DATABASE MANAGEMENT SYSTEMS (IT 2040)

LECTURE 06 – JAVA DATABASE CONNECTIVITY

LECTURE CONTENT

- JDBC

LEARNING OUTCOMES

- At the end of this lecture students would be able to
  - Write a command line program in java to access different databases

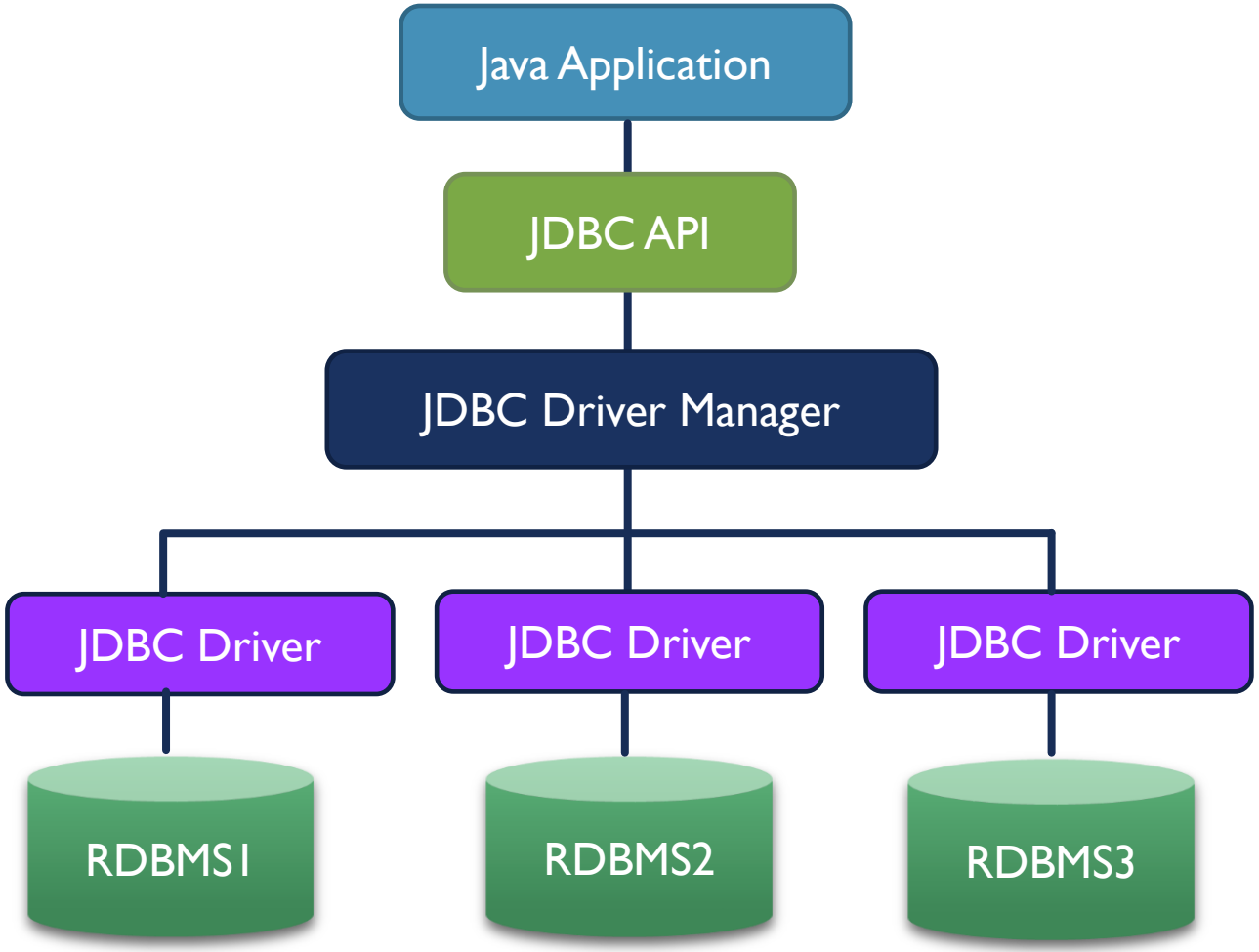
INTRODUCTION

- SQL is the standard common language used for storing in and obtaining information from relational databases.
- Even though SQL was standardized, still different applications was needed to be developed to access different DBMSs.
- A common way for an application to access a relational database was highly desirable.

JDBC

- Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases.
- The application program interface lets you to connect to a database, to interact with that database via SQL and retrieve results from the database that could be used within a java application.

JDBC ARCHITECTURE



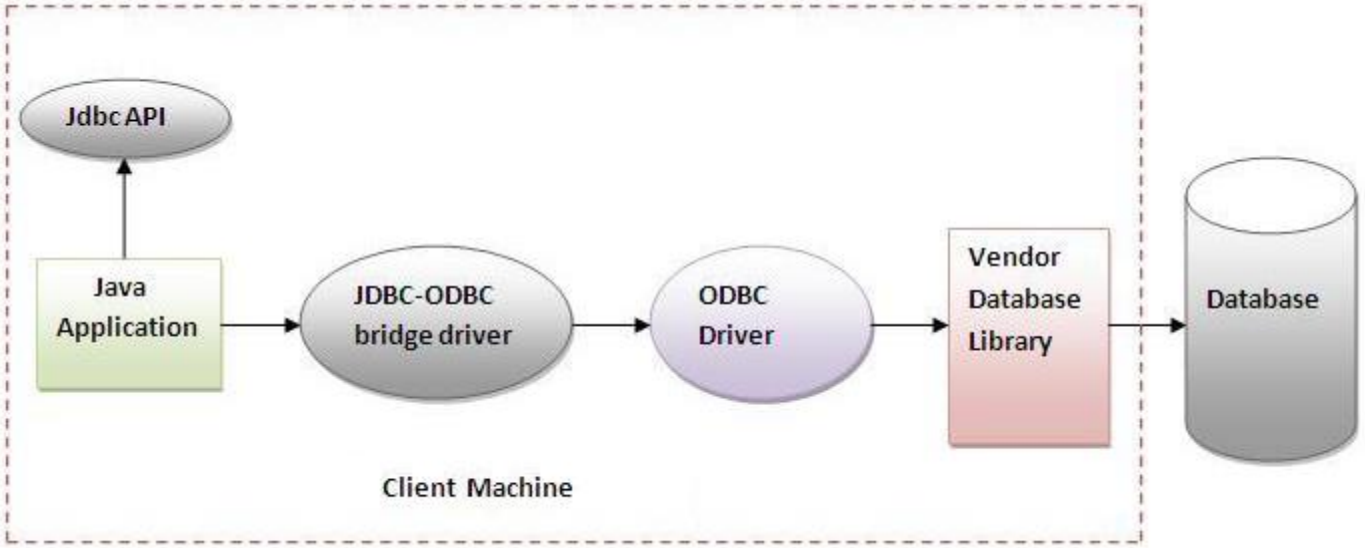
- The JDBC API provides the application-to-JDBC Manager connection.
- The JDBC driver manager supports the JDBC Manager-to-Driver Connection. It manages a list of database drivers. Matches connection requests from the java application with the proper database driver.
- JDBC Driver handles the communications with the database server.

TYPES OF JDBC DRIVERS

- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation in to four categories which are :
  - Type 1: JDBC-ODBC Bridge Driver
  - Type 2: Native API Driver
  - Type 3: Network Pure Java Driver
  - Type 4: Thin Driver(100% Pure Java Driver)

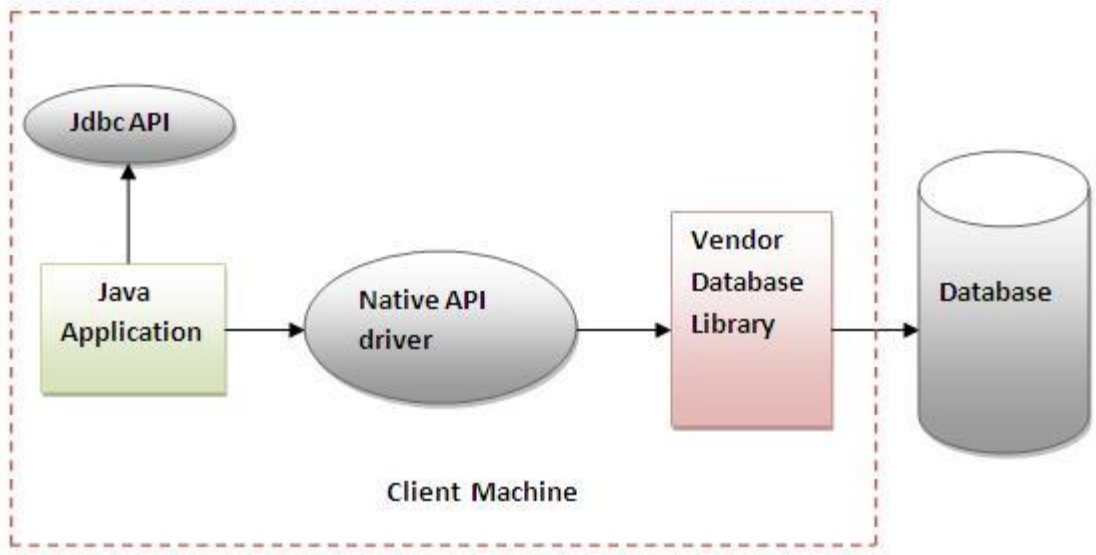
JDBC-ODBC BRIDGE DRIVER

- The JDBC-ODBC bridge driver uses ODBC driver installed in a client machine to connect to the database.
- The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.



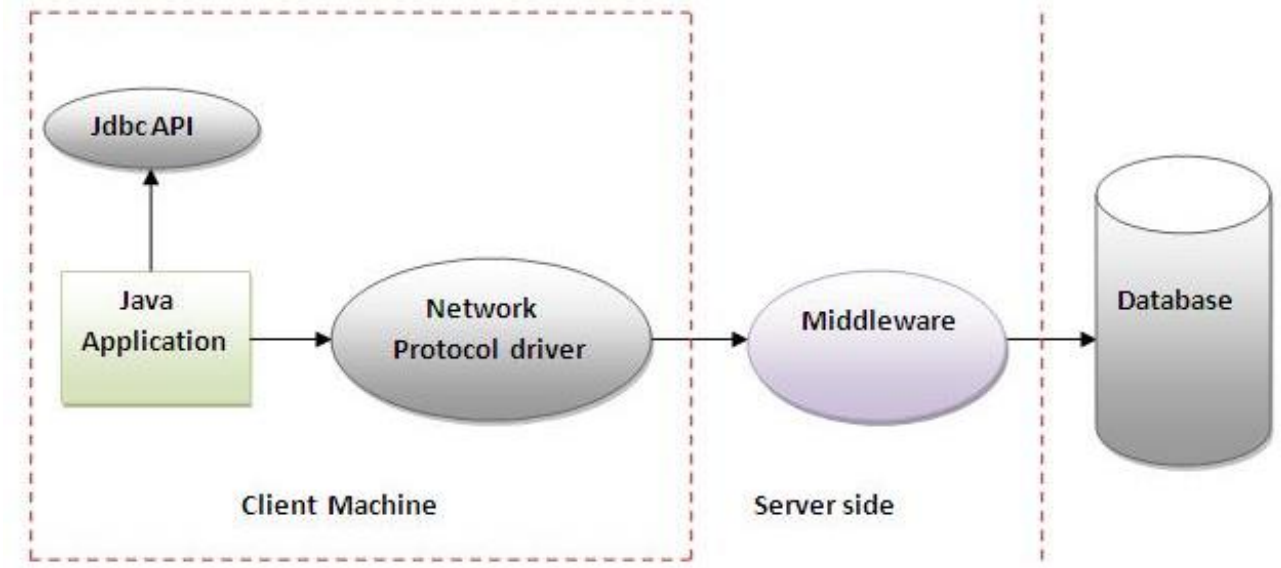
NATIVE API DRIVER

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database a specific database, such as IBM, Informix, Oracle or Sybase.
- These drivers are 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.



NETWORK PURE JAVA DRIVER

- In a Type 3 driver, a three-tier approach is used to access databases.
- The JDBC clients translate JDBC calls in to middleware vendor's protocol which in return is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.



THIN DRIVER(100% PURE JAVA DRIVER)

- Written entirely in Java
- Converts JDBC calls into packets that are sent over the network in the proprietary format
- used by the specific database.
- This is the highest performance driver available for the database and is usually provided by the vendor itself.

CREATING A JDBC APPLICATION

- **Import the packages**: Import the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Load the driverRegister the JDBC driver**: Initialize a driver in order to open a communication channel with the database.
- **Open a connection**: Create a Connection object, which represents a physical connection with the database.
- **Execute a query**: Using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set**: retrieve the data from the result set.
- **Clean up the environment**: Closing all database resources versus relying on the JVM's garbage collection.

STEP 1 : IMPORT THE PACKAGES

- In this step the packages containing the JDBC classes needed for database programming are imported.
- The Java.sql package contains the entire JDBC API that sends SQL statements to relational databases and retrieves the results of executing those SQL statements.
- Some commonly used interfaces in the package are as follows.
  - The Driver interface represents a specific JDBC implementation for a particular database system.
  - Connection represents a connection to a database.
  - The Statement, PreparedStatement, and CallableStatement interfaces support the execution of various kinds of SQL statements.
  - ResultSet is a set of results returned by the database in response to a SQL query.

STEP 2: REGISTERING DATABASE DRIVERS

- In order to use a driver, it should be registered in the program.
- The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it.
  - Ex : Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();

STEP 3 : OPEN A CONNECTION

- After the driver is loaded to the memory a connection to the database could be established using the **DriverManager.getConnection()** method.
- There are three parameters to the method which are URL of the database, user name and the password.
  - Ex:
    - String url = “jdbc:odbc:dbConn”
    - Connection c = DriverManager.getConnection(url,“user",“password");
- This connection should be closed later using Connection.close()

STEP 4 : EXECUTING QUERIES

- 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 commands and receive data from your database.
- A summary of statements is available below :

Interface	Recommended use
Statement	Useful when static SQL statements are used at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Suitable when SQL statements are used many times. The PreparedStatement interface accepts input parameters at runtime
CallableStatement	Used to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

STEP 4 : EXECUTING QUERIES – STATEMENT OBJECT

- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.
  - boolean execute (String SQL): Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
  - int executeUpdate (String SQL): Returns the number 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
    - Ex: int rowsUpd = st.executeUpdate("INSERT INTO Customers (cid, cname) VALUES '123A', 'Sampath')");
  - 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.
    - Ex: ResultSet s = st.executeQuery("SELECT cname FROM Customers");

STEP 4 : EXECUTING QUERIES – PREPARED STATEMENTS

- This statement gives you the flexibility of supplying arguments dynamically
- 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.
- All of the Statement object's methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object.
    - Ex:

```
String sql = "UPDATE Employees set age=? WHERE id=?";
PreparedStatement pstmt = conn.prepareStatement(sql);
stmt.setInt(1, 35); // This would set age
stmt.setInt(2, 102); // This would set ID
int rows = stmt.executeUpdate();
```

STEP 4 : EXECUTING QUERIES – CALLABLE STATEMENTS

- Following steps are used in calling a procedure within a java program
- Firstly, we must prepare a CallableStatement object
  - CallableStatement cStmt = conn.prepareCall("exec getEmpName ?, ?");
- Bind parameters
  - stmt.setInt(1, 102);
- Register output parameters if any
  - stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
- Execute the stored procedure
  - stmt.execute();
- Get any output parameters if any
  - String empName = stmt.getString(2);

STEP 5 : EXTRACT DATA FROM THE RESULT SET

- The SQL statements that read data from a database query, return the data 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 next() method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.
- There is a get method for each of the possible data types There is a get method for each of the possible data types. Get method could be called by providing the column index or column name
- Ex:
  - rs = stmt.executeQuery("SELECT cname, salary FROM Customer");

```
while (rs.next())
{
    String name = rs.getString(1);
    float salary = rs.getFloat(2);
}
```

SQL EXCEPTIONS

- JDBC Exception handling is very similar to the Java Excpetion handling but for JDBC, the most common exception you'll deal with is java.sql.SQLException.
- When an exception occurs, an object of type SQLException will be passed to the catch clause.
- The passed SQLException object has the following methods available for retrieving additional information about the exception:
  - A string describing the error: getMessage()
  - An integer error code that is specific to each vendor: getErrorCode()

A SAMPLE PROGRAM

```
String url="jdbc:odbc:dsn";
String query="select ProductID,ProductName,UnitPrice from Products";
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
Connection con=DriverManager.getConnection(url);
Statement s=con.createStatement();
ResultSet res=s.executeQuery(query);
System.out.println("Product ID \t Product Name \t\t\t Unit Price");
        while(res.next())
        {
                int pID=res.getInt(1);
                String pName=res.getString(2);
                float price=res.getFloat(3);
                System.out.println(pID+"\t"+pName+"\t\t"+price);
        }
        con.close();
}
```