have a syntactic meaning. Some streams can be tokenized in a really straight-forward way. For example, one may tokenize a text into words, which usually are separated by spaces and punctuation. Other streams may require a parser (as you will see in assignment 2). Luckily for us, tokenizing arithmetic expressions is relatively easy.

In the case of the calculator we have three different token types:

1. Numbers (e.g. 0, 1, 10001, 13212)
2. Operators (+, -, *, /, '∧')

3. Parentheses

Step one of this assignment consist of writing a method `TokenList readTokens(String input)` which parses all tokens from a given String, and returns a list of Tokens in the order in which they appear in the stream. We have given you an interface `Token` which you should implement. You can create one ore more classes that implement the interface `Token`.

As shown in the following example, making a distinction between the tokens is simple using the built-in `Scanner` class in Java.

```
static final String OPERATOR_TOKENS = "+-*/^";

boolean nextTokenIsDouble(String token) {
    Scanner in = new Scanner(token);
    return in.hasNextDouble();
}

public TokenList readTokens(String input) {
    Scanner in = new Scanner(input);
    TokenList result = ...
    // Loop
    ...
    String token = in.next();

    if (tokenIsDouble(token)) {
        result.add(parseNumber(token));
    } else if (tokenIsOperator(token)) {
        result.add(parseOperator(token));
    } else if (...) {

    } else {
        // Error
    }
    ...
    return result;
}

private NumberToken readNumber(Scanner in) {
    double number = ...;
    new NumberToken(number);
}
```

TASK: *Write a method readTokens() which takes as input a* `String` *and returns a list of tokens (*`TokenList`*) as output.*

1.2 Reverse-Polish Notation

RPN [1] is a mathematical notation, which in contrast to the normal infix notation, has the operator after its operands. Since, if an operator is encountered, its operands are already known, the result of the operator can immediately be computed. This allows RPN to be parsed easily by a computer. RPN assumes the notion of a *stack*, a linear datatype which allows for adding an element to the top, and removing the element on the top. You will have to implement this datastructure yourself.

The algorithm for evaluating RPN expressions is fairly straightforward:

Data: list of tokens
Result: Numerical value of the expression
while *there are input tokens left* **do**
  Read the next token from input;
  if *the token is a number* **then**
    Push it onto the stack;
  else if *the token is an operator* **then**
    Pop the top 2 numbers from the stack;
    Evaluate the operator, with the two numbers as arguments;
    Push the result of the operation back to the stack;
  end
end
if *there is only one value in the stack* **then**
  The value is the result of the expression.
else
  (Error) Invalid input, remaining tokens on stack.
end

Algorithm 1: The algorithm for evaluating an RPN expression

For example:

```
INPUT: 1 2 +
OUTPUT: 3

INPUT: 3 2 - 1 +
OUTPUT: 2
```

[1] For more info on RPN: `https://en.wikipedia.org/wiki/Reverse_Polish_notation`

TASK: *Write a method rpn() which takes as input a list of tokens (`TokenList`) and returns the result of the RPN expression as a double (`Double`).*
For example, the method may behave in the following manner:

```
// Stdin contains: 1 2 +
rpn(readTokens((new Scanner(System.in))).nextLine())
3.0
```

Some example code to get started with:

```
public Double rpn(TokenList tokens) {
    DoubleStack stack = ...;
}

private void performOperation(Token operator, DoubleStack stack) {
    double a = stack.pop();
    double b = stack.pop();

    if (operator.equals(PLUS_TOKEN)) {
        stack.append(a + b);
    } else if (...) {

    }
}
```

1.3 Shunting-yard algorithm

The Shunting-yard algorithm[2] [3] is an algorithm, first presented by the Dutch mathematician Edsger Dijkstra in 1961, which converts mathematical infix expressions into postfix (RPN) expressions.

By transforming our input into RPN, we can simply use the RPN calculator created in the last step to evaluate the desired "normal" arithmetic expressions!

The Shunting-yard algorithm can, in a simplified manner, be expressed as follows:

Data: list of tokens
Data: stack
Data: output list
Result: List of tokens in RPN
while *there are tokens to be read* **do**
  Read the next token from input;
  if *the token is a number* **then**
    Add it to the output list;
  else if *the token (o1) is an operator* **then**
    while *there is an operator (o2) on top of the stack with a higher or equal precedence* **do**
      pop o2 from the stack and add it to the output list;
    end
    push o1 onto the stack;
  end
  if *the token is a left parenthesis '('* **then**
    push it onto the stack;
  if *the token is a right parenthesis ')'* **then**
    while *the top of the stack is not a left parenthesis* **do**
      pop operator off the stack and add it to the output list.
    end
    Pop the left parenthesis from the stack;
end
while *there are operators on the stack* **do**
  pop operators off the stack and add to the output list.;
end

Algorithm 2: A simplified version of the Shunting-yard algorithm

[2]First described in `http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF`
[3]See also: `https://en.wikipedia.org/wiki/Shunting-yard_algorithm`

TASK: *Write a method shuntingYard() which takes as input a list of tokens (in infix notation) and returns a list of tokens in postfix (RPN) notation.*
For example, the method may behave in the following manner:

```
// Standard input contains: 1 + 3
shuntingYard(readTokens(new Scanner(System.in));
Result: ['1', '3', '+']
```

1.4 Combine everything!

You have now created the individual pieces necessary to evaluate calculator expressions! Now, the next step is writing the calculator program itself. It should wait for user input. On newline it should evaluate the expression and print the result on a new line.

Further information/ requirements:

- The skeleton for this assignment can be found here: `https://github.com/VU-Programming/AP-Calc-Skeleton`.
- If the input is invalid the program should not crash. You should make a best effort to print a relevant error message if the input is invalid.
- You are not allowed to use built-in arithmetic evaluation tools, nor are you allowed to use build-in datastructures like `ArrayList`.
- For the sake of simplicity, we assume that every token on the input is separated by exactly one whitespace.
- The input numbers may be in the interval

  $$]\texttt{Double.NEGATIVE\_INFINITY}, \texttt{Double.POSITIVE\_INFINITY}[$$

  .
- The maximum length of the input expression is unspecified.
- The methods `shuntingYard()` and `rpn()` should take a list of tokens (`TokenList` as input.
- Your main class should implement the interface `CalculatorInterface`.
- You may extend our interfaces and use your own interface-types as long as your conform to the `CalculatorInterface`.
- You do not have to take into consideration operator associativity. E.g. the correct output for 2^3^2 is 64.

- Expected operators:
  - '+' (addition)
  - '-' (subtraction)
  - '*' (multiplication)
  - '/' (division)
  - '∧' (exponentiation)

Appendix

```
/**
 * Token interface.
 * @elements characters of type char
 * @structure linear
 * @domain all rows of characters
 */
public interface Token {
    int     NUMBER_TYPE = 1,
            OPERATOR_TYPE = 2,
            PARENTHESIS_TYPE = 3;

    /**
     * @pre -
     * @post The value associated with this token has been returned a String.
     */
    String getValue();


    /**
     * @pre -
     * @post The type of this object, represented as an int, has been returned.
     */
    int getType();

    /**
     * @pre -
     * @post The precedence of the token, represented by an int,
     * has been returned. Higher int's signify a higher precedence.
     * If token type does not need a precedence,
     * the result of this method is -1.
     */
    int getPrecedence();

}

interface CalculatorInterface {

    /*
     * @param in String of tokens to be parsed
     * @return the list of arithmetic tokens from the String input
     */
    TokenList readTokens(String input);

    /*
     * @param tokens A list of tokens signifying an RPN expression.
     * @return The result of the RPN expression.
     */
```

```
    Double rpn(TokenList tokens);

    /*
     * @param tokens A list of tokens signifying an arithmetic expression.
     * @return The arithmetic expression tokens converted into
     * Reverse-Polish-Notation.
     */
    TokenList shuntingYard(TokenList tokens);
}

/**
 * @elements Tokens of the type Token
 * @structure linear
 * @domain All lists of tokens
 */
interface TokenList {

    /**
     * @pre -
     * @post The token 'token' has been added at the end of the TokenList,
     *       preserving the previous order.
     */
    void add(Token token);


    /**
     *
     * @pre index < size() and index >= 0
     * @post The element at location 'index' has been removed, preserving the previous order. The
     */
    void remove(int index);

    /**
     *
     * @param index The index to be set
     * @param token The value to set the element at location index to.
     * @pre   index < size();
     * @post  The element at location 'index' has the value 'token', preserving the previous orde
     */
    void set(int index, Token token);

    /**
     *
     * @param index The index of the element to be returned.
     * @return The element and index index.
     * @pre index < size();
     * @post The element at index 'index' has been returned.
     */
    Token get(int index);
```

```
    /**
     * @pre -
     * @post The number of elements in the list has been returned.
     */
    int size();


}

/**
 * @elements numbers of the type Double
 * @structure linear
 * @domain all rows of Doubles
 **/
public interface DoubleStack {

    /**
     * @pre -
     * @post Double element is now at the top of the stack.
     */
    void push(Double element);

    /**
     * @pre The stack is not empty
     * @post The element at the top of the stack is returned and deleted.
     */
    Double pop();

    /**
     * @pre The stack is not empty
     * @post The element at the top of the stack is returned.
     */
    Double top();

    /**
     * @post The number of elements on the stack is returned
     */
    int size();
}

/**
 * @elements Tokens of the type Token
 * @structure linear
 * @domain all rows of tokens
 **/
public interface TokenStack {

    /**
```

```
     * @pre -
     * @post token Token is now at the top of the stack.
     */
    void push(Token token);

    /**
     * @pre The stack is not empty
     * @post The token at the top of the stack is returned and deleted.
     */
    Token pop();

    /**
     * @pre The stack is not empty
     * @post The token at the top of the stack is returned.
     */
    Token top();

    /**
     * @pre -
     * @post The number of elements on the stack is returned
     */
    int size();
}
```