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 stack0, 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.)

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

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

b. Write your stack implementations (ArrayStack<T> and ListStack<T>) using the stack interface.

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

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

e. Create a simple main class **QuestionOne** that creates two stacks of Integers of capacity 20. The first stack uses the ArrayStack<Integer> and the other uses the ListStack<Integer>. 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<T> 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 the four arithmetic operations (+, -, *, /) only. Any other input is to be wrong and prints an error message.

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 ((operand operation operand)) mathematical expression with parentheses from left to right and decide whether the brackets are balanced or not. For example, (5*{7+22} is not valid, but (5*{7+22}) is valid.

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 (display a message about the error and return false). 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.

Some hints:

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

-parantheses: ( )　　　　- angle brackets: < >

-curly braces: { }　　　　-square brackets: [ ]

These must be defined as constants in the class.

2) Use a stack of characters to store your open brakets (new ArrayStack<Character>(100)).

3) to read a character from the string at position i, use **charAt** (e.g., expression.charAt(i)).

4) the length of a string can be retrived using length() (e.g., expression.length()).

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)

The method only takes balanced expressions with balanced parenthesis. The return is a string that has the Postfix expression of the input string.

c) Evaluating a Postfix expression

Write a method that evaluates 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 parenthesis 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:

-To simplify your task, you can 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 the operands and operations 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:

Deliverables:

- MyCalculator.java

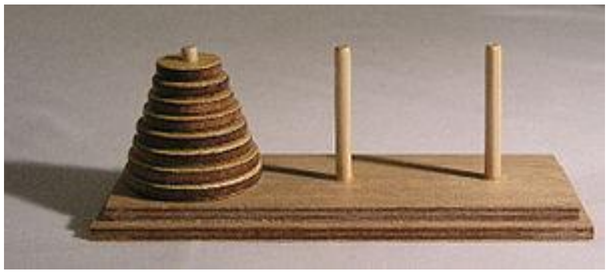
-QuestionTwo.java (most part is provided to you below)

```
import java.util.Scanner;
public class QuestionTwo{
    public static void main(String[] args)
    {   //ready to get input from the user
        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
          if (expression.equalsIgnoreCase("q"))
            {finished=true;
            } else
            { System.out.println(expression);
             if(MyCalculator.isBalanced(expression)){
             System.out.println(expression +" is balanced");
             String PostfixExpression = MyCalculator.infixToPostfix(expression);
             System.out.println(PostfixExpression +" is  infix representation");
             System.out.println(MyCalculator.evaluate(PostfixExpression) +"is  its value");
              }else
            System.out.println(expression +"is not balanced");
          }
        }
        calcScan.close();
    }
}
```

Q3) Towers of Hanoi (35 marks):

You are required to design an algorithm which solves the **Tower of Hanoi** puzzle (see the figure below). The objective of the puzzle is to move **n** discs from one tower to another, but with some rules in place as described below.

In your algorithm, the towers will be represented with an array of stacks called **rods** with size **3 (three towers)**. All three of the stacks have the same capacity which is **n.**


.

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

The goal of the puzzle: is to move all these discs 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) disc from one of the rods and sliding it onto another rod, on top of the other discs 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 (**k**th) 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 array **rods (an array of stacks)**, and the size **n** of the storage of the rods (i.e., Stack<Integer>[] rods). We will use the ArrayStack<Integer> implem entation to create instances of the stack.
- It has a constructor that takes the **towercapacity**(i.e., **n)** and creates the new array r ods,(rods= **new** ArrayStack<Integer>[3];). It then initializes the three stacks (each with capacity n: rods[0] = **new** ArrayStack<Integer>(n)) with the first rod containing the **n** integers and the remaining two are empty stacks
- 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 i t 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 mo ve is not legal.
- Implements Boolean **move**(int m, int a, int b, int c) moves m discs from tower **a** to t ower b using tower c as an intermediate storage. It prints out all the movements of t he discs as they appear. It returns false if the moves are not legal. This could be beca

use a doesn t have m discs, rod b has exceeded capacity or you are trying to store a l arge disc on top of a smaller one.

- Implement the method void **showTowerStates**() prints out the contents of the rods. For example:

  First tower = 0,1,2,3,4,5
  second tower =
  Third tower =
  Should be printed before the game is started when n=6

- Method void **solvegame()** solves the game from the initial state where all discs are s tored in rods[0].
- **Optional challenge**: develop a 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 a demo solution.
- The program should create a **new TowersofHanoi(n)** instance of the game, shows the towers state before the game starts. (a print out of what is inside the rods)

- 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 optional challenge used: 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(from the initial state) 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,
        NumMoves= pow(2, n)-1,
    2. If n is odd
       for i = 1 to NumMoves:
            if i%3 == 1: legal movement of top disc between source rod and
destination rod
            if i%3 == 2: legal movement top disc between source rod and
middle rod
           if i%3 == 0: legal movement top disc between middle rod  and
destination rod

     else If n is even
       for i = 1 to NumMoves:
            if i%3 == 1: legal movement top disc between source pole and
middle rod
           if i%3 == 2: legal movement of top disc between source rod and
destination rod
           if i%3 == 0: legal movement top disc between middle rod  and
destination rod
```