**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management & Engineering**

**Computer Engineering Department**

Program: MCA, Semester I

**Course: Java Programming**

**Experiment No.06**

**Part B**

**(Students must submit the soft copy as per the following segments. The soft copy must be uploaded on the Blackboard within two hours of your practical session.)**

|  |  |
| --- | --- |
| **Roll no.: A073** | **Name: Aryan Srivastava** |
| **Class: MCA SEM1** | **Batch: B3** |
| **Date of Experiment:** | **Date of Submission:** |
| **Grade:** |  |

1. **Program scenario and Program code:**

**Qn1.**

import java.util.InputMismatchException;

import java.util.Scanner;

class linear{

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int[] array = {10, 20, 30, 40, 50};

        try {

            System.out.println("Enter the element to search for:");

            int target = scanner.nextInt();

            boolean found = false;

            int index = -1;

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

                try {

                    if (array[i] == target) {

                        index = i;

                        found = true;

                        break;

                    }

                } catch (ArrayIndexOutOfBoundsException e) {

                    System.out.println("Array index out of bounds: " + e.getMessage());

                }

            }

            if (found) {

                System.out.println("element found: " + index);

            } else {

                System.out.println("element not found");

            }

        } catch (InputMismatchException e) {

            System.out.println("please enter an integer.");

        } finally {

            scanner.close();

        }

    }

}

**Qn2.**

class Stack {

    private int[] stack;

    private int top;

    private int capacity;

    public Stack(int size) {

        stack = new int[size];

        capacity = size;

        top = -1;

    }

    public void push(int value) {

        if (top == capacity - 1) {

            throw new StackOverflowError("Stack is full. Cannot push " + value);

        }

        stack[++top] = value;

    }

    public int pop() {

        if (top == -1) {

            throw new IllegalStateException("Stack is empty. Cannot pop.");

        }

        return stack[top--];

    }

    public boolean isEmpty() {

        return top == -1;

    }

    public boolean isFull() {

        return top == capacity - 1;

    }

    public int peek() {

        if (top == -1) {

            throw new IllegalStateException("Stack is empty. Cannot peek.");

        }

        return stack[top];

    }

}

public class StackDemo {

    public static void main(String[] args) {

        Stack stack = new Stack(5);

        try {

            System.out.println("Top element is: " + stack.peek());

            stack.push(10);

            stack.push(20);

            System.out.println("Top element is: " + stack.peek());

            stack.pop();

            stack.pop();

            stack.pop();

        } catch (StackOverflowError | IllegalStateException e) {

            System.out.println("Exception: " + e.getMessage());

        }

    }

}

**Qn3.**

public class ExceptionDemo {

    public static void main(String[] args) {

        int[] array = {1, 2, 3};

        try {

            System.out.println(array[3]);

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("Exception caught: " + e.getMessage());

        }

        String invalidNumber = "abc";

        try {

            int number = Integer.parseInt(invalidNumber);

        } catch (NumberFormatException e) {

            System.out.println("Exception caught: " + e.getMessage());

        }

    }

}

**Qn4.**

class CourseFullException extends Exception {

    public CourseFullException(String message) {

        super(message);

    }

}

class IneligibleStudentException extends Exception {

    public IneligibleStudentException(String message) {

        super(message);

    }

}

class Course {

    private int capacity;

    private int enrolledStudents;

    public Course(int capacity) {

        this.capacity = capacity;

        this.enrolledStudents = 0;

    }

    public void enrollStudent(int age) throws CourseFullException, IneligibleStudentException {

        if (age < 20) {

            throw new IneligibleStudentException("Age must be 20 or older.");

        }

        if (enrolledStudents >= capacity) {

            throw new CourseFullException("Course is full");

        }

        enrolledStudents++;

        System.out.println("successfully");

    }

}

// Example usage

public class EnrollmentSystem {

    public static void main(String[] args){

        Course course = new Course(2);

        try {

            course.enrollStudent(19);

        } catch (IneligibleStudentException e) {

            System.out.println("Exception caught: " + e.getMessage());

        } catch (CourseFullException e) {

            System.out.println("Exception caught: " + e.getMessage());

        }

        try {

            course.enrollStudent(21);

            course.enrollStudent(22);

            course.enrollStudent(23);

        } catch (IneligibleStudentException e) {

            System.out.println("Exception caught: " + e.getMessage());

        } catch (CourseFullException e) {

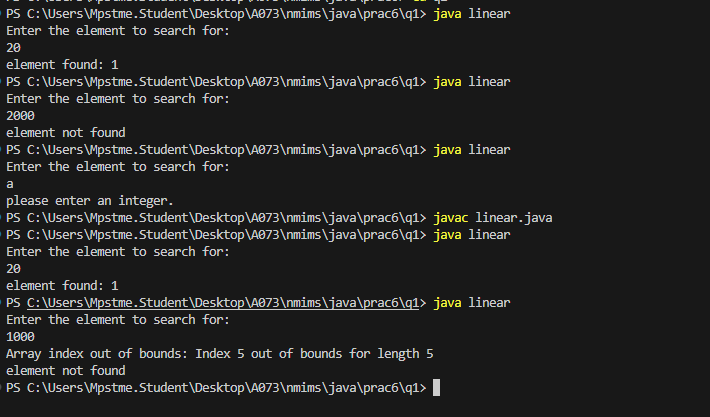
            System.out.println("Exception caught: " + e.getMessage());

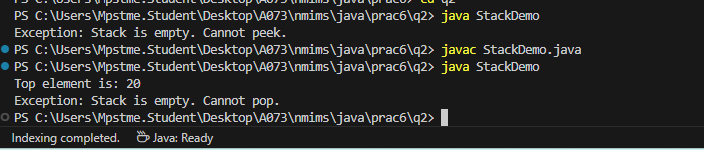
        }

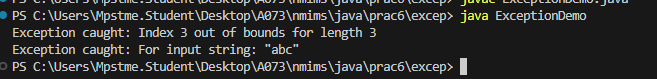
    }

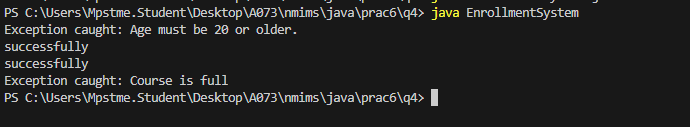
}

1. **Output:**

**Qn1.  
**

**Qn2.   
**

**Qn3.**

**Qn4.  
**

1. **Observation learning and conclusion**

successfully implemented Exception handeling

1. **Questions of Curiosity :**

**a. Final vs. Finally**

final: A keyword used to declare constants, prevent method overriding, or prevent inheritance of a class.

finally: A block used in exception handling that executes after a try-catch block, regardless of whether an exception was thrown.

**b. Throw vs. Throws**

throw: A keyword used to explicitly throw an exception from a method or block of code.

throws: A declaration in a method signature that indicates the method may throw specific exceptions, requiring callers to handle them.

**c. Checked Exceptions vs. Unchecked Exceptions vs. Errors**

Checked Exceptions: Exceptions that are checked at compile-time; must be either caught or declared in the method signature (e.g., IOException).

Unchecked Exceptions: Exceptions that are not checked at compile-time; can occur during runtime and do not need to be explicitly handled (e.g., NullPointerException).

Errors: Serious issues that are typically outside the control of the application (e.g., OutOfMemoryError); they are not meant to be caught or handled.