
H2K Infosys, LLC
Dream, Strive & Achieve Victory

H2K Infosys, LLC
Dream, Strive & Achieve Victory
H2K Infosys, LLC
Dream, Strive & Achieve Victory

www.H2KINFOSYS.com
USA: 770-777-1269
Training@H2KInfosys.com
UK : (020) 3371 7615

# Java Training Material

# JDCB

# What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

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

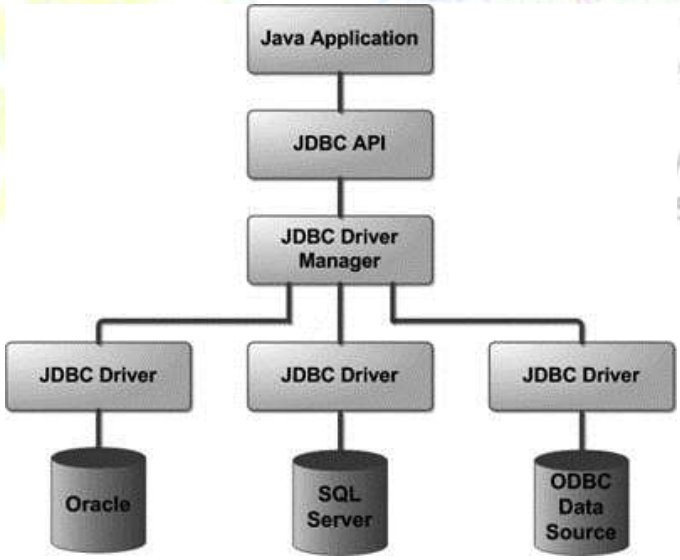
- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

# JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **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.




# Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub-protocol.

The first driver that recognizes a certain sub-protocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver 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 :** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class handles any errors that occur in a database application

# What is JDBC Driver ?

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

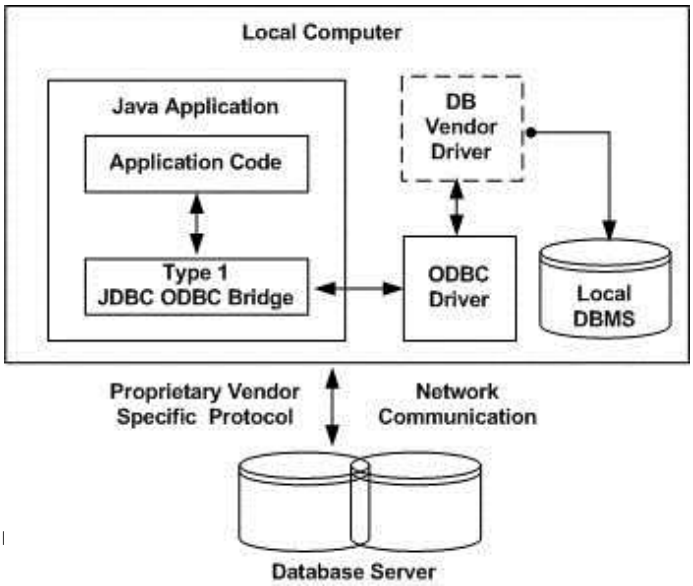
For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

# JDBC Drivers Types:

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below:

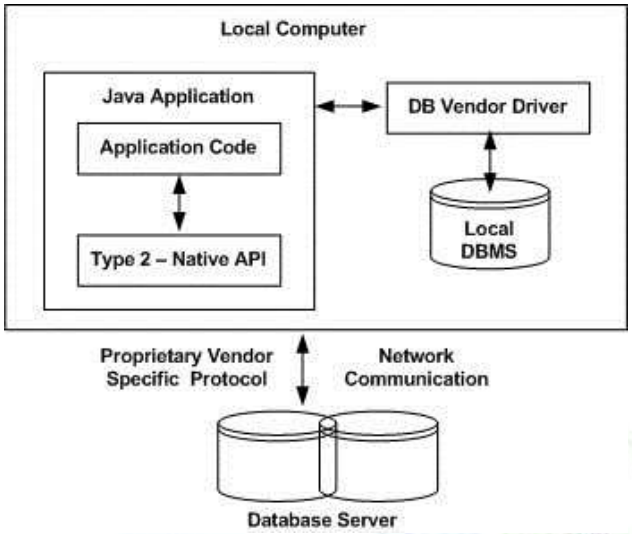
# Type 1: JDBC-ODBC Bridge Driver:

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



# Type 2: JDBC-Native API:

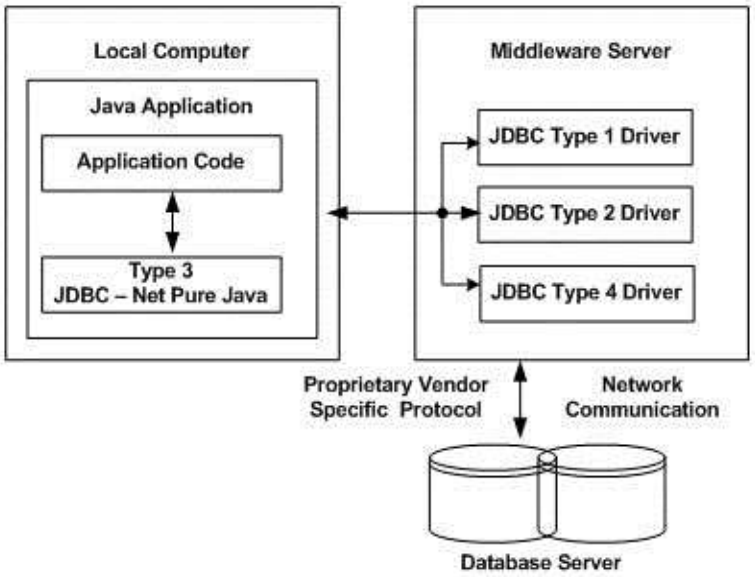
In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.




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

# Type 3: JDBC-Net pure Java:

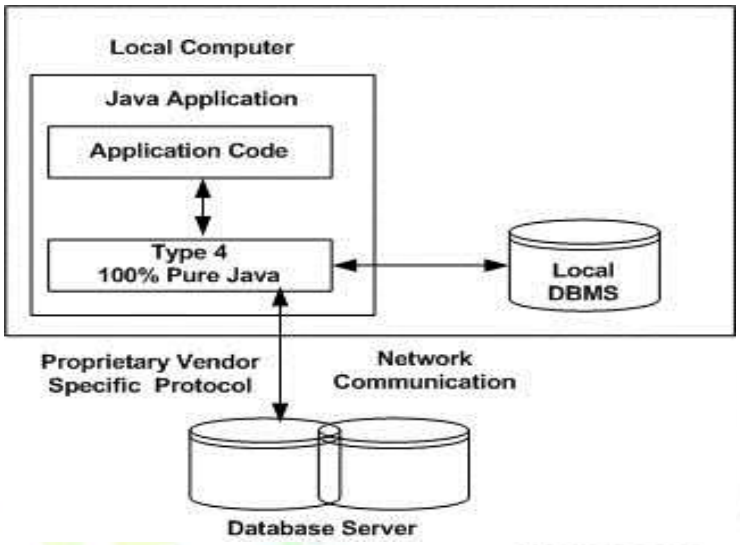
In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded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



## Type 4: 100% pure Java:

In a Type 4 driver, a pure Java-based driver that communicates directly with 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.






## Which Driver should be used?



If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4. If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver. Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database. The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

## Register JDBC Driver:

**Approach (I) - Class.forName():**

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

**Approach (II) - DriverManager.registerDriver():**

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

## Database URL Formulation:

After you've loaded the driver, you can establish a connection using the DriverManager.getConnection() method.

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String 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 |

## Create Connection Object:

**Using a database URL with a username and password:**

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

**Using only a database URL:**

```
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

**Using a database URL and a Properties object:**

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

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.

Following table provides a summary of each interface's purpose to understand how do you decide which interface to use?

| Interfaces | Recommended Use |
|---|---|
| **Statement** | Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters. |
| **PreparedStatement** | Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime. |
| **CallableStatement** | Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters. |

## The Statement Objects:

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   // Exception Logging
}
finally {
   stmt.close();
}
```



- **boolean execute(String SQL)** : Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

## The PreparedStatement Objects:

This statement gives you the flexibility of supplying arguments dynamically.

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

```
  . . .
}
catch (SQLException e) {
  . . .
}
finally {
  . pstmt.close();
}
```

All parameters in JDBC are represented by the **?** Symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The setXXX() methods bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

## ResultSet

The SQL statements that read data from a database query return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generated the ResultSet is created.

JDBC provides following connection methods to create statements with desired ResultSet:

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**

## Type of ResultSet:

| Type | Description |
|---|---|
| ResultSet.TYPE_FORWARD_ONLY | The cursor can only move forward in the result set. |
| ResultSet.TYPE_SCROLL_INSENSITIVE | The cursor can scroll forwards and backwards, and the result set is not sensitive to changes made by others to the database that occur after the result set was created. |
| ResultSet.TYPE_SCROLL_SENSITIVE. | The cursor can scroll forwards and backwards, and the result set is sensitive to changes made by others to the database that occur after the result set was created. |

# Concurrency of ResultSet:

The possible RSConcurrency are given below, If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

| Concurrency | Description |
|---|---|
| ResultSet.CONCUR_READ_ONLY | Creates a read-only result set. This is the default |
| ResultSet.CONCUR_UPDATABLE | Creates an updateable result set. |

```
try {
  Statement stmt = conn.createStatement(
                   ResultSet.TYPE_FORWARD_ONLY,
                   ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
  ....
}
finally {
  stmt.close();
}
```

There are several methods in the ResultSet interface that involve moving the cursor, including:

| Methods | Description |
|---|---|
| public void beforeFirst() throws SQLException | Moves the cursor to just before the first row |
| public void afterLast() throws SQLException | Moves the cursor to just after the last row |
| public boolean first() throws SQLException | Moves the cursor to the first row |
| public void last() throws SQLException | Moves the cursor to the last row. |
| public boolean absolute(int row) throws SQLException | Moves the cursor to the specified row |
| public boolean relative(int row) throws SQLException | Moves the cursor the given number of rows forward or backwards from where it currently is pointing. |
| public boolean previous() throws SQLException | Moves the cursor to the previous row. This method returns false if the previous row is off the result set |
| public boolean next() throws SQLException | Moves the cursor to the next row. This method returns false if there are no more rows in the result set |
| public int getRow() throws SQLException | Returns the row number that the cursor is pointing to. |
| public void moveToInsertRow() throws SQLException | Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered. |
| public void moveToCurrentRow() throws SQLException | Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing |

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions:

- One that takes in a column name.
- One that takes in a column index.

| Methods | Description |
|---|---|
| public int getInt(String columnName) throws SQLException | Returns the int in the current row in the column named columnName |
| public int getInt(int columnIndex) throws SQLException | Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on. |

Similarly there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type:

- One that takes in a column name.
- One that takes in a column index.

| Methods | Description |
|---|---|
| public void updateString(int columnIndex, String s) throws SQLException | Changes the String in the specified column to the value of s. |
| public void updateString(String columnName, String s) throws SQLException | Similar to the previous method, except that the column is specified by its name instead of its index. |

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

| Methods | Description |
|---|---|
| public void updateRow() | Updates the current row by updating the corresponding row in the database. |
| public void deleteRow() | Deletes the current row from the database |
| public void refreshRow() | Refreshes the data in the result set to reflect any recent changes in the database. |

| | |
|---|---|
| public void cancelRowUpdates() | Cancels any updates made on the current row. |
| public void insertRow() | Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row |

The JDBC driver converts the Java data type to the appropriate JDBC type before sending it to the database. It uses a default mapping for most data types.

| SQL | JDBC/Java | setXXX | updateXXX |
|---|---|---|---|
| VARCHAR | java.lang.String | setString | updateString |
| CHAR | java.lang.String | setString | updateString |
| LONGVARCHAR | java.lang.String | setString | updateString |
| BIT | boolean | setBoolean | updateBoolean |
| NUMERIC | java.math.BigDecimal | setBigDecimal | updateBigDecimal |
| TINYINT | byte | setByte | updateByte |
| SMALLINT | short | setShort | updateShort |
| INTEGER | int | setInt | updateInt |
| BIGINT | long | setLong | updateLong |
| REAL | float | setFloat | updateFloat |
| FLOAT | float | setFloat | updateFloat |
| DOUBLE | double | setDouble | updateDouble |
| VARBINARY | byte[ ] | setBytes | updateBytes |
| BINARY | byte[ ] | setBytes | updateBytes |
| DATE | java.sql.Date | setDate | updateDate |
| TIME | java.sql.Time | setTime | updateTime |
| TIMESTAMP | java.sql.Timestamp | setTimestamp | updateTimestamp |
| CLOB | java.sql.Clob | setClob | updateClob |
| BLOB | java.sql.Blob | setBlob | updateBlob |
| ARRAY | java.sql.Array | setARRAY | updateARRAY |
| REF | java.sql.Ref | SetRef | updateRef |
| STRUCT | java.sql.Struct | SetStruct | updateStruct |

## Commit & Rollback

If your JDBC Connection is in auto-commit mode, which it is by default, then every SQL statement is committed to the database upon its completion.Following code will turn off auto-commit:

```
conn.setAutoCommit(false);
```

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows:

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback( );
```

```
//STEP 1. Import required packages
import java.sql.*;

public class JDBCExample {
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";

  //  Database credentials
  static final String USER = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 4: Execute a query
    System.out.println("Creating statement...");
    Statement stmt = conn.createStatement();
    String sql;
    sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);

    //STEP 5: Extract data from result set
    while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");

      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
    }
    //STEP 6: Clean-up environment
    rs.close();
    stmt.close();
    conn.close();
  }catch(SQLException se){
    //Handle errors for JDBC
    se.printStackTrace();
  }catch(Exception e){
    //Handle errors for Class.forName
    e.printStackTrace();
  }finally{
    //finally block used to close resources
    try{
      if(conn!=null)
         conn.close();
    }catch(SQLException se){
```

```
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");
}//end main
}//end JDBCExample
```