University of Sheffield

COM2008 : Systems Design and Security

Tutorial 03 - The JDBC

Department of Computer Science

Contents

Chapter 1

Introduction

1.1 Learning Objectives

1. MySQL Connector Integration: After completing this tutorial, students should be able to download the MySQL Connector for Java, choose the appropriate version, and add it to their Integrated Development Environment (IDE).

2. Database Table Creation: Students will learn how to create a new table in a MySQL database, including defining table structure, specifying data types, and creating primary keys.

3. Java Package Organization: Upon finishing the tutorial, students should understand the significance of organizing Java classes into packages, how it helps in code structure, and how to create and use packages effectively.

4. Understanding POJOs: Students will grasp the concept of Plain Old Java Objects (POJOs) and learn how to create Java classes that conform to the principles of POJO design.

5. Database Connection Handling: After this tutorial, students should be proficient in managing database connections in Java, including opening, using, and closing connections safely.

6. Database Operations: Students will gain the ability to perform common database operations such as inserting, retrieving, updating, and deleting records in a MySQL database using Java and SQL queries.

7. Error Handling: The tutorial will teach students how to handle exceptions and errors that may arise during database operations, ensuring robust and reliable code.

8. Integration of Java with MySQL: Students will understand how to integrate Java applications with MySQL databases, facilitating data storage and retrieval.

Chapter 2

How to add MySQL Connector

2.1 MySQL connector Download

In order to download the MySQL Connector for Java, please download it from this ***link***. For the 'Select Operating System' dropdown select 'Platform Independent'.

Figure 2.1: MySQL Connector Download

Once the download is complete, unzip the connector. There should be a '.jar' file inside.

2.2 Adding the Connector to an IDE

2.2.1 IntelliJ IDEA

First start by creating a new project. Then follow the steps below.

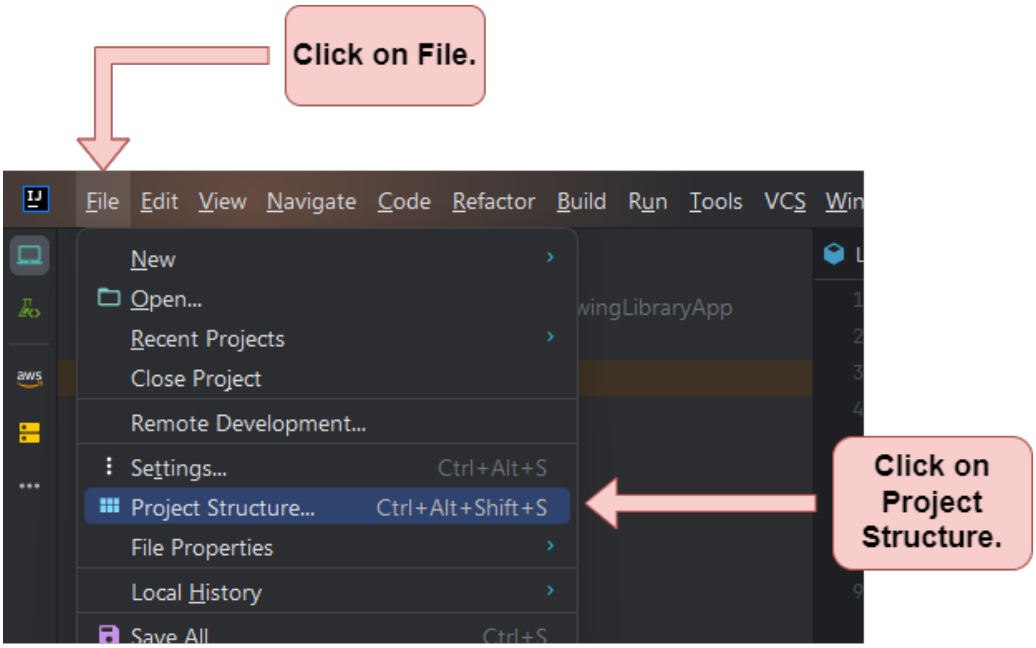

Figure 2.2: Step 1 - Opening the Project Structure Window

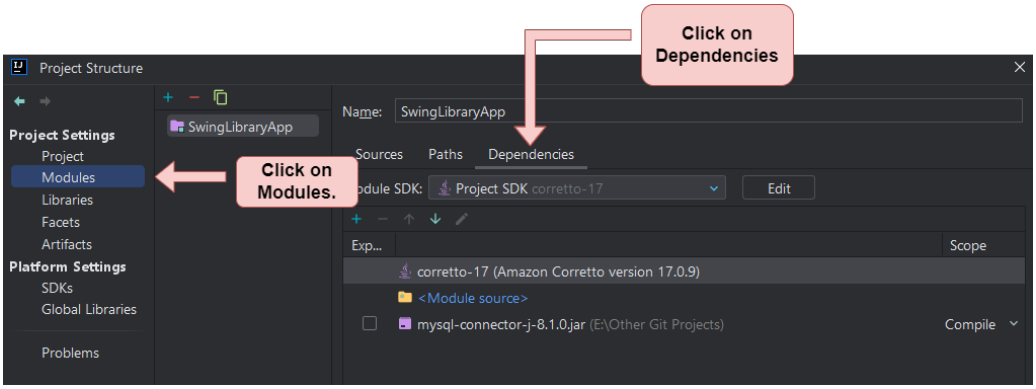

Figure 2.3: Step 2 - Navigating to the Dependencies Tab

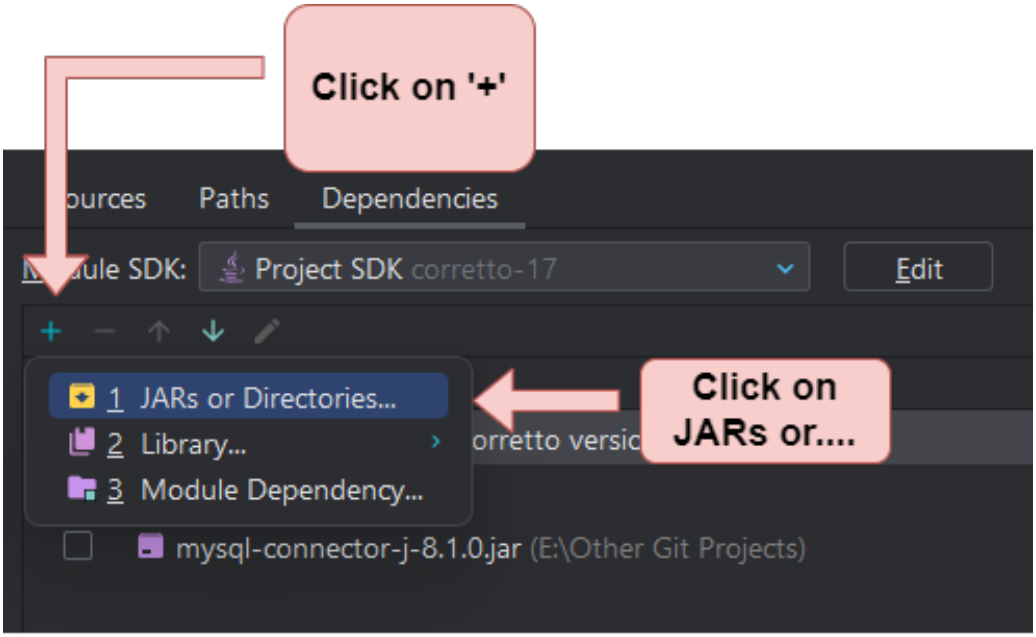

Figure 2.4: Step 3 - Adding the MySQL JAR File

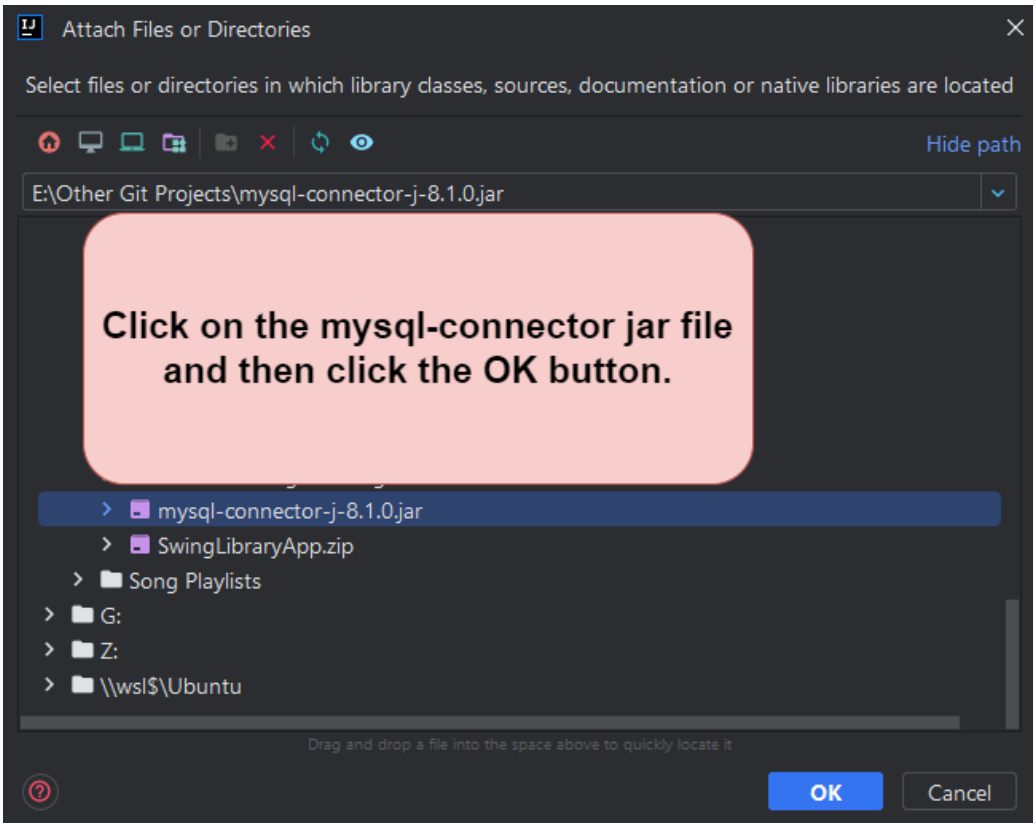

Figure 2.5: Step 4 - Navigating to and Selecting the MySQL JAR File

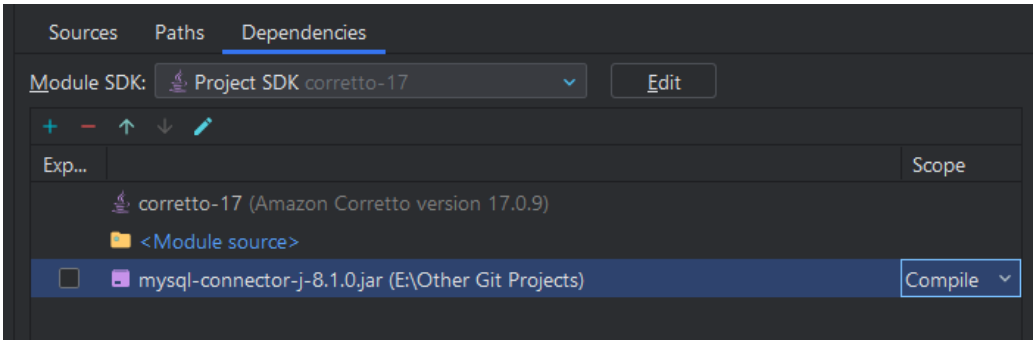

Figure 2.6: The JAR has been added Successfully.

2.2.2 Visual Studio Code

Please note that the java extension pack should be installed on visual studio code. ***Link to Extension Pack***

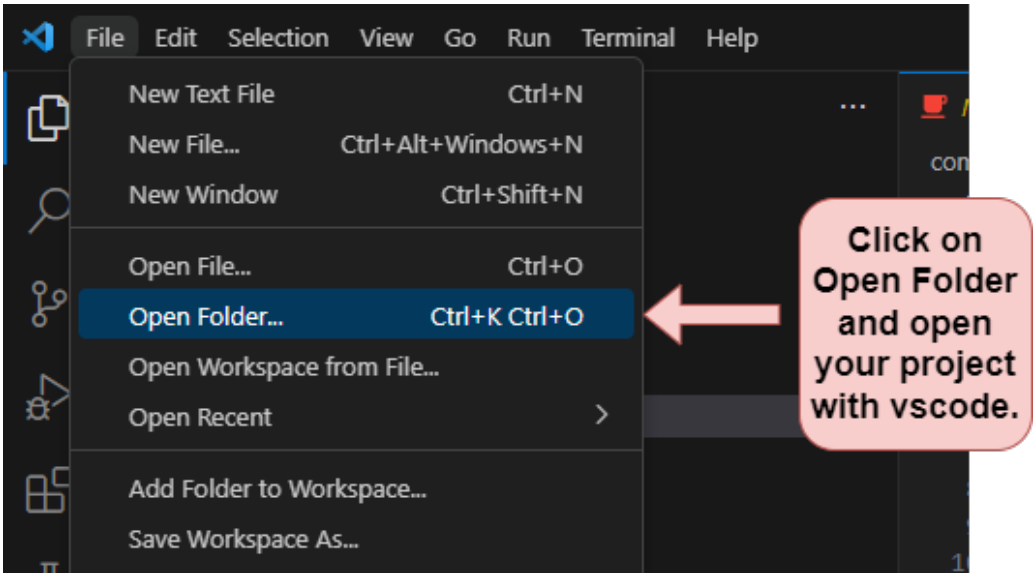

Figure 2.7: Step 1 - Open the Directory for the Project

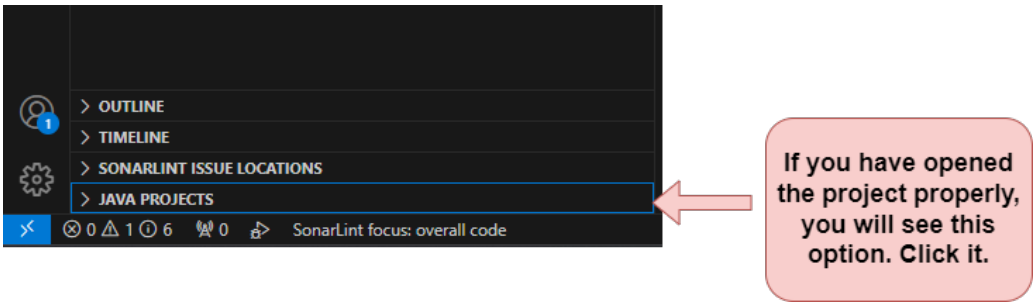

Figure 2.8: Step 2 - Opening the Java Projects Option

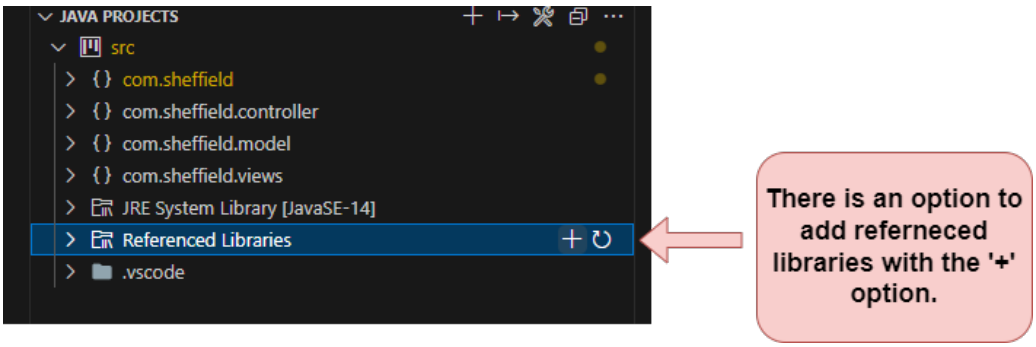

Figure 2.9: Step 3 - Adding the JAR File

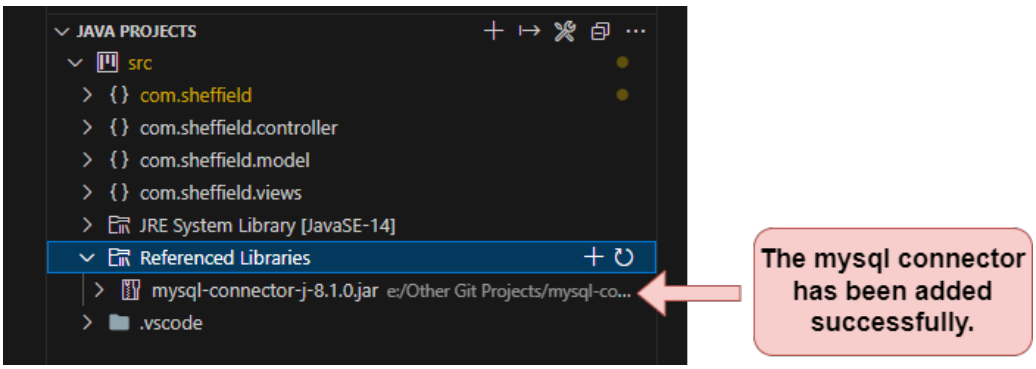

Figure 2.10: The JAR has been added Successfully.

Chapter 3

Discussion

3.1 Creating a Books Table

If you have not dropped last weeks table, please run the `DROP` command on both the Authors and Books Table. You need to create a new Books table and as previously mentioned, please append your username to the end of the table. For example if your username is **ab1ab**, please use the table name **Books_ab1ab**.

```
CREATE TABLE Books (
  isbn CHAR(13) NOT NULL,
  title VARCHAR(100) NULL,
  author_name VARCHAR(100) NULL,
  publication_year INT NULL,
  genre VARCHAR(30) NULL,
  price DECIMAL(8,2) NULL,
  PRIMARY KEY (isbn),
);
```

3.2 Packaging in Java

Creating packages in a Java project is essential for maintaining a well-organized, readable, and maintainable codebase. Packages offer a structured way to group related classes and components, making it easier for developers to locate and work with specific parts of the project. They help prevent naming conflicts, promote code encapsulation, and facilitate namespace management. Additionally, packages enable code reusability, making it possible to create libraries and frameworks that can be used in various projects. They play a crucial role in access control, security, and documenting code. Furthermore, packages aid in managing dependencies, testing, and version control, ensuring that the project remains efficient, manageable, and scalable, particularly in larger and more complex software development endeavors.

For this project we have used the `com.sheffield` package.

3.3 Classes in the 'com.sheffield' Package

3.3.1 The Book Class

The Java class, named `Book`, represents a book entity with various attributes such as ISBN, title, author name, publication year, genre, and price. It features a constructor for initializing a book object with these attributes, along with getter and setter methods for each attribute that include validation to ensure data integrity. Private validation methods are used to check the validity of each attribute, such as ISBN format, title length, and non-negativity of prices. Additionally, the class includes a method for generating a string representation of the book object. This class is designed to encapsulate and manage book-related data, ensuring that the data conforms to specified validation criteria.

POJO stands for "Plain Old Java Object." It is a design pattern and an architectural style in software development, particularly in the context of object-oriented programming. A POJO is a class that adheres to a set of principles that make it simple, straightforward, and free from unnecessary complexity, often used to represent data structures or entities in a clean and self-contained manner. The key characteristics of a POJO include:

- **No Special Requirements**: A POJO should not rely on any specific framework or library, such as JavaBeans, EJB, or other specialized technologies. It should be a simple Java class without any dependencies on external components.
- **Public Fields or Accessors**: A POJO should typically expose its attributes as public fields or provide simple getter and setter methods (accessors) for accessing and modifying its properties. These accessors should follow JavaBean naming conventions.
- **Serializable**: In some cases, POJOs are designed to be serializable to support data exchange or storage.
- **No Business Logic**: A POJO should not contain complex business logic or behavior. Its primary purpose is to represent data or entities.
- **Encapsulation**: While it may provide accessors for its attributes, a POJO encapsulates its state, and its attributes are typically private or protected fields.

Now, regarding the Book class:

The Book class provided can be considered a POJO. It adheres to the principles of a POJO as follows:

- It's a plain Java class with no dependencies on external frameworks or libraries.
- It has private fields to encapsulate its attributes (ISBN, title, authorName, etc.) and provides getter and setter methods for accessing and modifying these attributes, following JavaBean naming conventions.
- It represents a data structure for a book entity, containing attributes but not containing complex business logic.

- The class is not tied to any specific technology or framework, making it suitable for use in various contexts.

The BigDecimal Class

The `BigDecimal` class in Java is a part of the java.math package and is designed to handle arbitrary-precision decimal arithmetic. It's used for precise calculations involving decimal numbers, making it ideal for financial and monetary calculations where precision and rounding are critical.

`BigDecimal` is different from primitive data types like double or float in that it represents numbers in a decimal format with a fixed number of significant digits, allowing it to avoid common rounding errors associated with binary floating-point representations. It stores numbers as an arbitrary number of digits and can handle very large or very small values with great accuracy.

Key features of `BigDecimal` include:

- **Arbitrary Precision**: `BigDecimal` can represent numbers with an arbitrary number of decimal places. This makes it suitable for situations where high precision is required.
- **Rounding Control**: You can specify the rounding mode when performing arithmetic operations, ensuring that results are rounded in the desired manner, such as rounding half-up or half-down.
- **Immutability**: `BigDecimal` objects are immutable, which means that operations on `BigDecimal` values do not modify the original values but instead create new `BigDecimal` instances with the result.
- **Support for Mathematical Operations**: `BigDecimal` provides methods for basic mathematical operations like addition, subtraction, multiplication, and division, as well as more advanced functions like square roots and logarithms.
- **Comparison and Equality**: You can compare `BigDecimal` values for equality and order them, which is essential for sorting and filtering monetary values.
- **Exception Handling**: `BigDecimal` operations may throw exceptions like `ArithmeticException` when dealing with division by zero or other exceptional cases. This helps ensure that errors are properly handled.

Overall, BigDecimal is the recommended choice for handling monetary and financial calculations in Java because of its precision and control over rounding, which minimizes the risk of introducing errors in financial applications. It's a reliable tool for working with decimal numbers in scenarios where accuracy is paramount.

3.3.2 The DatabaseConnectionHandler Class

The `DatabaseConnectionHandler` class serves as a utility for managing database connections in a Java application. It defines a class with several responsibilities. It encapsulates a database connection to a MySQL database, storing the database URL, username, and password as constants. It provides methods for opening and closing the connection, as well as a getter method to obtain the connection itself.

The use of final variables for `DB_URL`, `DB_USER`, and `DB_PASSWORD` is because the values cannot be changed after initialization. This ensures that these critical database connection parameters remain constant throughout the application's execution, enhancing security and preventing accidental changes to the connection details.

The `openConnection` method in the `DatabaseConnectionHandler` Java class establishes a connection to a MySQL database by utilizing the `DriverManager.getConnection` method with predefined database URL, username, and password. If the connection is successful, it stores the connection in the class's connection field for later use. However, if a `SQLException` occurs during the connection process, it's caught and its details are printed to the standard error output.

The `closeConnection` method is used to safely close the database connection when it's no longer needed, ensuring proper resource management. This is particularly important to release resources and free up database connections for other parts of the application or other users when they are done using them.

The connection variable is declared as a class member, allowing it to be shared across the application. The `getConnection` method provides a way to access this connection object from other parts of the application, allowing other classes and methods to interact with the database through this established connection. This encapsulation and controlled access to the connection object are good practices for maintaining database connections in a Java application.

3.3.3 The DatabaseOperations Class

The `DatabaseOperations` class encapsulates a set of methods for interacting with a database that stores information about books. Each method corresponds to a different database operation, and these methods use Java's JDBC (Java Database Connectivity) to execute SQL queries and updates. Here's an explanation of each method in the class:

- **`insertBook`**: This method is used to insert a new book into the database. It takes a Book object (newBook) and a database connection (Connection) as parameters. It prepares an SQL INSERT statement with placeholders for book attributes (ISBN, title, author name, publication year, genre, and price) and then executes the statement by setting the corresponding values from the newBook object. It prints the number of rows affected, indicating how many records were successfully inserted.

- `getAllBooks`: This method retrieves and prints all books from the database. It creates an SQL SELECT statement to fetch all records from the ”Books” table and then iterates through the result set, printing each book's information in a specific format. This method provides a way to display the entire catalog of books stored in the database.
- `getBookByISBN`: This method retrieves and prints a book from the database by its ISBN. It takes an ISBN as a parameter and queries the database using a WHERE clause to find a book with a matching ISBN. If a book with the provided ISBN is found, its information is printed; otherwise, a message is printed to indicate that the book was not found.
- `updateBook`: This method updates an existing book in the database based on its ISBN. It takes a Book object (newBook) and a database connection as parameters. It prepares an SQL UPDATE statement to modify the book's attributes (title, author name, publication year, genre, and price) and then executes the statement by setting the values from the newBook object. It prints the number of rows affected, indicating how many records were successfully updated.
- `deleteBook`: This method deletes a book from the database by its ISBN. It takes an ISBN as a parameter and prepares an SQL DELETE statement with a WHERE clause to target the specific book by its ISBN. The method then executes the statement and prints the number of rows affected, indicating how many records were successfully deleted.

In each method, potential `SQLExceptions` are caught and re-thrown to signal an error and ensure proper error handling in the calling code. This class provides a convenient way to interact with a database that stores book information, allowing for operations like adding, updating, retrieving, and deleting books.

3.4 The Main Class

The `Main` class in this Java application serves as the entry point and orchestrates various operations related to managing a collection of books in a database. Let's break down the code in this class step by step:

Main Method: The `public static void main(String[] args)` method is the entry point of the application. It performs the following tasks:

- **Database Connection Handling**: It creates an instance of DatabaseConnectionHandler (`databaseConnectionHandler`) to manage database connections and another instance of DatabaseOperations (`databaseOperations`) to perform various operations on the database.
- **Opening Database Connection**: The `databaseConnectionHandler.openConnection()` method is called to establish a connection to the database. If the connection is successful, it allows for interaction with the database.

- **Adding Books to the Database**: Two books (`book1` and `book2`) are created using the Book class and their details are provided. These books are inserted into the database using the `databaseOperations.insertBook` method, passing the book and the database connection obtained from `databaseConnectionHandler`.
- **Retrieving and Printing All Books**: The `databaseOperations.getAllBooks` method is called to retrieve all books from the database and print their details. This demonstrates how to fetch and display book records.
- **Updating Book Information**: The information of the previously added books is updated. For example, the publication year for book1 is changed to 1970, and the price of book2 is updated. These changes are applied to the database using the `databaseOperations.getBookByISBN` method after modifying the Book objects.
- **Deleting Books**: The `databaseOperations.deleteBook` method is used to delete book1 and book2 from the database.
- **Closing Database Connection**: In the finally block, the `databaseConnectionHandler.closeConnection()` method is called to close the database connection. This ensures proper resource management and releases the database connection for other use.
- **Exception Handling**: The code is enclosed in a try-catch block to handle any `SQLException` that might occur during database operations. Any exceptions are printed to the console for error reporting.

In summary, the Main class demonstrates the integration of a database into a Java application, covering common database operations like inserting, updating, retrieving, and deleting records. It utilizes the DatabaseConnectionHandler for managing connections and the DatabaseOperations class for performing database-specific operations on book records.

3.5 Implementing Methods in the DatabaseOperations Class

The code for these methods has not been provided. Please try to implement the code by following the screenshots below.

3.5.1 `insertBook`

```java
// Insert a new book into the database
public void insertBook(Book newBook, Connection connection) throws SQLException {
    try {
        // Create an SQL INSERT statement
        String insertSQL = "INSERT INTO Books (isbn, title, author_name,"+
        "publication_year, genre, price) VALUES (?, ?, ?, ?, ?, ?)";

        // Prepare and execute the INSERT statement
        PreparedStatement preparedStatement = connection.prepareStatement(insertSQL);
        preparedStatement.setString(1, newBook.getIsbn());
        preparedStatement.setString(2, newBook.getTitle());
        preparedStatement.setString(3, newBook.getAuthorName());
        preparedStatement.setInt(4, newBook.getPublicationYear());
        preparedStatement.setString(5, newBook.getGenre());
        preparedStatement.setBigDecimal(6, newBook.getPrice());

        int rowsAffected = preparedStatement.executeUpdate();
        System.out.println(rowsAffected + " row(s) inserted successfully.");
    } catch (SQLException e) {
        e.printStackTrace();
        throw e; // Re-throw the exception to signal an error.
    }
}
```

Figure 3.1: The `insertBook` Method

3.5.2 `getAllBooks`

```java
// Get all books from the database
public void getAllBooks(Connection connection) throws SQLException {
    try {
        String selectSQL = "SELECT * FROM Books";
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        ResultSet resultSet = preparedStatement.executeQuery();
        System.out.println("<================== GET ALL BOOKS ==================>");
        while (resultSet.next()) {
            // Print each book's information in the specified format
            System.out.println("{" +
                    "isbn='" + resultSet.getString("isbn") + "'" +
                    ", title='" + resultSet.getString("title") + "'" +
                    ", authorName='" + resultSet.getString("author_name") + "'" +
                    ", publicationYear='" + resultSet.getInt("publication_year") + "'" +
                    ", genre='" + resultSet.getString("genre") + "'" +
                    ", price='" + resultSet.getDouble("price") + "'" +
                    "}");
        }
        System.out.println("<====================================================>");
    } catch (SQLException e) {
        e.printStackTrace();
        throw e;// Re-throw the exception to signal an error.
    }
}
```

Figure 3.2: The `getAllBooks` Method

3.5.3 getBookByISBN

```java
// Get a book by ISBN
public void getBookByISBN(String isbn, Connection connection) throws SQLException {
    try {
        String selectSQL = "SELECT * FROM Books WHERE isbn=?";
        PreparedStatement preparedStatement = connection.prepareStatement(selectSQL);
        preparedStatement.setString(parameterIndex:1, isbn);
        ResultSet resultSet = preparedStatement.executeQuery();
        System.out.println(x:"<=================== BOOK BY ISBN ====================>");
        if (resultSet.next()) {
            System.out.println("{" +
                    "isbn='" + resultSet.getString(columnLabel:"isbn") + "'" +
                    ", title='" + resultSet.getString(columnLabel:"title") + "'" +
                    ", authorName='" + resultSet.getString(columnLabel:"author_name") + "'" +
                    ", publicationYear='" + resultSet.getInt(columnLabel:"publication_year") + "'" +
                    ", genre='" + resultSet.getString(columnLabel:"genre") + "'" +
                    ", price='" + resultSet.getDouble(columnLabel:"price") + "'" +
                    "}");
        } else {
            System.out.println("Book with ISBN " + isbn + " not found.");
        }
        System.out.println(x:"<=====================================================>");
    } catch (SQLException e) {
        e.printStackTrace();
        throw e;// Re-throw the exception to signal an error.
    }
}
```

Figure 3.3: The `getBookByISBN` Method

3.5.4 updateBook

```java
// Update an existing book in the database
public void updateBook(Book newBook, Connection connection) throws SQLException {
    try {
        String updateSQL = "UPDATE Books SET title=?, author_name=?,"+
        "publication_year=?, genre=?, price=? WHERE isbn=?";
        PreparedStatement preparedStatement = connection.prepareStatement(updateSQL);

        preparedStatement.setString(1, newBook.getTitle());
        preparedStatement.setString(2, newBook.getAuthorName());
        preparedStatement.setInt(3, newBook.getPublicationYear());
        preparedStatement.setString(4, newBook.getGenre());
        preparedStatement.setBigDecimal(5, newBook.getPrice());
        preparedStatement.setString(6, newBook.getIsbn());

        int rowsAffected = preparedStatement.executeUpdate();

        if (rowsAffected > 0) {
            System.out.println(rowsAffected + " row(s) updated successfully.");
        } else {
            System.out.println("No rows were updated for ISBN: " + newBook.getIsbn());
        }
    } catch (SQLException e) {
        e.printStackTrace();
        throw e;// Re-throw the exception to signal an error.
    }
}
```

Figure 3.4: The `updateBook` Method

3.5.5 `deleteBook`

```java
// Delete a book from the database by ISBN
public void deleteBook(String isbn, Connection connection) throws SQLException {
    try {
        String deleteSQL = "DELETE FROM Books WHERE isbn=?";
        PreparedStatement preparedStatement = connection.prepareStatement(deleteSQL);
        preparedStatement.setString(parameterIndex:1, isbn);

        int rowsAffected = preparedStatement.executeUpdate();

        if (rowsAffected > 0) {
            System.out.println(rowsAffected + " row(s) deleted successfully.");
        } else {
            System.out.println("No rows were deleted for ISBN: " + isbn);
        }
    } catch (SQLException e) {
        e.printStackTrace();
        throw e;// Re-throw the exception to signal an error.
    }
}
```

Figure 3.5: The `deleteBook` Method

Chapter 4

Conclusion

This tutorial, focused on the integration of Java with MySQL databases. It equipped students with essential skills and knowledge in database management and Java programming. The tutorial's primary learning objectives encompassed a range of key topics:

Firstly, students learned how to integrate the MySQL Connector for Java into their Integrated Development Environments (IDEs), which facilitated communication between Java applications and MySQL databases. They gained practical experience in downloading and adding this essential component. Secondly, the tutorial introduced the process of creating a new table in a MySQL database. This included defining the table's structure, specifying data types, and creating primary keys. Students understood the significance of maintaining a well-organized database schema.

Furthermore, the tutorial explored the importance of Java package organization. It taught students how to structure and group related classes into packages, enhancing code modularity and maintainability. In addition, students grasped the concept of Plain Old Java Objects (POJOs) and how to design Java classes that adhered to these principles. They understood the role of POJOs in representing data structures or entities in a clean and self-contained manner. The tutorial also delved into effective database connection handling, emphasizing the secure opening and closing of connections. Students learned how to manage database connections in a way that ensured proper resource management.

Moreover, students gained hands-on experience in performing common database operations, including inserting, retrieving, updating, and deleting records in a MySQL database. They used SQL queries and Java to interact with the database, honing practical skills in data manipulation. The tutorial underscored the importance of error handling to ensure robust and reliable code. Students were equipped with the knowledge to handle exceptions and errors that may occur during database operations, contributing to the stability of their software applications.

Overall, this tutorial offered a comprehensive exploration of the integration of Java and MySQL databases, allowing students to apply theoretical knowledge in practical scenarios. It equipped them with the skills and best practices necessary for designing and implement-

ing secure systems that involve databases.