Assignment # 2

Can be solved individually or in groups of 2 students

Create a zip file containing the required deliverables with all files containing the students names and numbers (one submission per team)

Q1) Creating the stack classes

You are required to write your own generic stack implementation in Java that you will use in questions 2 and 3. **(30 marks)**

a. Create a stack interface Stack<T> that uses a generic type T and has the following abstract methods:

- **isEmpty, isFull**, **peek (returns T)**, **pop(returns T),** void **push(T), void clear()** to clear the contents of the stack, and int **size (returns the number of elements in the stack).**

- **void reverse()** reverses the contents of the stack (in your implementation you can use an additional local stack in your method to help you with the reverse operation.)

- **void display()** prints out the contents of the stack (Hint: you can obtain the contents of your stack elements using toString())

b. Write your two stack implementations (ArrayStack and ListStack) using the stack interface.

c. Your stack implementations must have all instance variables as private.

d. Both implementations must a constructor that initializes the capacity of the stack. For the ListStack , the stack is full when the size reaches the initial capacity.

e. You must also implement *public String toString()* in both classes. It returns a String representation of the stack contents.

c. Create a simple main class QuestionOne that creates two stacks of Integers of capacity=20 one that uses the ArrayStack and the other uses the ListStack implementations. Push 20 values in each: in the first (1,2,3,…,20) and in the second (20, 19,..,13,.., 1). Call the following methods for both stacks in the given order: display, reverse, display, peek, pop, pop, reverse, size, isFull, isEmpty, display, clear, display, isEmpty.

Deliverable: folder Q1

Stack.java

ArrayStack.java

ListStack.java

QuestionOne.java

Q2) Simple calculator (35 marks):

You are to design a simple calculator using the ArrayStack implementation in Q1 to perform additions, subtractions, multiplications and divisions. The user may enter an arithmetic expression in infix using numbers (0 to 9), parentheses and arithmetic operations (+, -, *, /). The first step to do so is to create a utility class **MyCalculator** that will have the following methods:

a) Input of an expression and checking Balanced Parenthesis:

public static Boolean isBalanced(String expression);

This is a static method that will read a string representing an infix mathematical expression with parentheses from left to right and decide whether the brackets are balanced or not. To discover whether a string is balanced each character is read in turn. The character is categorized as an opening parenthesis, a closing parenthesis, or another type of character. Values of the third category are ignored for now. When a value of the first category is encountered, the corresponding close parenthesis is stored in the stack. For example, when a "(" is read, the character ")" is pushed on the stack. When a "{" is encountered, the character pushed is "}". The topmost element of the stack is therefore the closing value we expect to see in a well balanced expression.

When a closing character is encountered, it is compared to the topmost item in the stack. If they match, the top of the stack is popped and execution continues with the next character. If they do not match an error is reported. An error is also reported if a closing character is read and the stack is empty. If the stack is empty when the end of the expression is reached then the expression is well balanced.

For this application a parenthesis can be one of the following:

- parantheses: ( )
- curly braces: { }
- square brackets: [ ]
- angle brackets: < >

These must be defined as constants in the class.

b) Infix to Postfix conversion

Write a method that converts Infix expressions to postfix expressions using your stack implementation: public static String infixToPostfix(String infix)

c) Evaluating a Postfix expression

Write a method thatevaluates a postfix expression: public static **double evaluate(string postfix);**

d) Your main class **QuestionTwo**

To evaluate your methods create the class **QuestionTwo** It must contain a **public static void main(String[] args)** method to run your code. This main method should ask the user to input

one infix expression per line until the user types “q” or “Q”. After every input, it should first test if the expression has balanced paranthesis and tells the user if the expression is balanced or not.

If the expression is balanced it will convert it into a postfix expression, displays the expression and then evaluates it and outputs the results and then ask for the next input.

If the expression is not balanced it tells the user that and asks for a new input.

Some rules:

-You should assume no whitespace in the infix expression provided by the user. For example, your code should work for inputs like **“2+(3×2)”.** The postfix that you will print, however, should have tokens separated by whitespace. For example, “3×21” should have the postfix

“3 21 ×”, and not “321×”.

Here is a suggested part of your main that you can use

```
import java.util.Scanner;
public class QuestionTwo{
    public static void main(String[] args)
    {
        Scanner calcScan = new Scanner(System.in);
        Boolean finished=false;
        while (!finished)
        {
            System.out.println("Enter a postfix expression or q to quit:
");
            String expression = calcScan.nextLine();
            System.out.println(expression);
            expression.trim();//omits leading and trailing whitepsaces
            System.out.println(expression);
            if (expression.equalsIgnoreCase("q"))
            {
                finished=true;
            } else
            {//use your MyCalculator methods here
            }

        }
    }
}
```

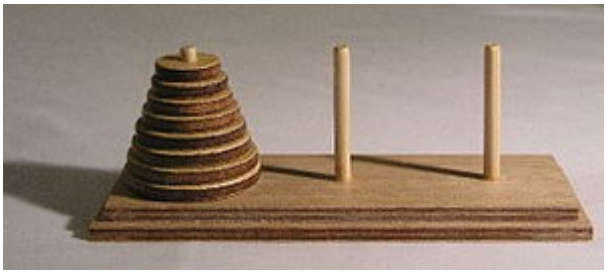
Deliverables:

- MyCalculator.java

QuestionTwo.java

Q3) Towers of Hanoi (35 marks):

You are required to design an algorithm which solves the **Tower of Hanoi** puzzle (see the figure below). You have an array of stacks (use the ListStack implementation) named **rods** with size **3**. All three of the stacks are in the same size which is **n.**



At the beginning: rods[0] (first stack) is full of discs marked with positive integers from **1** to **n** where **1** is at the top and **n** is at the bottom (i.e., you need three stacks of Integers with the first initialized to store the n Integers). The other two stacks are empty.

The goal of the puzzle: is to move all these disks from **rods**[0] to the third rod, **rods**[2], with the help of rods[1], without breaking any rules of the Tower of Hanoi puzzle.

The rules of the game are:

- Only one disc may be moved at a time.
- Each move consists of taking the top (smallest) disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Hints: To move a stack of k discs from rod A to rod B, we first move k-1 discs from A to C, then move the remaining (kth) disc from rod A to B, and then finally move all k-1 discs on C to B. Also, if you closely examine the puzzle you will see that there is at most one legal move between any two rods.

Deliverables:

`TowersofHanoi.java`

- A class that has the **rods**, and the size **n** of the storage of the rods. It has a constructor that takes the **towercapacity**(i.e., **n)** and creates the new array rods,(i.e., `Stack<Integer>[] rods = new ArrayStack[3];`). It then initializes the three stacks (each with capacity n) with the first rod containing the **n** integers.
- Implements Boolean legalMove(int a, int b) returns true if it is legal to move a disc from rod a to rod b. (rods are referred to as rod 0, rod 1, rod 2)
- Implements Boolean move(int a, int b) that moves a disc from a to b after ensuring it is a legal move and prints out a sentence " disc x moved from rod y to rod z" with x,y, z representing the right number for the disc and the rods. It returns false if the move is not legal.
- Implements Boolean **move**(int m, int a, int b, int c) moves m discs from tower **a** to tower b using tower c as an intermediate storage. It prints out all the movements of the discs as they appear. It returns false if the moves are not legal. This could be because a doesn t have m discs, rod b has exceeded capacity or you are trying to store a large disc on top of a smaller one.
- Implement the method void **showTowerStates**() prints out the contents of the rods.

- Method void **solvegame()** solves the game from the initial state where all discs are stored in rods[0].
- Bonus (5 marks): method void solvecurrent() solves the game at any state for the rods.

PlaytowerofHanoi.java

- Your main program that asks the user how many discs, **n, (up to 6 discs) ,** and then asks the user if he/she wants to play the game or see the solution.
- The program should create a new **TowersofHanoi** instance of the game, shows the towers state before the game starts.
- If the user selects to play then you repeatedly ask him to select a move of one disk from one Rod to another. You need to check if a move is legal or not and prompt the user to re-enter a legal move if it is not. The program should tell the user that he/she has lost the game after a maximum of 2^n-1 legal moves (if bonus used: and solves the game from the current state showing the user the solution). If the user solves the game the program should congratulate him/her and terminate.

- If the user chooses to see the solution then the program should solve it and display the steps.
- **<u>You cannot use recursion</u>** in solving this problem (i.e., no method can call itself).
- Hint: There is always one legal movement between any two rods. Here is a **possible** way to solve the towers of Hanoi problem:

```
1. Calculate the total number of moves required i.e. "pow(2, n)-1" here

2. If number of disks (i.e. n) is even then interchange destination
   rod and auxiliary rod.
3. for i = 1 to total number of moves:
     if i%3 == 1:
          legal movement of top disk between source rod and
        destination rod
     if i%3 == 2:
          legal movement top disk between source pole and
        auxiliary rod
     if i%3 == 0:
        legal movement top disk between auxiliary rod
        and destination rod
```