# Lab 3 - Evaluating infix arithmetic expressions

Like before, you are to continue doing "***Pair Programming***" with your assigned partner. Remember to switch roles halfway through lab (teams of 3 should switch every hour).

This is likely to be the MOST time-consuming lab this block. Please plan accordingly.

***Preliminaries***

In this lab you will use stacks and queues to implement a number of programs which, combined, evaluate infix integer arithmetic expressions like ***21 + 4 * 5***.

You will first implement a *tokenizer* program that takes a string representing a complete infix integer arithmetic expression and breaks it into pieces called tokens. In expression ***21 + 4 * 5***, the tokens are ***21***, ***+***, ***4***, ***** and ***5***. Once we have a sequence of tokens, we can convert the sequence into a form that a computer can more easily evaluate and then evaluate the sequence in that form.

For example, consider the previous infix integer arithmetic expression: ***21 + 4 * 5***. It is called an infix expression because the operators are placed between the operands: ***+*** between ***21*** and ***4*** and ***** between ***(21 + 4)*** and ***5***. The *tokenizer* separates the expression into the tokens ***21***, ***+***, ***4***, ***** and ***5*** and places them in a queue in that order.

Next, an *infix-to-postfix* converter program reorders the tokens so that the expression is in postfix order with the operators following the operands. For our example, this process results in **| *21* | *4* | *5* | * | + |** (meaning that multiplication of ***4*** by ***5*** should occur first, followed by the addition of ***21*** to the result).

If the infix expression contained parentheses, they are eliminated since they are not needed to evaluate the postfix expression.

Finally, a *postfix expression evaluator* program uses a stack to evaluate the expression:

```
      | 21 | 4  | 5 | * | + |
=     | 21 | 20 | + |
=     | 41 |
```

Note that the order of operators in the postfix expression and the order of operations in the evaluation process follow the same order that we would use to evaluate the original infix expression.

Take some time to work through an exercise using pen and paper; if you can't do this by hand then you SHOULD NOT even try to program it. Start by producing the output of the *tokenizer* program for input expression **344+6*90/ 3 + (9-3)%5**. Use this output as input to the *infix-to-postfix* program in order to produce its output. Finally, use the output from *the infix-to-postfix* program as input to the *postfix expression evaluator* program and find the expression value; you should get **525**.

**Show your complete work to your lab TA or instructor BEFORE you proceed.**

**PS: you are given complete classes or starter code for some classes/methods to be implemented in this lab; please complete the missing pieces WITHOUT changing the given code**

***Problem 1: Building and Testing the Token class hierarchy***

Rather than using strings to represent tokens, we will classify tokens into a class hierarchy of token types. The root of the hierarchy will be a *Token* interface. There are two basic types of tokens: arithmetic operators like ***+*** and ***** and operands like ***21*** and ***4***; in addition, there are two special types: left parentheses and right parentheses. The UML diagram shown next represents the complete hierarchy, as we will need it.

```
                <interface>
                    Token
                    + Type : static enum { OPERATOR, OPERAND, LEFT_PARENTHESIS, RIGHT_PARENTHESIS }
                    + getType() : Token.Type
                    + toString() : String
                              |
        +------------------------------------+-----------------+----------------------------+
        |                                    |                 |                            |
<abstract class>                         <class>           <class>                 <abstract class>
   Operator                          LeftParenthesis    RightParenthesis                Operand
+ abstract getPrecedence():int                                                 + abstract getValue():int
+ abstract evaluate(Operand, Operand):Operand                                               |
        |                                                                                   |
   +----+------+-----------+-----------+-----------+                                        |
   |           |           |           |           |                                        |
<class>     <class>     <class>     <class>     <class>                                  <class>
  Plus       Minus       Times       Divide     Remainder                            IntegerLiteral
```

Start by moving the *expressions* package from this lab folder into your *JavaPackages* folder (BUT keep classes *InfixPostfixDriver, TokenizerTest*, *InfixPostfixTest* and *PostfixEvaluatorTest* inside your lab folder).

Implement the hierarchy inside this *expressions* package (in *JavaPackages* folder) as you did for the *worker* hierarchy, but note that this hierarchy is really much simpler, even if it does contain more classes.

The complete *Token* interface is already given to you. You are also given the complete *Operator* and *Remainder* classes; use them as is and as templates to complete other similar classes in the hierarchy.

Most of the methods will just return a constant value or the value of an instance variable. Implement each method at the highest possible level in the hierarchy that doesn't require using *if* or *switch* statements; methods in *abstract* classes that are not implemented there must be declared *abstract*. Note the following:

*Operator* classes:

- method `getPrecedence` returns a precedence of 1 for the *Plus* and *Minus* classes, 2 for *Times*, *Divide* and *Remainder* (saying that operators *****, **/** and **%** have higher precedence over **+** and **-**)
- method `evaluate` performs the appropriate operation on the values of the two operands and then returns a new *IntegerLiteral* operand constructed on the result

*Operand* classes:

- Abstract class *Operand* contains only *abstract* methods; it is present to allow for more kinds of operands in the future
- One *IntegerLiteral* class constructor should take a *String* parameter and throw an *IllegalArgumentException* if the string can't be parsed into an `int` (use `Integer.parseInt`)
- A second constructor should take an `int`
- The class should have an `int` instance variable to hold the value of the literal

Leaf classes: (i.e., classes across the bottom of the UML diagram and the two parenthesis classes)

- Override the `toString` method for each of the leaf classes. For example, for class *Divide* `toString` method should return "/", while for *IntegerLiteral* it should return a *String* representing the integer value returned by its `getValue` method

Again, remember to use the given *Operator* and *Remainder* classes as templates to complete other similar classes in the hierarchy.

Include appropriate javadoc comments for each class. Include a package statement in all these class files and make sure you save all classes in the *expressions* package inside your *JavaPackages* folder.

This lab requires the use of the *Stack* and *Queue* data structures. We will use Java's version of both:

– Class *Stack<E>*: http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html
– Interface *Queue<E>*: http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html implemented by the *ArrayDeque<E>* class: http://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html

*Important note about casting*: There will be places where you need to treat a token specifically as a particular kind of token; you may, for instance, need to call the `getValue` method of an *Operand* token. If the variable type you're using is more general than the actual type of the token (probably *Token*) you may get a compiler error telling you that an object of that type (*Token*) doesn't have a method like that (`getValue`). If you know that the object is in fact a more specific subtype (*Operand*), you can cast it to the subtype and the compiler error will change to a warning about an unchecked cast. You will probably need to do that in this lab.

Be sure that all code in this problem of the lab compiles without errors before you go on. You may show this problem to the TA or lab instructor at this point.

***Problem 2: A Tokenizer for expressions – class Tokenizer (the challenge begins here!!!!)***

Complete the supplied *Tokenizer* class inside the *expressions* package using the following guidelines.

The `makeToken` method gets a string that should represent one of the tokens in the hierarchy above. You are already given some of the code and will need to complete this method after understanding what it does as explained next.

Method `makeToken` determines which type of token it is, calls the appropriate token constructor and returns a new token (e.g., `new Plus()` token object is created and returned for **+**). I recommend using a *switch-case* statement defined over the first character of the parameter string. All tokens except for integer literals should be made up of a single character, so the method should throw an exception if the string length is not one unless the first character is a digit. (We're not allowing negative literals in this lab; use (0 - 5) to get -5.) The method should also throw an exception if the string doesn't match any of the token types.

The `parseString` method gets a string and scans it for tokens. Since tokens are not necessarily separated by whitespace, it is necessary to write the *tokenizer* by hand rather than using Java's *StringTokenizer* class. Again, you are already given some of the code and will need to complete this method after understanding what it does based on the algorithm provided next:

Algorithm for method **`parseString(tokensStr: String) : Queue<Token>`**

```
start with an empty queue of tokens (ArrayDeque<Token>)
set i <-- 0
while i < tokensStr.length do
    if character at location i in tokensStr is an operator or a
       parenthesis then
```

```
            call makeToken on that character
            add the resulting token to the queue
            increment i
        else if character at location i in tokensStr is a digit then
            loop to find the next position j <= tokensStr.length that is not
                 a digit
            call makeToken on a substring of tokensStr from location i to j-1
                 inclusive
            add the resulting token to the queue
            set i <-- j
        else if character at location i in tokensStr is a whitespace then
            increment i
        else [ it's an illegal character at the start of a token ]
            throw an IllegalArgumentException
        end if
    end while
    return the queue
```

Note that you can use method `Character.isDigit(char c)` to test if `char c` is a digit; similarly, method `Character.isWhitespace` tests for white spaces. Also note that if you use method `substring` method to extract substrings from a given string, the character at the terminal position in the `substring` method is NOT included in the result (i.e., `str.substring(i,j)` returns a string with the characters in positions `i` to `j-1`).

When you have completed the *Tokenizer* class so that it compiles without errors, use the provided *TokenizerTest* JUnit test class (available in your lab folder) to test that the classes you have implemented for this lab so far work and interact properly. This test class is complete and tests the *Tokenizer* class, which, in turn, uses all the others.

***Problem 3: Evaluating infix integer arithmetic expressions***

In this problem, you will implement a program that inputs a string representing a simple integer arithmetic expression and outputs the value of that expression. It begins with the *Tokenizer* you built earlier and adds two more parts, an *infix-to-postfix expression converter* and a *postfix expression evaluator*.

***Part 3.1: class InfixToPostfix***

Complete class *InfixToPostfix* inside your *expressions* package in your *JavaPackages* folder. It includes the following three methods each given to with some starting code for you to complete based on the algorithms shown below:

```
//takes an infixQueue and returns the equivalent postfixQueue
public static Queue<Token> convert(Queue<Token> infixQueue);

private static void processRightParenthesis(Stack<Token> opStack,
                                            Queue<Token> postfixQueue);

private static void processOperator(Stack<Token> opStack,
                                    Queue<Token> postfixQueue,
                                    Operator op);
```

The `convert` method is the only *public* method of the *InfixToPostfix* class; it is the method that performs the actual conversion by processing tokens from the `infixQueue` one-by-one, moving operands (*IntegerLiteral* tokens, in this case) directly into a queue called `postfixQueue` and pushing operators and left parentheses onto a stack called `opStack`.

Operators require special handling, since they must be moved to the queue after their second operand, and we must also account for operator precedence before we push the operator onto the `opStack` stack; when we encounter a right parenthesis, we have to process any operators pushed onto the stack since the last left parenthesis. The complete algorithm for method `convert` is as follows:

**`convert(infixQueue : Queue<Token>) : Queue<Token>`**

```
if infixQueue is empty then
    throw an IllegalArgumentException with message
            "Empty infix expression"
end if
set postfixQueue <-- empty Queue<Token>
set opStack <-- empty Stack<Token>
while infixQueue is not empty do
    dequeue nextTok from infixQueue
    if nextTok is an operand (i.e. nextTok instanceof Operand) then
        enqueue nextTok to postfixQueue
    else if nextTok is a left parenthesis then
        push nextTok onto opStack
    else if nextTok is a right parenthesis then
        call processRightParenthesis
    else [ nextTok is an operator ]
        call processOperator
    end if
end while
while opStack is not empty do
    pop nextOp from opStack
    if nextOp is a left parenthesis then
          throw an IllegalArgumentException with message
                  "Unmatched left parenthesis"
    end if
    enqueue nextOp to postfixQueue
end while
return postfixQueue
```

To make the code for the `convert` method easier to implement and understand, we broke out two parts of the algorithm into separate sub-algorithms, namely, `processRightParenthesis` and `processOperator`. These methods are called when we encounter a right parenthesis or an operator in the infix expression queue, respectively. Since they only make sense when they're called by `convert`, we make them *private* methods.

When we encounter a right parenthesis, all we have to do is pop all the operands from `opStack` and enqueue them into `postfixQueue` until we find the matching left parenthesis on the stack; if we reach the bottom of the stack before finding a matching left parenthesis, the right parenthesis must be unmatched and we throw an exception. The algorithm for the `processRightParenthesis` method is as follows:

The complete algorithm for method **`processRightParenthesis(Stack<Token> opStack,`**
**`Queue<Token> postfixQueue) : void`**

```
while opStack is not empty and top of opStack is not a left parenthesis
do
    pop an element from opStack and enqueue it to postfixQueue
end while
if opStack is empty then
    throw an IllegalArgumentException with message
            "Unmatched right parenthesis"
end if
pop the left parenthesis from opStack and discard
```

When we encounter an operator, we need to push it onto `opStack`. First, however, we need to check any operators already on the stack; as long as there is an operator on the top of the stack with greater or equal precedence to the new operator, we need to pop that operator and enqueue it to `postfixQueue`. Once we reach the bottom of the stack or find a left parenthesis on the top of the stack, we can push the new operator and return. The algorithm for the `processOperator` method is shown below.

The complete algorithm for method **`processOperator(opStack : Stack<Token>,`**
**`postfixQueue : Queue<Token>,`**
**`op : Operator) : void`**

```
if opStack is not empty then
    set topTok <-- top element of opStack [ peek without popping ]
end if
while opStack is not empty and topTok is not a left parenthesis and
      precedence of parameter op is <= precedence of topTok do
    pop opStack and enqueue it to postfixQueue
    if opStack is not empty then
        set topTok <-- top element of opStack [ without popping ]
    end if
end while
push op onto opStack
```

When done, use the provided *InfixPostfixTest.java* test program (available in your lab folder) to test your *InfixToPostfix* program. Additionally, I am providing you with a driver program which requests the user to input a string representing an infix arithmetic expression, calls `Tokenizer.parseString` on the input, calls `InfixToPostfix.convert` on the tokenizer output queue and then displays the resulting infix queue using the iterator. Use may use this program to test your *InfixToPostfix* class more thoroughly. Verify that it correctly converts well-formed infix expressions to the equivalent postfix expression and that it throws an exception on mismatched parentheses. You should predict your expected outcome for each test and then check whether you got the expected result.

### ***Part 3.2: Evaluating a postfix expression – class PostfixEvaluator***

In this part, you'll create and implement a class with a `static method` that takes a queue of tokens in postfix order and returns an `int` that is the value of the expression. You'll then use a JUnit test class and a driver program to test the evaluation.

<u>Create a new class</u> called `PostfixEvaluator` inside your *expressions* package in your *JavaPackages* folder with the following method:

**public static int** evaluate(Queue<Token> postfixQueue):

This evaluate method works by processing tokens from the postfixQueue, again one by one, in this case pushing operands (not operators) onto a stack. Whenever we encounter an operator, there should be two operands for that operator already on the stack. We pop the two operands (note that the right operand would pop BEFORE the left operand because we're using a stack), perform the operation and push the result onto the stack. When the whole queue is processed, the stack should contain a single *IntegerLiteral* representing the value of the expression. The algorithm for evaluate is as follows:

**evaluate(postfixQueue : Queue<Token>) : int**

```
    if postfixQueue is empty then
        throw an IllegalArgumentException with message
                "Empty postfix expression"
    end if
    set opStack <-- an empty Stack<Operand>
    while postfixQueue is not empty do
        dequeue nextTok from postfixQueue
        if nextTok is an operand then
            push nextTok onto opStack
        else [ nextTok is an operator ]
            if opStack is empty then
                throw an IllegalArgumentException with message
                        "Operator with no operands"
            end if
            pop rightOperand from opStack
            if opStack is empty then
                throw an IllegalArgumentException with message
                        "Operator with only one operand"
            end if
            pop leftOperand from opStack
            call evaluate on nextTok with parameters leftOperand and
                                                     rightOperand
            push result back onto opStack
        end if
    end while
    pop result from opStack
    if opStack is not empty then
       throw an IllegalArgumentException with message
             "Too many operands"
    end if
    return result
```

To test your work, use the *PostfixEvaluatorTest.java* test program. Again, you should also use the provided *InfixPostfixDriver.java* program to test your code more thoroughly. The driver program calls the PostfixEvaluator.evaluate method and displays the result but first you need to uncomment the last two lines in the driver program. I recommend you keep the display of the postfix expressions for completeness and to aid in error checking. Test your program on a wide variety of infix expressions (both well-formed and ill-formed) to verify that it correctly evaluates the expressions.