420-202-RE	Assignment 4 Stacks, Queues, Exceptions	Due Date: April 28, 2017

1 Objectives

- Practice using the stack data type and its methods.
- Practice using the queue data type and its methods.
- Learn about how Java Implements stacks and queues.
- Understand prefix, infix, and postfix expression formats.
- Use a queue to store the tokens in an arithmetic expression.
- Use stacks to evaluate arithmetic expressions.
- Practice Exception handling in Java.

2 Representaion of Arithmetic Expressions

Arithmetic expressions involving binary operators (such as `+`, `-`, `*`, `/`, etc.) may be written in a variety of notations depending on where an operator is placed relative to its operands:

Expression in English	prefix	infix	postfix
2 plus 3	`+ 2 3`	`2 + 3`	`2 3 +`
2 times 3 plus 4	`+ * 2 3 4`	`( 2 * 3 ) + 4`	`2 3 * 4 +`
2 times the sum of 3 and 4	`* 2 + 3 4`	`2 * ( 3 + 4 )`	`2 3 4 + *`

Notice that in all three notations, the operands appear in the same order, but the operators do not.

To avoid ambiguity, the *infix* notation requires the use of parentheses depending on operator associativity and precedence rules. For example, if the operator `*` has higher precedence than the operator `+`, then the parentheses in `( 2 * 3 ) + 4` are optional, whereas the parentheses in `2 * ( 3 + 4 )` are mandatory.

Unlike *infix* expressions, however, *prefix* and *postfix* expressions *do not* require the use of cumbersome parentheses, precedence rules, or rules for association, making it convenient for programmers to implement evaluation of customary infix expressions.

The algorithms for converting to and evaluating postfix and prefix expressions have similar complexity, but postfix expressions are somewhat simpler to evaluate on computers.

3 Traditional Infix Evaluation Algorithms

Traditional algorithms for evaluating infix expressions involve three basic steps:

1. Extract the "operand" and "operator" tokens from an infix expression into a *queue* of string tokens.

   For example, the infix expression `(22 + 3)` is tokenized into the queue `["(", "22", "+", "3", ")"]`, with `"("` at the front of the queue.

2. Convert a given queue of infix tokens into an equivalent queue of postfix tokens.

   For example, convert the queue `["(", "22", "+", "3", ")"]` of infix tokens to this queue `["22", "3", "+"]` of postfix tokens.

3. Evaluate the queue of postfix tokens from step 2.

   For example, evaluate the postfix queue `["22", "3", "+"]` of tokens to `25`.

4 A Direct Infix Evaluation Algorithm

A less traditional way to evaluate an infix expression is to combine steps 2 and 3 of the traditional approach into one step as follows. We start with two empty stacks, one to hold intermediate integer values and the other to hold the operators. As we *dequeue* (remove) the tokens from the infix queue, we place the number tokens on the value stack immediately; however, before we place an operator token on the operator stack we must ensure that its precedence is strictly greater (>) than that of the operator at the top of the operator stack. Otherwise, we repeatedly apply the following algorithm until the precedence condition is satisfied or until the operator stack becomes empty.

Algorithm applyStackTopOperator()

1 Pop an operator `op` from the operator stack,
2 Pop an operand `v2` from the value stack,
3 Pop an operand `v1` from the value stack,
4 Apply the operation `result = v1 op v2`,
5 Push `result` onto the value stack.

The idea is to perform an operation as soon as the operator and operands in that operation are known. Using this algorithm, we effectively eliminate one operator `op` and two operands, `v1` and `v2`, replacing them by a single operand `result`. That is, each application of the algorithm above reduces the number of tokens by two, and when used repeatedly under the precedence condition mentioned above, it will reduce the original expression to only one operand (*the answer*) at the top of the value stack. Here is the algorithm just described in details:

5 Algorithm evaluate(infixQueue)

Input : infixQueue, a queue of strings representing the tokens in an infix expression.
Output: The *integer* value of the infix expression
Assert : Each token in the input queue represents either one of the operators `"+"`, `"-"`, `"*"`, `"/"`, and `"%"`, an integer operand, or a left `"("` or right `")"` parenthesis.
Throws: Nothing yet, but on your to-do list!

```
1  Prepare an empty stack valStack to store Integer objects (the operands).
2  Prepare an empty stack opStack to store Character objects (the operators).
3  Prepare an iterator qIter to scan the infixQueue
4  while qIter.hasNext() do
5      Let token = qIter.next()
6      if token is an operand then
7          Push token onto valStack
8      else if token is a left parenthesis then
9          Push token onto opStack
10     else if token is a right parenthesis then
11         while opStack is not empty and its top element is not a left parenthesis
           do
12             applyStackTopOperator()
13         Pop top element from opStack (which must be a left parenthesis) and
           discard it.
14     else if token is one of "+", "-", "*", "/", and "%", operators then
15         while opStack is not empty and the token's precedence is less than or
           equal to that of the operator at the top of opStack do
16             applyStackTopOperator()
17         push token onto opStack
18     else // invalid token in the infix queue
19         Throw an IllegalArgumentException exception with a massage
           informing the user about what has gone wrong here.
20 while opStack is not empty do
21     applyStackTopOperator()
22 Pop and return the top value of valStack, which must be the only value in
   valStack
```

Algorithm 1: An Infix Evaluation Algorithm

Here is an example of tracing the algorithm above supplied with the input infix expression

$$(1+2)*(3-4)/((5-6)*(7+8)/5)$$

Algorithm Step	Current token	Unprocessed tokens	Operator stack bottom ··· top	Value stack bottom ··· top
		(1+2)*(3-4)/((5-6)*(7+8)/5)		
9	(	1+2)*(3-4)/((5-6)*(7+8)/5)	(	
7	1	+2)*(3-4)/((5-6)*(7+8)/5)	(	1
15	+	2)*(3-4)/((5-6)*(7+8)/5)	(+	1
7	2	)*(3-4)/((5-6)*(7+8)/5)	(+	1 2
11	)	*(3-4)/((5-6)*(7+8)/5)		3
15	*	(3-4)/((5-6)*(7+8)/5)	*	3
9	(	3-4)/((5-6)*(7+8)/5)	* (	3
7	3	-4)/((5-6)*(7+8)/5)	* (	3 3
15	-	4)/((5-6)*(7+8)/5)	* (-	3 3
7	4	)/((5-6)*(7+8)/5)	* (-	3 3 4
11	)	/((5-6)*(7+8)/5)	*	3 -1
15	/	((5-6)*(7+8)/5)	/	-3
9	(	(5-6)*(7+8)/5)	/ (	-3
9	(	5-6)*(7+8)/5)	/ ((	-3
7	5	-6)*(7+8)/5)	/ ((	-3 5
15	-	6)*(7+8)/5)	/ ((-	-3 5
7	6	)*(7+8)/5)	/ ((-	-3 5 6
11	)	*(7+8)/5)	/ (	-3 -1
15	*	(7+8)/5)	/ (*	-3 -1
9	(	7+8)/5)	/ (* (	-3 -1
7	7	+8)/5)	/ (* (	-3 -1 7
15	+	8)/5)	/ (* (+	-3 -1 7
7	8	)/5)	/ (* (+	-3 -1 7 8
11	)	/5)	/ (*	-3 -1 15
15	/	5)	/ (/	-3 -15
7	5	)	/ (/	-3 -15 5
11	)		/	-3 -3
21				1 ← answer

6 Your Task

Implement the following class diagrams, except the methods highlighted in yellow, for which sample source code is provided. (Recall that `static` methods are underlined in UML.)

Expression	A concrete class
– `infix : String`	Stores a user supplied infix expression.
+ `Expression(expr : String ) :`	Initializes `infix` to `expr`.
+ `Expression() :`	Equivalent to `this("0")`
+ `reset(infix : String) : void`	Resets `this.infix` to `infix`.
– `Tokenize() : Queue<String>`	Tokenizes `infix` into a queue of `String` tokens, and returns that queue. See source code on page 8.
– `evaluate(q : Queue<String>) : Integer`	Evaluates the expression whose infix tokens are stored in a queue represented by `q`. Implements Algorithm 1 on page 3, throwing exceptions where runtime sources of errors may potentially exist in the algorithm. Details appear on page 10.
+ `eval() : Integer`	Evaluates and returns the value of *this* `infix`. Delegates evaluation task to the `evaluate` method above, but handles any possible exceptions thrown by `evaluate`. See sample source code on page 9.
- `precedence(op : String) : int`	Returns the precedence of the supplied operator `op`. It returns `0` for `"("`, `"1"` for `"+"` and `"-"`, and `"2"` for `"*"`, `"/"`, or `"%"`; otherwise, it throws an `IllegalArgumentException` if `op` is invalid.
– `applyStackTopOperator( opStack : Stack<String>, valStack : Stack<Integer>) : void`	Implements the algorithm on page 2. Presents four potential source of runtime errors; specifics are listed on page 10.
+ `toString() : String`	Returns a neatly formated version of `infix`. Specifically, it returns a string formed by the tokens in `infix` with a single space after each tokens. For example if `infix` is `"(1 +2)+(3/2*5)"`, then it returns `"( 1 + 2 ) + ( 3 / 2 * 5 )"`.

As you can see the bulk of your task involves implementing Algorithm 1 in the `evaluate` Method.

You are supplied with complete sample source code for `Tokenize()` on page 8, `eval()` on page 9, as well as the driver code on page 6

7 Test Driver Program

```
public static void main(String[] args)
{
   Scanner keyboardScan = new Scanner(System.in);
   System.out.println("This program calculates infix expressions with integer operands");
   System.out.println("Use only one of / % * + - binary operators in your expressions");
   System.out.println("Enter bye to quit");

   Expression expr = new Expression(); // expr = 0
   while (true)
   {
      System.out.print("? ");
      String infix = keyboardScan.nextLine();
      if (infix.equalsIgnoreCase("bye"))
      {
         break;
      }
      expr.reset(infix);
      Integer result = expr.eval();
      System.out.println(": " + expr + "\n= " + result);
   }
}
```

A Sample Program Run

```
This program calculates infix expressions with integer operands
Use only one of / % * + - binary operators in your expressions
Enter bye to quit

? (1+2  )*(  3-4)/((5-6)   *(7+8  )/  5)
: ( 1 + 2 ) * ( 3 - 4 ) / ( ( 5 - 6 ) * ( 7 + 8 ) / 5 )
= 1

? (1  +2)+(3/2*5)
: ( 1 + 2 ) + ( 3 / 2 * 5 )
= 8

? (((((1)+2)+3)+4)+5)
: ( ( ( ( ( 1 ) + 2 ) + 3 ) + 4 ) + 5 )
= 15

? (((((1)+2)*3)+4)*5)
: ( ( ( ( ( 1 ) + 2 ) * 3 ) + 4 ) * 5 )
= 65

? (1) + (2)
: ( 1 ) + ( 2 )
= 3
```

A Sample Program Run

```
? (1)
: ( 1 )
= 1

? ()
Invalid infix expression: something wrong; perhaps missing an operand
: ( )
= null

? (1+2
Invalid infix expression: Missing right parenthesis
: ( 1 + 2
= null

? 1+2)
Invalid infix expression: Missing left parenthesis
: 1 + 2 )
= null

? 1) + (2
Invalid infix expression: Missing left parenthesis
: 1 ) + ( 2
= null

? 2/(1-1)
Invalid arithmetic operation: Division by zero not defined
: 2 / ( 1 - 1 )
= null

? 100%1
: 100 % 1
= 0

? 100/1
: 100 / 1
= 100

? 2 + +
Invalid infix expression: Missing operand
: 2 + +
= null

? (1) (2) (3)
Invalid infix expression: too many operands
: ( 1 ) ( 2 ) ( 3 )
= null

? bye
```

8 String Tokenization

Using an `StringTokenizer` object, the `tokenize` method tokenizes `infix` into a queue of `String` tokens, and returns that queue. To simplify the process, it works with a copy of `infix` with all the spaces removed; as an added bonus, it tokenizes an expression such as `"1 2 3+4"` into [`"123"`, `"+"`, `"4"`].

Infix Tokenizer

```
1  private Queue<String> Tokenize()
2  {
3     // prepare a queue of string items
4     Queue<String> queue = new LinkedList<String>();
5     // remove all the spaces in infix
6     String infixWithNoSpaces = infix.replaceAll("[\t\n ]", "");
7     // define the characters allowed in an infix expression
8     String delimiters = "*/+-()%";
9     // create a StringTokenizer, telling it to return the delimiters as tokens.
10    StringTokenizer tokenizer =
11           new StringTokenizer(infixWithNoSpaces, delimiters, true);
12    while (tokenizer.hasMoreTokens())
13    {
14       String token = tokenizer.nextToken();
15       queue.add(token); // queue up the tokens
16    }
17
18    return queue; // return our queue of infix tokens
19 }
```

9 public Integer eval()

Evaluates and returns the value of `infix`. It delegates the actual evaluation task to the `static` method `evaluate`, but completely handles any possible exceptions thrown by `evaluate`.

Exception Handling

```
public Integer eval()
{
   Integer result = null;
   try
   {
      Queue<String> infixQueue = Tokenize(); // tokenize
      result = evaluate(infixQueue); // evaluate
      return result; // done
   } // unless exceptions were thrown by Tokenize() and/or evaluate(...) above
   catch(NumberFormatException e)
   {
      System.out.println("Operand must be an integer: " + e.getMessage());
   }
   catch(ArithmeticException e)
   {
      System.out.println("Invalid arithmetic operation: " + e.getMessage());
   }
   catch(IllegalArgumentException e)
   {
      System.out.println("Invalid infix expression: " + e.getMessage());
   }
   catch (Exception e) // catch any other exceptions
   {
      System.out.println("Unexpected error: " + e.getMessage());
   }
   // returns null to indicate that evaluation was aborted
   // due to one of the exceptions above
   return null;
}
```

10 Expect the Unexpected

There are a number of points in the algorithms on pages 2 and 3 where potential sources of runtime errors may exist. Such "hot spots" and what to do about them are listed in the tables below:

Exceptional Points in algorithm `applyStackTopOperator()`

Step	What if	Throw this
1	`opStack` is empty?	`IllegalArgumentException("Missing oprator")`
1	`op` is `"("`?	`IllegalArgumentException("Missing right parenthesis")`
2	`valStack` is empty?	`IllegalArgumentException("Missing operand")`
3	`valStack` is empty?	`IllegalArgumentException("Missing operand")`
4	`op` is is `"/"` or `"%"`, and `v2` is zero?	`ArithmeticException("Division by zero not defined")`

Exceptional Points in algorithm `evaluate(...)`

Step	What if	Throw this
13	opStack is empty or opStack's top is not a `"("`?	`IllegalArgumentException("Missing left parenthesis")`
22	`valStack` is empty?	`IllegalArgumentException("something wrong; perhaps missing an operand")`
22	`valStack` has more than one element?	`IllegalArgumentException("too many operands");`

11 Stacks: LIFO Collections

Recall that a stack is a collection in which elements are removed in in the reverse of the order in which they were added. A stack is like a stack of trays in a cafeteria; people add a tray to the top of the stack and remove a tray from the top of the stack. As such, stacks are referred to as *last-in, first-out* (**LIFO**) data structures.

Here is and example of how to create and process a stack of strings in Java:

```
Stack<String> stack = new Stack<>();

stack.push("1");
stack.push("2");
stack.push("3");

//look at top element ("3"), without taking it off the stack.
String topElement = stack.peek();

String e3 = stack.pop(); //returns and removes the string "3" is from the top of the stack.
String e2 = stack.pop(); //returns and removes the string "2" is from the top of the stack.
String e1 = stack.pop(); //returns and removes the string "1" is from the top of the stack.
```

The `Stack<E>` class in the `java.util` package extends class `Vector`, providing the following five operations that allow a vector to be treated as a stack:

The `Stack<E>` class

Method	Behavior
`E push(E e)`	Pushes an element `e` onto the top of this stack.
`E pop()`	Removes the element at the top of this stack and returns that element.
`E peek()`	Returns, but does not remove, the element at the top of this stack.
`boolean empty()`	Tests if this stack is empty.

For more information see:

`https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html`

`https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html`

12 Queues: FIFO collections

Recall that a queue is a collection in which elements are removed in the same order in which they were added. A queue is like a line at a grocery store, where customers (elements) add themselves at the back of the line and are served from the front of the line. As such, queues are referred to as *first-in, first-out* (**FIFO**) data structures.

Here is and example of how to create and process a queue of integers in Java:

```
Queue<Integer> intQue = new LinkedList<>();

intQue.add(11); intQue.add(22); intQue.add(33);

//look at front element (11), without removing it.
Integer firstInline = intQue.element();

//remove and return front element (11).
firstInline = intQue.remove();

//access via Iterator
Iterator<Integer> iterator = intQue.iterator();
while (iterator.hasNext()) {
   Integer value = iterator.next();
   // process value
}
//access via enhanced for-loop
for (Integer value : intQue){ // process value
}
```

The `Queue<E>` interface in the `java.util` package extends the `Collection<E>` interface, adding primary queue operations, each in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation).

The `Queue<E>` interface

Throws exception	Returns special value	Behavior
`boolean add(E e)`	`boolean offer(E e)`	adds an element `e` to the end of the queue
`E remove()`	`E poll()`	removes an element from the front of the queue
`E element()`	`E peek()`	Returns, but does not remove, the element at the front of the queue

For more information see:

https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Evaluation

Evaluation Criteria	
Correctness of execution of your program	60%
Proper use of required Java concepts	20%
Java API documentation style	10%
Comments on nontrivial steps in code, Choice of meaningful variable names, Indentation and readability of program	10%