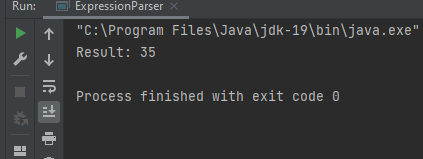
Muzamil Sikander cs221226

Dsa Lab Final

**Question 2** Design a stack-based expression parser that can evaluate complex mathematical expressions, including arithmetic operations and nested parentheses.

Code:

public class ExpressionParser {  
  
 private static class CustomStack<T> {  
 private Object[] array;  
 private int size;  
 private static final int *DEFAULT\_CAPACITY* = 10;  
  
 public CustomStack() {  
 this.array = new Object[*DEFAULT\_CAPACITY*];  
 this.size = 0;  
 }  
  
 public void push(T element) {  
 ensureCapacity();  
 array[size++] = element;  
 }  
  
 public T pop() {  
 if (isEmpty()) {  
 throw new IllegalStateException("Stack is empty");  
 }  
 T element = (T) array[--size];  
 array[size] = null; // Avoid memory leak  
 return element;  
 }  
  
 public T peek() {  
 if (isEmpty()) {  
 throw new IllegalStateException("Stack is empty");  
 }  
 return (T) array[size - 1];  
 }  
  
 public boolean isEmpty() {  
 return size == 0;  
 }  
  
 private void ensureCapacity() {  
 if (size == array.length) {  
 int newCapacity = array.length \* 2;  
 Object[] newArray = new Object[newCapacity];  
 System.*arraycopy*(array, 0, newArray, 0, size);  
 array = newArray;  
 }  
 }  
 }  
  
 public static int evaluateExpression(String expression) {  
 CustomStack<Integer> values = new CustomStack<>();  
 CustomStack<Character> operators = new CustomStack<>();  
  
 for (int i = 0; i < expression.length(); i++) {  
 char ch = expression.charAt(i);  
  
 if (ch == ' ') {  
 continue; // Skip spaces  
 }  
  
 if (Character.*isDigit*(ch)) {  
 int num = 0;  
 while (i < expression.length() && Character.*isDigit*(expression.charAt(i))) {  
 num = num \* 10 + (expression.charAt(i) - '0');  
 i++;  
 }  
 values.push(num);  
 i--; // Move back one position to correctly handle the next character in the loop  
 } else if (ch == '(') {  
 operators.push(ch);  
 } else if (ch == ')') {  
 while (operators.peek() != '(') {  
 values.push(*applyOperator*(operators.pop(), values.pop(), values.pop()));  
 }  
 operators.pop(); // Pop '('  
 } else if (*isOperator*(ch)) {  
 while (!operators.isEmpty() && *hasPrecedence*(ch, operators.peek())) {  
 values.push(*applyOperator*(operators.pop(), values.pop(), values.pop()));  
 }  
 operators.push(ch);  
 }  
 }  
  
 while (!operators.isEmpty()) {  
 values.push(*applyOperator*(operators.pop(), values.pop(), values.pop()));  
 }  
  
 return values.pop();  
 }  
  
 private static boolean isOperator(char ch) {  
 return ch == '+' || ch == '-' || ch == '\*' || ch == '/';  
 }  
  
 private static boolean hasPrecedence(char op1, char op2) {  
 if (op2 == '(' || op2 == ')') {  
 return false;  
 }  
 return (op1 == '\*' || op1 == '/') && (op2 == '+' || op2 == '-');  
 }  
  
 private static int applyOperator(char operator, int b, int a) {  
 switch (operator) {  
 case '+':  
 return a + b;  
 case '-':  
 return a - b;  
 case '\*':  
 return a \* b;  
 case '/':  
 if (b == 0) {  
 throw new ArithmeticException("Cannot divide by zero");  
 }  
 return a / b;  
 default:  
 throw new IllegalArgumentException("Invalid operator: " + operator);  
 }  
 }  
  
 public static void main(String[] args) {  
 String expression = "((2 + 6) \* 5) - 5";  
 int result = *evaluateExpression*(expression);  
 System.*out*.println("Result: " + result);  
 }  
}

Output :

Question 1:

Code:

import java.io.File;  
import java.io.FileNotFoundException;  
import java.util.Scanner;  
  
class Queue {  
 private int maxSize;  
 private int[] queueArray;  
 private int front;  
 private int rear;  
 private int nItems;  
  
 public Queue(int size) {  
 this.maxSize = size;  
 this.queueArray = new int[maxSize];  
 this.front = 0;  
 this.rear = -1;  
 this.nItems = 0;  
 }  
  
 public void insert(int value) {  
 if (rear == maxSize - 1) {  
 rear = -1;  
 }  
 queueArray[++rear] = value;  
 nItems++;  
 }  
  
 public int remove() {  
 int temp = queueArray[front++];  
 if (front == maxSize) {  
 front = 0;  
 }  
 nItems--;  
 return temp;  
 }  
  
 public int peekFront() {  
 return queueArray[front];  
 }  
  
 public boolean isEmpty() {  
 return (nItems == 0);  
 }  
  
 public boolean isFull() {  
 return (nItems == maxSize);  
 }  
}  
public class Scheduler {  
 private Queue queue;  
  
 public Scheduler(int size) {  
 this.queue = new Queue(size);  
 }  
  
 public void executeProcessesFromFile(String fileName) {  
 try {  
 File file = new File(fileName);  
 Scanner scanner = new Scanner(file);  
  
 int quantum = scanner.nextInt(); // Number of instructions to execute at a time  
 int numProcesses = scanner.nextInt(); // Total number of processes  
  
 for (int i = 1; i <= numProcesses; i++) {  
 queue.insert(i);  
 }  
  
 while (!queue.isEmpty()) {  
 int currentProcess = queue.remove();  
 System.*out*.println("Executing " + quantum + " instructions of process p" + currentProcess);  
  
 // Decrease the total instructions of the current process  
 quantum--;  
 if (quantum == 0) {  
 System.*out*.println("Process p" + currentProcess + " has finished execution");  
 if (!queue.isEmpty()) {  
 quantum = scanner.nextInt(); // Get the next quantum from the file  
 }  
 } else {  
 queue.insert(currentProcess); // Put the process back into the queue  
 }  
 }  
  
 scanner.close();  
 } catch (FileNotFoundException e) {  
 System.*out*.println("File not found: " + e.getMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 Scheduler scheduler = new Scheduler(10); // Initialize the scheduler with a maximum size  
  
 if (args.length > 0) {  
 scheduler.executeProcessesFromFile(args[0]); // Use the provided file name from command-line argument  
 } else {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.print("Enter the file name: ");  
 String fileName = scanner.nextLine();  
 scheduler.executeProcessesFromFile(fileName);  
 scanner.close();  
 }  
 }  
}

Output:

