Guidelines:

- **Make sure to submit your files to Google Classroom before the deadline,** otherwise, your work won't be considered for grading.
- **Submit only the java files as separate files (i.e. not as zipped files).**
- **Create a class for every program named based on the question name then write all the necessary methods and/or the main method**

PostfixEvaluator:

In this exercise, you are expected to convert the pseudo code in Lesson 5 Slide 46 into Java code.

- Create a class called PostfixEvaluator.java that will contain all the methods.
- Implement the method **private static boolean isOperator(char op);** which returns **true** if the string *op* is one of the allowed operators: +, -, /, *, **false** otherwise
- Implement the method **private static double evalOp(char op, Deque<Double> stack);** which applies the operation *op* on the first two values at the top of the *stack*.
- Implement the method **public static double eval(String postfix)**; that applies the pseudo code in slide 46. You will need to use the two methods implemented before.
- **eval** method should catch any kind of errors in the postfix expression (i.e. throw an Exception with a message). If no errors are detected, it returns the result of the expression. Check the test cases in Slide 47 of Lesson 5.
- Test you code in a main method.

InfixToPostfix:

In this exercise, you are expected to convert the pseudo code in Lesson 5 Slides 53 and 54 into Java code.

- Create a class called InfixtoPostfix.java that will contain all the methods. The class should contain two private data fields: Deque<Character> operatorStack, and StringBuilder postfix which will hold the result of the program.
- Implement the method **private static int precedence(char op);** which returns a number that represents the precedence of the operators (*op*) as follows:
  - 1 for + and -
  - 2 for * and /
- Implement the method **public static String convertToPostfix(String infix)**; that applies the pseudo code in slide 53. You will need to use the **isOperator** method from PostfixEvaluator exercise, and a method **processOperator** that is explained below. The method will return the resulting postfix expression as a String, or throw an exception is there is any syntax error.

- Implement the method **private static void processOperator(char op);** which applies the pseudo code in slide 54. You will need to use the **precedence** method.
- Test you code in a main method.

DecimalToBinary:

Create a class DecimalToBinary that contains the method convert. The method uses a stack to convert a decimal number into binary.

convert will insert the remainder of dividing the decimal number by 2 into a stack, then divide the decimal by 2. This process will be repeated while the decimal number is greater than 0.

The binary digits are then popped from the stack one at a time and appended to the right-hand end of the string. The binary string is then returned.

Method prototype: **public static String convert(int decimal);**

Test you code by running the following main method:

```
public static void main(String[] args) {
      for (int i = 0; i < 20; i++)
            System.out.println(DecimalToBinary.convert(i));
}
```

PostfixtoInfix

In this exercise, you are expected to convert postfix expressions into infix in a class called **PostfixToInfix.java**.

This is done exactly as postfix evaluation, but instead of calculating the result of an operator, you are expected to create a string of the form "**(lhs op rhs)**", where the lhs is the left hand side operand, rhs is the right hand side operand, and op is the operator. This resulting string is then pushed into the stack. The final result would be at the top of the stack.

Your program should

- Read a postfix expression from the user.
- Convert the postfix expression into an infix expression and display it to the user.
- Display an appropriate message if the postfix expression is not valid.
- Operators to be considered are +, -, *, /, %.

Design your class just like the **PostfixEvaluator** class.

Assignment Exercise : InfixToPostfixParens:

Implement InfixToPostfixParens.java class which provides a method **convert** that converts an infix expression ***with parenthesis*** into postfix expression. This is done by considering that the parenthesis is an operator with least precedence compared to all other mathematical operators.

Besides, the method **processOperator** needs to be changed as follows:

- Push each opening parenthesis onto the stack as soon as it is scanned. Note that the open parenthesis could be the first token.
- When a closing parenthesis is scanned, pop all operators up to and including the matching opening parenthesis, inserting all operators popped (except for the opening parenthesis) in the postfix string.
- A closing parenthesis is considered processed when an opening parenthesis is popped from the stack and the closing parenthesis is not placed on the stack.

Non-balanced parathesis can be detected when an opening parenthesis could not be popped from the stack or an opening parenthesis remains in the stack after processing all tokens.

Bonus Question: Infix Compiler:

You can combine the algorithms for converting between infix to postfix and for evaluating postfix to evaluate an infix expression directly. To do so you need **two stacks**: one to contain operators and the other to contain operands.

When an operand is encountered, it is pushed onto the operand stack. When an operator is encountered, it is processed as described in the infix to postfix algorithm. When an operator is popped off the operator stack, it is processed as described in the postfix evaluation algorithm: The top two operands are popped off the operand stack, the operation is performed, and the result is pushed back onto the operand stack.

Design and write a program to evaluate infix expressions directly using this combined algorithm.