JDBC Introduction

The JDBC (Java Database Connectivity) API defines interfaces and classes for writing database applications in Java by making database connections. Using JDBC you can send SQL, PL/SQL statements to almost any relational database. JDBC is a Java API for executing SQL statements and supports basic SQL functionality. It provides RDBMS access by allowing you to embed SQL inside Java code. Because Java can run on a thin client, applets embedded in Web pages can contain downloadable JDBC code to enable remote database access. You will learn how to create a table, insert values into it, query the table, retrieve results, and update the table with the help of a JDBC Program example.

Although JDBC was designed specifically to provide a Java interface to relational databases, you may find that you need to write Java code to access non-relational databases as well. This site teaches you database programming with jdbc and java using a oracle Database

JDBC Product Components

JDBC includes four components:

1. **The JDBC API —**

   The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

   The JDBC API is part of the Java platform, which includes the *Java™ Standard Edition* (Java™ SE) and the *Java™ Enterprise Edition* (Java™ EE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.
2. **JDBC Driver Manager —**

   The JDBC `DriverManager` class defines objects which can connect Java applications to a JDBC driver. `DriverManager` has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

   The Standard Extension packages `javax.naming` and `javax.sql` let you use a `DataSource` object registered with a *Java Naming and Directory Interface™* (JNDI) naming service to establish a connection with a data source. You can use

* JDBC is a pure database connectivity interface which is a pure Java API used to execute SQL statements. JDBC provides a set of classes and interface that can be used by the developer to write database application. Basically JDBC interaction in its simplest form can be broken down into four steps: 1) Open a connection to the database 2) Execute a SQL statement (3) [illegible]

4) Used the connection to the database

either connecting mechanism, but using a DataSource object is recommended whenever possible.

3. **JDBC Test Suite —**

   The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

4. **JDBC-ODBC Bridge —**

   The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

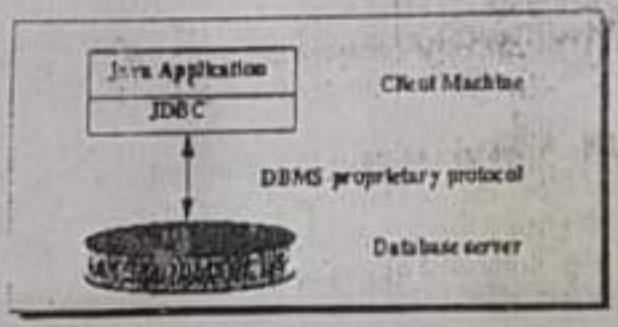
This Trail uses the first two of these these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

JDBC Architecture

Two-tier and three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

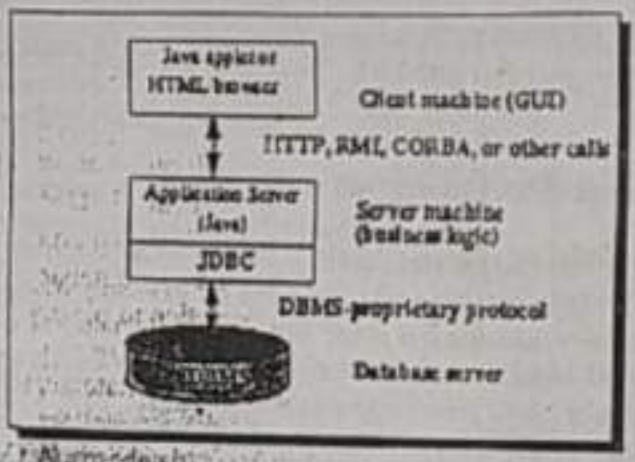
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

Java Database Connectivity Steps

Before you can create a java jdbc connection to the database, you must first import the java.sql package.

1. Loading a database driver,

In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC

Bridge driver is commonly used. The return type of the Class.forName(String ClassName) method is "Class". Class is a class in java.lang package.

```
try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //Or
any other driver
}
catch(Exception x){
      System.out.println( "Unable to load the driver
class!" );
}
```

2. Creating a oracle jdbc Connection

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system. Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object. A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases. A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

JDBC URL Syntax:: jdbc: <subprotocol>: <subname>

- *Each driver has its own subprotocol*
- *Each subprotocol has its own syntax for the source*

Example: For example, we're using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
try{
      Connection dbConnection =
DriverManager.getConnection(url, "loginName", "Password")
}
catch( SQLException x ){
      System.out.println( "Couldn't get connection!" );
}
```

3. Creating a jdbc Statement object,

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

Statement statement = dbConnection.createStatement();

A statement object is used to send and execute SQL statements to a database

Three kinds of Statements

Statement: Execute simple sql queries without parameters.
Statement createStatement()
Creates an SQL Statement object.

Prepared Statement: Execute precompiled sql queries with or without parameters
PreparedStatement prepareStatement(String sql)
returns a new PreparedStatement object. PreparedStatement objects are precompiled SQL statements.

Callable Statement: Execute a call to a database stored procedure.
CallableStatement prepareCall(String sql)
returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

4. Executing a SQL statement with the Statement object, and returning a java ResultSet.

Statement interface defines methods that are used to interact with database via the execution of SQL statements. The Statement class has three methods for executing statements: executeQuery(), executeUpdate(), and execute(). For a SELECT statement, the method to use is *executeQuery*. For statements that create or modify tables, the method to use is *executeUpdate*. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. *execute*() executes an SQL statement that is written as String object.

ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

EXAMPLE code

- **The following simple code fragment gives a simple example of these three steps:**

```
                "jdbc:myDriver:wombat", "myLogin","myPassword");

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
  int x = rs.getInt("a");
  String s = rs.getString("b");
  float f = rs.getFloat("c");
  }
```

Types of JDBC drivers

This topic defines the Java(TM) Database Connectivity (JDBC) driver types. Driver types are used to categorize the technology used to connect to the database. A JDBC driver vendor uses these types to describe how their product operates. Some JDBC driver types are better suited for some applications than others.

Type 1

Type 1 drivers are "bridge" drivers. They use another technology such as Open Database Connectivity (ODBC) to communicate with a database. This is an advantage because ODBC drivers exist for many Relational Database Management System (RDBMS) platforms. The Java Native Interface (JNI) is used to call ODBC functions from the JDBC driver.

A Type 1 driver needs to have the bridge driver installed and configured before JDBC can be used with it. This can be a serious drawback for a production application. Type 1 drivers cannot be used in an applet since applets cannot load native code.

Type 2

Type 2 drivers use a native API to communicate with a database system. Java native methods are used to invoke the API functions that perform database operations. Type 2 drivers are generally faster than Type 1 drivers.

Type 2 drivers need native binary code installed and configured to work. A Type 2 driver also uses the JNI. You cannot use a Type 2 driver in an applet since applets cannot load native code. A Type 2 JDBC driver may require some Database Management System (DBMS) networking software to be installed.

The Developer Kit for Java JDBC driver is a Type 2 JDBC driver.

Type 3

These drivers use a networking protocol and middleware to communicate with a server. The server then translates the protocol to DBMS function calls specific to DBMS.

Type 3 JDBC drivers are the most flexible JDBC solution because they do not require any native binary code on the client. A Type 3 driver does not need any client installation.

Type 4

A Type 4 driver uses Java to implement a DBMS vendor networking protocol. Since the protocols are usually proprietary, DBMS vendors are generally the only companies providing a Type 4 JDBC driver.

Type 4 drivers are all Java drivers. This means that there is no client installation or configuration. However, a Type 4 driver may not be suitable for some applications if the underlying protocol does not handle issues such as security and network connectivity well.

JDBC Driver Types

There are 4 types of JDBC drivers. Commonest and most efficient of which are type 4 drivers. Here is the description of each of them:

- **JDBC Type 1 Driver** - They are JDBC-ODBC Bridge drivers. They delegate the work of data access to ODBC API. They are the slowest of all. SUN provides a JDBC/ODBC driver implementation.
- **JDBC Type 2 Driver** - They mainly use native API for data access and provide Java wrapper classes to be able to be invoked using JDBC drivers.
- **JDBC Type 3 Driver** - They are written in 100% Java and use vendor independent Net-protocol to access a vendor independent remote listener. This listener in turn maps the vendor independent calls to vender dependent ones. This extra step adds complexity and decreases the data access efficiency.
- **JDBC Type 4 Driver** - Tney are also written in 100% Java and are the most efficient among all driver types

Steps required to implement a Java Database Program

JDBC is a set of Java API for executing SQL Statements. This API Consists of a set of classes and interfaces to enable programmers to write pure Database Applications. It is possible to access various relational databases like Sybase, Oracle, Informix using JDBC.

STEP 1: Loading Drivers

First you have to load the appropriate driver. You can use one driver from the available four drivers. However, JDBC-ODBC Driver is the most preferred driver among developers. In order to load the driver, you have to give the following syntax:

```
Class.ForName("sun.jdbc.odbc.JdbcOdbcDriver")
```

STEP 2: Making the Connection

The getConnection() method of the Driver Manager class is called to obtain the Connection Object. The syntax looks like this:

```
Connection conn = DriverManager.getConnection("jdbc:odbc:<DSN NAME>");
```

Here note that getConnection() is a static method, meaning it should be accessed along with the class associated with the method. The DSN Name is the name which you gave in the Control Panel while registering the Database or Table.

STEP 3: Creating JDBC Statements

A Statement object is what send your SQL Query to the Database Management System. You simply create a statement object and then execute it. It takes an instance of active connection to create a statement object. We have to use our earlier created Connection Object "conn" here to create the Statement object "stmt". The code looks like this:

```
Statement stmt = conn.createStatement();
```

STEP 4: Executing the Statement

In order to execute the query, you have to obtain the Result Set object similar to Record Set in Visual Basic and call the executeQuery() method of the Statement interface. You have to pass a SQL Query like select * from students as a parameter to the executeQuery() method. Actually, the RecordSet object contains both the data returned by the query and the methods for data retrieval. The code for the above step looks like this:

```
ResultSet rs = stmt.executeQuery("select * from student");
```

If you want to select only the name field you have to issue a SQL Syntax like

```
Select Name from Student
```

The executeUpdate() method is called whenever there is a delete or an update operation.

STEP 5: Looping through the ResultSet

The ResultSet object contains rows of data that is parsed using the next() method like rs.next(). We use the getXXX method of the appropriate type to retrieve the value in each field.

If the first field in each row of ResultSet is Name (Stores String value), then getString method is used. Similarly, if the Second field in each row stores int type, then getInt() method is used like:

```
System.out.println(rs.getInt("ID"));
```

STEP 6: Closing the Connection and Statement Objects

After performing all the above steps, you have to close the Connection and RecordSet Objects appropriately by calling the close() method. For example, in our above code we will close the object as

```
conn.close()
```

and statement object as

```
stmt.close()
```