Module 1: Cheat Sheet: Fundamentals of Databases and JDBC

Package/ Method	Description	Code Example
Creating a table in a database	To create a table for storing book information, use the CREATE command.This creates a table named 'Books' with four columns: BookID, Title, Author, and Price. The PRIMARY KEY ensures that each book has a unique ID, and NOT NULL means the Title column cannot be empty.	`CREATE TABLE Books ( BookID INT PRIMARY KEY, Title VARCHAR(100) NOT NULL, Author VARCHAR(100), Price DECIMAL(10, 2) );`
Add a new column to the table	You can use the ALTER command to edit the books table.The 'ADD' command followed by the column name and data type adds the column to the table. In this code, we are adding an integer column called 'PublicationYear' to the 'Books' table.	`ALTER TABLE Books ADD PublicationYear INT;`
Remove a table	Use the DROP TABLE command to remove the table entirely.	`DROP TABLE Books;`
Add a new record	To add a new book, use the INSERT command and specify values for the BookID, Title, Author, and Price. This adds a new row to the 'Books' table.	`INSERT INTO Books (BookID, Title, Author, Price) VALUES (1, 'The Art of SQL', 'Stephane Faroult', 39.99);`
Update a record	If the price of a book changes, update it using the UPDATE command and use SET to specify the new price. The WHERE clause ensures that only the book with BookID 1 is updated.	`UPDATE Books SET Price = 29.99 WHERE BookID = 1;`
Delete a record	To delete a book, use the DELETE command and use the WHERE clause to specify the BookID.	`DELETE FROM Books WHERE BookID = 1;`
Retrieve records through filtering	To retrieve all books written by a specific author, use the SELECT command with the	`SELECT * FROM Books WHERE Author = 'Stephane Faroult';`

Package/ Method	Description	Code Example
	WHERE clause acting as a filter to specify the name of the author.	

Create a simple procedure	To create a simple procedure to calculate the total price of all books in the Books table, use the CREATE PROCEDURE command.	`CREATE PROCEDURE CalculateTotalPrice()`
		`BEGIN`
		`    SELECT SUM(Price) AS TotalPrice`
		`    FROM Books;`
		`END;`

Call a procedure	Call a procedure whenever you need it by using the CALL statement.	`CALL CalculateTotalPrice();`

Include a driver for a MySQL database	To include the driver for a MySQL database, add the 'MySQL Connector/J' dependency to your project.	`<dependency>`
		`    <groupId>mysql</groupId>`
		`    <artifactId>mysql-connector-java</artifactId>`
		`    <version>8.0.33</version>`
		`</dependency>`

Establish a connection to a MySQL database	In this code, the 'DriverManager.getConnection()' part establishes the connection using the database URL, username, and password. The 'try' command is part of the 'try-with-resources' statement which automatically closes the connection to prevent any resource leaks. When the 'try' block completes (successfully or due to an exception), the 'close()' method is automatically called on all resources declared in the parentheses. The 'catch' command is part of the 'try-catch' statement that handles exceptions and prevents abnormal termination of the program, making debugging easier.	` import java.sql.Connection;`
		`import java.sql.DriverManager;`
		`import java.sql.SQLException;`
		`public class DatabaseConnection {`
		`    public static void main(String[] args) {`
		`        String url = "jdbc:mysql://localhost:3306/bookstore";`
		`        String user = "root";`
		`        String password = "password";`
		`        try (Connection connection = DriverManager.getConnection(url, user, password)) {`
		`            if (connection != null) {`
		`                System.out.println("Connected to the database!");`
		`            }`
		`        } catch (SQLException e) {`
		`            System.out.println("Error connecting to the database");`
		`            e.printStackTrace();`
		`        }`
		`    }`
		`}`

Add a dependency code to create a connection pool	Use Apache Commons Database Connection Pooling (or DBCP to create a basic connection pool. Add a dependency code section to your pom.xml file.	`<dependency>`
		`    <groupId>org.apache.commons</groupId>`
		`    <artifactId>commons-dbcp2</artifactId>`
		`    <version>2.9.0</version>`
		`</dependency>`

Package/ Method	Description	Code Example

Java code needed to implement the connection pool	In this code, the 'BasicDataSource' part manages the connection pool. The 'setInitialSize()' part specifies the number of connections in the pool. The 'getConnection()' part retrieves a connection from the pool.	
		```
		import org.apache.commons.dbcp2.BasicDataSource;
		import java.sql.Connection;
		import java.sql.ResultSet;
		import java.sql.Statement;
		public class ConnectionPoolExample {
		    public static void main(String[] args) {
		        // Create a connection pool
		        BasicDataSource dataSource = new BasicDataSource();
		        dataSource.setUrl("jdbc:mysql://localhost:3306/bookstore");
		        dataSource.setUsername("root");
		        dataSource.setPassword("password");
		        dataSource.setInitialSize(5); // Number of initial connections in the pool
		        try (Connection connection = dataSource.getConnection();
		             Statement statement = connection.createStatement();
		             ResultSet resultSet = statement.executeQuery("SELECT * FROM Books")) {
		            while (resultSet.next()) {
		                System.out.println("Book Title: " + resultSet.getString("Title"));
		            }
		        } catch (Exception e) {
		            e.printStackTrace();
		        }
		    }
		}
		```

Use a PreparedStatement object to insert data	This code inserts a new book into the table with the title "New Book", the author "Author X", and price of "25.99." The SQL query includes placeholders (using a question mark) for values that are later replaced with actual data using the 'setString()' and 'setDouble()' methods. The 'executeUpdate()' method executes the INSERT command and returns the number of rows affected.	
		```
		PreparedStatement preparedStatement = connection.prepareStatement(
		  "INSERT INTO Books (Title, Author, Price) VALUES (?, ?, ?)");
		preparedStatement.setString(1, "New Book");
		preparedStatement.setString(2, "Author X");
		preparedStatement.setDouble(3, 25.99);
		int rowsAffected = preparedStatement.executeUpdate();
		System.out.println("Rows inserted: " + rowsAffected);
		```

Use a PreparedStatement object to update data	This code updates the book's price to "30.99." UPDATE modifies rows based on the condition in the WHERE clause. Reusing the same query structure for different parameter values improves efficiency.	
		```
		PreparedStatement preparedStatement = connection.prepareStatement(
		  "UPDATE Books SET Price = ? WHERE Title = ?");
		preparedStatement.setDouble(1, 30.99);
		preparedStatement.setString(2, "New Book");
		int rowsAffected = preparedStatement.executeUpdate();
		System.out.println("Rows updated: " + rowsAffected);
		```

Use a PreparedStatement object to query existing data	The SELECT query retrieves the data, and the parameters make the query dynamic. Iterating over the ResultSet allows you to process each row returned.	
		```
		PreparedStatement preparedStatement = connection.prepareStatement(
		  "SELECT * FROM Books WHERE Author = ?");
		preparedStatement.setString(1, "Author X");
		ResultSet resultSet = preparedStatement.executeQuery();
		while (resultSet.next()) {
		  System.out.println("Title: " + resultSet.getString("Title"));
		  System.out.println("Price: " + resultSet.getDouble("Price"));
		}
		END;
		```

Use a Callable Statement object using an input parameter	This is an example of using a CallableStatement object to call a stored procedure by using an input parameter. In this code, the procedure 'GetBooksByAuthor'	
		```
		 CREATE PROCEDURE GetBooksByAuthor(IN authorName VARCHAR(255))
		BEGIN
		  SELECT Title, Price FROM Books WHERE Author = authorName;
		END;
		CallableStatement callableStatement = connection.prepareCall("{CALL GetBooksByAuthor(?)}");
		callableStatement.setString(1, "Author X");
		```

Package/ Method	Description	Code Example

	takes an input parameter '(authorName)' and retrieves books written by that author. The setString(1, "Author X") method sets the value of input parameter 1, which is 'authorName' in the stored procedure, to "Author X".	`ResultSet resultSet = callableStatement.executeQuery();`
		`while (resultSet.next()) {`
		`  System.out.println("Title: " + resultSet.getString("Title"));`
		`  System.out.println("Price: " + resultSet.getDouble("Price"));`
		`}`

Use a Callable Statement object using input and output parameters	This is an example of using a CallableStatement object to call a stored procedure by using input and output parameters. In this code, the output parameters allow the stored procedure to return values to the Java application. The 'registerOutParameter()' method specifies the type of the output parameter, which in this example is java.sql.Types.DOUBLE.	`CREATE PROCEDURE GetTotalSales(OUT totalSales DOUBLE)`
		`BEGIN`
		`  SELECT SUM(Price) INTO totalSales FROM Books;`
		`END;`
		`CallableStatement callableStatement = connection.prepareCall("{CALL GetTotalSales(?)}");`
		`callableStatement.registerOutParameter(1, java.sql.Types.DOUBLE);`
		`callableStatement.execute();`
		`double totalSales = callableStatement.getDouble(1);`
		`System.out.println("Total Sales: " + totalSales);`

Use a PreparedStatement object to insert data and retrieve a generated key	In this code, the auto-generated keys are returned using 'getGeneratedKeys()', which is useful for tracking newly inserted records.	`PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "INSERT INTO Books (Title, Author, Price) VALUES (?, ?, ?)",`
		`  Statement.RETURN_GENERATED_KEYS);`
		`preparedStatement.setString(1, "Generated Key Book");`
		`preparedStatement.setString(2, "Author Y");`
		`preparedStatement.setDouble(3, 30.00);`
		`preparedStatement.executeUpdate();`
		`ResultSet generatedKeys = preparedStatement.getGeneratedKeys();`
		`if (generatedKeys.next()) {`
		`  int newBookID = generatedKeys.getInt(1);`
		`  System.out.println("Generated Book ID: " + newBookID);`
		`}`

Handle SQL exceptions	In this code, the 'try' and 'catch' statements catch any exceptions that happen during the 'try' block when creating the PreparedStatement and executing the query. If anything fails for any reason the exception is passed to the 'catch' block and a message appears on screen enabling the developer to troubleshoot and fix the issue. Catching exceptions ensures your application can gracefully handle errors such as invalid queries or missing tables.	`try {`
		`  PreparedStatement preparedStatement = connection.prepareStatement(`
		`    "SELECT * FROM NonExistentTable");`
		`  preparedStatement.executeQuery();`
		`} catch (SQLException e) {`
		`  System.err.println("SQL Error: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Connect to database before performing CRUD operations	In this code, we use several 'String' connection parameters, the first being the 'url,' which specifies the bookstore database's location,then 'user' and 'password' strings, which are used to authenticate the user, establishing the database connection. Then, the 'getConnection' method of the DriverManager class is used to create the connection and store it in a variable called 'connection' for future use.	`Connection connection = null;`
		`try {`
		`  String url = "jdbc:mysql://localhost:3306/bookstore";`
		`  String user = "root";`
		`  String password = "password";`
		`  connection = DriverManager.getConnection(url, user, password);`
		`  System.out.println("Connected to the database successfully!");`
		`} catch (SQLException e) {`
		`  System.err.println("Connection failed: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Package/ Method	Description	Code Example

Add a book to the inventory	This is an example of adding a new book to the inventory by providing information about the book's title, author, price, genre, publication date, and publisher. It uses a simple Statement object, which is easier to understand, but it's vulnerable to SQL injection. In this code, the 'try' block creates a simple statement. Then, the query is executed to insert a book into the Books table, with the specified values for the table columns. Further, 'execute Update' inserts the new row into the database. The number of rows affected is then printed out and as usual, any exception that occurs is passed into the 'catch' block.	`try (Statement statement = connection.createStatement()) {`
		`  String insertSQL = "INSERT INTO Books (Title, Author, Price, Genre, PublicationDate, Publi`
		`            "VALUES ('Advanced Java', 'Jane Doe', 45.99, 'Programming', '2023-01-10', 1)";`
		`  int rowsAffected = statement.executeUpdate(insertSQL);`
		`  System.out.println("Rows inserted: " + rowsAffected);`
		`} catch (SQLException e) {`
		`  System.err.println("Error inserting data: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Add a book to the inventory using a PreparedStatement object	In this code, the 'try' block creates a PreparedStatement object, using the 'connection dot prepare Statement' method.Then, the query is executed to insert a book into the Books table, using parameters as dynamic values instead of hard-coded specific values. These parameters are set using the set String, set Double, set Date, and set Int methods. Then, the 'execute Update' method on the 'preparedStatement' object, inserts the new row into the database. The number of rows affected is then printed out, and any exception that occurs is passed into the 'catch' block.	`try (PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "INSERT INTO Books (Title, Author, Price, Genre, PublicationDate, PublisherID) VALUES (?,`
		`  preparedStatement.setString(1, "Advanced Java");`
		`  preparedStatement.setString(2, "Jane Doe");`
		`  preparedStatement.setDouble(3, 45.99);`
		`  preparedStatement.setString(4, "Programming");`
		`  preparedStatement.setDate(5, java.sql.Date.valueOf("2023-01-10"));`
		`  preparedStatement.setInt(6, 1);`
		`  int rowsAffected = preparedStatement.executeUpdate();`
		`  System.out.println("Rows inserted: " + rowsAffected);`
		`} catch (SQLException e) {`
		`  System.err.println("Error inserting data: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Add new book using a stored procedure and a CallableStatement object	In this code, the 'try' block creates a CallableStatement object, using the 'connection dot prepare Call' method. Then, the query is executed to insert a book into the Books table, using parameters as dynamic values instead of hard-coded specific values. These parameters are set using the set String, set Double, set Date, and set Int methods. Then, the 'execute Update' method on the 'callableStatement' object, inserts the new row into the database. Any exception that occurs is passed into the 'catch' block. You can see that this CallableStatement is very similar to the PreparedStatement. The key difference is that CallableStatement is specifically designed to execute stored procedures, whereas PreparedStatement is used for executing parameterized SQL queries directly.	`CREATE PROCEDURE AddBook(`
		`  IN bookTitle VARCHAR(255),`
		`  IN bookAuthor VARCHAR(255),`
		`  IN bookPrice DOUBLE,`
		`  IN bookGenre VARCHAR(100),`
		`  IN publicationDate DATE,`
		`  IN publisherID INT`
		`)`
		`BEGIN`
		`  INSERT INTO Books (Title, Author, Price, Genre, PublicationDate, PublisherID)`
		`  VALUES (bookTitle, bookAuthor, bookPrice, bookGenre, publicationDate, publisherID);`
		`END;`
		`try (CallableStatement callableStatement = connection.prepareCall("{CALL AddBook(?, ?, ?, ?,`
		`  callableStatement.setString(1, "Advanced Java");`
		`  callableStatement.setString(2, "Jane Doe");`
		`  callableStatement.setDouble(3, 45.99);`
		`  callableStatement.setString(4, "Programming");`
		`  callableStatement.setDate(5, java.sql.Date.valueOf("2023-01-10"));`
		`  callableStatement.setInt(6, 1);`
		`  callableStatement.execute();`
		`  System.out.println("Book added successfully using stored procedure.");`
		`} catch (SQLException e) {`
		`  System.err.println("Error calling stored procedure: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Retrieve all books using a simple Statement	In this code, the statement query is executed using 'SELECT asterisk FROM Books', and the results are traversed and printed out. As before a 'catch' block captures any exceptions that occur.	`try (Statement statement = connection.createStatement();`
		`   ResultSet resultSet = statement.executeQuery("SELECT * FROM Books")) {`
		`  while (resultSet.next()) {`
		`    System.out.println("Book ID: " + resultSet.getInt("BookID"));`
		`    System.out.println("Title: " + resultSet.getString("Title"));`
		`    System.out.println("Author: " + resultSet.getString("Author"));`
		`    System.out.println("Price: " + resultSet.getDouble("Price"));`
		`  }`
		`} catch (SQLException e) {`

Package/ Method	Description	Code Example

		`System.err.println("Error retrieving data: " + e.getMessage());`
		`e.printStackTrace();`
		`}`

Retrieve all books of a specific genre using PreparedStatement	This code creates and uses a PreparedStatement object using the 'connection dot prepare Statement' method, and it retrieves all books from the 'Programming' genre, by using 'SELECT asterisk FROM Books' with the 'WHERE Genre equals question mark' clause.	`try (PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "SELECT * FROM Books WHERE Genre = ?")) {`
		`  preparedStatement.setString(1, "Programming");`
		`  ResultSet resultSet = preparedStatement.executeQuery();`
		`  while (resultSet.next()) {`
		`    System.out.println("Title: " + resultSet.getString("Title"));`
		`    System.out.println("Price: " + resultSet.getDouble("Price"));`
		`  }`
		`} catch (SQLException e) {`
		`  System.err.println("Error retrieving data: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Pagination example using PreparedStatement	The query used in this code is 'SELECT asterisk FROM Books' but it also specifies 'ORDER BY Price' and in descending order with a maximum limit of only the top 5 most expensive books.It prints out the results and uses 'try' and 'catch' blocks for SQL exceptions.	`try (PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "SELECT * FROM Books ORDER BY Price DESC LIMIT 5 OFFSET 0")) {`
		`  ResultSet resultSet = preparedStatement.executeQuery();`
		`  while (resultSet.next()) {`
		`    System.out.println("Title: " + resultSet.getString("Title"));`
		`    System.out.println("Price: " + resultSet.getDouble("Price"));`
		`  }`
		`} catch (SQLException e) {`
		`  System.err.println("Error with pagination: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Change the price of a book using PreparedStatement	In this code, the UPDATE command sets the new price and uses the WHERE clause to filter on the title.	`try (PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "UPDATE Books SET Price = ? WHERE Title = ?")) {`
		`  preparedStatement.setDouble(1, 49.99);`
		`  preparedStatement.setString(2, "Advanced Java");`
		` int rowsUpdated = preparedStatement.executeUpdate();`
		`  System.out.println("Rows updated: " + rowsUpdated);`
		`} catch (SQLException e) {`
		`  System.err.println("Error updating data: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Remove all books of a specific genre using PreparedStatement	In this code, the DELETE command removes all the books of the 'Programming' genre from the Books table by using the 'WHERE Genre equals question mark' clause to filter on the genre and specifying the 'Programming' genre with a set String method.	`try (PreparedStatement preparedStatement = connection.prepareStatement(`
		`  "DELETE FROM Books WHERE Genre = ?")) {`
		`  preparedStatement.setString(1, "Programming");`
		`  int rowsDeleted = preparedStatement.executeUpdate();`
		`  System.out.println("Rows deleted: " + rowsDeleted);`
		`} catch (SQLException e) {`
		`  System.err.println("Error deleting data: " + e.getMessage());`
		`  e.printStackTrace();`
		`}`

Package/ Method	Description	Code Example

Author(s)

Upkar Lidder
Sangeeta Srinivasan