Core Java

Java Database Connectivity

Lesson Objectives

- In this lesson, you will learn:
  - Introduction to JDBC
  - Database Connectivity Architecture
  - JDBC APIs
  - Database Access Steps
  - Using RowSets
  - Using Transactions
  - JDBC Best Practices

Introduction to JDBC

- Let us discuss Java Database Connectivity (JDBC) with respect to the following:
  - What is JDBC?
  - What does JDBC do?
  - Why JDBC?

- **Java Database Connectivity (JDBC)** is a standard SQL database access interface, providing uniform access to a wide range of relational databases.

- JDBC is used to allow Java applications to connect to the database and perform different data manipulation operations such as insertion, modification, deletion, and so on.

- JDBC provides different set of APIs to perform operations related to database.

What does JDBC do?

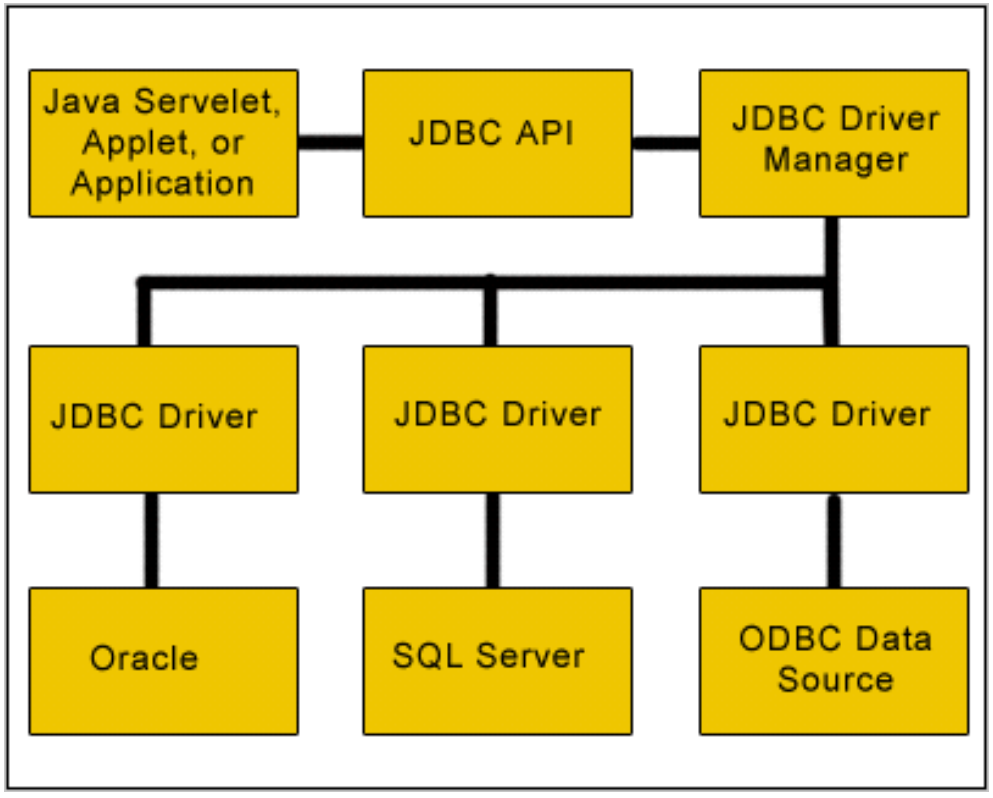
- JDBC allows us to construct SQL statements and embed them inside Java API calls.

- JDBC makes it possible to do three things:
  - Establish a connection with a database.
  - Send SQL statements.
  - Process the results.

Why JDBC?

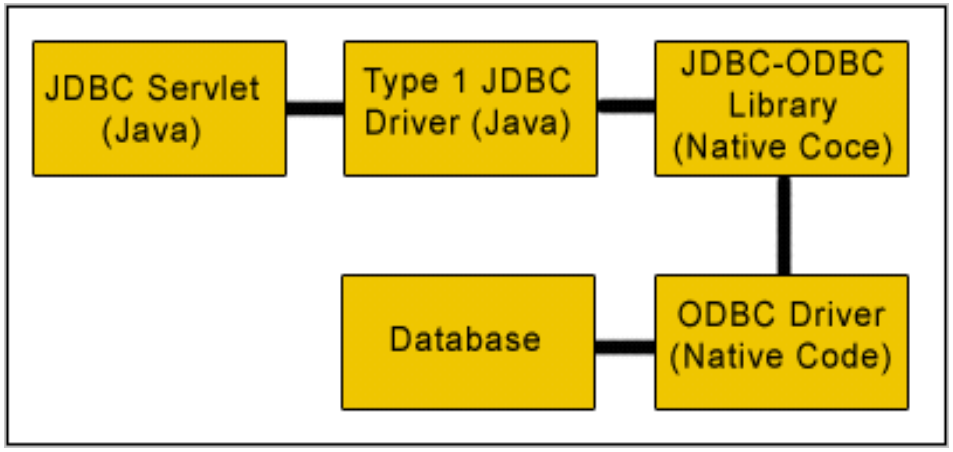
- JDBC exhibits the following features:
  - Java is a write once, run anywhere language.
  - Java based clients are thin clients.
  - It is suited for network centric models.
  - It provides a clean, simple, uniform vendor independent interface.
  - JDBC supports all the advanced features of latest SQL version
  - JDBC API provides a rich set of methods.
  - With the JDBC API, no configuration is required on the client side.

- Layers of JDBC Architecture



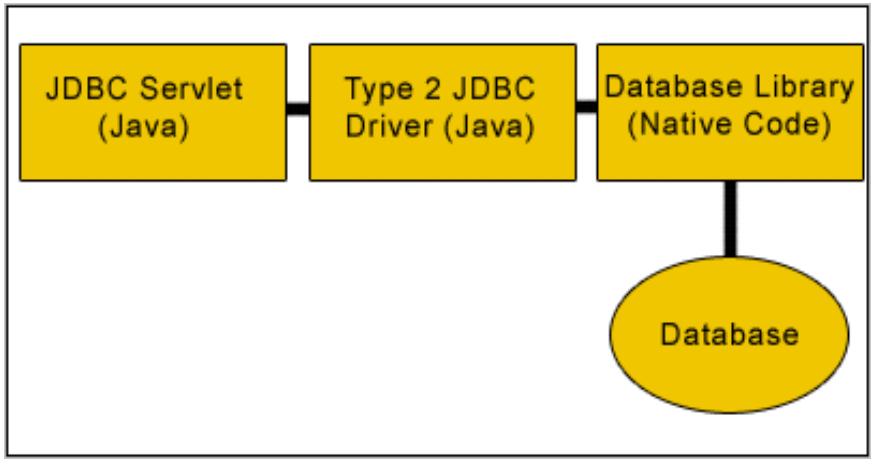
Type 1 – JDBC-ODBC Bridge

- Type 1 – JDBC-ODBC Bridge



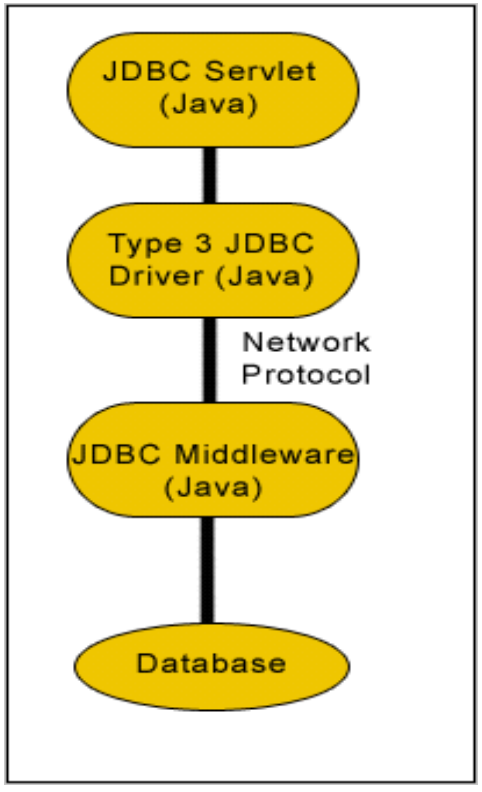
Type 2 – Java Native API

- Type 2 - Java Native API



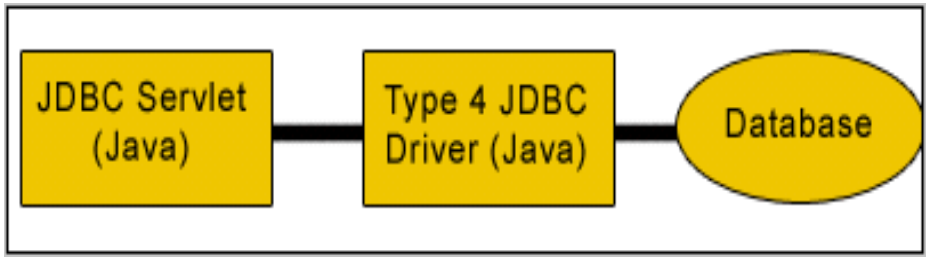
Type 3 – Java to Network Protocol

- Type 3 – Java to Network Protocol



Type 4 – Java to Database Protocol

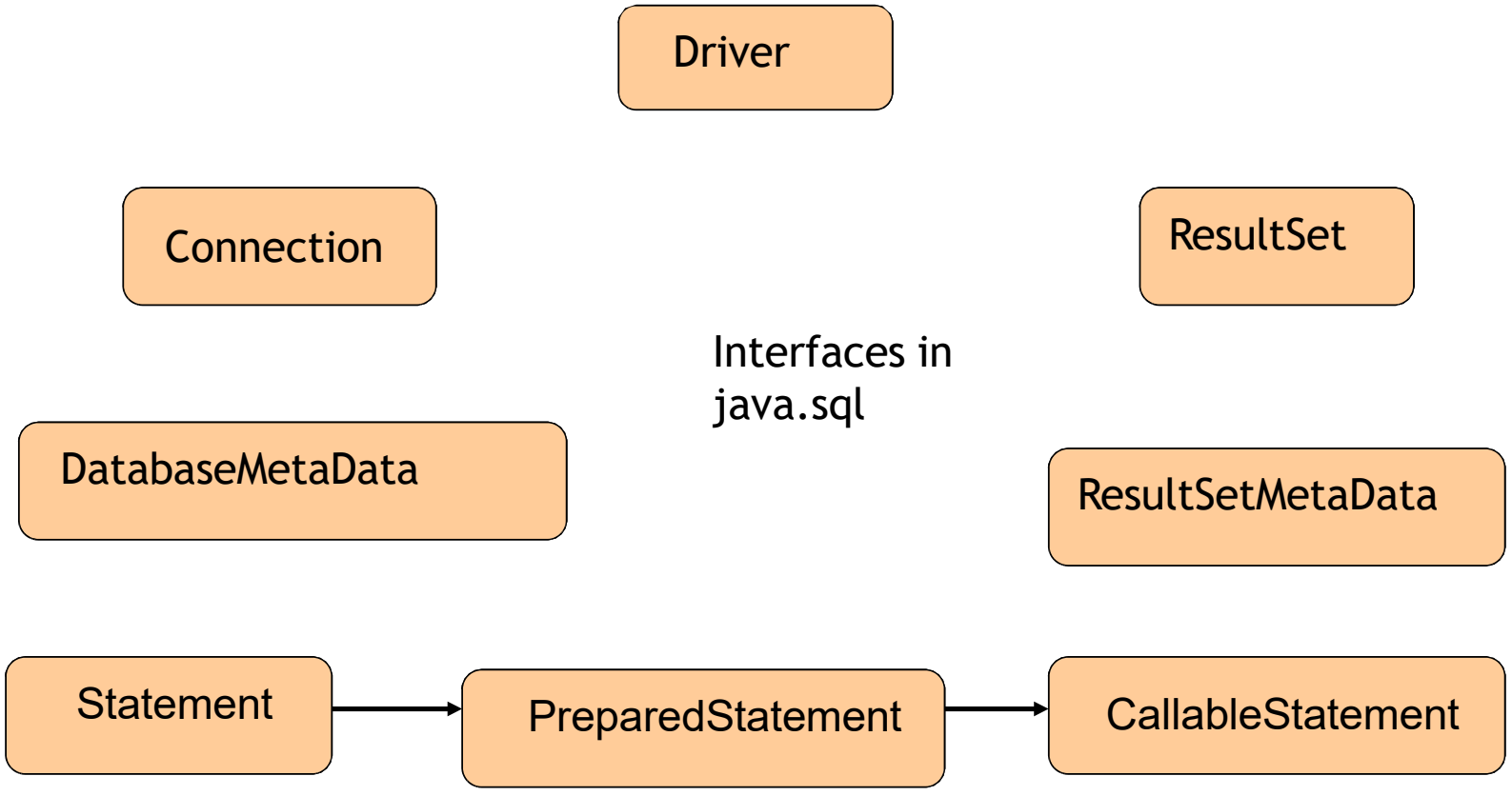
- Type 4 - Java to Database Protocol



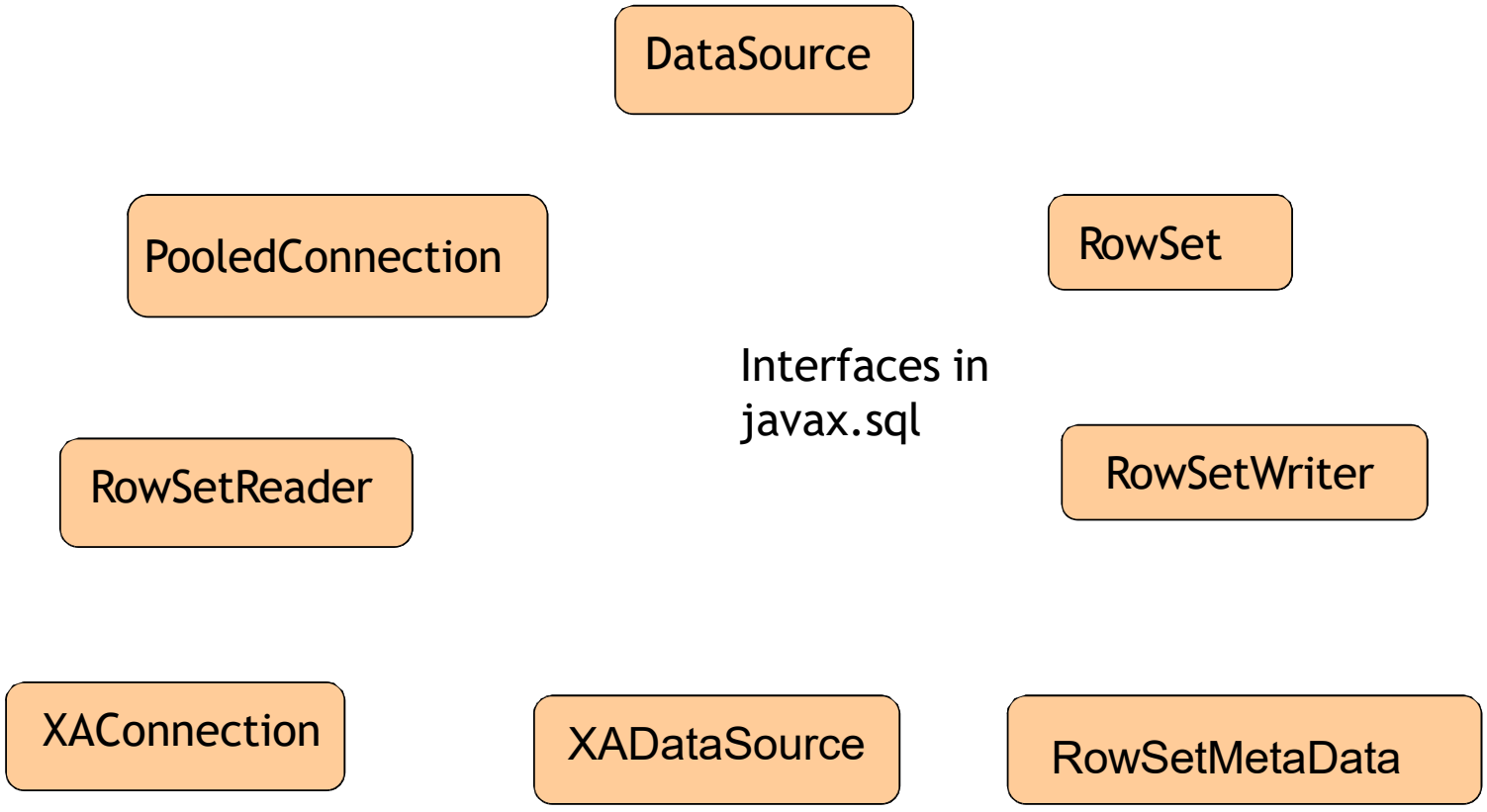
JDBC API Packages

- JDBC packages:
  - java.sql.*
  - javax.sql.*

java.sql package



javax.sql package



Database Access Steps

- Database Access takes you through the following steps:
  - Import the packages
  - Register/load the driver
  - Establish the connection
  - Creating JDBC Statements
  - Getting Data from a Table
  - Insert data into Table
  - Update table data

Import Packages

- **Step 1:** Import the java.sql and javax.sql packages
  - These packages provides the API for accessing and processing data stored in a data source.
  - They include:
    - import java.sql.*;
    - import javax.sql.*;

Register Driver

- **Step 2:** Register/load the driver
  - Class.forName("oracle.jdbc.driver.OracleDriver");
  - Class.forName("com.mysql.jdbc.Driver");

  OR

  - DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
  - DriverManager.registerDriver (new com.mysql.jdbc.Driver());

Establish the Connection

- **Step 3:** Establish the connection with the database using registered driver
  - String url = "jdbc:oracle:thin:@hostname:1521:database";
  - Connection conn = DriverManager.getConnection (url, "scott", "tiger");
  - String url = "jdbc:mysql://localhost:3306/test";
  - Connection conn = DriverManager.getConnection (url, “root", “root”);

Create Statement

- **Step 4:** Create Statement
  - Statement:
    - Statement st=conn.createStatement();
  - PreparedStatement:
    - PreparedStatement pst=conn.prepareStatement(“select * from emp where eno=?”);

      pst.setInt(1,100);
  - CallableStatement:
    - CallableStatement cs=conn.prepareCall(“{ call add() }”) ;

Retrieve Data from Table

- **Step 5:** Retrieve data from table
  - Statement:

```
Statement st=conn.createStatement();
ResultSet rs=st.executeQuery("select * from emp");
while(rs.next()){
System.out.println("Emo No = "+rs.getInt("eno"));
System.out.println("Emo Name = "+rs.getString("ename"));
}
```

  - Output:
    - Display the no and name of all employees

Insert Data into Table

- **Step 6:** Insert data into table
  - Statement:

```
Statement st=conn.createStatement();
st.executeUpdate("insert into emp values(110,'XYZ','Pune')");
System.out.println(" New Record is inserted"));
```

Update Table Data

- **Step 7:** Update table data
  - Statement:

```
Statement st=conn.createStatement();
st.executeUpdate("update emp set ecity='Mumbai' where
eno<1000)");
System.out.println("Table updated"));
```

Demo: Database Access Demonstration

- Demo with the following:
  - Select
  - Insert
  - Delete
  - Parameterized Query

Using RowSets

- A **RowSet object** contains a set of rows from a result set or some other source of tabular data, such as a file or spreadsheet.
- RowSet types include the following:
  - **JdbcRowSet:** It is a connected scrollable, updatable RowSet
  - **CachedRowset:** It is a disconnected RowSet
  - **WebRowSet:** It is a connected RowSet that uses the HTTP protocol internally to talk to a Java Servlet that provides data access.
  - **FilteredRowSet:** It is a WebRowSet with content filter.
  - **JoinRowSet:** It joins multiple RowSets.

JdbcRowSet

- A **JDBCRowSet** is a connected RowSet, which has a live connection to the database.

```
...
RowSet rowset = new OracleJDBCRowSet();
rowset.setUrl("jdbc:oracle:thin:@192.168.67.177:trgdb”);
rowset.setUsername(“user2");
rowset.setPassword(“user2");
rowset.setCommand("SELECT empno, ename, sal FROM
emp");
rowset.execute();
while (rowset.next()) {
System.out.println("empno: " + rowset.getInt(1));
System.out.println("ename: " + rowset.getString(2));
System.out.println("sal: " + rowset.getInt(3));
} ...
```

CachedRowSet

- A **CachedRowSet** is a RowSet in which the rows are cached and the RowSet is disconnected.

```
...
RowSet rowset = new OracleCachedRowSet();
rowset.setUrl("jdbc:oracle:thin:@192.168.67.177:trgdb”);
rowset.setUsername(“user2");
rowset.setPassword(“user2");
rowset.setCommand("SELECT empno, ename, sal FROM
emp");
rowset.execute();
while (rowset.next()) {
System.out.println("empno: " + rowset.getInt(1));
System.out.println("ename: " + rowset.getString(2));
System.out.println("sal: " + rowset.getInt(3));
} ...
```

Using Transaction

- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.
- Java application uses Java Transaction API(JTA) to manage the transactions.

Disabling Auto-commit Mode

- When a connection is created, it is in **auto-commit** mode.
- To allow two or more statements to be grouped into a transaction is to disable auto-commit mode.
- Example:
  - con.setAutoCommit(false);

Committing a Transaction

- After you complete execution of all the SQL statements, you should commit the transaction by calling **commit().**

```
 ...
con.setAutoCommit(false);
PreparedStatement updateSal = con.prepareStatement(
"UPDATE Employee set sal =sal+1000   WHERE empno>?");
updateSal.setInt(1, 50);
updateSales.executeUpdate();
PreparedStatement updateDept = con.prepareStatement(
"UPDATE Employee set deptno=10 WHERE empno>?");
updateTotal.setInt(1, 30);
updateDept.executeUpdate();
con.commit();
...
```

When do you call the Rollback Method

- The **rollback()** method plays an important role in preserving data integrity in the transaction.
- When you want to undone the changes of half done transaction due to SQLException:

```
        ...
try {
        con.setAutoCommit(false);
        // perform transactions
        con.commit();
        con.setAutoCommit(true);
    } catch (SQLException e) {
      con .rollback() ;
       }
```

Demo : Transactions

- Transaction

Connection Pooling

- Let us discuss Connection Pooling with respect to the following points:
  - Introduction to Connection Pooling
  - DataSource Object
  - DataSource versus DriverManager

- Acquiring and releasing connection is expensive.
- **Database connection pooling** promotes reusability of the connection.
- It increases the application performance, concurrency, and scalability.
- It is implemented as DataSource Object.

DataSource Object

- **DataSource object** can be thought of as a factory for connections to the particular data source.
- A DataSource has a set of properties that identify and describe the real world data source.
- A data source can reside on a **remote server**, or it can be on a **local desktop machine**.
- A call to the **getConnection()** method returns a connection object.

Example

- Let us see an example for using DataSource Object:

```
...
OracleDataSource ods = new OracleDataSource();

   ods.setUser(“user2");
   ods.setPassword(“user2");
   ods.setDriverType("thin");
   ods.setNetworkProtocol("tcp");

ods.setURL("jdbc:oracle:thin:@192.168.67.177:1521:trgdb")
;
   Connection conn = ods.getConnection();
```

contd.

Example

```
Statement stmt = conn.createStatement ();
 ResultSet rset = stmt.executeQuery ("select ename from Emp");
while (rset.next ()){
 System.out.println (“Employee name is " + rset.getString (1));
 }
```

DataSource versus DriverManager

- Let us do a quick comparison between DataSource and DriverManager:

DataSource	Driver Manager
Link with database is retained after closing the database connection.	Link is lost.
Database connection is reusable.	Database connection is not reusable.
Connection pooling feature is used.	Connection pooling is not implemented.
Distributed transactions are possible.	Distributed transactions are not possible
JNDI lookup is used.	JNDI lookup is not used.

Demo : DataSource

- DataSource

JDBC Best Practices

- Let us discuss some of the best practices in JDBC:
  - Selection of Driver
  - Tune connection pool size
  - Close resources as soon as you're done with them
  - Turn-Off Auto-Commit – group updates into a transaction