**CLUB ADVISOR CONTROLLER**

Class Variables: Two class variables are declared - fileName of type String and stage of type Stage. fileName will hold the path of the FXML file to be loaded, and stage will reference the current stage (window) of the application.

 This method is used to load a new stage (window) in the application. It takes an ActionEvent (which is triggered when a user interacts with the GUI) and a String (the path of the FXML file to be loaded). It uses the FXMLLoader to load the FXML file, gets the current stage from the event source, creates a new scene with the loaded FXML file, and sets this scene on the stage.

This method is called when the “Add Club” button is clicked. It sets the fileName to the path of the CreateClub.fxml file and calls the stageLoader method to load this file.

onClubXPressClick Methods: These methods are called when the respective “Club X Press” buttons are clicked. They set the fileName to the path of the PressClub.fxml file and call the stageLoader method to load this file.

**PRESS CLUB CONTROLLER**

private Connection connections = SCMSEnvironment.getInstance().makeSqlDBConnection();

This line establishes a database connection using a **Connection** object. It appears to be using a method **makeSqlDBConnection** from the **SCMSEnvironment** class to create the database connection.

public void stageLoader(ActionEvent event, String fileName) throws IOException { //STAGE LOADER METHOD  
 Parent root = FXMLLoader.*load*(getClass().getResource(fileName));  
 stage = (Stage) ((Node) event.getSource()).getScene().getWindow();  
 Scene scene = new Scene(root);  
 stage.setScene(scene);  
 stage.show();  
}

This method, **stageLoader**, is used for loading a new stage (window) in the JavaFX application. It takes an **ActionEvent** and a **String** fileName as parameters. Here's what it does:

* Loads an FXML file specified by the fileName parameter using **FXMLLoader**.
* Retrieves the current stage from the event source.
* Creates a new **Scene** with the loaded FXML content and sets it to the current stage.
* Finally, it shows the stage to display the new content.
* public void onRemoveStudentClick(ActionEvent event) throws Exception{  
   String fileName="/SCMS/FxmlFiles/DeleteStudent.fxml"; //open the page  
   stageLoader(event,fileName);  
    
  }

This method, **onRemoveStudentClick**, is associated with an event and is used to open a new window (stage) for removing a student from a club. It calls the **stageLoader** method to load the specified FXML file.

public void onViewStudentsButtonClick(ActionEvent event) throws Exception{  
 String fileName="/SCMS/FxmlFiles/view Students.fxml"; //open the page  
 stageLoader(event,fileName);  
  
}

This method, **onViewStudentsButtonClick**, is associated with an event and is used to open a new window (stage) for viewing students in a club. It also calls the **stageLoader** method to load the specified FXML file.

public boolean checkIfAdvisorManagesClub(String clubId, String advisorId) {  
 String query = "SELECT 1 FROM Club WHERE clubId = ? AND idOfAdvisor = ?";  
  
 boolean advisorManagesClub = false;  
  
 try (PreparedStatement statement = connections.prepareStatement(query)) {  
 statement.setString(1, clubId);  
 statement.setString(2, advisorId);  
  
 ResultSet resultSet = statement.executeQuery();  
 if (resultSet.next()) {  
 advisorManagesClub = true;  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
  
 }  
  
 return advisorManagesClub;

This method, **checkIfAdvisorManagesClub**, checks if a club advisor manages a specific club. It takes the **clubId** and **advisorId** as parameters.

Constructs an SQL query that checks if there's a row in the **Club** table with the specified **clubId** and **advisorId**.

statement.setString(1, clubId);

statement.setString(2, advisorId);

These lines set the values for the placeholders in the prepared statement. **statement.setString(1, clubId)** sets the value of the first placeholder to the **clubId** parameter, and **statement.setString(2, advisorId)** sets the value of the second placeholder to the **advisorId** parameter.

ResultSet resultSet = statement.executeQuery();

This line executes the SQL query and stores the result in a **ResultSet** object named **resultSet**. The **ResultSet** contains the results of the query, and it can be used to check if any rows were returned.

if (resultSet.next()) {

advisorManagesClub = true;

}

This block of code checks if there is at least one row in the **resultSet** (result of the query). The **resultSet.next()** method moves the cursor to the first row, and if it returns **true**, it means there is at least one matching row in the **Club** table. In that case, **advisorManagesClub** is set to **true** to indicate that the advisor manages the club.

} catch (SQLException e) {

e.printStackTrace();

}

This catch block handles any **SQLException** that might occur during the execution of the SQL query. If an exception is thrown, it is caught, and an error message is printed to the console using **e.printStackTrace()**

return advisorManagesClub;

Finally, this line returns the value of the advisorManagesClub boolean variable. If the advisor manages the club, it returns true; otherwise, it returns false.

So, this method checks whether a club advisor manages a specific club by executing an SQL query and returning a boolean result.

public void deleteClub(String clubId) { //modify to pass the object and use getters to get infor  
  
 try {  
 // Delete all students from the Club\_Student table  
 String deleteStudentsQuery = "DELETE FROM Club\_Student WHERE clubId = ?";  
 try (PreparedStatement studentsStatement = connections.prepareStatement(deleteStudentsQuery)) {  
 studentsStatement.setString(1, clubId);  
 studentsStatement.executeUpdate();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("All students associated with Club ID " + clubId + " have been removed.");  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

public void deleteClub(String clubId) {

This line defines a public method named **deleteClub**, which takes a **clubId** parameter. This method is responsible for deleting a club, specifically by removing all students associated with that club from the club\_student table in the database.

// Delete all students from the Club\_Student table

String deleteStudentsQuery = "DELETE FROM Club\_Student WHERE clubId = ?";

Here, a SQL query is defined as a String variable named deleteStudentsQuery. This query is designed to delete all rows from the Club\_Student table where the clubId matches the specified clubId. The ? is a placeholder for the actual value of clubId, which will be provided later.

try (PreparedStatement studentsStatement = connections.prepareStatement(deleteStudentsQuery)) {

A nested try-with-resources block begins here. It prepares a SQL statement using the **deleteStudentsQuery** to delete students associated with the specified club.

studentsStatement.setString(1, clubId);

This line sets the value of the first placeholder (**?**) in the prepared statement to the value of the **clubId** parameter. This parameter will be used as a condition for deleting students associated with this club.

studentsStatement.executeUpdate();

This line executes the SQL **DELETE** query on the database by calling the **executeUpdate** method on the prepared statement. This action will remove all students from the **Club\_Student** table where the **clubId** matches the specified value.

} catch (SQLException e) {

e.printStackTrace();

}

This catch block handles any **SQLException** that might occur during the execution of the SQL delete query. If an exception is thrown, it is caught, and an error message is printed to the console using **e.printStackTrace()**.

System.out.println("All students associated with Club ID " + clubId + " have been removed.");

After successfully deleting students, this line prints a message to the console indicating that all students associated with the specified **clubId** have been removed.

} catch (Exception e) {

e.printStackTrace();

}

The outer catch block here is meant to catch any exceptions that might occur outside the inner try block. However, it uses a generic **Exception** type, which is not very specific. It also prints the exception to the console using **e.printStackTrace()**.

In summary, the **deleteClub** method is designed to delete a club and, as a part of this operation, it removes all students associated with that club from the **Club\_Student** table in the database. It uses SQL **DELETE** statements to achieve this. If any exceptions occur during this process, they are caught and printed to the console.

public ClubAdvisor getClubAdvisor(String clubAdvisorId) throws SQLException {  
 String advisorQuery = "SELECT \* FROM ClubAdvisor WHERE id = ?";  
 String clubsQuery = "SELECT \* FROM Club WHERE idOfAdvisor = ?";  
  
 try (PreparedStatement advisorStatement = this.connections.prepareStatement(advisorQuery);  
 PreparedStatement clubsStatement = this.connections.prepareStatement(clubsQuery)) {  
 advisorStatement.setString(1, clubAdvisorId);  
 clubsStatement.setString(1, clubAdvisorId);  
  
 ResultSet advisorResultSet = advisorStatement.executeQuery();  
 ResultSet clubsResultSet = clubsStatement.executeQuery();  
  
 if (advisorResultSet.next()) {  
 String id = advisorResultSet.getString("id");  
 String firstName = advisorResultSet.getString("firstName");  
 String lastName = advisorResultSet.getString("lastName");  
 String dateOfBirth = advisorResultSet.getString("dateOfBirth");  
 String password = advisorResultSet.getString("password");  
  
 ArrayList<Club> managingClubs = new ArrayList<>();  
 while (clubsResultSet.next()) {  
 String clubId = clubsResultSet.getString("clubId");  
 String clubName = clubsResultSet.getString("name");  
  
 // Create a Club object and add it to the managingClubs list  
 Club club = new Club(clubId,clubName,id);  
 managingClubs.add(club);  
 }  
  
 return new ClubAdvisor(id, firstName, lastName, dateOfBirth, password, managingClubs);  
 }  
 }  
  
 return null; // ClubAdvisor not found  
}

public ClubAdvisor getClubAdvisor(String clubAdvisorId) throws SQLException {

This line defines a public method named **getClubAdvisor** that takes a **clubAdvisorId** as a parameter. It is declared to potentially throw a **SQLException** if a database-related error occurs during the execution.

String advisorQuery = "SELECT \* FROM ClubAdvisor WHERE id = ?";

String clubsQuery = "SELECT \* FROM Club WHERE idOfAdvisor = ?";

These lines declare two SQL query strings, **advisorQuery** and **clubsQuery**. The **advisorQuery** retrieves information about a club advisor with the specified **id**, and the **clubsQuery** retrieves information about the clubs managed by a club advisor with the specified **idOfAdvisor**.

try (PreparedStatement advisorStatement = this.connections.prepareStatement(advisorQuery);

PreparedStatement clubsStatement = this.connections.prepareStatement(clubsQuery)) {

This block of code starts a try-with-resources block, which initializes two **PreparedStatement** objects: **advisorStatement** and **clubsStatement**. These statements are used to execute the two SQL queries. Both statements are associated with the **connections** database connection created earlier.

advisorStatement.setString(1, clubAdvisorId);

clubsStatement.setString(1, clubAdvisorId);

These lines set the parameter for the prepared statements. They bind the **clubAdvisorId** to the first placeholder (**?**) in both **advisorStatement** and **clubsStatement**, ensuring that the SQL queries will use the provided **clubAdvisorId**.

ResultSet advisorResultSet = advisorStatement.executeQuery();

ResultSet clubsResultSet = clubsStatement.executeQuery();

These lines execute the SQL queries by calling the **executeQuery** method on the **advisorStatement** and **clubsStatement** objects. The results of the queries are stored in **ResultSet** objects: **advisorResultSet** and **clubsResultSet**.

if (advisorResultSet.next()) {

This line checks if there is at least one result row in the **advisorResultSet**. If **next()** returns **true**, it means there is a matching club advisor in the **ClubAdvisor** table.

String id = advisorResultSet.getString("id");

String firstName = advisorResultSet.getString("firstName");

String lastName = advisorResultSet.getString("lastName");

String dateOfBirth = advisorResultSet.getString("dateOfBirth");

String password = advisorResultSet.getString("password");

If a matching club advisor is found, these lines extract various attributes (e.g., **id**, **firstName**, **lastName**, **dateOfBirth**, **password**) from the result row in **advisorResultSet**.

ArrayList<Club> managingClubs = new ArrayList<>();

This line initializes an **ArrayList** named **managingClubs** to store the clubs managed by the club advisor.

while (clubsResultSet.next()) {

This loop iterates through each row in the **clubsResultSet**, which contains information about clubs managed by the advisor.

String clubId = clubsResultSet.getString("clubId");

String clubName = clubsResultSet.getString("name");

// Create a Club object and add it to the managingClubs list

Club club = new Club(clubId, clubName, id);

managingClubs.add(club);

Inside the loop, for each club managed by the advisor, it extracts the **clubId** and **name** from the **clubsResultSet** and creates a **Club** object with these details. This **Club** object is then added to the **managingClubs** list.

return new ClubAdvisor(id, firstName, lastName, dateOfBirth, password, managingClubs);

After collecting all the necessary information about the club advisor and their managed clubs, this line creates a new **ClubAdvisor** object using the gathered attributes and the **managingClubs** list. This **ClubAdvisor** object is returned as the result of the method.

return null; // ClubAdvisor not found

If the loop doesn't find a matching club advisor, it returns **null** to indicate that no club advisor with the given **clubAdvisorId** was found.

In summary, this method retrieves information about a club advisor from a database, along with the list of clubs they manage. If the advisor is found, a **ClubAdvisor** object is created and returned; otherwise, it returns **null** to indicate that the advisor was not found.

public void onDeleteClubClick(ActionEvent event) throws Exception { //using the advisorID create the object  
 confirmation = confirmText.getText(); //getting the stuff from the text fields  
 clubIdDelete = clubIdToDelete.getText();  
 advisorIdWhoIsDeleting =advisorIdDeleting.getText();  
  
 if (confirmation.equals("CONFIRM")) {  
  
 // Checking if the advisor manages the club  
 if (!checkIfAdvisorManagesClub(clubIdDelete, advisorIdWhoIsDeleting)) {  
  
 currentClubAdvisor = getClubAdvisor(advisorIdWhoIsDeleting);  
 currentClubAdvisor.deleteClub(clubIdDelete);  
 deletingStatus.setText("An advisor with That ID does not manage a Club");  
 deletingStatus1.setText("Club not deleted");  
  
  
 // Proceed with club deletion  
 deleteClub(clubIdDelete); //have to change and then the advisor cant make a new one  
  
 //clearing the text fields  
 confirmText.clear();  
 clubIdToDelete.clear();  
 advisorIdDeleting.clear();  
  
 } else {  
  
 deletingStatus.setText("An advisor with That ID does not manage a Club");  
 deletingStatus1.setText("Club not deleted");  
  
 }  
 } else {  
 deletingStatus1.setText("Deletion not confirmed. Club was not deleted.");  
 }  
}

public void onDeleteClubClick(ActionEvent event) throws Exception {

This line defines a method named **onDeleteClubClick**, which is associated with a button click event (**ActionEvent**). The method can throw an **Exception**.

confirmation = confirmText.getText();

This line retrieves the text entered into a **TextField** with the variable name **confirmText** and assigns it to the variable **confirmation**. The text entered is expected to be the word "CONFIRM."

clubIdDelete = clubIdToDelete.getText();

advisorIdWhoIsDeleting = advisorIdDeleting.getText();

These lines retrieve the text entered into two other **TextField** components, **clubIdToDelete** and **advisorIdDeleting**, and assign the entered values to the variables **clubIdDelete** and **advisorIdWhoIsDeleting**, respectively.

if (confirmation.equals("CONFIRM")) {

This line starts an **if** statement to check if the value of the **confirmation** variable is equal to the string "CONFIRM." It's used to confirm that the user intends to delete a club.

// Checking if the advisor manages the club

if (!checkIfAdvisorManagesClub(clubIdDelete, advisorIdWhoIsDeleting)) {

Inside the previous **if** block, this line checks if the advisor (specified by **advisorIdWhoIsDeleting**) manages the club (specified by **clubIdDelete**). It uses the **checkIfAdvisorManagesClub** method to make this determination. If the advisor does not manage the club, the code in this block is executed.

currentClubAdvisor = getClubAdvisor(advisorIdWhoIsDeleting);

This line calls the **getClubAdvisor** method to retrieve information about the club advisor specified by **advisorIdWhoIsDeleting**. The retrieved information is stored in the **currentClubAdvisor** variable.

currentClubAdvisor.deleteClub(clubIdDelete);

The code then calls the **deleteClub** method on the **currentClubAdvisor** to delete the club specified by **clubIdDelete**. This step assumes that the advisor manages the club, as it's only executed if the advisor is not found to manage the club.

deletingStatus.setText("An advisor with That ID does not manage a Club");

deletingStatus1.setText("Club not deleted");

These lines set text in two different **Label** components, **deletingStatus** and **deletingStatus1**, to inform the user that the specified advisor does not manage the club, and therefore, the club is not deleted.

// Proceed with club deletion

deleteClub(clubIdDelete);

This line calls the **deleteClub** method to remove all students associated with the club specified by **clubIdDelete**.

//clearing the text fields

confirmText.clear();

clubIdToDelete.clear();

advisorIdDeleting.clear();

These lines clear the text entered in the **TextField** components (**confirmText**, **clubIdToDelete**, **advisorIdDeleting**) to prepare the user interface for the next operation.

} else {

This part of the code handles the case when the **confirmation** is not equal to "CONFIRM," which means the user did not confirm the club deletion.

deletingStatus1.setText("Deletion not confirmed. Club was not deleted.");

In this case, the text in the **deletingStatus1** **Label** is set to inform the user that the deletion was not confirmed and the club was not deleted.

In summary, this method handles the deletion of a club based on user input and checks if the advisor manages the club before proceeding with the deletion. It provides informative messages to the user about the status of the deletion operation.

**REMOVE STUDENT CONTROLLER**

public class RemoveStudentController {

This line declares the class **RemoveStudentController**, which is a controller class likely used for managing the removal of students from clubs in the SCMS application.

private Connection connections = SCMSEnvironment.getInstance().makeSqlDBConnection();

Here, a **Connection** object is declared and initialized with a database connection obtained from **SCMSEnvironment**, which is a class that manages the environment settings for the SCMS application.

ClubAdvisor currentClubAdvisor = null;

Club currentClub = null;

They will likely hold information about the current club advisor and club being manipulated in the process of removing a student.

public Student getStudent(String studentId) {  
 String query = "SELECT \* FROM Student WHERE id = ?";  
 Student student = null;  
  
 try (PreparedStatement statement = connections.prepareStatement(query)) {  
 statement.setString(1, studentId);  
  
 ResultSet resultSet = statement.executeQuery();  
 if (resultSet.next()) {  
 // Retrieve student details from the result set  
 String id = resultSet.getString("id");  
 String firstName = resultSet.getString("firstName");  
 String lastName = resultSet.getString("lastName");  
 String dateOfBirth = resultSet.getString("dateOfBirth");  
 String password = resultSet.getString("password");  
 // Add more fields as needed  
  
 // Create a Student object with the retrieved data  
 student = new Student(id, firstName, lastName, dateOfBirth,password);  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
  
 }  
  
 return student;  
}

public Student getStudent(String studentId) {

This line defines a method named **getStudent**, which takes a **studentId** as an input and returns a **Student** object. It's responsible for querying the database to retrieve student information based on the provided **studentId**.

String query = "SELECT \* FROM Student WHERE id = ?";

It retrieves all fields from the "Student" table where the "id" matches the provided **studentId**. This query is parameterized with a placeholder ("?") for the student ID.

Student student = null;

A **Student** object is declared and initialized as **null**. This will be used to store the student information retrieved from the database.

try (PreparedStatement statement = connections.prepareStatement(query)) {

Here, a prepared statement is created using the database connection (**connections**) and the SQL query (**query**) defined earlier. A prepared statement is used to safely execute SQL queries with parameters.

statement.setString(1, studentId);

This line sets the value of the first parameter ("?") in the prepared statement to the **studentId**, effectively binding the **studentId** to the query.

ResultSet resultSet = statement.executeQuery();

The prepared statement is executed, and the result is stored in a **ResultSet** object, which is used to retrieve data from the database.

if (resultSet.next()) {

This conditional checks if there is at least one row in the result set. If there is a match in the database (i.e., a student with the provided **studentId** exists), the **if** block is executed.

// Retrieve student details from the result set

String id = resultSet.getString("id");

String firstName = resultSet.getString("firstName");

String lastName = resultSet.getString("lastName");

String dateOfBirth = resultSet.getString("dateOfBirth");

String password = resultSet.getString("password");

// Add more fields as needed

Inside the **if** block, the code retrieves various fields from the result set using **resultSet.getString(fieldName)**. These fields include the student's ID, first name, last name, date of birth, and password. You can add more fields as needed to complete the **Student** object.

// Create a Student object with the retrieved data

student = new Student(id, firstName, lastName, dateOfBirth, password);

This line creates a new **Student** object with the retrieved data and assigns it to the **student** variable. This object now represents the student whose information was retrieved from the database.

}

} catch (SQLException e) {

e.printStackTrace();

}

After retrieving the data or encountering an error, the **try-catch** block handles any potential **SQLExceptions**. If an error occurs during database access, it will be caught and printed to the console. The method concludes by returning the **student** object.

public ClubAdvisor getClubAdvisor(String clubAdvisorId) throws SQLException {  
  
 String query = "SELECT \* FROM ClubAdvisor WHERE id = ?"; //this is the query  
  
 try (PreparedStatement statement = this.connections.prepareStatement(query)) {  
 statement.setString(1, clubAdvisorId);  
 ResultSet resultSet = statement.executeQuery(); //resultset is used to store the data from DB  
 if (resultSet.next()) {  
 String id = resultSet.getString("id");  
 String firstName = resultSet.getString("firstName");  
 String lastName = resultSet.getString("lastName");  
 String dateOfBirth = resultSet.getString("dateOfBirth");  
 String password = resultSet.getString("password");  
  
 return new ClubAdvisor(id, firstName, lastName, dateOfBirth, password); // Create a new ClubAdvisor object  
 }  
 }  
 return null;  
}

public ClubAdvisor getClubAdvisor(String clubAdvisorId) throws SQLException {

This line defines a method named **getClubAdvisor**, which takes a **clubAdvisorId** as an input and is expected to return a **ClubAdvisor** object. The method is responsible for querying the database to retrieve information about a club advisor based on the provided **clubAdvisorId**.

String advisorQuery = "SELECT \* FROM ClubAdvisor WHERE id = ?";

String clubsQuery = "SELECT \* FROM Club WHERE idOfAdvisor = ?";

Two SQL queries are defined as strings. The **advisorQuery** retrieves all fields from the "ClubAdvisor" table where the "id" matches the provided **clubAdvisorId**. The **clubsQuery** retrieves all fields from the "Club" table where the "idOfAdvisor" matches the provided **clubAdvisorId**.

try (PreparedStatement advisorStatement = this.connections.prepareStatement(advisorQuery);

PreparedStatement clubsStatement = this.connections.prepareStatement(clubsQuery)) {

Here, two prepared statements are created. One (**advisorStatement**) is used to execute the **advisorQuery**, and the other (**clubsStatement**) is used to execute the **clubsQuery**. Both statements are associated with the same database connection (**this.connections**).

advisorStatement.setString(1, clubAdvisorId);

clubsStatement.setString(1, clubAdvisorId);

These lines set the first parameter in each prepared statement to the **clubAdvisorId**, effectively binding the **clubAdvisorId** to the respective queries.

ResultSet advisorResultSet = advisorStatement.executeQuery();

ResultSet clubsResultSet = clubsStatement.executeQuery();

Both prepared statements are executed, and the results are stored in **ResultSet** objects. **advisorResultSet** holds the result of the **advisorQuery**, and **clubsResultSet** holds the result of the **clubsQuery**.

if (advisorResultSet.next()) {

This conditional checks if there is at least one row in the **advisorResultSet**. If there is a match in the "ClubAdvisor" table (i.e., a club advisor with the provided **clubAdvisorId** exists), the **if** block is executed.

// Retrieve club advisor details from the result set

String id = advisorResultSet.getString("id");

String firstName = advisorResultSet.getString("firstName");

String lastName = advisorResultSet.getString("lastName");

String dateOfBirth = advisorResultSet.getString("dateOfBirth");

String password = advisorResultSet.getString("password");

Inside the **if** block, the code retrieves various fields from the **advisorResultSet** using **resultSet.getString(fieldName)**. These fields include the advisor's ID, first name, last name, date of birth, and password. You can add more fields as needed to complete the **ClubAdvisor** object.

ArrayList<Club> managingClubs = new ArrayList<>();

An **ArrayList** of **Club** objects named **managingClubs** is initialized. This list will be used to store information about the clubs managed by the club advisor.

String clubId = clubsResultSet.getString("clubId");

String clubName = clubsResultSet.getString("name");

// Create a Club object and add it to the managingClubs list

Club club = new Club(clubId, clubName, id);

managingClubs.add(club);

}

nside the loop, the code retrieves the **clubId** and **clubName** from the **clubsResultSet** and creates a new **Club** object for each club. These **Club** objects are then added to the **managingClubs** list.

return new ClubAdvisor(id, firstName, lastName, dateOfBirth, password, managingClubs);

After retrieving the club advisor's information and the clubs they manage, a new **ClubAdvisor** object is created and returned. This object encapsulates all the details of the club advisor and the clubs they manage.

return null; // ClubAdvisor not found

}

This line explicitly returns **null** to indicate that the club advisor with the given ID was not found in the database.

This method is responsible for retrieving information about a club advisor and the clubs they manage based on a provided **clubAdvisorId** and returning it as a **ClubAdvisor** object. It also demonstrates how to handle related data from multiple database tables.

public String getStudentFirstName(String studentId) {  
 String firstName = null;  
 String query = "SELECT firstName FROM Student WHERE id = ?";  
  
 try (PreparedStatement statement = connections.prepareStatement(query)) {  
 statement.setString(1, studentId);  
  
 ResultSet resultSet = statement.executeQuery();  
 if (resultSet.next()) {  
 firstName = resultSet.getString("FirstName");  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
  
 return firstName;  
}

public String getStudentFirstName(String studentId) {

This method is named getStudentFirstName and is used to retrieve the first name of a student based on their studentId. It takes the studentId as input and is expected to return the first name as a string.

String firstName = null;

A string variable named **firstName** is declared and initialized to **null**. This variable will be used to store the first name of the student.

String query = "SELECT firstName FROM Student WHERE id = ?";

An SQL query is defined as a string. This query selects the "firstName" field from the "Student" table where the "id" matches the provided **studentId**. It retrieves only the first name of the student.

try (PreparedStatement statement = connections.prepareStatement(query)) {

A prepared statement is created for executing the SQL query. It uses the database connection (**connections**) and the SQL query (**query**) defined earlier.

statement.setString(1, studentId);

The **studentId** is set as a parameter in the prepared statement, binding it to the query.

ResultSet resultSet = statement.executeQuery();

The prepared statement is executed, and the result is stored in a **ResultSet** object.

if (resultSet.next()) {

This conditional checks if there is at least one row in the **resultSet**. If there is a match in the database (i.e., a student with the provided **studentId** exists), the **if** block is executed.

firstName = resultSet.getString("FirstName");

Inside the **if** block, the code retrieves the first name of the student from the **resultSet** using **resultSet.getString(fieldName)**.

}

} catch (SQLException e) {

e.printStackTrace();

}

After retrieving the data or encountering an error, the **try-catch** block handles any potential **SQLExceptions**. If an error occurs during database access, it will be caught and printed to the console.

return firstName;

The method concludes by returning the **firstName**. If a match is found, the first name is returned; otherwise, it remains **null**. This method allows you to retrieve the first name of a student based on their ID.

public void saveRemovedStudent(String clubId, String studentId, String studentFirstName, String reason) {  
 try {  
  
 String query = "INSERT INTO RemovedStudents (clubId, studentId, studentFirstName, reason) VALUES (?, ?, ?, ?)";  
  
 try (PreparedStatement statement = connections.prepareStatement(query)) {  
 statement.setString(1, clubId);  
 statement.setString(2, studentId);  
 statement.setString(3, studentFirstName);  
 statement.setString(4, reason);  
  
 int rowsInserted = statement.executeUpdate();  
  
 if (rowsInserted > 0) {  
 System.*out*.println("Removed student added to the database.");  
 } else {  
 System.*out*.println("Failed to add the removed student to the database.");  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

public void saveRemovedStudent(String clubId, String studentId, String studentFirstName, String reason) {

This method is named **saveRemovedStudent** and is used to save information about a removed student in the database. It takes the club ID, student ID, student's first name, and a reason for removal as inputs.

String query = "INSERT INTO RemovedStudents (clubId, studentId, studentFirstName, reason) VALUES (?, ?, ?, ?)";

An SQL query is defined as a string. This query inserts a new row into the "RemovedStudents" table with values for club ID, student ID, student's first name, and the reason for removal. The query uses placeholders to insert values dynamically.

try (PreparedStatement statement = connections.prepareStatement(query)) {

A prepared statement is created for executing the SQL insert query. It uses the database connection (**connections**) and the SQL query (**query**) defined earlier.

statement.setString(1, clubId);

statement.setString(2, studentId);

statement.setString(3, studentFirstName);

statement.setString(4, reason);

The method sets the values for the placeholders in the prepared statement, binding the club ID, student ID, student's first name, and reason to the query.

int rowsInserted = statement.executeUpdate();

The **executeUpdate** method is called on the prepared statement to execute the insert query and update the database. The result, which is the number of rows inserted, is stored in **rowsInserted**.

if (rowsInserted > 0) {

System.out.println("Removed student added to the database.");

} else {

System.out.println("Failed to add the removed student to the database.");

}

The code then checks if any rows were successfully inserted. If **rowsInserted** is greater than 0, it indicates that the insertion was successful, and a success message is printed. If no rows were inserted (i.e., **rowsInserted** is 0), it indicates a failure, and an error message is printed.

} catch (SQLException e) {

e.printStackTrace();

}

The code also handles any **SQLExceptions** that may occur during database operations. If an error occurs, it is caught and printed to the console.

public void removeStudentFromClub(String clubId, String studentId) {  
 try {  
  
 String query = "DELETE FROM Club\_Student WHERE clubid = ? AND id = ?";  
  
 try (PreparedStatement statement = connections.prepareStatement(query)) {  
 statement.setString(1, clubId);  
 statement.setString(2, studentId);  
  
 int rowsDeleted = statement.executeUpdate();  
  
 if (rowsDeleted > 0) {  
 System.*out*.println("Student removed from club.");  
 } else {  
 System.*out*.println("Failed to remove the student from the club.");  
 }  
  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

public void removeStudentFromClub(String clubId, String studentId) {

This method, named **removeStudentFromClub**, is responsible for removing a student from a club. It takes the **clubId** and **studentId** as inputs.

String query = "DELETE FROM Club\_Student WHERE clubid = ? AND id = ?";

An SQL query is defined as a string. This query deletes a row from the "Club\_Student" table where the **clubId** and **studentId** match the provided values. It is effectively removing the association between the student and the club.

try (PreparedStatement statement = connections.prepareStatement(query)) {

A prepared statement is created for executing the SQL delete query. It uses the database connection (**connections**) and the SQL query (**query**) defined earlier.

statement.setString(1, clubId);

statement.setString(2, studentId);

The code sets the values for the placeholders in the prepared statement, binding the **clubId** and **studentId** to the query.

int rowsDeleted = statement.executeUpdate();

The **executeUpdate** method is called on the prepared statement to execute the delete query and update the database. The result, which is the number of rows deleted, is stored in **rowsDeleted**.

if (rowsDeleted > 0) {

System.out.println("Student removed from the club.");

} else {

System.out.println("Failed to remove the student from the club.");

}

The code then checks if any rows were successfully deleted. If **rowsDeleted** is greater than 0, it indicates that the removal was successful, and a success message is printed. If no rows were deleted (i.e., **rowsDeleted** is 0), it indicates a failure, and an error message is printed.