



Unit 10

JDBC:

Overview of JDBC API: Overview of JDBC API, Types of driver JDBC interface and classes for connecting and retrieving data JDBC interface and classes for connecting and retrieving data, Implement JDBC Type of Statement: Type of Statement, Creating and closing a PreparedStatement, Creating and closing a CallableStatement, Commonly used Methods of ResultSet: Batch Processing: What is Batch Processing, Transaction Management, Transaction Rollback RowSet and RowSet Types :RowSet and RowSet, Implement CallableStatement.



What is Database?

- Database is a collection of information ordered in such a way that computer program can quickly select desired pieces of data.
- In other word, **Electronic filing system**
- Conventional databases are organized by fields, records and files.

Databases

Query to create DATABASE:
create database Details;

DATABASE Name: Details
(Collection of tables)



TABLE Name: Person

RollNo	Name	Age
1	Ojaswita	11
2	Shaanvi	4
3	Saanchi	2

**Row
(Tuple)**



**COLUMN
(FIELDS)**



Query to create TABLE:
**create table Person(RollNo int,
Name varchar(20),
Age int);**

Query to Fetch/Get Data from TABLE:

SELECT * from Person; //all columns data

select Name, Age from Person where RollNo=2;

Query to insert data into TABLE:

INSERT INTO Person (RollNo, Name, Age) VALUES(4,“Norah”,3);

INSERT INTO Person VALUES (4,“Norah”,3);

Creating databases and tables

CREATING DATABASE AND TABLES USING MYSQL

Database Tables

- A database can contain one or more tables
- Each table has unique name
- Tables consists of rows and columns. **Rows** are called as **tuples** and **columns** are called as **fields**

Queries

- A query is used to extract data from the database in a readable format according to the user's request.
- A query is an inquiry into database using the SELECT statement.

Creating databases and tables

Creating The MySQL Database

- Since all tables are stored in a database, so first we have to create a database before creating tables.
- The **CREATE DATABASE** statement is used to create a database in MySQL.

Syntax: **CREATE DATABASE** database_name

Adding Tables To mysql Database / Creating Tables

- Since our database is created now it's time to add some tables to it. The CREATE TABLE statement is used to create a table in MySQL database.

- **Syntax:**

```
CREATE TABLE table_name
```

```
{
```

```
column_name1 data_type,
```

```
column_name2 data_type,
```

```
column_name3 data_type,
```

```
...
```

```
}
```

MySQL Data Types

- MySQL supports a number of different data types, the most important ones are summarized below.

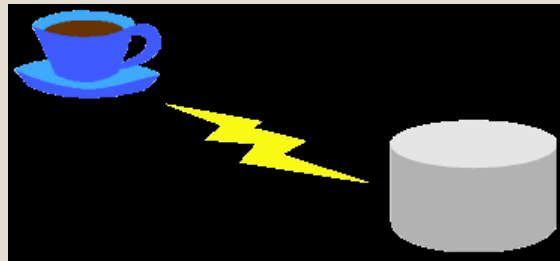
Field Type	Description
INT	A numeric type that can accept values in the range of 2147483648 to 2147483647
DECIMAL	A numeric type with support for floating-point or decimal numbers
CHAR	A string type with maximum size of 255 characters and a variable length
VARCHAR	A string type with maximum size of 255 characters and a variable length
TEXT	A string type with a maximum size of 65535 character
DATE	A date field in the YYYY-MM-DD format
TIME	A time field in the HH:MM:SS format
DATETIME	A combined date/time type in the YYYY-MM-DD HH:MM:SS

Why use Database?

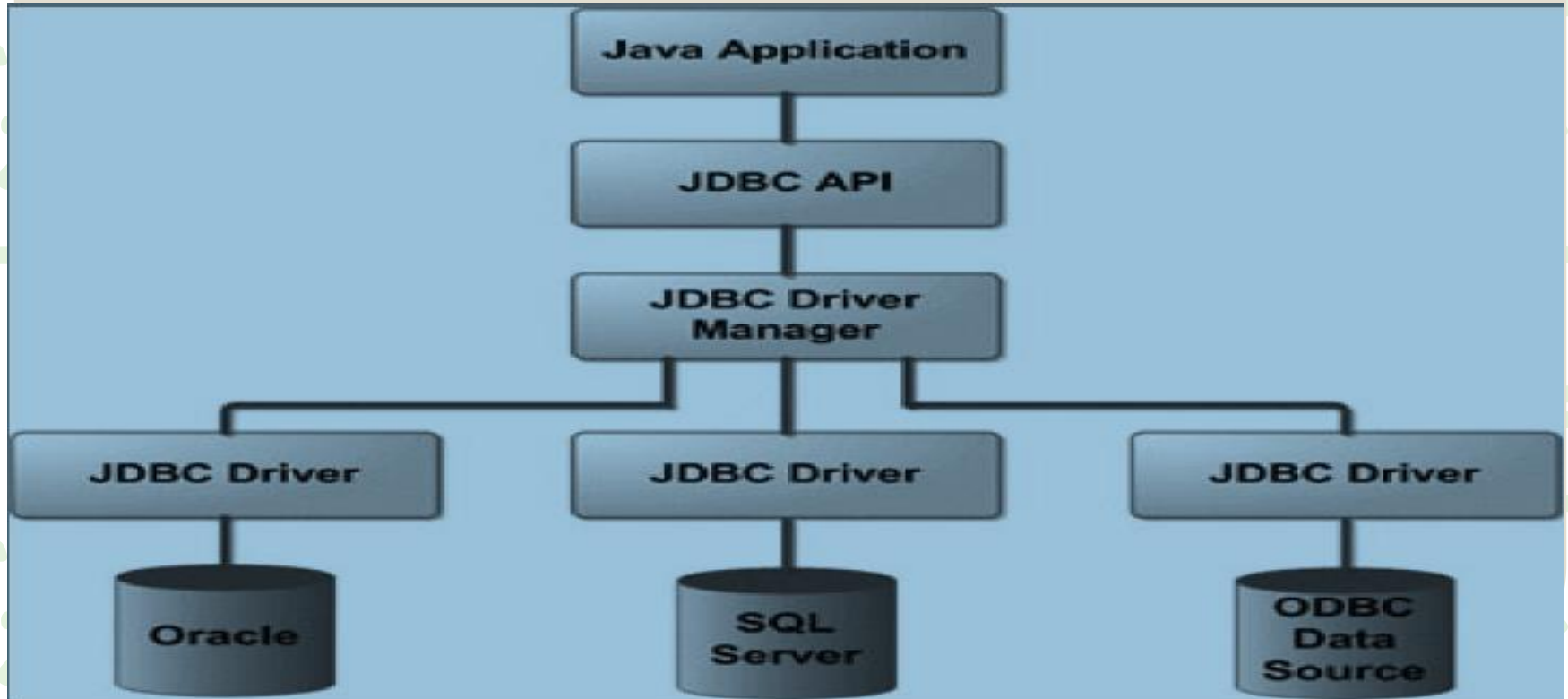
- Retrieving Information: Allows us to regain information more proficiently, more flexibly and more artistically.
- Storing Information: Enables us to store larger amount of information than a normal file.
- Manipulating Information: A database can control the information it contains.
- Printing Information: Enables us to print our information on labels, fax cover sheet, reports and letters.

What is JDBC?

- Java Database Connectivity is a Java API for executing SQL statements and supports basic SQL functionality.
- **JDBC is API that defines interface and classes for writing database applications in java by making database connections.**
- It is a program designed to access many popular DB products on a number of OS platforms.
- We can send SQL,PL/SQL statements to almost any relational DB.



JDBC Architecture



JDBC

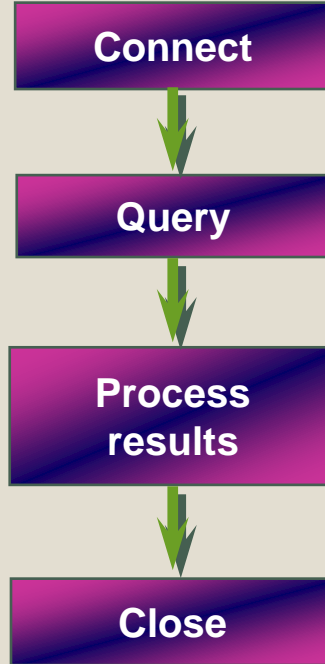
- **JDBC API**
 - Provide facility for accessing the relational database from Java programming language.
 - JDBC API consist of two packages
 - `java.sql`
 - `javax.sql`(Java Enterprise Edition: client-side and server-side)
- **JDBC DriverManager**
 - Manages a list of database drivers
 - Matches connection from java application with the proper database driver using communicating sub protocol.

JDBC

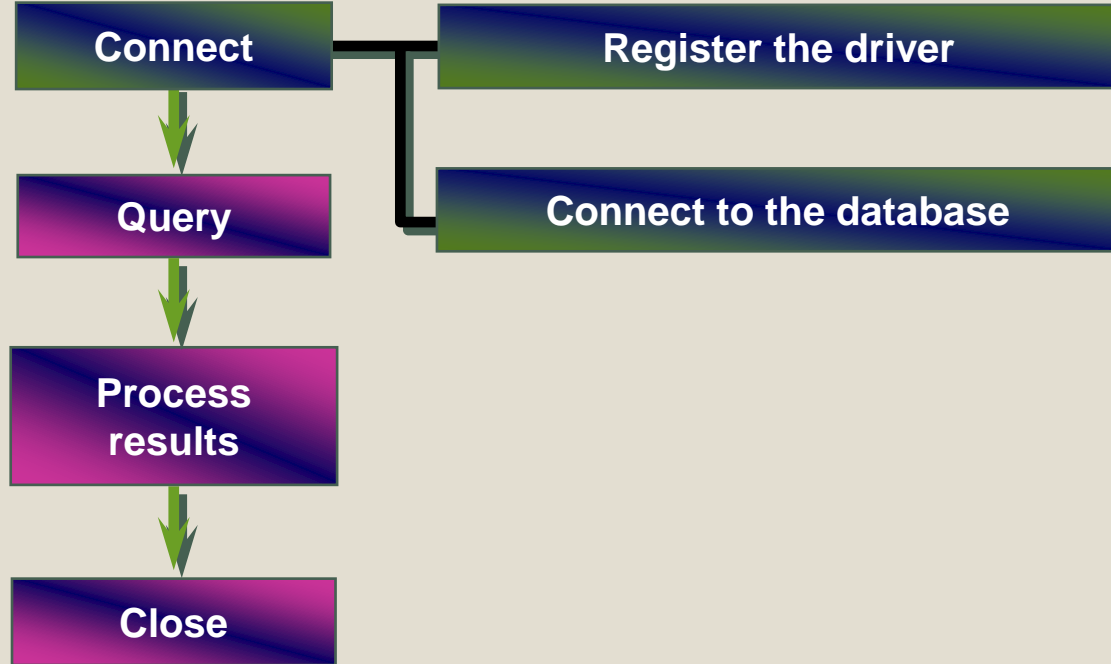
- The main responsibility for JDBC driver manager is to load all the drivers found in the system property
- **JDBC Driver**
 - Is an interpreter that translates **JDBC method calls** to **vendor-specific database commands**



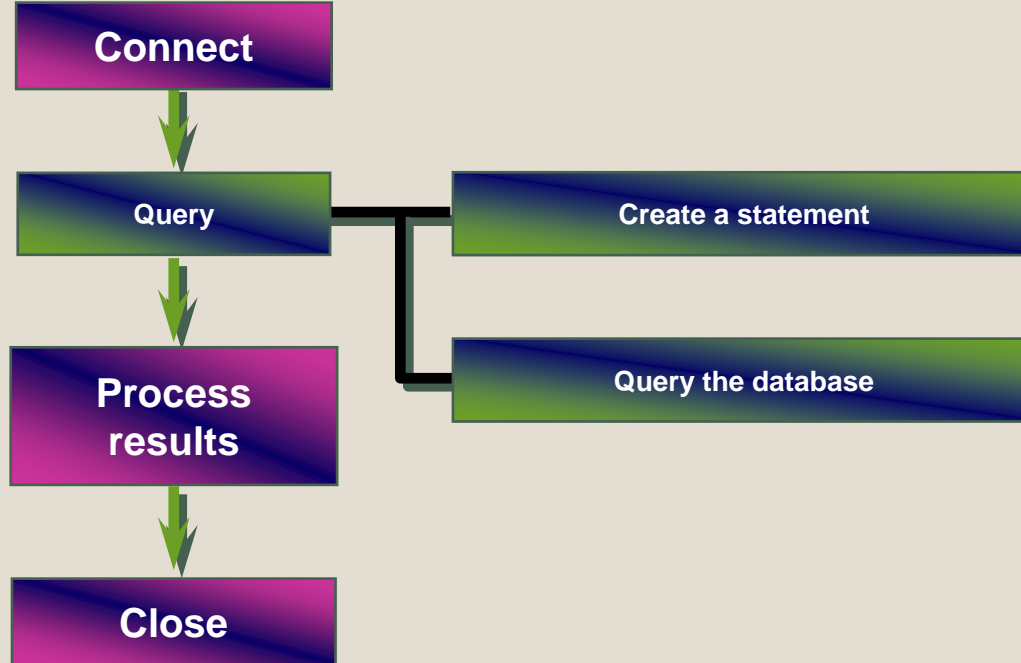
Overview of Querying a Database With JDBC



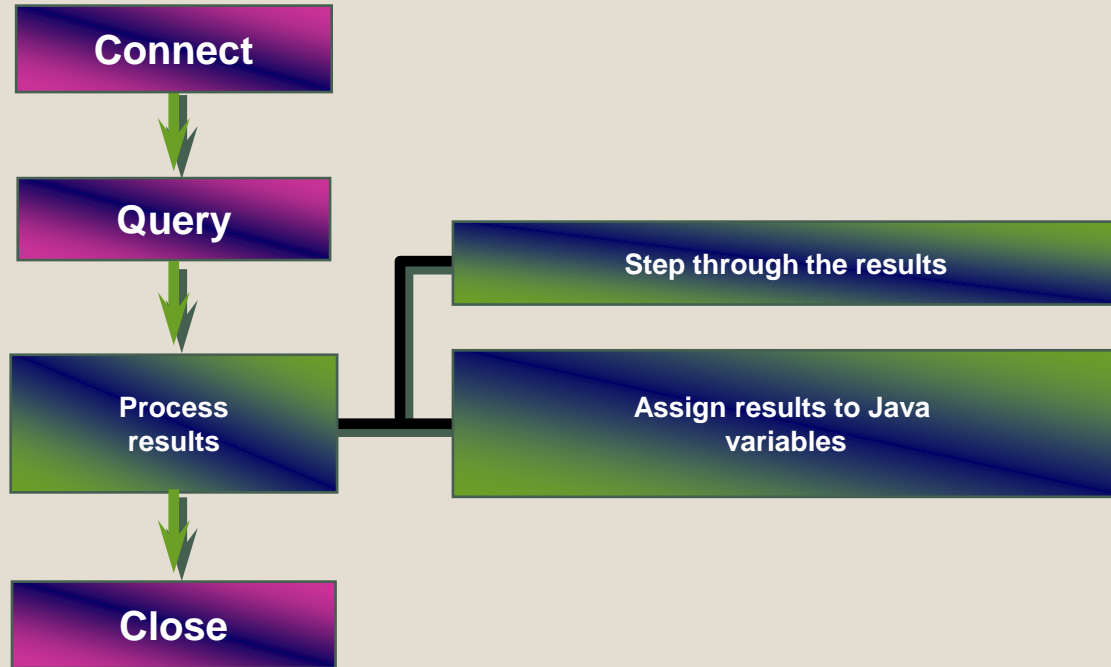
Stage 1: Connect



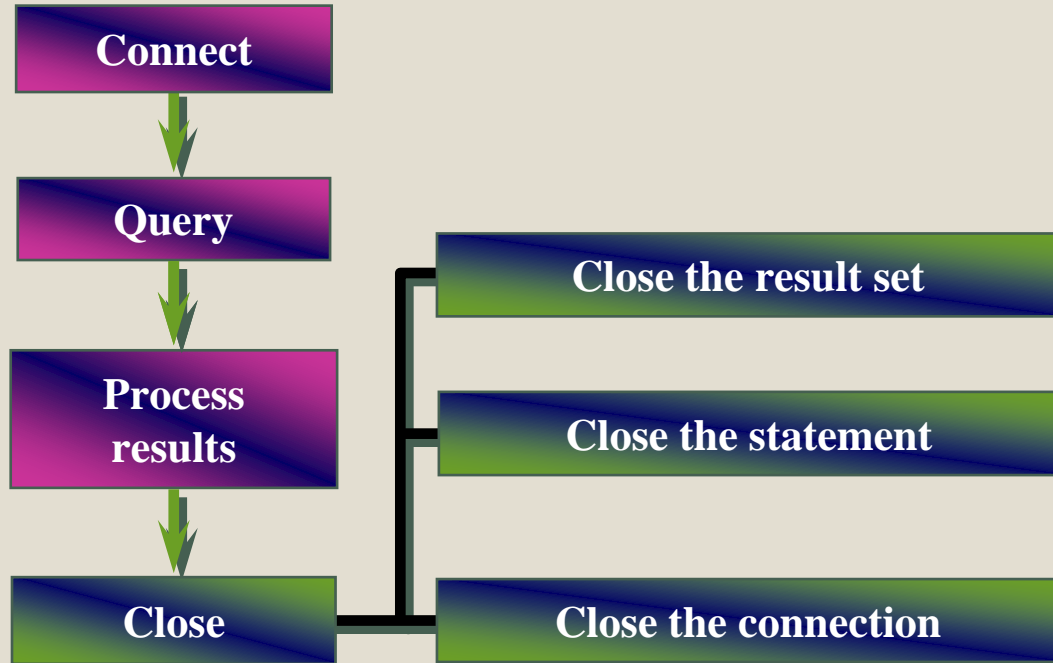
Stage 2: Query



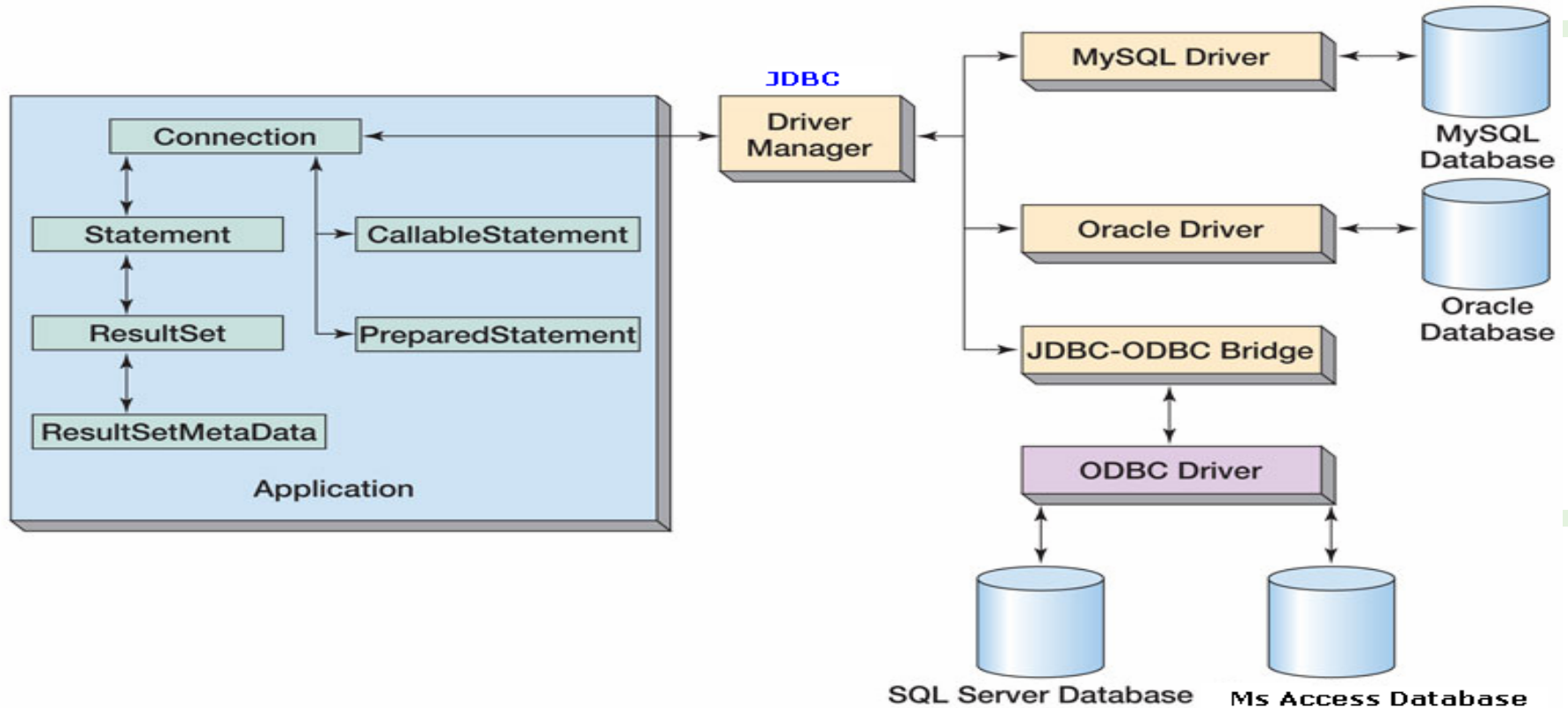
Stage 3: Process the Results



Stage 4: Close



JDBC Components



Java DB Connectivity Steps

- Step 1:
 - To start with java JDBC connection to the database, we must first **import java.sql package.**
- Step 2:
 - **Loading a database driver:**
 - The driver class is loaded by calling `Class.forName()` with the driver name as an argument.
 - The return type of the *`Class.forName(String classname)`* method is “Class” & it is present in **java.lang package.**
 - For ex.
 - *`Class.forName(“com.mysql.jdbc.Driver”);`*
 - *`Class.forName('sun.jdbc.odbc.JdbcOdbcDriver');`*
 - *`Class.forName('oracle.jdbc.driver.OracleDriver“);`*

Java DB Connectivity Steps(Contd.)

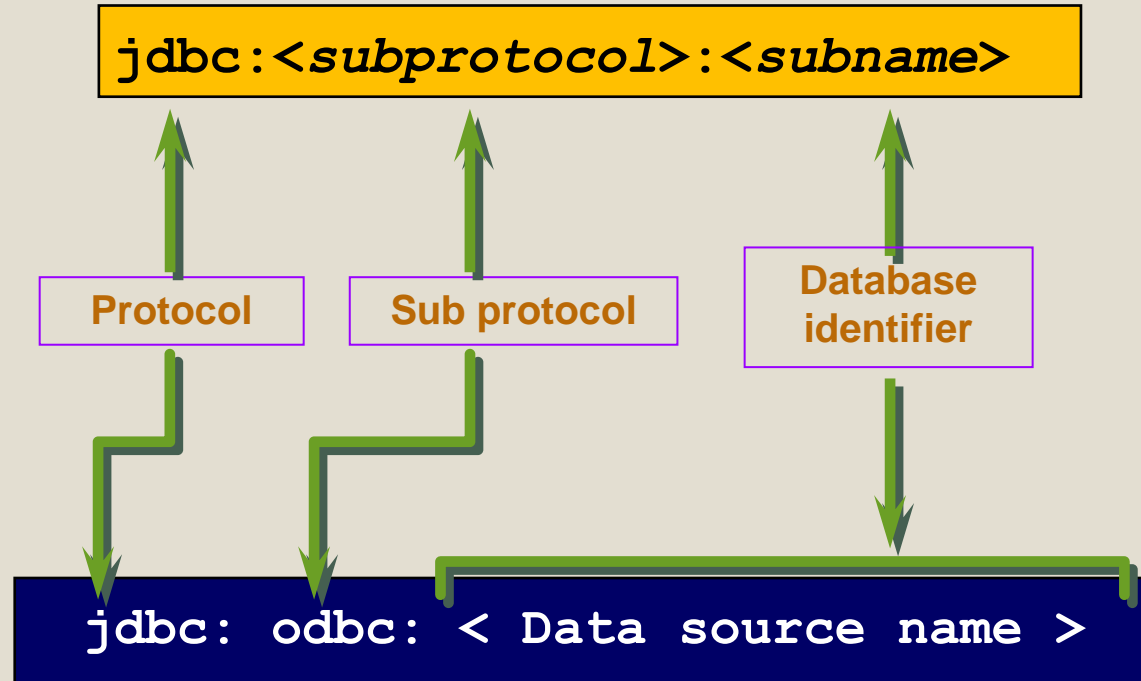
- Step 3:
 - **Creating Jdbc Connection**
 - The JDBC **DriverManager** class defines objects which can connect Java application to a JDBC driver. It manages the JDBC drivers that are installed on the system.
 - It uses ***getConnection()*** method to establish a connection to a database and it returns connection object.
 - An application can have one or more connections with a single database or it can have many connections with different database.
 - **Jdbc URL**
 - JDBC uses a URL to identify the database connection.
 - Each driver has its own sub protocol
 - each sub protocol has its own syntax for the source

JDBC URLs

- For example.

- For MS Access/SQL server Connection

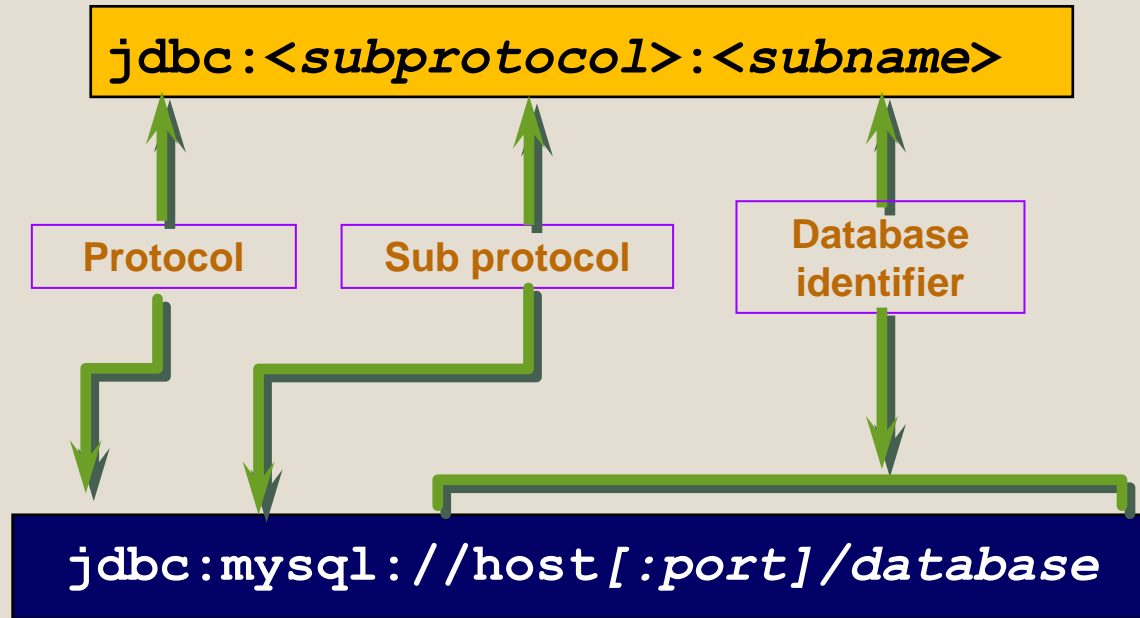
Connection con = DriverManager.getConnection("jdbc:odbc:dsn_name");



JDBC URLs

- For MySQL Connection

Connection con= DriverManager.getConnection("URL of DB","loginname","passwd");

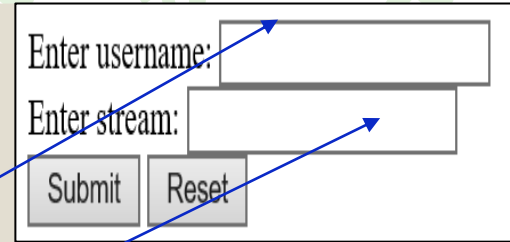


DatabaseHandling.java

```
import java.util.*;
import java.sql.*;

public class DatabaseHandling
{ public static void main(String args[])
{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter student name : ");
    String name=sc.next();
    System.out.println("Enter stream: ");
    String stream=sc.next();
```

```
try
{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/student",
                                                "root", "ganpat");
```



Enter username:

Enter stream:

Java DB Connectivity Steps(Contd.)

- **Step 4:**
 - **Creating JDBC Statement Object:**
 - Once connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database through the established connection.
 - To execute SQL statements, we have 3 kinds of statements:
 - **Standard Statement:**
 - It executes simple *SQL queries without parameters*. This statements creates an SQL Statement object.
Syntax: Statement stmt=con.createStatement();

Java DB Connectivity Steps(Contd.)

b) Prepared Statement:

- It executes a precompiled *SQL queries with or without parameters*. It returns PreparedStatement object.

Syntax:

```
PreparedStatement pstmt= con.prepareStatement("insert into detail values(?,?,?)");  
pstmt.setInt(1, x);  
pstmt.setString(2, y);  
pstmt.setInt(3, x);
```

c) Callable Statement:

- It executes a call to a database stored procedure. It returns new CallableStatement object.

Syntax:

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


DatabaseHandling.java

```
Scanner sc = new Scanner(System.in);  
System.out.println("Enter student name : ");  
String name=sc.next();  
System.out.println("Enter stream: ");  
String stream=sc.next();
```

```
try
```

```
{
```

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

```
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","ganpat");
```

```
    PreparedStatement pstmt= con.prepareStatement("insert into sdetail values(?, ?);"  
    pstmt.setString(1, name);  
    pstmt.setString(2, stream);
```



OR

```
    PreparedStatement pst = con.prepareStatement("select * from sdetail where name=?");  
    pst.setString(1,name);
```



Java DB Connectivity Steps(Contd.)

- **Step 5:**

- **Executing SQL Statement:**

The Statement class has three methods for executing statements

a) *executeQuery()*- for select statement.

b) *executeUpdate()*- for creating or modifying tables.

c) *execute()*- executes SQL statement that is written as String object.

- **Step 6:**

- **Take data**

- *executeUpdate()* always return **integer value** indicating how many row affected.

- *executeQuery()*- always returns **ResultSet object** which hold table data.

- *execute()*- always return **Boolean value** .

Java DB Connectivity Steps(Contd.)

- **ResultSet** provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. The *next()* method is used to successively step through the rows of the tabular results.

- For ex.

```
ResultSet rs= stmt.executeQuery("select * from tablename");  
while(rs.next()) {}
```

Step 7: Close The Connection:

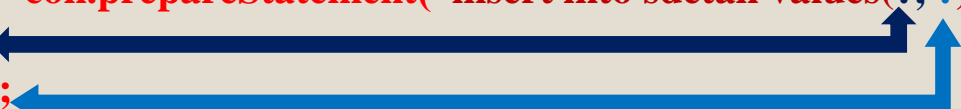
- Database is valuable resource and also consumes lot of memory for creation and execution, therefore it is necessary to close the connection.

Syntax: *rs.close();
stmt.close();
con.close();*

DatabaseHandling.java


```
String name=sc.next();  
String stream=sc.next();  
try  
{  
    Class.forName("com.mysql.jdbc.Driver");  
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","pat");
```

```
    PreparedStatement pstmt= con.prepareStatement("insert into sdetail values(?, ?)");  
    pstmt.setString(1, name);  
    pstmt.setString(2, stream);  
    int row=pstmt.executeUpdate();
```



OR

```
    PreparedStatement pst = con.prepareStatement("select * from sdetail where name=?");  
    pst.setString(1, name);  
    ResultSet rs=pst.executeQuery();  
    while(rs.next()) { out.println("Hi "+rs.getString(1)); }
```



Example:

Table Name: sdetail

name	stream
SHAANVI	Arts
SAANCHI	IT
OJASWITAA	CS

```
PreparedStatement pst = con.prepareStatement("select * from sdetail");
```

```
ResultSet rs=pst.executeQuery();
```

```
while(rs.next())
```

```
{
```

```
    out.println("Hi "+rs.getString(1)+"<br>");
```

```
}
```

OUTPUT:

Hi SHAANVI

Hi SAANCHI

Hi OJASWITAA

Explanation: getString(1) -> name;

getString(2) -> stream

EID	ENAME	ESALARY	EAGE
1	SAANCHI	100000	12
2	OJASWITAA	5000	12

```
PreparedStatement pst = con.prepareStatement("select * from sdetail where name=?");  
ResultSet rs=pst.executeQuery();
```

OR

```
int row=pst.executeUpdate();
```

OR

```
Boolean ans=pst.execute();
```

DatabaseHandling.java

```
String name=sc.next();  
String stream=sc.next();
```

```
try  
{  
    Class.forName("com.mysql.jdbc.Driver");  
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","pat");  
    PreparedStatement pstmt= con.prepareStatement("insert into sdetail values(?, ?)");  
    pstmt.setString(1, name);  
    pstmt.setString(2, stream);  
    int row=pstmt.executeUpdate();  
    if(row==1)  
    {  
        out.println("Data updated Successfully");  
    }  
}  
catch(Exception e)  
{  
    System.out.println(e);  
}
```



```

@WebServlet(urlPatterns = {"/Servlet"})
public class Servlet extends HttpServlet {
    protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","patkar");
            Statement stmt=con.createStatement();
            String query="insert into register values('admin','patkar')";
            Boolean ret = stmt.execute(query);
            out.println("<br><b>*****For Insert Query*****<br>Return value is :</b> " + ret.toString());

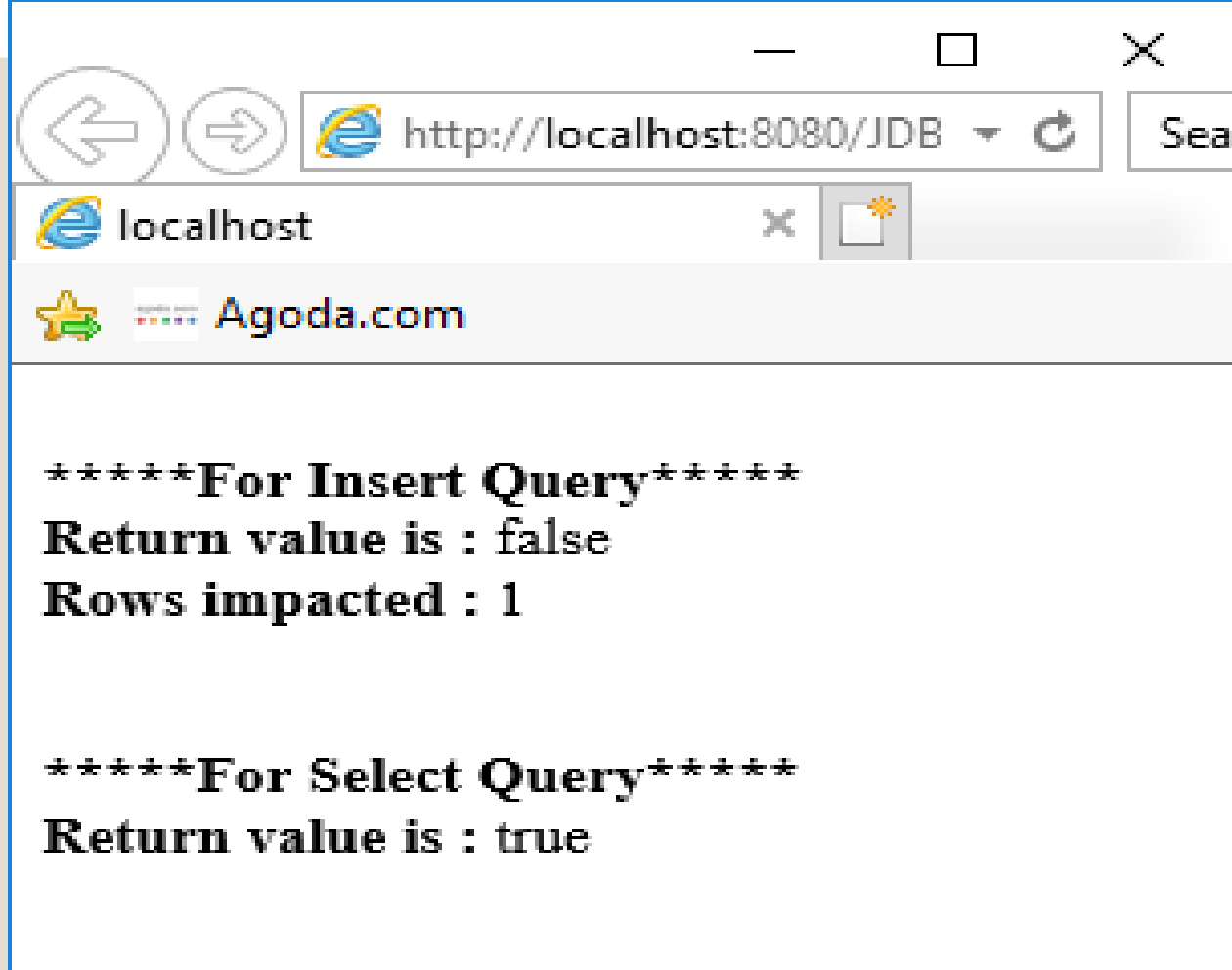
            PreparedStatement pstmt=con.prepareStatement(query);
            int rows = pstmt.executeUpdate(query);
            out.println("<br><b>Rows impacted :</b> " + rows);

            String query2="select * from register";
            Boolean ret2 = stmt.execute(query2);
            out.println("<br><br><br><b>*****For Select Query*****<br>Return value is :</b> " + ret2.toString());

        }

        catch (Exception e)
        {
            out.println(e);
        }
    }
}

```

```
// Let us select all the records and display them.
```

```
sql = "SELECT id, first, last, age FROM Employees";  
ResultSet rs = stmt.executeQuery(sql);
```

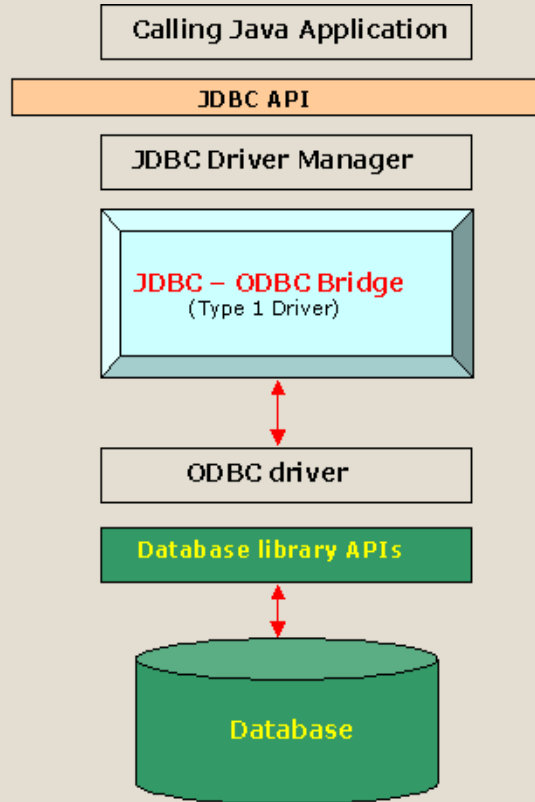
```
//STEP 5: Extract data from result set
```

```
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);  
}
```

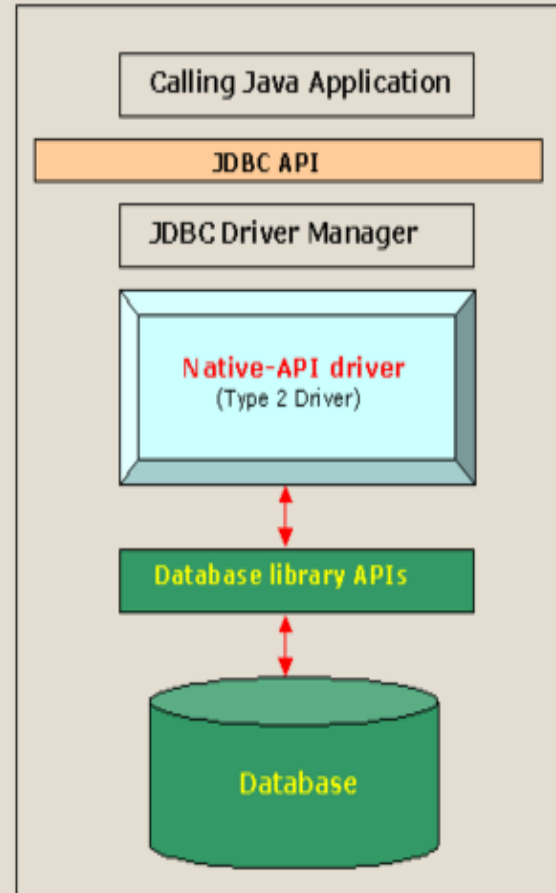
JDBC Driver Types

- Type 1
 - **JDBC-ODBC Bridge**
- Type 2
 - **Native API Driver**
- Type 3
 - **Network Protocol Driver or Middleware Driver**
- Type 4
 - **Database protocol Driver (100% Java)**

Type 1: JDBC-ODBC Bridge

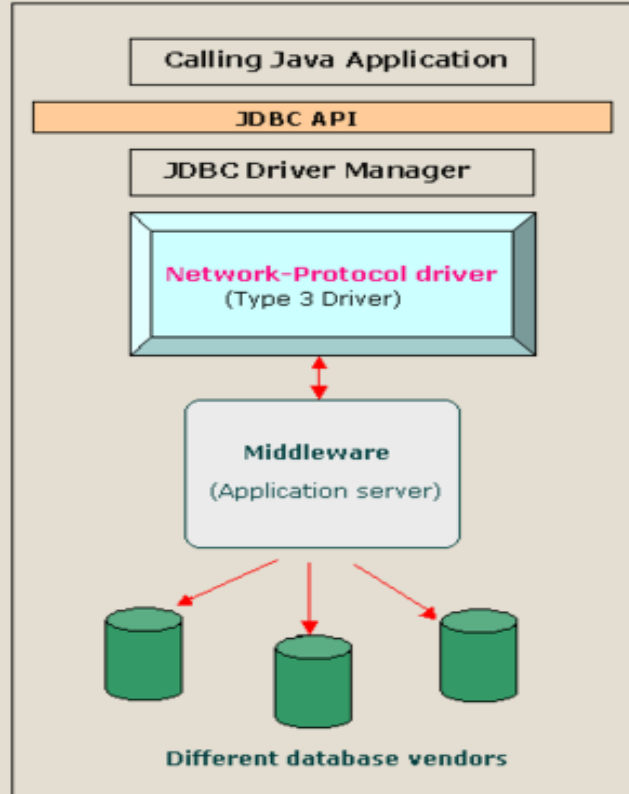


Type 2: Native API Driver

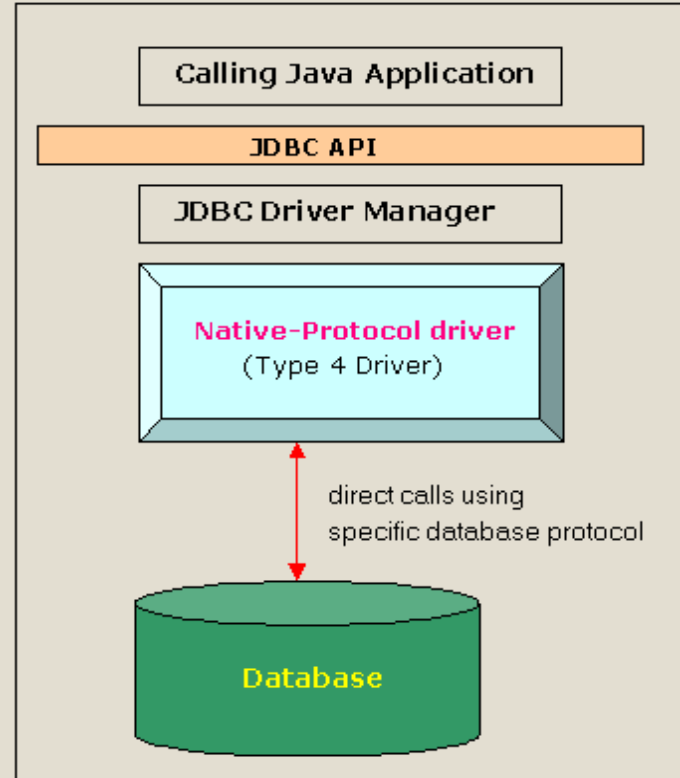


Types of Driver

Type 3: Network Protocol Driver



Type 4: Database Protocol Driver



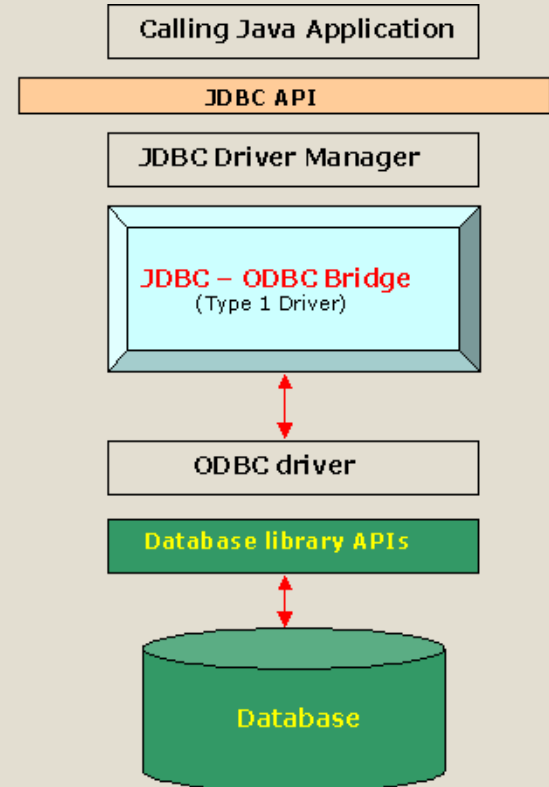
Types of Driver

What is ODBC ? (Reference)

- **ODBC (Open Database Connectivity)**
 - A set of APIs for database access
 - Originally only for windows platforms, later extended to non-windows platforms
 - ODBC must be set up on every client
- Sun provide a JDBC-ODBC bridge driver, `sun.jdbc.odbc.JdbcOdbcDriver`.
- This driver is native (binary/machine level) code not java, and is closed source

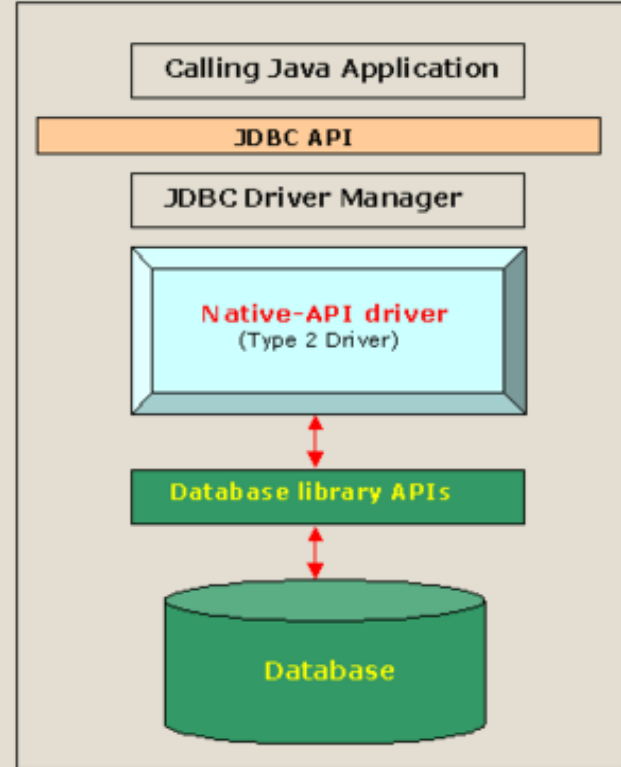
JDBC-ODBC Bridge

- Acts as “bridge” between JDBC and another database connectivity mechanism such as ODBC
- Provide access to most standard ODBC drivers.
- Translates all JDBC calls into ODBC calls and sends them to the ODBC driver.



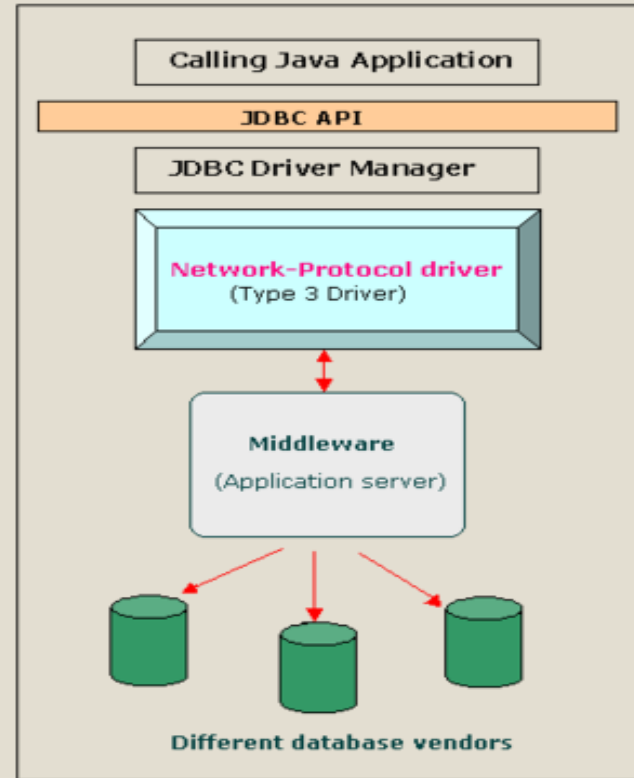
Native API Driver

- This driver converts JDBC API calls into DBMS precise client API calls.
- Converts the JDBC call into database specific call for database such as ORACLE
- It is specific to particular database
- Requires that some binary code be loaded on each client computer.



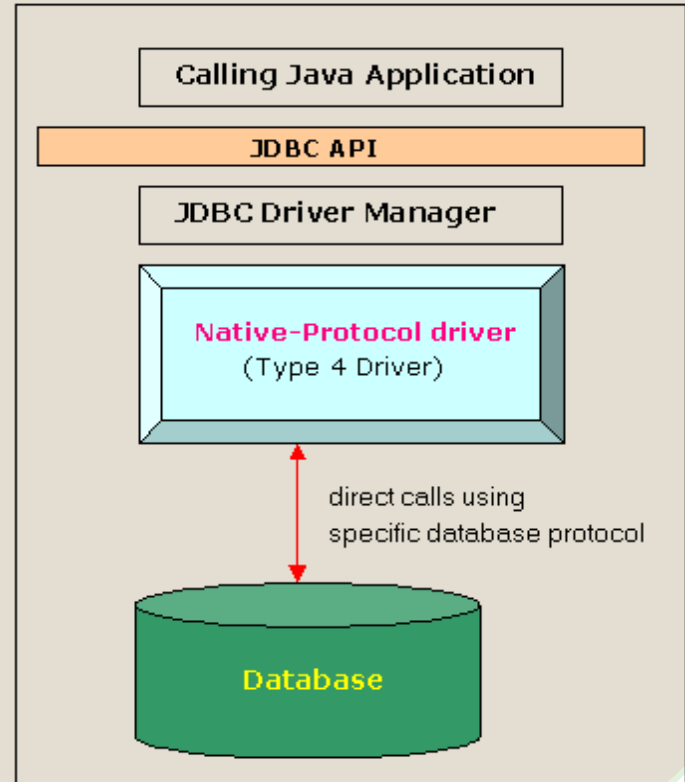
Network Protocol Driver

- Pure Java Driver for Database Middleware
- Uses of a middle tier between the calling program and the database.
- The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.
- Follows a three tier communication approach
- Code not installed on the client
- A single driver can actually provide access to multiple databases

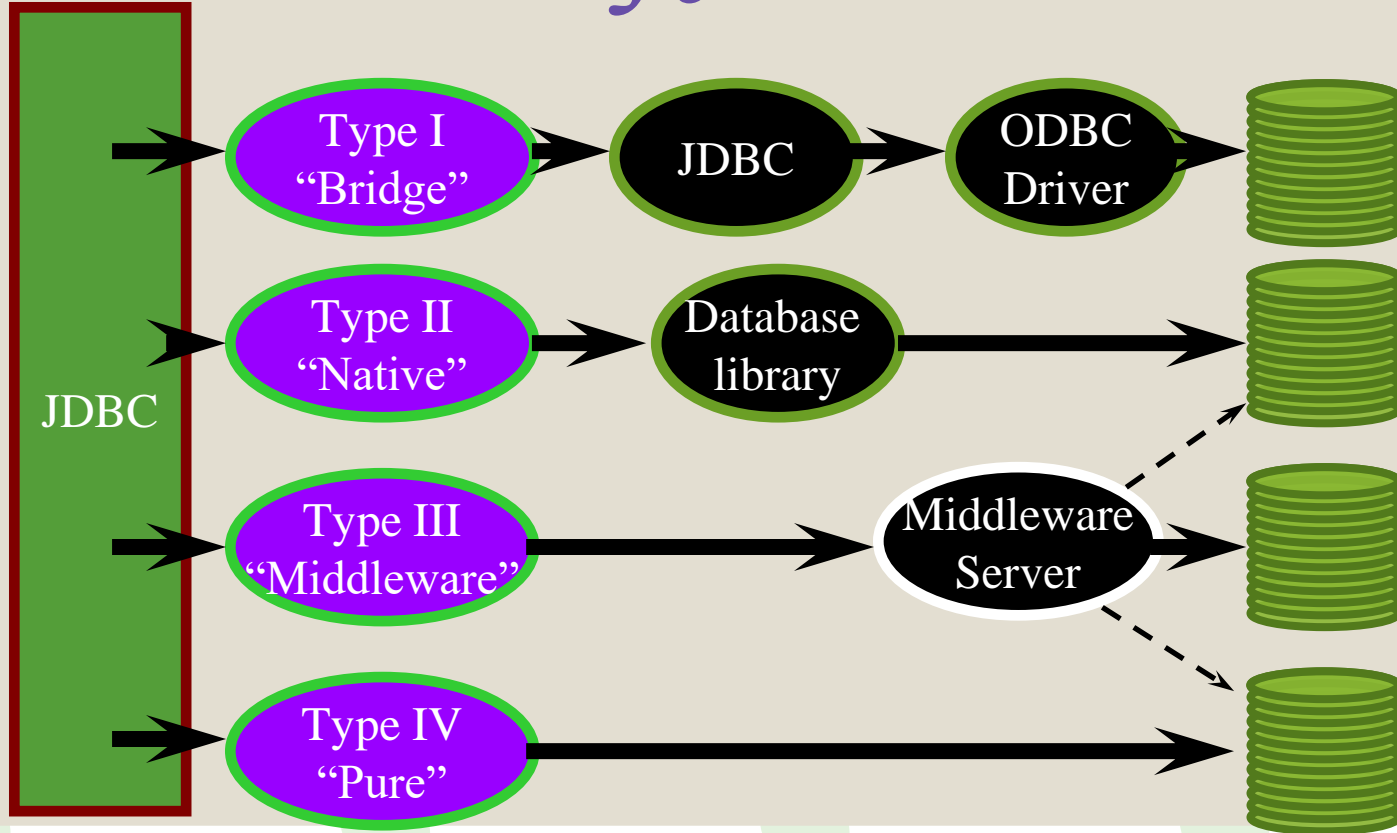


Database Protocol Driver

- Direct to Database **Pure Java Driver**
- Converts JDBC calls directly into a vendor-specific database protocol.
- They install inside the Java Virtual Machine of the client.
- This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls.
- Unlike the type 3 drivers, it does not need associated software to work.
- MySQL's Connector/J driver is a Type 4 driver.



Overview of JDBC Drivers



Batch Processing

- Instead of executing a single query, **we can execute a batch (group) of queries**. It makes the performance fast.
- The **java.sql.Statement** and **java.sql.PreparedStatement** interfaces provide methods for batch processing.
- Example

Below is an example of batch processing

- Load the driver class
- Create Connection
- Create Statement
- **Add query in the batch**
- **Execute Batch**
- Close Connection

Batch Processing (contd.)

```
Statement stmt=con.createStatement();  
stmt.addBatch("insert into user420 values(190,'abhi',40000)");  
stmt.addBatch("insert into user420 values(191,'umesh',50000)");  
stmt.executeBatch();//executing the batch  
con.commit();  
con.close();
```

Transaction Management

- A transaction is a **set of one or more statements that is executed as a unit**, so either all of the statements are executed, or none of the statements is executed.
- **Disabling Auto-Commit Mode**
When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.
- The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. Consider con is an active connection:
con.setAutoCommit(false);
- **Committing Transactions**
After the auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly. All statements executed after the previous call to the method commit are included in the current transaction and committed together as a unit.
con.Commit();

```
String InsertDataQuery = "INSERT INTO EmployeeTb VALUES(?,?,?)";
```

```
con.setAutoCommit(false);
```

```
PreparedStatement InsertData = con.prepareStatement(InsertDataQuery);
```

```
System.out.println("Enter number of employees data to be inserted:");
```

```
int n=sc.nextInt();
```

```
String ename[]=new String[n];
```

```
int age[]=new int[n+1];
```

```
int salary[]=new int[n+1];
```

```
for(int i=0;i<n;i++)
```

```
{
```

```
System.out.println("Enter employee name : ");
```

```
ename[i] = sc.next();
```

```
System.out.println("Enter employee age : ");
```

```
age[i] = sc.nextInt();
```

```
System.out.println("Enter employee salary : ");
```

```
salary[i] = sc.nextInt();
```

```
InsertData.setString(1,ename[i]);
```

```
InsertData.setInt(2,age[i]);
```

```
InsertData.setInt(3,salary[i]);
```

```
InsertData.addBatch();
```

```
}
```

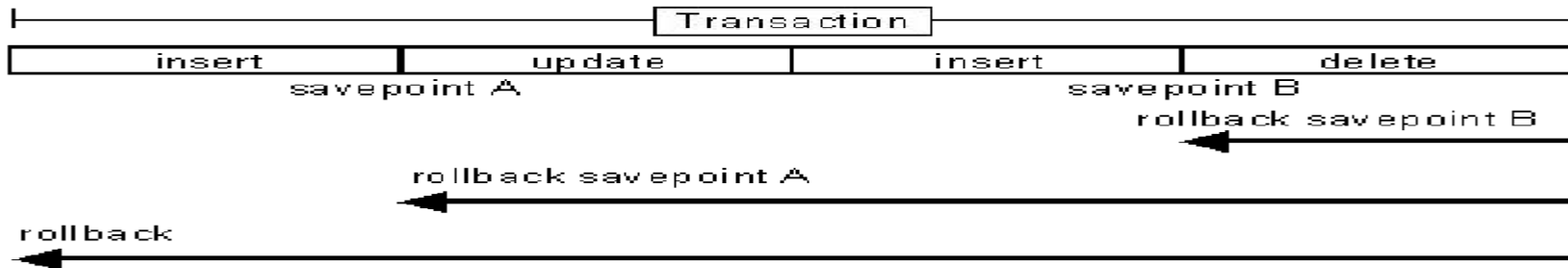
```
InsertData.executeBatch();
```

```
con.commit();
```

Batch Processing & Transaction Management

Transaction Rollback

- The method **Connection.setSavepoint**, sets a Savepoint object within the current transaction. The **Connection.rollback** method is overloaded to take a Savepoint argument.
- //Setting the save point
- **Savepoint savePoint = con.setSavepoint("MyCheckPoint");**
- **con.commit();**
- **con.rollback();**
- Or
- **con.rollback(mySavepoint);**




```
try{
Class.forName("com.mysql.jdbc.Driver"); //driver
String driverUrl = "jdbc:mysql://localhost:3306/demo";
Connection con = DriverManager.getConnection(driverUrl,"root","ganpat");
con.setAutoCommit(false);

Statement st = con.createStatement();
st.executeUpdate("insert into EmployeeTb values('OOP',29,9000)");
st.executeUpdate("update EmployeeTb set name='xavier' where age=29");
Savepoint svpt = con.setSavepoint("mysp");
st.executeUpdate("insert into EmployeeTb values('ICT',12,10080)");
con.rollback(svpt);
con.commit();
}
catch(Exception e){}
```

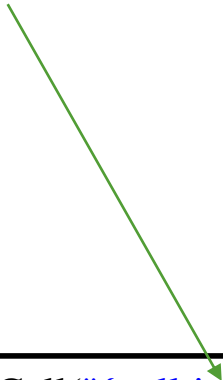
#	name	age	salary
1	Ganpat	13	9000
2	Ganpat	13	9000
3	xavier	29	9000

Implementing Callable Statement

- CallableStatement interface is used to call the **stored procedures and functions**.
- We can have **business logic on the database by the use of stored procedures and functions** that will make the performance better because these are precompiled.

Implementing Callable Statement

```
create or replace procedure "INSERTDATA"  
(id IN NUMBER,  
name IN VARCHAR2)  
is  
begin  
insert into user420 values(id,name);  
end;  
/
```



```
create table user420(id number(10),  
name varchar2(200));
```

```
CallableStatement stmt=con.prepareCall("{call insertData(?,?)}");  
stmt.setInt(1,1011);  
stmt.setString(2,"Amit");  
stmt.execute();  
  
System.out.println("success");
```

RowSet

A JDBC *RowSet* object holds tabular data in a style that makes it more adaptable and simpler to use than a result set.

```
JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();  
rowSet.setUrl("jdbc:mysql://localhost:3306/demo");  
rowSet.setUsername("root");  
rowSet.setPassword("ganpat");  
  
rowSet.setCommand("select * from employeeTb");  
rowSet.execute();
```

JDBC Classes

- **DriverManager**
 - Manages JDBC Drivers
 - Used to Obtain a connection to a Database
- **Types**
 - Defines constants which identify SQL types
- **Date**
 - Used to Map between java.util.Date and the SQL DATE type
- **Time**
 - Used to Map between java.util.Date and the SQL TIME type
- **TimeStamp**
 - Used to Map between java.util.Date and the SQL TIMESTAMP type

JDBC Interfaces

- **Driver**
 - All JDBC Drivers must implement the Driver interface. Used to obtain a connection to a specific database type
- **Connection**
 - Represents a connection to a specific database
 - Used for creating statements
 - Used for managing database transactions
 - Used for accessing stored procedures
 - Used for creating callable statements
- **Statement**
 - Used for executing SQL statements against the database

JDBC Interfaces

- **ResultSet**
 - Represents the result of an SQL statement
 - Provides methods for navigating through the resulting data
- **PreparedStatement**
 - An SQL statement (which can contain parameters) is compiled and stored in the database
- **CallableStatement**
 - Used for executing stored procedures
- **DatabaseMetaData**
 - Provides access to a database's system catalogue
- **ResultSetMetaData**
 - Provides information about the data contained within a ResultSet