विशाल जाधव सरांचे

VJTech Academy

Inspiring Your Success...

UNIT-VI Interacting with Database

Syllabus

- Introduction to JDBC, ODBC
- JDBC architecture: Two tier and three tier models
- Types of JDBC drivers, Class Class, DriverManager class, Connection interface, Statement interface, PreparedStatement interface, ResultSet Interface

Introduction to JDBC, ODBC

❖ JDBC

- **JDBC (Java Database Connectivity)** is a Java-based API that allows Java applications to connect and interact with relational databases like MySQL, Oracle, SQL Server, etc.
- JDBC provides a standard interface for querying and updating data in a database, allowing developers to interact with databases without needing to understand the underlying database systems.

Key Components of JDBC:

1. **JDBC Drivers**: These are platform-specific implementations that enable communication between Java applications and the database. There are four types of JDBC drivers:
    - **Type 1**: JDBC-ODBC bridge driver.
    - **Type 2**: Native-API driver.
    - **Type 3**: Network Protocol driver.
    - **Type 4**: Thin driver (pure Java driver).
2. **JDBC API**: It provides various interfaces and classes such as `Connection`, `Statement`, `ResultSet`, and `DriverManager` for interacting with the database.
3. **Connection**: This represents a session with the database. It's used to create `Statement` objects and control transactions.
4. **Statement**: This interface is used to execute SQL queries against the database. Types of statements include `Statement`, `PreparedStatement`, and `CallableStatement`.
5. **ResultSet**: This interface holds the result of a query, allowing Java applications to iterate over the rows and retrieve data.

❖ ODBC

- **ODBC (Open Database Connectivity)** is a standard API for accessing database management systems (DBMS).
- It was developed by Microsoft to provide a common interface for database connectivity, regardless of the DBMS being used.
- ODBC allows applications to interact with databases using SQL, and it is widely supported by various database systems.

Key Components of ODBC:

1. **ODBC Driver**: An ODBC driver is required to communicate with the specific DBMS. It translates ODBC calls into commands that the database understands.
2. **ODBC Driver Manager**: The driver manager manages the interaction between an application and the database driver. It ensures that the correct driver is used and manages the database connections.
3. **Data Source Name (DSN)**: A DSN stores the connection details for a database, such as the driver, database location, username, and password.

Key Differences between JDBC and ODBC:

Feature	JDBC	ODBC
Platform	Java-specific (cross-platform)	Platform-specific (mainly Windows)
Language	Java	C/C++
API Type	Java API for database connectivity	C API for database connectivity
Ease of Use	Easy to use within Java applications	Requires configuration of DSN and ODBC drivers
Driver Support	Database-specific JDBC drivers	Database-specific ODBC drivers
Flexibility	More flexible with Java applications	Can be used with any C/C++ application
Connection Type	Uses `DriverManager` for connection	Uses `ODBC Driver Manager` for connection

Both JDBC and ODBC serve similar purposes: to provide a standardized way of connecting to databases. However, JDBC is optimized for Java, while ODBC is a more general standard suitable for any programming language that supports C-based libraries.

JDBC Architecture: Two-Tier and Three-Tier Models:

- The architecture of JDBC (Java Database Connectivity) defines how Java applications interact with databases.
- It can be implemented in two primary models: **Two-Tier** and **Three-Tier**.
- These models define the layers of interaction between the client (Java application) and the database.

❖ Two-Tier Architecture:

- In a **Two-Tier Architecture**, the client (Java application) directly communicates with the database server.
- The communication happens through the JDBC API, and the database server provides the data directly to the client.

Key Characteristics:

- **Client**: The Java application (running on the client machine) sends SQL queries to the database and receives the results.
- **Database Server**: The database system is installed on the server-side. The database handles the query execution and sends results back to the client.

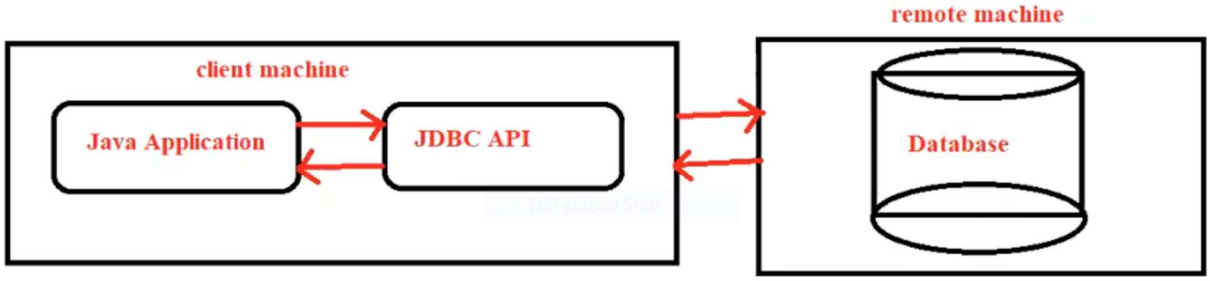


Fig. Two-Tier Application Architecture

In this model, the client directly interacts with the database using JDBC drivers (usually Type 4 drivers), and there is no intermediate layer between the client and the database.

❖ Three-Tier Architecture:

- In a **Three-Tier Architecture**, the client communicates with a **middle layer** (usually called an **Application Server** or **Business Logic Layer**), and the middle layer communicates with the database.
- This model is more scalable and flexible than the two-tier model.

Key Characteristics:

- **Client**: The client (Java application) is responsible for the user interface and can send requests to the middle-tier server (usually through HTTP or other protocols).
- **Application Server (Middle Layer)**: This layer processes business logic, handles database connectivity, and serves as an intermediary between the client and the database. The application server manages database interactions, such as executing SQL queries and managing transactions.
- **Database Server**: The database server is responsible for storing and managing data. It processes SQL queries and returns results to the application server.

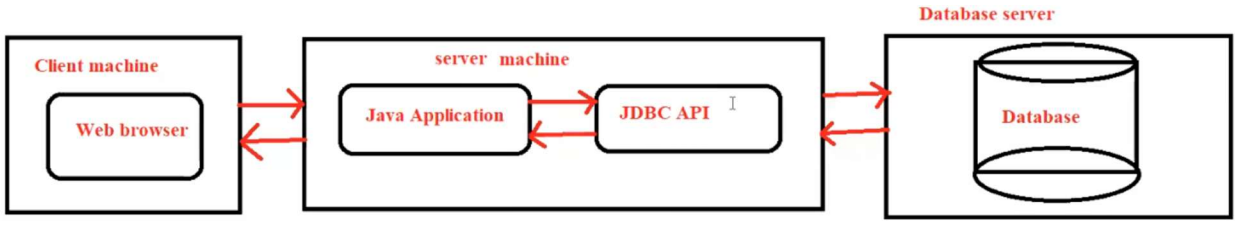


Fig. Three-Tier Application Architecture

In a three-tier architecture, the client might send HTTP requests to a web server (Application Server), which then connects to a database using JDBC to retrieve the data and send it back to the client.

Comparison of Two-Tier vs. Three-Tier Architecture

Feature	Two-Tier Architecture	Three-Tier Architecture
Layers	Client and Database Server	Client, Application Server, and Database Server
Communication	Direct communication between client and DB	Indirect communication through an application server
Scalability	Limited scalability	Highly scalable, middle layer can be expanded
Complexity	Simple architecture	More complex due to the addition of an application server
Flexibility	Tightly coupled client-server interaction	Decoupled layers, offering more flexibility
Performance	Faster response time (no middle layer)	Slower response time due to additional layer
Maintainability	Harder to maintain with many clients	Easier to maintain due to centralized business logic in the middle layer
Security	Less secure (direct DB connection)	More secure (application server handles DB access)

Types of Database Drivers:

JDBC (Java Database Connectivity) drivers are platform-specific implementations that allow Java applications to connect to a database. There are **four types** of JDBC drivers, each differing in how they connect to the database and handle the communication between the client and the database server.

❖ Type 1 Driver: JDBC-ODBC Bridge Driver:

- The **Type 1 driver** uses ODBC (Open Database Connectivity) to connect to the database.
- The Java application communicates with the ODBC driver that communicate with the database.
- The JDBC-ODBC bridge acts as a translator between JDBC and ODBC.
- The Java application sends JDBC calls to the JDBC-ODBC bridge. The bridge translates these calls into ODBC calls, which are then sent to the database.

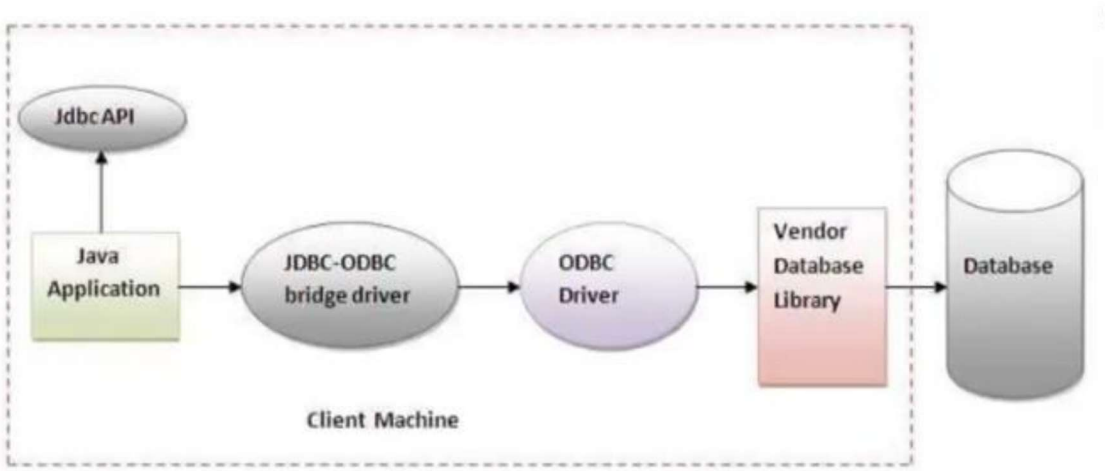


Figure- JDBC-ODBC Bridge Driver

Advantages:

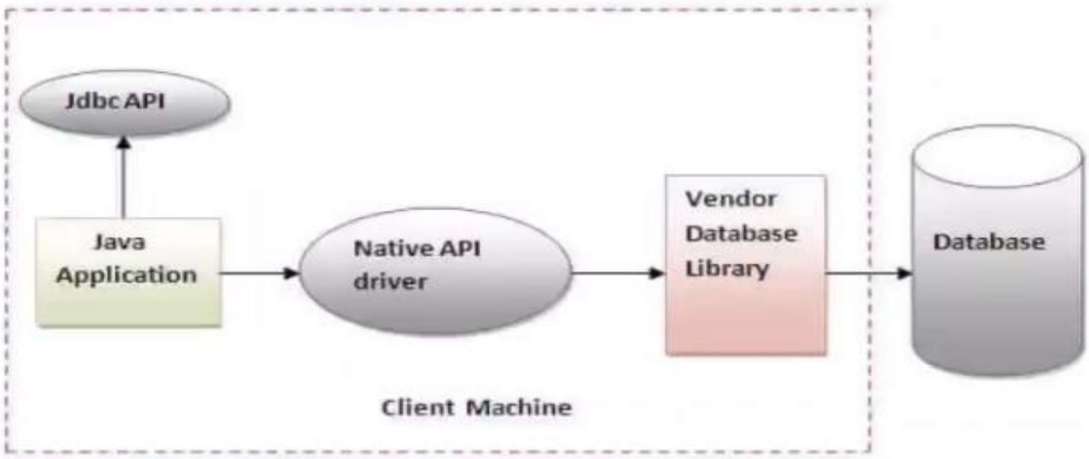
- It can be used with any database that provides an ODBC driver.
- Simple to use

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

❖ Type 2 Driver: Native-API Driver:

- The **Type 2 driver** uses a native database client (e.g. C or C++ based API) to connect to the database.
- The Java application communicates with the database through a native library, which is specific to the database.
- The **driver communicates with the database** using a **native DBMS (Database Management System) API**.
- Requires the installation of the **native client** on the client machine.



Advantages:

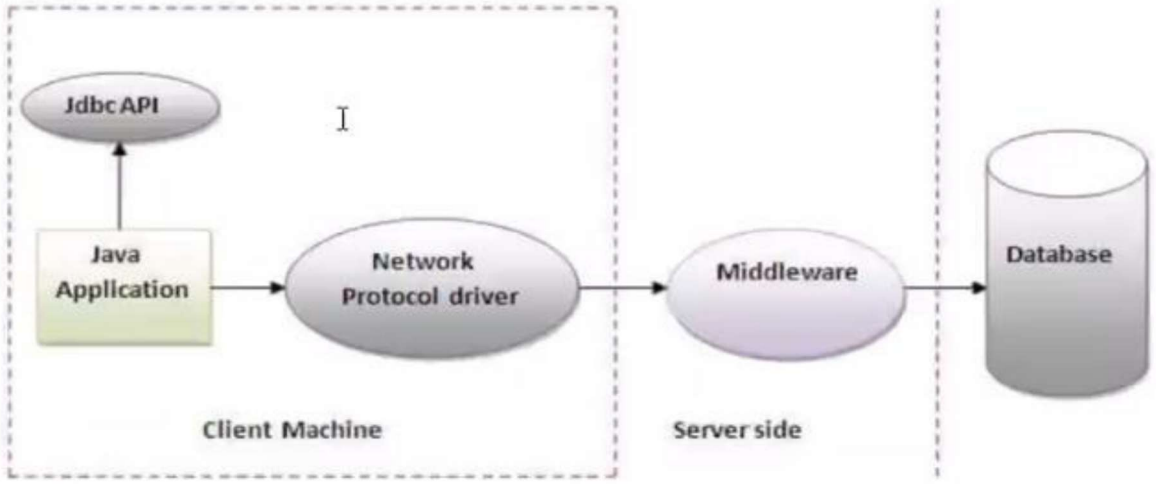
- Performance upgraded than JDBC-ODBC bridge driver.
- Simple to use

Disadvantages:

- The Native driver needs to be installed on each client machine.
- The Vendor client library needs to be installed on client machine.

❖ Type 3 Driver: Network Protocol Driver:

- The **Type 3 driver** communicates with the database through a middleware server.
- This middleware server translates the network protocol used by the database into JDBC calls.
- The driver does not require native database libraries on the client machine.
- The **Java application** sends requests to a **middleware server**
- The middleware server **translates the requests** into a specific protocol used by the database.
- The **database** then communicates with the middleware, and the results are sent back to the Java application.



Advantages:

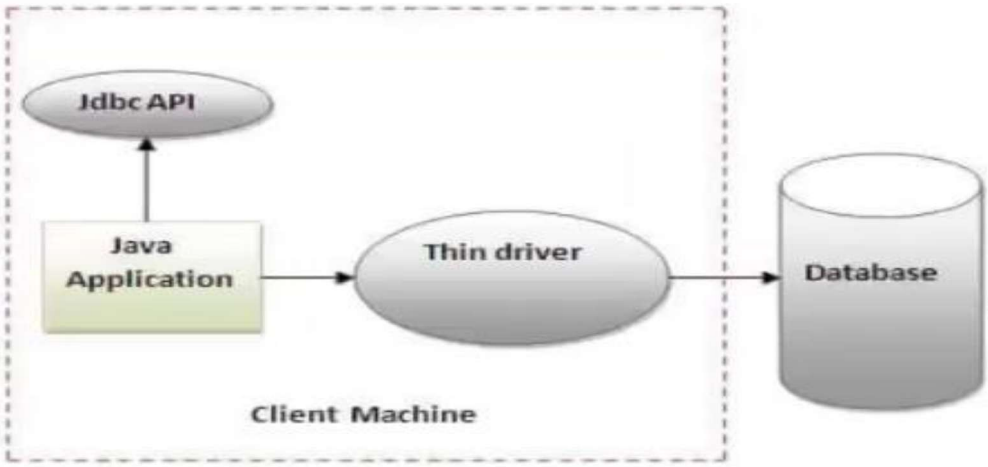
- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
- Simple to use

Disadvantages:

- Network is required on client
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database specific coding to be done in the middle tier.

❖ Type 4 Driver: Thin Driver:

- The **Type 4 driver** is a **pure Java driver** that communicates directly with the database using the database's native protocol.
- The driver does not require any native client libraries or middleware servers.
- The Type 4 driver is written entirely in Java.
- It directly communicates with the database using **database-specific protocols**.
- The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver.



Advantages:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantages:

- Drivers depend on the Database.

Java Database Connectivity with 5 Steps:

There are 5 steps to connect any java application with the database using JDBC.

These steps are as follows:

1) Register the Driver class

2) Create connection

3) Create statement

4) Execute queries

5) Close connection

1. Register the Driver class (Class):

- The forName() method of class Class is used to register the driver class.
- This method is used to dynamically load the driver class.
- In Java, the `Class` class is used to represent classes and interfaces at runtime.
- The `Class` class is used to load JDBC drivers.
- We can load a driver using the `forName()` method of the `Class` class, which dynamically loads the driver into memory.
- **Syntax of forName() method:**

  public static void forName(String className)
- **Example:**

  Java program is loading oracle driver to establish database connection.

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

2. Create the connection object:

- The `DriverManager` class manages a list of database drivers. When a connection request is made, it selects an appropriate driver from its list and establishes a connection with the database.
- The getConnection() method of DriverManager class is used to establish connection with the database.
- **Syntax of getConnection() method:**

```
1. Public static Connection getConnection(String url, String
   username, String password)
2. Public static Connection getConnection(String url)
```

- **Example:**

  Below code is used to establish the connection with the Oracle database

  Connection con=DriverManager.getConnection("jdbc:oracle:thin: @localhost:1521:xe", "system","password");

3. Create the Statement object:

- The createStatement() method of Connection interface is used to create statement.
- The object of statement is responsible to execute queries with the database.
- Creates a `Statement` object for sending SQL statements to the database.
- The `Statement` interface is used to execute basic SQL queries and updates (INSERT, SELECT, UPDATE, DELETE).
- It is used for static SQL queries without parameters.
- **Syntax of createStatement() method:**

  public Statement createStatement()

- **Example to create the statement object:**

  Statement stmt=con.createStatement()

4. Execute the query:

- Following methods of Statement interface is used to execute queries to the database.

Key Methods:

- executeQuery(String sql): Executes a query that returns a ResultSet object (e.g., SELECT queries).
- executeUpdate(String sql): Executes an SQL statement that updates data in the database (e.g., INSERT, UPDATE, DELETE).
- execute(String sql): Executes a SQL statement and returns a boolean to indicate if it returns a result set.
- close(): Closes the Statement object and releases resources.

- **Example to execute query:**

  stmt.execute("insert into values(1010,'Dennis',90)");

6. Close the connection:

- The close() method of Connection interface is used to close the connection.
- **Syntax of close() method**

  public void close() throws SQLException

- **Example of Close Connection:**

  Con.close();

❖ Java Program to establish connection with oracle database:

```
import java.sql.*;
class JDBCConnectionDemo
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection ("jdbc:oracle:thin:@localhost:1521:xe", "system","system");
		System.out.println("Connection Established successfully");

		con.close();
		System.out.println("Connection Closed successfully");
	}
}
```

❖ Java Program to create table in database:

```
import java.sql.*;
class CreateTableDemo
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection ("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection Established successfully");

		Statement stmt=con.createStatement();

		stmt.execute("create table Student(rollno number(4),name varchar(10),marks number(2))");
		System.out.println("Table created successfully");

		con.close();

	}
}
```

❖ Java Program to insert new row into table:

```
import java.sql.*;
class InsertRowIntoTable
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection Established successfully");

		Statement stmt=con.createStatement();

		stmt.execute("insert into Student values(3030,'Lisa',85)");
		System.out.println("New Row Inserted successfully");

		con.close();

	}
}
```

❖ Java Program to update row in table:

```
import java.sql.*;
class UpdateRowIntoTable
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection Established successfully");

		Statement stmt=con.createStatement();

		stmt.execute("update Student set name='Peter' where rollno=2020");
		System.out.println("Row Updated successfully");

		con.close();

	}
}
```

❖ Java Program to delete row from table:

```
import java.sql.*;
class DeleteRowFromTable
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection Established successfully");

		Statement stmt=con.createStatement();

		stmt.execute("delete from Student where rollno=3030");
		System.out.println("Row Deleted successfully");

		con.close();

	}
}
```

❖ Java Program to fetch all records from table:

```
import java.sql.*;
class FetchRecordsFromTable
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");
		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection created successfully");

		Statement stmt=con.createStatement();

		System.out.println("******STUDENT MNGT SYSTEM*******");
		System.out.println("ROLLNO\tNAME\tMARKS");
		System.out.println("==============================");
		ResultSet rs=stmt.executeQuery("select * from Student");
		while(rs.next())
		{
			System.out.println(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+rs.getInt(3));
		}
		con.close();
		System.out.println("Connection closed successfully");
	}
}
```

PreparedStatement Interface in JDBC:

- The PreparedStatement interface is a subinterface of Statement in JDBC.
- It is used to execute precompiled SQL queries that may contain input parameters.
- It is more efficient than using the Statement interface for executing queries, especially when executing the same query multiple times with different parameters.

Key Features of `PreparedStatement`

1. **Precompiled SQL**: `PreparedStatement` precompiles the SQL query, so the database doesn't need to compile the query repeatedly.
2. **Parameterized Queries**: You can use symbol (`?`) in the SQL query, and later put values to those symbols at runtime.
3. **Improved Performance**: Because the SQL query is precompiled, it improves performance when executing the same query multiple times with different parameters.

Key Methods in `PreparedStatement`

Here are some of the key methods provided by the `PreparedStatement` interface:

1. `setXXX(int parameterIndex, XXX value)`

- **Description**: This method is used to set the values of the input parameters (the symbol `?` in the SQL query).
- `XXX` represents the data type (e.g., `setString`, `setInt`, `setDate`, etc.).
- `parameterIndex` is the index of the parameter in the SQL query.
- **Common `setXXX` Methods**:
  - `setString(int parameterIndex, String value)`
  - `setInt(int parameterIndex, int value)`
  - `setDouble(int parameterIndex, double value)`
  - `setBoolean(int parameterIndex, boolean value)`
  - `setDate(int parameterIndex, Date value)`

Example:

```
import java.sql.*;
class InsertRowIntoTablePrepared
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");

		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection Established successfully");

		PreparedStatement pstmt=con.prepareStatement("insert into Student values(?,?,?)");

		pstmt.setInt(1,9090);		// Set the first parameter (rollno)
		pstmt.setString(2,"Peter");	// Set the second parameter (name)
		pstmt.setInt(3,85);		// Set the third parameter (marks)

		pstmt.executeUpdate();
		System.out.println("New Row Inserted successfully");

		con.close();

	}
}
```

ResultSet Interface:

- In Java, the `ResultSet` interface is part of the `java.sql` package and represents the result set of a database query.
- It is used to retrieve data from a database after executing a query using JDBC (Java Database Connectivity).
- The `ResultSet` object provides methods to retrieve and manipulate data in a tabular format (like rows and columns).

Key Features:

1. You can move through the rows of a `ResultSet` one by one, forward or backward.
2. It allows access to data in columns of the current row using column names or indices.
3. In some cases, the `ResultSet` can be updateable, meaning you can modify the data directly within the result set and update the database.
4. **Types of ResultSets**: The `ResultSet` can be of different types, such as:
    - **Forward-only** (default): You can only move forward through the rows.
    - **Scrollable**: You can move both forward and backward through the rows.
    - **Updatable**: You can modify the data.

Methods of ResultSet:

- `public boolean next()`: Moves the cursor to the next row in the result set. Returns `true` if there is another row, otherwise `false`.
- `public boolean previous()`: Moves the cursor to the previous row in a scrollable result set.
- `public boolean first()`: is used to move the cursor to the first row in result set object.
- `public boolean last()`: is used to move the cursor to the last row in result set object.
- `public int getInt(int columnIndex)`: Retrieves the value of the specified column index as an integer.
- `public int getInt(String columnName)`: Retrieves the value of the specified column name as an integer.
- `public String getString(int columnIndex)`: Retrieves the value of the specified column index as a string.
- `public String getString(String columnName)`: Retrieves the value of the specified column name as a string.

- `public double getDouble(String columnName)` : Retrieves the value of the specified column name as a double.
- `public double getDouble(int columnIndex)` : Retrieves the value of the specified column index as a double.
- `close()` : Closes the `ResultSet` and releases the resources it holds.

Example: Java Program to fetch all records from table Using ResultSet interface:

```
import java.sql.*;
class FetchRecordsFromTable
{
	public static void main(String args[])throws Exception
	{
		Class.forName("oracle.jdbc.driver.OracleDriver");
		System.out.println("Driver loaded successfully");
		Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","system");
		System.out.println("Connection created successfully");

		Statement stmt=con.createStatement();

		System.out.println("******STUDENT MNGT SYSTEM*******");
		System.out.println("ROLLNO\tNAME\tMARKS");
		System.out.println("==============================");
		ResultSet rs=stmt.executeQuery("select * from Student");
		while(rs.next())
		{
			System.out.println(rs.getInt(1)+"\t"+rs.getString(2)+"\t"+rs.getInt(3));
		}
		con.close();
		System.out.println("Connection closed successfully");
	}
}
```