SCHOOL ENROLLMENT SYSTEM

CS102 Project

Abstract

The SchoolCourseEnrolmentSystem is a Java-based console application designed to simulate a simplified course management system within a university setting. It enables key academic roles — including students, instructors, and administrators — to interact with a shared course database through role-specific features.

Shoug Alomran

2234103192@psu.edu.sa

Table of Contents

Project Structure

Overall Project Structure

---

1. Core Entity Classes

- **`User<T>`** (Generic Abstract Base Class)
  - Fields: ID, password, email, phone, role, address, etc.
  - Subclasses:
    - `Student`
    - `Instructor`
    - `Administrator`
- **`Student`**
  - Handles course enrollment, dropping, grade viewing, credit limit management, personal info update.
- **`Instructor`**
  - Handles viewing students, grading assessments, updating course details, updating personal info.
- **`Administrator`**
  - Handles adding/removing/updating students/instructors, managing courses, assigning instructors, viewing statistics.
- **`Course`**
  - Represents a course with attributes like course name, code, schedule, description, capacity, enrollment status, instructor, list of enrolled students.
- **`Assessment`**
  - Represents exam scores and assessments for students (Quiz, Midterm, Final, Project).

---

2. Utility Classes

- **`Helpers`**
  - Contains reusable methods for:
    - Input validation,
    - Safe input handling,
    - Login workflows,
    - Role assignment,
    - Profile updating,
    - Displaying role-specific menus,
    - Enrolling/Dropping courses,
    - Admin-level management actions.

---

3. Data Management Class

- **`dataManager`**
  - Handles:
    - Persistent storage and loading of:
      - Students (`students.txt`)
      - Instructors (`instructors.txt`)
      - Administrators (`administrators.txt`)
      - Courses (`courses.txt`)
      - Assessments (`assessments.txt`)
    - CRUD Operations:
      - Add/Remove/Update Students, Instructors, Administrators, Courses, Assessments.
    - Utility operations:
      - Assign instructors to courses,
      - Enroll students,
      - Drop students,
      - View enrollment statistics,
      - Generate reports.

---

4. Main Application Class

- **`SchoolCourseEnrolmentSystem`**
  - The main entry point (`main()`).
  - Manages:
    - Initializing default data,
    - User authentication and login,
    - Role-specific menus and session management,
    - Data saving and loading via `dataManager`.

Helpers Class

Purpose

The `Helpers` class provides **utility functions** that support multiple parts of the course enrollment system. It is designed to:

- Handle **safe input collection**,
- **Validate** user information (like password, email, phone),
- **Manage login/logout workflows**,
- **Display menus** for Students, Instructors, and Administrators,
- **Assist with student, instructor, and course operations** (e.g., adding, removing, updating),
- **Generate system statistics and reports**.

It ensures smoother operation across the application by avoiding repeated code and centralizing common tasks.

Structure and Key Components

1. Global Input Scanner

```
public static final Scanner input = new Scanner(System.in);
```

- Single `Scanner` object used throughout the system, avoiding multiple Scanner issues.

---

2. Generic Login System

```
public static <T extends User<T>> T login(List<T> usersList, T tempUser)
```

- Supports **any type of user** (`Student`, `Instructor`, `Administrator`) via Java generics.
- Handles login with **ID first**, then calls `validatePasswordWithRetries()` for password checking.
- Password entry is retried **up to 3 times** before failing.

---

3. Validation Methods

- **`ValidatePassword`**, **`ValidateEmail`**, **`ValidatePhoneNumber`**, **`ValidateAddress`**, **`ValidateName`**:
    - Ensure that user inputs meet basic system standards (e.g., password > 8 characters, email contains '@' and '.', phone is exactly 10 digits).
    - Simple exception handling inside each validator for safety.

---

4. Safe Input Collection

- **`getSafeIntInput(String prompt)`**:
    - Repeatedly prompts for a valid integer input.
    - Catches `InputMismatchException` or parsing errors automatically.
    - Prevents program crashes from bad user inputs.

---

5. Role Management

- **`checkValidityOfRole()`**:
    - Ensures that user roles are properly selected from the allowed options (Student, Instructor, Admin).
- **`updateRolePrompt(currentRole)`**:
    - Allows the user to decide whether to update a role during profile updates.

---

6. Field Updating and Creation

- **`updateFieldWithPrompt`**:
    - Asks the user if they want to update a particular field.
    - Uses a **Predicate** for validation (e.g., `Helpers::ValidateEmail`).

- A **Predicate** is a **built-in Java interface** (from `java.util.function`).
- It represents a **function** that:
- **Takes one input** (for example, a `String`)
- **Returns true or false** (a `boolean`).

- You can pass **any validation rule** (email, phone, password) without writing duplicate code.
- It makes `updateFieldWithPrompt()` and `createUserWithPrompt()` **work for any field** — not just email or password.

Without Predicate, you would have needed to write separate methods for every field, like:

- `updateEmail()`
- `updatePassword()`
- `updatePhoneNumber()`

- **`createUserWithPrompt`**:
  - Prompts the user until they provide a valid value for a new field.

---

7. ID Generation

- **`GenerateRandomID()`**:
  - Creates a 10-digit random number ID for students or instructors.
  - Reduces manual ID entry errors.

- **`StringBuilder`** is a Java class used to build and modify strings faster and more efficiently than regular `String` objects.
- Unlike normal `String`, a `StringBuilder` does not create a new object every time you add or change something.
- Instead, it modifies the same object in memory, which is much faster when doing lots of changes (like adding many characters).

---

8. User Interface Menus

- **`showStudentMenu()`, `showInstructorMenu()`, `showAdminMenu()`**:
  - Print simple, easy-to-read text menus for different user roles.

---

9. Specific Operational Helpers

Helpers provides **pre-built methods** for frequent operations by role:

- **Instructor Helpers**:
  - `updateCourseInfo`
  - `updateInstructorProfile`
- **Student Helpers**:
  - `enroll_In_Course_Student`
  - `dropCourse`
  - `updateStudentProfile`
- **Admin Helpers**:
  - `addStudent`
  - `removeStudent`
  - `removeInstructor`
  - `addInstructor`
  - `addCourse`
  - `adminUpdateStudentProfile`
  - `adminUpdateInstructorProfile`
  - `assignInstructor`
  - `closeCourse`

Each of these methods:

- Collects needed input safely,
- Validates it,
- Calls the relevant core method from the `Administrator`, `Student`, or `Instructor` classes.

---

10. Statistics and Reports

- **`viewEnrollmentStatistics()`**:
  - Summarizes enrollment across all courses: total students enrolled, and whether open courses exist.
- **`generateReports()`**:
  - Identifies the most popular course by enrollment count.
  - Useful for system monitoring and academic analysis.

User Class

Purpose

The `User` class serves as the **abstract base class** for all system users in the **School Course Enrollment System**.
It defines the **common attributes** and **common behaviors** that all types of users (Students, Instructors, Administrators) must share, ensuring consistency and reducing code duplication.

It also **enforces a contract** through **abstract methods** (`login()` and `logout()`) that each subclass must implement individually.

Structure and Key Components

1. Attributes

The `User` class defines the following common attributes:

- **`name`**: The full name of the user.
- **`ID`**: A unique identification number.
- **`password`**: The user's account password.
- **`email`**: The user's email address.
- **`phoneNumber`**: The user's phone number.
- **`role`**: The user's role (`STUDENT`, `INSTRUCTOR`, or `ADMIN`).
- **`address`**: The user's residential address.

These attributes capture the essential profile information required to identify and manage users within the system.

---

2. Enum: Role

- Defines the **allowed types of users** in a strongly-typed way.
- Helps prevent invalid role assignment elsewhere in the system.

---

3. Constructors

- **Default Constructor**: Required for object creation during file reading or empty initialization.
- **Parameterized Constructor**:
  - Initializes all user fields at the moment of object creation.
  - Utilizes **setters** for initialization, ensuring any future validation hooks automatically apply.

---

4. Setters and Getters

- Standard **get** and **set** methods are implemented for all fields.
- Validation logic (such as format checking) is expected to be handled **outside** of setters before calling them (to avoid bloating the class with validation responsibilities).

---

5. Abstract Methods

```
public abstract T login(List<T> list, String id, String password);
public abstract String logout(T user);
```

- **login()**: Forces all subclasses to define how users log into the system, based on their type.
- **logout()**: Forces subclasses to define their logout behavior.
- By keeping these methods abstract, the system **enforces role-specific behavior** at the child class level.

Student Class

Purpose

The `Student` class models a **student user** within the course enrollment system. It extends the generic `User<Student>` class and adds attributes and methods specifically related to a student's academic activities, such as:

- Managing course enrollment,
- Viewing grades,
- Monitoring credit limits,
- Updating personal information.

It simulates a real-world student portal within an academic management system.

Structure and Key Components

1. Inheritance

```
public class Student extends User<Student>
```

- `Student` inherits from a generic `User` class, enabling shared functionality (login, logout) across all user types.
- Defines **role-specific behavior** for students.

---

2. Attributes

- `creditLimit`: The maximum number of credit hours the student can enroll in.
- `enrolledCourses`: A list of `Course` objects that the student is currently enrolled in.

The `enrolledCourses` list is initialized safely inside the constructor to avoid `NullPointerException`.

---

3. Constructors

- **Default Constructor**: Allows for an empty `Student` object if needed.
- **Parameterized Constructor**:

- Calls the parent `User` constructor to set common fields.
- Sets the student's `creditLimit`.
- Initializes the list of enrolled courses (new list if none provided).

---

4. Methods

Authentication

- **`login`**:
  - Searches the list of students to authenticate based on ID and password.
  - Prints appropriate login success/failure messages.
- **`logout`**:
  - Returns a string confirming successful logout.

Enrollment Management

- **`totalCreditLimit`**:
  - Calculates the total number of credit hours from all currently enrolled courses.
- **`enroll_In_Course`**:
  - Enrolls the student into a course, but **only if**:
    - The student is not already enrolled,
    - The course is not full,
    - Enrollment is open,
    - Credit limits allow it.
  - Automatically updates both:
    - Student's `enrolledCourses` list,
    - Course's `enrolledStudents` list.
- **`dropCourse`**:
  - Allows the student to drop a course, **only if** enrollment is still open.
  - Ensures students cannot drop closed courses.

Credit Management

- **`viewCreditLimit`**:
  - Displays the student's total allowed credit limit and how much credit is still available.

    - Properly handles null course lists.
- **`enrolledInHours`**:
    - Calculates total enrolled hours by iterating through current enrolled courses.

Grades Management

- **`viewGrades`**:
    - Provides a menu-driven interface for students to:
        - View specific exam scores,
        - View all grades in a course,
        - View average grades in a course.
    - Uses a temporary `Assessment` object to access the `Assessment` class's grade-viewing methods.

Course Exploration

- **`viewEnrolledCourses`**:
    - Lists all courses a student is currently enrolled in, displaying course names, codes, credit hours, and schedules.
- **`viewAvailableCourses`**:
    - Lists all available (not full and open) courses for the student to consider enrolling in.

Personal Information Management

- **`updateStudentPersonalInfo`**:
    - Allows a student to update their password, email, phone number, and address.

Utility

- **`toString`**:
    - Overrides `toString()` to print the student's ID and name neatly — useful for administrative or system-wide listings.

Administrator Class

Purpose

The `Administrator` class models a **system administrator** within the course enrollment system. It provides methods for administrators to **manage students, instructors, and courses**, as well as to **authenticate themselves**.

It plays a critical role in ensuring that the system remains organized by enabling admin users to:

- Add, remove, and update users,
- Add and manage courses,
- Assign instructors to courses,
- Open or close courses for enrollment.

Structure and Key Components

1. Inheritance

```
public class Administrator extends User<Administrator>
```

- `Administrator` extends a generic `User` class, leveraging a flexible architecture.
- The use of generics (`User<Administrator>`) suggests a system design that handles multiple user types uniformly (Student, Instructor, Admin).

---

2. Constructors

- **Default Constructor**: Required to allow instantiating an empty Administrator object (useful for serialization, frameworks, or manual field-setting).
- **Parameterized Constructor**: Initializes the administrator's personal information and passes it to the superclass `User`, setting the role specifically to `ADMIN`.

---

3. Methods

Authentication

- **`login`**:
    - Searches a list of administrators for matching ID and password.
    - Prints feedback about success or failure.
    - Returns the matched `Administrator` object or `null`.
    - **Minor Issue**: "No admin record" message prints inside the loop instead of after the loop ends. (This could lead to repeated wrong messages during search.)
- **`logout`**:
    - Returns a simple string confirming logout.

User Management

- **`removeStudent` and `removeInstructor`**:
    - Use a `Iterator` to safely traverse and remove elements while avoiding `ConcurrentModificationException`.
    - Print messages indicating success or failure.
- **`addInstructor` and `addStudent`**:
    - Add new instructors or students to their respective lists.
    - Confirm addition via console messages.
- **`updateStudent` and `updateInstructor`**:
    - Locate the student or instructor by ID.
    - If found, update their details (name, email, password, phone, role, address).
    - If not found, display an appropriate message.

Course Management

- **`addCourse`**:
    - Adds a new course to the list.
    - Prints confirmation of success or failure.
- **`assignInstructor`**:
    - Assigns an instructor to a course **only if** no instructor is assigned already.
    - Prevents overwriting an existing instructor unless explicitly removed.

- **`closeCourse`**:
    - Closes enrollment for a course if:
        - The course is found,
        - The course is at full capacity,
        - Or if it is already closed.
    - Provides feedback accordingly.

Viewing Records

- **`viewStudentList` and `viewInstructorList`**:
    - Display a list of all students or instructors using their `toString()` method.
    - Handle the case when the list is empty or null.

Instructor Class

Purpose

The `Instructor` class models a **instructor** within the school course enrollment system. It extends a generic `User<Instructor>` class, meaning it inherits basic user-related attributes (like ID, password, etc.) and specializes them with instructor-specific functionalities such as:

- Logging in and logging out,
- Managing students,
- Grading students,
- Updating course information,
- Updating personal information.

It closely reflects the real-world responsibilities of an instructor in an academic system.

Structure and Key Components

1. Inheritance

```
public class Instructor extends User<Instructor>
```

- Inherits from a generic `User` class.
- By specifying `User<Instructor>`, the class follows a generic design pattern, likely making it flexible for handling different types of users (e.g., Admin, Student, Instructor).

---

2. Constructors

- **Default Constructor**: Needed to create an empty `Instructor` object.
- **Parameterized Constructor**: Accepts full personal and professional details, passing most fields to the superclass (`User`) via `super()`.
    - Note: Although `enrolledStudents` is passed into the constructor, it is currently unused inside the body (which might be intentional or could be an oversight).

---

3. Methods

`login(List<Instructor> listOfInstructors, String id, String password)`

- **Overrides** the generic `User.login()` method.
- Authenticates an instructor by comparing input credentials to stored instructors.
- Provides console feedback on successful or failed login.

`logout(Instructor instructor)`

- **Overrides** the generic `User.logout`() method.
- Returns a simple message indicating the instructor has logged out.

`viewEnrolledStudents(Instructor  instructor,List<Course> listOfCourses)`

- Allows an instructor to:
  - See a list of students enrolled in their assigned courses.
  - Ensures that only the instructor's courses are shown.
  - Handles the cases where no students are enrolled or where the instructor teaches no courses.

`gradeStudent(List<Assessment> listOfGrades, List<Course> listOfCourses)`

- Provides a **multi-step, user-driven process** to assign or update a student's grade:
  1. Instructor chooses a course they teach.
  2. Instructor chooses a student enrolled in that course.
  3. Instructor selects an assessment type (Quiz, Midterm, Final, etc.).
  4. Instructor inputs a score (validated to be between 0–100).
  5. Score is either updated if it already exists, or a new assessment is created.
- Handles invalid inputs gracefully, with meaningful error messages.

`updateCourseInfo(String courseCode, List<Course> listOfCourses, String newSchedule, String newDescription)`

- Lets an instructor:
  - Update the **schedule** and **description** of a course **only** if they are assigned to it.
- Handles cases when:

    - Course is not found,
    - Another instructor is assigned,
    - No instructor is assigned.

`updateInstructorPersonalInfo(String newPassword, String newEmail, String newPhoneNumber, String newAddress)`

- Allows instructors to update their **own account information** (password, email, phone, address).
- Assumes the instructor is already authenticated, so no additional checks are needed.

`toString()`

- Overrides `toString()` to provide a readable summary of the instructor:
    - Displays ID and Name neatly.
- Useful for debugging, displaying instructors in lists, etc.

Course Class

Purpose:

The Course class models the concept of a **university course** within a course enrollment system. It captures all essential course attributes — including the course name, code, schedule, description, instructor, enrollment status, capacity, assessments, and students enrolled.

It is designed to:

- **Store** detailed information about a course,
- **Allow** checking if the course is full,
- **Manage** students and assessments associated with it.

Structure and Key Components

1. Enumerations

- The `EnrollmentStatusEnum` defines whether a course is `Open` (available for enrollment) or `Closed`.
- Using an `enum` improves type safety, ensures only valid enrollment statuses can be assigned, and improves code readability.

---

2. Attributes (Private Fields)

- **Basic Information**: `courseName`, `courseCode`, `schedule`, `description`.
- **Enrollment Management**:
  - `enrollmentStatus`: current status of the course (open/closed).
  - `capacity`: maximum number of students allowed.
  - `enrolledStudents`: a list that tracks all students currently enrolled.
- **Academic Details**:
  - `grades`: a list of assessments (like quizzes, midterms) associated with the course.
  - `creditHours`: number of credit hours for the course.
- **Instructor**:
  - `instructor`: an object representing the professor or teacher assigned to the course.

Note: The `enrolledStudents` list is initialized during course creation to avoid `NullPointerException` issues later.

---

3. Constructors

- **Full Constructor**: Initializes all attributes and ensures a course object is created fully populated.
- **Default Constructor**: Provides flexibility if an empty `Course` needs to be instantiated and later populated.

---

4. Setters and Getters

- Each field has corresponding getter and setter methods to control access and modification.
- **Validation Logic** is included:
    - `courseName`, `courseCode`, and `capacity` are validated against empty/null or invalid values.
    - `creditHours` must be greater than 0.
    - If invalid values are detected, the setters print an error message and avoid updating the field.
- The class is defensive and robust against bad inputs.

---

5. Methods

`isFull()`

- Checks if the number of enrolled students has reached or exceeded the course's maximum `capacity`.
- Prevents over-enrollment by returning a boolean.

Assessment Class

Purpose:

The Assessment class models **individual evaluation events** (like quizzes, midterms, finals, and projects) within a course enrollment system. It connects a **student**, a **course**, and a **specific assessment type** with the student's **score** for that assessment.

It also provides **operations** to manage grades:

- Assigning and updating grades,
- Viewing all grades for a course,
- Viewing averages,
- Viewing specific exam scores.

Structure and Key Components

1. Enumerations

- This enum defines **standardized types of assessments**.
- It **restricts** assessment types to valid options, improving consistency and avoiding user errors (e.g., typos).

---

2. Attributes (Private Fields)

- `studentId`: ID of the student who took the assessment.
- `courseCode`: The course to which the assessment belongs.
- `examType`: The type of exam (e.g., `Quiz_1`, `Midterm_2`).
- `score`: The numerical score earned by the student.
- `assessmentName`: A descriptive name for the assessment (although often it mirrors `examType`).

This tightly links **student**, **course**, and **performance data**.

---

3. Constructor

- One main constructor that initializes all necessary fields.
- This enforces that no `Assessment` object is incomplete when created.

---

4. Setters and Getters

- Standard getters and setters for all attributes.
- No complex validation inside the setters — straightforward setting of values.

---

5. Methods

`assignGrade`

- Takes a list of assessments and:
    - **Updates** an existing score if the assessment already exists for the student-course-exam combination.
    - **Adds** a new assessment entry if no matching one is found.
- Provides console feedback indicating if the grade was updated or newly recorded.

`viewAllGradesForCourse`

- Allows a student to view **all grades** for a specific course they are enrolled in.
- It dynamically lists available courses, prompts the student to select, and displays their scores.
- If no grades are found, it gracefully informs the user.

`viewTotalAverageGrade`

- Calculates the **average score** across all assessments for a student in a particular course.
- Provides meaningful feedback even when no grades are available.

`viewSpecificExamScore`

- Lets a student **select a course** and then **select a specific exam type** to see their individual score.
- Handles invalid selections (both for courses and exam types) gracefully.
- If a score is not found for the selected exam, it clearly communicates that to the user.

Main

Purpose

The `SchoolCourseEnrolmentSystem` class is the **main entry point** for the course enrollment system application.
It provides the **primary control flow** for:

- Authenticating users (Students, Instructors, Administrators),
- Navigating role-specific menus,
- Managing user actions (e.g., enrolling in courses, grading students, managing users and courses),
- Persisting data across sessions.

It acts as the **central coordinator** between the user interface (text menus), system logic (`Student`, `Instructor`, `Administrator` classes), and data management (`dataManager`).

Structure and Key Components

---

1. Administrator Default Instance

```
private static Administrator DefaultAdmin = new Administrator(...);
```

- A **default administrator** (“Shoug”) is created at startup to ensure that there is at least one admin available in the system.
- This ensures that administrative tasks (like adding students or instructors) are always possible, even on a fresh start.

---

2. Main Method Flow

```
public static void main(String[] args)
```

- **Initialize** the singleton `dataManager` instance.
- **Load** all previously saved data using `dataManagerInstance.loadAllData()`.
- **Add** the default administrator (`DefaultAdmin`) to the system (even if loaded data exists).

---

3. Role Selection Loop

- Welcomes the user and repeatedly prompts:

  `"Enter your role from the current options: (Student/Instructor/Admin) or exit to finish."`

- Based on the entered role, it directs the user to **role-specific workflows**:
  - **Student**
  - **Instructor**
  - **Administrator**

Each role has its own login, menu, actions, and logout handling.

Student Workflow

- **Login**: Using `Helpers.login().`
- **Menu Options**:
  1. View available courses
  2. Enroll in a course
  3. View enrolled courses
  4. View credit limit
  5. Drop a course
  6. View grades
  7. Update personal profile
  8. Logout
- **After certain actions** (like updating profiles), the student's data is saved immediately via `dataManagerInstance.saveStudents().`

Instructor Workflow

- **Login**: Using `Helpers.login().`
- **Menu Options**:
  1. View enrolled students in assigned courses
  2. Grade students
  3. Update course information

4. Update personal profile
5. Logout

- **Saves changes** immediately after critical updates, such as course or profile updates.

Administrator Workflow

- **Login**: Using `Helpers.login()`.
- **Menu Options**:
  1. Add students
  2. Remove students
  3. View student list
  4. Add instructors
  5. Remove instructors
  6. View instructor list
  7. Add course
  8. Close course
  9. Update student profiles
  10. Update instructor profiles
  11. Assign instructors to courses
  12. View enrollment statistics
  13. Generate reports
  14. Logout
- **Saves updates** to students and instructors whenever changes are made.

Exit Flow

- If the user types "exit", the system:
  - Saves all data via `dataManagerInstance.saveAllData()`.
  - Displays a goodbye message.
  - Safely terminates the application.

Exception Handling

- The entire main loop is wrapped in a `try-catch` block.
- Any unexpected error during the program's execution will not crash the system — instead, a generic error message is printed.

Code:

Main (Driver Class)

```
package schoolcourseenrolmentsystem;

import java.util.*;

public class SchoolCourseEnrolmentSystem {
  private static Administrator DefaultAdmin = new Administrator(“Shoug”, “1122334455”, “Default12345”,
      “S.Alomran@gmail.com”,
      “0531110904”, “Qirawan district”);

  public static void main(String[] args) {
    dataManager dataManagerInstance = dataManager.getInstance();
    Scanner input = Helpers.input;

    dataManagerInstance.loadAllData();

    try {
      boolean exit = false;
      dataManagerInstance.addAdministrator(DefaultAdmin);
      System.out.println(“\nWelcome to School Course Enrolment System!”);
      while (!exit) {
        System.out.println(
            “Enter your role from the current options: (Student/Instructor/Admin) or exit to finish.”);
        String role = input.next();
        input.nextLine();

        switch (role.toLowerCase()) {
          case “student”:
            Student tempStudent = new Student();
            Student loggedInStudent = Helpers.login(dataManagerInstance.getAllStudents(), tempStudent);

            if (loggedInStudent == null) {
              continue;
            }
            boolean studentExit = false;
```

```
while (!studentExit) {
    Helpers.showStudentMenu();
    int studentChoice = Helpers.getSafeIntInput(“Option: “);

    switch (studentChoice) {
        case 1:
            loggedInStudent.viewAvailableCourses(dataManagerInstance.getAllCourses());
            break;
        case 2:
            Helpers.enroll_In_Course_Student(dataManagerInstance.getAllCourses(),
                    loggedInStudent);
            break;
        case 3:
            loggedInStudent.viewEnrolledCourses();
            break;
        case 4:
            loggedInStudent.viewCreditLimit(dataManagerInstance.getAllStudents());
            break;
        case 5:
            Helpers.dropCourse(dataManagerInstance.getAllCourses(), loggedInStudent);
            break;
        case 6:
            loggedInStudent.viewGrades(dataManagerInstance.getAllAssessments());
            break;
        case 7:
            Helpers.updateStudentProfile(loggedInStudent);
            dataManagerInstance.saveStudents(); // Save after profile update
            break;
        case 8:
            loggedInStudent.logout(loggedInStudent);
            studentExit = true;
            break;
        default:
            System.out.println(“Invalid option.”);
    }
}
break;
```

```
        case “instructor”:
            Instructor tempInstructor = new Instructor();
            Instructor loggedInInstructor = Helpers.login(dataManagerInstance.getAllInstructors(),
                    tempInstructor);

            if (loggedInInstructor == null) {
                continue;
            }
            boolean exitInstructor = false;
            while (!exitInstructor) {
                Helpers.showInstructorMenu();
               int instructorChoice = Helpers.getSafeIntInput(“Option: “);

                switch (instructorChoice) {
                    case 1:
                        loggedInInstructor.viewEnrolledStudents(loggedInInstructor,
                                dataManagerInstance.getAllCourses());
                        break;
                    case 2:
                        loggedInInstructor.gradeStudent(dataManagerInstance.getAllAssessments(),
                                dataManagerInstance.getAllCourses());
                        break;
                    case 3:
                        Helpers.updateCourseInfo(loggedInInstructor, dataManagerInstance.getAllCourses());
            dataManagerInstance.saveCourses(); // Save after course update
                        break;
                    case 4:
                        Helpers.updateInstructorProfile(loggedInInstructor);
       dataManagerInstance.saveInstructors(); // Save after profile update
                        break;
                    case 5:
                        loggedInInstructor.logout(loggedInInstructor);
                        exitInstructor = true;
                        break;
                    default:
                        System.out.println(“Invalid option.”);
                }
```

```
                        }
                        break;
                    case “admin”:
                        Administrator temp = new Administrator();
                        Administrator loggedInAdministrator = Helpers.login(dataManagerInstance.getAllAdministrators(),
temp);

                        if (loggedInAdministrator == null) {
                            continue;
                        }
                        boolean exitAdmin = false;
                        while (!exitAdmin) {
                            Helpers.showAdminMenu();
                            int adminChoice = Helpers.getSafeIntInput(“Option: “);

                            switch (adminChoice) {
                                case 1:
Helpers.addStudent(loggedInAdministrator, dataManagerInstance.getAllStudents());
                                    break;
                                case 2:
Helpers.removeStudent(loggedInAdministrator, dataManagerInstance.getAllStudents());
                                    break;
                                case 3:
         loggedInAdministrator.viewStudentList(dataManagerInstance.getAllStudents());
                                    break;
                                case 4:
Helpers.addInstructor(loggedInAdministrator,dataManagerInstance.getAllInstructors());
                                    break;
                                case 5:
Helpers.removeInstructor(loggedInAdministrator, dataManagerInstance.getAllInstructors());
                                    break;
                                case 6:
loggedInAdministrator.viewInstructorList(dataManagerInstance.getAllInstructors());
                                    break;
                                case 7:
Helpers.addCourse(loggedInAdministrator, dataManagerInstance.getAllCourses(),
dataManagerInstance.getAllInstructors());
```

```
                                break;
                            case 8:
Helpers.closeCourse(loggedInAdministrator, dataManagerInstance.getAllCourses());
                                break;
                            case 9:
Helpers.adminUpdateStudentProfile(loggedInAdministrator, dataManagerInstance.getAllStudents());
                      dataManagerInstance.saveStudents(); // Save after profile update
                                break;
                            case 10:
Helpers.adminUpdateInstructorProfile(loggedInAdministrator,dataManagerInstance.getAllInstructors());
                   dataManagerInstance.saveInstructors(); // Save after profile update
                                break;
                            case 11:
Helpers.assignInstructor(loggedInAdministrator, dataManagerInstance.getAllCourses(),
dataManagerInstance.getAllInstructors());
                                break;
                            case 12:
                                dataManagerInstance.viewEnrollmentStatistics();
                                break;
                            case 13:
                                dataManagerInstance.generateReports();
                                break;
                            case 14:
                              loggedInAdministrator.logout(loggedInAdministrator);
                                exitAdmin = true;
                                break;
                            default:
                                System.out.println("Invalid option.");
                        }
                    }
                    break;
                case "exit":
                    exit = true;
                dataManagerInstance.saveAllData(); // Save all data before exiting
                    System.out.println("Goodbye.");
                    break;
                default:
                    System.out.println(“Invalid role. Try again.”);
                }
            }
        } catch (Exception e) {
            System.out.println(“Error: “ + e.getMessage());
        }
    }
}
```

User Class

```
package schoolcourseenrolmentsystem;

import java.util.List;

public abstract class User<T> {
    // Attributes
    // enum is short for enumeration
    public enum Role {
        STUDENT,
        INSTRUCTOR,
        ADMIN;
    }

    private String name, ID, password, email, phoneNumber;
    private Role role;
    private String address;

    public User() { }

    // Constructor
    public User(String name, String id, String password, String email, String phoneNumber, Role role, String address) {
        setName(name);
        setId(id);
        setPassword(password);
        setEmail(email);
        setPhoneNumber(phoneNumber);
        setRole(role);
        setAddress(address);
    }
    //======================= Setters and Getters =======================
    // we do validation in prior to calling this method to avoid bloating it
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getId() {
        return ID;
    }

    public void setId(String ID) {
        this.ID = ID;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
```

```
    public void setPhoneNumber(String phoneNumber) {
        // research on how to ensure that phone numbers follow the KSA pattern
        this.phoneNumber = phoneNumber;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }



    public Role getRole() {
        return role;
    }

    public void setRole(Role role) {
        this.role = role;
    }
//====================== Methods ======================
    // Abstract Methods
    public abstract T login(List<T> list, String id, String password);

    public abstract String logout(T user);
}
```

Student Class

```
package schoolcourseenrolmentsystem;

import java.util.*;

public class Student extends User<Student> {
    // Attributes
    private int creditLimit;
    private List<Course> enrolledCourses;
    public Student() { }
    // Constructor
    public Student(String name, String id, String password, String email, String phoneNumber, String address, int
creditLimit, List<Course> enrolledCourses) {
        super(name, id, password, email, phoneNumber, Role.STUDENT, address);
        setCreditLimit(creditLimit);
        // If the list is null, create a new one. Fixes NullPointerException.
        if (enrolledCourses != null) {
            this.enrolledCourses = enrolledCourses;
        } else {
            this.enrolledCourses = new ArrayList<>();
        }
    }
    //====================== Setters and Getters ======================
    public List<Course> getEnrolledCoursesList() {
        return enrolledCourses;
    }

    public int getCreditLimit() {
        return creditLimit;
    }

    public void setCreditLimit(int creditLimit) {
        if (creditLimit < 0) {
            System.out.println("Must be greater than 0");
            return;
        }
        this.creditLimit = creditLimit;
    }

    //====================== Methods ======================

    @Override
    public Student login(List<Student> listOfStudents, String id, String password) {
        for (Student student : listOfStudents) {
            if (student.getId().equals(id) && student.getPassword().equals(password)) {
                System.out.println("Student " + student.getName() + " logged in.");
            return student;
          }
        }
        System.out.println(“No student record was found with the ID and password provided.”);
        return null;
    }

    @Override
    public String logout(Student student) {
        return (“Student “ + student.getName() + “ logged out.”);
    }

    // Count the hours of the courses already enrolled
    public int totalCreditLimit() {
        int total = 0;
        // This loop is to count how many courses and their toal hours
        for (Course course : enrolledCourses) {
          total += course.getCreditHours();
        }
        return total;
    }

    public void enroll_In_Course(Course course) {
        // 1. make sure credit limit is not excited and that the capacity is avaliable
        if (enrolledCourses.contains(course)) {
          System.out.printf(“\nYou are already enrolled in (%s) (%s).”, course.getCourseName(),
course.getCourseCode());
        } else if (course.isFull()) {
          System.out.println(“\nCannot enroll in “ + course.getCourseName() + “ — course is full.”);
        } // Check if the enrollment status is open or not
        else if (course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Closed) {
          System.out.println(“\nEnrollment status is closed.”);
        } // Previously enrolled hours + new hours
        else if (totalCreditLimit() + course.getCreditHours() > creditLimit) {
          System.out.println(“\nCannot enroll in “ + course.getCourseName() + “ — credit limit exceeded.”);
        } else {
          /// Adding the current student to the course's list of enrolled students.
```

```
            course.getEnrolledStudents().add(this);
            /// Adding the course to the student's list of enrolled courses.
            this.enrolledCourses.add(course);
            System.out.println("\n" + getName() + " enrolled successfully");
        }
    }

    public void dropCourse(Course course) {
        if (!enrolledCourses.contains(course)) {
            System.out.printf("%s is not enrolled in (%s) (%s).", getName(), course.getCourseName(),
course.getCourseCode());
        } else if (enrolledCourses.contains(course)
                && course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Closed) {
            System.out.println("Drop period is closed.");
        } else if (enrolledCourses.contains(course)
                && course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Open) {
            this.enrolledCourses.remove(course);
            System.out.printf("%s has successfully dropped (%s) (%s).", getName(), course.getCourseName(),
course.getCourseCode());
        }
    }

    public void viewCreditLimit(List<Student> listOfStudents) {
        for (Student student : listOfStudents) {
            System.out.println("Your credit limit is: " + student.getCreditLimit());

            // (condition) ? (value if true) : (value if false);
            int enrolledHours = (student.getEnrolledCoursesList() != null) ? student.enrolledInHours() : 0;
            System.out.println("Remaining credit limit is: " + (student.getCreditLimit() - enrolledHours) + " hours.");
            break;
        }
    }

    public int enrolledInHours() {
        int total = 0;
        if (enrolledCourses != null) {
            for (Course course : enrolledCourses) {
```

```
            total += course.getCreditHours();
         }
      }
      return total;
   }

   public void viewEnrolledCourses() {
      if (enrolledCourses.isEmpty()) {
         System.out.println(getName() + " is not enrolled in any courses.");
      } else {
         System.out.println(getName() + " is enrolled in: ");
         int index = 1;
         for (Course course : enrolledCourses) {
            System.out.printf(index + "- (%s) (%s) (%s). The schedule is: %s.", course.getCourseName(),
course.getCourseCode(), course.getCreditHours(), course.getSchedule());
            index++;
         }
      }
   }

   public void viewGrades(List<Assessment> listOfGrades) {
      System.out.println("\nWhat would you like to view?\n1. View a specific exam score.\n2. View all grades in a
course.\n3. View total average in a course.");

      int choice = Helpers.getSafeIntInput("\nOption: ");
      Assessment temp = new Assessment(null, null, null, 0.0, null);

      if (choice == 1) {
         temp.viewSpecificExamScore(listOfGrades, this);
      } else if (choice == 2) {
         temp.viewAllGradesForCourse(listOfGrades, this);
      } else if (choice == 3) {
         List<Course> courses = this.getEnrolledCoursesList();

         if (courses.isEmpty()) {
            System.out.println("\nYou are not enrolled in any courses.");
            return;
         }
         System.out.println("\nSelect a course:");
         for (int i = 0; i < courses.size(); i++) {
            System.out.printf((i + 1) + ".%s  ( %s )", courses.get(i).getCourseName(), courses.get(i).getCourseCode());
         }
         int courseChoice = Helpers.getSafeIntInput("\nOption: ");
         if (courseChoice < 1 || courseChoice > courses.size()) {
            System.out.println("Invalid course selection.");
            return;
         }
         String courseCode = courses.get(courseChoice - 1).getCourseCode();
         temp.viewTotalAverageGrade(listOfGrades, this.getId(), courseCode);
      } else {
         System.out.println("Invalid choice.");
      }
   }

   public void viewAvailableCourses(List<Course> listOfCourses) {
      boolean avaliableCourses = false;
      System.out.println("\nAvailable courses are: ");
      for (Course course : listOfCourses) {
         if (course.isFull() == false && course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Open) {
            avaliableCourses = true;
            System.out.println("- (Course name: " + course.getCourseName() + ") (Course code: " +
course.getCourseCode() + ") (Course schedule: " + course.getSchedule() + ") (Course instructor: " +
course.getInstructor()+ ") (Course capacity: " + course.getCapacity() + ").\n");
         }
      }
      if (!avaliableCourses) {
         System.out.println("No courses are avaliable.");
      }
   }

   public void updateStudentPersonalInfo(String newPassword, String newEmail, String newPhoneNumber,
         String newAddress) {
      setPassword(newPassword);
      setEmail(newEmail);
      setPhoneNumber(newPhoneNumber);
      setAddress(newAddress);
      System.out.println("\nStudent " + getName() + "'s information updated.");
   }

   @Override
   public String toString() {
      return String.format("-ID: %s Name: %s", this.getId(), this.getName());
   }
}
```

Instructor Class

```
package schoolcourseenrolmentsystem;

import java.util.*;

public class Instructor extends User<Instructor> {
    // Constructor
    public Instructor() {
    }

    public Instructor(String name, String id, String password, String email, String phoneNumber,
            String address, List<Student> enrolledStudents) {
        super(name, id, password, email, phoneNumber, Role.INSTRUCTOR, address);
    }

    // ======================= Methods =======================
    @Override
    public Instructor login(List<Instructor> listOfInstructors, String id, String password) {
        for (Instructor instructor : listOfInstructors) {
            if (instructor.getId().equals(id) && instructor.getPassword().equals(password)) {
           System.out.println("Instructor " + instructor.getName() + " logged in.\n");
                return instructor;
            }
        }
        System.out.println("No instructor record was found with the ID and password provided.");
        return null;
    }
    @Override
    public String logout(Instructor instructor) {
        return ("Instructor " + instructor.getName() + " logged out.");
    }

    public void viewEnrolledStudents(Instructor instructor, List<Course> listOfCourses) {
        boolean enrolledCourse = false;
        // Look through all the courses
        for (Course course : listOfCourses) {
            // Make sure that the instructor is actually assigned to the course
            if (course.getInstructor() != null && course.getInstructor().getId().equals(instructor.getId())) {
                enrolledCourse = true;
                // The instructor might teach multiple courses
                System.out.printf("\n(Course: %s) (Code: %s)", course.getCourseName(), course.getCourseCode());
                course.getEnrolledStudents();

                // Make sure there are actually students enrolled in the course.
                // Update the list.
                List<Student> courseStudents = course.getEnrolledStudents();
                if (courseStudents == null || courseStudents.isEmpty()) {
                    System.out.println("No students enrolled in this course.");
                } else {
                    for (Student student : courseStudents) {
                        System.out.printf("\n-Student: %s, ID: %s, Email: %s, Phone number: %s", student.getName(),
student.getId(), student.getEmail(), student.getPhoneNumber());
                    }
                }
            }
        }
        if (!enrolledCourse) {
            System.out.println("Instructor " + instructor.getName() + " is not assigned to any courses.");
        }
    }

    public void gradeStudent(List<Assessment> listOfGrades, List<Course> listOfCourses) {
        Scanner input = new Scanner(System.in);
        try {
            // Step 1: Show instructor's courses
            List<Course> instructorCourses = new ArrayList<>();
            System.out.println("\nCourses you're assigned to:");
            for (Course course : listOfCourses) {
                if (course.getInstructor() != null && course.getInstructor().getId().equals(this.getId())) {
                    instructorCourses.add(course);
                    System.out.println((instructorCourses.size()) + ". " + course.getCourseName() + " (" +
course.getCourseCode() + ")");
                }
            }
            if (instructorCourses.isEmpty()) {
                System.out.println("You're not assigned to any courses.");
                return;
            }
            int courseChoice = Helpers.getSafeIntInput("Select a course to grade: ");
            if (courseChoice < 1 || courseChoice > instructorCourses.size()) {
                System.out.println("Invalid selection.");
                return;
            }
            Course selectedCourse = instructorCourses.get(courseChoice - 1);

            // Step 2: Select student
            List<Student> enrolled = selectedCourse.getEnrolledStudents();
            if (enrolled.isEmpty()) {
                System.out.println("No students enrolled in this course.");
                return;
            }
            System.out.println("\nStudents in this course:");
            for (int i = 0; i < enrolled.size(); i++) {
                System.out.printf((i + 1) + ".%s  (ID: %s)", enrolled.get(i).getName(), enrolled.get(i).getId());
            }

           int studentChoice = Helpers.getSafeIntInput("\nSelect a student to grade: ");
            if (studentChoice < 1 || studentChoice > enrolled.size()) {
                System.out.println("Invalid student selection.");
                return;
```

```
}
Student selectedStudent = enrolled.get(studentChoice - 1);

// Step 3: Choose exam type
Assessment.ExamType[] examTypes = Assessment.ExamType.values();
System.out.println("\nSelect the assessment type to grade:");
for (int i = 0; i < examTypes.length; i++) {
  System.out.println((i + 1) + ". " + examTypes[i]);
}
int examChoice = Helpers.getSafeIntInput("Choice: ");
if (examChoice < 1 || examChoice > examTypes.length) {
  System.out.println("Invalid selection.");
  return;
}
Assessment.ExamType selectedExamType = examTypes[examChoice - 1];

// Step 4: Enter score
double score;
while (true) {
  System.out.print("Enter score (0–100): ");
  try {
    score = input.nextDouble();
    input.nextLine(); // Buffer
    if (score < 0 || score > 100) {
      System.out.println("Score must be between 0 and 100.");
    } else {
      break;
    }
  } catch (InputMismatchException e) {
    System.out.println("Invalid input. Please enter a number.");
    input.nextLine(); // Clear input
  }
}

// Step 5: Assign or update grade
boolean updated = false;
for (Assessment g : listOfGrades) {
```

```
            if (g.getStudentId().equals(selectedStudent.getId())
                    && g.getCourseCode().equals(selectedCourse.getCourseCode())
                    && g.getExamType().equals(selectedExamType)) {
                g.setScore(score);
                System.out.println("Grade updated successfully.");
                updated = true;
                break;
            }
        }
        if (!updated) {
            Assessment newAssessment = new Assessment(selectedStudent.getId(),
selectedCourse.getCourseCode(), selectedExamType, score, selectedExamType.toString());
            listOfGrades.add(newAssessment);
            System.out.println("Grade recorded successfully.");
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public void updateCourseInfo(String courseCode, List<Course> listOfCourses, String newSchedule, String
newDescription) {

    // Now find the actual Course object
    Course courseToUpdate = null;
    for (Course course : listOfCourses) {
        if (course.getCourseCode().equals(courseCode)) {
            courseToUpdate = course;
            break;
        }
    }
    if (courseToUpdate == null) {
        System.out.println("Course with code " + courseCode + " not found.");
        return;
    }
    if (courseToUpdate != null) {
        Instructor assignedInstructor = courseToUpdate.getInstructor();
```

```
            if (assignedInstructor != null && assignedInstructor.getId().equals(this.getId())) {
                courseToUpdate.setSchedule(newSchedule);
                courseToUpdate.setDescription(newDescription);

                System.out.printf("Instructor %s's course information updated for %s", assignedInstructor.getName(),
courseToUpdate.getCourseName());
            } else if (assignedInstructor != null) {
                System.out.printf("%s with the ID number: %s is not assigned to %s.", assignedInstructor.getName(),
assignedInstructor.getId(), courseToUpdate.getCourseName());
            }
            // You don't see it but its basically (assignedInstructor == null);
            else {
                System.out.printf("No instructor assigned to %s.", courseToUpdate.getCourseName());
            }
        }
    }

    public void updateInstructorPersonalInfo(String newPassword, String newEmail, String newPhoneNumber, String
newAddress) {
        // No need to check if they match because instructors are already logged in.
        setPassword(newPassword);
        setEmail(newEmail);
        setPhoneNumber(newPhoneNumber);
        setAddress(newAddress);
        System.out.println("Instructor " + getName() + "'s information updated.");
    }

    @Override
    public String toString() {
        return String.format("-ID: %s Name: %s", this.getId(), this.getName());
    }
}
```

Administrator Class

```
package schoolcourseenrolmentsystem;
```

```
import java.util.*;
import schoolcourseenrolmentsystem.Course.EnrollmentStatusEnum;

public class Administrator extends User<Administrator> {
   // Constructors
   // We must define the default constructor in case we define any other custom
   // constructors
   public Administrator() {
   }

   public Administrator(String name, String id, String password, String email, String phoneNumber, String address) {
      super(name, id, password, email, phoneNumber, User.Role.ADMIN, address);
   }

   // Methods

   // Login/logout & role
   @Override
   public Administrator login(List<Administrator> listOfAdministrators, String id, String password) {
      Administrator returnAdmin = null;
      for (Administrator administrator : listOfAdministrators) {
         if (administrator.getId().equals(id) && administrator.getPassword().equals(password)) {
     System.out.println("Administrator " + administrator.getName() + " logged in.");
            returnAdmin = administrator;
            break;
         }
        System.out.println("No admin record was found with the ID and password provided.");
      }
      return returnAdmin;
   }

   @Override
   public String logout(Administrator administrators) {
      return ("Administrator " + administrators.getName() + " logged out.");
   }

   // Add and remove users
```

```
    /// Loop used to iterate over all the students/instructors.
    /// An Iterator keeps track of the current position safely, and when you call
    /// remove(), it tells the collection to safely remove the current element
    /// without invalidating the loop.
    public boolean removeStudent(List<Student> listOfStudents, String targetId) {
        Iterator<Student> iterator = listOfStudents.iterator();
        while (iterator.hasNext()) {
            Student specificStudent = iterator.next();
            if (specificStudent.getId().equals(targetId)) {
                iterator.remove();
                System.out.println(specificStudent.getName() + " with the ID: " + specificStudent.getId() + " has been
successfully removed");
                return true;
            }
        }
        System.out.println(targetId + " is not found.");
        return false;
    }

    public boolean removeInstructor(List<Instructor> listOfInstructors, String targetId) {
        // Loop used to iterate over all the instructors
        Iterator<Instructor> iterator = listOfInstructors.iterator();
        while (iterator.hasNext()) {
            Instructor specificInstructor = iterator.next();
            if (specificInstructor.getId().equals(targetId)) {
                iterator.remove();
                System.out.println(specificInstructor.getName() + " with the ID: " + specificInstructor.getId() + " removed
successfully.");
                return true;
            }
        }
        System.out.println("Instructor with the ID: " + targetId + " not found.");
        return false;
    }

    public void addInstructor(Instructor specificInstructor, List<Instructor> listOfInstructors) {
        listOfInstructors.add(specificInstructor);
        System.out.printf("Instructor added: %s with the ID: %s", specificInstructor.getName(), specificInstructor.getId());
    }

    public void addStudent(Student specificStudent, List<Student> listOfStudents) {
        listOfStudents.add(specificStudent);
        System.out.printf("Student added: %s with the ID: %s", specificStudent.getName(), specificStudent.getId());
    }

    // Update user info.

    public void updateStudent(String targetId, List<Student> listOfStudents, String newName, String newId, String
newPassword, String newEmail, String newPhoneNumber, User.Role newRole, String newAddress) {

        Student updateStudentInfo = null;
        for (Student student : listOfStudents) {
            if (student.getId().equals(targetId)) {
                updateStudentInfo = student;
                break;
            }
        }
        if (updateStudentInfo != null) {
            updateStudentInfo.setName(newName);
            updateStudentInfo.setPassword(newPassword);
            updateStudentInfo.setEmail(newEmail);
            updateStudentInfo.setPhoneNumber(newPhoneNumber);
            updateStudentInfo.setRole(newRole);
            updateStudentInfo.setAddress(newAddress);
        } else {
            System.out.println("Student with the ID " + targetId + " not found.");
        }
    }

    public void updateInstructor(String instructorTargetId, List<Instructor> listOfInstructors, String newName, String
newPassword, String newEmail, String newPhoneNumber, User.Role newRole, String newAddress) {

        Instructor updateInstructorInfo = null;
        for (Instructor instructor : listOfInstructors) {
            if (instructor.getId().equals(instructorTargetId)) {
                updateInstructorInfo = instructor;
                break;
            }
        }
        if (updateInstructorInfo != null) {
            updateInstructorInfo.setName(newName);
            updateInstructorInfo.setPassword(newPassword);
            updateInstructorInfo.setEmail(newEmail);
            updateInstructorInfo.setPhoneNumber(newPhoneNumber);
            updateInstructorInfo.setRole(newRole);
            updateInstructorInfo.setAddress(newAddress);
        } else {
            System.out.println("Instructor with the ID " + instructorTargetId + " not found.");
        }
    }
    // Course related.
    public void addCourse(List<Course> listOfCourses, Course specificCourse) {
        if (listOfCourses.add(specificCourse)) {
            System.out.printf("Course added: (%s) (%s) succesfully.", specificCourse.getCourseName(),
specificCourse.getCourseCode());
        } else {
            System.out.println("Course already exists: " + specificCourse.getCourseName());
        }
    }

    public void assignInstructor(Course specificCourse, Instructor specificInstructor) {
        if (specificCourse.getInstructor() == null) {
            specificCourse.setInstructor(specificInstructor);
            System.out.printf("Instructor %s is assigned to %s", specificInstructor.getName(),
specificCourse.getCourseName());
        } else if (specificCourse.getInstructor().equals(specificInstructor)) {
            System.out.printf("Instructor %s is already assigned to %s", specificInstructor.getName(),
specificCourse.getCourseName());
        } else {
            System.out.println("Course already has a different instructor assigned.");
        }
    }

    public void closeCourse(Course specificCourse, List<Course> listOfCourses) {
        if (listOfCourses.contains(specificCourse)) {
            if (specificCourse.getEnrolledStudents().size() >= specificCourse.getCapacity()) {
                System.out.println(specificCourse.getCourseName() + " is at full capacity.");
                specificCourse.setEnrollmentStatus(EnrollmentStatusEnum.Closed);
            } else if (specificCourse.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Closed) {
                System.out.println(specificCourse.getCourseName() + " enrollment status is closed.");
            }
        } else if (!listOfCourses.contains(specificCourse)) {
            System.out.println(specificCourse.getCourseName() + " not found.");
        } else {
            System.out.println(specificCourse.getCourseName() + " still has available capacity.");
        }
    }

    public void viewStudentList(List<Student> listOfStudents) {
        if (listOfStudents == null || listOfStudents.isEmpty()) {
            System.out.println("\nNo students are on record.");
        } else {
            for (Student student : listOfStudents) {
                System.out.println(student.toString());
            }
        }
    }

    public void viewInstructorList(List<Instructor> instructors) {
        if (instructors == null || instructors.isEmpty()) {
            System.out.println("\nNo instructors are on record.");
        } else {
            for (Instructor instructor : instructors) {
                System.out.println(instructor.toString());
            }
        }
    }
```

```
}
```

Assessments

```
package schoolcourseenrolmentsystem;

import java.util.*;

public class Assessment {
    // Defines the types of exams/assessments available (quizzes, midterms, final
    // exam, project).
    public enum ExamType {
        Quiz_1, Quiz_2, Quiz_3, Quiz_4, Midterm_1, Midterm_2, Final, Project

    }

    // Attributes
    private String studentId;
    private String courseCode;
    private ExamType examType;
    private double score;
    private String assessmentName;

    // Constructor
    public Assessment(String studentId, String courseCode, ExamType examType, double score, String
assessmentName) {
        setStudentId(studentId);
        setCourseCode(courseCode);
        setExamType(examType);
        setScore(score);
        setAssessmentName(assessmentName);
    }
    //====================== Setters and Getters ======================
    public String getStudentId() {
        return studentId;
    }

    public void setStudentId(String studentId) {
```

```
    this.studentId = studentId;
}

public String getCourseCode() {
    return courseCode;
}

public void setCourseCode(String courseCode) {
    this.courseCode = courseCode;
}

public ExamType getExamType() {
    return examType;
}

public void setExamType(ExamType examType) {
    this.examType = examType;
}

public double getScore() {
    return score;
}

public void setScore(double score) {
    this.score = score;
}

public String getAssessmentName() {
    return assessmentName;
}

public void setAssessmentName(String assessmentName) {
    this.assessmentName = assessmentName;
}

//====================== Methods======================
    public void assignGrade(List<Assessment> listOfGrades, String studentId, String courseCode,
Assessment.ExamType examType, double score) {
        boolean gradeUpdated = false;

        for (Assessment grade : listOfGrades) {
            if (grade.getCourseCode().equals(courseCode)
                    && grade.getStudentId().equals(studentId)
                    && grade.getExamType() == examType) {
                grade.setScore(score); // Update existing score
                System.out.println("Grade updated successfully.");
                gradeUpdated = true;
                break;
            }
        }
        if (!gradeUpdated) {
            Assessment newGrade = new Assessment(studentId, courseCode, examType, score, examType.name());
            listOfGrades.add(newGrade);
            System.out.println("Grade recorded successfully.");
        }
    }

    public void viewAllGradesForCourse(List<Assessment> listOfGrades, Student student) {
        List<Course> courses = student.getEnrolledCoursesList();
        if (courses.isEmpty()) {
            System.out.println("You are not enrolled in any courses.");
            return;
        }
        System.out.println("Select a course to view all your grades:");
        for (int i = 0; i < courses.size(); i++) {
            System.out.printf((i + 1) + ".%s (%s).", courses.get(i).getCourseName(), courses.get(i).getCourseCode());
        }
        int courseChoice = Helpers.getSafeIntInput("Option: ");
        if (courseChoice < 1 || courseChoice > courses.size()) { //Option can't be less than 1 or more than listed courses.
            System.out.println("Invalid course selection.");
            return;
        }
        String courseCode = courses.get(courseChoice - 1).getCourseCode();
        boolean found = false;

        for (Assessment grade : listOfGrades) {
            if (grade.getStudentId().equals(student.getId()) && grade.getCourseCode().equals(courseCode)) {
                System.out.println(grade.getExamType() + ": " + grade.getScore());
                found = true;
            }
        }
        if (!found) {
            System.out.println("No grades found for this course.");
        }
    }

    public void viewTotalAverageGrade(List<Assessment> listOfGrades, String studentId, String courseCode) {
        double total = 0;
        int count = 0;

        for (Assessment grade : listOfGrades) {
            if (grade.getStudentId().equals(studentId) && grade.getCourseCode().equals(courseCode)) {
                total += grade.getScore();
                count++;
            }
        }
        if (count > 0) {
            double average = total / count;
            System.out.println("Your total average grade in " + courseCode + " is: " + average);
        } else {
            System.out.println("No grades found for this course.");
        }
    }

    public void viewSpecificExamScore(List<Assessment> listOfGrades, Student student) {
        List<Course> listOfCourses = student.getEnrolledCoursesList();
        if (listOfCourses.isEmpty()) {
            System.out.println("You are not enrolled in any courses.");
            return;
        }
        System.out.println(“Select a course: “);
        for (int i = 0; i < listOfCourses.size(); i++) {
            System.out.printf((i + 1) + “.%s (%s).\n”, listOfCourses.get(i).getCourseName(),
      listOfCourses.get(i).getCourseCode());
        }
        int courseChoice = Helpers.getSafeIntInput(“Choice: “);
        if (courseChoice < 1 || courseChoice > listOfCourses.size()) {
            System.out.println(“Invalid selection.”);
            return;
        }

        String courseCode = listOfCourses.get(courseChoice - 1).getCourseCode();

        // Choose exam type
        Assessment.ExamType[] types = Assessment.ExamType.values();
        System.out.println(“Select an exam to view:”);
        for (int i = 0; i < types.length; i++) {
            System.out.println((i + 1) + “. “ + types[i]);
        }

        int examChoice = Helpers.getSafeIntInput(“Choice: “);
        if (examChoice < 1 || examChoice > types.length) {
            System.out.println(“Invalid exam type.”);
            return;
        }

        Assessment.ExamType selected = types[examChoice - 1];
        boolean found = false;
        for (Assessment assessment : listOfGrades) {
            if (assessment.getStudentId().equals(student.getId()) &&
                assessment.getCourseCode().equals(courseCode) &&
                assessment.getExamType() == selected) {
                System.out.println(“Your “ + selected + “ score: “ + assessment.getScore());
                found = true;
            }
        }
```

```
        if (!found) {
            System.out.println("No " + selected + " score found for this course.");
        }
    }
}
```

Course Class

```
package schoolcourseenrolmentsystem;

import java.util.*;

public class Course {
    // Similar to final but with 2+ options.
    // Enum to represent course enrollment status
    public enum EnrollmentStatusEnum {
        Closed,
        Open
    }

    // Attributes
    private String courseName, courseCode, schedule, description;
    private EnrollmentStatusEnum enrollmentStatus;
    private Instructor instructor;
    private int capacity;
    private List<Assessment> grades;
    private int creditHours;

    // Fixes java.lang.StackOverflowError
    private List<Student> enrolledStudents; // List of students currently enrolled

    // Constructor
    public Course(String courseName, String courseCode, String schedule, String description,
            EnrollmentStatusEnum enrollmentStatus, Instructor instructor, int capacity, List<Assessment> listOfGrades,
            int creditHours) {
        setCourseName(courseName);
        setCourseCode(courseCode);
        setSchedule(schedule);
        setDescription(description);
        setEnrollmentStatus(enrollmentStatus);
        setInstructor(instructor);
        setCapacity(capacity);
        setAssessment(listOfGrades);
        setCreditHours(creditHours);
        // Important: initialize enrolledStudents list to avoid null pointer exceptions
        this.enrolledStudents = new ArrayList<>();
    }

    // Default (empty) constructor
    public Course() {
    }

    // ====================== Setters and Getters ======================
    public void setEnrolledStudents(List<Student> enrolledStudentList) {
        this.enrolledStudents = enrolledStudentList;
    }

    public EnrollmentStatusEnum getEnrollmentStatus() {
        return enrollmentStatus;
    }

    public void setEnrollmentStatus(EnrollmentStatusEnum enrollmentStatus) {
        this.enrollmentStatus = enrollmentStatus;
    }

    public String getCourseName() {
        return courseName;
    }

    public void setCourseName(String courseName) {
        if (courseName == null || courseName.isEmpty()) {
            System.out.println("Course name cannot be null or empty.");
            return;
        } else {
            this.courseName = courseName;
        }
    }

    public int getCreditHours() {
        return creditHours;
    }

    public void setCreditHours(int creditHours) {
        if (creditHours > 0) {
            this.creditHours = creditHours;
        } else {
            System.out.println(“Credit hours can not be <= 0”);
            return;
        }
    }

    public List<Student> getEnrolledStudents() {
        return enrolledStudents;
    }

    public String getCourseCode() {
        return courseCode;
    }

    public void setCourseCode(String courseCode) {
        if (courseCode == null || courseCode.isEmpty()) {
            System.out.println(“Course code cannot be null or empty.”);
            return;
        } else {
            this.courseCode = courseCode;
        }
    }

    public String getSchedule() {
        return schedule;
    }
```

```
public void setSchedule(String schedule) {
    // Schedule is allowed to be null because instructors can update it later.
    if (schedule == null || schedule.isEmpty()) {
        this.schedule = null;
    } else {
        this.schedule = schedule;
    }
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    // Description is allowed to be null because instructors can update it later.
    if (description == null || description.isEmpty()) {
        this.description = null;
    } else {
        this.description = description;
    }
}



public Instructor getInstructor() {
    return instructor;
}

public void setInstructor(Instructor instructor) {
    if (instructor == null) {
        this.instructor = null;
        return;
    }
    this.instructor = instructor;
}

public int getCapacity() {
    return capacity;
```

```
}

public void setCapacity(int capacity) {
    if (capacity > 0) {
        this.capacity = capacity;
    } else {
        System.out.println("Error: Capacity must be > 0.");
        return;
    }
}

public List<Assessment> getGrades() {
    return grades;
}

public List<Assessment> setAssessment(List<Assessment> listOfGrades) {
    if (listOfGrades == null) {
        this.grades = null;
        return null;
    } else {
        this.grades = listOfGrades;
        return listOfGrades;
    }
}

public void setGrades(List<Assessment> listOfGrades) {
    if (listOfGrades == null) {
        this.grades = null;
    } else {
        this.grades = listOfGrades;
    }
}


// ======================= Methods =======================
// Check if the course has reached its capacity
public boolean isFull() {
        return enrolledStudents != null && this.getEnrolledStudents().size() >= capacity;
    }
}
```

Helpers Class

```
package schoolcourseenrolmentsystem;

import java.util.*;
import java.util.function.Predicate;

public class Helpers {
    public static final Scanner input = new Scanner(System.in);

    /// <T extends User<T>> This method works with any User (Admin, Instructor,
    /// Student). T inherits from User<T>.
    /// The single T will return a single object of type T.
    public static <T extends User<T>> T login(List<T> usersList, T tempUser) {
        T loggedInUser = null;
        try {
            System.out.print("\nPlease enter your ID: ");
            String id = input.next();
            input.nextLine(); // Buffer

            // Check if the ID exists
            boolean idExists = false;
            for (T user : usersList) {
                if (user.getId().equals(id)) {
                    idExists = true;
                    break;
                }
            }
            if (!idExists) {
                System.out.println("No user found with the entered ID.");
            }
            loggedInUser = validatePasswordWithRetries(usersList, tempUser, id);
            return loggedInUser;
        } catch (Exception e) {
```

```
        return loggedInUser;
    }
}

public static <T extends User<T>> T validatePasswordWithRetries(List<T> usersList, T tempUser, String tempID) {
    T loggedInUser = null;
    int attempts = 0;
    final int maxAttempts = 3;

    while (attempts < maxAttempts) {
        System.out.print("\nPlease enter your password: ");
        String password = input.nextLine();

        loggedInUser = tempUser.login(usersList, tempID, password);

        if (loggedInUser != null) {
            return loggedInUser;
        } else {
            attempts++;
            if (attempts < maxAttempts) {
                System.out.println("Incorrect password. Try again (" + (maxAttempts - attempts) + " attempts left).");
            }
        }
    }
    System.out.println("Too many failed attempts. Returning to main menu.");
    return null;
}

public static boolean ValidatePassword(String password) {
    boolean isValid = false;
    try {
        if (password != null && password != "" && password.length() > 8) {
            isValid = true;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return isValid;
}

public static boolean ValidateEmail(String email) {
    boolean isValid = false;
    try {
        if (email != null && email != "" && email.contains("@") && email.contains(".")) {
            isValid = true;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return isValid;
}

public static boolean ValidatePhoneNumber(String phoneNumber) {
    boolean isValid = false;
    try {
        if (phoneNumber != null && phoneNumber != "" && phoneNumber.length() == 10) {
            isValid = true;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return isValid;
}

public static boolean ValidateAddress(String address) {
    boolean isValid = false;
    try {
        if (null != address && address != "" && address.length() > 3) {
            isValid = true;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return isValid;
```

```
}


public static Boolean ValidateName(String name) {
    boolean isValid = false;
    try {
        if (null != name && name != "" && name.length() > 3) {
            isValid = true;
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    return isValid;
}
public static User.Role checkValidityOfRole() {
    while (true) {
        System.out.println("Select a role: \n1. STUDENT \n2. INSTRUCTOR \n3. ADMIN \nOption:");
        try {
            int Option = Helpers.getSafeIntInput("\nOption: ");

            switch (Option) {
                case 1:
                    return User.Role.STUDENT;
                case 2:
                    return User.Role.INSTRUCTOR;
                case 3:
                    return User.Role.ADMIN;
                default:
                System.out.println("Invalid selection. Please enter 1, 2, or 3.");
            }
        } catch (InputMismatchException e) {
            System.out.println("Invalid input. Please enter a number.");
            input.nextLine(); // clear the invalid input
        }
    }
}


/// Method created to avoid InputMismatchException
```

```
public static int getSafeIntInput(String prompt) {
        try {
            int number = -1; // -1 is typically related to false or not true. This is a default number.
            boolean valid = false;
            while (!valid) {
                System.out.print(prompt);
                //number = Integer.parseInt(input.nextLine());
                number= input.nextInt();
                input.nextLine();
                valid = true;
            }
            return number;
        } catch (Exception e) {
            System.out.println("Invalid input. Please enter a number.");
            input.nextLine(); // Clear the invalid input
            return getSafeIntInput(prompt); // Retry
        }
    }

    public static User.Role updateRolePrompt(User.Role currentRole) {
        System.out.println("\nDo you want to change the role? \n1.Yes \n2.No");
        int choice = getSafeIntInput("\nOption: ");
        input.nextLine();

        if (choice == 1) {
            return checkValidityOfRole(); // Assuming you already have this
        } else {
            System.out.println("Role remains unchanged.");
            return currentRole;
        }
    }



    // Predicate <> helps us use premade validators.
    public static String updateFieldWithPrompt(String fieldName, String currentValue, Predicate<String> validator) {
        System.out.println("\nDo you want to change your " + fieldName + "? \n1.Yes \n2.No");
        int choice = Helpers.getSafeIntInput("\nOption: ");
    if (choice == 1) {
        System.out.print("\nEnter new " + fieldName + ": ");
        String newValue = input.nextLine();
        while (!validator.test(newValue)) {
            System.out.println("Invalid " + fieldName + ". Please try again.");
            System.out.print("\nEnter new " + fieldName + ": ");
            newValue = input.nextLine();
        }
        return newValue;
    } else {
        System.out.println(fieldName + " remains unchanged.");
        return currentValue;
    }
}

public static String createUserWithPrompt(String fieldName, Predicate<String> validator) {
    String inputValue;
    while (true) {
        System.out.print("\nEnter " + fieldName + ": ");
        inputValue = input.nextLine();
        if (!validator.test(inputValue)) { // .test() method is just how you invoke the logic inside the Predicate<>
            System.out.println("Invalid " + fieldName + ". Please try again.");
        } else {
            break;
        }
    }
    return inputValue;
}
public static String GenerateRandomID() {
    String numbers = "0123456789";
    int ID_Length = 10;
    Random random = new Random();

    StringBuilder id = new StringBuilder(ID_Length);

    for (int i = 0; i < ID_Length; i++) {
        int randomIndex = random.nextInt(numbers.length());
        char randomChar = numbers.charAt(randomIndex);
        id.append(randomChar);
    }
    return id.toString();
}

// MENUS
public static void showStudentMenu() {
    System.out.println("");
    System.out.println("\n---------Student Menu---------");
    System.out.println("1. View avaliable courses.");
    System.out.println("2. Enroll in course.");
    System.out.println("3. View enrolled courses.");
    System.out.println("4. View credit limit.");
    System.out.println("5. Drop course.");
    System.out.println("6. View grades.");
    System.out.println("7. Update personal information.");
    System.out.println("8. Logout.");
}

public static void showInstructorMenu() {
    System.out.println("");
    System.out.println("\n---------Instructor Menu---------");
    System.out.println("1. View enrolled students.");
    System.out.println("2. Grade assignments.");
    System.out.println("3. Update course information.");
    System.out.println("4. Update personal information.");
    System.out.println("5. Logout.");
}

public static void showAdminMenu() {
    System.out.println("");
    System.out.println("\n---------Administrator Menu---------");
    System.out.println("1.  Add Students.");
    System.out.println("2.  Remove Students.");
    System.out.println("3.  View student list.");
```

```
        System.out.println(“4.  Add Instructors.”);
        System.out.println(“5.  Remove Instructors.”);
        System.out.println(“6.  View instructor list.”);
        System.out.println(“7.  Add Course.”);
        System.out.println(“8.  Close Course.”);
        System.out.println(“9.  Update Student Info.”);
        System.out.println(“10. Update Instructor Info.”);
        System.out.println(“11. Assign Instructor to Course.”);
        System.out.println(“12. View Enrollment Statistics.”);
        System.out.println(“13. Generate Reports.”);
        System.out.println(“14. Logout.”);
    }

    // CASES

    // INSTRUCTOR CASES
    public static void updateCourseInfo(Instructor instructor, List<Course> listOfCourses) {
        try {
            System.out.println(“\n---------Update Course Information---------\nEnter the course code of the course you
would like to update: “);
            String courseCode = input.next();
            input.nextLine(); // Buffer

            System.out.print(“\nEnter new schedule:”);
            String newSchedule = input.nextLine();

            System.out.print(“\nEnter new description: “);
            String newDescription = input.nextLine();

            instructor.updateCourseInfo(courseCode, listOfCourses, newSchedule, newDescription);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void updateInstructorProfile(Instructor instructor) {
        try {
            System.out.println(“\n---------Update Personal Profile---------”);
            System.out.println(“\nPlease fill out the form to update your profile.”);

            String updatedPassword = Helpers.updateFieldWithPrompt(“password”, instructor.getPassword(),
Helpers::ValidatePassword); // Helpers::ValidatePassword is like an inhanced for-each loop
            String updateEmail = Helpers.updateFieldWithPrompt(“email”, instructor.getEmail(), Helpers::ValidateEmail);
            String updatePhone = Helpers.updateFieldWithPrompt(“phone number”, instructor.getPhoneNumber(),
Helpers::ValidatePhoneNumber);
            String updateAddress = Helpers.updateFieldWithPrompt(“address”, instructor.getAddress(),
Helpers::ValidateAddress);
            instructor.updateInstructorPersonalInfo(updatedPassword, updateEmail, updatePhone, updateAddress);
            System.out.println(“Instructor “ + instructor.getName() + “'s information updated.”);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    // STUDENT CASES
    public static void enroll_In_Course_Student(List<Course> listOfCourses, Student student) {
        try {
            System.out.println(“\n---------Enroll In Course---------”);

            // Filter available courses
            List<Course> availableCourses = new ArrayList<>();
            for (Course course : listOfCourses) {
                if (!course.isFull() &&
                        course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Open &&
!student.getEnrolledCoursesList().contains(course)) {
                    availableCourses.add(course);
                }
            }

            // Display results
            if (availableCourses.isEmpty()) {
                System.out.println(“No available courses to enroll in.”);
            } else {
                System.out.println(“Courses available to enroll in:”);
                for (int i = 0; i < availableCourses.size(); i++) {
                    Course course = availableCourses.get(i);
                    System.out.printf("%d. %s (%s)\n", (i + 1), course.getCourseName(), course.getCourseCode());
                }
    System.out.print("\nEnter the number of the course you would like to enroll in: ");
                int selectedCourse = input.nextInt();
                input.nextLine(); // Consume newline

                if (selectedCourse >= 1 && selectedCourse <= availableCourses.size()) {
                    Course chosenCourse = availableCourses.get(selectedCourse - 1);
                    student.enroll_In_Course(chosenCourse);
                } else {
                    System.out.println("Invalid selection: " + selectedCourse);
                }
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public static void dropCourse(List<Course> listOfCourses, Student student) {
        try {
            System.out.println("\n---------Unenroll In Course---------");
            System.out.print("\nEnter course code of the course you would like to drop: ");
            String dropCourseCode = input.next();
            input.nextLine(); // Buffer

            Course courseCodeToDrop = null;

            for (Course course : listOfCourses) {
                if (course.getCourseCode().equalsIgnoreCase(dropCourseCode)) {
                    courseCodeToDrop = course;
                    break;
                }
            }
            if (courseCodeToDrop != null) {
                student.dropCourse(courseCodeToDrop);
            } else {
           System.out.println(“No course found with the code” + dropCourseCode + “'.”);
            }
          } catch (Exception e) {
            System.out.println(e.getMessage());
          }
       }

       public static void updateStudentProfile(Student specificStudent) {
          try {
            System.out.println(“\n---------Update Personal Profile---------”);
      System.out.println(“\nPlease enter the needed information to update your profile.”);
            String updatedPassword = Helpers.updateFieldWithPrompt(“password”, specificStudent.getPassword(),
    Helpers::ValidatePassword); // Helpers::ValidatePassword is like an inhanced for-each loop
            String updateEmail = Helpers.updateFieldWithPrompt(“email”, specificStudent.getEmail(),
    Helpers::ValidateEmail);
            String updatePhone = Helpers.updateFieldWithPrompt(“phone number”, specificStudent.getPhoneNumber(),
    Helpers::ValidatePhoneNumber);
            String updateAddress = Helpers.updateFieldWithPrompt(“address”, specificStudent.getAddress(),
    Helpers::ValidateAddress);

            specificStudent.updateStudentPersonalInfo(updatedPassword, updateEmail, updatePhone, updateAddress);
          } catch (Exception e) {
            System.out.println(e.getMessage());
          }
       }

       // ADMIN CASES
       public static void addStudent(Administrator administrator, List<Student> listOfStudents) {
          try {
            System.out.println(“\n---------Create New Student---------”);
            String studentName = ““;
            while (true) {
              System.out.print(“\nEnter student name: “);
              studentName = input.nextLine();

              if (!Helpers.ValidateName(studentName)) {
                  System.out.println(“Invalid name. Please enter a valid name.”);
                  continue;
              }
              break;
            }
            String studentID;
            while (true) {
              studentID = GenerateRandomID();
              boolean exists = false;
              for (Student student : listOfStudents) {
                  if (student.getId().equals(studentID)) {
                      exists = true;
                      break;
                  }
              }
              if (!exists) {
                  break; // We found a unique ID, exit the loop
              }
            }
            String createPassword = Helpers.createUserWithPrompt(“password”, Helpers::ValidatePassword);
            String createEmail = Helpers.createUserWithPrompt(“email”, Helpers::ValidateEmail);
            String createPhoneNumber = Helpers.createUserWithPrompt(“phone number”,
    Helpers::ValidatePhoneNumber);
            String createAddress = Helpers.createUserWithPrompt(“address”, Helpers::ValidateAddress);

            System.out.print(“\nEnter credit limit: “);
            int creditLimit = input.nextInt();
            input.nextLine(); // Buffer

            Student newStudent = new Student(studentName, studentID, createPassword, createEmail,
    createPhoneNumber, createAddress, creditLimit, new ArrayList<>());

            administrator.addStudent(newStudent, listOfStudents);

          } catch (Exception e) {
            System.out.println(e.getMessage());
          }
```

```
}

public static void removeStudent(Administrator administrator, List<Student> listOfStudents) {
    try {
        System.out.println("\n---------Remove Student---------");
        System.out.print("\nEnter the ID of the student to remove: ");
        String targetId = input.next();
        input.nextLine(); // Buffer
        administrator.removeStudent(listOfStudents, targetId);

    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("Invalid option.");
}

public static void removeInstructor(Administrator administrator, List<Instructor> listOfInstructors) {
    try {
        System.out.println("\n---------Remove Instructor---------");
        System.out.print("\nEnter the ID of the instructor to remove: ");
        String targetId = input.next();
        input.nextLine(); // Buffer
        administrator.removeInstructor(listOfInstructors, targetId);
    } catch (Exception e) {
        System.out.println("Invalid option.");
    }
}

public static void addInstructor(Administrator administrator, List<Instructor> listOfInstructors) {
    try {
        System.out.println("\n---------Create New Instructor---------");
        String instructorName = "";
        while (true) {
            System.out.print("\nEnter instructor name: ");
            instructorName = input.nextLine();
            if (!Helpers.ValidateName(instructorName)) {
                System.out.println("Invalid name. Please enter a valid name.");
                continue;
            }
            break;
        }
        String instructorID;
        while (true) {
            instructorID = GenerateRandomID();
            boolean exists = false;
            for (Instructor instructor : listOfInstructors) {
                if (instructor.getId().equals(instructorID)) {
                    exists = true;
                    break;
                }
            }
            if (!exists) {
                break;
            }
        }
        String createPassword = Helpers.createUserWithPrompt("password", Helpers::ValidatePassword);
        String createEmail = Helpers.createUserWithPrompt("email", Helpers::ValidateEmail);
        String createPhoneNumber = Helpers.createUserWithPrompt("phone number",
Helpers::ValidatePhoneNumber);
        String createAddress = Helpers.createUserWithPrompt("address", Helpers::ValidateAddress);

        Instructor newInstructor = new Instructor(instructorName, instructorID, createPassword, createEmail,
createPhoneNumber, createAddress, new ArrayList<>());

        administrator.addInstructor(newInstructor, listOfInstructors);
    } catch (Exception e) {
        System.out.println("Invalid option.");
    }
}

public static void addCourse(Administrator administrator, List<Course> listOfCourses,
        List<Instructor> listOfInstructors) {
    try {
        System.out.println("\n---------Create Course---------");
        System.out.print("\nEnter the course's information that you would like to create.\n\nEnter course name: ");
        String courseName = input.nextLine();

        System.out.print("\nEnter course code: ");
        String courseCode = input.nextLine();

        System.out.print("\nEnter course capacity: ");
        int courseCapacity = input.nextInt();
        input.nextLine(); // Buffer

        System.out.print("\nEnter course credit hours: ");
        int courseCreditHours = input.nextInt();
        input.nextLine(); // Buffer

        Course newCourse = new Course(courseName, courseCode, null, null, null, null, courseCapacity, null,
courseCreditHours);

        System.out.println("\nWould you like to assign instructor now? \n1. Assign now. \n2. Assign later.");
        int assignChoice = Helpers.getSafeIntInput("\nOption: ");

        if (assignChoice == 1) {
          System.out.print("\nAssign instructor by ID: ");
          String assignInstructorID = input.nextLine();

          // Validate instructor list is not empty.
          if (listOfInstructors.isEmpty()) {
            System.out.println("\nNo instructors available. Please add an instructor before creating a course.");
            return; // Stop the method early
          }
          Instructor selectedInstructor = null;
          for (Instructor instructor : listOfInstructors) {
            if (instructor.getId().equals(assignInstructorID)) {
              selectedInstructor = instructor;
              break;
            }
          }
          administrator.assignInstructor(newCourse, selectedInstructor);
```

```
            if (selectedInstructor == null) {
    System.out.println("\nInstructor with ID " + assignInstructorID + " not found.");
            }
        }
        if (assignChoice == 2) {
            System.out.println("To be assigned.");
        }

        // Assign course status
        System.out.println("\nEnter course status: \n1. Open \n2.Closed");
        int courseStatus = Helpers.getSafeIntInput("Option: ");

        if (courseStatus == 1) {
            newCourse.setEnrollmentStatus(Course.EnrollmentStatusEnum.Open);
        } else if (courseStatus == 2) {
            newCourse.setEnrollmentStatus(Course.EnrollmentStatusEnum.Closed);
        }
        administrator.addCourse(listOfCourses, newCourse);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public static void adminUpdateStudentProfile(Administrator administrator, List<Student> listOfStudents) {
    try {
        System.out.print("\n---------Update Student Profile---------\nPlease enter the requiered information
below.\nStudent's name: ");
        String name = input.nextLine();

        System.out.print("\nStudent ID: ");
        String ID = input.next();
        input.nextLine(); // Buffer

        Student studentToUpdate = null;

        for (Student student : listOfStudents) {
            if (student.getName().equals(name) && student.getId().equals(ID)) {
                studentToUpdate = student;
                break;
            }
        }
        if (studentToUpdate != null) {
    System.out.println("Update " + studentToUpdate.getName() + " details below: ");

            String updateName = Helpers.updateFieldWithPrompt("name", studentToUpdate.getName(),
Helpers::ValidateName);
            String updatePassword = Helpers.updateFieldWithPrompt("password", studentToUpdate.getPassword(),
Helpers::ValidatePassword);
            String updateEmail = Helpers.updateFieldWithPrompt("email",
studentToUpdate.getEmail(),Helpers::ValidateEmail);
            String updatePhoneNumber = Helpers.updateFieldWithPrompt("phone number",
studentToUpdate.getPhoneNumber(), Helpers::ValidatePhoneNumber);
            String updateAddress = Helpers.updateFieldWithPrompt("address", studentToUpdate.getAddress(),
Helpers::ValidateAddress);
            User.Role updateRole = Helpers.updateRolePrompt(studentToUpdate.getRole());

            // Option to update credit limit
            System.out.printf("\nDo you want to update %s's credit limit?\n1. Yes\n2. No",
studentToUpdate.getName());
            int creditLimitChoice = input.nextInt();

            if (creditLimitChoice == 1) {
                System.out.print("\nEnter new credit limit: ");
                int updatedStudentcreditLimit = input.nextInt();
                studentToUpdate.setCreditLimit(updatedStudentcreditLimit);
            } else if (creditLimitChoice == 2) {
                System.out.println("Credit limit number remains unchanged.");
            }
            // Apply updates
            studentToUpdate.setName(updateName);
            studentToUpdate.setPassword(updatePassword);
            studentToUpdate.setEmail(updateEmail);
            studentToUpdate.setPhoneNumber(updatePhoneNumber);
            studentToUpdate.setAddress(updateAddress);
            studentToUpdate.setRole(updateRole);

    System.out.println("Student " + updateName + "'s profile updated successfully.");
        }
     } catch (Exception e) {
        System.out.println(e.getMessage());
     }
    }

    public static void adminUpdateInstructorProfile(Administrator administrator, List<Instructor> listOfInstructors) {
     try {
        System.out.println("\n---------Update Instuctor Profile---------\nPlease enter the requiered information below.");

        System.out.print("\nEnter instructor name: ");
        String instructorName = input.nextLine();

        System.out.print("\nEnter instructor ID: ");
        String instructorId = input.next();
        input.nextLine(); // Buffer

        Instructor instructorToUpdate = null;

        for (Instructor instructor : listOfInstructors) {
            if (instructor.getName().equals(instructorName) && instructor.getId().equals(instructorId)) {
                instructorToUpdate = instructor;
                break;
            }
        }
        if (instructorToUpdate != null) {
            System.out.println("Update " + instructorToUpdate.getName() + "'s details below: ");

            String updateName = Helpers.updateFieldWithPrompt("name", instructorToUpdate.getName(),
Helpers::ValidateName);
            String updatePassword = Helpers.updateFieldWithPrompt("password", instructorToUpdate.getPassword(),
Helpers::ValidatePassword);
```

```
            String updateEmail = Helpers.updateFieldWithPrompt(“email”,
instructorToUpdate.getEmail(),Helpers::ValidateEmail);
            String updatePhoneNumber = Helpers.updateFieldWithPrompt(“phone
number”,instructorToUpdate.getPhoneNumber(), Helpers::ValidatePhoneNumber);
            String updateAddress = Helpers.updateFieldWithPrompt(“address”, instructorToUpdate.getAddress(),
Helpers::ValidateAddress);
            User.Role updateRole = Helpers.updateRolePrompt(instructorToUpdate.getRole());
            // Apply updates
            instructorToUpdate.setName(updateName);
            instructorToUpdate.setPassword(updatePassword);
            instructorToUpdate.setEmail(updateEmail);
            instructorToUpdate.setPhoneNumber(updatePhoneNumber);
            instructorToUpdate.setAddress(updateAddress);
            instructorToUpdate.setRole(updateRole);

  System.out.println(“Instructor “ + updateName + “'s profile updated successfully.”);
          }
        } catch (Exception e) {
          System.out.println(e.getMessage());
        }
    }

    public static void assignInstructor(Administrator administrator, List<Course> listOfCourses,
          List<Instructor> listOfInstructors) {
        try {
          System.out.println(“\n---------Available Courses---------”);

          for (int i = 0; i < listOfCourses.size(); i++) {
          /// The i+1 is for the indext of which the admin will select. The i is for
          /// 'this' spesific course.
            System.out.printf(“%d. %s (%s).\n”, (i + 1), listOfCourses.get(i).getCourseName(),
listOfCourses.get(i).getCourseCode());
          }
   System.out.print(“\nEnter the number of the course to assign the instructor to: “);
          int courseIndex = input.nextInt() - 1;
          input.nextLine(); // Buffer
          // Basic validation.
          if (courseIndex < 0 || courseIndex >= listOfCourses.size()) {
            System.out.println("Invalid course selection.");
            return;
          }
          System.out.print("\nEnter the ID of the instructor to assign: ");
          String instructorID2 = input.next();
          input.nextLine(); // Buffer

          Instructor assignInstructor = null;

          for (Instructor instructor : listOfInstructors) {
            if (instructor.getId().equals(instructorID2)) {
                assignInstructor = instructor;
                break;
            }
          }
          if (assignInstructor != null) {
            Course selectedCourse = listOfCourses.get(courseIndex);
            administrator.assignInstructor(selectedCourse, assignInstructor);
          } else {
          System.out.println("Instructor with ID " + instructorID2 + " not found.");
          }
        } catch (Exception e) {
          System.out.println(e.getMessage());
        }
    }

    public static void closeCourse(Administrator administrator, List<Course> listOfCourses) {
        try {
          System.out.println("\n---------Close Courses---------\nEnter the details requiered below.\nEnter course name: ");
          String closeCourseName = input.nextLine();
          System.out.print("\nEnter course code: ");
          String closeCourseCode = input.next();
          input.nextLine(); // Buffer

          Course courseToClose = null;
```

```
            for (Course course : listOfCourses) {
                if (course.getCourseName().equals(closeCourseName) &&
course.getCourseCode().equals(closeCourseCode)) {
                    courseToClose = course;
                    break;
                }
            }
            if (courseToClose != null) {
                administrator.closeCourse(courseToClose, listOfCourses);
            } else {
                System.out.println("Course not found.");
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    // Statistics.
    public static void viewEnrollmentStatistics(List<Course> listOfCourses) {
        int totalEnrolled = 0;
        boolean openCoursesExist = false;

        System.out.println("\n---------Enrollment Statistics---------");
        for (Course course : listOfCourses) {
            int count = course.getEnrolledStudents() != null ? course.getEnrolledStudents().size() : 0;
            totalEnrolled += count;
            System.out.printf("Course: %s (%s) - Enrolled: %d", course.getCourseName(), count);
            if (course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Open) {
                openCoursesExist = true;
            }
        }
   System.out.println("Total enrolled students across all courses: " + totalEnrolled);
   System.out.println("Open courses available: " + (openCoursesExist ? "Yes" : "No"));
    }

    public static void generateReports(List<Course> listOfCourses) {
        System.out.println("\n---------Enrollment Report---------");
        // Most popular course
        Course mostPopular = null;
        int maxEnrolled = 0;

        for (Course course : listOfCourses) {
            int enrolled = course.getEnrolledStudents() != null ? course.getEnrolledStudents().size() : 0;
            if (enrolled > maxEnrolled) {
                mostPopular = course;
                maxEnrolled = enrolled;
            }
        }
        if (mostPopular != null) {
            System.out.printf("\nMost popular course: %s (%s) - Enrolled: %d.", mostPopular.getCourseName(),
                    mostPopular.getCourseCode(), maxEnrolled);
        } else {
            System.out.println("No enrollments found.");
        }
    }
}
```

Data Manager Class

```
package schoolcourseenrolmentsystem;

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

public class dataManager {

   private static dataManager instance = null;

    // Main data lists
    private List<Instructor> instructors;
    private List<Administrator> administrators;
    private List<Student> students;
    private List<Course> courses;
    private List<Assessment> grades;

    // File paths
```

```
private static final String STUDENTS_FILE = “students.txt”;
private static final String INSTRUCTORS_FILE = “instructors.txt”;
private static final String ADMINISTRATORS_FILE = “administrators.txt”;
private static final String COURSES_FILE = “courses.txt”;
private static final String ASSESSMENTS_FILE = “assessments.txt”;
private static final String DELIMITER = “\\|”;

// Private constructor to prevent instantiation
dataManager() {
    instructors = new ArrayList<>();
    administrators = new ArrayList<>();
    students = new ArrayList<>();
    courses = new ArrayList<>();
    grades = new ArrayList<>();
    System.out.println(“Testing Constructor”);
}

public static synchronized dataManager getInstance() {
    if (instance == null) {
        instance = new dataManager();
    }
    return instance;
}

// Load all data from files
void loadAllData() {
    loadStudents();
    loadInstructors();
    loadAdministrators();
    loadCourses();
    loadAssessments();
}

// Save all data to files
public void saveAllData() {
    saveStudents();
    saveInstructors();
```

```
        saveAdministrators();
        saveCourses();
        saveAssessments();
    }

    // Load Students from file
    private void loadStudents() {
        try (BufferedReader reader = new BufferedReader(new FileReader(STUDENTS_FILE))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Loading student: " + line);
                String[] parts = line.split(DELIMITER);
                if (parts.length >= 7) {
                    String name = parts[0];
                    String id = parts[1];
                    String password = parts[2];
                    String email = parts[3];
                    String phoneNumber = parts[4];
                    String address = parts[5];
                    int creditLimit = Integer.parseInt(parts[6]);

                    List<Course> enrolledCourses = new ArrayList<>();

                    if (parts.length > 7 && !parts[7].isEmpty()) {
                        String[] courseCodes = parts[7].split(",");
                        for (String code : courseCodes) {
                            Course course = getCourseByCode(code.trim());
                            if (course != null) {
                                enrolledCourses.add(course);
                            }
                        }
                    }
                    Student student = new Student(name, id, password, email, phoneNumber, address,
creditLimit,enrolledCourses);
                    System.out.println(student);
                    if (isStudentIdUnique(id)) {
                        students.add(student);
                }
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println(“Students file not found. Starting with empty student list.”);
    } catch (IOException e) {
        System.out.println(“Error reading students file: “ + e.getMessage());
    } catch (NumberFormatException e) {
        System.out.println(“Error parsing student data: “ + e.getMessage());
    }
}
// Save Students to file
void saveStudents() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(STUDENTS_FILE))) {
        for (Student student : students) {
            StringBuilder courseCodes = new StringBuilder();
            for (Course course : student.getEnrolledCoursesList()) {
                if (courseCodes.length() > 0) {
                    courseCodes.append(“,”);
                }
                courseCodes.append(course.getCourseCode());
            }
            String line = String.join(“|”,
                    student.getName(),
                    student.getId(),
                    student.getPassword(),
                    student.getEmail(),
                    student.getPhoneNumber(),
                    student.getAddress(),
                    String.valueOf(student.getCreditLimit()),
                    courseCodes.toString());
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        System.out.println(“Error writing students file: “ + e.getMessage());
    }
```

```
}

// Load Instructors from file
private void loadInstructors() {
    try (BufferedReader reader = new BufferedReader(new FileReader(INSTRUCTORS_FILE))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(DELIMITER);
            if (parts.length >= 6) {
                String name = parts[0];
                String id = parts[1];
                String password = parts[2];
                String email = parts[3];
                String phoneNumber = parts[4];
                String address = parts[5];
                Instructor instructor = new Instructor(name, id, password, email, phoneNumber, address, new
ArrayList<>());
                if (isInstructorIdUnique(id)) {
                    instructors.add(instructor);
                }
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Instructors file not found. Starting with empty instructor list.");
    } catch (IOException e) {
        System.out.println("Error reading instructors file: " + e.getMessage());
    }
}
// Save Instructors to file
void saveInstructors() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(INSTRUCTORS_FILE))) {
        for (Instructor instructor : instructors) {
            String line = String.join("|",
                    instructor.getName(),
                    instructor.getId(),
                    instructor.getPassword(),
                    instructor.getEmail(),
                instructor.getPhoneNumber(),
                instructor.getAddress());
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        System.out.println("Error writing instructors file: " + e.getMessage());
    }
}
// Load Administrators from file
private void loadAdministrators() {
    try (BufferedReader reader = new BufferedReader(new FileReader(ADMINISTRATORS_FILE))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(DELIMITER);
            if (parts.length >= 6) {
                String name = parts[0];
                String id = parts[1];
                String password = parts[2];
                String email = parts[3];
                String phoneNumber = parts[4];
                String address = parts[5];
                Administrator admin = new Administrator(name, id, password, email, phoneNumber, address);
                if (isAdminIdUnique(id)) {
                    administrators.add(admin);
                }
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Administrators file not found. Starting with default admin.");
    } catch (IOException e) {
        System.out.println("Error reading administrators file: " + e.getMessage());
    }
}
// Save Administrators to file
private void saveAdministrators() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(ADMINISTRATORS_FILE))) {
        for (Administrator admin : administrators) {
            String line = String.join("|",
                    admin.getName(),
                    admin.getId(),
                    admin.getPassword(),
                    admin.getEmail(),
                    admin.getPhoneNumber(),
                    admin.getAddress());
                        writer.write(line);
                    writer.newLine();
        }
    } catch (IOException e) {
        System.out.println("Error writing administrators file: " + e.getMessage());
    }
}
// Load Courses from file
private void loadCourses() {
    try (BufferedReader reader = new BufferedReader(new FileReader(COURSES_FILE))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split(DELIMITER);
            if (parts.length >= 8) {
                String courseName = parts[0];
                String courseCode = parts[1];
                String schedule = parts[2].isEmpty() ? null : parts[2];
                String description = parts[3].isEmpty() ? null : parts[3];
                Course.EnrollmentStatusEnum status = Course.EnrollmentStatusEnum.valueOf(parts[4]);
                Instructor instructor = parts[5].isEmpty() ? null : getInstructorById(parts[5]);
                int capacity = Integer.parseInt(parts[6]);
                int creditHours = Integer.parseInt(parts[7]);
                List<Student> enrolledStudents = new ArrayList<>();
                if (parts.length > 8 && !parts[8].isEmpty()) {
                    String[] studentIds = parts[8].split(",");
                    for (String id : studentIds) {
                        Student student = getStudentById(id.trim());
                        if (student != null) {
                            enrolledStudents.add(student);
                        }
                    }
                }
                Course course = new Course(courseName, courseCode, schedule, description, status, instructor,
capacity, new ArrayList<>(), creditHours);
                course.setEnrolledStudents(enrolledStudents);
                if (isCourseCodeUnique(courseCode)) {
                    courses.add(course);
                }
            }
        }
    } catch (FileNotFoundException e) {
    System.out.println("Courses file not found. Starting with empty course list.");
    } catch (IOException e) {
        System.out.println("Error reading courses file: " + e.getMessage());
    } catch (IllegalArgumentException e) {
        System.out.println("Error parsing course data: " + e.getMessage());
    }
}
// Save Courses to file
void saveCourses() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(COURSES_FILE))) {
        for (Course course : courses) {
            StringBuilder studentIds = new StringBuilder();
            for (Student student : course.getEnrolledStudents()) {
                if (studentIds.length() > 0) {
                    studentIds.append(",");
                }
                studentIds.append(student.getId());
            }
            String instructorId = course.getInstructor() != null ? course.getInstructor().getId() : "";
        String schedule = course.getSchedule() != null ? course.getSchedule() : "";
            String description = course.getDescription() != null ? course.getDescription() : "";
            String line = String.join("|",
                    course.getCourseName(),
                    course.getCourseCode(),
                    schedule,
                    description,
                    course.getEnrollmentStatus().toString(),
                    instructorId,
                    String.valueOf(course.getCapacity()),
                    String.valueOf(course.getCreditHours()),
                    studentIds.toString());
                writer.write(line);
                writer.newLine();
            }
        } catch (IOException e) {
            System.out.println("Error writing courses file: " + e.getMessage());
        }
    }
    // Load Assessments from file
    private void loadAssessments() {
        try (BufferedReader reader = new BufferedReader(new FileReader(ASSESSMENTS_FILE))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(DELIMITER);
                if (parts.length >= 5) {
                    String studentId = parts[0];
                    String courseCode = parts[1];
                    Assessment.ExamType examType = Assessment.ExamType.valueOf(parts[2]);
                    double score = Double.parseDouble(parts[3]);
                    String assessmentName = parts[4];
                    Assessment assessment = new Assessment(studentId, courseCode, examType, score,
assessmentName);
                    grades.add(assessment);
                }
            }
        } catch (FileNotFoundException e) {
            System.out.println("Assessments file not found. Starting with empty assessment list.");
        } catch (IOException e) {
            System.out.println("Error reading assessments file: " + e.getMessage());
        } catch (IllegalArgumentException e) {
            System.out.println("Error parsing assessment data: " + e.getMessage());
        }
```

```
}
// Save Assessments to file
private void saveAssessments() {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(ASSESSMENTS_FILE))) {
        for (Assessment assessment : grades) {
            String line = String.join("|",
                    assessment.getStudentId(),
                    assessment.getCourseCode(),
                    assessment.getExamType().toString(),
                    String.valueOf(assessment.getScore()),
                    assessment.getAssessmentName());
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        System.out.println("Error writing assessments file: " + e.getMessage());
    }
}
// Methods for Students
public void addStudent(Student student) {
    if (student == null) {
        System.out.println("Error: Student cannot be null.");
        return;
    }
    if (isStudentIdUnique(student.getId())) {
        students.add(student);
        saveStudents();
        System.out.printf("Student added: %s with ID: %s%n", student.getName(), student.getId());
    } else {
  System.out.println("Error: Student ID " + student.getId() + " already exists.");
    }
}

public boolean removeStudent(String studentId) {
    Iterator<Student> iterator = students.iterator();
    while (iterator.hasNext()) {
        Student student = iterator.next();
        if (student.getId().equals(studentId)) {
            iterator.remove();
            saveStudents();
            System.out.printf("Student %s with ID: %s removed successfully.%n", student.getName(), studentId);
            return true;
        }
    }
    System.out.println("Student with ID: " + studentId + " not found.");
    return false;
}

public Student getStudentById(String studentId) {
    for (Student student : students) {
        if (student.getId().equals(studentId)) {
            return student;
        }
    }
    return null;
}

public List<Student> getAllStudents() {
    return students;
}

// Methods for Instructors
public void addInstructor(Instructor instructor) {
    if (instructor == null) {
        System.out.println("Error: Instructor cannot be null.");
        return;
    }
    if (isInstructorIdUnique(instructor.getId())) {
        instructors.add(instructor);
        saveInstructors();
        System.out.printf("Instructor added: %s with ID: %s%n", instructor.getName(), instructor.getId());
    } else {
        System.out.println("Error: Instructor ID " + instructor.getId() + " already exists.");
    }
```

```
}

public boolean removeInstructor(String instructorId) {
    Iterator<Instructor> iterator = instructors.iterator();
    while (iterator.hasNext()) {
        Instructor instructor = iterator.next();
        if (instructor.getId().equals(instructorId)) {
            for (Course course : courses) {
                if (course.getInstructor() != null && course.getInstructor().getId().equals(instructorId)) {
                    course.setInstructor(null);
                }
            }
            iterator.remove();
            saveInstructors();
            saveCourses();
            System.out.printf("Instructor %s with ID: %s removed successfully.%n", instructor.getName(),instructorId);
            return true;
        }
    }
    System.out.println("Instructor with ID: " + instructorId + " not found.");
    return false;
}

public Instructor getInstructorById(String instructorId) {
    for (Instructor instructor : instructors) {
        if (instructor.getId().equals(instructorId)) {
            return instructor;
        }
    }
    return null;
}

public List<Instructor> getAllInstructors() {
    return instructors;
}

// Methods for Administrators
```

```
  public void addAdministrator(Administrator admin) {
      if (admin == null) {
          System.out.println("Error: Administrator cannot be null.");
          return;
      }
      if (isAdminIdUnique(admin.getId())) {
          administrators.add(admin);
          saveAdministrators();
          System.out.printf("Administrator added: %s with ID: %s%n", admin.getName(), admin.getId());
      } else {
System.out.println("Error: Administrator ID " + admin.getId() + " already exists.");
      }
  }

  public Administrator getAdminById(String adminId) {
      for (Administrator admin : administrators) {
          if (admin.getId().equals(adminId)) {
              return admin;
          }
      }
      return null;
  }

  public List<Administrator> getAllAdministrators() {
      return administrators;
  }

  // Methods for Courses
  public void addCourse(Course course) {
      if (course == null) {
          System.out.println("Error: Course cannot be null.");
          return;
      }
      if (isCourseCodeUnique(course.getCourseCode())) {
          courses.add(course);
          saveCourses();
```

```
            System.out.printf("Course added: %s (%s) successfully.%n", course.getCourseName(),
course.getCourseCode());
        } else {
            System.out.println("Error: Course code " + course.getCourseCode() + " already exists.");
        }
    }

    public boolean removeCourse(String courseCode) {
        Iterator<Course> iterator = courses.iterator();
        while (iterator.hasNext()) {
            Course course = iterator.next();
            if (course.getCourseCode().equals(courseCode)) {
                for (Student student : course.getEnrolledStudents()) {
                    student.getEnrolledCoursesList().remove(course);
                }
                iterator.remove();
                saveCourses();
                saveStudents();
                System.out.printf("Course %s (%s) removed successfully.%n", course.getCourseName(), courseCode);
                return true;
            }
        }
        System.out.println("Course with code: " + courseCode + " not found.");
        return false;
    }

    public Course getCourseByCode(String courseCode) {
        for (Course course : courses) {
            if (course.getCourseCode().equals(courseCode)) {
                return course;
            }
        }
        return null;
    }

    public List<Course> getAllCourses() {
        return courses;
    }
    // ======================Methods for Assessments======================
    public void addAssessment(Assessment assessment) {
        if (assessment == null) {
            System.out.println("Error: Assessment cannot be null.");
            return;
        }
        grades.add(assessment);
        saveAssessments();
        System.out.println("Assessment recorded successfully.");
    }

    public void updateAssessment(String studentId, String courseCode, Assessment.ExamType examType, double
score) {
        for (Assessment assessment : grades) {
            if (assessment.getStudentId().equals(studentId) &&
                    assessment.getCourseCode().equals(courseCode) &&
                    assessment.getExamType() == examType) {
                assessment.setScore(score);
                saveAssessments();
                System.out.println("Assessment updated successfully.");
                return;
            }
        }
        Assessment newAssessment = new Assessment(studentId, courseCode, examType, score, examType.name());
        grades.add(newAssessment);
        saveAssessments();
        System.out.println("Assessment recorded successfully.");
    }

    public List<Assessment> getAssessmentsByStudentAndCourse(String studentId, String courseCode) {
        List<Assessment> studentAssessments = new ArrayList<>();
        for (Assessment assessment : grades) {
            if (assessment.getStudentId().equals(studentId) && assessment.getCourseCode().equals(courseCode)) {
                studentAssessments.add(assessment);
            }
        }
        return studentAssessments;
    }

    public List<Assessment> getAllAssessments() {
        return grades;
    }

    // Validation methods
    private boolean isStudentIdUnique(String studentId) {
        return students.stream().noneMatch(student -> student.getId().equals(studentId));
    }

    private boolean isInstructorIdUnique(String instructorId) {
        return instructors.stream().noneMatch(instructor -> instructor.getId().equals(instructorId));
    }
    private boolean isAdminIdUnique(String adminId) {
        return administrators.stream().noneMatch(admin -> admin.getId().equals(adminId));
    }

    private boolean isCourseCodeUnique(String courseCode) {
        return courses.stream().noneMatch(course -> course.getCourseCode().equals(courseCode));
    }

    //====================== Utility methods======================
    public void assignInstructorToCourse(String courseCode, String instructorId) {
        Course course = getCourseByCode(courseCode);
        Instructor instructor = getInstructorById(instructorId);

        if (course == null) {
          System.out.println("Error: Course with code " + courseCode + " not found.");
            return;
        }
        if (instructor == null) {
      System.out.println("Error: Instructor with ID " + instructorId + " not found.");
            return;
        }
```

```
        if (course.getInstructor() == null) {
            course.setInstructor(instructor);
            saveCourses();
            System.out.printf("Instructor %s assigned to %s.%n", instructor.getName(), course.getCourseName());
        } else if (course.getInstructor().equals(instructor)) {
            System.out.printf("Instructor %s is already assigned to %s.%n", instructor.getName(),
course.getCourseName());
        } else {
            System.out.println("Course already has a different instructor assigned.");
        }
    }

    public void enrollStudentInCourse(String studentId, String courseCode) {
        Student student = getStudentById(studentId);
        Course course = getCourseByCode(courseCode);

        if (student == null) {
            System.out.println("Error: Student with ID " + studentId + " not found.");
            return;
        }
        if (course == null) {
         System.out.println("Error: Course with code " + courseCode + " not found.");
            return;
        }
        student.enroll_In_Course(course);
        if (course.getEnrolledStudents().contains(student)) {
            saveStudents();
            saveCourses();
        }
    }

    public void dropStudentFromCourse(String studentId, String courseCode) {
        Student student = getStudentById(studentId);
        Course course = getCourseByCode(courseCode);

        if (student == null) {
            System.out.println("Error: Student with ID " + studentId + " not found.");
            return;
        }
        if (course == null) {
          System.out.println("Error: Course with code " + courseCode + " not found.");
            return;
        }

        student.dropCourse(course);
        if (!course.getEnrolledStudents().contains(student)) {
            saveStudents();
            saveCourses();
        }
    }

    public void closeCourse(String courseCode) {
        Course course = getCourseByCode(courseCode);
        if (course == null) {
          System.out.println("Error: Course with code " + courseCode + " not found.");
            return;
        }

        if (course.getEnrolledStudents().size() >= course.getCapacity()) {
            System.out.println(course.getCourseName() + " is at full capacity.");
            course.setEnrollmentStatus(Course.EnrollmentStatusEnum.Closed);
        } else if (course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Closed) {
System.out.println(course.getCourseName() + " enrollment status is already closed.");
        } else {
        System.out.println(course.getCourseName() + " still has available capacity.");
            course.setEnrollmentStatus(Course.EnrollmentStatusEnum.Closed);
        }
        saveCourses();
    }
    public void viewEnrollmentStatistics() {
        int totalEnrolled = 0;
        boolean openCoursesExist = false;

        System.out.println("\n---------Enrollment Statistics---------");
        for (Course course : courses) {
            int count = course.getEnrolledStudents() != null ? course.getEnrolledStudents().size() : 0;
            totalEnrolled += count;
            System.out.printf("Course: %s (%s) - Enrolled: %d%n", course.getCourseName(), course.getCourseCode(),
count);
            if (course.getEnrollmentStatus() == Course.EnrollmentStatusEnum.Open) {
                openCoursesExist = true;
            }
        }
        System.out.println("Total enrolled students across all courses: " + totalEnrolled);
   System.out.println("Open courses available: " + (openCoursesExist ? "Yes" : "No"));
    }

    public void generateReports() {
        System.out.println("\n---------Enrollment Report---------");

        Course mostPopular = null;
        int maxEnrolled = 0;

        for (Course course : courses) {
            int enrolled = course.getEnrolledStudents() != null ? course.getEnrolledStudents().size() : 0;
            if (enrolled > maxEnrolled) {
                mostPopular = course;
                maxEnrolled = enrolled;
            }
        }
        if (mostPopular != null) {
            System.out.printf("Most popular course: %s (%s) - Enrolled: %d.%n", mostPopular.getCourseName(),
mostPopular.getCourseCode(), maxEnrolled);
        } else {
            System.out.println("No enrollments found.");
        }
    }
}
```