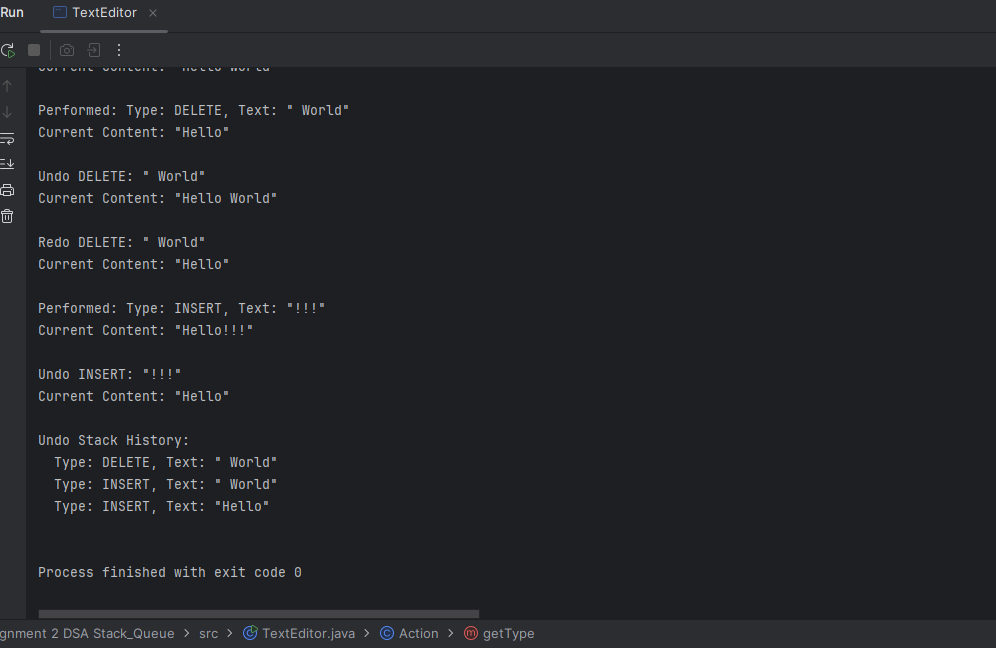
Scenario 01

**Simulate the Undo and Redo operations of a text editor using two stacks – one for undo and one for redo operations.**

import java.util.Stack;  
enum ActionType {  
 *INSERT*,*DELETE*}  
class Action{  
 private ActionType type;  
 private String text;  
  
 public Action(ActionType type, String text) {  
 this.type = type;  
 this.text = text;  
 }  
  
 public ActionType getType() {  
 return type;  
 }  
  
 public String getText() {  
 return text;  
 }  
 public String toString() {  
 return "Type: " + type + ", Text: \"" + text + "\"";  
 }  
}  
  
  
class EditorControl {  
 private final Stack<Action> undoStack = new Stack<>();  
 private final Stack<Action> redoStack = new Stack<>();  
 private String content = "";  
  
  
 public void performAction(ActionType type, String text) {  
 if (type == null || text == null || text.isEmpty()) {  
 System.*out*.println("Invalid action or text.");  
 return;  
 }  
  
 Action action = new Action(type, text);  
  
 switch (type) {  
 case *INSERT*:  
 content += text;  
 break;  
 case *DELETE*:  
 int len = text.length();  
 if (content.length() >= len) {  
 content = content.substring(0, content.length() - len);  
 } else {  
 System.*out*.println("Delete length exceeds current content. No action performed.");  
 return;  
 }  
 break;  
 default:  
 System.*out*.println("Unknown action type.");  
 }  
 undoStack.push(action);  
 redoStack.clear();  
 System.*out*.println("Performed: " + action);  
 showContent();  
 }  
  
  
 public void undo() {  
 if (undoStack.isEmpty()) {  
 System.*out*.println("Nothing to undo.");  
 return;  
 }  
  
 Action lastAction = undoStack.pop();  
 switch (lastAction.getType()) {  
 case *INSERT*:  
 if (content.length() >= lastAction.getText().length()) {  
 content = content.substring(0, content.length() - lastAction.getText().length());  
 }  
 break;  
 case *DELETE*:  
 content += lastAction.getText();  
 break;  
 default:  
 System.*out*.println("Unknown action type in undo.");  
 }  
 redoStack.push(lastAction);  
 System.*out*.println("Undo " + lastAction.getType() + ": \"" + lastAction.getText() + "\"");  
 showContent();  
 }  
  
  
  
 public void redo() {  
 if (redoStack.isEmpty()) {  
 System.*out*.println("Nothing to redo.");  
 return;  
 }  
  
 Action action = redoStack.pop();  
 switch (action.getType()) {  
 case *INSERT*:  
 content += action.getText();  
 break;  
 case *DELETE*:  
 if (content.length() >= action.getText().length()) {  
 content = content.substring(0, content.length() - action.getText().length());  
 }  
 break;  
 default:  
 System.*out*.println("Unknown action type in redo.");  
 }  
  
  
 undoStack.push(action);  
 System.*out*.println("Redo " + action.getType() + ": \"" + action.getText() + "\"");  
 showContent();  
 }  
  
  
 public void showContent() {  
 System.*out*.println("Current Content: \"" + content + "\"\n");  
 }  
  
  
 public void showHistory() {  
 System.*out*.println("Undo Stack History:");  
 for (int i = undoStack.size() - 1; i >= 0; i--) {  
 System.*out*.println(" " + undoStack.get(i));  
 }  
 System.*out*.println();  
 }  
}  
  
public class TextEditor {  
 public static void main(String[] args) {  
 EditorControl editor = new EditorControl();  
  
 editor.performAction(ActionType.*INSERT*, "Hello");  
 editor.performAction(ActionType.*INSERT*, " World");  
 editor.performAction(ActionType.*DELETE*, " World");  
 editor.undo();  
 editor.redo();  
 editor.performAction(ActionType.*INSERT*, "!!!");  
 editor.undo();  
 editor.showHistory();  
 }  
}



Scenario 02

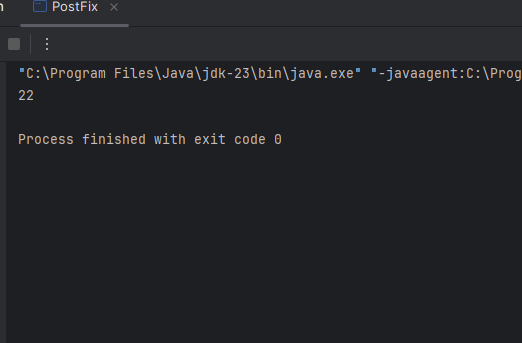
**Build a simple helpdesk queue system where customers are served in the order they arrive (FIFO).**

import java.util.LinkedList;  
import java.util.Queue;  
  
class Customer{  
  
 private final int ticketNumber;  
 private String name;  
 private String issueDescription;  
  
 public Customer(int ticketNumber, String name, String issue) {  
 this.ticketNumber = ticketNumber;  
 this.name = name;  
 this.issueDescription = issue;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getIssueDescription() {  
 return issueDescription;  
 }  
  
 public void setIssueDescription(String issueDescription) {  
 this.issueDescription = issueDescription;  
 }  
  
 @Override  
 public String toString() {  
 return "Ticket #" + ticketNumber + ": " + name + " - " + issueDescription;  
 }  
}  
class CustomerHelpDeskQueue{  
 Queue<Customer> urgentQueue=new LinkedList<>();  
 Queue<Customer> normalQueue=new LinkedList<>();  
 private int ticketNumber=100;  
 public void enqueue(String name,String issue){  
 enqueue(name,issue,"normal");  
 }  
  
 public void enqueue(String name,String issue,String priority){  
 if (isBlank(name) || isBlank(issue) || isBlank(priority)) {  
 System.*out*.println("Error: Name and issue description must not be blank.");  
 }  
  
 Customer newCustomer = new Customer(ticketNumber++, name.trim(), issue.trim());  
  
 if (priority.equalsIgnoreCase("urgent")) {  
 urgentQueue.add(newCustomer);  
 System.*out*.println("Urgent ticket created: " + newCustomer);  
 }  
 else {  
 normalQueue.add(newCustomer);  
 System.*out*.println("Normal ticket created: " + newCustomer);  
 }  
  
 }  
  
 public void dequeue() {  
 if (!urgentQueue.isEmpty()) {  
 Customer served = urgentQueue.poll();  
 System.*out*.println("Serving URGENT Customer:" + served);  
 }  
 else if (!normalQueue.isEmpty()) {  
 Customer served = normalQueue.poll();  
 System.*out*.println("Serving NORMAL Customer:" + served);  
 }  
 else {  
 System.*out*.println("No customer to serve in the queue.");  
 }  
 }  
 public void peek() {  
 if (!urgentQueue.isEmpty()) {  
 Customer next = urgentQueue.peek();  
 System.*out*.println("Next URGENT customer: " + next);  
 } else if (!normalQueue.isEmpty()) {  
 Customer next = normalQueue.peek();  
 System.*out*.println("Next NORMAL customer: " + next);  
 } else {  
 System.*out*.println("No customers in queue.");  
 }  
 }  
  
 public void displayQueue() {  
  
 if (urgentQueue.isEmpty() && normalQueue.isEmpty()) {  
 System.*out*.println("Queue is empty.");  
 return;  
 }  
 System.*out*.println("Current Queue:");  
 for (Customer customer : urgentQueue) {  
 System.*out*.println("[URGENT] ==>> " + customer);  
 }  
 for (Customer customer : normalQueue) {  
 System.*out*.println("[NORMAL] ==>> " + customer);  
 }  
 }  
  
  
 private boolean isBlank(String str) {  
 return str == null || str.trim().isEmpty();  
 }  
  
  
}  
  
public class CustomerHelpDesk {  
 public static void main(String[] args) {  
  
 CustomerHelpDeskQueue helpdesk = new CustomerHelpDeskQueue();  
  
 helpdesk.enqueue("Ali", "Cannot Access Email");  
  
 helpdesk.enqueue("Bilal", "Laptop not start");  
  
 helpdesk.enqueue("Amjad", "Forgot my password", "urgent");  
  
 helpdesk.enqueue("Hamza", "Printer is not working");  
  
 helpdesk.enqueue("Ahmad", "System is crash", "urgent");  
  
 helpdesk.displayQueue();  
  
 helpdesk.peek();  
 helpdesk.dequeue();  
 helpdesk.dequeue();  
 helpdesk.displayQueue();  
  
 }  
}

Scenario 03

**Problem Statement: Evaluate a postfix expression using a Stack.**

import java.util.Stack;  
  
public class PostFix {  
 public static int evaluate(String exp){  
 if (exp==null|| exp.trim().isEmpty()){  
 throw new IllegalArgumentException("Expresion must not be null or empty");  
 }  
 Stack<Integer> stack=new Stack<>();  
 String []tokens=exp.trim().split(" ");  
  
 for (String token :tokens){  
 if(*isNumber*(token)){  
 stack.push(Integer.*parseInt*(token));  
 }  
 else if (*isOperator*(token)) {  
 if(stack.size()<2){  
 throw new IllegalArgumentException("Error ==>> Insufficient Operands");  
 }  
 int b=stack.pop();  
 int a=stack.pop();  
  
 int result=*applyOperator*(a,b,token);  
 stack.push(result);  
  
 }  
 else {  
 throw new IllegalArgumentException("Error ==>> Invalid token '" + token + "'");  
  
 }  
  
 }  
  
 if(stack.size()!=1){  
 throw new IllegalArgumentException("Error ==>> Invalid Expression");  
 }  
 return stack.pop();  
 }  
  
  
 private static boolean isNumber(String num){  
 try{  
 Integer.*parseInt*(num);  
 return true;  
 }  
 catch (NumberFormatException e){  
 return false;  
 }  
 }  
  
 private static boolean isOperator(String op){  
 String myOp="+-\*/%^";  
 return myOp.contains(op)&&op.length()==1;  
 }  
  
 private static int applyOperator(int a, int b, String operator) {  
 switch (operator) {  
 case "+":  
 return a + b;  
 case "-":  
 return a - b;  
 case "\*":  
 return a \* b;  
 case "/":  
 if (b == 0) {  
 throw new ArithmeticException("Error ==>> Division by zero.");  
 }  
 return a / b;  
 case "%":  
 if (b == 0) {  
 throw new ArithmeticException("Error ==>> Modulus by zero.");  
 }  
 return a % b;  
 case "^":  
 return (int) Math.*pow*(a, b);  
 default:  
 throw new IllegalArgumentException("Unsupported operator: " + operator);  
 }  
 }  
  
  
  
  
  
 public static void main(String[] args) {  
 System.*out*.println(*evaluate*("5 6 + 2 \*"));  
  
  
  
 }  
}



Scenario 04

**Manage aircraft takeoff using a queue-based runway system where planes take off in order of arrival.**

import java.util.LinkedList;  
import java.util.Queue;  
  
class Aircraft {  
 String flightNumber;  
 String destination;  
  
 public Aircraft(String flightNumber, String destination) {  
 this.flightNumber = flightNumber;  
 this.destination = destination;  
 }  
  
  
 public String toString() {  
 return "Flight: #" + flightNumber + " --> Destination: " + destination;  
 }  
}  
  
class RunwaySystem {  
 Queue<Aircraft> queue = new LinkedList<>();  
  
 public boolean isFlightNumberUnique(String flightNumber) {  
 for (Aircraft flight : queue) {  
 if (flight.flightNumber.equalsIgnoreCase(flightNumber)) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 public boolean isBlank(String str){  
 return str==null||str.trim().isEmpty();  
 }  
  
 public void addFlight(String flightNumber, String destination) {  
 if (isBlank(flightNumber)||isBlank(destination)) {  
 System.*out*.println("FlightNumber and destination should not blank.");  
 return;  
 }  
 if (!isFlightNumberUnique(flightNumber)) {  
 System.*out*.println("Error --> Flight number '" + flightNumber + "' already exist.");  
 return;  
 }  
 Aircraft newFlight = new Aircraft(flightNumber.trim(), destination.trim());  
 queue.offer(newFlight);  
 System.*out*.println("Added --> " + newFlight);  
 }  
  
 public void authorizeTakeoff() {  
 if (queue.isEmpty()) {  
 System.*out*.println("No flight for takeoff.");  
 return;  
 }  
 Aircraft flight = queue.poll();  
 System.*out*.println("Authorized for Takeoff: " + flight);  
 }  
  
 public void peekNextFlight() {  
 if (queue.isEmpty()) {  
 System.*out*.println("No flight for takeoff.");  
 return;  
 }  
 System.*out*.println("Next Flight --> " + queue.peek());  
 }  
  
 public void displayQueue() {  
 if (queue.isEmpty()) {  
 System.*out*.println("No flight in queue.");  
 return;  
 }  
 System.*out*.println("\nFlights waiting for takeoff:");  
 for (Aircraft flight : queue) {  
 System.*out*.println(" ==>> " + flight);  
 }  
 }  
}  
  
public class AirportTakeOffManager {  
 public static void main(String[] args) {  
 RunwaySystem runway = new RunwaySystem();  
  
 runway.addFlight("PK301", "Lahore");  
 runway.addFlight("EK712", "Dubai");  
 runway.addFlight("BA215", "London");  
 runway.addFlight("PK301", "Karachi");  
  
 runway.displayQueue();  
  
 runway.peekNextFlight();  
 runway.authorizeTakeoff();  
 runway.authorizeTakeoff();  
  
 runway.displayQueue();  
 }  
}

