Experiment-5

Name: Shalini Kumari **UID: 22BET10202**
Branch: BE-IT **Section/Group: 22BET_IOT703/A**
Semester: 6th **Date of Performance: 19th Feb ,2025**
Subject Name: Project Based Learning in Java with Lab
Subject Code: 22ITH-329

Problem-1

1. **Aim:**

   Write a Java program to calculate the sum of a list of integers using autoboxing and unboxing. Include methods to parse strings into their respective wrapper classes (e.g., Integer.parseInt()).

2. **Objective:** To develop a Java program that efficiently calculates the sum of a list of integers using autoboxing and unboxing, while parsing string inputs into their respective wrapper classes (Integer) and handling invalid data through exception management.

3. **Code:**

```
import java.util.ArrayList;
import java.util.List;

public class SumOfIntegers {

    // Method to parse a list of strings into integers
    public static List<Integer> parseStringListToIntegers(List<String> stringList) {
        List<Integer> integerList = new ArrayList<>();
        for (String str : stringList) {
            try {
                // Parsing string to int and autoboxing into Integer
                integerList.add(Integer.parseInt(str));
            } catch (NumberFormatException e) {
                System.out.println("Invalid input skipped: " + str);
            }
        }
```

```
        return integerList;
    }

    // Method to calculate the sum of integers (demonstrates unboxing)
    public static int calculateSum(List<Integer> numbers) {
        int sum = 0;
        for (Integer num : numbers) {
            sum += num; // Unboxing: Integer to int
        }
        return sum;
    }

    public static void main(String[] args) {
        // Example list of strings (some valid, some invalid)
        List<String> inputStrings = List.of("10", "20", "abc", "30", "5", "xyz");

        // Parsing strings to integers
        List<Integer> numbers = parseStringListToIntegers(inputStrings);

        // Calculating the sum
        int totalSum = calculateSum(numbers);

        // Displaying the result
        System.out.println("The sum of valid integers is: " + totalSum);
    }
}
```

4. Output:

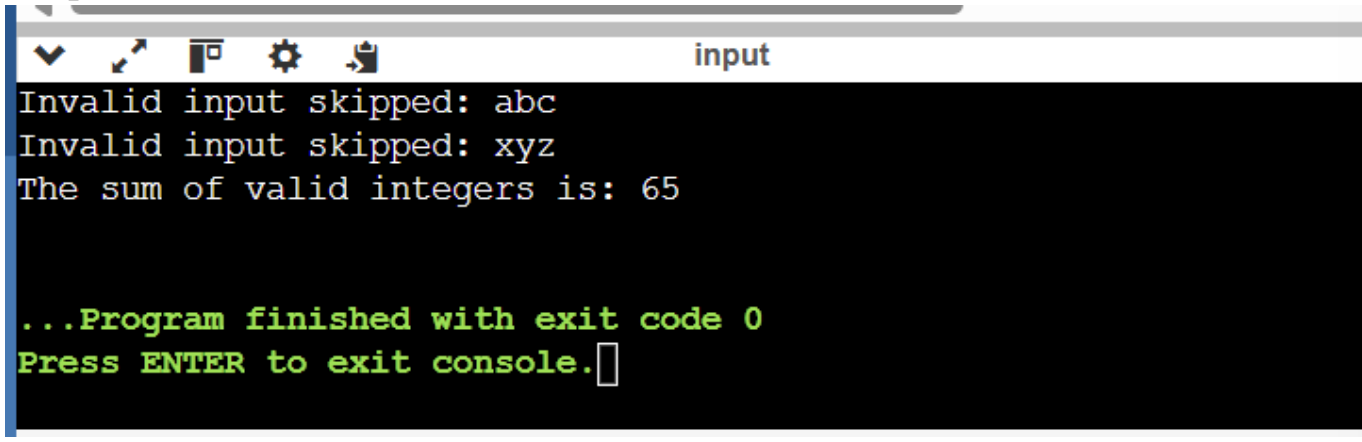

Fig.1. Sum of Integers

5. Learning Outcomes:

Understand and apply **autoboxing** and **unboxing** concepts in Java.

Utilize **wrapper classes** and methods like `Integer.parseInt()` for data parsing.

Implement **exception handling** to manage invalid inputs efficiently.

Process and manage data using **collections** (`ArrayList`) for optimized performance.

4o

Problem-2

1. Aim:

Create a Java program to serialize and deserialize a Student object

2. Objective:

To develop a Java program that demonstrates the concepts of **serialization** and **deserialization** by saving the state of a `Student` object to a file and restoring it, showcasing efficient file handling and object persistence in Java.

3. Code:

```
import java.io.*;

class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int id;

    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String toString() {
        return "Student{name='" + name + "', id=" + id + "}";
    }
}

public class SerializeDeserialize {
    public static void main(String[] args) {
        Student student = new Student("John Doe", 12345);

        // Serialize the student object
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("student.ser"))) {
            oos.writeObject(student);
            System.out.println("Serialization done.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialize the student object
```

```
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.ser"))) {
            Student deserializedStudent = (Student) ois.readObject();
            System.out.println("Deserialization done.");
            System.out.println(deserializedStudent);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

4. Output:

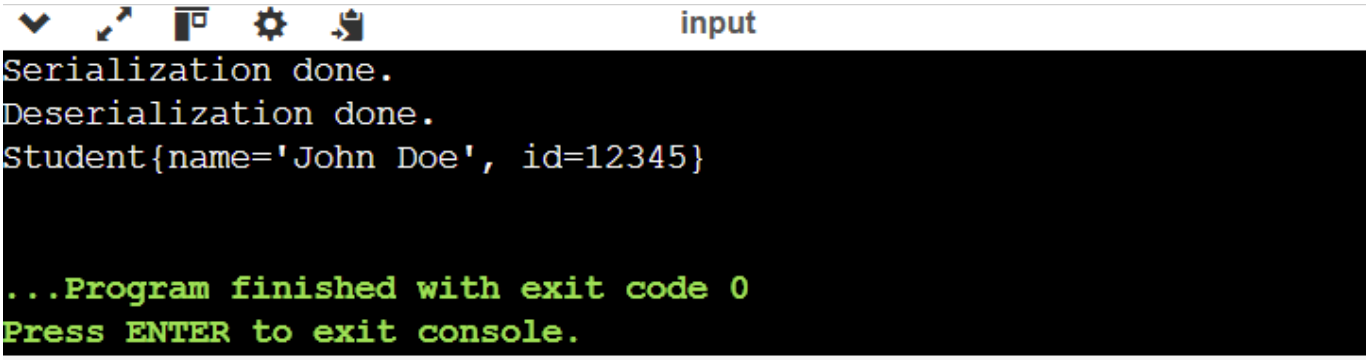

Fig.2. Successful Serialization and Deserialization of a Student Object

5. Learning Outcomes:

Understand how to save and restore objects using serialization and deserialization.

Learn to use the Serializable interface in Java.

Perform file handling with FileOutputStream and FileInputStream.

Problem-3

1. **Aim:** Create a menu-based Java application with the following options. 1.Add an Employee 2. Display All 3. Exit If option 1 is selected, the application should gather details of the employee like employee name, employee id, designation and salary and store it in a file. If option 2 is selected, the application should display all the employee details. If option 3 is selected the application should exit.
2. **Objective:** To develop a menu-driven Java application that allows adding employee details to a file, displaying all stored employee records, and exiting the program using efficient file handling techniques.
3. **Code:**

```
import java.io.*;
import java.util.*;

class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int id;
    private String designation;
    private double salary;

    public Employee(String name, int id, String designation, double salary) {
        this.name = name;
        this.id = id;
        this.designation = designation;
        this.salary = salary;
    }

    public String toString() {
        return "Employee{name='" + name + "', id=" + id + ", designation='" + designation + "',
salary=" + salary + "}";
    }
}

public class EmployeeManagement {
    private static final String FILE_NAME = "employees.ser";
    private static List<Employee> employees = new ArrayList<>();
```

```
public static void main(String[] args) {
    loadEmployees();
    Scanner scanner = new Scanner(System.in);

    while (true) {
        System.out.println("1. Add an Employee");
        System.out.println("2. Display All");
        System.out.println("3. Exit");
        System.out.print("Select an option: ");

        int option = scanner.nextInt();
        scanner.nextLine(); // consume newline

        switch (option) {
            case 1:
                addEmployee(scanner);
                break;
            case 2:
                displayAllEmployees();
                break;
            case 3:
                saveEmployees();
                System.exit(0);
                break;
            default:
                System.out.println("Invalid option. Please try again.");
        }
    }
}

private static void addEmployee(Scanner scanner) {
    System.out.print("Enter employee name: ");
    String name = scanner.nextLine();

    System.out.print("Enter employee id: ");
    int id = scanner.nextInt();
```

```
        scanner.nextLine(); // consume newline

        System.out.print("Enter designation: ");
        String designation = scanner.nextLine();

        System.out.print("Enter salary: ");
        double salary = scanner.nextDouble();

        employees.add(new Employee(name, id, designation, salary));
        System.out.println("Employee added.");
    }

    private static void displayAllEmployees() {
        for (Employee emp : employees) {
            System.out.println(emp);
        }
    }

    private static void saveEmployees() {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(FILE_NAME))) {
            oos.writeObject(employees);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @SuppressWarnings("unchecked")
    private static void loadEmployees() {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(FILE_NAME))) {
            employees = (List<Employee>) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            // Ignore - file may not exist yet
        }
    }
}
```

4. Output:

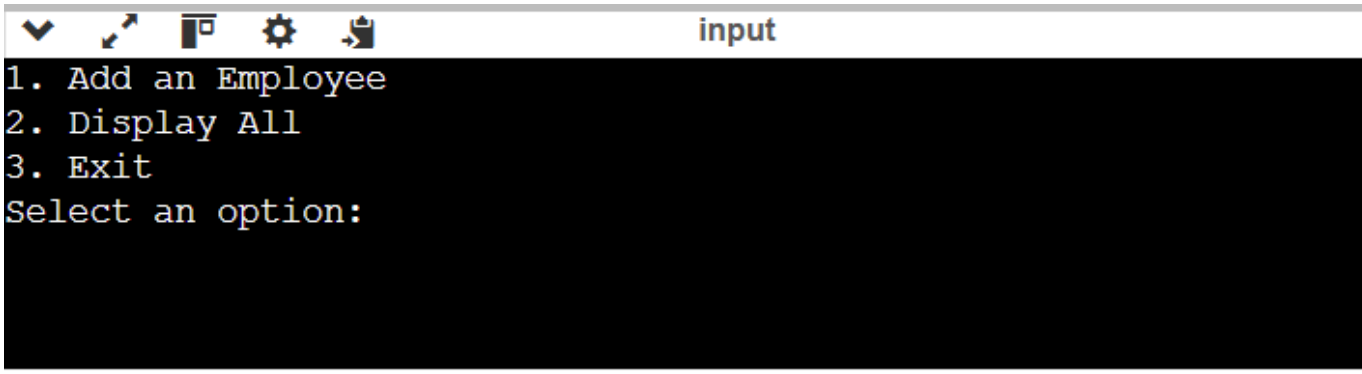

Fig.3. Displaying Employee Management Operations: Add, Display, and Exit

5. Learning Outcomes:

Understand how to create **menu-driven applications** in Java using control statements.

Perform **file handling** operations to read from and write employee details to a file.

Implement **object storage and retrieval** for managing multiple employee records.

Handle **user input** efficiently using classes like `Scanner`.

Apply **loops and conditional statements** to manage application flow and user choices.