Chapter 05
Java Data-Base Connectivity

Contents:

Java as Database Front End

Java offers several benefits to the developer creating a front-end application for a database server. Java is 'Write Once Run Everywhere' language. This means that Java programs may be deployed without recompilation on any computer architectures and operating systems that possesses a Java Virtual Machine.

In addition there is a cost associated with deployment and maintenance of the hardware and software of any system (client) the corporation owns Systems such as Windows PC, Macintosh and Unix desktop centric clients (fat clients) can cost corporations between $10,000 to $15,000 per installation seat. Java technology has made it possible for any company to use smaller system footprint. These systems are based on Java chip set and run any and all Java programs from built-in Java operating system.

Java based clients (thin clients) that operate with minimum of hardware resources, yet run the complete Java environment are expected to cost around $70 per seat. According to studies, saving for the corporations moving 10,000 fat clients to thin clients systems could be much as $100 million annually.

There are many industrial-strength DBMS available in the market. These include Oracle DB2, Sybase and many other popular brands. The challenge to Sun Microsystems faced in the late 1990s was to develop a way for Java developer to write a high level code that accesses all popular DBMSs.

The Sun Microsystems met the challenge in 1996 with the creation of JDBC driver for JDBC API. Both were created out of necessity, because until then Java wasn't industrial strength programming language since Java was unable to access the DBMS.

The JDBC driver developed by Sun wasn't driver at all. It was specification that described the detail functionality of JDBC driver. DBMS manufacturers and third-party vendors encouraged to build JDBC drivers that confirmed to Sun's specifications. Those firm that built JDBS drivers for their product could tap into growing Java applications market.

The specifications required a JDBC driver to be a translator that converted low-level proprietary DBMS messages to low-level messages understood by JDBC API and vice-versa. This meant that Java programmer could use high-level Java data-objects defined in the JDBC API to write a routine that interacted with the DBMS. Java data objects convert the routine into low-level message that conform to the JDBC driver specification and send them to the JDBC driver. The JDBC driver translates the routine into low-level messages that understood and processed by DBMS.

Database client-server methodology

Relational databases are the most common DBMS. A main characteristic of a relational database is the absolute separation between physical and logical data. Data is accessed through the associated logical model to avoid supplying

physical storage locations and to reduce the limitations imposed by using physical information.

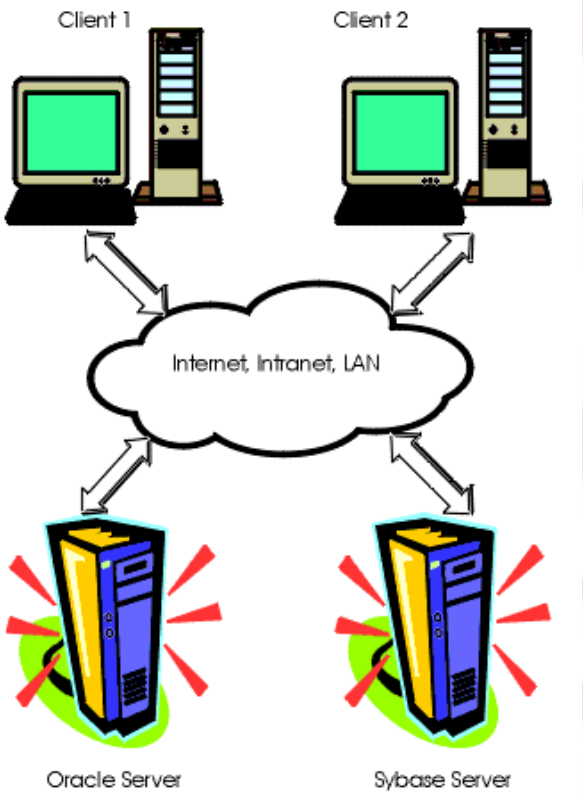


Fig. Database client/server architecture

Relational databases allow the definition of relations and integrity rules between data sets. E.F. Codd developed this model at the IBM San Jose Research Lab in the 1970s. A language to handle, define, and control data was also developed at the IBM lab: SQL. SQL stands for Structured Query Language. SQL is a query language that interacts with a DBMS. It allows data access without supplying physical access plans, data retrieval as sets of records, and the performing of complex computations on the data.

Software Architectures

The first generation of client-server architectures is called two-tiered. It contains two active components: the client, which requests data, and the server, which delivers data. Basically, the application's processing is done separately for database queries and updates, and for user interface presentations. Usually the network binds the back end to the front end, although both tiers could be present on the same hardware.

For example, hundreds or thousands of airline seat reservation applications can connect to a central DBMS to request, insert, or modify data. While the clients process the graphics and data entry validation, the DBMS does

all the data processing. Actually, it is inadvisable to overload the database engine with data processing that is irrelevant to the server, thus some processing usually also happens on the clients. The typical client-server architecture is shown in Figure below:

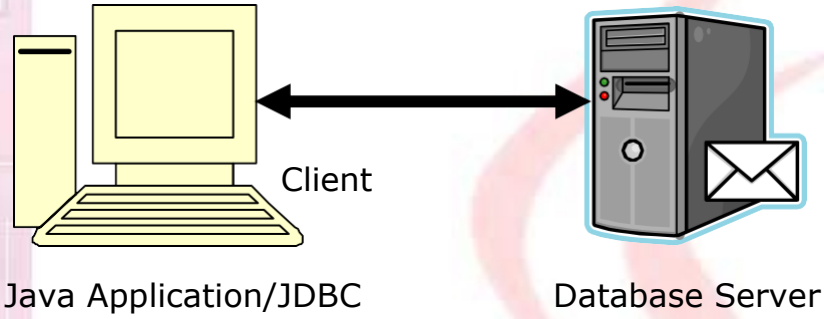


Fig. Two-tier client server architecture

The two tiers are often called as Application layer includes JDBC drivers, business logic and user interfaces whereas second layer i.e. Database layer consists of RDBMS server.

Advantages:

- It is simple in design.
- Client-side scripting offloads work onto the client

Drawbacks:

- Fat client.
- It is inflexible.

Although the two-tiered architecture is common, another design is starting to appear more frequently. To avoid embedding the application's logic at both the database side and the client side, a third software tier may be inserted. In three-tiered architectures, most of the business logic is frozen in the middle tier. In this architecture, when the business activity or business rules change, only the middleware must be modified. Figure below illustrates the three-tier architecture.

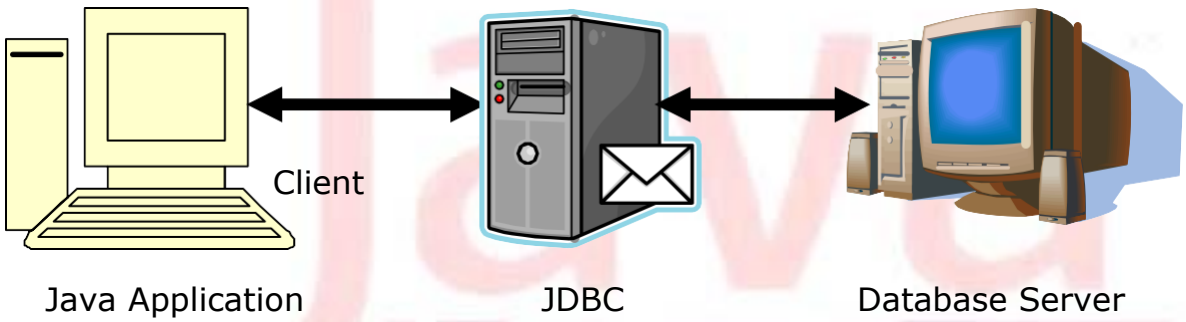


Fig. Three-tier client/server architecture

Advantages:

- Flexible: It can change one part without affecting others.

- It can connect to different databases without changing code.
- Specialization: presentation / business logic / data management.
- It can cache queries.
- It can implement proxies and firewalls.

Drawbacks:

- Higher complexity
- Higher maintenance
- Lower network efficiency
- More parts to configure (and buy)

What is JDBC?

The JDBC stands for Java Database Connectivity. What is this JDBC besides a nifty acronym? It refers to several things, depending on context:

- It's a specification for using data sources in Java applets and applications.
- It's an API for using low-level JDBC drivers.
- It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
- It's based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems.

The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. This interface is used by the developer so he doesn't need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other Java packages such as java.awt. It's not currently a part of the standard Java Developer's Kit (JDK) distribution, but it is slated to be included as a standard part of the general Java API as the java.sql package. Soon after its official incorporation into the JDK and Java API, it will also become a standard package in Java-enabled Web browsers, though there is no definite timeframe for this inclusion. The exciting aspect of the JDBC is that the drivers necessary for connection to their respective databases do not require any pre-installation on the clients: A JDBC driver can be downloaded along with an applet!

The JDBC project was started in January of 1996, and the specification was frozen in June of 1996. Javasoft sought the input of industry database vendors so that the JDBC would be as widely accepted as possible when it was ready for release. And, as we can see from this list of vendors who have already endorsed the JDBC, it's sure to be widely accepted by the software industry:

- Borland International, Inc.
- Bulletproof
- Cyber SQL Corporation
- DataRamp
- Dharma Systems, Inc.

- Gupta Corporation
- IBM's Database 2 (DB2)
- Imaginary (mSQL)
- Informix Software, Inc.
- Intersoft
- Intersolv
- Object Design, Inc.
- Open Horizon
- OpenLink Software
- Oracle Corporation
- Persistence Software
- Presence Information Design
- PRO-C, Inc.
- Recital Corporation
- RogueWave Software, Inc.
- SAS Institute, Inc. ™
- SCO
- Sybase, Inc.
- Symantec
- Thunderstone
- Visigenic Software, Inc.
- WebLogic, Inc.
- XDB Systems, Inc.

The JDBC is heavily based on the ANSI SQL-92 standard, which specifies that a JDBC driver should be SQL-92 entry-level compliant to be considered a 100 percent JDBC-compliant driver. This is not to say that a JDBC driver has to be written for an SQL-92 database; a JDBC driver can be written for a legacy database system and still function perfectly. Even though the driver does not implement every single SQL-92 function, it is still a JDBC driver. This flexibility will be a major selling point for developers who are bound to legacy database systems but who still want to extend their client applications.

The JDBC Structure

The JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the high-level application interface. The low-level programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases. JDBC drivers are quite flexible: They can be local data sources or remote database servers. The implementation of the actual connection to the data source/database is left entirely to the JDBC driver. The structure of the JDBC includes these key concepts:

- The goal of the JDBC is a DBMS independent interface, a "generic SQL database access framework," and a uniform interface to different data sources.
- The programmer writes only one database interface; using JDBC, the program can access any data source without recoding.

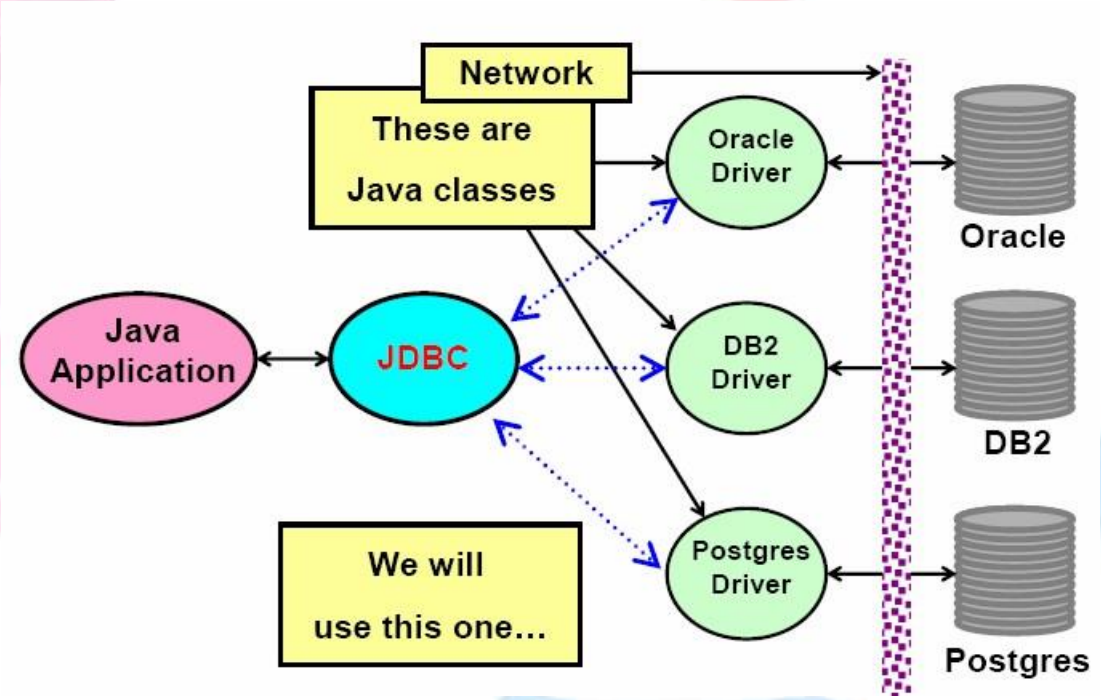


Fig. The JDBC Architecture

JDBC drivers

Sun has defined four categories of JDBC drivers. The categories delineate the differences in architecture for the drivers. One difference between architectures lies in whether a given driver is implemented in native code or in Java code. Native code means whatever machine code is supported by a particular hardware configuration. For example, a driver may be written in C and then compiled to run on a specific hardware platform. Another difference lies in how the driver makes the actual connection to the database. The four driver types are as follows:

Type 1 Driver: JDBC/ODBC Bridge

This type uses bridge technology to connect a Java client to a third-party API such as Open DataBase Connectivity (ODBC). Sun's JDBC-ODBC bridge is an example of a Type 1 driver. These drivers are implemented using native

code. This driver connects Java to a Microsoft ODBC (Open Database Connectivity) data source.

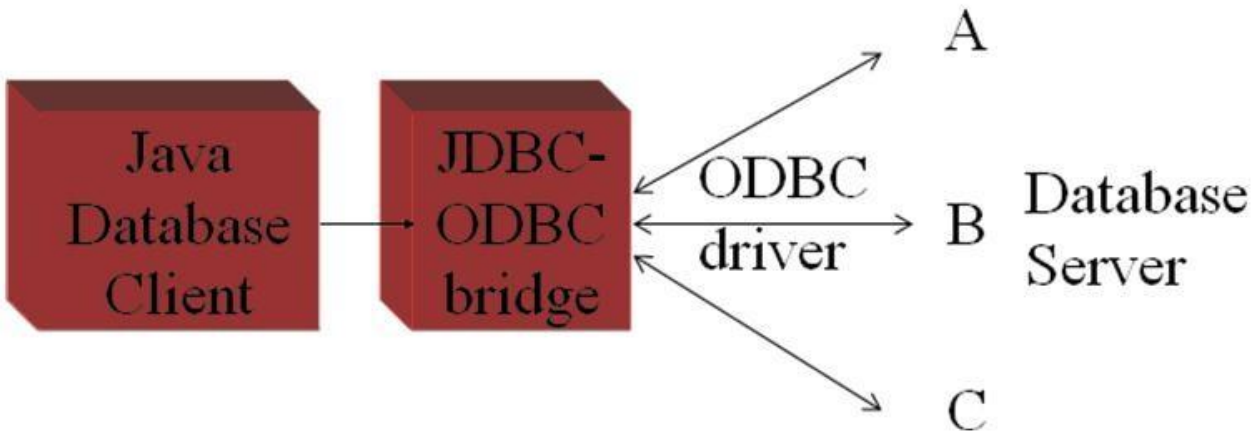


Fig. Type 1 Driver

The Java 2 Software Development Kit from Sun Microsystems, Inc. includes the JDBC-to-ODBC bridge driver (sun.jdbc.odbc.JdbcOdbcDriver). This driver typically requires the ODBC driver to be installed on the client computer and normally requires configuration of the ODBC data source. The bridge driver was introduced primarily to allow Java programmers to build data-driven Java applications before the database vendors had Type 3 and Type 4 drivers.

Type 2 Driver: Native API Driver

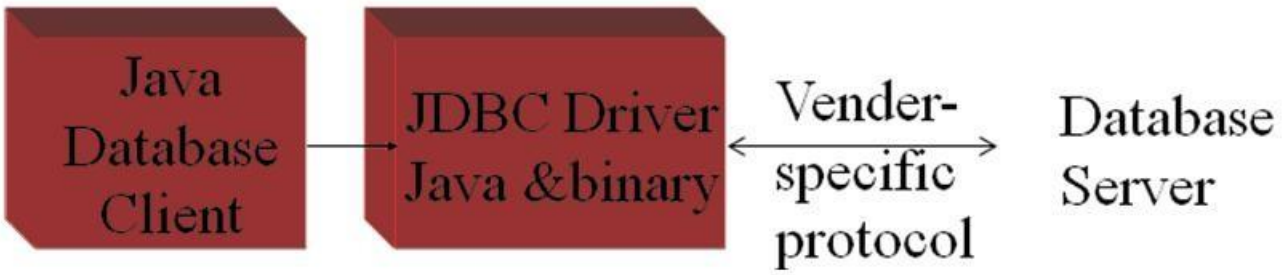


Fig. Type 2 Driver

This type of driver wraps a native API with Java classes. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver. Because a Type 2 driver is implemented using local native code, it is expected to have better performance than a pure Java driver. These drivers enable JDBC programs to use database-specific APIs (normally written in C or C++) that allow client programs to access databases via the Java Native Interface. This driver type translates JDBC into database-specific code. Type 2 drivers were introduced for reasons similar to the Type 1 ODBC bridge driver.

Type 3 Driver: Network Protocol, Pure Java Driver

These drivers take JDBC requests and translate them into a network protocol that is not database specific. These requests are sent to a server, which translates the database requests into a database-specific protocol.

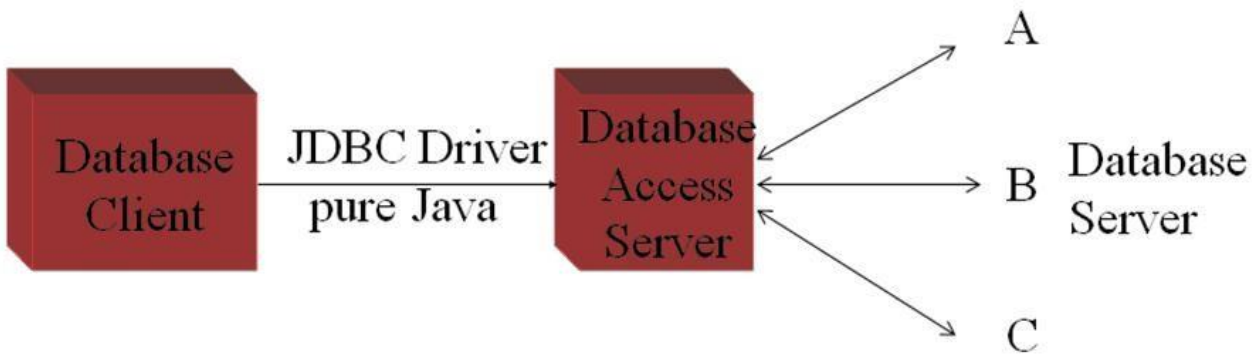


Fig. Type 3 Driver

This type of driver communicates using a network protocol to a middle-tier server. The middle tier in turn communicates to the database. Oracle does not provide a Type 3 driver. They do, however, have a program called Connection Manager that, when used in combination with Oracle's Type 4 driver, acts as a Type 3 driver in many respects.

Type 4 Driver: Native Protocol, Pure Java Driver

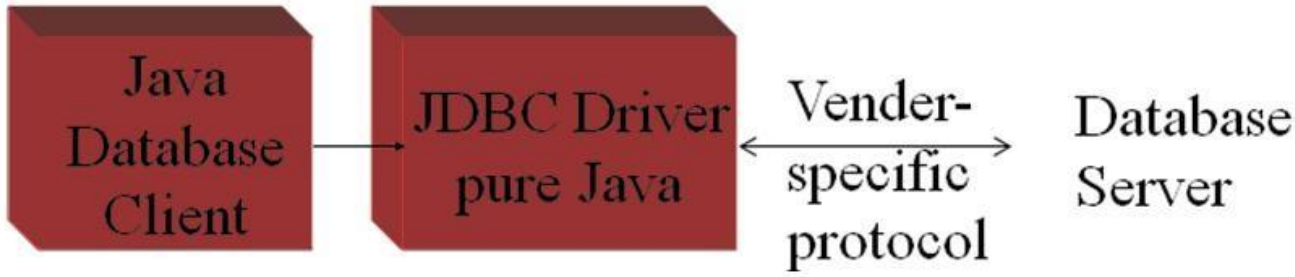


Fig. Type 4 Driver

These convert JDBC requests to database-specific network protocols, so that Java programs can connect directly to a database. This type of driver, written entirely in Java, communicates directly with the database. No local native code is required. Oracle's thin driver is an example of a Type 4 driver.

The JDBC API

The JDBC API is contained in two packages named java.sql and javax.sql. The java.sql package contains core Java objects of JDBC API. There are two distinct layers within the JDBC API: the application layer, which database-application developers use and driver layer which the drivers vendors implement. The connection between application and driver layers is illustrated in figure below:

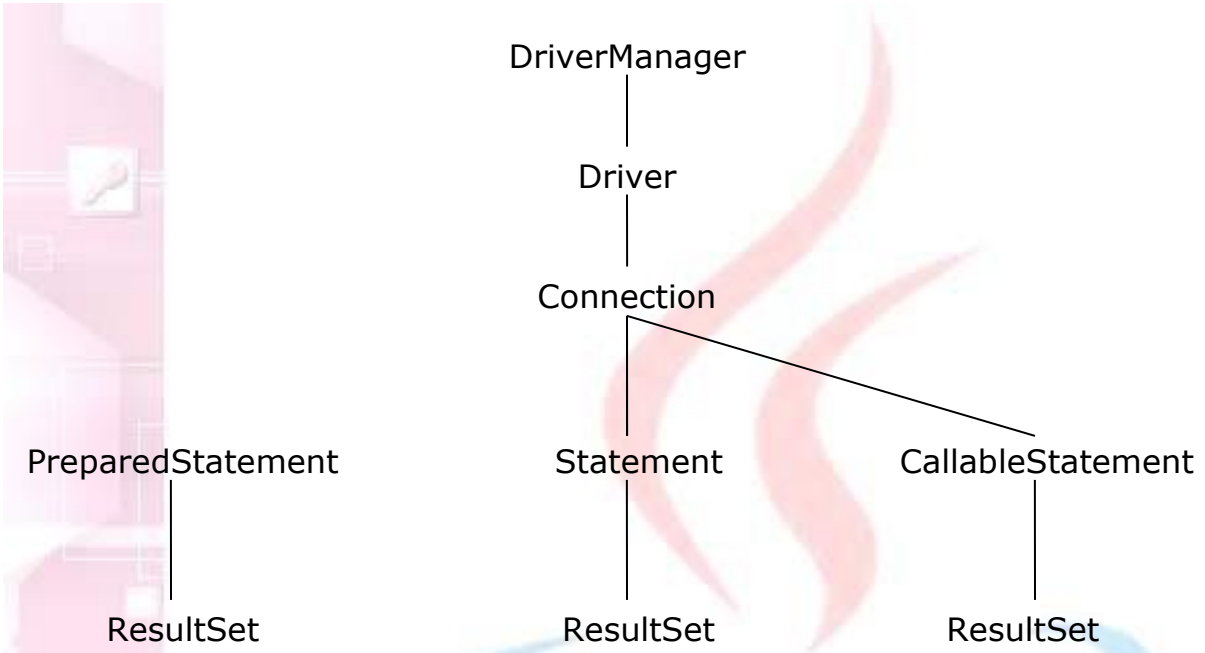


Fig. The JDBC API

There are four main interfaces that every driver layer must implement and one class that bridges the Application and driver layers. The four interfaces are Driver, Connection, Statement and ResultSet. The Driver interface implementation is where the connection to the database is made. In most applications, Driver is accessed through DriverManager class.

The JDBC process

Accessing JDBC / ODBC Bridge with the database

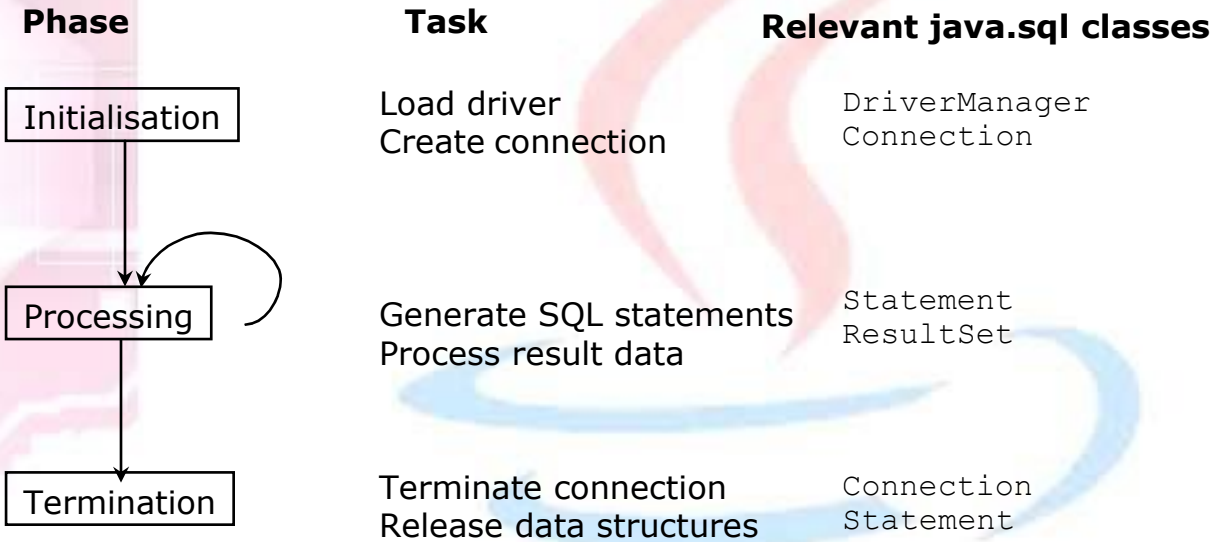
Before actual performing the Java database application, we associate the connection of database source using JDBC – ODBC Bridge. The steps are as follows:

1. Go to Control Panel -> Administrative Tools -> Data Sources.
2. Open Data Sources ODBC icon.

3. Select the tab with heading "User DSN".
4. Click on 'Add' button.
5. Select the appropriate driver as per the database to be used. (e.g. Microsoft ODBC driver for Oracle to access Oracle Database
6. Click finish button and the corresponding ODBC database setup window will appear.

7. Type DSN name and provide the required information such as user name and password for the database (.mdb files) of Microsoft Access Database etc. and click on OK button.
8. Our DSN name will get appeared in user data sources.

There are six different steps to use JDBC in our Java application program. These can be shown diagrammatically as below:



1. Load the driver
2. Define and establish the Connection
3. Create a Statement object
4. Execute a query
5. Process the results
6. Close the connection

Loading the JDBC driver

The JDBC drivers must be loaded before the Java application connects to the DBMS. The `Class.forName()` is used to load the JDBC driver. The developer must write routine that loads the JDBC / ODBC Bridge. The bridge driver called sun.jdbc.odbc.JdbcOdbcDriver. It is done in following way:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Connect to the DBMS

After loading the driver the application must get connected to DBMS. For this we use DriverManager.getConnection() method. The DriverManager is highest class in Java.sql hierarchy and is responsible for managing driver related information.

The DriverManager.getConncetion() method is passed the URL of the database and user ID and password required by the database. The URL is the string object that contains the driver name that is being accessed by the Java program.

The DriverManager.getConncetion() method returns Connection interface that is used throughout the process to reference the database. The signature of this method is:

```
Connection DriverManager.getConncetion(String url,
                    String userID, String password);
```

Here, the URL format is specified as follows:

```
<protocol>:<subprotocol>:<dsn-name>
```

The 'protocol' is a JDBC protocol that is used to read the URL. The 'subprotocol' is JDBC driver name and 'dsn-name' is the name of the database that we provided while creating JDBC Bridge though control panel. We use the following URL for our application:

```
jdbc:odbc:customer
```

here, 'customer' is an example of DSN name given to our database. The user name and password are also provided at the time of creating DSN. It is not compulsory to provide the username and password. For example:

```
Conncetion con;
con = DriverManager.getConnection("jdbc:odbc:customer",
                                  "micro", "pitch");
```

Create Statement object

The createStatement( ) method of Connection interface is used to create the Statement object which is then used to execute the query. For example:

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

Execute the query

The executeQuery() method of Statement object is used execute and process the query which returns the ResultSet object. ResultSet is the object which actually contains the result returned by the query. For example:

```
ResultSet rs = st.executeQuery("select * from customer");
```

Here, the 'customer' is neither database name nor DSN name but it is a table name.

Process the results

The ResultSet object is assigned the results received from the DBMS after the query is processed. The ResultSet object consists of methods used to interact with data that is returned by the DBMS to Java application program. For example, the next() method is used to proceed throughout the result set. It returns true, if the data is available in result set to read.

The ResultSet also contains several getXxx( ) methods to read the value from particular column of current row. For example, getString("name") will read the value from column 'name' in the form of string. Instead of passing column name as parameter, we can pass column as parameter also. Such as, getString(1). For example:

```
String name;
int age;
do
{
      name = rs.getString("name");
      age = rs.getInt("age");
      System.out.println(name+"="+age);
} while(rs.next());
```

Terminate the Connection

The Connection to the DBMS is terminated by using the close() method of the Connection object once Java program has finished accessing the DBMS. The close( ) method throws as exception if problem is encountered when disengaging the DBMS. For example:

```
con.close();
```

The close() method of Statement object is used to close the statement object to stop the further processing.

Statement Objects

Once the connection to the database is opened, the Java application creates and sends a query to access data contained in the database. One of three type of statement objects is used to execute the query immediatelt. A PreparedStatement is used to execute the compiled query and CallableStetement is used to execute the stored procedure.

Statement object

The Statement object is used whenever a Java program needs to immediately execute a query without first having query compiled. The Statement contains three different methods depending upon the type of query these will be used.

1. `executeQuery()`

This method returns the ResultSet object that contains rows, columns and metadata that represent data requested by the query. Its signature is:

```
ResultSet executeQuery(String query);
```

Generally, this method is used to execute only the 'SELECT' query of the SQL.

2. `executeUpdate()`

This method is used to execute the queries that contain INSERT, DELETE and UPDATE statements. This method returns integer indicating the number of rows that were updated by the query. Its signature is:

```
int executeUpdate(String query);
```

For example:

```
int rows = st.executeUpdate("DELETE FROM EMPLOYEES
                                    WHERE STATUS=0");
```

3. `execute()`

It executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts we must then use the methods getResultSet( ) or getUpdateCount( ) to retrieve the result, and getMoreResults( ) to move to any subsequent result(s). Signature is as follows:

```
public boolean execute(String sql)
```

For example:

```
if(st.execute())
  rs = st.getResultSet();
```

Signatures of other methods:

```
public ResultSet getResultSet()
public int getUpdateCount()
public boolean getMoreResults()
```

PreparedStatement object

A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution method is called. Compiling a query is an overhead that is acceptable if the query is called once. However, compiling process can become an expensive overhead if the query is executed several times by the same program during the same session.

A SQL query can be precompiled and executed by using the PreparedStatement object. In such cases a query is created similar to other queries. However, a question mark is given on the place for the value that is inserted into the query after it is compiled. It is the value that changes each time the query is executed.

For doing this process, we need to construct the query with question marks such as,

```
"select * from nation where population > ?"
```

Such type of the query is passed as the parameter to the prepareStatement( ) method of the Connection object which then returns the PreparedStatement object. For example:

```
String query = "select * from nation where population > ?";
PreparedStatement ps = prepareStatement(query);
```

Once the PreparedStatement object is obtained, the setXxx( ) methods of it can be used to replace question mark with the value passed to setXxx() method. There are a number of setXxx() methods available in PreparedStatement object, each of which specifies the data type of value that is being passed to setXxx() method. For example, considering the above query again,

```
ps.setInt(1, 100000);
```

This method requires two parameters. First parameter is an integer that identifies position of the question mark placeholder and second is the value that replaces the question mark. If the query contains two question marks we have to pass second value also using setXxx() method.

Now, we need to use appropriate execute method depending upon type of the query without any parameters. Such as,

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

This will generate the ResultSet object as the execution of the query. The PreparedStatement contain all three execute methods but without any parameters as given below:

```
ResultSet executeQuery( )
int executeUpdate( )
boolean execute( )
```

The setXxx( ) methods:

```
void setBoolean(int index, boolean value);
void setByte(int index, byte value);
void setDate(int index, Date value);
void setDouble(int index, double value);
void setFloat(int index, float value);
void setInt(int index, int value);
void setLong(int index, long value);
void setObject(int index, Object value);
void setShort(int index, short value);
void setString(int index, String value);
```

Example:
Consider the following database:

	Roll	Name	Marks
	1	Rakhee	75
	2	Amit	49
	3	Ajita	63
	4	Rahul	78
	5	Minal	67
	6	Karthik	71

```
import java.sql.*;
class StudentData
{
   public static void main(String args[])
   {
      try
      {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con =
                  DriverManager.getConnection("jdbc:odbc:stud");
       PreparedStatement ps = con.prepareStatement("select *
                  from Student where Marks > ?");
       ps.setInt(1,70); //set question marks place holder
       ResultSet rs = ps.executeQuery(); //execute
```

```
            System.out.println("Students having marks > 70 are:");
            while(rs.next())
                System.out.println(rs.getString(2));
            con.close();
           }
          catch(Exception e){ }
      }
  }
```

Output:

```
Students having marks > 70 are:
Rakhee
Rahul
Karthik
```

CallableStatement Object

The CallableStatement is used to call the stored procedures from within a JDBC application program. A stored procedure is a block of code and is identified by a unique name. The type style of code depends upon the DBMS vendor and can be written in PL/SQL, Transact-SQL, C or another programming language. The stored procedure is executed by invoking name of the stored procedure. For example, a stored procedure written in PL/SQL as given below:

```
CREATE PROCEDURE sp_interest
(id IN INTEGER,
bal IN OUT FLOAT) IS
BEGIN
SELECT balance
INTO bal
FROM account
WHERE account_id = id;

bal := bal + bal * 0.03;

UPDATE account
SET balance = bal
WHERE account_id = id;

END;
```

The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are IN, OUT, INOUT. The IN parameter contains the data that needs to be passed to the stored procedure whose value is assigned using setXxx() method. Whereas, OUT parameter contains the value returned by the stored procedure, if any. The OUT parameter must be registered using registerOutParameter() method and afterwards this is

retrieved by using getXxx() method. The INOUT parameter is a single parameter that is used to both pass information and retrieve information from a stored procedure.

Consider above example, the name of the stored procedure is given as, sp_interest. Its definition is very similar to those of Java method definition. The variable 'id' is an input integer parameter passed to this procedure and variable 'bal' is the float parameter acting as input and output both. The stored procedure contains the SQL query code to perform certain operations depending upon input value to the stored procedure and it returns the value in variable 'bal'. We can now write our code to call this procedure to pass the parameters and to retrieve the information.

After establishing the connection, the prepareCall() method of the Connection object is passed with query of stored procedure call. It returns the object of CallableStatement. The OUT parameter of the procedure must be registered using registerOutParemeter() method which contains following general form:

```
public void registerOutParameter(int parameterIndex, int sqlType)
```

here, *parameterIndex* refers to the index of the parameter passed to that stored procedure. And *sqlType* is type of the value which is expected to retrieve from stored procedure. Generally, *sqlType* is the value of type java.sql.Types.
