**Student Information System (SIS)**

**Name**: Kannan G  
**Superset ID**: 5372153

**Task 1. Database Design:**

1. Create the database named "SISDB"

A black screen with white text

Description automatically generated

2. Define the schema for the Students, Courses, Enrollments, Teacher, and Payments tables based on the provided schema. Write SQL scripts to create the mentioned tables with appropriate data types, constraints, and relationships.

a. Students

b. Courses

c. Enrollments

d. Teacher

e. Payments

A computer screen shot of a black screen

Description automatically generated

A computer screen with white text

Description automatically generated

A computer screen shot of white text

Description automatically generated

A computer screen shot of a black and white screen

Description automatically generated

A computer screen with white text

Description automatically generated

3. Create an ERD (Entity Relationship Diagram) for the database.

A diagram of a diagram

Description automatically generated

4. Create appropriate Primary Key and Foreign Key constraints for referential integrity.

5. Insert at least 10 sample records into each of the following tables.

i. Students

ii. Courses

iii. Enrollments

iv. Teacher

v. Payments

A computer screen with white text

Description automatically generated

A computer screen shot of white text

Description automatically generated

A computer screen with white text

Description automatically generated

A screen shot of a computer

Description automatically generated

A computer screen with white text

Description automatically generated

**Tasks 2: Select, Where, Between, AND, LIKE:**

1. Write an SQL query to insert a new student into the "Students" table with the following details:

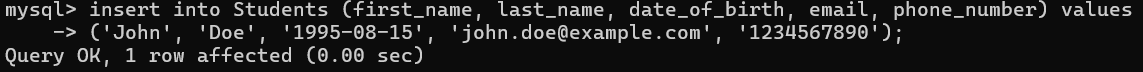
a. First Name: John

b. Last Name: Doe

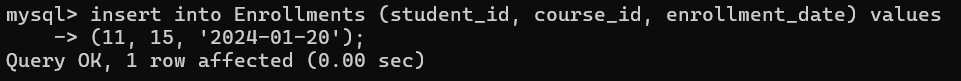
c. Date of Birth: 1995-08-15

d. Email: john.doe@example.com

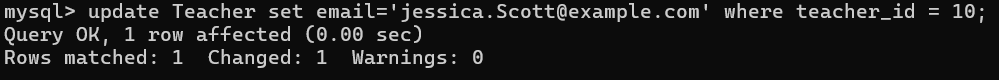
e. Phone Number: 1234567890



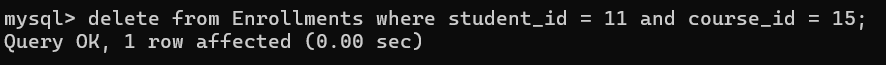
2. Write an SQL query to enroll a student in a course. Choose an existing student and course and insert a record into the "Enrollments" table with the enrollment date.



3. Update the email address of a specific teacher in the "Teacher" table. Choose any teacher and modify their email address.

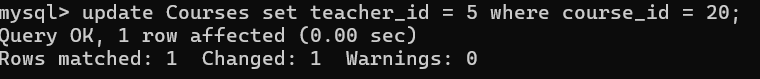


4. Write an SQL query to delete a specific enrollment record from the "Enrollments" table. Select an enrollment record based on the student and course.



5. Update the "Courses" table to assign a specific teacher to a course. Choose any course and

teacher from the respective tables.

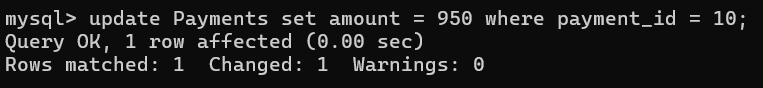


6. Delete a specific student from the "Students" table and remove all their enrollment records

from the "Enrollments" table. Be sure to maintain referential integrity.



7. Update the payment amount for a specific payment record in the "Payments" table. Choose any payment record and modify the payment amount.



**Task 3. Aggregate functions, Having, Order By, GroupBy and Joins:**

1. Write an SQL query to calculate the total payments made by a specific student. You will need to join the "Payments" table with the "Students" table based on the student's ID.

A screen shot of a computer code

Description automatically generated

2. Write an SQL query to retrieve a list of courses along with the count of students enrolled in each course. Use a JOIN operation between the "Courses" table and the "Enrollments" table.

A computer screen with white text

Description automatically generated

3. Write an SQL query to find the names of students who have not enrolled in any course. Use a LEFT JOIN between the "Students" table and the "Enrollments" table to identify students

without enrollments.

A screenshot of a computer

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

4. Write an SQL query to retrieve the first name, last name of students, and the names of the

courses they are enrolled in. Use JOIN operations between the "Students" table and the

"Enrollments" and "Courses" tables.

A computer screen with white text

Description automatically generated

5. Create a query to list the names of teachers and the courses they are assigned to. Join the

"Teacher" table with the "Courses" table.

A computer screen shot of a black screen

Description automatically generated

6. Retrieve a list of students and their enrollment dates for a specific course. You'll need to join the "Students" table with the "Enrollments" and "Courses" tables.

A computer screen shot of a program

Description automatically generated

7. Find the names of students who have not made any payments. Use a LEFT JOIN between the "Students" table and the "Payments" table and filter for students with NULL payment records.

A screen shot of a computer code

Description automatically generated

8. Write a query to identify courses that have no enrollments. You'll need to use a LEFT JOIN

between the "Courses" table and the "Enrollments" table and filter for courses with NULL

enrollment records.

A screen shot of a computer

Description automatically generated

9. Identify students who are enrolled in more than one course. Use a self-join on the "Enrollments" table to find students with multiple enrollment records.

A computer screen shot of a black screen

Description automatically generated

10. Find teachers who are not assigned to any courses. Use a LEFT JOIN between the "Teacher" table and the "Courses" table and filter for teachers with NULL course assignments.

A computer screen shot of a black screen

Description automatically generated

**Task 4. Subquery and its type:**

1. Write an SQL query to calculate the average number of students enrolled in each course. Use aggregate functions and subqueries to achieve this.

A computer screen with white text

Description automatically generated

2. Identify the student(s) who made the highest payment. Use a subquery to find the maximum payment amount and then retrieve the student(s) associated with that amount.

A computer screen with white text

Description automatically generated

3. Retrieve a list of courses with the highest number of enrollments. Use subqueries to find the course(s) with the maximum enrollment count.

A computer screen with white text

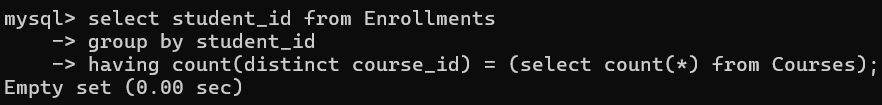
Description automatically generated

4. Calculate the total payments made to courses taught by each teacher. Use subqueries to sum payments for each teacher's courses.

A screenshot of a computer

Description automatically generated

5. Identify students who are enrolled in all available courses. Use subqueries to compare a student's enrollments with the total number of courses.



6. Retrieve the names of teachers who have not been assigned to any courses. Use subqueries to find teachers with no course assignments.

A screen shot of a computer

Description automatically generated

7. Calculate the average age of all students. Use subqueries to calculate the age of each student based on their date of birth.

A screen shot of a computer code

Description automatically generated

8. Identify courses with no enrollments. Use subqueries to find courses without enrollment records.

A screen shot of a computer

Description automatically generated

9. Calculate the total payments made by each student for each course they are enrolled in. Use subqueries and aggregate functions to sum payments.

A screen shot of a computer

Description automatically generated

10. Identify students who have made more than one payment. Use subqueries and aggregate

functions to count payments per student and filter for those with counts greater than one.

A computer screen with white text

Description automatically generated

11. Write an SQL query to calculate the total payments made by each student. Join the "Students" table with the "Payments" table and use GROUP BY to calculate the sum of payments for each student.

A screenshot of a computer screen

Description automatically generated

12. Retrieve a list of course names along with the count of students enrolled in each course. Use JOIN operations between the "Courses" table and the "Enrollments" table and GROUP BY to count enrollments.

A computer screen with white text

Description automatically generated

13. Calculate the average payment amount made by students. Use JOIN operations between the "Students" table and the "Payments" table and GROUP BY to calculate the average.

A screen shot of a computer

Description automatically generated

**Implement OOPs**

**Task 1: Define Classes**

Define the following classes based on the domain description:

Student class with the following attributes:

• Student ID

• First Name

• Last Name

• Date of Birth

• Email

• Phone Number

Course class with the following attributes:

• Course ID

• Course Name

• Course Code

• Instructor Name

Enrollment class to represent the relationship between students and courses. It should have attributes:

• Enrollment ID

• Student ID (reference to a Student)

• Course ID (reference to a Course)

• Enrollment Date

Teacher class with the following attributes:

• Teacher ID

• First Name

• Last Name

• Email

Payment class with the following attributes:

• Payment ID

• Student ID (reference to a Student)

• Amount

• Payment Date

**Task 2: Implement Constructors**

Implement constructors for each class to initialize their attributes. Constructors are special methods that are called when an object of a class is created. They are used to set initial values for the attributes of the class. Below are detailed instructions on how to implement constructors for each class in your Student Information System (SIS) assignment:

Student Class Constructor

In the Student class, you need to create a constructor that initializes the attributes of a student when an instance of the Student class is created

SIS Class Constructor

If you have a class that represents the Student Information System itself (e.g., SIS class), you may also implement a constructor for it. This constructor can be used to set up any initial configuration for the SIS.

Repeat the above process for each class Course, Enrollment, Teacher, Payment by defining constructors that initialize their respective attributes.

**Task 3: Implement Methods**

Implement methods in your classes to perform various operations related to the Student Information

System (SIS). These methods will allow you to interact with and manipulate data within your system.

Below are detailed instructions on how to implement methods in each class:

Implement the following methods in the appropriate classes:

Student Class:

• EnrollInCourse(course: Course): Enrolls the student in a course.

• UpdateStudentInfo(firstName: string, lastName: string, dateOfBirth: DateTime, email: string,

phoneNumber: string): Updates the student's information.

• MakePayment(amount: decimal, paymentDate: DateTime): Records a payment made by the

student.

• DisplayStudentInfo(): Displays detailed information about the student.

• GetEnrolledCourses(): Retrieves a list of courses in which the student is enrolled.

• GetPaymentHistory(): Retrieves a list of payment records for the student.

Course Class:

• AssignTeacher(teacher: Teacher): Assigns a teacher to the course.

• UpdateCourseInfo(courseCode: string, courseName: string, instructor: string): Updates course information.

• DisplayCourseInfo(): Displays detailed information about the course.

• GetEnrollments(): Retrieves a list of student enrollments for the course.

• GetTeacher(): Retrieves the assigned teacher for the course.

Enrollment Class:

• GetStudent(): Retrieves the student associated with the enrollment.

• GetCourse(): Retrieves the course associated with the enrollment.

Teacher Class:

• UpdateTeacherInfo(name: string, email: string, expertise: string): Updates teacher information.

• DisplayTeacherInfo(): Displays detailed information about the teacher.

• GetAssignedCourses(): Retrieves a list of courses assigned to the teacher.

Payment Class:

• GetStudent(): Retrieves the student associated with the payment.

• GetPaymentAmount(): Retrieves the payment amount.

• GetPaymentDate(): Retrieves the payment date.

SIS Class (if you have one to manage interactions):

• EnrollStudentInCourse(student: Student, course: Course): Enrolls a student in a course.

• AssignTeacherToCourse(teacher: Teacher, course: Course): Assigns a teacher to a course.

• RecordPayment(student: Student, amount: decimal, paymentDate: DateTime): Records a

payment made by a student.

• GenerateEnrollmentReport(course: Course): Generates a report of students enrolled in a

specific course.

• GeneratePaymentReport(student: Student): Generates a report of payments made by a specific student.

• CalculateCourseStatistics(course: Course): Calculates statistics for a specific course, such as the number of enrollments and total payments.

**Student.java:**

package entity;

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

public class Student {

private int studentId;

private String firstName;

private String lastName;

private LocalDate dateOfBirth;

private String email;

private String phoneNumber;

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

private List<Payment> payments = new ArrayList<>();

public Student() {

super();

// TODO Auto-generated constructor stub

}

public Student(int studentId, String firstName, String lastName, LocalDate dateOfBirth, String email,String phoneNumber) {

super();

this.studentId = studentId;

this.firstName = firstName;

this.lastName = lastName;

this.dateOfBirth = dateOfBirth;

this.email = email;

this.phoneNumber = phoneNumber;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public LocalDate getDateOfBirth() {

return dateOfBirth;

}

public void setDateOfBirth(LocalDate dateOfBirth) {

this.dateOfBirth = dateOfBirth;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getPhoneNumber() {

return phoneNumber;

}

public void setPhoneNumber(String phoneNumber) {

this.phoneNumber = phoneNumber;

}

public void setStudentId(int studentId) {

this.studentId = studentId;

}

public void enrollInCourse(Course course) {

if (!enrolledCourses.contains(course)) {

enrolledCourses.add(course);

System.out.println("Student " + studentId + " enrolled in course " + course.getCourseName());

} else {

System.out.println("Student already enrolled in this course.");

}

}

public void updateStudentInfo(String firstName, String lastName, LocalDate dateOfBirth, String email, String phoneNumber) {

this.firstName = firstName;

this.lastName = lastName;

this.dateOfBirth = dateOfBirth;

this.email = email;

this.phoneNumber = phoneNumber;

}

public void makePayment(double amount, LocalDate paymentDate) {

Payment payment = new Payment(new Random().nextInt(10000),this, amount, paymentDate);

payments.add(payment);

System.out.println("Payment of Rs." + amount + " recorded.");

}

public void displayStudentInfo() {

System.out.println("ID: " + studentId + ", Name: " + firstName + " " + lastName + ", DOB: " + dateOfBirth + ", Email: " + email + ", Phone: " + phoneNumber);

}

public List<Course> getEnrolledCourses() {

return enrolledCourses;

}

public List<Payment> getPaymentHistory() {

return payments;

}

public int getStudentId() {

return studentId;

}

}

**Course.java:**

package entity;

import java.util.ArrayList;

import java.util.List;

public class Course {

private int courseId;

private String courseName;

private String courseCode;

private String instructorName;

private Teacher teacher;

private List<Enrollment> enrollments = new ArrayList<>();

public Course() {

super();

// TODO Auto-generated constructor stub

}

public String getCourseCode() {

return courseCode;

}

public void setCourseCode(String courseCode) {

this.courseCode = courseCode;

}

public String getInstructorName() {

return instructorName;

}

public void setInstructorName(String instructorName) {

this.instructorName = instructorName;

}

public void setCourseId(int courseId) {

this.courseId = courseId;

}

public void setCourseName(String courseName) {

this.courseName = courseName;

}

public Course(int courseId, String courseName, String courseCode, String instructorName) {

super();

this.courseId = courseId;

this.courseName = courseName;

this.courseCode = courseCode;

this.instructorName = instructorName;

}

public void assignTeacher(Teacher teacher) {

this.teacher = teacher;

}

public void updateCourseInfo(String courseCode, String courseName, String instructorName) {

this.courseCode = courseCode;

this.courseName = courseName;

this.instructorName = instructorName;

}

public void displayCourseInfo() {

System.out.println("Course: " + courseName + ", Code: " + courseCode + ", Instructor: " + instructorName);

}

public void addEnrollment(Enrollment enrollment) {

if (!enrollments.contains(enrollment)) {

enrollments.add(enrollment);

}

}

public List<Enrollment> getEnrollments() {

return enrollments;

}

public Teacher getTeacher() {

return teacher;

}

public String getCourseName() {

return courseName;

}

public int getCourseId() {

return courseId;

}

}

**Enrollment.java:**

**package** entity;

**import** java.time.LocalDate;

**public** **class** Enrollment {

**private** **int** enrollmentId;

**private** Student student;

**private** Course course;

**private** LocalDate enrollmentDate;

**public** Enrollment() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Enrollment(**int** enrollmentId, Student student, Course course, LocalDate enrollmentDate) {

**super**();

**this**.enrollmentId = enrollmentId;

**this**.student = student;

**this**.course = course;

**this**.enrollmentDate = enrollmentDate;

}

**public** Student getStudent() {

**return** student;

}

**public** Course getCourse() {

**return** course;

}

**public** **int** getEnrollmentId() {

**return** enrollmentId;

}

**public** **void** setEnrollmentId(**int** enrollmentId) {

**this**.enrollmentId = enrollmentId;

}

**public** LocalDate getEnrollmentDate() {

**return** enrollmentDate;

}

**public** **void** setEnrollmentDate(LocalDate enrollmentDate) {

**this**.enrollmentDate = enrollmentDate;

}

**public** **void** setStudent(Student student) {

**this**.student = student;

}

**public** **void** setCourse(Course course) {

**this**.course = course;

}

}

**Teacher.java:**

package entity;

import java.util.ArrayList;

import java.util.List;

public class Teacher {

private int teacherId;

private String firstName;

private String lastName;

private String email;

private String expertise;

private List<Course> assignedCourses = new ArrayList<>();

public Teacher() {

super();

// TODO Auto-generated constructor stub

}

public Teacher(int teacherId, String firstName, String lastName, String email) {

super();

this.teacherId = teacherId;

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getExpertise() {

return expertise;

}

public void setExpertise(String expertise) {

this.expertise = expertise;

}

public void setTeacherId(int teacherId) {

this.teacherId = teacherId;

}

public void updateTeacherInfo(String name, String email, String expertise) {

this.firstName = name;

this.email = email;

this.expertise = expertise;

}

public void displayTeacherInfo() {

System.out.println("ID: " + teacherId + "Teacher: " + firstName + " " + lastName + ", Email: " + email + ", Expertise: " + expertise);

}

public void addAssignedCourse(Course course) {

assignedCourses.add(course);

}

public List<Course> getAssignedCourses() {

return assignedCourses;

}

public int getTeacherId() {

return teacherId;

}

}

**Payment.java:**

**package** entity;

**import** java.time.LocalDate;

**public** **class** Payment {

**private** **int** paymentId;

**private** Student student;

**private** **double** amount;

**private** LocalDate paymentDate;

**public** Payment() {

**super**();

// **TODO** Auto-generated constructor stub

}

**public** Payment(**int** paymentId, Student student, **double** amount, LocalDate paymentDate) {

**super**();

**this**.paymentId = paymentId;

**this**.student = student;

**this**.amount = amount;

**this**.paymentDate = paymentDate;

}

**public** **int** getPaymentId() {

**return** paymentId;

}

**public** **void** setPaymentId(**int** paymentId) {

**this**.paymentId = paymentId;

}

**public** **double** getAmount() {

**return** amount;

}

**public** **void** setAmount(**double** amount) {

**this**.amount = amount;

}

**public** **void** setStudent(Student student) {

**this**.student = student;

}

**public** **void** setPaymentDate(LocalDate paymentDate) {

**this**.paymentDate = paymentDate;

}

**public** Student getStudent() {

**return** student;

}

**public** **double** getPaymentAmount() {

**return** amount;

}

**public** LocalDate getPaymentDate() {

**return** paymentDate;

}

}

**SIS.java:**

**package** entity;

**import** java.time.LocalDate;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Random;

**public** **class** SIS {

**private** List<Student> students;

**private** List<Course> courses;

**private** List<Teacher> teachers;

**private** List<Enrollment> enrollments;

**private** List<Payment> payments;

**public** SIS() {

// **TODO** Auto-generated constructor stub

**super**();

students = **new** ArrayList<>();

courses = **new** ArrayList<>();

teachers = **new** ArrayList<>();

enrollments = **new** ArrayList<>();

payments = **new** ArrayList<>();

}

**public** **void** enrollStudentInCourse(Student student, Course course) {

student.enrollInCourse(course);

Enrollment enrollment = **new** Enrollment(**new** Random().nextInt(1000), student, course, LocalDate.*now*());

course.addEnrollment(enrollment);

enrollments.add(enrollment);

}

**public** **void** assignTeacherToCourse(Teacher teacher, Course course) {

course.assignTeacher(teacher);

teacher.addAssignedCourse(course);

System.***out***.println("Assigned " + teacher.getFirstName()+" "+teacher.getLastName() + " to course: " + course.getCourseName());

}

**public** **void** recordPayment(Student student, **double** amount, LocalDate paymentDate) {

student.makePayment(amount, paymentDate);

System.***out***.println("Payment of Rs." + amount + " recorded for student: " + student.getFirstName()+" "+ student.getLastName());

}

**public** **void** generateEnrollmentReport(Course course) {

System.***out***.println("Enrollment Report for course: " + course.getCourseName());

List<Enrollment> enrollments = course.getEnrollments();

**if** (enrollments.isEmpty()) {

System.***out***.println("No students enrolled yet.");

}

**else** {

**for** (Enrollment e : enrollments) {

e.getStudent().displayStudentInfo();;

//System.out.println("- " + student.getFirstName() + " " + student.getLastName());

}

}

}

**public** **void** generatePaymentReport(Student student) {

System.***out***.println("Payment Report for student: " + student.getFirstName());

List<Payment> payments = student.getPaymentHistory();

**if** (payments.isEmpty()) {

System.***out***.println("No payments recorded.");

}

**else** {

**for** (Payment p : payments) {

System.***out***.println("Paid Rs." + p.getPaymentAmount() + " on " + p.getPaymentDate());

}

}

}

**public** **void** calculateCourseStatistics(Course course) {

List<Enrollment> enrollments = course.getEnrollments();

**int** totalEnrolled = enrollments.size();

**double** totalPayments = 0;

**for** (Enrollment e : enrollments) {

**for** (Payment p : e.getStudent().getPaymentHistory()) {

totalPayments += p.getPaymentAmount();

}

}

System.***out***.println("Course Statistics for: " + course.getCourseName());

System.***out***.println("Total Enrollments: " + totalEnrolled);

System.***out***.println("Total Payments by Enrolled Students: Rs." + totalPayments);

}

}

**Use the Methods**

In your driver program or any part of your code where you want to perform actions related to the Student Information System, create instances of your classes, and use the methods you've

implemented.

Repeat this process for using other methods you have implemented in your classes and the SIS class.

**MainModule.java:**

package main;

import java.sql.Connection;

import java.time.LocalDate;

import java.util.List;

import dao.PaymentService;

import dao.StudentService;

import entity.Course;

import entity.Payment;

import entity.SIS;

import entity.Student;

import entity.Teacher;

import util.DBConnUtil;

import util.DBPropertyUtil;

public class MainModule {

public static void main(String[] args) {

// TODO Auto-generated method stub

Connection con = DBConnUtil.getDbConnection();

if(con != null) {

System.out.println(con);

System.out.println("Connected!");

}

else {

System.out.println("failed to connect");

}

// Create SIS object

SIS sis = new SIS();

// Create Students

Student student1 = new Student(101, "Alice", "Johnson", LocalDate.of(2002, 5, 15), "alice@email.com", "9876543210");

Student student2 = new Student(102, "Bob", "Smith", LocalDate.of(2001, 8, 20), "bob@email.com", "9123456780");

// Create Teachers

Teacher teacher1 = new Teacher(201, "Dr.", "Brown", "dr.brown@school.com");

teacher1.setExpertise("Mathematics");

// Create Courses

Course course1 = new Course(301, "Calculus", "MATH101", "Dr. Brown");

Course course2 = new Course(302, "Physics", "PHY101", "Dr. Einstein");

// Assign teacher to a course

sis.assignTeacherToCourse(teacher1, course1);

// Enroll students in course

sis.enrollStudentInCourse(student1, course1);

sis.enrollStudentInCourse(student2, course1);

sis.enrollStudentInCourse(student2, course2);

// Record payments

sis.recordPayment(student1, 1500.00, LocalDate.of(2025, 4, 10));

sis.recordPayment(student2, 1800.00, LocalDate.of(2025, 4, 10));

sis.recordPayment(student2, 1200.00, LocalDate.of(2025, 4, 11));

// Display Student Info

student1.displayStudentInfo();

student2.displayStudentInfo();

// Display Teacher Info

teacher1.displayTeacherInfo();

// Display Course Info

course1.displayCourseInfo();

course2.displayCourseInfo();

// Generate Reports

sis.generateEnrollmentReport(course1);

sis.generatePaymentReport(student2);

sis.calculateCourseStatistics(course1);

}

}

**Sample-output:**

A screenshot of a computer

Description automatically generated

**Task 4: Exceptions handling and Custom Exceptions**

Implementing custom exceptions allows you to define and throw exceptions tailored to specific

situations or business logic requirements.

**Create Custom Exception Classes**

You'll need to create custom exception classes that are inherited from the System.Exception class or one of its derived classes (e.g., System.ApplicationException). These custom exception classes will allow you to encapsulate specific error scenarios and provide meaningful error messages.

**Throw Custom Exceptions**

In your code, you can throw custom exceptions when specific conditions or business logic rules are violated. To throw a custom exception, use the throw keyword followed by an instance of your custom exception class.

• **DuplicateEnrollmentException**: Thrown when a student is already enrolled in a course and tries to enroll again. This exception can be used in the EnrollStudentInCourse method.

• **CourseNotFoundException**: Thrown when a course does not exist in the system, and you

attempt to perform operations on it (e.g., enrolling a student or assigning a teacher).

• **StudentNotFoundException**: Thrown when a student does not exist in the system, and you

attempt to perform operations on the student (e.g., enrolling in a course, making a payment).

• **TeacherNotFoundException**: Thrown when a teacher does not exist in the system, and you

attempt to assign them to a course.

• **PaymentValidationException**: Thrown when there is an issue with payment validation, such as an invalid payment amount or payment date.

• **InvalidStudentDataException**: Thrown when data provided for creating or updating a student is invalid (e.g., invalid date of birth or email format).

• **InvalidCourseDataException**: Thrown when data provided for creating or updating a course is invalid (e.g., invalid course code or instructor name).

• **InvalidEnrollmentDataException**: Thrown when data provided for creating an enrollment is invalid (e.g., missing student or course references).

• **InvalidTeacherDataException**: Thrown when data provided for creating or updating a teacher is invalid (e.g., missing name or email).

• **InsufficientFundsException**: Thrown when a student attempts to enroll in a course but does not have enough funds to make the payment.

**DuplicateEnrollmentException.java:**

**package** exception;

**public** **class** DuplicateEnrollmentException **extends** Exception{

**public** DuplicateEnrollmentException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**CourseNotFoundException.java:**

**package** exception;

**public** **class** CourseNotFoundException **extends** Exception{

**public** CourseNotFoundException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**StudentNotFoundException.java:**

**package** exception;

**public** **class** StudentNotFoundException **extends** Exception{

**public** StudentNotFoundException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**TeacherNotFoundException.java:**

**package** exception;

**public** **class** TeacherNotFoundException **extends** Exception{

**public** TeacherNotFoundException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**PaymentValidationException.java:**

**package** exception;

**public** **class** PaymentValidationException **extends** Exception{

**public** PaymentValidationException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**InvalidStudentDataException.java:**

**package** exception;

**public** **class** InvalidStudentDataException **extends** Exception{

**public** InvalidStudentDataException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**InvalidCourseDataException.java:**

**package** exception;

**public** **class** InvalidCourseDataException **extends** Exception{

**public** InvalidCourseDataException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**InvalidEnrollmentDataException.java:**

**package** exception;

**public** **class** InvalidEnrollmentDataException **extends** Exception{

**public** InvalidEnrollmentDataException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**InvalidTeacherDataException.java:**

**package** exception;

**public** **class** InvalidTeacherDataException **extends** Exception{

**public** InvalidTeacherDataException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**InsufficientFundsException.java:**

**package** exception;

**public** **class** InsufficientFundsException **extends** Exception{

**public** InsufficientFundsException(String message) {

// **TODO** Auto-generated constructor stub

**super**(message);

}

}

**Task 5: Collections**

**Implement Collections:**

Implement relationships between classes using appropriate data structures (e.g., lists or dictionaries) to maintain associations between students, courses, enrollments, teachers, and payments.

These relationships are essential for the Student Information System (SIS) to track and manage student enrollments, teacher assignments, and payments accurately.

**Define Class-Level Data Structures**

You will need class-level data structures within each class to maintain relationships. Here's how to

define them for each class:

**Student Class:**

Create a list or collection property to store the student's enrollments. This property will hold references to Enrollment objects.

Example: List<Enrollment> Enrollments { get; set; }

**Course Class:**

Create a list or collection property to store the course's enrollments. This property will hold references to Enrollment objects.

Example: List<Enrollment> Enrollments { get; set; }

**Enrollment Class:**

Include properties to hold references to both the Student and Course objects.

Example: Student Student { get; set; } and Course Course { get; set; }

**Teacher Class:**

Create a list or collection property to store the teacher's assigned courses. This property will hold

references to Course objects.

Example: List<Course> AssignedCourses { get; set; }

**Payment Class:**

Include a property to hold a reference to the Student object.

Example: Student Student { get; set; }

**Update Constructor(s)**

In the constructors of your classes, initialize the list or collection properties to create empty collections when an object is instantiated.

Repeat this for the Course, Teacher, and Payment classes, where applicable.

**Task 6: Create Methods for Managing Relationships**

To add, remove, or retrieve related objects, you should create methods within your SIS class or each relevant class.

• **AddEnrollment**(student, course, enrollmentDate): In the SIS class, create a method that adds an enrollment to both the Student's and Course's enrollment lists. Ensure the Enrollment object references the correct Student and Course.

• **AssignCourseToTeacher**(course, teacher): In the SIS class, create a method to assign a course to a teacher. Add the course to the teacher's AssignedCourses list.

• **AddPayment**(student, amount, paymentDate): In the SIS class, create a method that adds a

payment to the Student's payment history. Ensure the Payment object references the correct

Student.

• **GetEnrollmentsForStudent**(student): In the SIS class, create a method to retrieve all enrollments for a specific student.

• **GetCoursesForTeacher**(teacher): In the SIS class, create a method to retrieve all courses

assigned to a specific teacher.

**Create a Driver Program**

A driver program (also known as a test program or main program) is essential for testing and

demonstrating the functionality of your classes and methods within your Student Information System (SIS) assignment. In this task, you will create a console application that serves as the entry point for your SIS and allows you to interact with and test your implemented classes and methods.

**Add References to Your SIS Classes**

Ensure that your SIS classes (Student, Course, Enrollment, Teacher, Payment) and the SIS class (if you have one to manage interactions) are defined in separate files within your project or are referenced properly.

If you have defined these classes in separate files, make sure to include using statements in your driver program to access them:

**Implement the Main Method**

In the console application, the Main method serves as the entry point for your program. This is where you will create instances of your classes, call methods, and interact with your Student Information System.

In the Main method, you create instances of your classes (e.g., Student, Course, and SIS) and then interact with your Student Information System by calling methods and handling exceptions.

**SIS.java:**

**package** entity;

**import** java.time.LocalDate;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Random;

**import** exception.CourseNotFoundException;

**import** exception.DuplicateEnrollmentException;

**import** exception.InvalidCourseDataException;

**import** exception.InvalidEnrollmentDataException;

**import** exception.InvalidStudentDataException;

**import** exception.PaymentValidationException;

**import** exception.StudentNotFoundException;

**import** exception.TeacherNotFoundException;

**public** **class** SIS {

**private** List<Student> students;

**private** List<Course> courses;

**private** List<Teacher> teachers;

**private** List<Enrollment> enrollments;

**private** List<Payment> payments;

**public** SIS() {

// **TODO** Auto-generated constructor stub

**super**();

students = **new** ArrayList<>();

courses = **new** ArrayList<>();

teachers = **new** ArrayList<>();

enrollments = **new** ArrayList<>();

payments = **new** ArrayList<>();

}

**public** **void** enrollStudentInCourse(Student student, Course course) **throws** DuplicateEnrollmentException{

**for** (Enrollment e : enrollments) {

**if** (e.getStudent().equals(student) && e.getCourse().equals(course)) {

**throw** **new** DuplicateEnrollmentException("Student is already enrolled in this course.");

}

}

student.enrollInCourse(course);

Enrollment enrollment = **new** Enrollment(**new** Random().nextInt(1000), student, course, LocalDate.*now*());

course.addEnrollment(enrollment);

enrollments.add(enrollment);

}

**public** **void** assignTeacherToCourse(Teacher teacher, Course course) **throws** TeacherNotFoundException, CourseNotFoundException{

**if** (!teachers.contains(teacher)) {

**throw** **new** TeacherNotFoundException("Teacher not found in the system.");

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found in the system.");

}

course.assignTeacher(teacher);

teacher.addAssignedCourse(course);

System.***out***.println("Assigned " + teacher.getFirstName()+" "+teacher.getLastName() + " to course: " + course.getCourseName());

}

**public** **void** recordPayment(Student student, **double** amount, LocalDate paymentDate) **throws** StudentNotFoundException, PaymentValidationException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found in the system.");

}

**if** (amount <= 0 || paymentDate.isAfter(LocalDate.*now*())) {

**throw** **new** PaymentValidationException("Payment amount must be positive.");

}

student.makePayment(amount, paymentDate);

System.***out***.println("Payment of Rs." + amount + " recorded for student: " + student.getFirstName()+" "+ student.getLastName());

}

**public** **void** generateEnrollmentReport(Course course) **throws** InvalidEnrollmentDataException{

System.***out***.println("Enrollment Report for course: " + course.getCourseName());

**if** (course == **null**) {

**throw** **new** InvalidEnrollmentDataException("Course data is missing for enrollment.");

}

List<Enrollment> enrollments = course.getEnrollments();

**if** (enrollments.isEmpty()) {

System.***out***.println("No students enrolled yet.");

}

**else** {

**for** (Enrollment e : enrollments) {

e.getStudent().displayStudentInfo();;

//System.out.println("- " + student.getFirstName() + " " + student.getLastName());

}

}

}

**public** **void** generatePaymentReport(Student student) {

System.***out***.println("Payment Report for student: " + student.getFirstName());

List<Payment> payments = student.getPaymentHistory();

**if** (payments.isEmpty()) {

System.***out***.println("No payments recorded.");

}

**else** {

**for** (Payment p : payments) {

System.***out***.println("Paid Rs." + p.getPaymentAmount() + " on " + p.getPaymentDate());

}

}

}

**public** **void** calculateCourseStatistics(Course course)**throws** InvalidCourseDataException{

List<Enrollment> enrollments = course.getEnrollments();

**if** (course.getCourseCode() == **null** || course.getCourseCode().isEmpty()) {

**throw** **new** InvalidCourseDataException("Course code is invalid.");

}

**int** totalEnrolled = enrollments.size();

**double** totalPayments = 0;

**for** (Enrollment e : enrollments) {

**for** (Payment p : e.getStudent().getPaymentHistory()) {

totalPayments += p.getPaymentAmount();

}

}

System.***out***.println("Course Statistics for: " + course.getCourseName());

System.***out***.println("Total Enrollments: " + totalEnrolled);

System.***out***.println("Total Payments by Enrolled Students: Rs." + totalPayments);

}

**public** **void** addEnrollment(Student student, Course course, LocalDate enrollmentDate) **throws** DuplicateEnrollmentException, StudentNotFoundException, CourseNotFoundException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found: " + student.getStudentId());

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found: " + course.getCourseId());

}

**for** (Enrollment e : enrollments) {

**if** (e.getStudent().equals(student) && e.getCourse().equals(course)) {

**throw** **new** DuplicateEnrollmentException("Student already enrolled in this course.");

}

}

Enrollment enrollment = **new** Enrollment(**new** Random().nextInt(1000), student, course, enrollmentDate);

student.getEnrollments().add(enrollment);

course.getEnrollments().add(enrollment);

enrollments.add(enrollment);

System.***out***.println("Enrollment added successfully.");

}

**public** **void** assignCourseToTeacher(Course course, Teacher teacher) **throws** TeacherNotFoundException, CourseNotFoundException{

**if** (!teachers.contains(teacher)) {

**throw** **new** TeacherNotFoundException("Teacher not found.");

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found.");

}

course.assignTeacher(teacher);

teacher.addAssignedCourse(course);

System.***out***.println("Course " + course.getCourseName() + " assigned to " + teacher.getFirstName() + " " + teacher.getLastName());

}

**public** **void** addPayment(Student student, **double** amount, LocalDate paymentDate) **throws** StudentNotFoundException, PaymentValidationException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found.");

}

**if** (amount <= 0) {

**throw** **new** PaymentValidationException("Invalid payment amount.");

}

Payment payment = **new** Payment(**new** Random().nextInt(1000), student, amount, paymentDate);

student.getPaymentHistory().add(payment);

payments.add(payment);

System.***out***.println("Payment of Rs." + amount + " added for " + student.getFirstName());

}

**public** List<Enrollment> getEnrollmentsForStudent(Student student) {

**return** student.getEnrollments();

}

**public** List<Course> getCoursesForTeacher(Teacher teacher) {

**return** teacher.getAssignedCourses();

}

**public** **void** registerStudent(Student student) **throws** InvalidStudentDataException{

**if** (student.getEmail() == **null** || !student.getEmail().contains("@")) {

**throw** **new** InvalidStudentDataException("Invalid email for student.");

}

**if** (student.getDateOfBirth() == **null** || student.getDateOfBirth().isAfter(LocalDate.*now*())) {

**throw** **new** InvalidStudentDataException("Invalid date of birth.");

}

students.add(student);

}

**public** **void** registerCourse(Course course) **throws** InvalidCourseDataException{

**if** (course.getCourseName() == **null** || course.getCourseName().isEmpty()) {

**throw** **new** InvalidCourseDataException("Course name is required.");

}

courses.add(course);

}

**public** **void** registerTeacher(Teacher teacher) {

teachers.add(teacher);

}

}

**package** entity;

**import** java.time.LocalDate;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Random;

**import** exception.CourseNotFoundException;

**import** exception.DuplicateEnrollmentException;

**import** exception.InvalidCourseDataException;

**import** exception.InvalidEnrollmentDataException;

**import** exception.InvalidStudentDataException;

**import** exception.PaymentValidationException;

**import** exception.StudentNotFoundException;

**import** exception.TeacherNotFoundException;

**public** **class** SIS {

**private** List<Student> students;

**private** List<Course> courses;

**private** List<Teacher> teachers;

**private** List<Enrollment> enrollments;

**private** List<Payment> payments;

**public** SIS() {

// **TODO** Auto-generated constructor stub

**super**();

students = **new** ArrayList<>();

courses = **new** ArrayList<>();

teachers = **new** ArrayList<>();

enrollments = **new** ArrayList<>();

payments = **new** ArrayList<>();

}

**public** **void** enrollStudentInCourse(Student student, Course course) **throws** DuplicateEnrollmentException{

**for** (Enrollment e : enrollments) {

**if** (e.getStudent().equals(student) && e.getCourse().equals(course)) {

**throw** **new** DuplicateEnrollmentException("Student is already enrolled in this course.");

}

}

student.enrollInCourse(course);

Enrollment enrollment = **new** Enrollment(**new** Random().nextInt(1000), student, course, LocalDate.*now*());

course.addEnrollment(enrollment);

enrollments.add(enrollment);

}

**public** **void** assignTeacherToCourse(Teacher teacher, Course course) **throws** TeacherNotFoundException, CourseNotFoundException{

**if** (!teachers.contains(teacher)) {

**throw** **new** TeacherNotFoundException("Teacher not found in the system.");

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found in the system.");

}

course.assignTeacher(teacher);

teacher.addAssignedCourse(course);

System.***out***.println("Assigned " + teacher.getFirstName()+" "+teacher.getLastName() + " to course: " + course.getCourseName());

}

**public** **void** recordPayment(Student student, **double** amount, LocalDate paymentDate) **throws** StudentNotFoundException, PaymentValidationException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found in the system.");

}

**if** (amount <= 0 || paymentDate.isAfter(LocalDate.*now*())) {

**throw** **new** PaymentValidationException("Payment amount must be positive.");

}

student.makePayment(amount, paymentDate);

System.***out***.println("Payment of Rs." + amount + " recorded for student: " + student.getFirstName()+" "+ student.getLastName());

}

**public** **void** generateEnrollmentReport(Course course) **throws** InvalidEnrollmentDataException{

System.***out***.println("Enrollment Report for course: " + course.getCourseName());

**if** (course == **null**) {

**throw** **new** InvalidEnrollmentDataException("Course data is missing for enrollment.");

}

List<Enrollment> enrollments = course.getEnrollments();

**if** (enrollments.isEmpty()) {

System.***out***.println("No students enrolled yet.");

}

**else** {

**for** (Enrollment e : enrollments) {

e.getStudent().displayStudentInfo();;

//System.out.println("- " + student.getFirstName() + " " + student.getLastName());

}

}

}

**public** **void** generatePaymentReport(Student student) {

System.***out***.println("Payment Report for student: " + student.getFirstName());

List<Payment> payments = student.getPaymentHistory();

**if** (payments.isEmpty()) {

System.***out***.println("No payments recorded.");

}

**else** {

**for** (Payment p : payments) {

System.***out***.println("Paid Rs." + p.getPaymentAmount() + " on " + p.getPaymentDate());

}

}

}

**public** **void** calculateCourseStatistics(Course course)**throws** InvalidCourseDataException{

List<Enrollment> enrollments = course.getEnrollments();

**if** (course.getCourseCode() == **null** || course.getCourseCode().isEmpty()) {

**throw** **new** InvalidCourseDataException("Course code is invalid.");

}

**int** totalEnrolled = enrollments.size();

**double** totalPayments = 0;

**for** (Enrollment e : enrollments) {

**for** (Payment p : e.getStudent().getPaymentHistory()) {

totalPayments += p.getPaymentAmount();

}

}

System.***out***.println("Course Statistics for: " + course.getCourseName());

System.***out***.println("Total Enrollments: " + totalEnrolled);

System.***out***.println("Total Payments by Enrolled Students: Rs." + totalPayments);

}

**public** **void** addEnrollment(Student student, Course course, LocalDate enrollmentDate) **throws** DuplicateEnrollmentException, StudentNotFoundException, CourseNotFoundException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found: " + student.getStudentId());

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found: " + course.getCourseId());

}

**for** (Enrollment e : enrollments) {

**if** (e.getStudent().equals(student) && e.getCourse().equals(course)) {

**throw** **new** DuplicateEnrollmentException("Student already enrolled in this course.");

}

}

Enrollment enrollment = **new** Enrollment(**new** Random().nextInt(1000), student, course, enrollmentDate);

student.getEnrollments().add(enrollment);

course.getEnrollments().add(enrollment);

enrollments.add(enrollment);

System.***out***.println("Enrollment added successfully.");

}

**public** **void** assignCourseToTeacher(Course course, Teacher teacher) **throws** TeacherNotFoundException, CourseNotFoundException{

**if** (!teachers.contains(teacher)) {

**throw** **new** TeacherNotFoundException("Teacher not found.");

}

**if** (!courses.contains(course)) {

**throw** **new** CourseNotFoundException("Course not found.");

}

course.assignTeacher(teacher);

teacher.addAssignedCourse(course);

System.***out***.println("Course " + course.getCourseName() + " assigned to " + teacher.getFirstName() + " " + teacher.getLastName());

}

**public** **void** addPayment(Student student, **double** amount, LocalDate paymentDate) **throws** StudentNotFoundException, PaymentValidationException{

**if** (!students.contains(student)) {

**throw** **new** StudentNotFoundException("Student not found.");

}

**if** (amount <= 0) {

**throw** **new** PaymentValidationException("Invalid payment amount.");

}

Payment payment = **new** Payment(**new** Random().nextInt(1000), student, amount, paymentDate);

student.getPaymentHistory().add(payment);

payments.add(payment);

System.***out***.println("Payment of Rs." + amount + " added for " + student.getFirstName());

}

**public** List<Enrollment> getEnrollmentsForStudent(Student student) {

**return** student.getEnrollments();

}

**public** List<Course> getCoursesForTeacher(Teacher teacher) {

**return** teacher.getAssignedCourses();

}

**public** **void** registerStudent(Student student) **throws** InvalidStudentDataException{

**if** (student.getEmail() == **null** || !student.getEmail().contains("@")) {

**throw** **new** InvalidStudentDataException("Invalid email for student.");

}

**if** (student.getDateOfBirth() == **null** || student.getDateOfBirth().isAfter(LocalDate.*now*())) {

**throw** **new** InvalidStudentDataException("Invalid date of birth.");

}

students.add(student);

}

**public** **void** registerCourse(Course course) **throws** InvalidCourseDataException{

**if** (course.getCourseName() == **null** || course.getCourseName().isEmpty()) {

**throw** **new** InvalidCourseDataException("Course name is required.");

}

courses.add(course);

}

**public** **void** registerTeacher(Teacher teacher) {

teachers.add(teacher);

}

}

**Task 7: Database Connectivity**

**Database Initialization**:

Implement a method that initializes a database connection and creates tables for storing student,

course, enrollment, teacher, and payment information. Create SQL scripts or use code-first migration to create tables with appropriate schemas for your SIS.

**DBPropertyUtil.java:**

package util;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Properties;

public class DBPropertyUtil {

public static String getConnectionString(String fileName) throws IOException {

String connStr = null;

Properties props = new Properties();

FileInputStream fis = new FileInputStream(fileName);

props.load(fis);

String user = props.getProperty("user");

String password = props.getProperty("password");

String protocol = props.getProperty("protocol");

String system = props.getProperty("system");

String database = props.getProperty("database");

String port = props.getProperty("port");

connStr=protocol+"//"+system+":"+port+"/"+database+"?user="+user+"&password="+password;

return connStr;

}

}

**DBConnUtil.java:**

package util;

import java.io.IOException;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DBConnUtil {

private static final String fileName = "db.properties";

public static Connection getDbConnection() {

Connection con = null;

String connString = null;

try {

connString = DBPropertyUtil.getConnectionString(fileName);

}

catch(IOException e) {

System.out.println("Connection string creation failed");

e.printStackTrace();

}

if(connString != null) {

try {

con = DriverManager.getConnection(connString);

}

catch(SQLException e) {

System.out.println("Error while establishisng DBConnection...");

e.printStackTrace();

}

}

return con;

}

}

**Data Retrieval**:

Implement methods to retrieve data from the database. Users should be able to request information about students, courses, enrollments, teachers, or payments. Ensure that the data retrieval methods handle exceptions and edge cases gracefully.

**Data Insertion and Updating**:

Implement methods to insert new data (e.g., enrollments, payments) into the database and update existing data (e.g., student information). Use methods to perform data insertion and updating.

Implement validation checks to ensure data integrity and handle any errors during these operations.

**IStudentService.java:**

package dao;

import java.sql.SQLException;

import java.time.LocalDate;

import java.util.List;

import entity.Course;

import entity.Payment;

import entity.Student;

import exception.CourseNotFoundException;

import exception.DuplicateEnrollmentException;

import exception.InvalidStudentDataException;

import exception.PaymentValidationException;

import exception.StudentNotFoundException;

public interface IStudentService {

boolean addStudent(Student student) throws InvalidStudentDataException;

boolean updateStudentInfo(Student student, String firstName, String lastName, LocalDate dob, String email, String phoneNumber) throws StudentNotFoundException;

boolean deleteStudent(int studentId) throws StudentNotFoundException;

Student getStudentById(int studentId) throws StudentNotFoundException;

List<Student> getAllStudents();

void enrollInCourse(Student student, Course course) throws DuplicateEnrollmentException, StudentNotFoundException, CourseNotFoundException;

void makePayment(Student student, double amount, LocalDate paymentDate) throws StudentNotFoundException, PaymentValidationException;

void displayStudentInfo(Student student);

boolean enrollStudentWithPayment(Student student, Course course, double amount, LocalDate paymentDate)throws SQLException, StudentNotFoundException, CourseNotFoundException, PaymentValidationException;

List<Student> queryStudents(List<String> columns, String condition, String orderBy);}

**ICourseService.java:**

package dao;

import java.util.List;

import entity.Course;

import entity.Enrollment;

import entity.Teacher;

import exception.CourseNotFoundException;

import exception.InvalidCourseDataException;

public interface ICourseService {

boolean addCourse(Course course) throws InvalidCourseDataException;

boolean updateCourse(Course course, String courseName, String courseCode, int teacherId, String instructorName) throws CourseNotFoundException;

boolean deleteCourse(int courseId) throws CourseNotFoundException;

Course getCourseById(int courseId) throws CourseNotFoundException;

Course getCourseByCode(String courseCode); // Task 9 & 11

List<Course> getAllCourses();

}

**ITeacherService.java:**

package dao;

import java.util.List;

import entity.Course;

import entity.Teacher;

import exception.CourseNotFoundException;

import exception.InvalidTeacherDataException;

import exception.TeacherNotFoundException;

public interface ITeacherService {

boolean addTeacher(Teacher teacher) throws InvalidTeacherDataException;

boolean updateTeacher(Teacher teacher, String firstName, String lastName, String email, String expertise) throws TeacherNotFoundException;

boolean deleteTeacher(int teacherId) throws TeacherNotFoundException;

boolean assignCourseToTeacher(Course course, Teacher teacher) throws TeacherNotFoundException, CourseNotFoundException;

Teacher getTeacherById(int teacherId) throws TeacherNotFoundException;

List<Teacher> getAllTeachers();

}

**IEnrollmentService.java:**

package dao;

import java.time.LocalDate;

import java.util.List;

import entity.Course;

import entity.Enrollment;

import entity.Student;

import exception.CourseNotFoundException;

import exception.DuplicateEnrollmentException;

import exception.InvalidEnrollmentDataException;

import exception.StudentNotFoundException;

public interface IEnrollmentService {

boolean addEnrollment(Enrollment enrollment) throws InvalidEnrollmentDataException;

boolean removeEnrollment(int enrollmentId);

List<Enrollment> getEnrollmentsForStudent(Student student);

List<Enrollment> getAllEnrollments();

List<Enrollment> getEnrollmentsByCourse(String courseName);

Enrollment getEnrollmentById(int enrollmentId);

boolean enrollStudentInCourse(Student student, Course course, LocalDate enrollmentDate) throws StudentNotFoundException, CourseNotFoundException, DuplicateEnrollmentException;

}

**IPaymentService.java:**

package dao;

import java.time.LocalDate;

import java.util.List;

import entity.Payment;

import entity.Student;

import exception.PaymentValidationException;

import exception.StudentNotFoundException;

public interface IPaymentService {

boolean addPayment(Payment payment) throws PaymentValidationException; //same as recordPayment

boolean removePayment(int paymentId);

boolean recordPayment(Student student, double amount, LocalDate paymentDate) throws StudentNotFoundException, PaymentValidationException; // Task 10

List<Payment> getPaymentsByStudent(Student student);

List<Payment> getAllPayments();

Payment getPaymentById(int paymentId);

}

**Transaction Management**:

Implement methods for handling database transactions when enrolling students, assigning teachers, or recording payments. Transactions should be atomic and maintain data integrity. Use database transactions to ensure that multiple related operations either all succeed or all fail. Implement error handling and rollback mechanisms in case of transaction failures.

**StudentService.java:**

@Override

**public** **boolean** enrollStudentWithPayment(Student student, Course course, **double** amount, LocalDate paymentDate)

**throws** SQLException, StudentNotFoundException, CourseNotFoundException, PaymentValidationException {

// **TODO** Auto-generated method stub

**boolean** flag = **false**;

**try** {

con.setAutoCommit(**false**); // Begin transaction

// Enroll in course

PreparedStatement enrollStmt = con.prepareStatement(

"insert into enrollment (student\_id, course\_id, enrollment\_date) values(?, ?, ?)");

enrollStmt.setInt(1, student.getStudentId());

enrollStmt.setInt(2, course.getCourseId());

enrollStmt.setDate(3, Date.*valueOf*(LocalDate.*now*()));

enrollStmt.executeUpdate();

//Record payment

**if** (amount <= 0) {

**throw** **new** PaymentValidationException("Invalid payment amount.");

}

PreparedStatement paymentStmt = con.prepareStatement(

"insert into payment (student\_id, amount, payment\_date) values (?, ?, ?)");

paymentStmt.setInt(1, student.getStudentId());

paymentStmt.setDouble(2, amount);

paymentStmt.setDate(3, Date.*valueOf*(paymentDate));

paymentStmt.executeUpdate();

con.commit(); // Commit if all succeeded

flag = **true**;

System.***out***.println("Student enrolled and payment recorded successfully.");

}

**catch** (SQLException | PaymentValidationException e) {

con.rollback(); // Rollback if anything fails

System.***err***.println("Transaction failed: " + e.getMessage());

}

**finally** {

con.setAutoCommit(**true**); // Restore auto-commit mode

}

**return** flag;

}

**Dynamic Query Builder**:

Implement a dynamic query builder that allows users to construct and execute custom SQL queries to retrieve specific data from the database. Users should be able to specify columns, conditions, and sorting criteria. Create a query builder method that dynamically generates SQL queries based on user input. Implement parameterization and sanitation of user inputs to prevent SQL injection.

**StudentService.java:**

//Dynamic Query Builder

**public** List<Student> queryStudents(List<String> columns, String condition, String orderBy) {

List<Student> students = **new** ArrayList<>();

String columnPart = columns.isEmpty() ? "\*" : String.*join*(", ", columns);

String sql = "select " + columnPart + " from student";

**if** (condition != **null** && !condition.isEmpty()) {

sql += " where " + condition;

}

**if** (orderBy != **null** && !orderBy.isEmpty()) {

sql += " order by " + orderBy;

}

**try** (PreparedStatement pstmt = con.prepareStatement(sql)) {

ResultSet rs = pstmt.executeQuery();

**while** (rs.next()) {

Student s = **new** Student();

s.setStudentId(rs.getInt("student\_id"));

**if** (columns.contains("first\_name")) s.setFirstName(rs.getString("first\_name"));

**if** (columns.contains("last\_name")) s.setLastName(rs.getString("last\_name"));

**if** (columns.contains("email")) s.setEmail(rs.getString("email"));

**if** (columns.contains("dob")) s.setDateOfBirth(rs.getDate("dob").toLocalDate());

**if** (columns.contains("phone\_number")) s.setPhoneNumber(rs.getString("phone\_number"));

students.add(s);

}

} **catch** (SQLException e) {

System.***err***.println("Error executing dynamic query: " + e.getMessage());

}

**return** students;

}

**MainModule.java:**

package main;

import java.sql.Connection;

import java.time.LocalDate;

import java.util.List;

import java.util.Scanner;

import dao.CourseService;

import dao.EnrollmentService;

import dao.PaymentService;

import dao.StudentService;

import dao.TeacherService;

import entity.Course;

import entity.Enrollment;

import entity.Payment;

import entity.SIS;

import entity.Student;

import entity.Teacher;

import exception.CourseNotFoundException;

import exception.DuplicateEnrollmentException;

import exception.InsufficientFundsException;

import exception.InvalidCourseDataException;

import exception.InvalidEnrollmentDataException;

import exception.InvalidStudentDataException;

import exception.InvalidTeacherDataException;

import exception.PaymentValidationException;

import exception.StudentNotFoundException;

import exception.TeacherNotFoundException;

import util.DBConnUtil;

import util.DBInitializer;

import util.DBPropertyUtil;

public class MainModule {

public static void main(String[] args) {

// TODO Auto-generated method stub

Connection con = DBConnUtil.getDbConnection();

if(con != null) {

System.out.println(con);

System.out.println("Connected!");

}

else {

System.out.println("failed to connect");

}

// DBInitializer.initialize();//to create tables if it does not exist

Scanner sc = new Scanner(System.in);

StudentService studentService = new StudentService();

TeacherService teacherService = new TeacherService();

CourseService courseService = new CourseService();

EnrollmentService enrollmentService = new EnrollmentService();

PaymentService paymentService = new PaymentService();

while (true) {

System.out.println("\n--- Student Information System ---");

System.out.println("1. Enroll Student in Course");

System.out.println("2. Assign Course to Teacher");

System.out.println("3. Record Student Payment");

System.out.println("4. Get Enrollments by Student");

System.out.println("5. Get Payments by Student");

System.out.println("6. Get Enrollments by Course Name");

System.out.println("7. Exit");

System.out.print("Select Option: ");

int choice = sc.nextInt();

switch (choice) {

case 1 -> {

try {

System.out.print("Enter Student ID: ");

int sid = sc.nextInt();

Student student = studentService.getStudentById(sid);

System.out.print("Enter Course ID: ");

int cid = sc.nextInt();

Course course = courseService.getCourseById(cid);

if (enrollmentService.enrollStudentInCourse(student, course, LocalDate.now())) {

System.out.println("Student enrolled successfully.");

} else {

System.out.println("Enrollment failed.");

}

} catch (StudentNotFoundException | CourseNotFoundException | DuplicateEnrollmentException e) {

System.err.println("Error: " + e.getMessage());

}

}

case 2 -> {

try {

System.out.print("Enter Teacher ID: ");

int tid = sc.nextInt();

Teacher teacher = teacherService.getTeacherById(tid);

System.out.print("Enter Course ID: ");

int cid = sc.nextInt();

Course course = courseService.getCourseById(cid);

if (teacherService.assignCourseToTeacher(course, teacher)) {

System.out.println("Course assigned to teacher successfully.");

} else {

System.out.println("Assignment failed.");

}

} catch (TeacherNotFoundException | CourseNotFoundException e) {

System.err.println("Error: " + e.getMessage());

}

}

case 3 -> {

try {

System.out.print("Enter Student ID: ");

int sid = sc.nextInt();

Student student = studentService.getStudentById(sid);

System.out.print("Enter Payment Amount: ");

double amount = sc.nextDouble();

sc.nextLine();

System.out.print("Enter Payment Date (yyyy-mm-dd): ");

LocalDate paymentDate = LocalDate.parse(sc.nextLine());

if (paymentService.recordPayment(student, amount, paymentDate)) {

System.out.println("Payment recorded successfully.");

} else {

System.out.println("Payment failed.");

}

} catch (StudentNotFoundException | PaymentValidationException e) {

System.err.println("Error: " + e.getMessage());

}

}

case 4 -> {

try {

System.out.print("Enter Student ID: ");

int sid = sc.nextInt();

Student student = studentService.getStudentById(sid);

List<Enrollment> enrollments = enrollmentService.getEnrollmentsForStudent(student);

if (enrollments.isEmpty()) {

System.out.println("No enrollments found for this student.");

} else {

for (Enrollment e : enrollments) {

System.out.println(e);

}

System.out.println("Enrollments retrieved successfully.");

}

} catch (StudentNotFoundException e) {

System.err.println("Error: " + e.getMessage());

}

}

case 5 -> {

try {

System.out.print("Enter Student ID: ");

int sid = sc.nextInt();

Student student = studentService.getStudentById(sid);

List<Payment> payments = paymentService.getPaymentsByStudent(student);

if (payments.isEmpty()) {

System.out.println("No payments found for this student.");

} else {

for (Payment p : payments) {

System.out.println(p);

}

System.out.println("Payments retrieved successfully.");

}

} catch (StudentNotFoundException e) {

System.err.println("Error: " + e.getMessage());

}

}

case 6 -> {

sc.nextLine();

System.out.print("Enter Course Name: ");

String courseName = sc.nextLine();

List<Enrollment> enrollments = enrollmentService.getEnrollmentsByCourse(courseName);

if (enrollments.isEmpty()) {

System.out.println("No enrollments found for the course: " + courseName);

} else {

for (Enrollment e : enrollments) {

System.out.println(e);

}

System.out.println("Enrollments for course '" + courseName + "' retrieved successfully.");

}

}

case 7 -> {

System.out.println("Exiting Student Information System...");

sc.close();

System.exit(0);

}

default -> System.out.println("Invalid option. Please try again.");

}

}}

}

}

**Task 8: Student Enrollment**

In this task, a new student, John Doe, is enrolling in the SIS. The system needs to record John's

information, including his personal details, and enroll him in a few courses. Database connectivity is

required to store this information.

John Doe's details:

• First Name: John

• Last Name: Doe

• Date of Birth: 1995-08-15

• Email: john.doe@example.com

• Phone Number: 123-456-7890

John is enrolling in the following courses:

• Course 1: Introduction to Programming

• Course 2: Mathematics 101

The system should perform the following tasks:

• Create a new student record in the database.

• Enroll John in the specified courses by creating enrollment records in the database.

**Sample Output:**

**A screenshot of a computer

Description automatically generated**

A computer screen shot

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A computer screen shot of a computer

Description automatically generated

**Task 9: Teacher Assignment**

In this task, a new teacher, Sarah Smith, is assigned to teach a course. The system needs to update the course record to reflect the teacher assignment.

Teacher's Details:

• Name: Sarah Smith

• Email: sarah.smith@example.com

• Expertise: Computer Science

Course to be assigned:

• Course Name: Advanced Database Management

• Course Code: CS302

The system should perform the following tasks:

• Retrieve the course record from the database based on the course code.

• Assign Sarah Smith as the instructor for the course.

• Update the course record in the database with the new instructor information.

**Sample Output:**

**A close-up of a computer screen

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**A close-up of a computer screen

Description automatically generated**

**Task 10: Payment Record**

In this task, a student, Jane Johnson, makes a payment for her enrolled courses. The system needs to record this payment in the database.

Jane Johnson's details:

• Student ID: 101

• Payment Amount: $500.00

• Payment Date: 2023-04-10

The system should perform the following tasks:

• Retrieve Jane Johnson's student record from the database based on her student ID.

• Record the payment information in the database, associating it with Jane's student record.

• Update Jane's outstanding balance in the database based on the payment amount.

**Sample Output:**

**A white background with black text

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**Task 11: Enrollment Report Generation**

In this task, an administrator requests an enrollment report for a specific course, "Computer Science 101." The system needs to retrieve enrollment information from the database and generate a report.

Course to generate the report for:

• Course Name: Computer Science 101

The system should perform the following tasks:

• Retrieve enrollment records from the database for the specified course.

• Generate an enrollment report listing all students enrolled in Computer Science 101.

• Display or save the report for the administrator.

**Sample Output:**

**A computer screen shot of a computer program

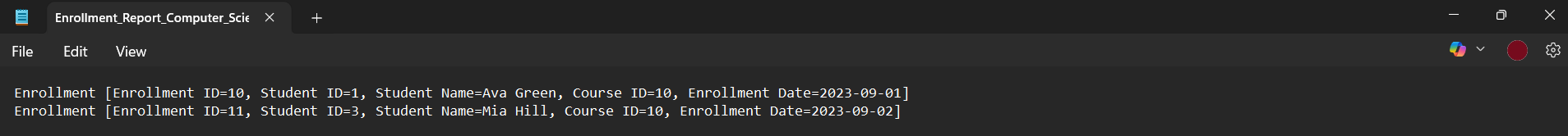
Description automatically generated**

**A computer screen shot of a computer

Description automatically generated**

**A computer screen shot of a computer code

Description automatically generated**

****