Java OOP Project: School Registration System

1. Project Introduction & Objectives

Welcome to the School Registration System project! In this assignment, you will simulate a real-world software engineering task: building a **Desktop Administrative Application** for a college registrar.

This project is comprehensive, designed to bridge the gap between theoretical Java knowledge and practical application development.

Key Learning Outcomes:

- **MVC Architecture**: Separating Data (Models), Logic (Services), and Interface (Views).
- **Java Swing GUI**: Building event-driven desktop applications with Tabs, Tables, and Forms.
- **Complex Business Logic**: Implementing interdependent rules (e.g., *Instructor Load* affects *Class Creation*).
- **Exception Handling**: Gracefully handling user errors via GUI popups.

2. Team Organization & Reporting

This is a collaborative group project. You must adhere to the following organizational structure:

1. **Team Size**: Students must form teams of **3 members**.
2. **Team Leader**: Each group must select one **Team Leader**.
   The Leader is responsible for setting up the repository (GitHub/GitLab) and handling the final submission.
3. **Collaboration**: All team members are expected to write code.

Required Deliverable: Team Project Report

In addition to the source code, every team must submit a single **Project Report (PDF)**. This report must include:

- **Task Distribution Table**: A clear breakdown of who did what.
  - *Example*: "Student A: StudentService & Models", "Student B: GUI Dashboard", "Student C: Registration Logic".
- **Problems & Challenges**: A section describing technical hurdles the team faced (e.g., "We struggled with parsing the pipe delimiter in CSVs") and how you solved them.
- **Architecture Diagram**: A simple diagram showing how your Classes interact.

3. System Architecture

You must implement a **Service-Layer Architecture**. This ensures your code is modular and testable.

```
[ DATA LAYER ]      [ SERVICE LAYER ]          [ VIEW LAYER ]
(.csv Files)   --->  (Business Logic)     --->  (Java Swing GUI)
    |                 |                      |
students.csv        StudentService            DashboardPanel
```

```
courses.csv        RegistrationService       CreateSectionForm
instructors.csv     (Validates Rules)         StudentListTable
```

4. Data Structure & Model Classes

Package: com.school.model

All models must include private fields, public constructors, getters/setters, and a toString() method for debugging.

A. Student

- **Fields**:
  - String id (Unique ID, e.g., "1001")
  - String name (e.g., "John Smith")
  - String major
  - List<ClassSession> enrolledClasses
- **Required Method**:
  - public int getCurrentCredits(): Iterates through enrolledClasses and sums the credits of the associated courses.

B. Instructor

- **Fields**:
  - String id (e.g., "I100")
  - String name (e.g., "Dr. Alan Turing")
  - List<String> qualifiedCourses (List of Course IDs they can teach)
  - List<ClassSession> teachingAssignment (Sections they are currently teaching)
- **Required Methods**:
  - public boolean canTeach(Course c): Returns true if c.getCourseId() is in qualifiedCourses.
  - public int getCurrentLoad(): Sums the credits of all sections in teachingAssignment.

C. Course

- **Fields**: courseId (e.g., "CS101"), name, credits (int).
- **Purpose**: Immutable representation of a catalog entry.

D. Classroom

- **Fields**: roomNumber, hasComputer (boolean), hasSmartboard (boolean).

E. ClassSession (The "Section")

- **Fields**:
  - Course course
  - Instructor instructor

- Classroom classroom
- int sectionNumber
- int maxCapacity
- List<Student> enrolledStudents

- **Behavior**:
  - public boolean isFull(): Returns true if enrollment size >= max capacity.

5. Service Layer & Logic

Package: com.school.service

A. Data Loaders (StudentService, InstructorService, etc.)

These services are responsible for initializing the system.

- **Requirement**: Use a Map<String, Model> (e.g., Map<String, Student>) to store data. This allows O(1) retrieval by ID.
- **Parsing Hint**: When reading instructors.csv (I100,Dr.Turing,CS101|CS102), use line.split("\\|") to handle the pipe delimiter for courses.

B. RegistrationService (The Core Engine)

This class manages the interaction between models.

Method 1: findEligibleInstructors(Course c)

- **Logic**: Iterate through all loaded Instructors. Return a List<Instructor> where instr.canTeach(c) is true.

Method 2: createClassSection(...)

- **Signature**:
  public ClassSession createClassSection(Course c, Instructor i, Classroom r, int capacity) throws SchoolException

- **Logic**:
  1. Check i.canTeach(c). If false, throw Exception ("Instructor not qualified").
  2. Check i.getCurrentLoad() + c.getCredits() > 9. If true, throw Exception ("Instructor load exceeded").
  3. If valid, create the session, add it to i.teachingAssignment, and store it in the system.

Method 3: registerStudent(Student s, ClassSession section)

- **Logic**:
  1. Check section.isFull().
  2. Check s.getCurrentCredits() + section.getCourse().getCredits() > 18.
  3. Check if s is already in the class (duplicate).
  4. If valid, add s to section.enrolledStudents AND add section to s.enrolledClasses.

6. User Interface (Java Swing)

Package: com.school.view

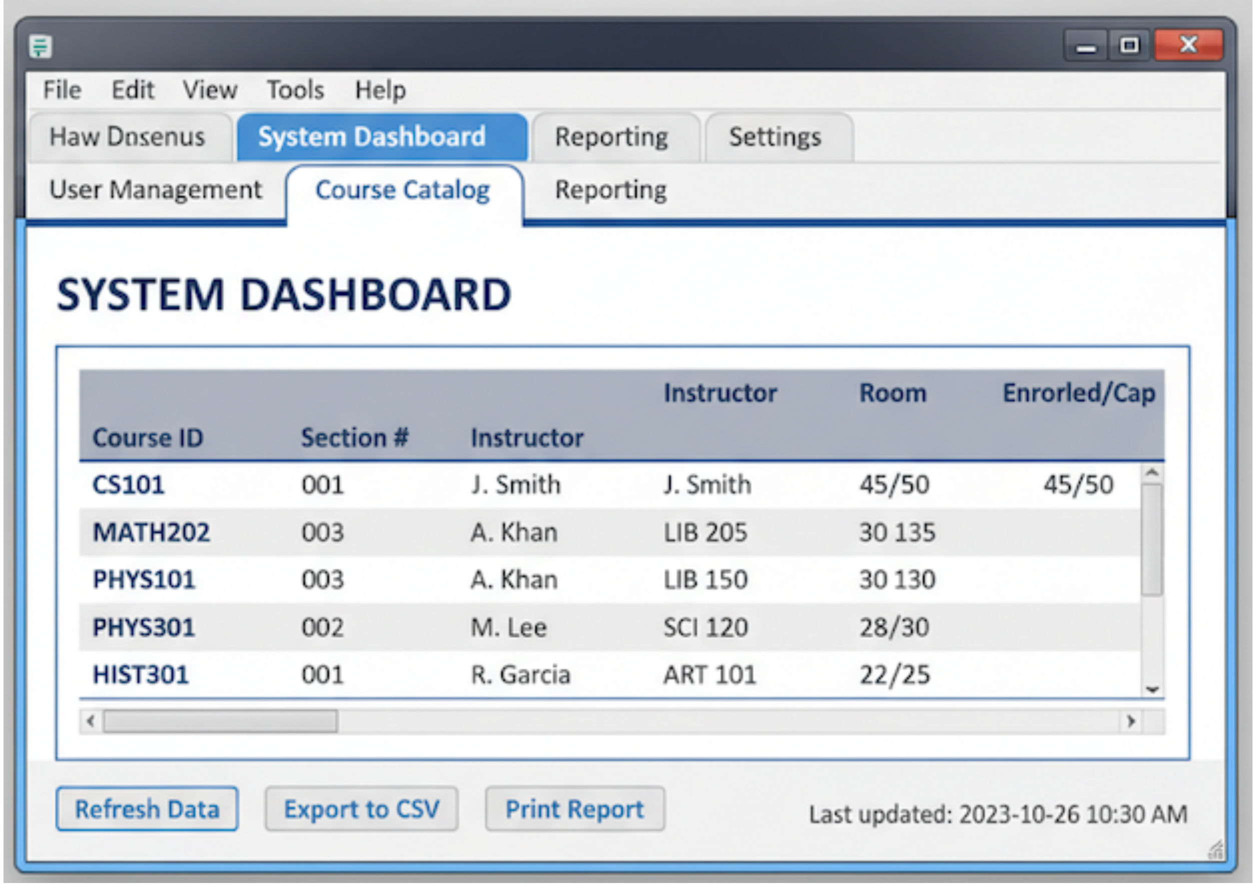
You will build a MainFrame that extends JFrame. The layout should use a JTabbedPane to separate different administrative tasks.

Tab 1: System Dashboard (Data View)

Objective: Visualize the loaded data.

- **Components**:
  - A JTable displaying "Active Class Sections".
  - Columns: [Course ID] [Section #] [Instructor] [Room] [Enrolled/Cap].
  - *Tip*: Use DefaultTableModel to allow adding rows dynamically.

Figure 1: Conceptual layout of the Dashboard tab showing active sections.



			Instructor	Room	Enrorled/Cap
Course ID	Section #	Instructor			
CS101	001	J. Smith	J. Smith	45/50	45/50
MATH202	003	A. Khan	LIB 205	30 135	
PHYS101	003	A. Khan	LIB 150	30 130	
PHYS301	002	M. Lee	SCI 120	28/30	
HIST301	001	R. Garcia	ART 101	22/25	

Last updated: 2023-10-26 10:30 AM

Tab 2: Administration (Action View)

Objective: Forms to create data.

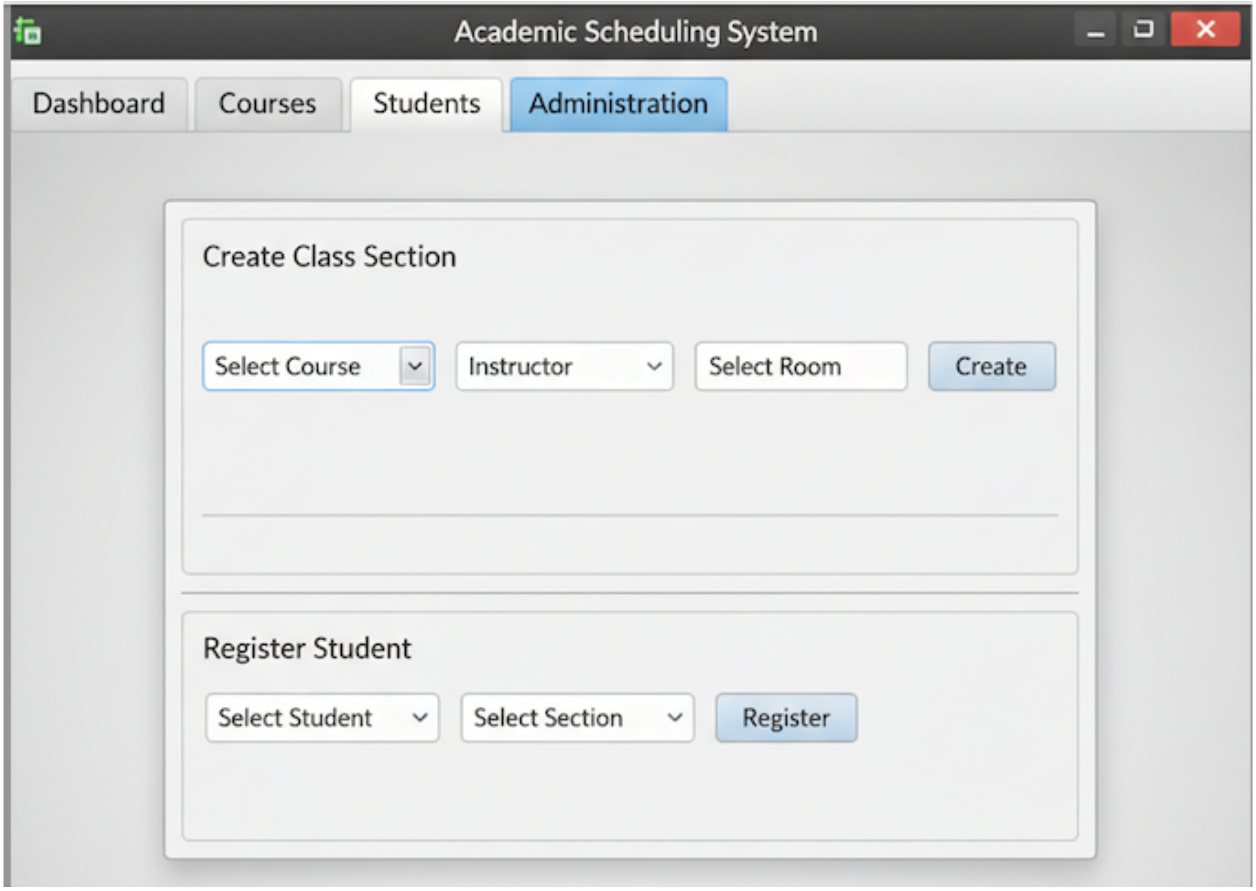
Form A: Create Class Section

- **Layout**: GridLayout or GridBagLayout.
- **Components**:
  - lblCourse + cmbCourse (JComboBox containing Course Objects).
  - lblInstructor + cmbInstructor (JComboBox).
  - lblRoom + cmbRoom.
  - btnCreate ("Create Section").
- **Dynamic Logic (Crucial)**:
  - Add an ActionListener to cmbCourse.
  - When a course is selected (e.g., "CS101"), clear cmbInstructor and repopulate it **only** with instructors returned by registrationService.findEligibleInstructors(selectedCourse).

Form B: Register Student

- **Components**:
  - cmbStudent (List of all students).
  - cmbSection (List of currently created ClassSessions).
  - btnRegister.

Figure 2: Conceptual layout of the creation form with dynamic dropdowns.



7. Business Rule Examples

Use these scenarios to test your application.

Scenario 1: Instructor Load Limit (9 Credits)

- **Context**: Dr. Newton teaches PHY101 (4cr) and MATH101 (4cr). Total Load = 8.
- **Action**: Try to assign Dr. Newton to MATH201 (3cr).
- **Result**: 8 + 3 = 11. **FAIL**.
- **GUI Feedback**: Show JOptionPane.showMessageDialog with message: *"Error: Dr. Newton has reached the maximum teaching load."*

Scenario 2: Student Credit Limit (18 Credits)

- **Context**: Student "Jane Doe" has 15 credits.
- **Action**: Try to register Jane for "Calculus I" (4cr).
- **Result**: 15 + 4 = 19. **FAIL**.
- **GUI Feedback**: *"Error: Registration would exceed maximum semester credits (18)."*

Scenario 3: Instructor Qualification

- **Context**: Dr. Turing is qualified for "CS101" and "CS102".
- **Action**: Try to assign Dr. Turing to "History 101".
- **Result**: **FAIL**.
- **Note**: Ideally, your GUI shouldn't even *show* Dr. Turing in the dropdown for History 101 (see "Dynamic Logic" above).

8. Required Data Files (Templates)

Create a data/ folder and ensure these files exist before running.

instructors.csv

```
I100,Dr. Alan Turing,CS101|CS102
I200,Dr. Isaac Newton,PHY101|MATH101|MATH201
I300,Prof. Shakespeare,ENG101|HIST101
```

students.csv

```
1001,John Smith,Computer Science
1002,Jane Doe,Mathematics
```

courses.csv

```
CS101,Intro to Java,4
```

```
MATH101,Calculus I,4
ENG101,English Comp,3
```

classrooms.csv

```
RM101,true,true
RM102,true,false
AUD100,false,true
```

9. Extra Credit Options (5 extra Points each)

You may implement **one** of the following features for extra credit. Each option is designed to test a specific advanced concept.

Option 1: Data Persistence (Saving State)

Currently, the system loses all created sections when closed.

- **Feature**: Add a "Save Data" button to the Dashboard.
- **Requirement**: When clicked, write the current state of Active Class Sections to a new CSV file (e.g., saved_sections.csv).
- **Bonus**: On startup, check if this file exists and reload the sections automatically.

Option 2: Search and Filter (Advanced Swing)

Managing 1000 students is hard with a simple list.

- **Feature**: Add a **Search Bar** and **Filter Dropdown** (by Major) to the Student List tab.
- **Requirement**: As the user types in the search bar, the JTable should update dynamically to show only matching students.
- **Hint**: Research TableRowSorter and RowFilter in Java Swing.

Option 3: Time Conflict Validation

Real schools have schedules.

- **Feature**: Add Day (Mon, Tue, etc.) and TimeSlot (e.g., 10:00 AM) to the ClassSession model.
- **Requirement**:
  - **Instructor Conflict**: Prevent creating a section if the Instructor is already teaching at that time.
  - **Student Conflict**: Prevent registering a student if they are already in a class at that time.

Option 4: Waitlist System

Classes fill up quickly.

- **Feature**: Implement a Waitlist Queue.

- **Requirement**:
  - If a class is full, allow the student to join a "Waitlist" (use a Queue<Student> data structure).
  - Add a "Drop Class" button. If a registered student drops, automatically move the first student from the Waitlist into the class.

Option 5: Student GPA Calculator

- **Feature**: Allow assigning grades to students.
- **Requirement**:
  - Add a Map<Course, Double> to the Student model to store grades (0.0 - 4.0).
  - Create a "Grading" tab where an Admin can assign a grade to a student for a specific class.
  - Calculate and display the student's **GPA** in the Student List table.

10. Alternative: Easier Terminal Version (Max 30 points if chose this version)

If you are struggling with the Java Swing GUI, you may opt for this simplified text-based version. The core logic and Service layer remain exactly the same.

Package: com.school.app

Implement a text-based menu using java.util.Scanner.

Requirements:

1. **Main Menu Loop**: The application must run in a loop until "Exit" is chosen.

   ```
   --- School System Menu ---
   1. Load Data (CSVs)
   2. View Students / Instructors / Courses
   3. Create Class Section
   4. Register Student
   5. View Section Reports
   6. Exit
   ```

2. **Interactive Flows**:
   - **Create Section**:
     1. User types Course ID (e.g., "CS101").
     2. System prints list of **qualified** instructors.
     3. User types Instructor ID.
     4. System checks "9 Credit Limit". If pass -> Success. If fail -> Print Error.
   - **Register Student**:
     1. User types Student ID.
     2. System shows available sections.
     3. User selects Section.
     4. System checks "18 Credit Limit" & "Capacity".
3. **Output Formatting**:
   - Use System.out.printf to print neat tables.

11. Generative AI Policy

You are allowed to use Generative AI tools to assist you in completing this project. However, strictly adhering to the following rules is mandatory:

1. **Full Comprehension Required**: You must be able to explain **every line of code** you submit. Using code you do not understand is a violation of the academic integrity policy for this course.
2. **Assessment Method**: Your grade for this project will **not** be determined solely by the code you submit. Instead, it will be determined by a **code review interview** with the instructor.
3. **The Interview**: During the interview, you will be asked to explain specific parts of your implementation, modify logic on the spot, or debug a scenario. If you cannot explain how your code works, you will **not receive credit**, regardless of whether the application runs correctly.

12. Interview & Evaluation Process

This project is graded based on a live interview. Attendance is mandatory.

1. **Mandatory Attendance**: Every team member must attend the interview. **Any member who does not attend the interview will receive a grade of 0**, regardless of their contribution to the codebase or report.
2. **Live Demonstration**:
   - You must run the project on your machine during the interview.
   - You must demonstrate **all functionality** (loading data, showing tables, creating sections, handling errors, etc.).
3. **Video Submission**:
   - You must record a short video (2-5 minutes) demonstrating the working application.
   - Upload this video to Brightspace along with your code and report.
4. **Q&A Session**:
   - After the demo, the instructor will conduct a Q&A session with **each team member individually**.
   - You may be asked to explain logic, trace code execution, or implement a small fix on the spot.
5. **Final Grade**: Your individual grade is determined at the conclusion of this interview based on your demonstration and answers.

13. Suggested File Structure

Your project directory should look like this. Ensure your packages match exactly.

```
SchoolRegistrationSystem/
├── data/
│   ├── students.csv
│   ├── courses.csv
│   ├── classrooms.csv
│   └── instructors.csv
├── src/
│   └── com/
│       └── school/
│           ├── app/
```

```
|           |   └── Main.java         // Entry Point (GUI Launcher or Terminal Loop)
|           ├── model/
|           |    ├── Student.java
|           |    ├── Course.java
|           |    ├── Classroom.java
|           |    ├── Instructor.java
|           |   └── ClassSession.java
|           ├── service/
|           |    ├── StudentService.java
|           |    ├── CourseService.java
|           |    ├── ClassroomService.java
|           |    ├── InstructorService.java
|           |   └── RegistrationService.java
|           └── view/                // (Required for GUI Path)
|               ├── SchoolSystemUI.java
|              └── ... (Other panels/forms if needed)
|
```