Experiment 5

Student Name: Rajvardhan Singh **UID: 22BCS11638**
Branch: BE-CSE **Section/Group: 903-B**
Semester: 6th **Date of Performance:**
Subject Name: Project based Learning Java **Subject Code: 22CSH-359**

Problem :- 1(Easy-Level)

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:

- Demonstrate **autoboxing** (automatic conversion of primitive to wrapper class).
- Demonstrate **unboxing** (automatic conversion of wrapper class to primitive).
- Parse strings into integers using `Integer.parseInt()`.
- Calculate the sum of a list of integers.

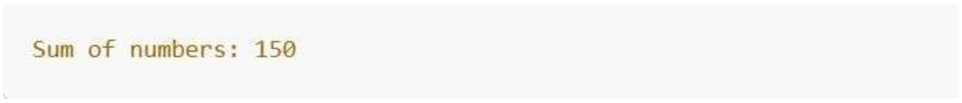
3. Algorithm:

- **Initialize a list** of integer values in string format.
- **Convert string values** to `Integer` objects using `Integer.parseInt()`.
- **Use autoboxing** to store primitive values in an `ArrayList<Integer>`.
- **Use unboxing** while iterating to compute the sum.
- **Print the total sum**.

4. Implementation :

```
import java.util.*;


public class AutoboxingSum {
    public static int calculateSum(List<Integer> numbers) {
        int sum = 0;
        for (Integer num : numbers) { // Unboxing (Integer → int)
            sum += num;
        }
        return sum;
    }

    public static void main(String[] args) {
        String[] numStrings = {"10", "20", "30", "40", "50"};
        List<Integer> numbers = new ArrayList<>();

        // Autoboxing: Converting primitive int to Integer and adding to list
        for (String numStr : numStrings) {
            numbers.add(Integer.parseInt(numStr)); // Parsing String → int (unboxing) →
Integer (autoboxing)
        }

        int sum = calculateSum(numbers);
        System.out.println("Sum of numbers: " + sum);
    }
}
```

5. Output:



```
Sum of numbers: 150
```

Problem:- 2(Medium-level)

1. Aim: Create a Java program to serialize and deserialize a Student object. The program should:

Serialize a Student object (containing id, name, and GPA) and save it to a file.

Deserialize the object from the file and display the student details.

Handle FileNotFoundException, IOException, and ClassNotFoundException using exception handling.

2. Objective:

- Implement **object serialization** using `ObjectOutputStream`.
- Implement **object deserialization** using `ObjectInputStream`.
- Use **exception handling** (`FileNotFoundException`, `IOException`, `ClassNotFoundException`).
- Store and retrieve a `Student` object from a file.

3. Algorithm:

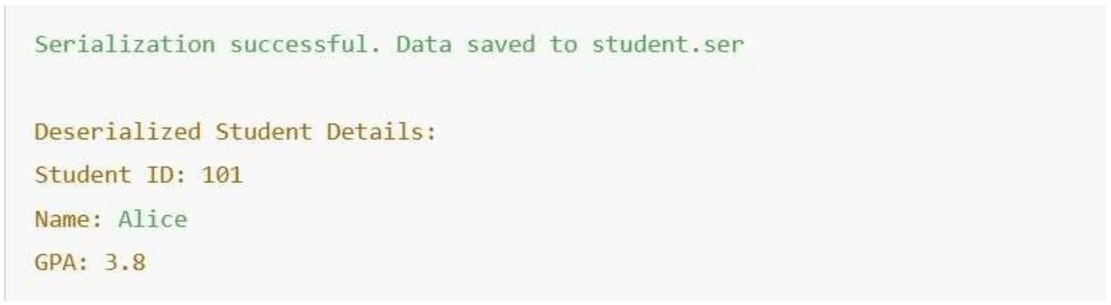
- **Define a** `Student` **class** implementing `Serializable`.
- **Create a method** to serialize a `Student` object and write it to a file.
- **Create a method** to deserialize the object from the file.
- **Use exception handling** for file operations.
- **Display student details** after deserialization.

4. Implementation/Code:

```
import java.io.*;

// Step 1: Create a Serializable Student class
```

```
class Student implements Serializable {
    private static final long serialVersionUID = 1L; // Ensure compatibility during
deserialization
    private int id;
    private String name;
    private double gpa;

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

    public void display() {
        System.out.println("Student ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("GPA: " + gpa);
    }
}

public class StudentSerialization {
    private static final String FILE_NAME = "student.ser";

    // Step 2: Serialize Student object
    public static void serializeStudent(Student student) {
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(FILE_NAME))) {
            out.writeObject(student);
            System.out.println("Serialization successful. Data saved to " + FILE_NAME);
        } catch (IOException e) {
            System.err.println("Error during serialization: " + e.getMessage());
        }
    }

    // Step 3: Deserialize Student object
    public static Student deserializeStudent() {
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(FILE_NAME))) {
            return (Student) in.readObject();
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + FILE_NAME);
        } catch (IOException e) {
            System.err.println("Error during deserialization: " + e.getMessage());
        } catch (ClassNotFoundException e) {
            System.err.println("Class not found: " + e.getMessage());
        }
        return null;
    }

    public static void main(String[] args) {
        // Create a Student object
        Student student1 = new Student(101, "Alice", 3.8);

        // Serialize the Student object
        serializeStudent(student1);
        // Deserialize the Student object
        Student deserializedStudent = deserializeStudent();
        if (deserializedStudent != null) {
            System.out.println("\nDeserialized Student Details:");
            deserializedStudent.display();
        }
    }
}
```

5. Output:



```
Serialization successful. Data saved to student.ser

Deserialized Student Details:
Student ID: 101
Name: Alice
GPA: 3.8
```

<u>Problem:- 3(Hard-level)</u>

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:

- Implement **file handling** for persistent data storage.
- Create a **menu-based system** for user interaction.
- Serialize and deserialize employee objects using **ObjectOutputStream** and **ObjectInputStream**.

- Handle exceptions (`IOException`, `FileNotFoundException`, `ClassNotFoundException`).

3. Algorithm:

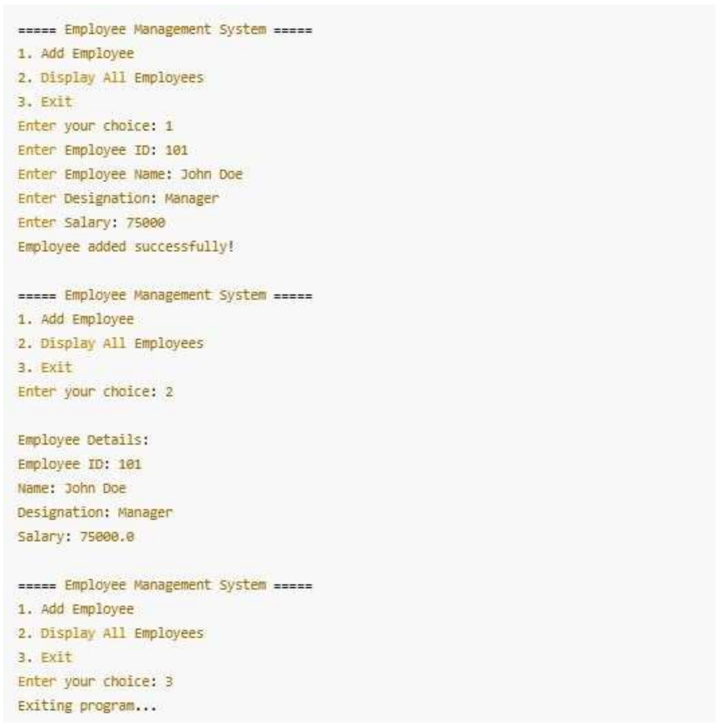
- **Create an** `Employee` **class** implementing `Serializable`.
- **Define methods** for adding employees and displaying employees.
- **Use a menu loop** to handle user input:
  - **Option 1:** Add an employee and store it in a file.
  - **Option 2:** Read all employees from the file and display them.
  - **Option 3:** Exit the program.
- **Use exception handling** for file operations.

4. Implementation/Code:

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

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

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

```
        this.salary = salary;
    }

    public void display() {
        System.out.println("\nEmployee ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Designation: " + designation);
        System.out.println("Salary: " + salary);
    }
}

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

    // Step 2: Load existing employees from file
    @SuppressWarnings("unchecked")
    public static void loadEmployees() {
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(FILE_NAME))) {
            employeeList = (List<Employee>) in.readObject();
        } catch (FileNotFoundException e) {
            employeeList = new ArrayList<>();
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Error loading employees: " + e.getMessage());
        }
    }
    // Step 3: Save employees to file
    public static void saveEmployees() {
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(FILE_NAME))) {
            out.writeObject(employeeList);
        } catch (IOException e) {
            System.err.println("Error saving employees: " + e.getMessage());
        }
    }

    // Step 4: Add a new employee
    public static void addEmployee(Scanner scanner) {
        System.out.print("Enter Employee ID: ");
        int id = scanner.nextInt();
        scanner.nextLine(); // Consume newline
        System.out.print("Enter Employee Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Designation: ");
        String designation = scanner.nextLine();
        System.out.print("Enter Salary: ");
        double salary = scanner.nextDouble();

        employeeList.add(new Employee(id, name, designation, salary));
        saveEmployees();
        System.out.println("Employee added successfully!");
    }
```

6. Output:



```
===== Employee Management System =====
1. Add Employee
2. Display All Employees
3. Exit
Enter your choice: 1
Enter Employee ID: 101
Enter Employee Name: John Doe
Enter Designation: Manager
Enter Salary: 75000
Employee added successfully!

===== Employee Management System =====
1. Add Employee
2. Display All Employees
3. Exit
Enter your choice: 2

Employee Details:
Employee ID: 101
Name: John Doe
Designation: Manager
Salary: 75000.0

===== Employee Management System =====
1. Add Employee
2. Display All Employees
3. Exit
Enter your choice: 3
Exiting program...
```

7. Learning Outcomes:

- **Object Serialization & File Handling** – Persisted and retrieved object data using serialization and exception handling.

- **Multi-threading & Synchronization** – Used synchronized threads to prevent race conditions in ticket booking.

- **Collection Framework & Data Management** – Utilized `ArrayList` for storing and managing objects dynamically.

- **Wrapper Classes & Autoboxing/Unboxing** – Converted between primitives and objects using Java wrapper classes.

- **Menu-Driven & Real-World Applications** – Built interactive CLI programs for employee management and ticket booking.