JDBC and Oracle Database Management System

Java Database Connectivity (JDBC) is an application programming interface (API) that defines how a client may access a database. It provides methods for querying and updating data in a database and is oriented towards relational databases.

JDBC Architecture

JDBC architecture consists of two layers:

1. **JDBC API**: This provides the application-to-JDBC Manager connection.
2. **JDBC Driver API**: This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

JDBC Drivers

JDBC drivers implement the defined interfaces in the JDBC API for interacting with a database server.

There are 4 types of JDBC drivers:

1. **JDBC-ODBC Bridge Driver**
2. **Native-API Driver (Partially Java Driver)**
3. **Network Protocol Driver (Fully Java Driver)**
4. **Thin Driver (Fully Java Driver)**

For Oracle Database, the Thin Driver is recommended as it is written entirely in Java, and extends the functionality provided by the other drivers.

Connecting to Oracle Database using JDBC

To connect to an Oracle Database, a client would need the following details:

- Driver class: The driver class for the Oracle database is `oracle.jdbc.driver.OracleDriver`.
- Connection URL: The connection URL for the Oracle database is `jdbc:oracle:thin:@hostname:port number:database name`.

Executing a SELECT operation in JAVA JDBC

Suppose there is a table `student` with the following columns,

```
CREATE TABLE student (
    Name VARCHAR2(20),
    Roll NUMBER PRIMARY KEY,
    Semester NUMBER,
    CGPA NUMBER(3,2)
);
```

Now let's have some data here,

```
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Abdullah', 101, 1, 3.50);
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Fatima', 102, 1, 3.25);
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Rahim', 103, 2, 3.70);
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Ayesha', 104, 2, 3.60);
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Kamal', 105, 3, 3.80);
INSERT INTO student (Name, Roll, Semester, CGPA) VALUES ('Sadia', 106, 3, 3.90);
```

Now suppose I want to retrieve information from this table. And run the query,

```
SELECT * FROM Students
```

But in the JAVA console, I want to see the output. There comes JDBC. The JAVA program looks like,

```
import java.sql.*;

public class main {
    public static void main(String[] args) {
        String Username = "Your_Username";
        String Password = "Your_Password";
        String URL = "jdbc:oracle:thin:@localhost:1521:xe";

        String Query = "SELECT * FROM Students";

        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection connection = DriverManager.getConnection(URL, Username, Password);

            Statement stmt = connection.createStatement();

            ResultSet result = stmt.executeQuery(Query);

            while(result.next()) {
                String name = result.getString("Name");
                String roll = result.getString("Roll");
                String semester = result.getString("Semester");
                String cgpa = result.getString("CGPA");
                System.out.println(name + " " + roll + " " + semester + " " + cgpa);
            }

            stmt.close();
            connection.close();
        }
        catch(SQLException e){
            System.out.println("Connection Failed : " + e.getMessage());
        }
        catch(ClassNotFoundException e){
            System.out.println("Driver Not Found : " + e.getMessage());
        }
    }
}
```

Let’s break down the provided Java code which uses JDBC to connect to an Oracle database and fetch data from the ‘Students’ table.

Importing Necessary Libraries

```
import java.sql.*;
```

This line imports the `java.sql` package which contains the JDBC API. All the classes and interfaces for JDBC reside in this package.

Main Class and Method

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

This is the starting point of the Java application. The `main` method is the entry point when the class is started from the command line.

Database Connection Details

```
String Username = "Your_Username";
String Password = "Your_Password";
String URL = "jdbc:oracle:thin:@localhost:1521:xe";
```

These lines define the credentials and URL for the database connection. Replace `"Your_Username"` and `"Your_Password"` with your actual database username and password. The URL specifies the database server’s address (`localhost`), port number (`1521`), and database name (`xe`).

SQL Query

```
String Query = "SELECT * FROM Students";
```

This line defines the SQL query to be executed. Here, it’s fetching all records from the ‘Students’ table.

Establishing Connection and Fetching Data

```
try{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection connection = DriverManager.getConnection(URL, Username, Password);
```

The `Class.forName()` method is used to load the JDBC driver class. The `DriverManager.getConnection()` method is used to establish a connection to the database.

```
Statement stmt = connection.createStatement();
ResultSet result = stmt.executeQuery(Query);
```

A `Statement` object is created from the `Connection` object. The `executeQuery()` method of the `Statement` object is used to execute the SQL query. The result is stored in a `ResultSet` object.

Processing the Result

```
while(result.next()) {
    String name = result.getString("Name");
    String roll = result.getString("Roll");
    String semester = result.getString("Semester");
    String cgpa = result.getString("CGPA");

    System.out.println(name + " " + roll + " " + semester + " " + cgpa);
}
```

The `next()` method of the `ResultSet` object is used to iterate through each row of the result. The `getString()` method is used to fetch the column values for each row.

Closing the Connection

```
stmt.close();
connection.close();
```

These lines close the `Statement` and `Connection` objects to free up database resources.

Handling Exceptions

```
catch(SQLException e){
    System.out.println("Connection Failed : " + e.getMessage());
}
catch(ClassNotFoundException e){
    System.out.println("Driver Not Found : " + e.getMessage());
}
```

These `catch` blocks handle any `SQLException` or `ClassNotFoundException` that may occur during the execution of the code. If an exception occurs, an error message is printed to the console.

That's it! This code connects to an Oracle database, executes a SQL query, and processes the result. It's a basic example of how to use JDBC in a Java application.

JDBC DDL operations

Creating a Table in JDBC

In JDBC, you can execute any SQL statement using the `execute()` method of the `Statement` object. Here's an example of how to create a table:

```
String createTableSQL = "CREATE TABLE Students("
        + "ID INT PRIMARY KEY, "
        + "NAME VARCHAR(20), "
        + "AGE INT, "
        + "GENDER VARCHAR(20))";

Statement stmt = connection.createStatement();
stmt.execute(createTableSQL);
```

In this example, a `CREATE TABLE` SQL statement is defined as a string. This statement creates a new table named 'Students' with four columns: 'ID', 'NAME', 'AGE', and 'GENDER'. The `execute()` method of the `Statement` object is then used to execute this SQL statement.

Altering a Table in JDBC

You can also alter a table in JDBC using the `execute()` method of the `Statement` object. Here's an example of how to add a new column to a table:

```
String alterTableSQL = "ALTER TABLE Students ADD EMAIL VARCHAR(50)";

Statement stmt = connection.createStatement();
stmt.execute(alterTableSQL);
```

In this example, an `ALTER TABLE` SQL statement is defined as a string. This statement adds a new column named ‘EMAIL’ to the ‘Students’ table. The `execute()` method of the `Statement` object is then used to execute this SQL statement.

JDBC DML operations

Select Operation in JDBC

The `SELECT` operation is used to retrieve data from the database. Here’s how you can perform a `SELECT` operation:

```
String selectSQL = "SELECT * FROM student";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(selectSQL);

while(rs.next()) {
    String name = rs.getString("Name");
    int roll = rs.getInt("Roll");
    int semester = rs.getInt("Semester");
    double cgpa = rs.getDouble("CGPA");

    System.out.println(name + " " + roll + " " + semester + " " + cgpa);
}
```

In this example, a `SELECT` SQL statement is defined as a string. The `executeQuery()` method of the `Statement` object is then used to execute this SQL statement. The result is stored in a `ResultSet` object, which is then iterated over to retrieve and print each record.

Insert Operation in JDBC

The `INSERT` operation is used to insert data into the database. Here’s how you can perform an `INSERT` operation:

```
String insertSQL = "INSERT INTO student (Name, Roll, Semester, CGPA)
VALUES ('John Doe', 1, 2, 3.5)";
Statement stmt = connection.createStatement();
int rowsAffected = stmt.executeUpdate(insertSQL);

System.out.println(rowsAffected + " row(s) inserted.");
```

In this example, an `INSERT` SQL statement is defined as a string. The `executeUpdate()` method of the `Statement` object is then used to execute this SQL statement. The number of affected rows is returned and printed.

Update Operation in JDBC

The `UPDATE` operation is used to update existing data in the database. Here's how you can perform an `UPDATE` operation:

```
String updateSQL = "UPDATE student SET Name = 'Jane Doe' WHERE Roll =
1";
Statement stmt = connection.createStatement();
int rowsAffected = stmt.executeUpdate(updateSQL);

System.out.println(rowsAffected + " row(s) updated.");
```

In this example, an `UPDATE` SQL statement is defined as a string. The `executeUpdate()` method of the `Statement` object is then used to execute this SQL statement. The number of affected rows is returned and printed.

Delete Operation in JDBC

The `DELETE` operation is used to delete data from the database. Here's how you can perform a `DELETE` operation:

```
String deleteSQL = "DELETE FROM student WHERE Roll = 1";
Statement stmt = connection.createStatement();
int rowsAffected = stmt.executeUpdate(deleteSQL);

System.out.println(rowsAffected + " row(s) deleted.");
```

In this example, a `DELETE` SQL statement is defined as a string. The `executeUpdate()`

method of the `Statement` object is then used to execute this SQL statement. The number of affected rows is returned and printed.

Prepared Statement

A **Prepared Statement** is a feature used in Java Database Connectivity (JDBC). It's a statement that is compiled by the database once and can be executed many times. The main advantage of using prepared statements is to prevent SQL Injection attacks.

How to use Prepared Statements in JDBC?

Here is a step-by-step guide on how to use prepared statements in JDBC:

1. **Load the driver**: The first step is to load the JDBC driver using the `Class.forName()` method.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2. **Create a connection**: After loading the driver, establish a connection to the database using the `DriverManager.getConnection()` method.

```
Connection connection = DriverManager.getConnection(URL, Username, Password);
```

3. **Create a Prepared Statement**: Once the connection is established, create a prepared statement using the `Connection.prepareStatement()` method.

```
PreparedStatement stmt = con.prepareStatement("insert into student values(?,?,?,?)");
```

4. **Set the parameters**: After creating the prepared statement, set the parameters using the appropriate setter methods like `setString()`, `setInt()`, etc.

```
stmt.setString(1,"John");
stmt.setInt(2,101);
stmt.setInt(3,1);
stmt.setFloat(4,9.8f);
```

5. **Execute the statement**: After setting the parameters, execute the statement using the `executeUpdate()` method.

```
int i = stmt.executeUpdate();
System.out.println(i + " records inserted");
```

6. **Close the connection**: Finally, close the connection using the `Connection.close()` method.

```
con.close();
```

Why use Prepared Statements?

Prepared Statements are used for different reasons:

- **Efficiency**: Prepared Statements are parsed and compiled only once by the database. This makes them more efficient when executing the same SQL statement multiple times.
- **Security**: They help prevent SQL Injection attacks as they automatically escape the special characters.
- **Readability**: They make the code more readable and easier to understand.

Example for Prepared Statement

```
import java.sql.*;

public class main {
    public static void main(String[] args) {
        String Username = "Your_Username";
        String Password = "Your_Password";
        String URL = "jdbc:oracle:thin:@localhost:1521:xe";

        try{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection connection = DriverManager.getConnection(URL, Username, Password);

            // Create a Prepared Statement
            PreparedStatement stmt = connection.prepareStatement("INSERT INTO student
                                     VALUES (?, ?, ?, ?)");

            // Set the parameters
            stmt.setString(1, "John");
            stmt.setInt(2, 101);
            stmt.setInt(3, 1);
            stmt.setFloat(4, 9.8f);

            // Execute the statement
            int i = stmt.executeUpdate();
            System.out.println(i + " records inserted");

            stmt.close();
            connection.close();
        }
        catch(SQLException e){
            System.out.println("Connection Failed : " + e.getMessage());
        }
        catch(ClassNotFoundException e){
            System.out.println("Driver Not Found : " + e.getMessage());
        }
    }
}
```

The code above has,

- **Database Connection Details**: The code starts by defining the username, password, and URL for the database connection. These are the credentials you need to connect to your Oracle database.
- **Try-Catch Block**: The main part of the code is wrapped in a try-catch block. This is used to catch and handle any exceptions (errors) that might occur when connecting to the database or executing the SQL statement.
- **Loading the Driver and Establishing Connection**: Inside the try block, the code first loads the Oracle JDBC driver and then establishes a connection to the Oracle database using the previously defined credentials.
- **Creating a Prepared Statement**: Once the connection is established, the code creates a prepared statement for inserting a record into the `student` table.
- **Setting Parameters and Executing the Statement**: The code then sets the parameters for the prepared statement (the values to be inserted into the table) and executes the statement. This inserts a new record into the `student` table.
- **Closing the Connection**: After the record has been inserted, the code closes the connection to the database.
- **Handling Exceptions**: If an exception occurs at any point (for example, if the database connection fails), the catch block will catch the exception and print an error message.

Retrieving MetaData of a table from oracle JDBC

```java
import java.sql.*;
public class task_3 {
    public static void main(String args[]) {
        String Username = "your_username";
        String Password = "your_password";
        String URL = "jdbc:oracle:thin:@localhost:1521:xe";

        String Query = "SELECT * FROM Student";



        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection connection = DriverManager.getConnection(URL, Username, Password);


            Statement statement = connection.createStatement();
            ResultSet rs = statement.executeQuery(Query);

            ResultSetMetaData rsmd = rs. getMetaData () ;
            for ( int i = 1; i ≤ rsmd . getColumnCount () ; i ++) {
                System . out . println ( rsmd . getColumnName (i) + " - " + rsmd .
                                         getColumnTypeName (i)) ;
            }
        }
        catch (SQLException e) {
            System.out.println(e);
        }
        catch (ClassNotFoundException e) {
            System.out.println(e);
        }
    }
}
```

Explaination

1. **Establish a Connection**: Connect to the Oracle database using the JDBC driver. This requires the database URL, username, and password.

```java
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection connection = DriverManager.getConnection(URL, Username, Password);
```

2. **Create a Statement**: Once the connection is established, create a statement object using the `createStatement()` method of the Connection object.

```java
Statement statement = connection.createStatement();
```

3. **Execute a Query**: Use the `executeQuery()` method of the Statement object to run a SQL query that selects all records from the table. This returns a ResultSet object.

```
ResultSet rs = statement.executeQuery(Query);
```

4. **Retrieve Metadata**: Get the metadata of the ResultSet using the `getMetaData()` method. This returns a ResultSetMetaData object.

```
ResultSetMetaData rsmd = rs.getMetaData();
```

5. **Access Metadata Information**: Use the methods of the ResultSetMetaData object to access the metadata information. For example, `getColumnCount()` returns the number of columns in the ResultSet, and `getColumnName(i)` and `getColumnTypeName(i)` return the name and type of the i-th column, respectively.

```
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i) + " - " +
rsmd.getColumnTypeName(i));
}
```

6. **Handle Exceptions**: Throughout the process, use try-catch blocks to handle any exceptions that might occur, such as SQLException and ClassNotFoundException.

```
catch (SQLException e) {
    System.out.println(e);
}
catch (ClassNotFoundException e) {
    System.out.println(e);
}
```

In the provided code, these steps are implemented to retrieve and print the names and types of all columns in the `Student` table.

Output

```
Name - VARCHAR2
Roll - NUMBER
Semester - NUMBER
CGPA - NUMBER
```