Object Oriented Programming using Java 13

Java Database Connectivity

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Introduction to JDBC

Introduction to JDBC

- Many enterprise level applications need to interact with databases for storing information
- For this purpose, we need an API, i.e. ODBC (Open Database Connectivity). The ODBC API was the database API to connect and execute query with the database. ODBC API uses ODBC Driver, which is platform dependent and unsecured
- Java has its own API, JDBC (Java Database Connectivity) that uses JDBC drivers (written in Java)
- JDBC drivers are more compatible with Java applications to provide database communication
- JDBC is a java API to connect and execute query with the database
- JDBC supports a wide level of portability and JDBC is simple and easy to use

Introduction to JDBC...

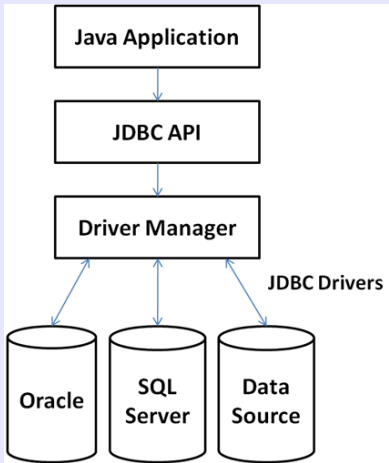
- In JDBC API, a programmer needs a specific driver to connect to specific database
- List of some popular drivers

RDBMS	Driver
Oracle	oracle.jdbc.driver.OracleDriver
MySQL	com.mysql.cj.jdbc.Driver
SyBase	com.Sybase.jdbc4.jdbc.SybDriver
SQLServer	com.microsoft.sqlserver.jdbc.SQLServerDriver
DB2	com.ibm.db2.jcc.DB2Driver

JDBC Architecture

JDBC Architecture

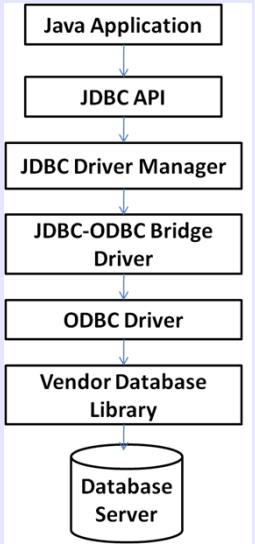
- The main function of the JDBC is to provide a standard abstraction for java applications to communication with databases. Java application that wants to communicate with a database has to be programmed using JDBC API
- The JDBC Driver is required to process the SQL requests and generate the results
- The JDBC driver has to play an important role in the JDBC architecture. The Driver manager uses some specific drivers to effectively connect with specific databases
- JDBC Driver is a software component that enables java application to interact with the database



Types of Drivers

Type-1 Driver (JDBC-ODBC Bridge Driver)

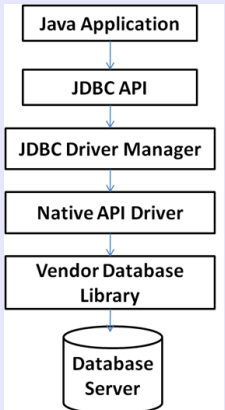
- Type-1 driver acts as a bridge between JDBC and other database connectivity mechanisms such as ODBC
- This driver uses ODBC driver to connect to the database. It converts JDBC method calls into the ODBC method calls
- Advantages
  - Easy to use
  - Can be easily connected to any database
- Disadvantages
  - Performance degraded because large no. of transactions
  - ODBC driver needs to be installed on the client machine



Types of Drivers...

Type-2 Driver (Partial JDBC Driver)

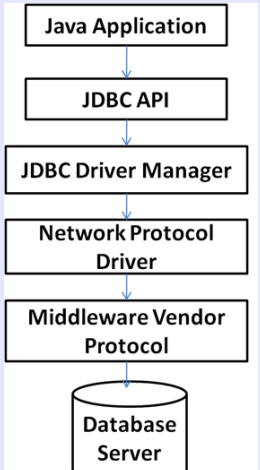
- Type-2 driver uses the client-side libraries of the database. So, this driver is also called as Native API Driver
- This driver converts JDBC method calls into native calls of the database API. Since it is not written entirely in Java, it is called as partial JDBC driver
- Advantages
  - Performance upgraded than JDBC-ODBC bridge driver
  - Suitable to use with server-side applications
- Disadvantages
  - This native driver needs to installed on each client machine
  - Vendor client library needs to be installed on client machine
  - It may increase the cost of application on different platforms



Types of Drivers...

Type-3 Driver (Pure Java Driver for Middleware)

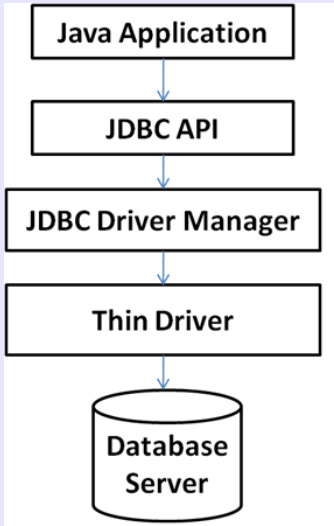
- Type-3 driver is completely implemented in Java
- This driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. So, it is called as Network Protocol Driver
- Advantages
  - No client side library is required on client-side
  - Pure java drivers and auto downloadable
- Disadvantages
  - Network support is required on client machine
  - This driver is costly compared to other drivers



Types of Drivers...

Type-4 Driver (Pure Java Driver with Direct Database Connection)

- Type-4 driver is a pure java driver, which converts JDBC calls directly into the vendor-specific database protocol. So, it is called as Thin Driver
- Advantages
  - This driver is pure java driver and auto downloadable
  - Better performance than all other drivers
  - No software is required at client side or server side
- Disadvantages
  - Drivers depend on the database



Steps in Java Application Development

- JDBC APIs are used by a java application to communicate with a database. In otherwords, we use JDBC connectivity code in java application to communicate with a database
- Steps to connect any java application with the database in java using JDBC:
  - Step1: Register the driver class
  - Step2: Creating connection
  - Step3: Creating statement
  - Step4: Executing SQL statements
  - Step5: Closing connection

Steps in Java Application Development...

Steps in Java Application Development...

- **Step1: (Register the driver class)**
  - Register the driver class with driver manager by using *forName()* method of *Class* class
  - Syntax: Class.forName(Driver Classname)
  - Ex: Class.forName(”oracle.jdbc.driver.OracleDriver”);
- **Step2: (Creating connection)**
  - Create a connection with database server by using *getConnection()* method of *DriverManager* class
  - Syntax: getConnection(String url, String name, String pwd)
  - Ex: Connection con = DriverManager.getConnection(”jdbc:oracle:thin:@localhost:1521:xe”, ”system”,”system”);
- **Step3: Creating statement**
  - *createStatement()* method of Connection interface is used to create statement. This statement object is responsible to execute SQL statements with the database
  - Syntax: createStatement()
  - Ex: Statement stmt = con.createStatement();

Steps in Java Application Development...

Steps in Java Application Development...

- **Step4: (Executing SQL statements)**
  - Execute the SQL statements by using *execute()* or *executeUpdate()* or *executeQuery()* method of statement interface
  - *execute()* method is used for CREATE or DROP statement
  - *executeQuery()* method is only used to execute SELECT statement. *executeUpdate()* method is used to execute all SQL statements except SELECT statement
  - Syntax: executeQuery(String query) executeUpdate(String query)
  - Ex: String query = ”Select * from emp”; Resultset rs=stmt.executeQuery(query);

    String query=”insert into emp values(507,’Asis’, 30)”; stmt.executeUpdate(query)
- **Step5: (Closing the connection)**
  - *close()* method of Connection interface is used to close the connection
  - Syntax: close()
  - Ex: con.close();

Connecting with Oracle Database

Connecting with Oracle Database

- For connecting java application with the Oracle database, the following information are required:
  - **Driver class**: ”oracle.jdbc.driver.OracleDriver”
  - **Connection URL**: ”jdbc:oracle:thin:@localhost:1521:xe” where, *jdbc* is the API, *oracle* is the database, *thin* is the driver, *localhost* is the server name on which oracle is running, *1521* is the port number and *xe* is the oracle service name
  - **username**: user name for oracle database
  - **password**: password of the oracle database user
- To connect java application with oracle database, *jdbc jar* file is required to be loaded

Statement and PreparedStatement

Statement and PreparedStatement

- Statement is used when you want to run SQL query once
  Statement st = con.createStatement();
  st.executeUpdate("UPDATE STUD SET Name="Ram" WHERE Roll=101");
- PreparedStatement is used when you want to use SQL statements many times. The PreparedStatement interface accepts input parameters at runtime
  PreparedStatement st = con.prepareStatemt("UPDATE STUD SET Name=? WHERE Roll=?");
  st.setString(1,"Ram");
  st.setInt(2, 101);
  st.executeUpdate();
- Statement is used to execute normal SQL queries; whereas PreparedStatement is used to execute dynamic SQL queries
- Parameters can't be passed at runtime in Statement; whereas parameters can be passed at runtime in PreparedStatement

Creating Table through JDBC

Creating Table through JDBC

```
import java.sql.*;
class CreateTable{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        st.execute("CREATE TABLE STUD20(Roll Number(7),Name Varchar(30),Age Number(2))");
        System.out.println("Table Created");
        st.close();
        con.close();
        }
    }
```

Inserting Record through JDBC

Inserting Record through JDBC

```
import java.sql.*;
import java.util.*;
class InsertData{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        st.executeUpdate("INSERT INTO STUD20 VALUES(101,'Ramesh',21)");
        System.out.println("Record Inserted");
        st.close();
        con.close();
        }
    }
```

Selecting/Querying Record through JDBC

Selecting/Querying Record through JDBC

- ResultSet is essentially a table of data where each row represents a record and each column represents a field in database. It has a cursor that points to the current row in the ResultSet and we can able to navigate in ResultSet by using the next(), previous(), first(), and last() methods
- Data can be retrieved by using different methods like getString(), getInt(), getDouble() etc.

```
import java.sql.*;
import java.util.*;
class SelectData{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        ResultSet rs=st.executeQuery("SELECT * FROM STUD20");
        System.out.println("Records Are:");
        while(rs.next()){
            System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getInt(3));
            }
        st.close();
        con.close();
        }
    }
```

Updating Record through JDBC

Updating Record through JDBC

```
import java.sql.*;
import java.util.*;
class UpdateData{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        st.executeUpdate("UPDATE STUD20 SET age=19 WHERE Roll=101");
        System.out.println("Record Updated");
        st.close();
        con.close();
        }
    }
```

Deleting Record through JDBC

Deleting Record through JDBC

```
import java.sql.*;
import java.util.*;
class DeleteData{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        st.executeUpdate("DELETE FROM STUD20 WHERE Roll=101");
        System.out.println("Record Deleted");
        st.close();
        con.close();
        }
    }
```

Dropping Table through JDBC

Dropping Table through JDBC

```
import java.sql.*;
class DropTable{
    public static void main(String args[])throws Exception{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:xe","system","system");
        System.out.println("Connected to Oracle Database");
        Statement st=con.createStatement();
        st.execute("DROP TABLE STUD20");
        System.out.println("Table Dropped");
        st.close();
        con.close();
        }
    }
```