JDBC CONNECTIVITY

JDBC provides framework to connect to relational databases from java programs.

- ***JDBC API provides industry-standard and database-independent connectivity between java applications and database servers.***

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

JDBC can be used with Java Applications, Java Applets, Java Servlets, Java ServerPages (JSPs) and Enterprise JavaBeans (EJBs)

JDBC Architecture

JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Managerconnection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

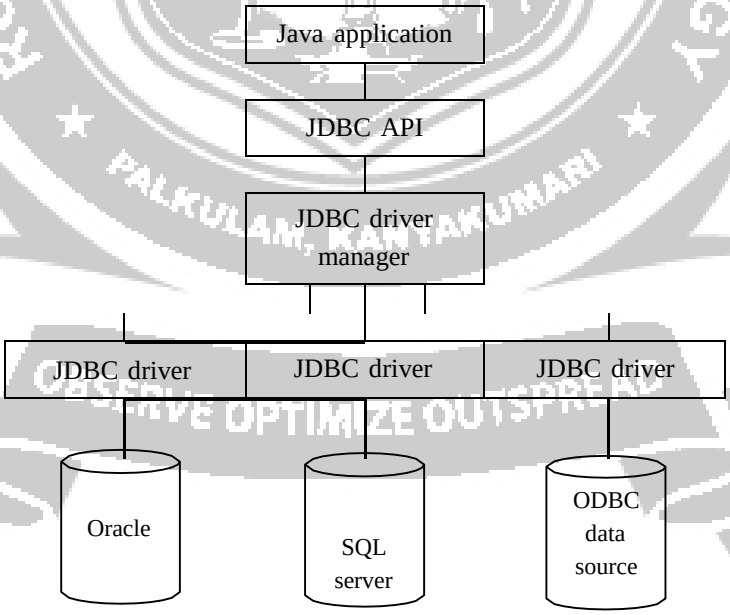

Figure 3.18 JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access.The JDBC API uses a driver manager and database-specific drivers to provide connectivity to heterogeneous databases.

The **JDBC driver manager** ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Components of JDBC

The following are some of the important components of JDBC:

DriverManager:

This class manages a list of database drivers. It is responsible for matching the connection requests from the java application with the proper database driver using communication sub protocol.

Driver:

This interface handles the communications with the database server.The DriverManager manages these objects.

Connection:

This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

Statement:

The objects created from this interface are used to submit the SQL statements to the database.

ResultSet:

These objects hold data retrieved from a database after executing an SQL query using Statement objects. It acts as an iterator that moves through its data.

SQLException:

This class handles any errors that occur in a database application**.**

JDBC Driver

JDBC drivers implement the defined interfaces in the JDBC API for interacting with the database server.The JDBC drivers open database connections and interact with it by sending SQL or database commands then receiving results with Java.

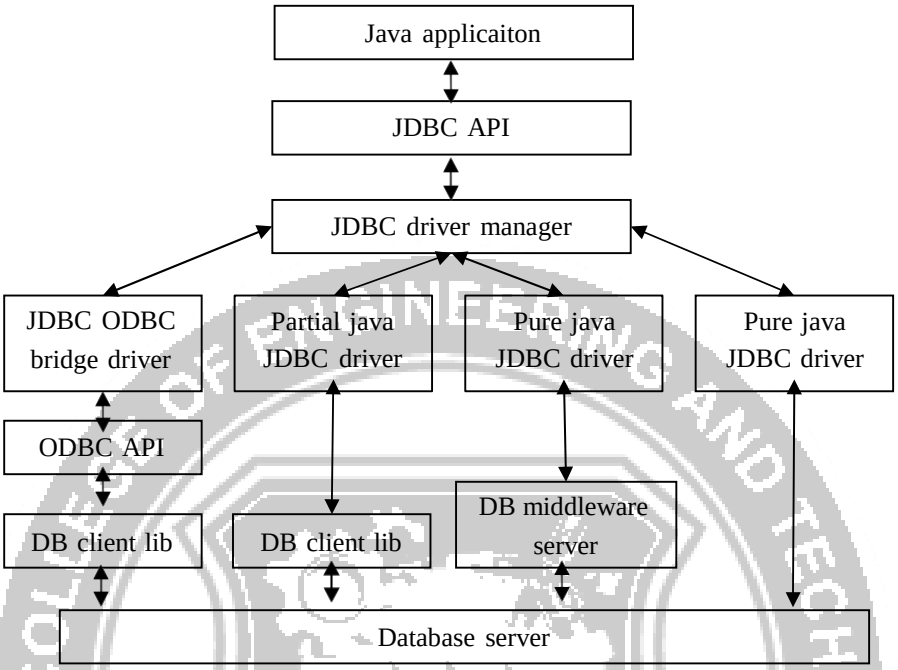

Figure 3.19 JDBC Drivers

1. **JDBC-ODBCBridge plus ODBC Driver (Type 1):**

   This driver uses ODBC driver to connect to database servers. The ODBC drivers must be installed in the machines from where the connection is established to JDBC drivers.This driver is almost obsolete and should be used only when other options are not available.

2. **Native API partly Java technology-enabled driver (Type 2):**

   This type of driver converts JDBC class to the client API for the RDBMS servers.The database client API should be installed at the machine from which we want to make database connection. Because of extra dependency on database client API drivers, this is also not preferred driver.

3. **Pure Java Driver for Database Middleware (Type 3):**

   This type of driver sends the JDBC calls to a middleware server that can connect to different type of databases. A middleware server must be installed to work with this kind of driver. This adds to extra network calls and slow performance. Hence this is also not widely used JDBC driver.

4. **Direct-to-Database Pure Java Driver (Type 4):**

   This is the preferred driver because it converts the JDBC calls to the network protocol understood by the database server. This solution doesn't require any extra APIs at the client side and suitable for database connectivity over the network. However for this solution, we should use database specific drivers, for example OJDBC jars provided by Oracle for Oracle DB and MySQL Connector/J for MySQL databases.

Creating a JDBC application

There are following six steps involved in building a JDBC application:

1. **Import the packages -** Include the packages containing the JDBC classes needed for database programming.
2. **Register the JDBC driver -** Initialize a driver so to open a communications channel with the database.
3. **Open a connection** - Using DriverManager.getConnection() method create a Connection object, which represents a physical connection with the database.
4. **Execute a query** - Statement for building and submitting an SQL statement to the database.
5. **Extract data from result set -** Use ResultSet.getXXX() method to retrieve the data from the result set.
6. **Clean up the environment** - Explicitly closes all database resources.

Connections using JDBC

The JDBC connections can be established in two ways:

- By using JDBC-ODBC bridge
- By using vendor specific JDBC driver

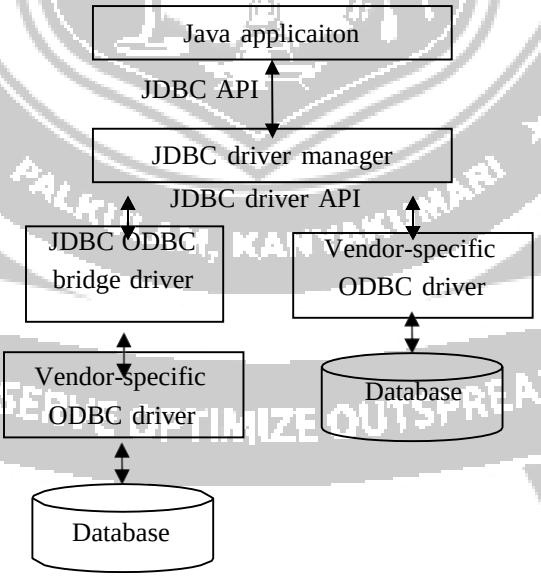

Figure 3.20 JDBC drivers

Define the Connection URL

URLs referring to databases use the jdbc: protocol and embed the server host, port, and database name (or reference) within the URL.

Establish the Connection

To make the actual network connection, pass the URL, database username, and database password to the getConnection method of the DriverManager class.The getConnection throws an SQLException.

Connection connection = DriverManager.getConnection(oracleURL, username, password);

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

Methods of connection class:

1. prepareStatement- Creates precompiled queries for submission to the database.
2. prepareCall- Accesses stored procedures in the database.
3. rollback/commit- Controls transaction management.
4. close- Terminates the open connection.
5. isClosed- Determines whether the connection timed out or was explicitly closed.

JDBC - Statements, PreparedStatement and CallableStatement

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 to send SQL or PL/SQL commands and receive data from the database. They also define methods that help bridge data type differences between Java and SQL data types used in a database

Interfaces	Recommended Use
Statement	Use for general-purpose access to thedatabase. Useful when using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Used when the SQL statements are to be executed many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when database stored procedures are to be executed. The CallableStatement interface can also accept runtime input parameters.

Create a Statement Object

A Statement object is used to send queries and commands to the database.It is created from the Connection using createStatement.

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

Most, but not all, database drivers permit multiple concurrent Statement objects to be open on the same connection.

Execute a Query or Update

The Statement object can be uses to send SQL queries by using the executeQuery method, which returns an object of type ResultSet.

Example:

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

The methods in the Statement class are:

Methods	Description
executeQuery	Executes an SQL query and returns the data in a ResultSet. The ResultSet may be empty, but never null.
executeUpdate	Used for UPDATE, INSERT, or DELETE commands. Returns the number of rows affected, which could be zero. Also provides support for Data Definition Language (DDL) commands, for example, CREATE TABLE, DROP TABLE, and ALTERTABLE.

executeBatch	Executes a group of commands as a unit, returning an array with the update counts for each command. Use addBatch to add a command to the batch group.
setQueryTimeout	Specifies the amount of time a driver waits for the result before throwing a SQLException.
getMaxRows/setMaxRows	Determines the number of rows a ResultSet may contain. Excess rows are silently dropped. The default is zero for no limit.

Process theResults

The simplest way to handle the results is to use the next method of ResultSet to move through the table a row at a time. Within a row, ResultSet provides various getXxx methods that take a column name or column index as an argument and return the result in a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types. To just display the results, use getString for most of the column types.

```
while(resultSet.next())
{System.out.println(resultSet.getString(1) + " " + resultSet.getString(2) + " " +
resultSet.getString("firstname") + " " resultSet.getString("lastname"));}
```

Methods	Description
next/previous	Moves the cursor to the next (any JDBC version) or previous (JDBC version 2.0 or later) row in the ResultSet, respectively.
relative/absolute	The relative method moves the cursor a relative number of rows, either positive (up) or negative (down). The absolute method moves the cursor to the given row number. If the absolute value is negative, the cursor is positioned relative to the end of the ResultSet (JDBC 2.0).
getXxx	Returns the value from the column specified by the column name or column index as an Xxx Java type (see java.sql.Types). Can return 0 or null if the value is an SQL NULL.
wasNull	Checks whether the last getXxx read was an SQL NULL.
findColumn	Returns the index in the ResultSet corresponding to the specified column name.
getRow	Returns the current row number, with the first row starting at 1 (JDBC 2.0).

getMetaData	Returns a ResultSetMetaData object describing the ResultSet.
ResultSetMetaData	Gives the number of columns and the column names.

Close the Connection

To close the connection explicitly close() is used.

connection.close();

Closing the connection also closes the corresponding Statement and ResultSet objects.

The PreparedStatement Objects

The PreparedStatement interface extends the Statement interface which gives added functionality with a couple of advantages over a generic Statement object. This statement gives the flexibility of supplying arguments dynamically.

Example:

```
PreparedStatement pstmt = null;
Try{ String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL);}
```

The CallableStatement Objects

A Connection object creates the CallableStatement object which would be used to execute a call to a database stored procedure.

Simple JDBC connection in a servlet

```
import java.io.IOException;import java.io.PrintWriter;import java.sql.Connection;
import java.sql.DriverManager;import java.sql.ResultSet;import java.sql.SQLException;
import java.sql.Statement;import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class JDBCServlet extends HttpServlet
{ public void doGet(HttpServletRequest inRequest, HttpServletResponse outResponse)
throws ServletException, IOException
  { PrintWriter out = null;
    Connection connection = null;
    Statement statement;
    ResultSet rs;
```

```
try {
        Class.forName("com.mysql.jdbc.Driver");
        connection =
DriverManager.getConnection("jdbc:mysql://localhost/products");
        statement = connection.createStatement();
        outResponse.setContentType("test/html");
        out = outResponse.getWriter();
        rs = statement.executeQuery("SELECT ID, title, price FROM product");
        out.println("<HTML><HEAD><TITLE>Products</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<UL>");
      while (rs.next()) {
                out.println("<LI>" + rs.getString("ID") + " " + rs.getString("title")
+ " " +    rs.getString("price"));
                }
     out.println("</UL>");
    out.println("</BODY></HTML>");        }
catch (ClassNotFoundException e)
{   out.println("Driver Error");    }
catch (SQLException e)
{   out.println("SQLException: " + e.getMessage());   }  }
 public void doPost(HttpServletRequest inRequest,   HttpServletResponse outResponse)
throws ServletException, IOException
{   doGet(inRequest, outResponse);  }}
```

ID	Title	Price
56	Soap	40
98	Shampoo	15

Batch Processing

Batch Processing allows grouping related SQL statements into a batch and submitting them with one call to the database. When several SQL statements are sent to the database at once, they can be clubbed together to reduce the amount of communication overhead. This is called **batch processing.** The following methods are used in batch processing:

S.No	Method	Description
1.	DatabaseMetaData. supportsBatchUpdates()	This method determines if the target database supports batch update processing.
2.	addBatch()	This method is used to add individual statements to the batch.
3.	executeBatch()	This method is used to start the execution of all the statements grouped together.
4.	clearBatch()	This method removes all the statements added with the addBatch() method. The statements cannot be selectively chosen.

The following code provides an example of a batch update using Statement object:

```
Statement stmt = conn.createStatement();
conn.setAutoCommit(false); // Set auto-commit to false
String SQL = "INSERT INTO Employees (id, first, last, age) " +
"VALUES(200,'Zia', 'Ali', 30)";
stmt.addBatch(SQL); // Add above SQL statement in the batch.
String SQL = "INSERT INTO Employees (id, first, last, age) " +
"VALUES(201,'Raj', 'Kumar', 35)"; // Create one more SQL statement
stmt.addBatch(SQL); // Add above SQL statement in the batch.
String SQL = "UPDATE Employees SET age = 35 " + "WHERE id = 100";
stmt.addBatch(SQL);
int[] count = stmt.executeBatch();// Create an int[] to hold returned values
conn.commit();//Explicitly commit statements to apply changes
```