Experiment 5

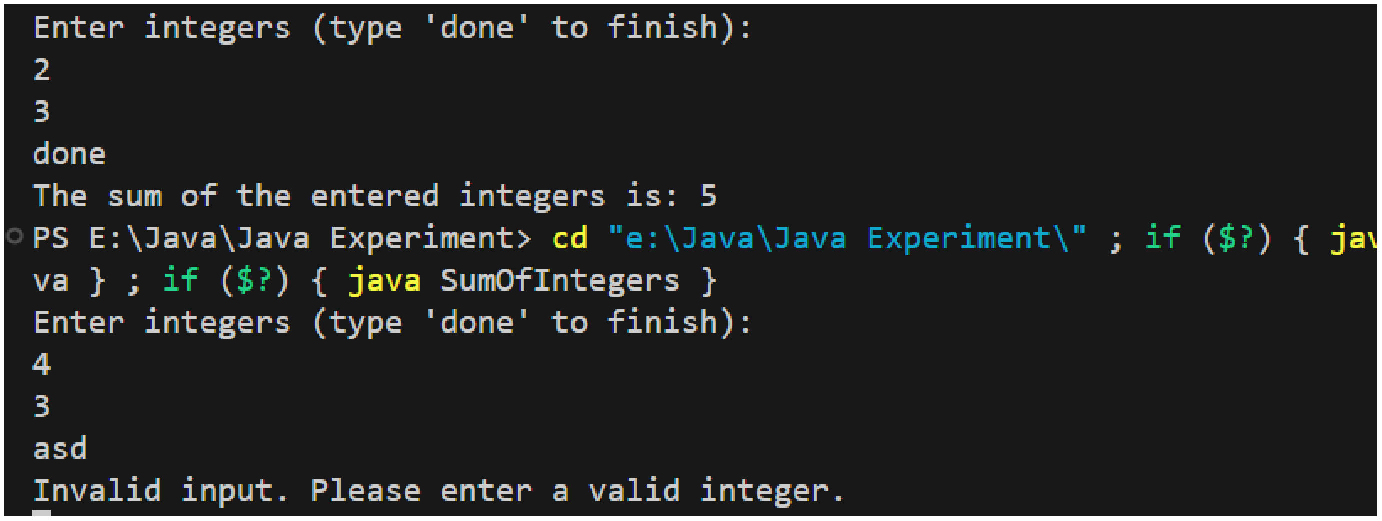
Student Name: Vishal Sah **UID:22BCS16978**
Branch:BE/CSE **Section/Group: 641/B**
Semester: 6 **Date of Performance:06/03/25**
Subject Name:JAVA **Subject Code: 22CSH-359**

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. **Implementation/Code:**

```java
import java.util.*;

public class SumOfIntegers {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int sum = 0;

        System.out.println("Enter integers (type 'done' to finish):");
        while (true) {
            String input = scanner.nextLine();
            if (input.equalsIgnoreCase("done")) break;
            try {
                sum += Integer.parseInt(input); // Autoboxing and Unboxing
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a valid integer.");
            }
        }

        System.out.println("The sum of the entered integers is: " + sum);
        scanner.close();
    }
}
```

3. Output


```
Enter integers (type 'done' to finish):
2
3
done
The sum of the entered integers is: 5
PS E:\Java\Java Experiment> cd "e:\Java\Java Experiment\" ; if ($?) { jav
va } ; if ($?) { java SumOfIntegers }
Enter integers (type 'done' to finish):
4
3
asd
Invalid input. Please enter a valid integer.
```

4. Learning Outcome

1. **Understanding Autoboxing and Unboxing:** Learners will comprehend how Java automatically converts between primitive types (e.g., `int`) and their corresponding wrapper classes (e.g., `Integer`).

2. **String Parsing to Wrapper Classes:** The program demonstrates how to convert a string input into an integer using `Integer.parseInt()`, including error handling for invalid inputs.

3. **Handling User Input:** Students will gain experience using the `Scanner` class to receive dynamic input from the user and control the input loop effectively.

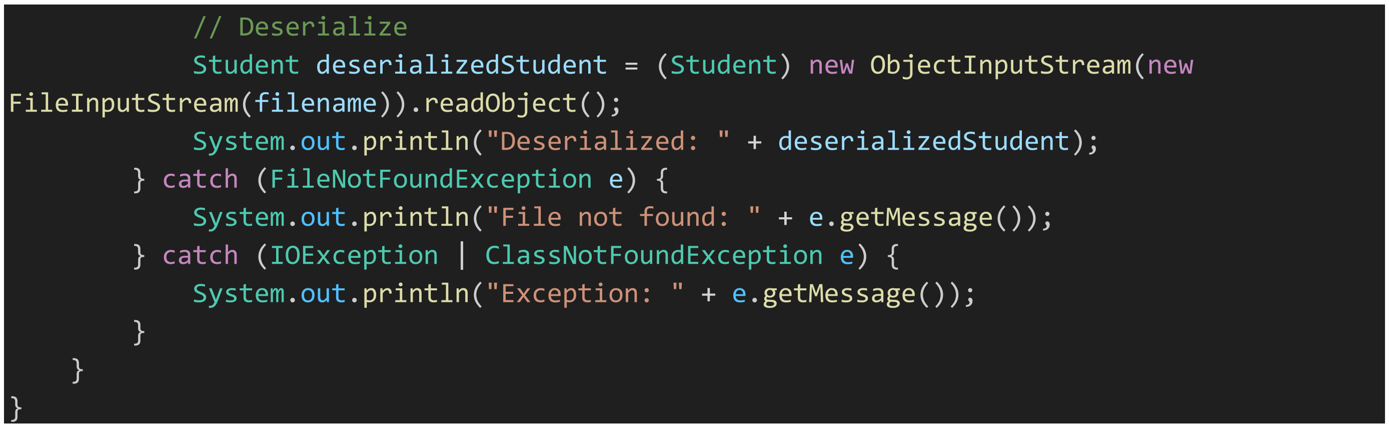
4. **Implementing Basic Exception Handling:** The example highlights the use of `try-catch` blocks to handle `NumberFormatException`, ensuring robust input validation.

Experiment – 5.2

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.

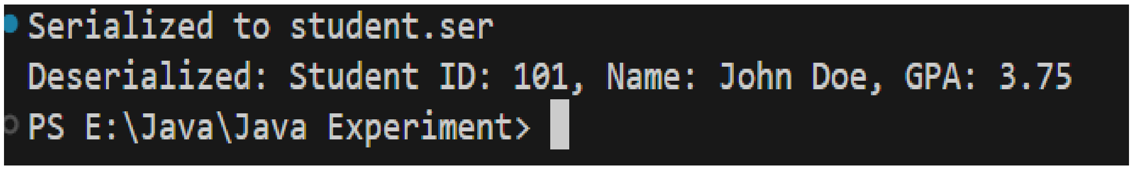
2- **Implementation/Code:**

```java
import java.io.*;

class Student implements Serializable {
    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;
    }

    @Override
    public String toString() {
        return "Student ID: " + id + ", Name: " + name + ", GPA: " + gpa;
    }
}

public class StudentSerialization {
    public static void main(String[] args) {
        String filename = "student.ser";
        Student student = new Student(101, "John Doe", 3.75);

        try {
            // Serialize
            new ObjectOutputStream(new FileOutputStream(filename)).writeObject(student);
            System.out.println("Serialized to " + filename);
```



```
            // Deserialize
            Student deserializedStudent = (Student) new ObjectInputStream(new
FileInputStream(filename)).readObject();
            System.out.println("Deserialized: " + deserializedStudent);
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

3- Output



```
Serialized to student.ser
Deserialized: Student ID: 101, Name: John Doe, GPA: 3.75
PS E:\Java\Java Experiment>
```

4- Learning Outcome

1. Understanding Serialization and Deserialization: Learn how to convert an object into a byte stream to save it to a file and restore it back using `ObjectOutputStream` and `ObjectInputStream`.

2. Implementing Exception Handling: Gain practical experience in handling FileNotFoundException`, `IOException`, and `ClassNotFoundException` using `try-catch` blocks.

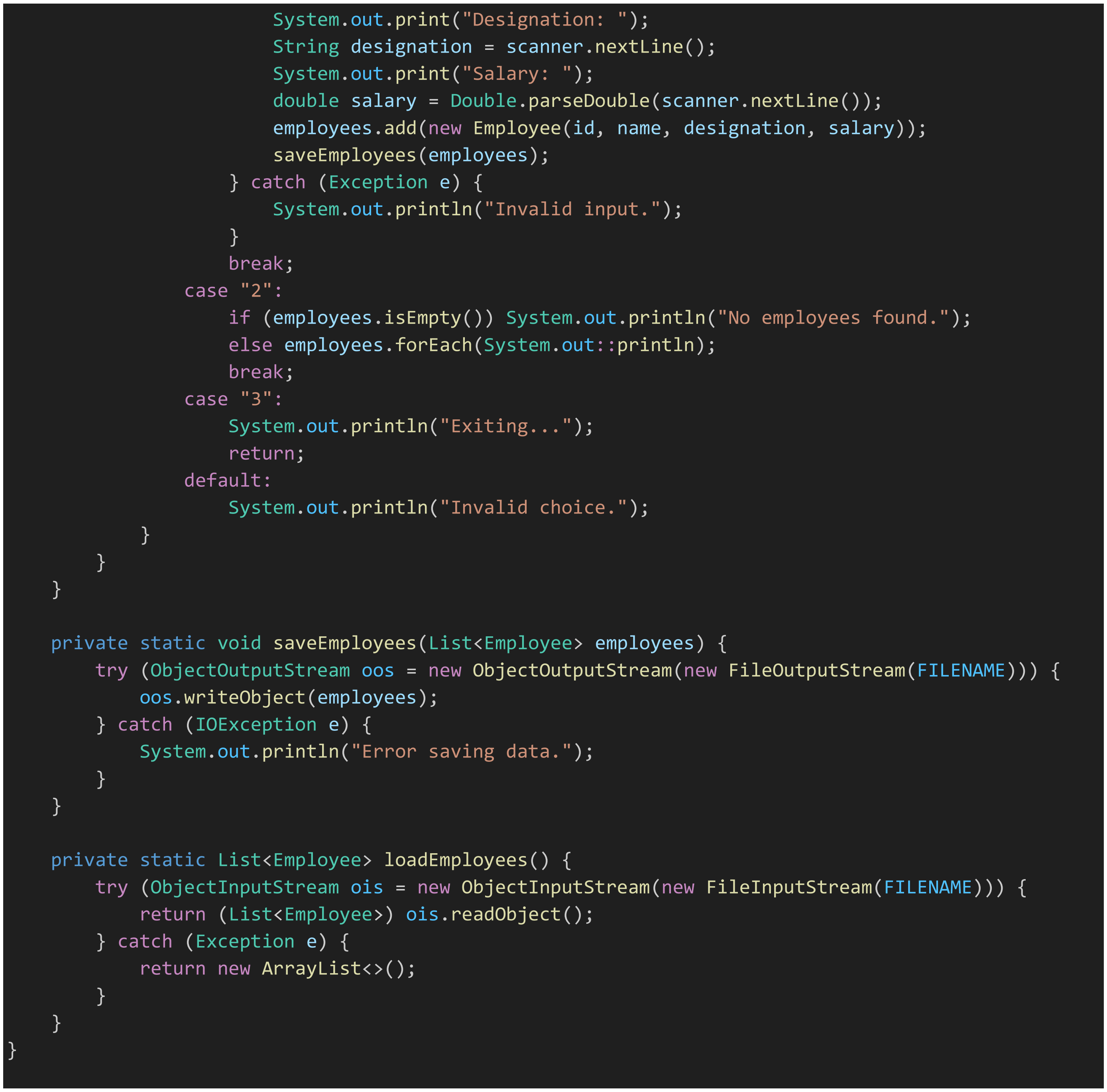
3. Using Java I/O Streams: Develop proficiency with file input and output operations, enabling effective data persistence.

Experiment 4

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.

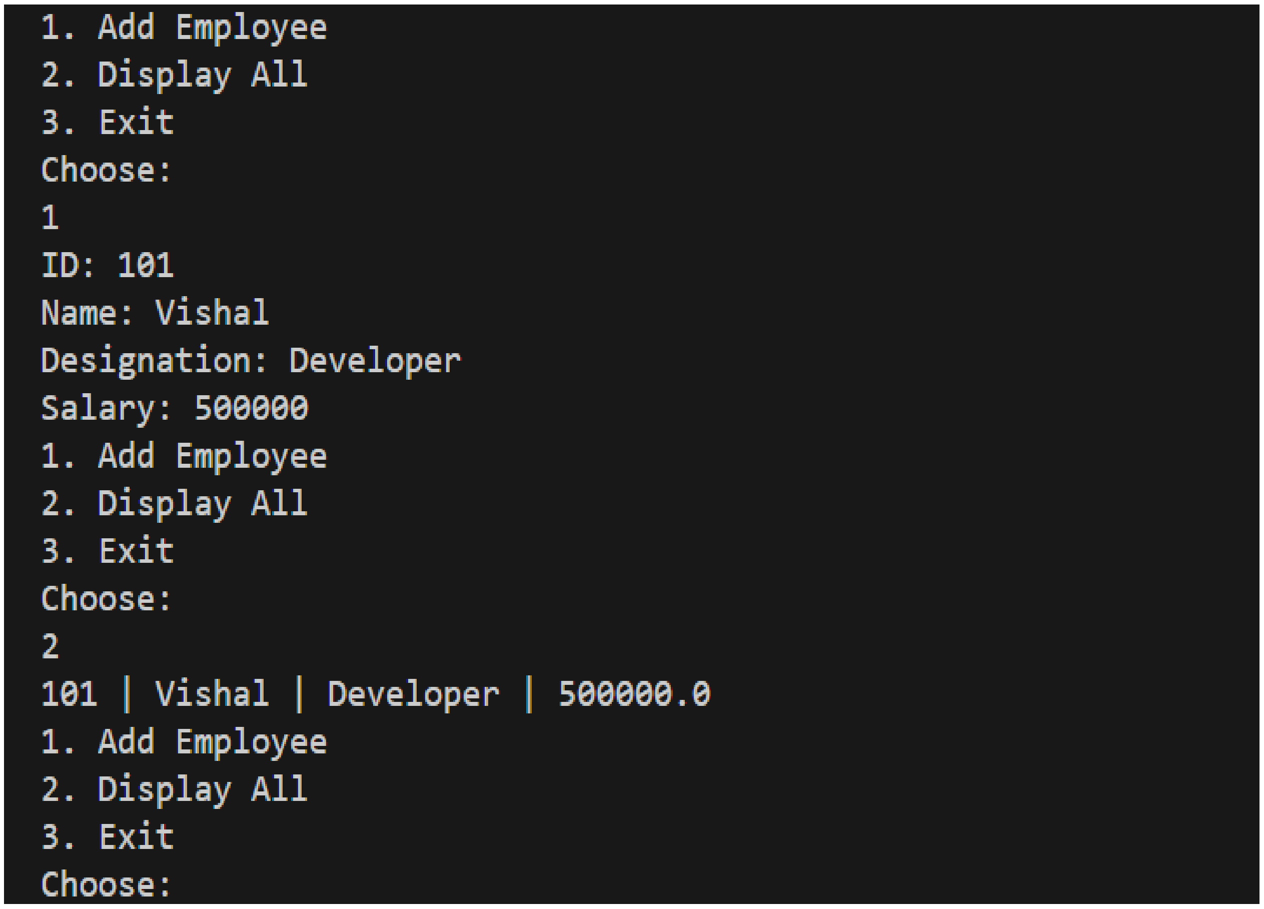
2- **Implementation/Code:**

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

class Employee implements Serializable {
    int id;
    String name, designation;
    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 String toString() {
        return id + " | " + name + " | " + designation + " | " + salary;
    }
}

public class EmployeeManagementApp {
    private static final String FILENAME = "employees.ser";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Employee> employees = loadEmployees();
        while (true) {
            System.out.println("1. Add Employee\n2. Display All\n3. Exit\nChoose: ");
            switch (scanner.nextLine()) {
                case "1":
                    try {
                        System.out.print("ID: ");
                        int id = Integer.parseInt(scanner.nextLine());
                        System.out.print("Name: ");
                        String name = scanner.nextLine();
```



```java
                        System.out.print("Designation: ");
                        String designation = scanner.nextLine();
                        System.out.print("Salary: ");
                        double salary = Double.parseDouble(scanner.nextLine());
                        employees.add(new Employee(id, name, designation, salary));
                        saveEmployees(employees);
                    } catch (Exception e) {
                        System.out.println("Invalid input.");
                    }
                    break;
                case "2":
                    if (employees.isEmpty()) System.out.println("No employees found.");
                    else employees.forEach(System.out::println);
                    break;
                case "3":
                    System.out.println("Exiting...");
                    return;
                default:
                    System.out.println("Invalid choice.");
            }
        }
    }

    private static void saveEmployees(List<Employee> employees) {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(FILENAME))) {
            oos.writeObject(employees);
        } catch (IOException e) {
            System.out.println("Error saving data.");
        }
    }

    private static List<Employee> loadEmployees() {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(FILENAME))) {
            return (List<Employee>) ois.readObject();
        } catch (Exception e) {
            return new ArrayList<>();
        }
    }
}
```

3- Output



```
1. Add Employee
2. Display All
3. Exit
Choose:
1
ID: 101
Name: Vishal
Designation: Developer
Salary: 500000
1. Add Employee
2. Display All
3. Exit
Choose:
2
101 | Vishal | Developer | 500000.0
1. Add Employee
2. Display All
3. Exit
Choose:
```

4- Learning Outcome :

1. **Serialization and Deserialization** Understand how to save and retrieve objects using Java's `ObjectOutputStream` and `ObjectInputStream` for persistent storage.

2. **Menu-Driven Application Design:** Gain experience in creating interactive console-based applications with dynamic user input handling.

3. **Exception Handling** Learn to manage input and file-related exceptions (`IOException`, `ClassNotFoundException`, `NumberFormatException`) to ensure the program runs smoothly.

4. **Collection Framework Usage:** Utilize `List<Employee>` to manage a dynamic collection of employee objects, demonstrating practical use of Java's collection framework.