JDBC Objects

Module 5

JDBC Objects: The Concept of JDBC
JDBC Driver Types; JDBC Packages
A Brief Overview of the JDBC process
Database Connection; Associating the JDBC/ODBC Bridge with the Database
Statement Objects, ResultSet
Transaction Processing
Metadata, Data types
Exceptions

The Concept of JDBC

Java Database Connectivity (JDBC):

•**Purpose**: Enables Java programs to interact with databases.

•**Functionality**: Provides methods for

--connecting to databases

-- executing SQL queries

-- manipulating data.

•**Components**: Uses **JDBC drivers** tailored for different database systems **(MySQL, Oracle, PostgreSQL, SQL Server, etc).**

Features:

-- It is a standard DB independent API to connect and execute query with the database

-- It's an **advancement of ODBC (platform dependent)** and acts as a interface between java application and database

-- It is a platform independent DB access/ technology

--Supports transactions management

Benefits: Facilitates seamless **integration of database operations within Java applications**.

Components of JDBC

There are four main components in JDBC architecture through which it can interact with a database.

1. **JDBC API:** It provides various classes and interfaces for easy communication with the database.
   - ✓ It provides two packages , which contain the java SE and Java EE platforms to exhibit WORA(write once run anywhere) capabilities.
   - ✓ The java.sql package contains interfaces and classes of JDBC API.

java.sql: This package provides APIs for data access and data process in a relational database, included in Java Standard Edition (java SE)

javax.sql: This package extends the functionality of java package by providing **datasource interface** for establishing connection pooling, statement pooling with a data source, included in Java Enterprise Edition (java EE)

--It also provides set of a standards to connect a database to a client application.

2. JDBC Driver manager:

-- is a **class** in **JDBC API**

that **loads a database-specific driver** in java application and **establishes a connection with a database.**

-- It also used to makes a **call to specific database** to the user request.

3. **JDBC Test suite**: It is used to **test the operations**(**insertion, deletion, updation**) being performed on DB's using JDBC Drivers.

4. **JDBC-ODBC Bridge Drivers**:

--It connects **database drivers to the database**.

--This **bridge translates the JDBC method call to the ODBC function call.**

--It makes use of the **sun.jdbc.odbc** package which includes a **native library to access ODBC characteristics**.

-- The bridge translates the **OO JDBC method call** to the **Procedural function call**

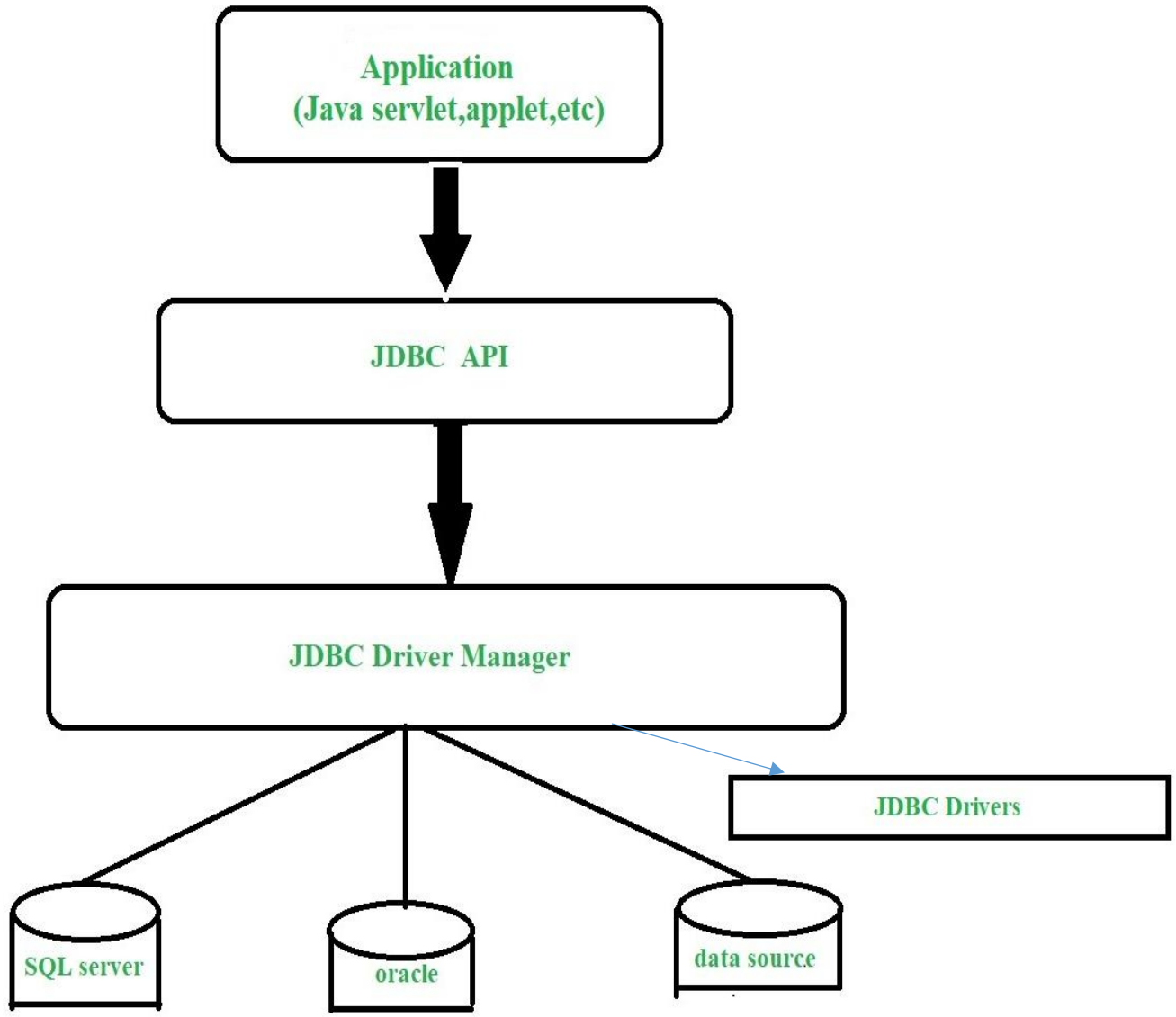
1.Application: It is a java applet or a servlet that communicates with a data source.

2.The JDBC API: The JDBC API **allows Java programs to execute SQL statements and retrieve results.**
--The important interfaces defined in JDBC API are as follows:
Driver interface , ResultSet Interface , RowSet Interface , PreparedStatement interface, Connection inteface,
--Classes defined in JDBC API are **DriverManager class, Types class, Blob class, clob class.**

3.DriverManager: It uses **some database-specific drivers to effectively connect enterprise applications to databases.**

4.JDBC drivers: To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

Architecture of JDBC



- Data sources are the db's that we can connect using this API

JDBC Driver Types

- JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand.
- **JDBC drivers** are the software components which implements interfaces in JDBC APIs to enable java application to interact with the database.
- JDBC uses **drivers** to translate generalized **JDBC calls into vendor-specific database calls**

Four Classes of JDBC drivers

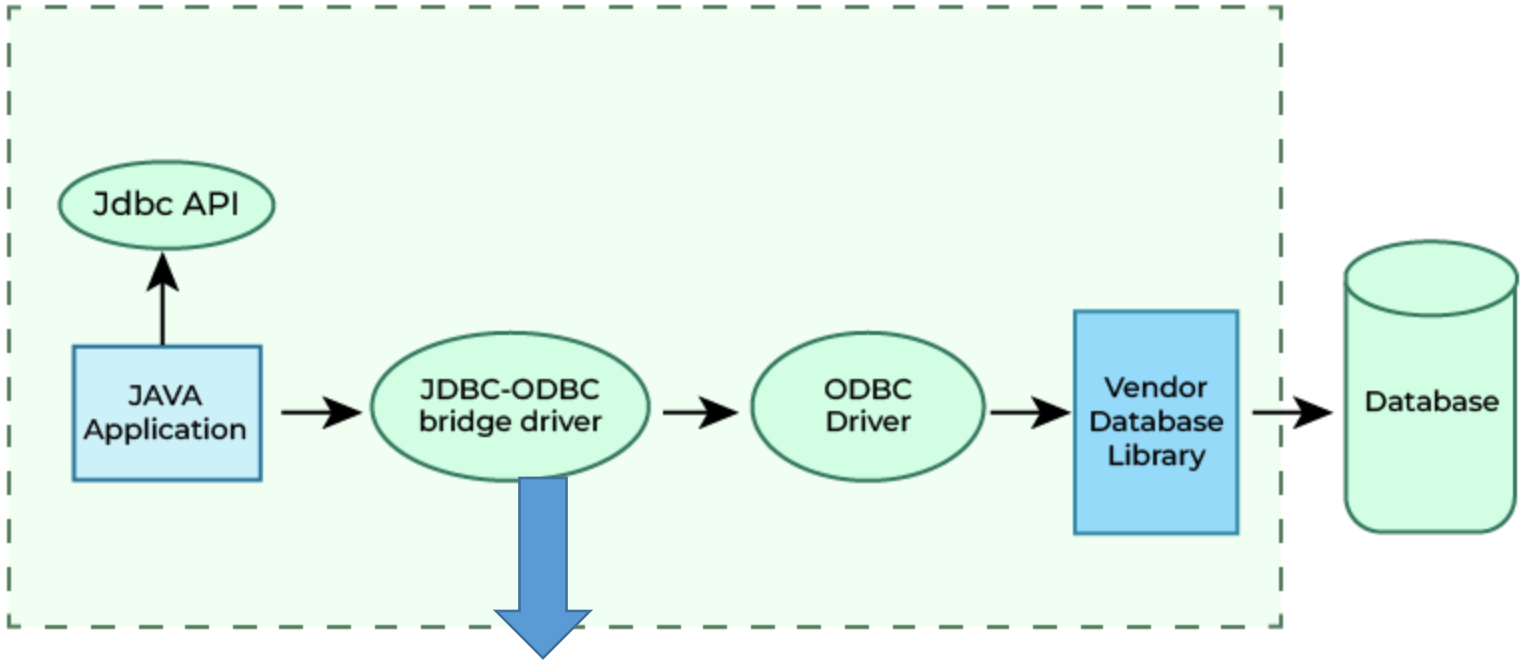
Type I- **JDBC to ODBC Bridge**
Type II – **Native API Driver/ Partially Java Driver**
Type III – **Network Protocol Driver/ Fully Java Driver**
Type IV- **Thin Driver /fully java Driver**

JDBC-ODBC bridge driver – Type 1 driver

Type-1 driver or JDBC-ODBC bridge driver **uses ODBC driver to connect to the database.**

The JDBC-ODBC bridge driver **converts JDBC method calls into the ODBC function calls.**

Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.



- uses the ODBC (Open Database Connectivity) API to communicate with the database.
- translates JDBC method calls into ODBC function calls.

  Therefore, to use this driver, an ODBC driver must be installed on the client machine where the Java application runs.

Advantages

- This **driver software** is built-in with JDK so **no need to install** separately.
- It is a database independent driver.

•Easy to use, especially for developers familiar with ODBC.

•Supports all ODBC-compliant databases.

•**Disadvantages:**

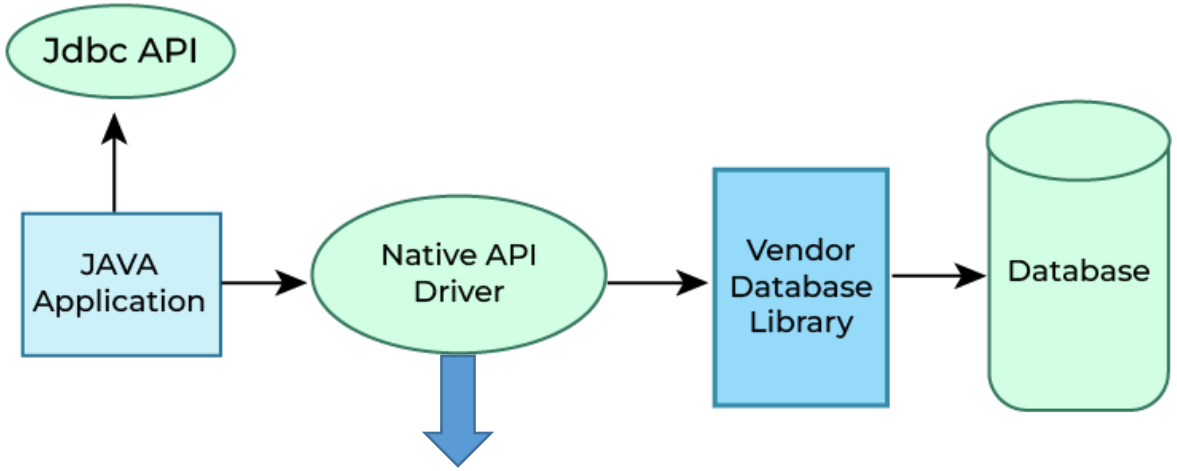
•Performance overhead due to multiple translation layers (JDBC to ODBC to database-specific protocol).

•Platform dependent and may not be suitable for deploying in different environments without ensuring ODBC driver availability.

JDBC

Drivers (Type II) Native API driver/ partly Java driver

- **In order to interact with different database, this driver needs their local API**, that's why **data transfer is much more secure** as compared to type-1 driver.
- This driver is not fully written in Java that is why it is also called Partially Java driver.



- uses a database-specific native client library (written in C, C++, etc.) to communicate directly with the database.
- It converts JDBC calls into native calls of the database API.

Advantage

•Native-API driver gives better performance than JDBC-ODBC bridge driver.

•Platform-specific, but does not require ODBC, making it somewhat more portable than Type 1.

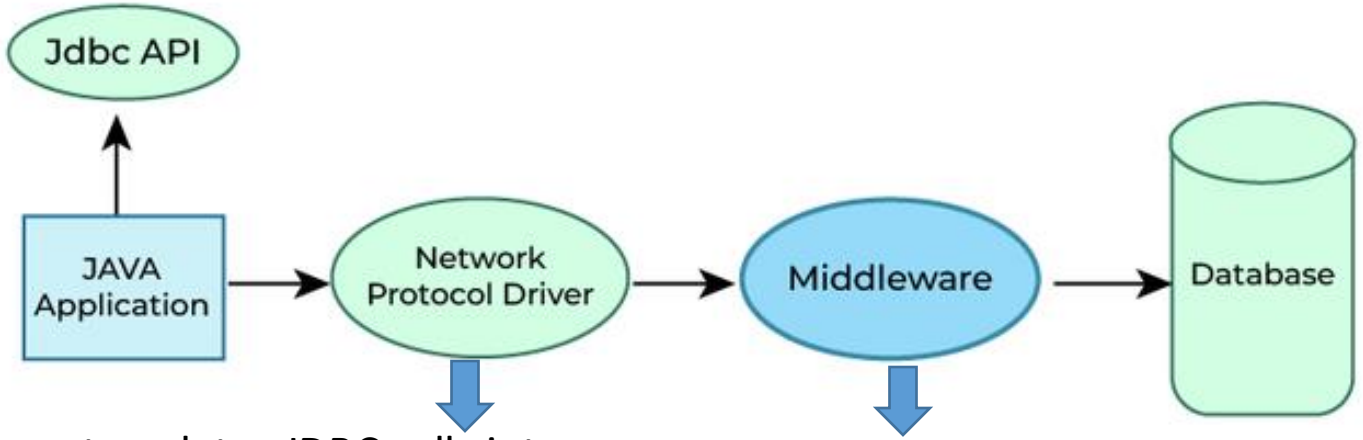
Disadvantages

- **Driver needs to be installed separately in individual client machines**
- Requires database-specific native library installation on client machines.
- Not entirely platform-independent.

JDBC

Drivers (Type III) Network Protocol Driver /(Pure Java driver) /middleware JDBC driver

- The Network Protocol driver uses **middleware (application server)** that **converts JDBC calls directly or indirectly into the vendor-specific database protocol.**
- **All the database connectivity drivers are present in a single server**, hence no need of individual client-side installation.



translates JDBC calls into a database-independent network protocol.

This server is a Java-based application server.
-Acts as an intermediary between the Java application and the database.

Advantages

•Type-3 drivers are fully written in Java, hence they are portable drivers.

•**No client side library** is required because of **application server** that can perform many tasks like auditing, load balancing, logging etc.

•**Switch facility** to switch over from one database to another database.

Disadvantages

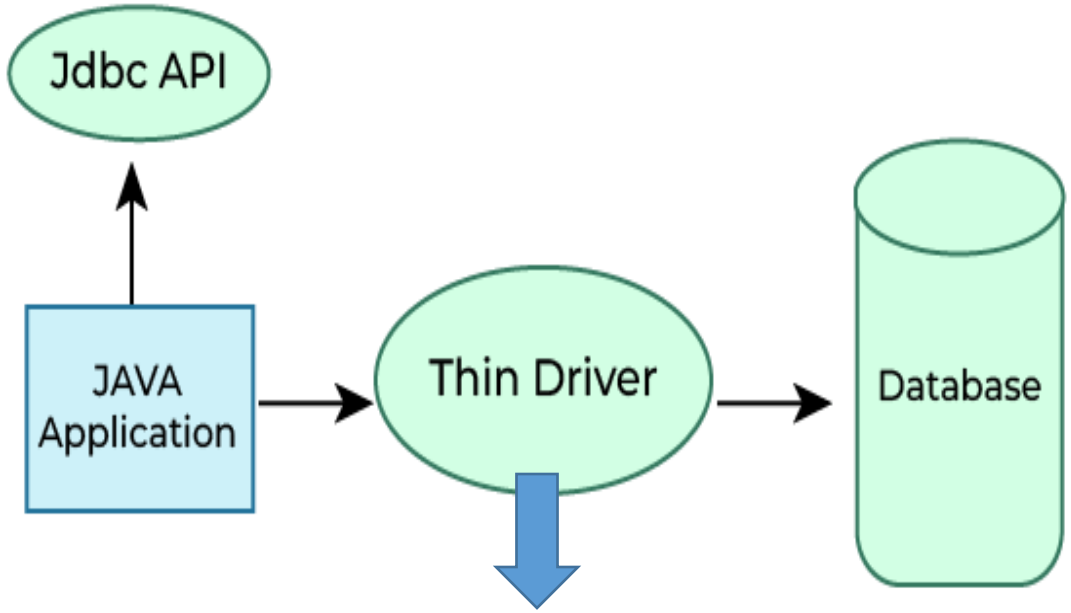
•Network support is required on client machine.

•Requires middleware server installation and configuration.

JDBC Drivers (Type IV) /(Direct-to-database pure Java driver)

--This driver interact directly with database.

--It does not require any native database library or middleware server, that is why it is also known as Thin Driver.



converts JDBC calls directly into the vendor-specific database protocol using Java libraries.

Advantages

- Fastest and most efficient JDBC driver type due to direct database communication.
- Platform-independent; runs on any system with Java support.
- Easy deployment and maintenance as it does not rely on external native components.

Disadvantage

- Vendor-specific; may require different drivers for different databases.
- Limited to databases that have a Type 4 JDBC driver available.

Summary

- **Type 1:** JDBC-ODBC Bridge driver. **Easy to use** but **has performance and platform dependency issues**.
- **Type 2:** Native-API, partly Java driver. Better performance than Type 1 but **requires native client library installation.**
- **Type 3:** Network Protocol driver (Pure Java driver). Provides **platform independence and centralized management but requires a middleware server.**
- **Type 4:** Thin driver (Direct-to-database pure Java driver). Fastest and most efficient, fully Java-based, and easy to deploy and maintain.

Steps to Connect Java Application with Database

The following steps are used to connect to a Database in Java:

Step 1 – Import the Packages

Step 2 – Load the drivers using the *forName() method*

Step 3 – Register the drivers *using DriverManager*

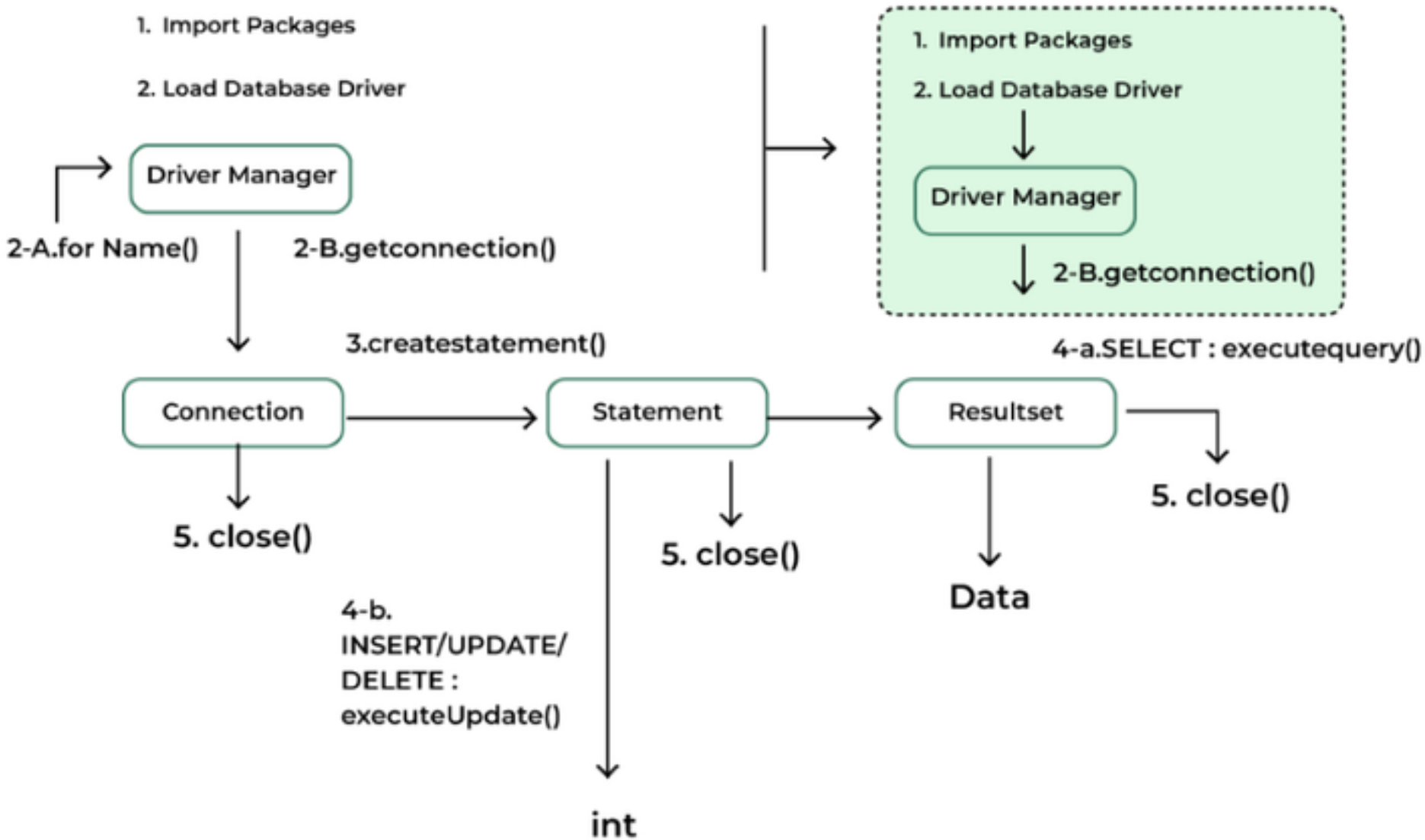
Step 4 – Establish a **connection** *using the* ***Connection class*** *object*

Step 5 – Create a statement

Step 6 – Execute the query

Step 7 – Close the connections

Java Database Connectivity



Step 1: Import the Packages:

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

Step 2: Loading the drivers : Before connecting to the database load and register appropriate JDBC drivers.

- Different databases have different drivers.
- Registration is to be done once in your program.
- You can **register a driver** in **one of two ways**.

2-A Class.forName()

- Load the **driver's class file** into **memory at the runtime**. No need of using **new or create objects.**

Below example uses **Class.forName()** to load the **Oracle driver.**

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

For mysql

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

2-B DriverManager.registerDriver()

- DriverManager is a Java inbuilt class with a static member register.
- **call the constructor** of the driver class at **compile time.**

For ex: Oracle driver registration

```
DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver())
```

Step 3: Establish a connection using the Connection class object

After loading the driver, establish connections as shown below :

Connection con = DriverManager.getConnection(url,user,password)

Where,

User, password: **Username & password from which your SQL command prompt can be accessed**.

con: **It is a reference to the Connection interface**.

Url: Uniform Resource Locator of the database and is created as below:

String url = " jdbc:oracle:thin:@localhost:1521:xe"

- Where oracle is **the database used**, thin is the **driver used**, @localhost is **the IP Address** where a **database is stored**, 1521 is **the port number** and xe is the **service provider**.
- All 3 parameters are of **String type** and are to be declared by the programmer before calling the function.

For Example:

```
String url = "jdbc:mysql://localhost:3306/database_name";
String user = "username";
String password = "password";
Connection conn = DriverManager.getConnection(url, user, password);
```

Step 4: Create a statement :

- Once a connection is established you can interact with the database.
- The **JDBCStatement, CallableStatement, and PreparedStatement interfaces** define the **methods** that enable you to **create SQL Statement.**

Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step --.creation of statement object.

Step 5: Execute the SQL queries

The most important part i.e executing the query. (SQL Query).

Multiple types of queries

•The query for **updating/inserting** a table in a database.

•The query for **retrieving data.**

- The **executeQuery()** method of the **Statement interface** is used to **execute queries of retrieving values** from the database.
    - **returns the object** of **ResultSet** that can be used to get all the records of a table.
- The **executeUpdate(sql query)** : of **Statement interface** is used to execute queries of DML or DDL operations **that do not return data**.

Example Pseudo Code:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM table_name");

while (rs.next()) {

   // Process the result set row

}
```

Ex:-

```
int m = st.executeUpdate(sql);
if (m==1)
   System.out.println("inserted successfully : "+sql);
else
   System.out.println("insertion failed");
```

Step 6: Closing the connections :

After you are done with the database operations, it's important to close the connections properly to release resources.

- By closing the connection, objects of **Statement and ResultSet** will be closed automatically.
- The **close() method** of the **Connection interface** is used to close the connection.

Ex:-

```
rs.close();
stmt.close();
con.close();
```

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

public class JDBCTest {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String user = "root";
        String password = "password";

        try {
            // Load and register the driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish connection
 Connection conn = DriverManager.getConnection(url, user, password);

            // Create statement
            Statement stmt = conn.createStatement();

 // Execute query
    ResultSet rs = stmt.executeQuery("SELECT * FROM users");

            // Process the result set
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String email = rs.getString("email");
                System.out.println("ID: " + id + ", Name: " + name + ",
Email: " + email);
            }

            // Close connections
            rs.close();
            stmt.close();
            conn.close();

        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Statement Objects- JDBC

- **Statement objects** allow you to execute basic SQL queries and retrieve the results through the ResultSet class.

Types of Statements in JDBC

- **The statement interface** is used to **create SQL basic statements in Java**
- provides **methods to execute queries** with the database.
- Different types of statements that are used in JDBC are…
    - Create Statement: which executes a query immediately
    - Prepared Statement: Which is used to execute compiled query
    - Callable Statement: which is used to execute strored procedure

1. **Create a Statement:**

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

Syntax:

Statement statement = connection.createStatement();

- Once the Statement object is created, there are **three ways to execute it.**

  - **boolean execute(String SQL): If the ResultSet object is retrieved, then it returns true else false is returned. When returns multiple results**

    -And 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. Int indicates number of rows get affected.**

    - used when you need a number for INSERT, DELETE or UPDATE statements.

  - **ResultSet executeQuery(String SQL): Returns one ResultSet object which comprises of rows, columns and metadata**

    -Used similarly as SELECT is used in SQL.

2. Prepared Statement:

- It represents a **recompiled SQL statement**, that can be executed many times.
- This accepts parameterized SQL queries.
- In this, “?” is used instead of the parameter, one can pass the parameter dynamically by using the methods of PREPARED STATEMENT at run time.

Illustration:

- Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

- INSERT INTO people VALUES ("Ayan",25);
- INSERT INTO people VALUES("Kriya",32);

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

```
String query = "INSERT INTO people(name, age)VALUES(?, ?)";

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setString(1,"Ayan");

ptstmt.setInt(2,25);

// where pstmt is an object name
```

Implementation:

Once the PreparedStatement object is created, there are **three ways to execute it:**

•***execute():*** This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.

•***executeQuery():*** Returns a ResultSet from the current prepared statement.

•***executeUpdate():*** Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in the current Prepared Statement.

Callable statement

- Callable statement object is used **to call Stored procedure(SP)**.
- SP is **a block of code and is identified by unique name**. the type and style of the code depends on DBMS vendor.
- Written using PL/SQL, Transact SQL, c
- **Executed** using the name of the SP
- Callable statement object uses 3 types of parameters when calling a SP. (IN,out,inOUT)
- **In** contains any data that needs **to be passed to the SP** and **values to it will be assigned using SETXXX()**
- **Out params contains the value** retured by the SP. Out parameter must be registered using **registerOutParameter().** And then it is retrieved by J2EE component using getxxx()
- INOUT param is a single param and does both

Callable Statement:

Syntax: To prepare a CallableStatement

CallableStatement cstmt = con.prepareCall("{call Procedure_name(?, ?}");

Implementation:

Once the callable statement object is created

execute() is used to perform the execution of the statement.

ResultSet interface

- The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.
- By default, ResultSet object can be moved forward only and it is not updatable.
- But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Reading the data From the Resultset

Listing 6-13 Reading data from the ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
```



```
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
```

```
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement();
      Results = DataRequest.executeQuery (query);
}
catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
boolean Records = Results.next();

if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
  do {
     FirstName = Results.getString ( 1 ) ;
      LastName = Results.getString ( 2 ) ;

      printrow = FirstName + " " + LastName;
     System.out.println(printrow);

  } while (Results.next() );
  DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Scrollable Resultset

Scrollable means that once the *ResultSet* object has been created, we can traverse through fetched records in any direction, forward and backward, as we like. This provides the ability to read the last record, first record, next record, and the previous record.

Note that the scrollable *ResultSet* object is the result of the execution of the *executeQuery()* method obtained through the instance of *Statement* or *PreparedStatement*.

The type of *ResultSet* object we like to create must be explicitly declared to the *Statement* object through defined scroll type constants.

- **ResultSet.TYPE_FORWARD_ONLY:** This is the default type.
- **ResultSet.TYPE_SCROLL_INSENSITIVE:** Enables back and forth movement, but is insensitive to *ResultSet* updates.
- **ResultSet.TYPE_SCROLL_SENSITIVE:** Enables back and forth movement, but is sensitive to *ResultSet* updates.

There are other constants used, such as *CONCUR_READ_ONLY*, which means that the *ResultSet* is not updatable. There is another constant, *CONCUR_UPDATABLE*, which signifies the opposite,

Using Scrollable Resultset

Listing 6-14 Using a scrollable virtual cursor.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
String printrow;
String FirstName;
String LastName;
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
      DataRequest = Db.createStatement(TYPE_SCROLL_INSENSITIVE);
      Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}
```

```
boolean Records = Results.next();
if (!Records ) {
    System.out.println("No data returned");
    System.exit(4);
}
try {
   do {
```

```
      Results.first();
     Results.last();
     Results.previous();
     Results.absolute(10);
     Results.relative(-2);
     Results.relative(2);
      FirstName = Results.getString ( 1 ) ;
      LastName = Results.getString ( 2 ) ;
      printrow = FirstName + " " + LastName;
     System.out.println(printrow);
  } while (Results.next() );
  DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Update ResultSet

Once the executeQuery() method of the Statement object returns a ResultSet, the updatexxx() method is used to change the value of a column in the current row of the ResultSet. The xxx in the updatexxx() method is replaced with the data type of the column that is to be updated.

The updatexxx() method requires two parameters. The first is either the number or name of the column of the ResultSet that is being updated and the second parameter is the value that will replace the value in the column of the ResultSet.

A value in a column of the ResultSet can be replaced with a NULL value by using the updateNull() method. The updateNull() method requires one parameter, which is the number of the column in the current row of the ResultSet. The updateNull() doesn't accept the name of the column as a parameter.

The updateRow() method is called after all the updatexxx() methods are called. The updateRow() method changes values in columns of the current row of the ResultSet based on the values of the updatexxx() methods.

Listing 6-17 illustrates how to update a row in a ResultSet. In this example, customer Mary Jones was recently married and changed her last name to Smith before processing the ResultSet. The updateString() method is used to change the value of the last name column of the ResultSet with 'Jones'. The change takes effect once the updateRow() method is called; however, this change only occurs in the ResultSet. The corresponding row in the table remains unchanged until an update query is run, which is discussed in the next chapter.

Listing 6-17 Updating the ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
}
```

```
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers WHERE
FirstName = 'Mary' and LastName = 'Smith'";
      DataRequest = Db.createStatement(ResultSet.CONCUR_UPDATABLE);
      Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}

boolean Records = Results.next();
if (!Records ) {
   System.out.println("No data returned");
   System.exit(4);
}
try {
      Results.updateString ("LastName", "Smith");
     Results.updateRow();
     DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Delete Row()

Results.deleteRow(0): deletes the current row

Insert Row into the resultset

Listing 6-18 Inserting a new row into the ResultSet.

```
String url = "jdbc:odbc:CustomerInformation";
String userID = "jim";
String password = "keogh";
Statement DataRequest;
ResultSet Results;
Connection Db;
try {
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver");
   Db = DriverManager.getConnection(url,userID,password);
 }
catch (ClassNotFoundException error) {
      System.err.println("Unable to load the JDBC/ODBC bridge." + error);
      System.exit(1);
}
catch (SQLException error) {
      System.err.println("Cannot connect to the database." + error);
      System.exit(2);
}
try {
      String query = "SELECT FirstName,LastName FROM Customers";
```

```
        DataRequest = Db.createStatement(CONCUR_UPDATABLE);
        Results = DataRequest.executeQuery (query);
}

catch ( SQLException error ){
     System.err.println("SQL error." + error);
      System.exit(3);
}

boolean Records = Results.next();
if (!Records ) {
    System.out.println("No data returned");
   System.exit(4);
}
try {
      Results.updateString (1, "Tom");  // updates the ResultSet
     Results.updateString (2, "Smith"); // updates the ResultSet
     Results.insertRow(); // updates the underlying database
     DataRequest.close();
}
catch (SQLException error ) {
   System.err.println("Data display error." + error);
   System.exit(5);
}
```

Connection Pool

Connecting to a db is performed on per client basis and cannot be shared with unrelated clients.

Connection pool is a **collection of db connections** that **are opened once and loaded on to memory** so these connection can be reused.

Clients use DataSource interface to interact with the connection pool

Connection pool itself is implemented by Application Server and it is hidden from the client

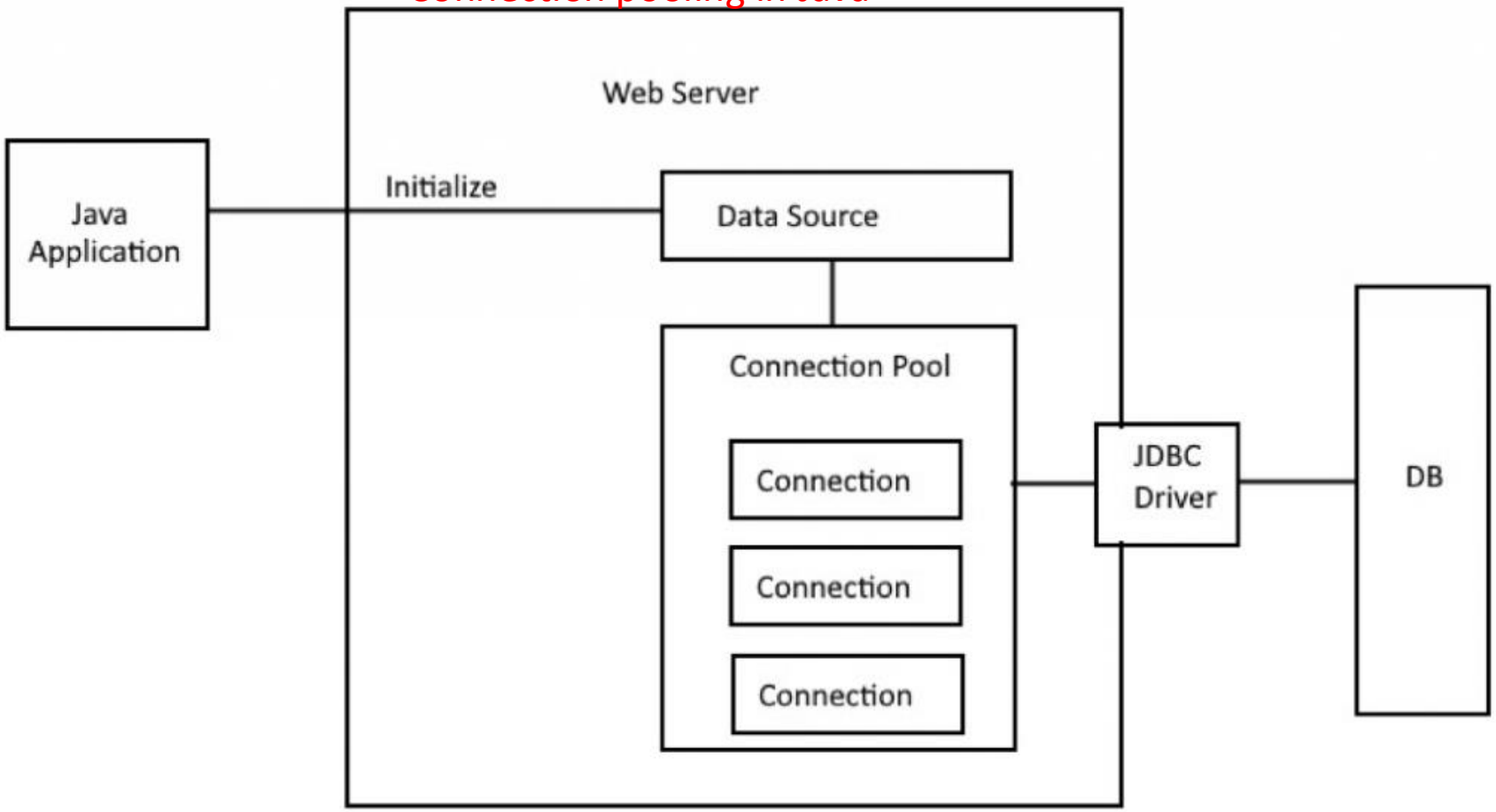
There are two types of connection made to the database.

-**Physical Connection** made by the application server using pooled connection objects.

-- these objected are cached and reused

- Logical Connection: made by client by calling the DataSource.getconnection(), which connects to the pooledconncetion object that already has a physical connection to the database.

Connection pooling in Java



We can create our own implementations of Connection pooling. Any connection pooling framework needs to do three tasks.

- Creating Connection Objects
- Manage usage of created Objects and validate them
- Release/Destroy Objects

How to access a connection from a Connection Pool

- **Connection pool** is accessed using **JNDI (Java Naming and Directroty Interface)**
  - It provides a **uniform way to find and access naming and directory services** which is independent of Naming and Directory services.
- **steps**
- **First J2EE component must obtain a handle to** the **JNDI Context**
- Next **JNDI lookup() is called** and **passed the name of the connection pool**, and **return Datasource object**
- **getconnection method of the Datasource is called** and returns the **logical connection to the database**

Listing 6-8 Connecting to a database using a pool connection.

```
Context ctext = new InitialContext();
DataSource pool = (DataSource)
ctext.lookup('java:comp/env/jdbc/pool');
Connection db = pool.getConnection();
// Place code to interact with the database here
db.close();
```

Transaction processing

Transaction processing in Java involves ensuring that operations on data within a system are executed in a reliable, consistent, and atomic manner.

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the auto-commit mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named conn, code the following to turn off auto-commit –

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

Commit & Rollback

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( );
```

The following example illustrates the use of a commit and rollback object –

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints –

- **setSavepoint(String savepointName)** – Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName)** – Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

The following example illustrates the use of a Savepoint object –

```java
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   //set a Savepoint
   Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
   String SQL = "INSERT INTO Employees " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees " +
                "VALUES (107, 22, 'Sita', 'Tez')";
   stmt.executeUpdate(SQL);
   // If there is no error, commit the changes.
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

Metadata

. A J2EE component can access metadata by using the DatabaseMetaData interface. The DatabaseMetaData interface is used to retrieve information about databases, tables, columns, and indexes among other information about the DBMS.

A J2EE component retrieves metadata about the database by calling the getMetaData() method of the Connection object. The getMetaData() method returns a DatabaseMetaData object that contains information about the database and its components.

Once the DatabaseMetaData object is obtained, an assortment of methods contained in the DatabaseMetaData object are called to retrieve specific metadata. Here are some of the more commonly used DatabaseMetaData object methods:

- **getDatabaseProductName()** Returns the product name of the database.
- **getUserName()** Returns the username.
- **getURL()** Returns the URL of the database
- **getSchemas()** Returns all the schema names available in this database.
- **getPrimaryKeys()** Returns primary keys.
- **getProcedures()** Returns stored procedure names.
- **getTables()** Returns names of tables in the database.

ResultSet Metadata

There are two types of metadata that can be retrieved from the DBMS. These are metadata that describes the database as mentioned in the previous section and metadata that describes the ResultSet. Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of the ResultSet object. This returns a ResultSetMetaData object, as is illustrated in the following code statement:

```
ResultSetMetaData rm = Result.getMetaData()
```

Once the ResultSet metadata is retrieved, the J2EE component can call methods of the ResultSetMetaData object to retrieve specific kinds of metadata. The more commonly called methods are as follows:

- **getColumnCount()** Returns the number of columns contained in the ResultSet.
- **getColumnName(int number)** Returns the name of the column specified by the column number.
- **getColumnType(int number)** Returns the data type of the column specified by the column number.

Quick Recap……

1. Describe the following concepts
i) Metatdata 2) Transaction Processing 3) Statement objects
2. Explain connection pooling with neat diagram and appropriate code snippets.
3. Explain the JDBC connectivity Process
4. Define drivers. Explain different types of JDBC drivers
5. What is resultset object. List the different metods to work with the resultset. Illustrate different types of operations performed using scrollable resultset objects with example.

End of Module 5

Dear Students ,
All the Best. Do well in the exams