과목 : 자료구조(가반)

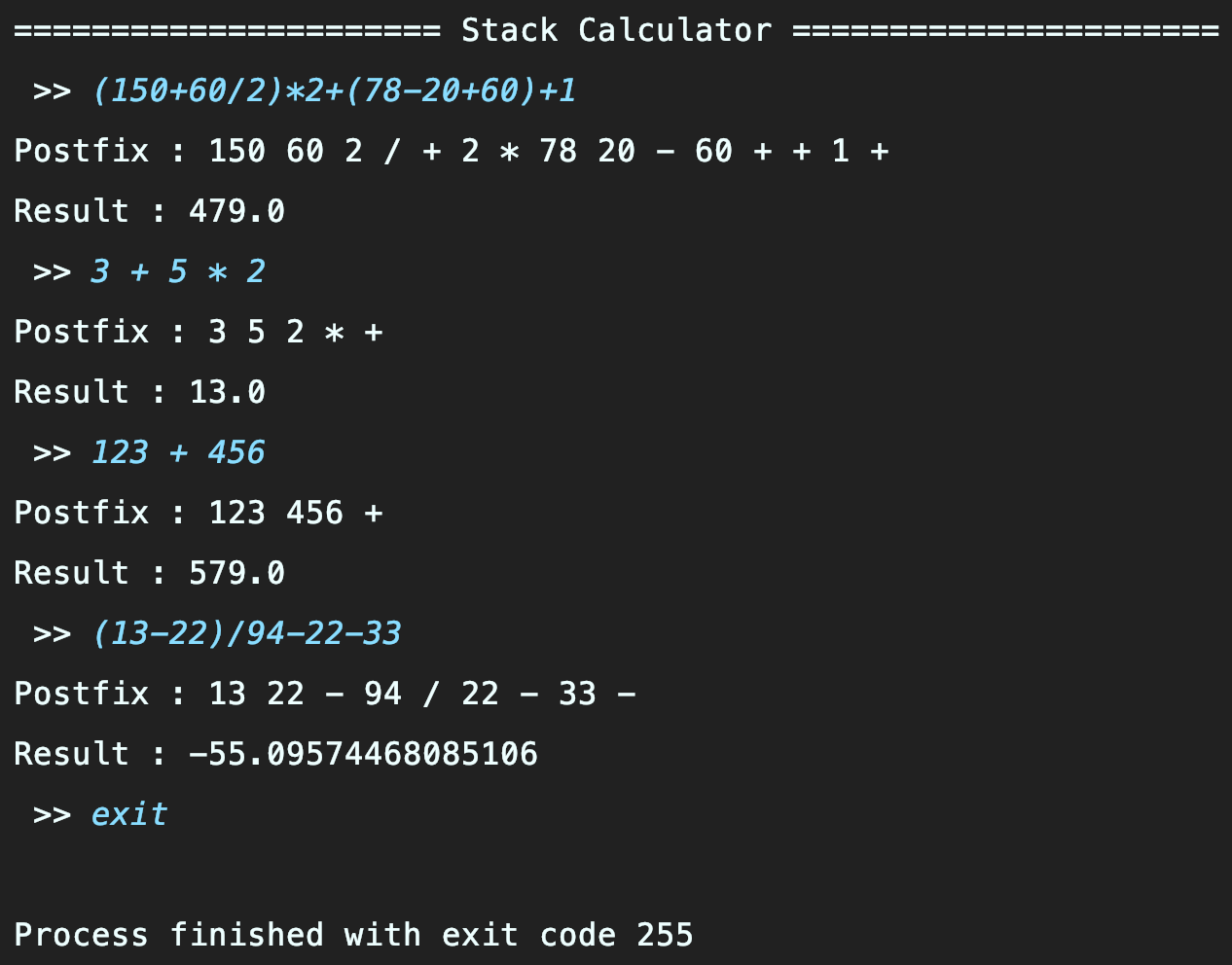
교수 : 신용태 교수

이름 : 김병준

학번 : 20162448

1. 수식 계산

* Result



1. 구문 검사

* Result

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

스크린샷이(가) 표시된 사진

자동 생성된 설명

1. 소스코드 (StackCalculator.java)

*import* java.util.Scanner;

*abstract class* Stack {

*int* top = -1;

*int* size = 100;

*boolean* isEmpty() {

*return* top == -1;

}

}

*class* Operator *extends* Stack {

*private char*[] stack = *new char*[size];

*void* push(*char* op) {

stack[++top] = op;

}

*char* pop() {

*return* stack[top--];

}

}

*class* Operand *extends* Stack {

*private double*[] stack = *new double*[size];

*void* push(*double* num) {

stack[++top] = num;

}

*double* pop() {

*return* stack[top--];

}

}

*public class* StackCalculator {

*public static void* main(String[] args) *throws* Exception {

System.out.println("====================== Stack Calculator ======================");

*while* (*true*) {

Scanner scanner = *new* Scanner(System.in);

System.out.print(" >> ");

String expression = scanner.nextLine();

*if* (expression.equals("exit")) System.*exit*(-1);

expression = *Preprocessor*(expression);

*if* (*syntaxCondition*(expression)) {

String post\_expression = *transPostfix*(expression);

*double* result = *calculate*(post\_expression);

System.out.println("Postfix : " + post\_expression);

System.out.println("Result : " + result);

}

}

}

*private static* String Preprocessor(String expression) {

String[] split\_expression = expression.split(" ");

StringBuilder fit\_expression = *new* StringBuilder();

*for*(String exp : split\_expression) {

fit\_expression.append(exp);

}

*return* fit\_expression.toString();

}

*private static* String transPostfix(String expression) {

StringBuilder post\_expression = *new* StringBuilder();

Operator operator = *new* Operator();

*for* (*int* i = 0; i < expression.length(); i++) {

*char* exp = expression.charAt(i);

*switch* (exp) {

*case* '(':

*case* '{':

*case* '[':

operator.push(exp);

*break*;

*case* ')':

*case* '}':

*case* ']':

*while* (*true*) {

*char* op = operator.pop();

*if* (!(op == '(') && !(op == '{') && !(op == '[')) {

post\_expression.append(op).append(" ");

} *else* {

*break*;

}

}

*break*;

*case* '+':

*case* '-':

*while* (*true*) {

*if* (operator.isEmpty()) {

*break*;

}

*char* op = operator.pop();

*if* (op == '+' || op == '-' || op == '\*' || op == '/') {

post\_expression.append(op).append(" ");

} *else* {

operator.push(op);

*break*;

}

}

operator.push(exp);

*break*;

*case* '\*':

*case* '/':

*while* (*true*) {

*if* (operator.isEmpty()) {

*break*;

}

*char* op = operator.pop();

*if* (op == '\*' || op == '/') {

post\_expression.append(op).append(" ");

} *else* {

operator.push(op);

*break*;

}

}

operator.push(exp);

*break*;

*default*:

post\_expression.append(exp);

*if*(expression.length() != i + 1) {

*char* op = expression.charAt(i + 1);

*if* (op == '+' || op == '-' || op == '\*' || op == '/'

|| op == ')' || op == '}' || op == ']') {

post\_expression.append(" ");

}

}

*break*;

}

}

*while* (!operator.isEmpty()) {

post\_expression.append(" ").append(operator.pop());

}

*return* post\_expression.toString();

}

*private static* Double calculate(String post\_expression) {

Operand operand = *new* Operand();

String[] expressionArr = post\_expression.split(" ");

*for* (String exp : expressionArr) {

*try* {

*double* number = Double.*parseDouble*(exp);

operand.push(number);

} *catch* (NumberFormatException e) {

*double* op1 = operand.pop();

*double* op2 = operand.pop();

*switch* (exp) {

*case* "+":

operand.push(op2 + op1);

*break*;

*case* "-":

operand.push(op2 - op1);

*break*;

*case* "\*":

operand.push(op2 \* op1);

*break*;

*case* "/":

operand.push(op2 / op1);

*break*;

}

}

}

*return* operand.pop();

}

*private static boolean* syntaxCondition(String expression) *throws* Exception {

Operator operator = *new* Operator();

*int* number\_count = 0;

*for* (*int* i = 0; i < expression.length(); i++) {

*char* exp = expression.charAt(i);

*switch* (exp) {

*case* '+':

*case* '-':

*case* '\*':

*case* '/':

*char* op = expression.charAt(i + 1);

*if* (op == '+' || op == '-' || op == '\*' || op == '/') {

*throw new* Exception("syntaxCondition(): Operator error");

}

*break*;

*case* '(':

*case* '{':

*case* '[':

operator.push(exp);

*break*;

*case* ')':

*if* (operator.isEmpty()) {

*throw new* Exception("syntaxCondition(): Syntax error");

} *else* {

*if* (!(operator.pop() == '(')) {

*throw new* Exception("syntaxCondition(): Syntax error");

}

}

*break*;

*case* '}':

*if* (operator.isEmpty()) {

*throw new* Exception("syntaxCondition(): Syntax error");

} *else* {

*if* (!(operator.pop() == '{')) {

*throw new* Exception("syntaxCondition(): Syntax error");

}

}

*break*;

*case* ']':

*if* (operator.isEmpty()) {

*throw new* Exception("syntaxCondition(): Syntax error");

} *else* {

*if* (!(operator.pop() == '[')) {

*throw new* Exception("syntaxCondition(): Syntax error");

}

}

*break*;

*default*:

*if*(exp < '0' || exp > '9')

*throw new* Exception("syntaxCondition(): There is not number or unavailable character in formular");

number\_count++;

*break*;

}

}

*if* (number\_count == 0 || number\_count == 1) {

*throw new* Exception("syntaxCondition(): There is no number or unavailable in formular");

}

*while* (!operator.isEmpty()) {

*char* op = operator.pop();

*if* (op == '+' || op == '-' || op == '\*' || op == '/'

|| op == '(' || op == '{' || op == '[') {

*throw new* Exception("syntaxCondition(): '" + op + "'Syntax error");

}

}

*return true*;

}

}