Class for further usage resultlist.add(tempoperandsholder.tostring()); tempoperan...

class PostFixConverterUtil { private String inputExpression; private Deque operatorStack = new ArrayDeque <>(); private List resultList = new ArrayList <>(); PostFixConverterUtil(String expression) { //Remove empty characters inputExpression = expression. replaceAll(" ",""); //Convert given expr to resultList convertExpressionToPostFix(); } /** * Converts the given expression to resultList so that it is easier to evaluated * This is a rough adaptation of the shunting yard algorithm */ private void convertExpressionToPostFix() { //Temp variable to hold value of the number StringBuilder tempOperandsHolder = new StringBuilder(); //Iterate the expression as a character array for(int i = 0; i != inputExpression.

length(); ++i) { if(Character. isDigit(inputExpression. charAt(i))) { /* * If we encounter a single digit, then keep on looking for other digits until * we encounter an operator i. e digits can be 1, 10, 100 etc., */ tempOperandsHolder. append(inputExpression. charAt(i)); while((i+1) != inputExpression.

length() && (Character. isDigit(inputExpression. charAt(i+1)) || inputExpression. charAt(i+1) == '.

')) { tempOperandsHolder. append(inputExpression. charAt(++i)); } /* If we exit out of the above loop and reach here, then we have encountered either an operator or end * of expression, so we put temp into the resultList and clear temp for further usage */ resultList. add(tempOperandsHolder.

toString()); tempOperandsHolder. setLength(0); } //Tokens other than digits i. e braces and operators are handled by pushing it //onto the operatorStack

else { pushToOpsStack(inputExpression. charAt(i)); } } //Finally move operators to resultList while(! operatorStack. isEmpty()) resultList.

add(operatorStack. removeLast(). toString()); } private void pushToOpsStack(char input) { if(operatorStack. isEmpty() || input == ‘(‘) operatorStack. addLast(input); else { if(input == ‘)’) { while(! operatorStack.

getLast(). equals(‘(‘)) { resultList. add(operatorStack. removeLast(). toString()); } operatorStack.

removeLast(); } else { if(operatorStack. getLast(). equals(‘(‘)) operatorStack. addLast(input); else { while(! operatorStack. isEmpty() && ! operatorStack. getLast(). equals(‘(‘) && calculatePrecedence(input) <= calculatePrecedence(operatorStack.

getLast())) { resultList. add(operatorStack. removeLast(). toString()); } operatorStack. addLast(input); } } } } /** * Returns precedence for entered character.

* and / have higher precedence followed by + and – * @param op * @return precedence */ private int calculatePrecedence(char op) { if (op == ‘+’ || op == ‘-‘) return 1; else if (op == ‘*’ || op == ‘/’) return 2; else return 0; } List getPostfixAsList() { return resultList; }}