Java Lab Assignment 5

A Java multithreaded application that simulates a banking system

Problem Statement

Design and implement a **Student Record Management System** using **Java** that allows for the management of student records (add, update, delete, search, and view) with persistent storage. The application must support **exception handling**, **file handling** (to store and retrieve data), **multithreading** (to simulate loading), and must leverage the **Java Collections Framework**. The system should allow sorting of students by marks, provide the option to search and delete student records, and display the sorted list of students. Additionally, the system should use **OOP** concepts like inheritance, abstraction, and interfaces to ensure modular and reusable code.

Learning Outcomes

Upon completion of this assignment, the students will be able to:

1. Design and implement an object-oriented system using classes, inheritance, and interfaces.
2. Use exception handling to ensure safe program execution and validation.
3. Implement file I/O for persistent data storage using Java's **BufferedReader** and **BufferedWriter**.
4. Use **Java Collections** (List, Map, Set) to manage and manipulate student records.
5. Sort student records using **Comparator** and display records via **Iterator**.
6. Implement and understand **multithreading** for responsive user interaction.
7. Apply **custom exceptions** and perform input validation.
8. Understand **modular programming** with packages for better code organization and reusability.

Class Hierarchy & Data Types

Class Hierarchy:

1. **Person** (abstract class)
    - Fields: name, email
    - Methods: displayInfo() (abstract)
2. **Student** (extends **Person**)

- Fields: rollNo, course, marks, grade
- Methods: inputDetails(), displayDetails(), calculateGrade()

3. **StudentManager** (implements **RecordActions** interface)
   - Methods: addStudent(), deleteStudent(), updateStudent(), searchStudent(), viewAllStudents()
4. **Loader** (implements **Runnable**)
   - Methods: run() (for simulating loading in multithreading)

Data Types:

- String: For student name, email, course.
- int: For rollNo.
- double: For marks.
- List<Student>: For storing students.
- Map<Integer, Student>: For storing students in a map with rollNo as the key.
- Thread: For multithreading to simulate a loading process.

Detailed Instructions

1. **Core Design**: Create the abstract class **Person** with basic fields like name and email, and extend it in the **Student** class. Include methods like inputDetails(), displayDetails(), and calculateGrade() based on marks.
2. **Interface Implementation**: Create a **RecordActions** interface and implement it in the **StudentManager** class. Include methods like addStudent(), deleteStudent(), updateStudent(), searchStudent(), and viewAllStudents(). Implement validations for duplicate rollNo.
3. **Exception Handling**: Implement appropriate **try-catch-finally** blocks for handling invalid input (marks outside the valid range, empty fields, invalid rollNo) and create custom exceptions like **StudentNotFoundException**.
4. **File I/O**: Implement **BufferedReader** and **BufferedWriter** to load and save student records from/to a file (students.txt). Handle file reading and writing with exception handling.
5. **Multithreading**: Use a Thread to simulate a delay when performing actions like adding or saving records, showing the loading state.
6. **Sorting and Display**: Implement sorting of student records by marks in descending order using **Comparator**. Use **Iterator** to display the records in a sorted order.

Expected Output

The program should output the following results based on user interaction:

Example Output:

```
===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 1
Enter Roll No: 101
Enter Name: Rahul
Enter Email: rahul@mail.com
Enter Course: B.Tech
Enter Marks: 85.0

===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 2
Roll No: 101
Name: Rahul
Email: rahul@mail.com
Course: B.Tech
Marks: 85.0
----------------------

===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 3
Enter name to search: Rahul
Student Info:
Roll No: 101
Name: Rahul
Email: rahul@mail.com
```

Course: B.Tech
Marks: 85.0

===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 4
Enter name to delete: Rahul
Student record deleted.

===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 5
Sorted Student List by Marks:
Roll No: 101
Name: Rahul
Email: rahul@mail.com
Course: B.Tech
Marks: 85.0

===== Capstone Student Menu =====
1. Add Student
2. View All Students
3. Search by Name
4. Delete by Name
5. Sort by Marks
6. Save and Exit
Enter choice: 6
Saved and exiting.

Guidelines to Students

1. **Code Structure**:
    - Ensure all classes are correctly placed within their respective packages (model, service, util).

- Use object-oriented principles like inheritance and interfaces for clean code.

2. **Modularity**:
   - Keep methods short, clear, and reusable.
   - Handle all exceptions with meaningful messages.
3. **File Handling**:
   - Handle file reading/writing operations with **BufferedReader** and **BufferedWriter**.
   - Ensure the program can load existing student records on startup and save updated records before exiting.
4. **Multithreading**:
   - Simulate a realistic loading experience by using a Thread class.

Improvements/Adjustments

1. **GUI Enhancement**:
   - Optional enhancement: Implement a simple **GUI** using **JavaFX** or **Swing** to replace the console-based interface.
2. **Advanced Sorting**:
   - Add sorting features for Student class (e.g., sorting by name or course).
3. **Custom Data Validation**:
   - Use more complex validations like validating email format and proper name formatting.
4. **Database Integration**:
   - (Optional) Integrate **SQLite** or another database for more robust data management.

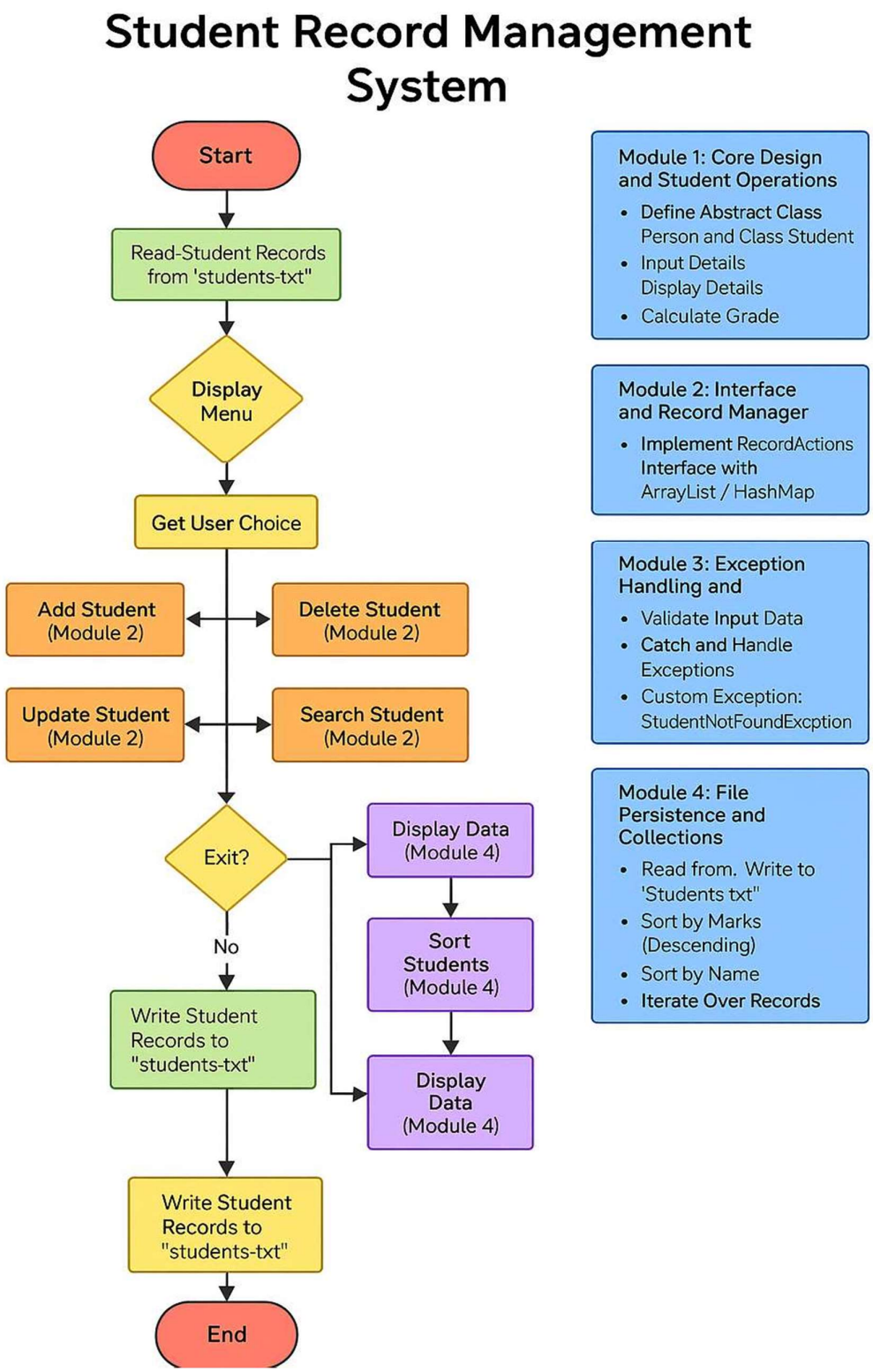
Submission Guidelines

1. **Code Submission**:
   - Submit the entire project folder with all Java source files.
   - Ensure proper indentation and readable code.
2. **Documentation**:
   - Include a brief **README** explaining how to run the project and how it handles different operations.
3. **File Storage**:
   - Make sure all student records are properly loaded from and saved to students.txt.

Performance Metrics (Out of 10 Marks)

Criteria	Marks
Core Design and Implementation	3
Interface and Record Manager	2
Exception Handling and Validation	1.5
File Handling and Persistence	1.5
Sorting and Display	1
Multithreading and Responsiveness	1

Flow Chart:


Student Record Management System
Start
Read-Student Records from 'students-txt"
Display Menu
Get User Choice
Add Student (Module 2)
Delete Student (Module 2)
Update Student (Module 2)
Search Student (Module 2)
Exit?
No
Write Student Records to "students-txt"
Display Data (Module 4)
Sort Students (Module 4)
Display Data (Module 4)
Write Student Records to "students-txt"
End
Module 1: Core Design and Student Operations
Define Abstract Class Person and Class Student
Input Details Display Details
Calculate Grade
Module 2: Interface and Record Manager
Implement RecordActions Interface with ArrayList / HashMap
Module 3: Exception Handling and
Validate Input Data
Catch and Handle Exceptions
Custom Exception: StudentNotFoundExcption
Module 4: File Persistence and Collections
Read from. Write to 'Students txt"
Sort by Marks (Descending)
Sort by Name
Iterate Over Records