

Day 13 - Facilitation Guide

(JDBC)

Index

- I. Introduction to Java Database Connectivity (JDBC)
- II. Connecting to MySql from Java
- III. Executing SQL Queries from Java

For (1.5 hrs) ILT

In the previous session, we explored advanced SQL queries, foreign key, joining tables and the different ways to filter data from databases.

We covered a variety of important concepts:

Foreign Key:

A field in a database table linking two tables, establishing a relationship. Typically refers to the primary key of another table, ensuring data consistency. For instance, in a library database, "Books" may have a foreign key "AuthorID" linked to the "Authors" table's primary key.

Filtering Data:

Selecting specific records from a table based on criteria. Done with the SELECT statement and WHERE clause. Uses operators like =, >, <, etc., and logical operators (AND, OR, NOT) to combine conditions.

Advanced SQL Queries:

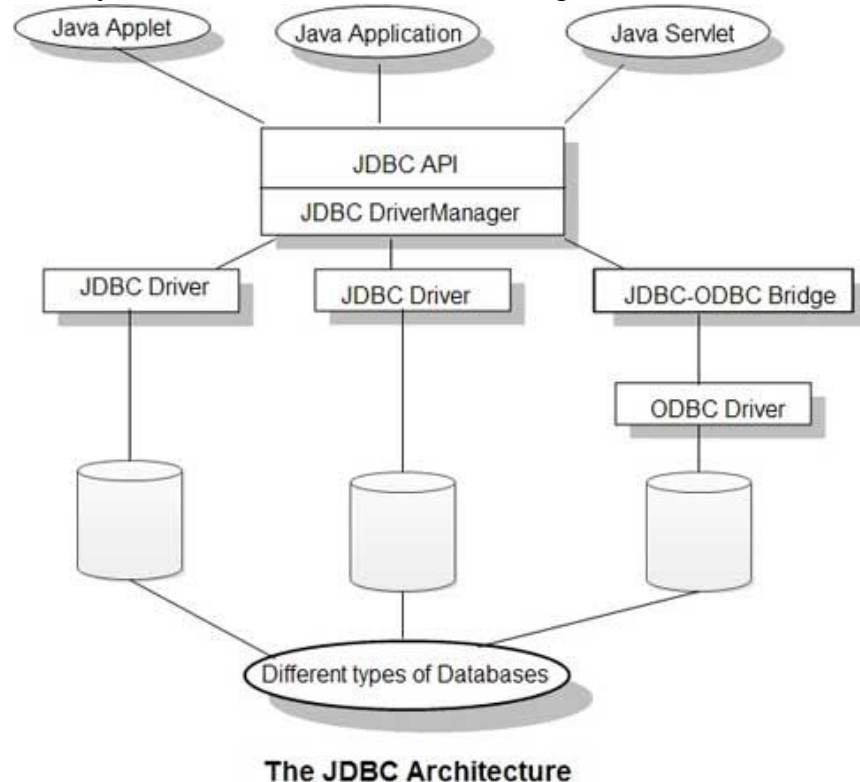
Go beyond basic SELECT and WHERE clauses. Include joins (INNER, LEFT, RIGHT, FULL OUTER) for combining data from multiple tables. Aggregation functions (SUM, AVG, COUNT, etc.) with GROUP BY for calculations on groups.

In this session, we will explore JDBC, which involves establishing a connection between Java and MySQL, as well as executing SQL queries using Java.

I. Introduction to JDBC

JDBC (Java Database Connectivity) is a Java-based framework and API (Application Programming Interface) that allows Java applications to interact with relational databases. It provides a standard interface for Java applications to connect to, query,

and manipulate data stored in various relational database management systems (RDBMS) such as MySQL, Oracle, SQL Server, PostgreSQL, and more.



Note: The full form of API is Application Programming Interface. It is a document which gives you the list of all the packages, classes, and interfaces, along with their fields and methods. Using these API's, the programmer can know how to use the methods, fields, classes, interfaces provided by Java libraries.

Before delving deeply into JDBC, it's essential to understand why there is a need for JDBC.

Java Database Connectivity (JDBC) is required for several reasons in Java applications:

- **Database Interaction:** JDBC provides a standardized and efficient way for Java applications to interact with relational databases. It allows applications to connect to databases, execute SQL queries, and retrieve, manipulate, and update data stored in the database.
- **Database Portability:** JDBC allows developers to write database-agnostic code. Java applications can use the same JDBC code to connect to and work with various relational database systems (e.g., MySQL, Oracle, SQL Server) without significant changes, promoting platform independence and code reusability.

- **Dynamic Data Retrieval:** With JDBC, Java applications can dynamically retrieve and manipulate data from databases based on user input or changing business requirements. This dynamic data access is essential for building flexible and responsive applications.
- **Database Transactions:** JDBC supports database transactions, allowing applications to group multiple database operations into a single transaction. This ensures data consistency and integrity by either committing all changes or rolling them back if an error occurs.

In summary, JDBC is required in Java applications to bridge the gap between the Java programming language and relational databases, enabling applications to store, retrieve, and manage data effectively. It provides a standardized, portable, and efficient means of database interaction, making it a fundamental component for building data-driven and dynamic Java applications.

Now let's understand the Key features and functionalities of JDBC:

- **Database Connectivity:** JDBC provides a way for Java applications to establish connections to databases, allowing them to communicate with and access data stored in those databases.
- **Database Driver:** JDBC requires a database-specific driver to be loaded into the Java application. These drivers act as a bridge between Java and the database, enabling the application to communicate with the database using the appropriate protocol.
- **Data Access:** JDBC allows Java programs to execute SQL (Structured Query Language) statements and retrieve, insert, update, and delete data in a database.
- **Transaction Management:** It supports transaction management, enabling applications to group multiple SQL operations into a single transaction and control their execution.

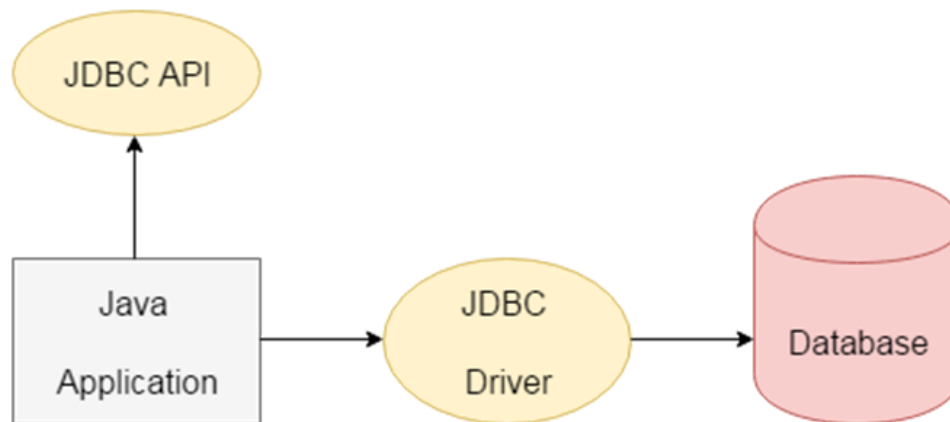
Common JDBC Components

The JDBC API offers a set of interfaces and classes. Now, let's go through each one individually, starting with Drivers and the Connection interface.

1. JDBC Drivers

A JDBC driver is a JDBC API implementation used for connecting to a particular type of database. JDBC (Java Database Connectivity) drivers are essential components for connecting Java applications to databases.

JDBC drivers serve as a crucial link between Java applications and databases, providing a standardized and efficient means of interacting with various DBMSs. They simplify the database access process, enhance code portability, and enable developers to build robust, database-driven applications in Java.

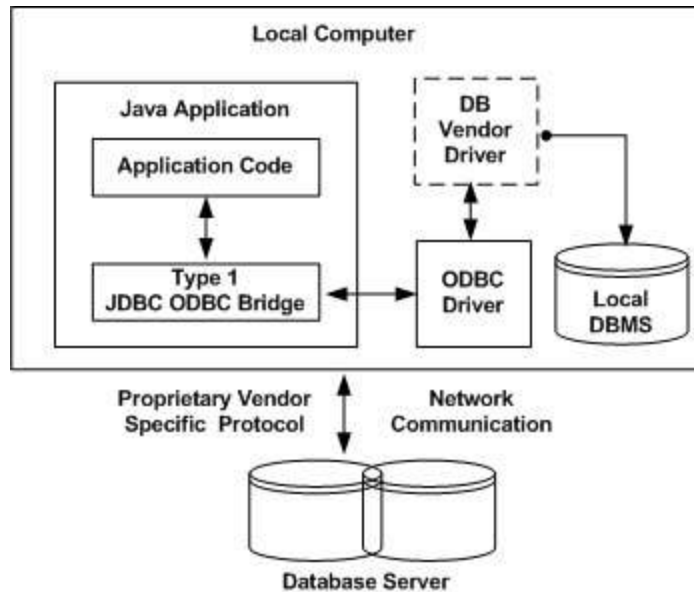


Let's now examine the various categories of drivers.

Types of JDBC drivers:

- **Type 1** – In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

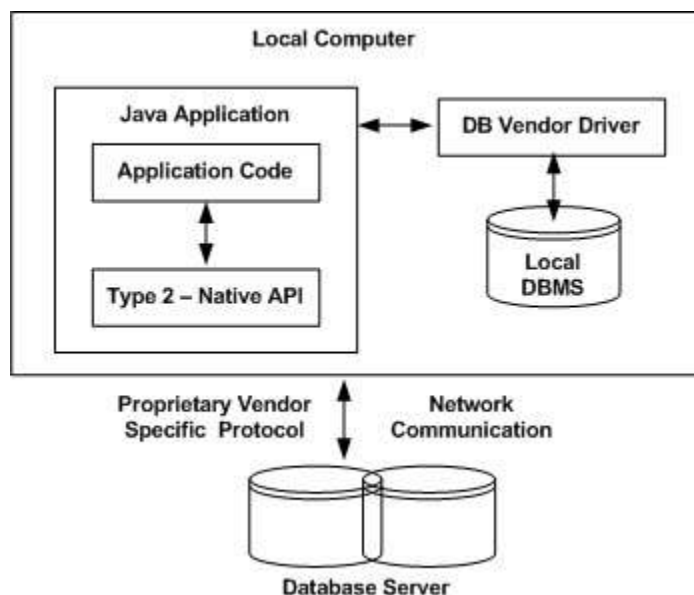
When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

- **Type 2** – In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

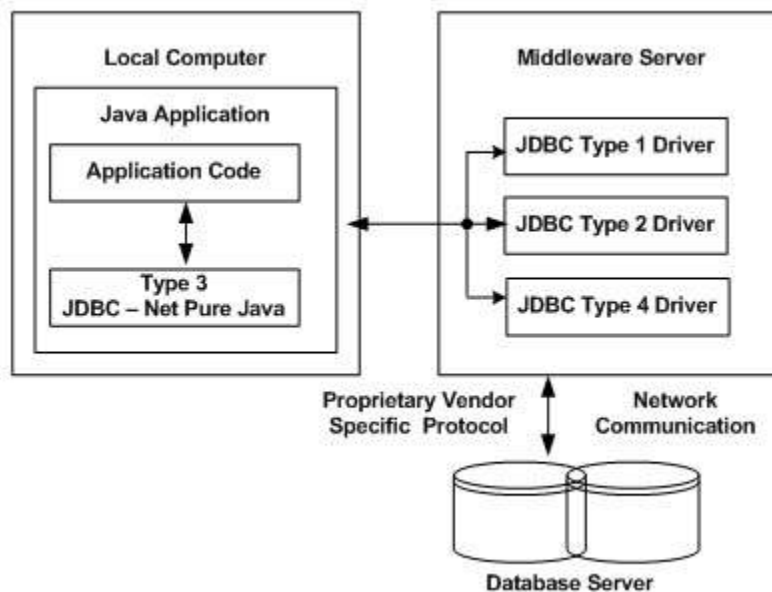
If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.



The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

- **Type 3** – In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

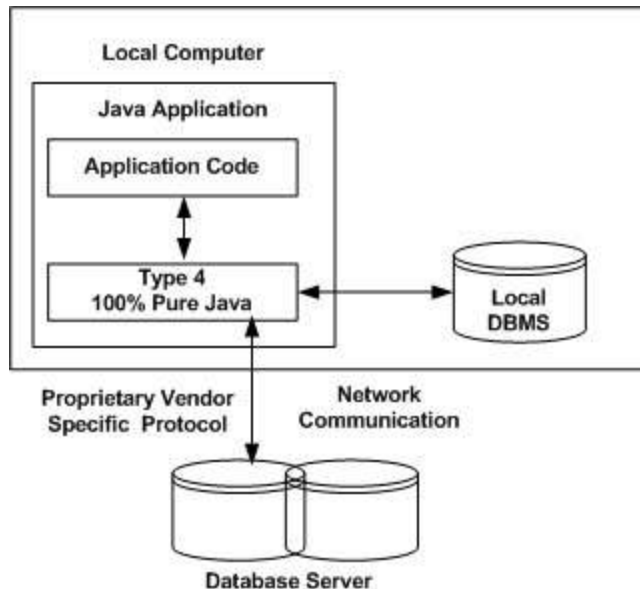
This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

- **Type 4** – In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

The most commonly used type is type 4, as it has the advantage of being platform-independent. Connecting directly to a database server provides better performance compared to other types. The downside of this type of driver is that it's database-specific – given each database has its own specific protocol.

2. Connection – A Connection is the session between java application and database. The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provides many methods for transaction management like commit(), rollback() etc.

Commonly used methods of Connection interface:

public Statement createStatement(): Creates a statement object that can be used to execute SQL queries.

public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

public void setAutoCommit(boolean status): It is used to set the commit status.By default it is true.

public void commit(): It saves the changes made since the previous commit/rollback permanent.

public void rollback(): Drops all changes made since the previous commit/rollback.

public void close(): Closes the connection and Releases a JDBC resources immediately.

3. DriverManager Class

The DriverManager class in JDBC (Java Database Connectivity) is a core component of the JDBC API provided by Java. It plays a crucial role in managing database connections and facilitating communication between Java applications and databases.

Here are some of the key methods of the DriverManager class:

- **registerDriver(Driver driver):** Registers a new database driver with the DriverManager. This method is rarely used because most JDBC drivers can automatically register themselves when their JAR files are loaded.
deregisterDriver(Driver driver):
- **deregisterDriver(Driver driver):** Deregisters a previously registered driver. This method is also not commonly used as drivers can typically be unloaded by garbage collection when no longer needed.
getConnection(String url):
- **getConnection(String url):** Attempts to establish a database connection using the specified database URL.
This is one of the most commonly used methods of DriverManager for creating database connections.
- **getConnection(String url, Properties info):** Similar to the previous method, but allows you to pass a Properties object containing additional connection properties.
- **getConnection(String url, String user, String password):** Establishes a database connection with the specified URL, username, and password.
- **getDrivers():** Returns an Enumeration of all the currently registered JDBC drivers. Useful for listing available drivers or diagnosing driver-related issues.

II. Connecting to MySql from Java

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

Here are the step-by-step instructions for connecting to a MySQL database from a Java application using JDBC:

Step 1: Download and Install MySQL Connector/J

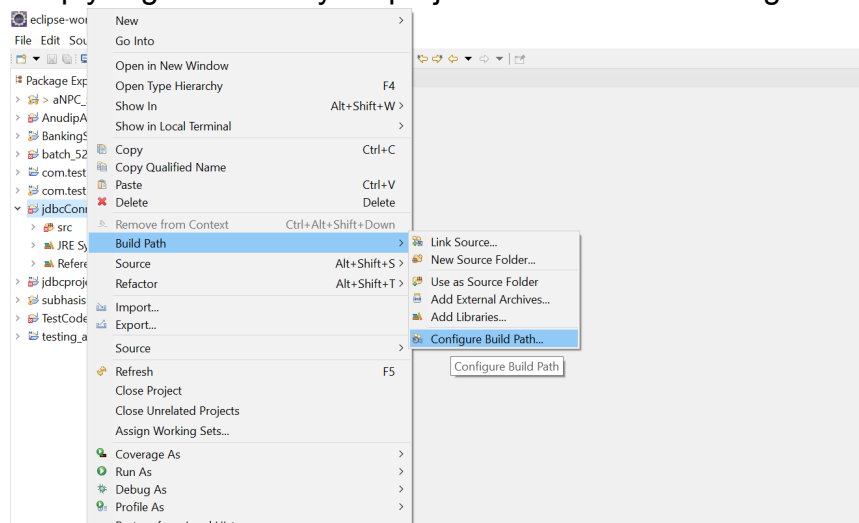
Before you can connect to MySQL from Java, you need to download the MySQL Connector/J driver. You can download it from the official MySQL website: [MySQL Connector/J Downloads](#).

Step 2: Include the JDBC Driver in Your Project

Add the downloaded JDBC driver JAR file to your Java project. You can do this by adding it to your project's classpath. In most integrated development environments (IDEs), you can right-click your project, go to "Build Path" or "Libraries," and add the JAR file.

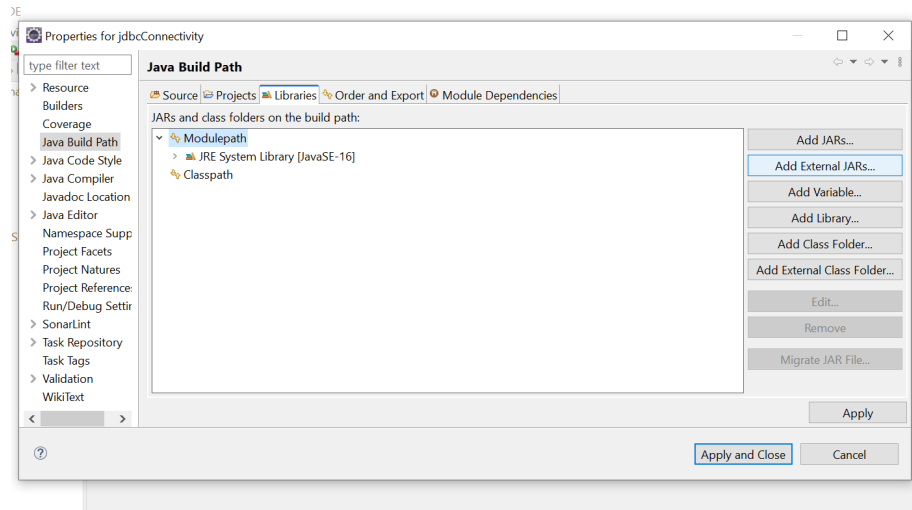
For example, to set the path in Eclipse

Simply Right Click on your project->Build Path->Configure Build Path

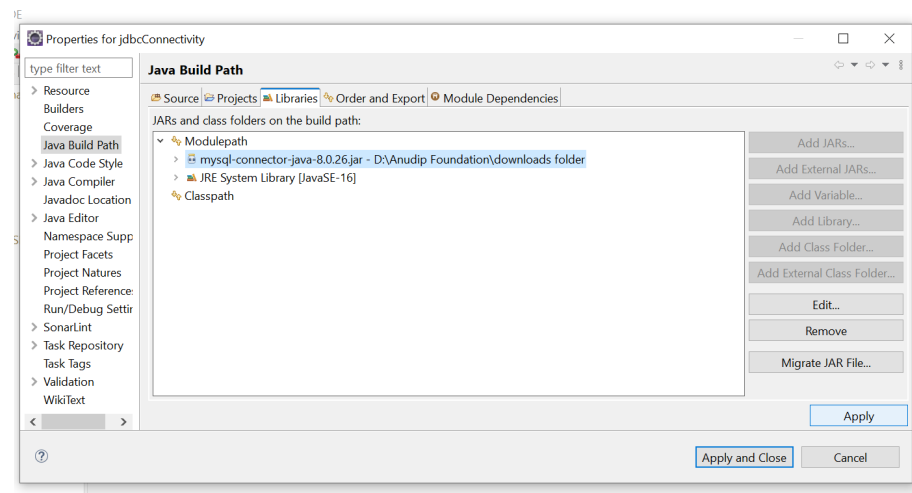


This will open the Properties of jdbcConnectivity dialog box as under

Select Java Build Path ->ModulePath->Add External Jars ->select your mysql Connector file from the path where you have downloaded it.



Now Click on Apply Button. The jar will be added as shown below



Next, let's register the driver using the `Class.forName()` method, which dynamically loads the driver class:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

In Java, the `Class` class is a special class provided by the Java runtime environment (JRE) that represents a class or interface during runtime. It's part of the Java Reflection API, which allows you to inspect and manipulate classes, methods, fields, and other elements of a Java program dynamically at runtime.

forName() method

The `forName()` method in Java is a static method provided by the `java.lang.Class` class, and it is commonly used for dynamic class loading. This method is primarily used to obtain a `Class` object for a given class name at runtime.

Step 3: Import the Necessary Packages

In your Java code, import the necessary JDBC packages at the beginning of your class file. These packages include:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

Step 4: Creating the Connection

To open a connection, we can use the *getConnection()* method of the *DriverManager* class. This method requires a connection URL *String* parameter:

To open a connection, we can use the *getConnection()* method of the *DriverManager* class. This method requires a connection URL *String* parameter:

```
try {
    Connection connection = DriverManager.getConnection(jdbcUrl, username,
password);
    // Connection established successfully
} catch (SQLException e) {
    e.printStackTrace();
    // Handle connection errors here
}
```

Step 5: Use the Database Connection

Once the connection is established, you can use it to perform various database operations such as executing SQL queries, inserting, updating, or retrieving data from the database.

Here's an example of executing a simple query:

```
try {
    Connection connection = DriverManager.getConnection(jdbcUrl, username,
password);

    // Create and execute a SQL query
    String sqlQuery = "SELECT * FROM your_table_name";
    // ...

    // Close the connection when done
    connection.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

```
    // Handle SQL errors here  
}
```

Note: Since the Connection is an AutoCloseable resource, we should use it inside a try-with-resources block. So we don't need to use the finally block to close the connection.

Let's look into the practical demonstration:

Note: Here We have created a separate class to connect with the database to achieve reusability.

```
package jdbcConnectivity;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
  
public class ConnectDB {  
    static Connection con=null;  
  
    public static Connection dbConnect()  
    {  
  
        try {  
            //Register the driver  
            Class.forName("com.mysql.cj.jdbc.Driver");  
  
            //established the connection  
  
            con=DriverManager.getConnection("jdbc:mysql://localhost:3306/studentmanagementsystem", "root", "mysql");  
  
        }catch (Exception e) {  
            System.out.println(e);  
        }  
  
        return con;  
    }  
}
```

Let's check if the Connection is established or not by calling the dbConnect() method from the ConnectDB class.

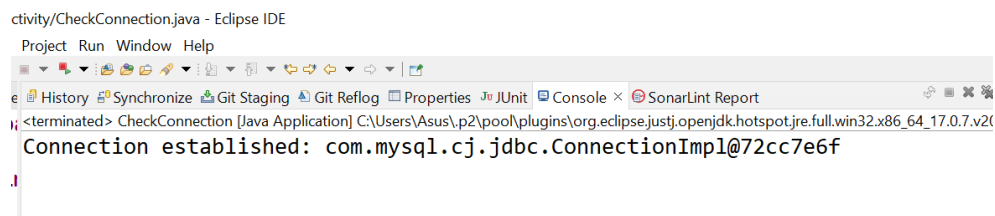
```
package jdbcConnectivity;
```

```

import java.sql.Connection;
public class CheckConnection {
    public static void main(String[] args) {
        try(Connection con=ConnectDB.dbConnect();
            )
        {
            System.out.println("Connection established: "+con);
        }catch (Exception e) {
            // TODO: handle exception
        }
    }
}

```

Output:



Question: What are the different types of JDBC drivers, and when would you choose one over the other?

III. Executing SQL Queries from Java

Once a connection is obtained we can interact with the database. The JDBC Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

To run SQL queries in Java, it's essential to grasp the fundamentals of the Statement interface.

Statement Interface:

The statement interface is used to create SQL basic statements in Java, it provides methods to execute queries with the database.

There are different types of statements that are used in JDBC as follows:

1. Statement- Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.

2. Prepared Statement- Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.

3. Callable Statement- Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Note: In this session, we will learn only Statement Interface.

Statement Interface: From the connection interface, you can create the object for this interface. It is generally used for general-purpose access to databases and is useful while using static SQL statements at runtime.

Syntax:

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

Implementation: Once the Statement object is created, there are three ways to execute it.

Here are the implementation:

boolean execute(String SQL): If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute SQL DDL statements or for dynamic SQL.

int executeUpdate(String SQL): Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.

ResultSet executeQuery(String SQL): Returns a ResultSet object. Used similarly as SELECT is used in SQL.

Note: After gaining practical experience with the Statement interface, we will delve into understanding the ResultSet interface.

Let's execute the query Step by step:

Step 1: Import JDBC Packages

Before you begin, make sure you have the JDBC driver for your database (e.g., MySQL, PostgreSQL) included in your project and import the necessary JDBC packages at the beginning of your Java class:

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.Statement;  
import java.sql.ResultSet;  
import java.sql.SQLException;
```

Step 2: Establish a Database Connection

To use the Statement interface, you first need to establish a connection to your database. Here's how you can do it:

```
// Define connection parameters
String jdbcUrl = "jdbc:mysql://localhost:3306/your_database";
String username = "your_username";
String password = "your_password";

// Create a connection object
Connection connection = DriverManager.getConnection(jdbcUrl, username, password);
```

Note: Replace your_database, your_username, and your_password with your database details.

For our database the connection string will be as under:

```
        try {
            //Register the driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            //established the connection

            con=DriverManager.getConnection("jdbc:mysql://localhost:3306/studentmanagementsy
            stem", "root", "mysql"); // the password you need to put as per your system

            return con;
        }
    }
```

Step 3: Create a Statement

Once you have a connection, you can create a Statement object:

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

Step 4: Execute SQL Statements

You can execute SQL statements using the execute(), executeQuery(), or executeUpdate() methods of the Statement interface, depending on the type of SQL statement you want to execute.

Example 1: Executing a Query (Create)

```
String sqlQuery = "create table User(uid int primary key,name varchar(30),address
varchar(50))";
ResultSet resultSet = statement.executeQuery(sqlQuery);

while (resultSet.next()) {
    // Process the results here
    int id = resultSet.getInt("id");
    String name = resultSet.getString("name");
    // ... (retrieve other columns)
}
```

Let us explore complete example:

```
package jdbcConnectivity;

import java.sql.Connection;
import java.sql.Statement;

public class createTable {

    public static void main(String[] args) {

        //Here we have called dbConnect() method from ConnectDB class to
        established the connection
        try(Connection conn=ConnectDB.dbConnect()) {

            //printing connection object to check connection has established or not
            System.out.println(conn);

            //creating statement object over here
            Statement stmt=conn.createStatement();

            //creating query here
            String sql="create table User(uid int primary key,name
            varchar(30),address varchar(50))";

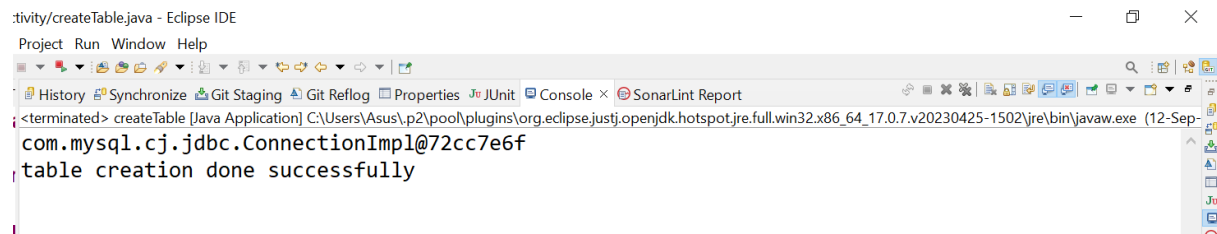
            //executing query over here
            stmt.executeUpdate(sql);
            System.out.println("table creation done successfully");

        }catch(Exception e)
        {
            System.out.println(e);
        }

    }
}
```

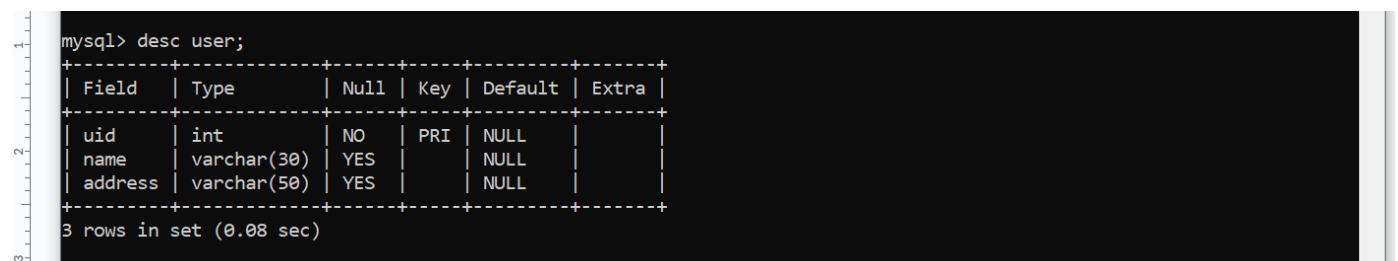

}

Output:



The screenshot shows the Eclipse IDE interface. The console window at the bottom displays the output of a Java application. The text in the console is: `<terminated> createTable [Java Application] C:\Users\Asus\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe (12-Sep-2023 10:00:00 AM) com.mysql.cj.jdbc.ConnectionImpl@72cc7e6f table creation done successfully`. The IDE's menu bar includes Project, Run, Window, and Help. The toolbar shows various icons for file operations and development tools.

Here we have used our predefined class to establish the connection. We have created a query to create a User table and execute the query.



The screenshot shows a MySQL command prompt window. The user has entered the command `mysql> desc user;`. The output is a table showing the structure of the 'user' table. The table has 6 columns: Field, Type, Null, Key, Default, and Extra. The rows are: uid (int, NO, PRI, NULL,), name (varchar(30), YES, , NULL,), and address (varchar(50), YES, , NULL,). The output ends with `3 rows in set (0.08 sec)`.

Field	Type	Null	Key	Default	Extra
uid	int	NO	PRI	NULL	
name	varchar(30)	YES		NULL	
address	varchar(50)	YES		NULL	

You can see the User table has been created successfully.

Knowledge check...

Now that we've covered driver class and basic understanding of Statement Interface, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1. What is the purpose of the DriverManager class in JDBC?

- A. To create instances of JDBC driver classes.
- B. To establish database connections.
- C. To execute SQL queries.
- D. To retrieve metadata about the database.

2. Which of the following statements is true about the DriverManager class in JDBC?

- A. It is an interface.
- B. It is used to execute SQL statements.
- C. It is responsible for managing a list of database drivers.
- D. It is used to retrieve metadata about the database.

3.Which method of the Statement interface is used to execute SQL queries that return multiple rows of data?

- A. executeQuery()
- B. executeUpdate()
- C. execute()
- D. executeInsert()

At the end of the quiz,the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.

Now let us explore the Resultset interface in details which is used mainly in SELECT query to retrieve the data before executing SELECT query:

Resultset interface:

The ResultSet interface in JDBC is used to retrieve and manipulate data from a database after executing a SQL query. It represents the result set of a query, which is essentially a table of data consisting of rows and columns. You can use ResultSet methods to navigate through the result set, retrieve data from specific columns, and perform various operations on the data.

Here's a brief overview of key concepts related to the ResultSet interface:

- **Creating a ResultSet:** You obtain a ResultSet object when you execute a SQL query using a Statement or PreparedStatement.
- **Navigating Through Rows:** ResultSet provides methods like next() to move the cursor to the next row, previous() to move to the previous row, and first() and last() to move to the first and last rows, respectively.
- **Retrieving Data:** You can retrieve data from the current row using methods like getInt(), getString(), getDouble(), etc., by specifying the column name or index. Iterating Through Rows: Typically, you use a loop to iterate through all the rows in the result set until next() returns false.

Syntax:

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM your_table");
while (resultSet.next()) {
    int id = resultSet.getInt("id");
    String name = resultSet.getString("name");
}
```

```
// ... (retrieve other columns)
}
```

- **Closing the ResultSet:** Always remember to close the ResultSet when you're done with it to release resources.

```
resultSet.close();
```

- **Handling SQL Exceptions:** ResultSet operations can throw SQLException, so it's essential to handle exceptions properly using try-catch blocks.

Syntax:

```
try {
    // ResultSet operations here
} catch (SQLException e) {
    e.printStackTrace();
}
```

The ResultSet interface is a crucial part of working with JDBC because it allows you to interact with the data retrieved from a database and process it in your Java application. Depending on your requirements, you may need to use different ResultSet methods to extract and manipulate data efficiently.

Let us explore select query with complete example:

Here we are retrieving Student's table data using Statement interface

```
package jdbcConnectivity;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class selectQuery {

    public static void main(String[] args) {
        //Creating connection and statement using try with resource statement
        //creating statement
        //establishing connection using predefined class
        try(Connection con=ConnectDB.dbConnect();
            Statement st=con.createStatement();)
        {

            //creating query
            String sql="select * from Student";
```

```

//executing query
ResultSet rs=st.executeQuery(sql);

//checking data is present or not
while(rs.next())
{

    System.out.println("Student Id: "+rs.getString("StudentID"));
    System.out.println("First name: "+rs.getString("FirstName"));
    System.out.println("LastName: "+rs.getString("LastName"));
    System.out.println("Date of birth: "+rs.getDate("DateOfBirth"));
    System.out.println("Gender: "+rs.getString("Gender"));
    System.out.println("email: "+rs.getString("Email"));
    System.out.println("Phone Number: "+rs.getString("Phone"));
    System.out.println("Marks: "+rs.getInt("marks"));

System.out.println("=====");

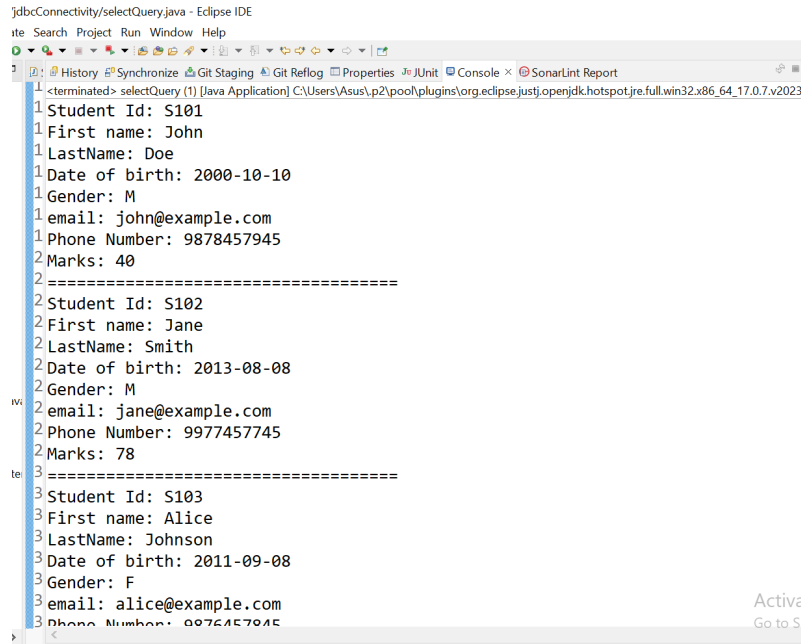
}

}catch(Exception e)
{
    System.out.println(e);
}

}

```

Output:



```
jdbcConnectivity/selectQuery.java - Eclipse IDE
1 <terminated> selectQuery (1) [Java Application] C:\Users\Asus\AppData\Local\Temp\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v2023
1 Student Id: S101
1 First name: John
1 LastName: Doe
1 Date of birth: 2000-10-10
1 Gender: M
1 email: john@example.com
1 Phone Number: 9878457945
2 Marks: 40
2 =====
2 Student Id: S102
2 First name: Jane
2 LastName: Smith
2 Date of birth: 2013-08-08
2 Gender: M
2 email: jane@example.com
2 Phone Number: 9977457745
2 Marks: 78
3 =====
3 Student Id: S103
3 First name: Alice
3 LastName: Johnson
3 Date of birth: 2011-09-08
3 Gender: F
3 email: alice@example.com
3 Phone Number: 9876457945
```

Here we have created a statement to execute a Select query. Here we have used try with resource to skip finally block for closing the Connections and Statement object.

Question:

How can you handle SQL NULL values when working with ResultSet objects, and what methods are available for checking and handling them?

Note: We will see the PreparedStatement interface and Callable statement in the upcoming session.

Exercise:

Use ChatGPT to explore Transaction Management:

Put the below problem statement in the message box and see what ChatGPT says.

Explain the concept of transaction management in JDBC. How can you start, commit, and roll back transactions?

```
Id: 1
Name:priya
Address: kolkata
salary: 30000
=====
Id: 2
Name:riya
Address: bangalore
salary: 56000
=====
```