LAB 7

PRACTICE

PRACTICE 7.1;

Simple implementation of stack using Java

CODE:

**package** PRACTICE;

**public** **class** EX7\_1 {

**private** **int** arr[];

**private** **int** top;

**private** **int** capacity;

// Constructor to initialize the stack

EX7\_1(**int** size){

arr = **new** **int**[size];

capacity = size;

top = -1;

}

// Utility function to add an element `x` to the stack

**public** **void** push(**int** x) {

**if**(isFull()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

System.***out***.println("Inserting " + x);

arr[++top] = x;

}

// Utility function to pop a top element from the stack

**public** **int** pop() {

// check for stack underflow

**if**(isEmpty()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

System.***out***.println("Removing " + peek());

// decrease stack size by 1 and (optionally) return the popped element

**return** arr[top--];

}

// Utility function to return the top element of the stack

**public** **int** peek() {

**if**(!isEmpty()) {

**return** arr[top];

}

**else** {

System.*exit*(-1);

}

**return** -1;

}

// Utility function to return the size of the stack

**public** **int** size() {

**return** top + 1;

}

// Utility function to check if the stack is empty or not

**public** **boolean** isEmpty() {

**return** top == -1; // or return size() == 0;

}

// Utility function to check if the stack is full or not

**public** **boolean** isFull() {

**return** top == capacity -1; // or return size() == capacity;

}

}

**class** Main{

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

EX7\_1 stack = **new** EX7\_1(3);

stack.push(1); // inserting 1 in the stack

stack.push(2); // inserting 2 in the stack

stack.pop(); // removing the top element (2)

stack.pop(); // removing the top element (1)

stack.push(3); // inserting 3 in the stack

System.***out***.println("The top element is " + stack.peek());

System.***out***.println("The size of stack is " + stack.size());

stack.pop(); // removing the top element (3)

// check if the stack is empty

**if**(stack.isEmpty()) {

System.***out***.println("The stack is empty");

}

**else** {

System.***out***.println("The stack is not empty");

}

}

}

ALGORITHM

The algorithm is to implement basic function of stack

STEP1[initialize] arr[] as int, top as int and capacity as int

STEP2[constructor] set arr[] := int [size] , capacity as size and top:=-1

Push(x)

1. [Stack already filled?]
2. IF isFull(), then print OVERFLOW and return
3. Set Top:= Top + 1 [increase TOP by 1]
4. Set arr[Top]:=x [insert x in new Top position]
5. Return

Pop()

1. [Stack has an empty to be removed]
2. IF isEmpty() , then print UNDERFLOW and return -1
3. Print REMOVING peek()
4. Set arr[Top] := Top-1 [Decrease Top by -1
5. Return

Peek()

1. IF isEmpty(), then return Top
2. ELSE return -1

Size()

1. Return Top:=Top+1

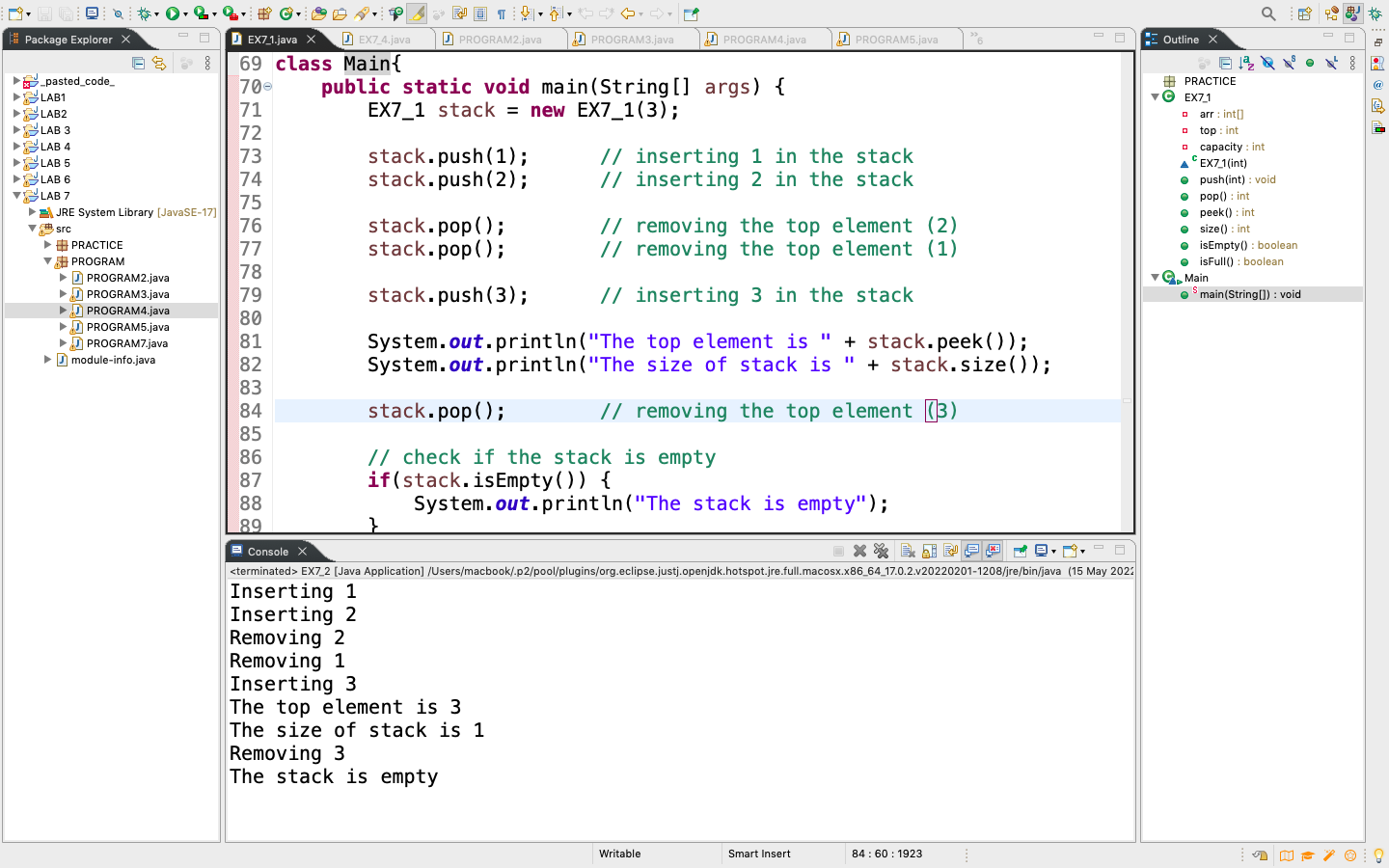
isEmpty()

1. Return Top=-1

isFull()

1. Retrun Top:=capacity -1

OUTPUT:



PRACTICE 7.2:

Using iterative program to reverse a string

CODE:

**package** PRACTICE;

**import** java.util.Stack;

**public** **class** EX7\_2 {

// Iterative function to reverse a given string.

**public** **static** **void** reverse(**char**[] c) {

// create an empty stack of characters

Stack<Character> stack = **new** Stack<>();

// push each character of the given string into thestack

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

stack.push(c[i]);

}

//start from index 0

**int** k=0;

// pop characters from the stack until it is empty

**while**(!stack.empty()) {

// assign each popped character back to the input string

c[k++] = stack.pop();

}

}

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

String str = "Welcome to usman intitute";

**char**[] c = str.toCharArray();

*reverse*(c);

str = **new** String(c);

System.***out***.print("Reverse of the given string is " + str);

}

}

ALGORITHM

The algorithm is to reverse the string using stack and the argument c used as array of char

Reverse( c )

STEP1[initialize] create stack of type Character name stack

STEP2[initialize] initialize I as iterating variable

STEP3[loop] repeat step 4 and 5 i<c.lenght

STEP4[increment] i=i+1

STEP5 push(c[i])

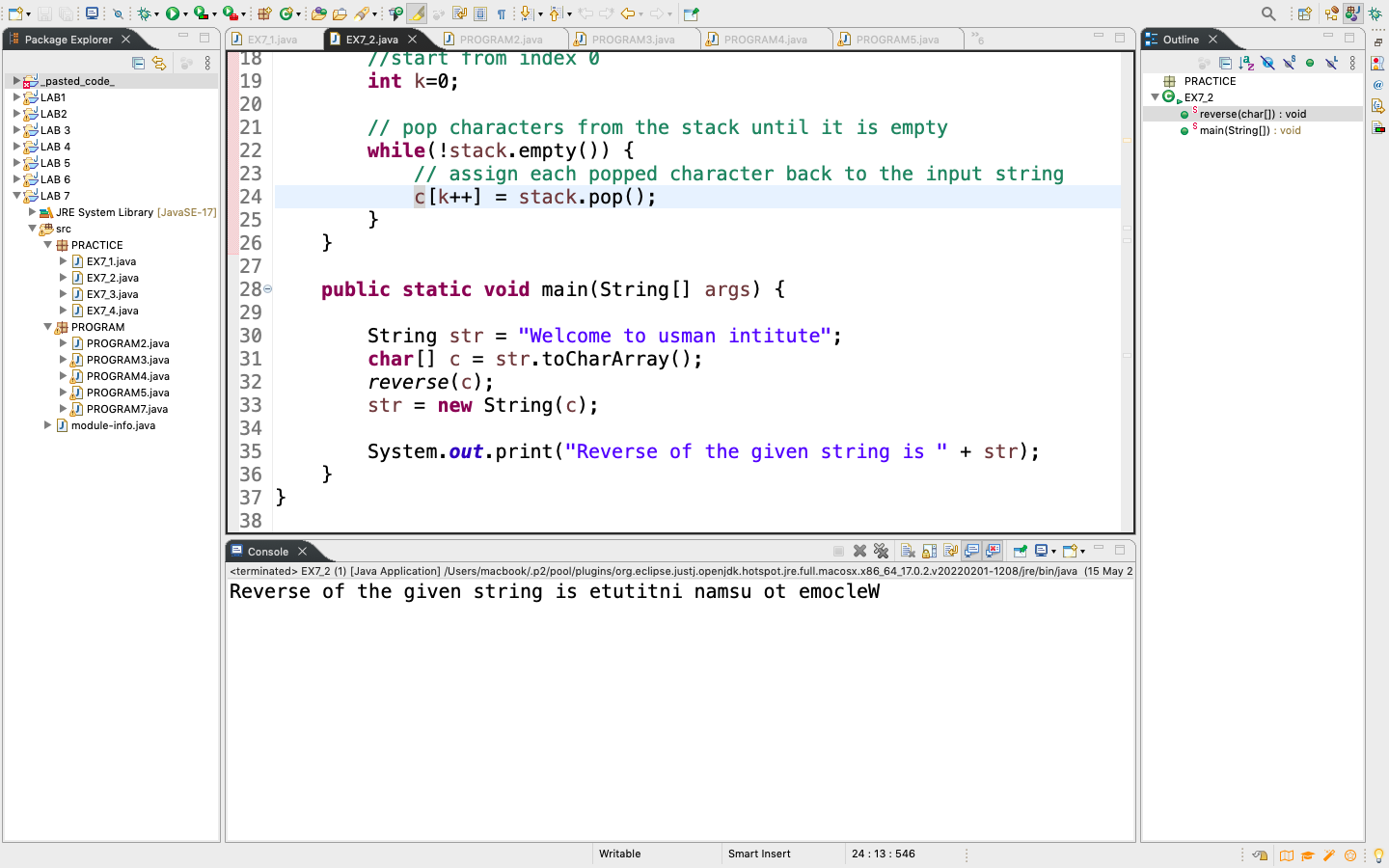
STEP6[initialize] initialize k and Set k:=0

STEP7[loop] repat step 8 while stack.empty()

STEP8 c[k=k+1] = stack.pop()

Exit.

OUTPUT:



PRACTICE 7.3:

Check if an expression is balanced or not

CODE:

**package** PRACTICE;

**import** java.util.Stack;

**public** **class** EX7\_3 {

// Function to check if the given expression is balanced or not

**public** **static** **boolean** isBalanced(String exp) {

// base case: length of the expression must be even

**if**(exp==**null** || exp.length() % 2 ==1) {

**return** **false**;

}

// take an empty stack of characters

Stack<Character> stack = **new** Stack<>();

// traverse the input expression

**for**(**char** c: exp.toCharArray()) {

// if the current character in the expression is an opening brace,

// push it into the stack

**if**(c == '(' || c == '{' || c == '[') {

stack.push(c);

}

// if the current character in the expression is a closing brace

**if**(c == ')' || c == '}' || c == ']') {

// return false if a mismatch is found (i.e., if the stack is empty,

// the expression cannot be balanced since the total number of opening

// braces is less than the total number of closing braces)

**if**(stack.empty()) {

**return** **false**;

}

// pop character from the stack

Character top = stack.pop();

// if the popped character is not an opening brace or does not pair

// with the current character of the expression

**if**((top == '(' && c!= ')') || (top == '{' && c!= '}')

|| (top == '[' && c!= ']')) {

**return** **false**;

}

}

}

// the expression is balanced only when the stack is empty at this point

**return** stack.empty();

}

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

String exp = "{()}[{}]";

**if** (*isBalanced*(exp)) {

System.***out***.println("The expression is balanced");

}

**else** {

System.***out***.println("The expression is not balanced");

}

}

}

ALGORITHM

The algorithm is to find the given expression is balanced or not and exp used as argument of type String

isBalanced( exp )

base case

STEP1[condition] IF exp=null or exp.length % 2 = 1

Return false

STEP2[initialize] create stack of type Character name stack

STEP3[loop] repeat step 4,5,6,7,8 and 9 for char c: exp.toCharArray()

STEP4[condition] IF c = ‘(’ or c = ‘{‘ or c = ‘[‘

STEP5 push(c)

STEP6[condition] IF c = ‘)’ or c = ‘}‘ or c = ‘]‘

STEP7[condition] IF stack = empty

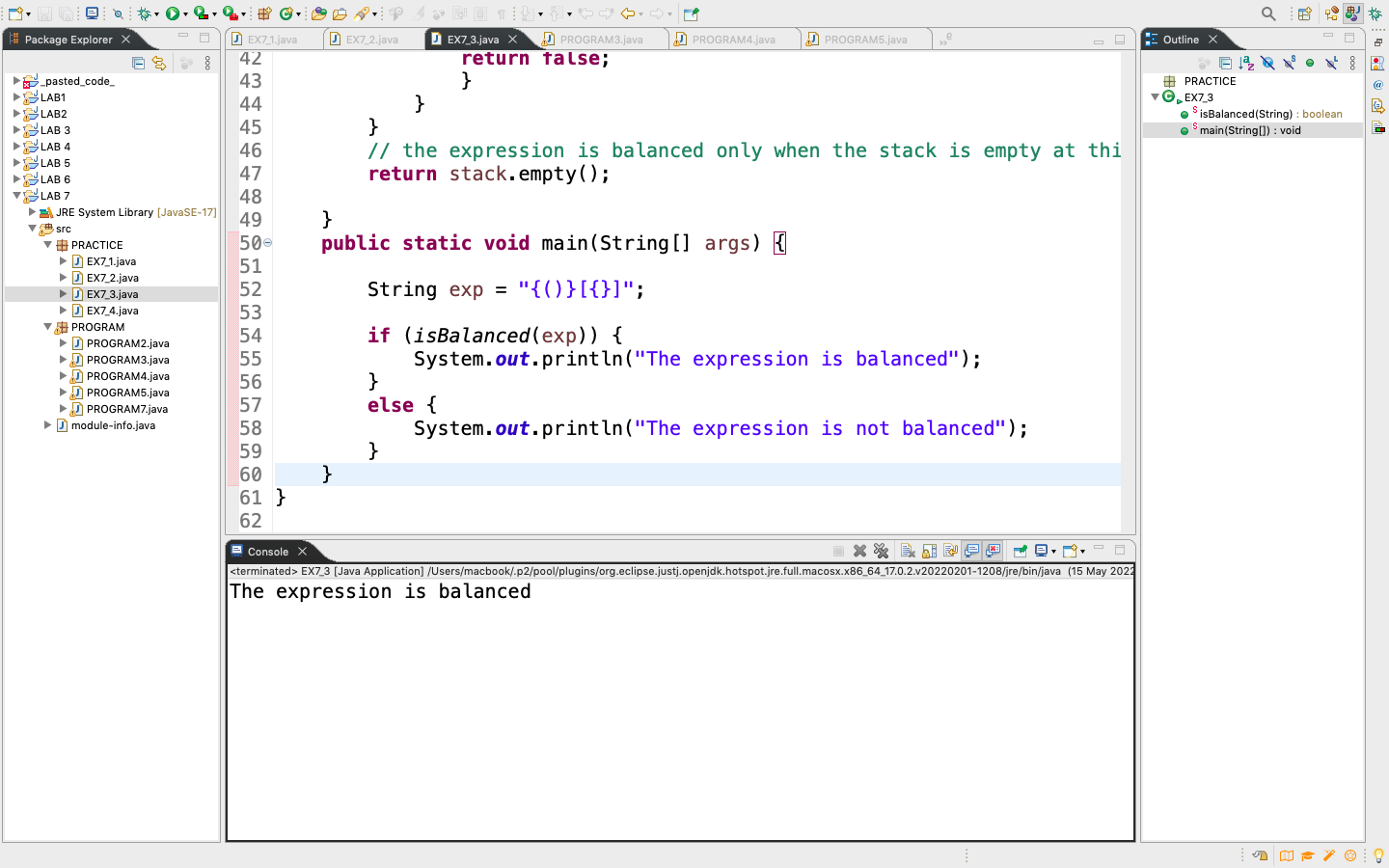
Return false

STEP8[initialize] Character top and set top:=stack.pop()

STEP9[condition] IF top=’(‘ and c!=’)’ or top=’{‘ and c!=’}’ top=’[‘ and c!=’]’

Return false

OUTPUT:



PRACTICE 7.4:

Convert an infix expression into a postfix expression

CODE:

**package** PRACTICE;

**import** java.util.Stack;

**public** **class** EX7\_4 {

// Utility function to return precedence of the given operator.

// Note that higher is the precedence, lower is its value

**public** **static** **int** perc(**char** c) {

// Multiplication and division

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

**return** 3;

}

**if**(c == '+' || c == '-') {

**return** 4;

}

//Bitwise AND

**if**(c == '&') {

**return** 8;

}

//Bitwise XOR operator

**if**(c == '^') {

**return** 9;

}

//Bitwise OR

**if**(c == '|') {

**return** 10;

}

// add more operators if needed

**return** Integer.***MAX\_VALUE***; // for opening bracket '('

}

// Utility function to check if a given token is an operand

**public** **static** **boolean** isoperand(**char** c) {

**return** (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||

(c >= '0' && c <= '9');

}

// Function to convert an infix expression to a postfix expression.

// This function expects a valid infix expression

**public** **static** String infixtoPostfix(String infix) {

//base case

**if**(infix == **null** || infix.length() == 0) {

**return** infix;

}

// create an empty stack for storing operators

Stack<Character> s = **new** Stack<>();

// create a string to store the postfix expression

String postFix = "";

// process the infix expression from left to right

**for**(**char** c: infix.toCharArray()) {

// Case 1. If the current token is an opening bracket '(',

// push it into the stack

**if**(c == '(') {

s.add(c);

}

// Case 2. If the current token is a closing bracket ')'

**else** **if**(c == ')') {

// pop tokens from the stack until the corresponding opening

// bracket '(' is removed. Append each operator at the end

// of the postfix expression

**while**(s.peek()!='(') {

postFix += s.pop();

}

s.pop();

}

// Case 3. If the current token is an operand, append it at the end

// of the postfix expression

**else** **if**(*isoperand*(c)) {

postFix +=c;

}

// Case 4. If the current token is an operator

**else** {

// remove operators from the stack with higher or equal precedence

// and append them at the end of the postfix expression

**while**(!s.isEmpty() && *perc*(c) >= *perc*(s.peek())) {

postFix += s.pop();

}

// finally, push the current operator on top of the stack

s.add(c);

}

}

// append any remaining operators in the stack at the end

// of the postfix expression

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

postFix += s.pop();

}

//return the postfix expression

**return** postFix;

}

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

String infix = "A\*(B\*C+D\*E)+F";

String postfix = *infixtoPostfix*(infix);

System.***out***.println(postfix);

}

}

ALGORITHM

infixtoPostfix(Q, P)  
Suppose Q is an arithmetic expression written in infix expression. This algorithm finds he equivalent post fix expression P

1. Push “(“ onto STACK and add “)” into the end of Q
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the stack is empty
3. If an operand is encounter, add it to P
4. If a left parenthesis is encountered , push it onto STACK
5. If an operand (x) is encountered, then
6. Repeatedly pop from the STACK and add to P each operator ( on the top of the STACK) which has the same precedence as of higher precedence than (x)
7. Add (x) to STACK

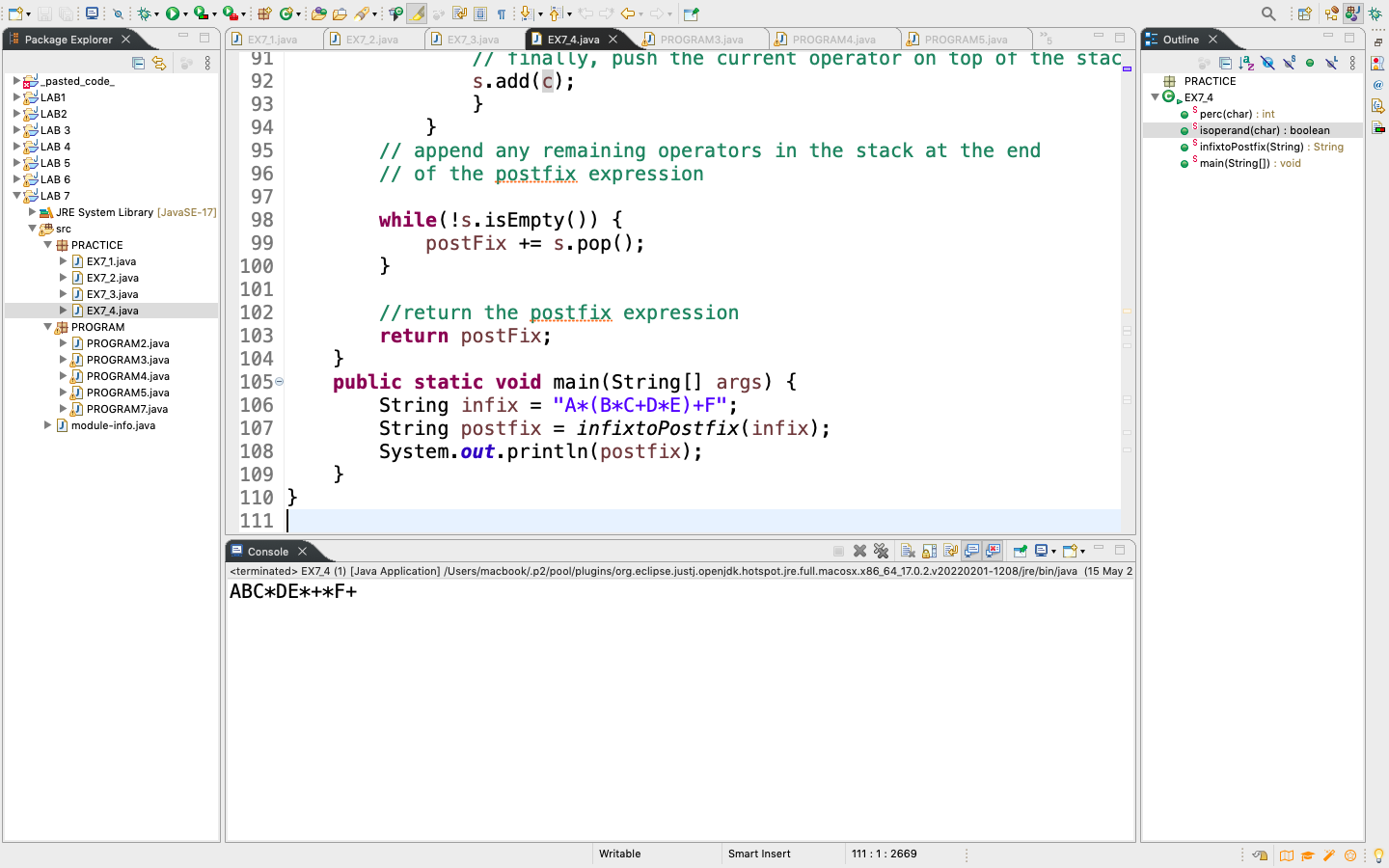
[End of IF structure]

1. If a right parenthesis is encountered, then:
2. Repeatedly pop from the STACK and add to P each operator ( on the top of the STACK) until a left parenthesis is encountered
3. Remove the left parenthesis [Do not add the left parenthesis to P]

[End of IF structure]

Exit

OUTPUT:



PROGRAMS

PROGRAM2:

Write an algorithm and thencode for solving the Depth First Search Algorithm.

CODE:

**package** PROGRAM;

**import** java.util.Scanner;

**public** **class** PROGRAM2 {

**private** **int** arr[];

**private** **int** top;

**private** **int** capacity;

// Constructor to initialize the stack

PROGRAM2(**int** size){

arr = **new** **int**[size];

capacity = size;

top = -1;

}

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

// Utility function to add an element `x` to the stack

**public** **void** push() {

**if**(isFull()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

**while**(capacity != size()) {

**int** x;

System.***out***.println("Enter number");

x = input.nextInt();

arr[++top] = x;

}

}

// Utility function to pop a top element from the stack

**public** **int** pop() {

// check for stack underflow

**if**(isEmpty()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

**while**(!isEmpty()) {

// decrease stack size by 1 and (optionally) return the popped element

System.***out***.println("Depth searching" + peek());

arr[top--] = peek();

}

**return** arr[top--];

}

// Utility function to return the top element of the stack

**public** **int** peek() {

**if**(!isEmpty()) {

**return** arr[top];

}

**else** {

System.*exit*(-1);

}

**return** -1;

}

**public** **int** size() {

**return** top + 1;

}

// Utility function to check if the stack is empty or not

**public** **boolean** isEmpty() {

**return** top == -1; // or return size() == 0;

}

// Utility function to check if the stack is full or not

**public** **boolean** isFull() {

**return** top == capacity -1; // or return size() == capacity;

}

}

**class** Mains{

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

PROGRAM2 stack = **new** PROGRAM2(3);

stack.push();

stack.pop();

}

}

ALGORITHM

The algorithm is to implement Depth first search function of stack

STEP1[initialize] arr[] as int, top as int and capacity as int

STEP2[constructor] set arr[] := int [size] , capacity as size and top:=-1

Push(x)

1. [Stack already filled?]
2. IF isFull(), then print OVERFLOW and return
3. While capacity != size
4. Set Top:= Top + 1 [increase TOP by 1]
5. Set arr[Top]:=x [insert x in new Top position]
6. Return

Pop()

1. [Stack has an empty to be removed]
2. IF isEmpty() , then print UNDERFLOW and return -1
3. While !isEmpty()
4. Print REMOVING peek()
5. Set arr[Top] := Top-1 [Decrease Top by -1
6. Return

Peek()

1. IF isEmpty(), then return Top
2. ELSE return -1

Size()

1. Return Top:=Top+1

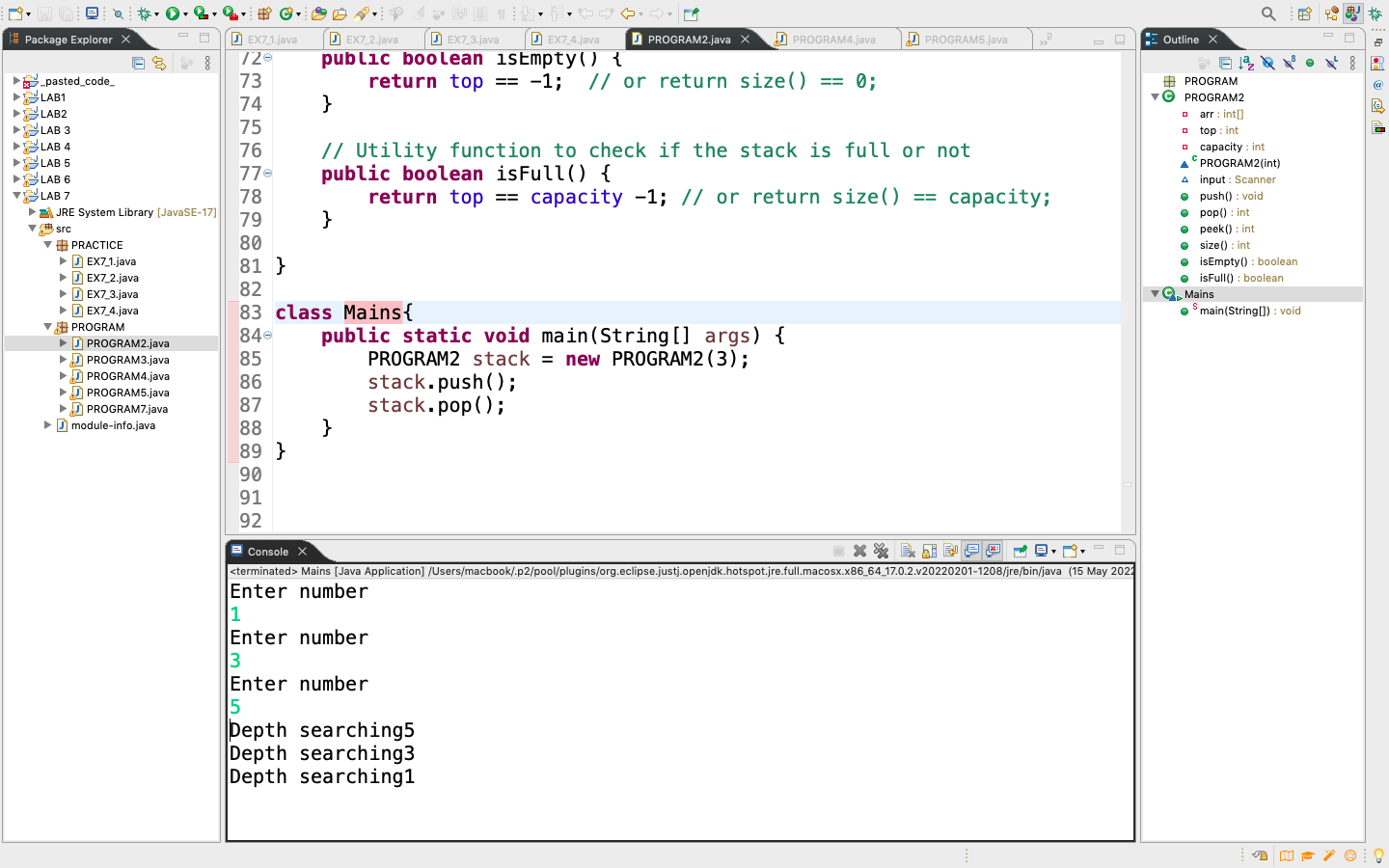
isEmpty()

1. Return Top=-1

isFull()

1. Retrun Top:=capacity -1

OUTPUT:



PROGRAM3:

Write an algorithm and then code to solve computations in postfix expression.

CODE:

**package** PROGRAM;

**import** java.util.Stack;

**class** StackNode

{

**public** String data;

**public** StackNode next;

**public** StackNode(String data, StackNode top)

{

**this**.data = data;

**this**.next = top;

}

}

**class** MyStack

{

**public** StackNode top;

**public** **int** count;

**public** MyStack()

{

**this**.top = **null**;

**this**.count = 0;

}

// Returns the number of element in stack

**public** **int** size()

{

**return** count;

}

**public** **boolean** isEmpty()

{

**if** (**this**.size() > 0)

{

**return** **false**;

}

**else**

{

**return** **true**;

}

}

// Add a new element in stack

**public** **void** push(String data)

{

// Make a new stack node

// And set as top

**this**.top = **new** StackNode(data, **this**.top);

// Increase node value

**this**.count++;

}

// Add a top element in stack

**public** String pop()

{

String temp = "";

**if** (**this**.isEmpty() == **false**)

{

// Get remove top value

temp = **this**.top.data;

**this**.top = **this**.top.next;

// Reduce size

**this**.count--;

}

**return** temp;

}

// Used to get top element of stack

**public** String peek()

{

**if** (!**this**.isEmpty())

{

**return** top.data;

}

**else**

{

**return** "";

}

}

}

**public** **class** PROGRAM3

{

// Check operator

**public** **boolean** isOperator(**char** text)

{

**if** (text == '+' ||

text == '-' ||

text == '\*' ||

text == '/' ||

text == '^' ||

text == '%')

{

**return** **true**;

}

**return** **false**;

}

// Check operands

**public** **boolean** isOperands(**char** text)

{

**if** ((text >= '0' && text <= '9') ||

(text >= 'a' && text <= 'z') ||

(text >= 'A' && text <= 'Z'))

{

**return** **true**;

}

**return** **false**;

}

// Converting the given postfix expression to

// infix expression

**public** **void** postfixToInfix(String postfix)

{

// Get the size

**int** size = postfix.length();

// Create stack object

MyStack s = **new** MyStack();

// Some useful variables which is using

// of to storing current result

String auxiliary = "";

String op1 = "";

String op2 = "";

**boolean** isValid = **true**;

**for** (**int** i = 0; i < size && isValid; i++)

{

// Check whether given postfix location

// at [i] is an operator or not

**if** (isOperator(postfix.charAt(i)))

{

// When operator exist

// Check that two operands exist or not

**if** (s.size() > 1)

{

op1 = s.pop();

op2 = s.pop();

auxiliary = "(" + op2 + postfix.charAt(i) + op1 + ")";

s.push(auxiliary);

}

**else**

{

isValid = **false**;

}

}

**else** **if** (isOperands(postfix.charAt(i)))

{

// When get valid operands

auxiliary = ""+postfix.charAt(i);

s.push(auxiliary);

}

**else**

{

// Invalid operands or operator

isValid = **false**;

}

}

**if** (isValid == **false**)

{

// When have something wrong

System.***out***.println("Invalid postfix : " + postfix);

}

**else**

{

// Display calculated result

System.***out***.println(" Postfix : " + postfix);

System.***out***.println(" Infix : " + s.pop());

}

}

**public** **static** **void** main(String[] args)

{

PROGRAM3 task = **new** PROGRAM3();

String postfix = "ABC\*DE\*+\*F+";

task.postfixToInfix(postfix);

}

}

ALGORITHM

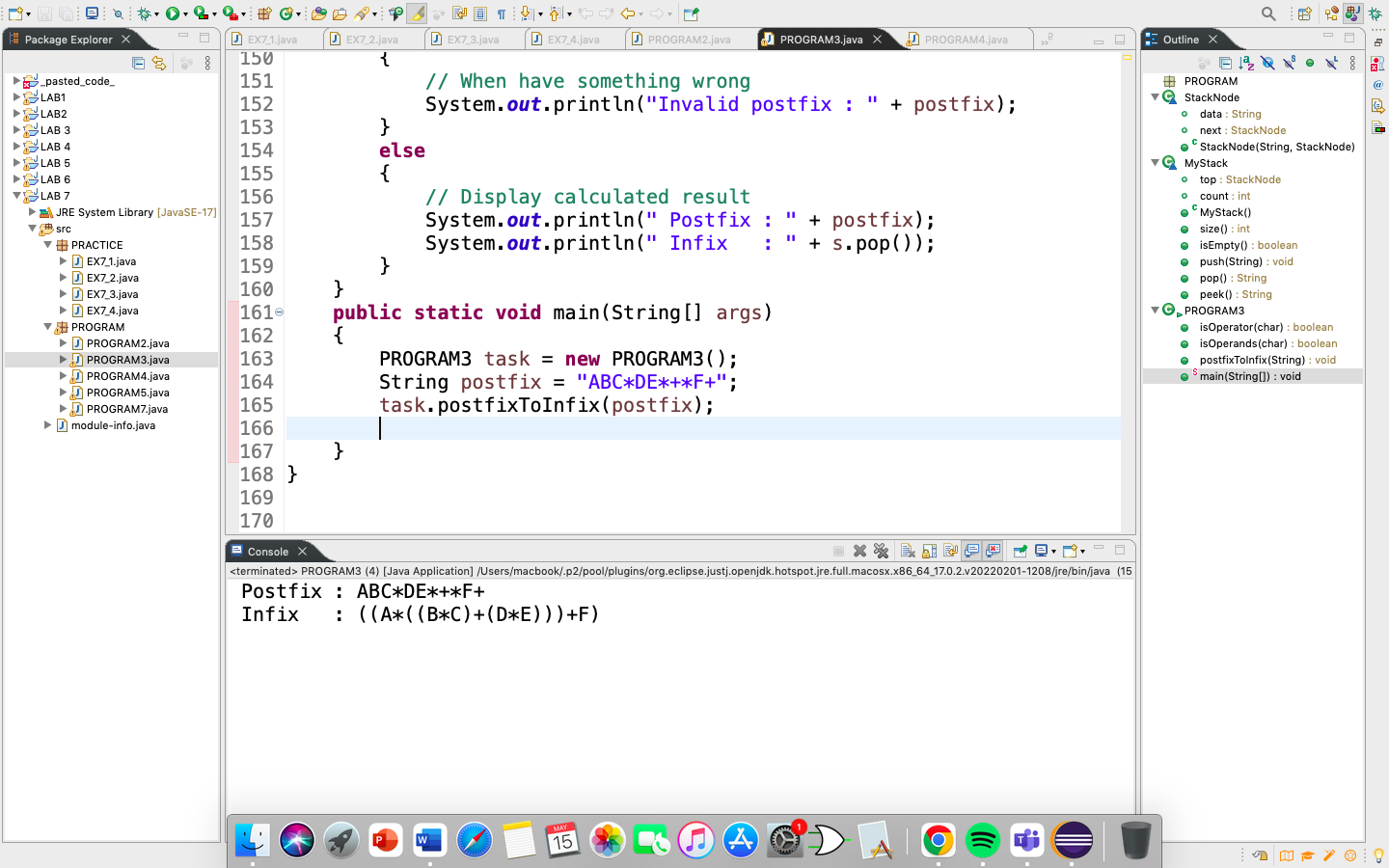
This Algorithm finds the VALUE of an arithmetic expression P written in postfix expression notation

1. Add a right parenthesis “)” at the end of P
2. Scan P from left to right and repeat step 3 and 4 for each element of P until the sentinel “)” is encountered
3. If an operand is encountered, put it on STACK
4. If an operator (x) is encountered, then:
5. Remove the two top elements of STACK, where A is the top element and B is the next-to-top element
6. Evaluate B (x) A
7. Place the result of (b) back on STACK

[End of if structure]

1. [End of step 2 loop]
2. Set VALUE to the top element of STACK
3. Exit

OUTPUT:



PROGRAM4:

Write an algorithm and code to make mobile phone application such that the caller history will come in such a way that last person calls show at the top and then previous and etc.

CODE:

**package** PROGRAM;

**import** java.util.Scanner;

**public** **class** PROGRAM4 {

**private** String arr1[];

**private** **int** top;

**private** **int** capacity;

// Constructor to initialize the stack

PROGRAM4(**int** size){

arr1 = **new** String[size];

capacity = size;

top = -1;

}

**public** **void** push1(String g) {

**if**(isFull()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

arr1[++top] = g;

}

**public** String pop1() {

**if**(isEmpty()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

**while**(!isEmpty()) {

System.***out***.println("Call history: " + peek1());

arr1[top--] = peek1();

}

**return** arr1[top--];

}

**public** String peek1() {

**if**(!isEmpty()) {

**return** arr1[top];

}

**else** {

System.*exit*(-1);

}

**return** "";

}

**public** **int** size() {

**return** top + 1;

}

// Utility function to check if the stack is empty or not

**public** **boolean** isEmpty() {

**return** top == -1; // or return size() == 0;

}

// Utility function to check if the stack is full or not

**public** **boolean** isFull() {

**return** top == capacity -1; // or return size() == capacity;

}

}

**class** Main{

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

PROGRAM4 stack = **new** PROGRAM4(3);

stack.push1("shaheer '123'");

stack.push1("huzaifa '3472'");

stack.push1("zahir '7462'");

stack.pop1();

}

}

ALGORITHM

The algorithm is to implement Call history function of stack

STEP1[initialize] arr[] as int, top as int and capacity as int

STEP2[constructor] set arr[] := int [size] , capacity as size and top:=-1

Push(x)

1. [Stack already filled?]
2. IF isFull(), then print OVERFLOW and return
3. Set Top:= Top + 1 [increase TOP by 1]
4. Set arr[Top]:=x [insert x in new Top position]
5. Return

Pop()

1. [Stack has an empty to be removed]
2. IF isEmpty() , then print UNDERFLOW and return -1
3. While !isEmpty()
4. Print Call history peek()
5. Set arr[Top] := Top-1 [Decrease Top by -1
6. Return

Peek()

1. IF isEmpty(), then return Top
2. ELSE return -1

Size()

1. Return Top:=Top+1

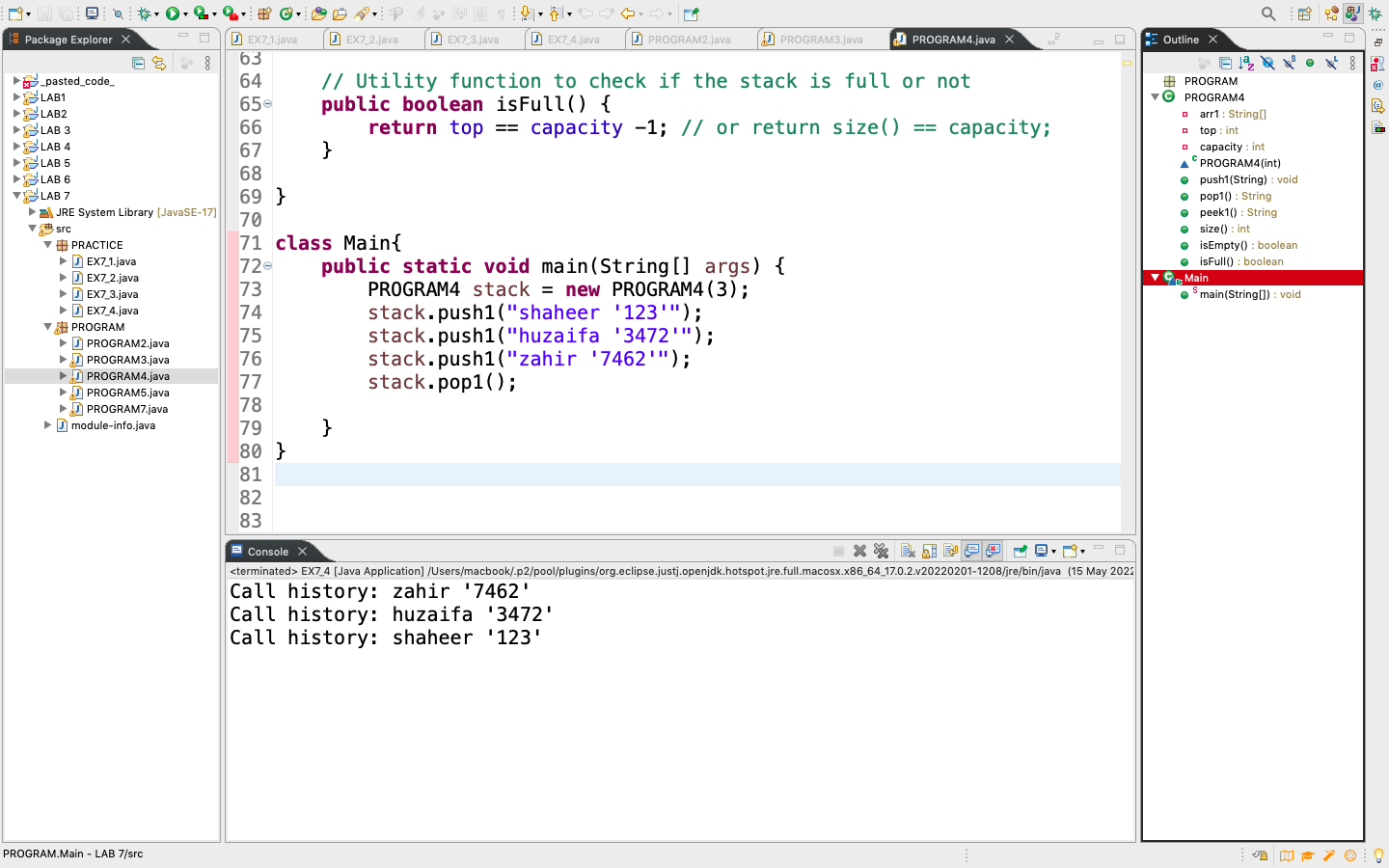
isEmpty()

1. Return Top=-1

isFull()

1. Retrun Top:=capacity -1

OUTPUT:



PROGRAM6:

Write an application in which you have placed 10 books in a stack then pop these book and search if you find the desired book or not using stack.

CODE:

**package** PROGRAM;

**import** java.util.Scanner;

**public** **class** PROGRAM5 {

**private** String arr1[];

**private** **int** top;

**private** **int** capacity;

// Constructor to initialize the stack

PROGRAM5(**int** size){

arr1 = **new** String[size];

capacity = size;

top = -1;

}

**public** **void** push1(String g) {

**if**(isFull()) {

System.***out***.println("Overflow\nProgram Terminated\n");

System.*exit*(-1);

}

arr1[++top] = g;

}

**public** String pop1(String b) {

**while**(!isEmpty()) {

String a = peek1();

**if**(a == "The Stranger") {

System.***out***.println("yes the book is present " + b);

}

arr1[top--] = peek1();

}

**return** arr1[top--];

}

**public** String peek1() {

**if**(!isEmpty()) {

**return** arr1[top];

}

**else** {

System.*exit*(-1);

}

**return** "";

}

**public** **int** size() {

**return** top + 1;

}

// Utility function to check if the stack is empty or not

**public** **boolean** isEmpty() {

**return** top == -1; // or return size() == 0;

}

// Utility function to check if the stack is full or not

**public** **boolean** isFull() {

**return** top == capacity -1; // or return size() == capacity;

}

}

**class** g{

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

PROGRAM5 stack = **new** PROGRAM5(10);

stack.push1("One Hundred Years of Solitude");

stack.push1("War and Peace");

stack.push1("The Odyssey");

stack.push1("Pride and Prejudice");

stack.push1("Middlemarch");

stack.push1("Gulliver's Travels");

stack.push1("The Stranger");

stack.push1("Jane Eyre");

stack.push1("Midnight's Children");

stack.push1("The Lord of the Rings");

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

String b;

System.***out***.println("Enter book name");

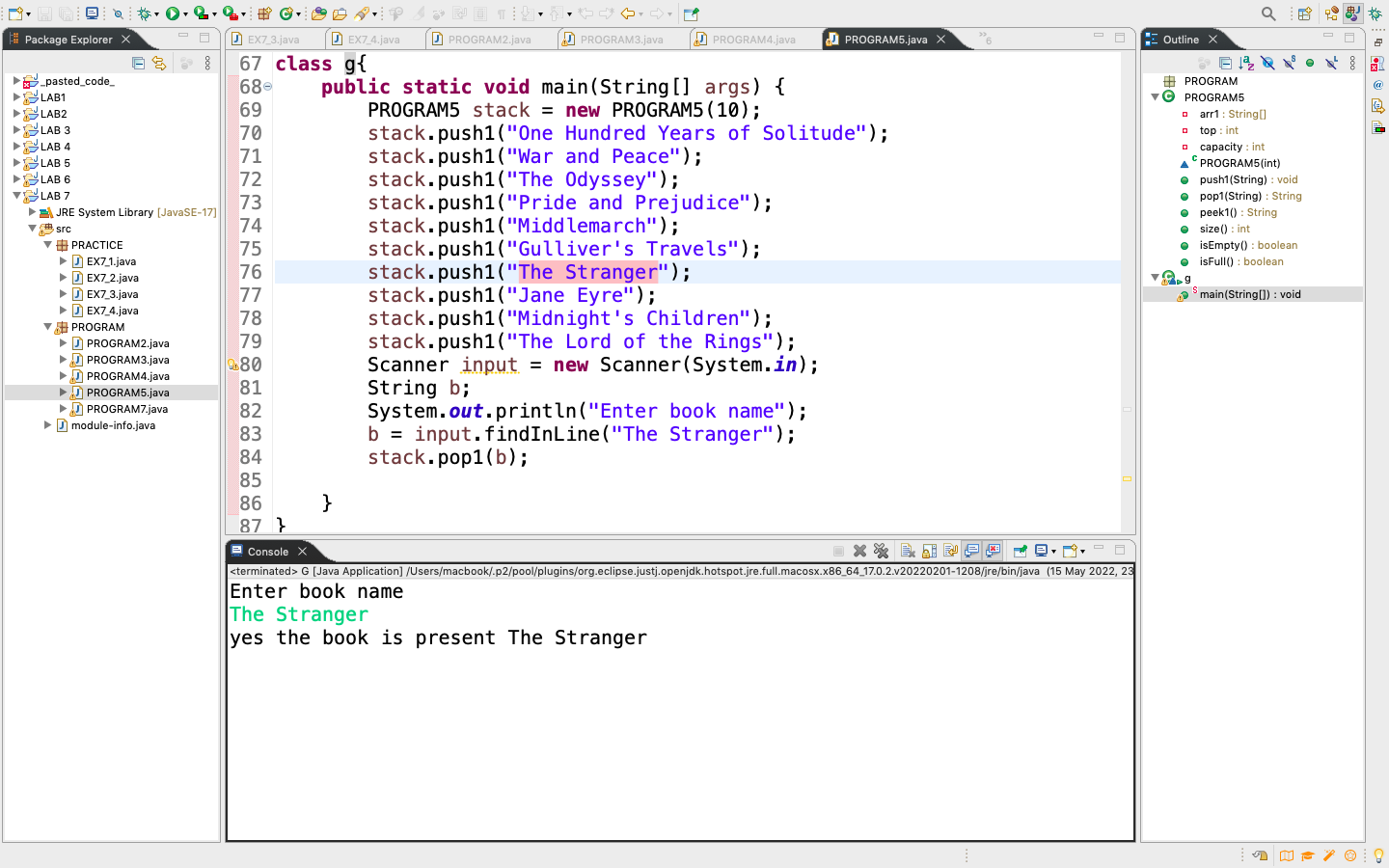
b = input.findInLine("The Stranger");

stack.pop1(b);

}

}

OUTPUT:



PROGRAM7:

You need to develop an application of Tower of Hanoi for this use Stack to move you disks.

CODE:

package PROGRAM;

import java.util.Stack;

import java.util.Scanner;

public class PROGRAM7 {

public static int N;

/\* Creating Stack array \*/

public static Stack<Integer>[] tower = new Stack[4];

public static void main(String[] args)

{

Scanner input = new Scanner(System.in);

tower[1] = new Stack<Integer>();

tower[2] = new Stack<Integer>();

tower[3] = new Stack<Integer>();

/\* Accepting number of disks \*/

System.out.println("Enter number of disks");

int num = input.nextInt();

N = num;

tower(num);

}

/\* Function to push disks into stack \*/

public static void tower(int n)

{

for (int i = n; i > 0; i--)

tower[1].push(i);

display();

move(n, 1, 2, 3);

}

/\* Recursive Function to move disks \*/

public static void move(int n, int a, int b, int c)

{

if (n > 0)

{

move(n-1, a, c, b);

int d = tower[a].pop();

tower[c].push(d);

display();

move(n-1, b, a, c);

}

}

/\* Function to display \*/

public static void display()

{

System.out.println(" A | B | C");

System.out.println("---------------");

for(int i = N - 1; i >= 0; i--)

{

String d1 = " ", d2 = " ", d3 = " ";

try

{

d1 = String.valueOf(tower[1].get(i));

}

catch (Exception e){

}

try

{

d2 = String.valueOf(tower[2].get(i));

}

catch(Exception e){

}

try

{

d3 = String.valueOf(tower[3].get(i));

}

catch (Exception e){

}

System.out.println(" "+d1+" | "+d2+" | "+d3);

}

System.out.println("\n");

}

}

OUTPUT:

