JDBC

FANDS INFONET PVT. LTD.
1, Sheshadri, 1, Rambaug Colony, Paud Road, Pune 411 038.
Web - www.fandsindia.com
Email - fands@vsnl.com
Tel - 91-20-25463981 / 25468882

Java DataBase Connectivity(JDBC)

Java offers several benefits to the developer creating a front-end application for a database server. Java is a “write once, run anywhere” language. This means that Java programs may be deployed without recompilation on any of the architecture and operating systems that possesses a Java Virtual Machine. For large corporations, just having a common development platform is a big saving: no longer are programmers required to write separate applications for the many platforms a large corporation may have. Java is also attractive to third party developers – a single Java program can answer the needs of both large and small customers.

This chapter examines Java database Connectivity (JDBC), including how to use the current JDBC API to connect Java applications and applets to database servers.

1.1 The JDBC API

The JDBC API is designed to allow developers to create database front ends without having to continually rewrite their code. Despite standards set by the ANSI committee, each database system vendor has a unique way of connecting and, in some cases, communicating with their system.

The ability to create robust, platform-independent applications and Web-based applets prompted developers to consider using Java to develop front-end connectivity solutions. At the outset, third party software developers met the need by providing proprietary solutions, using native methods to integrate client-side libraries or creating a third tier and a new protocol.

The Java Software Division, Sun Microsystems’ division responsible for the development of Java products, worked in conjunction with the database and database-tool vendors to create a DBMS-independent mechanism that would allow developers to write their client side applications without concern for the particular database being used. The result is the JDBC API, which is part of the core JDK1.2.

JDBC provides application developers with a single API that is uniform and database independent. The API provides a standard to write to, and a standard that considers all of the various application designs. The secret is a set of Java interfaces that are implemented

by a driver. The driver takes care of the translation of the standard JDBC calls into the specific calls required by the database it supports. In the following fig., the application is written once and moved to various drivers. The application remains same; the drivers change. Drivers may be used to develop the middle tier of a multi-tier database design, also known as middleware.

JDBC is not a derivative Microsoft's Open Database Connectivity specification (ODBC). JDBC is written entirely in Java and ODBC is a C interface. While ODBC is usable by non-C languages, like Visual Basic, it has the inherent development risks of C, such as memory leaks. However, both JDBC and ODBC are based on the X/Open SQL Command Level Interface (CLI). Having the same conceptual base allowed work on the API to proceed quickly and make acceptance and learning of the API easier .Sun provides a JDBC-ODBC bridge that translates JDBC to ODBC. This implementation, done with native methods, is very small and efficient.

In general, there are two levels of interfaces in the JDBC API; the Application Layer, where the developer uses the API to make calls to the database via SQL and retrieve the results, and the Driver Layer, which handles all communication with a specific Driver implementation.

1.2 The API Components

The Application Layer, which database-application developers use, and the Driver Layer, which the driver vendors implement .It is important to understand the Driver Layer, if only to realize that the driver creates some of the objects used at the Application Layer. The Figure below illustrates the connection between the Driver and Application layers.

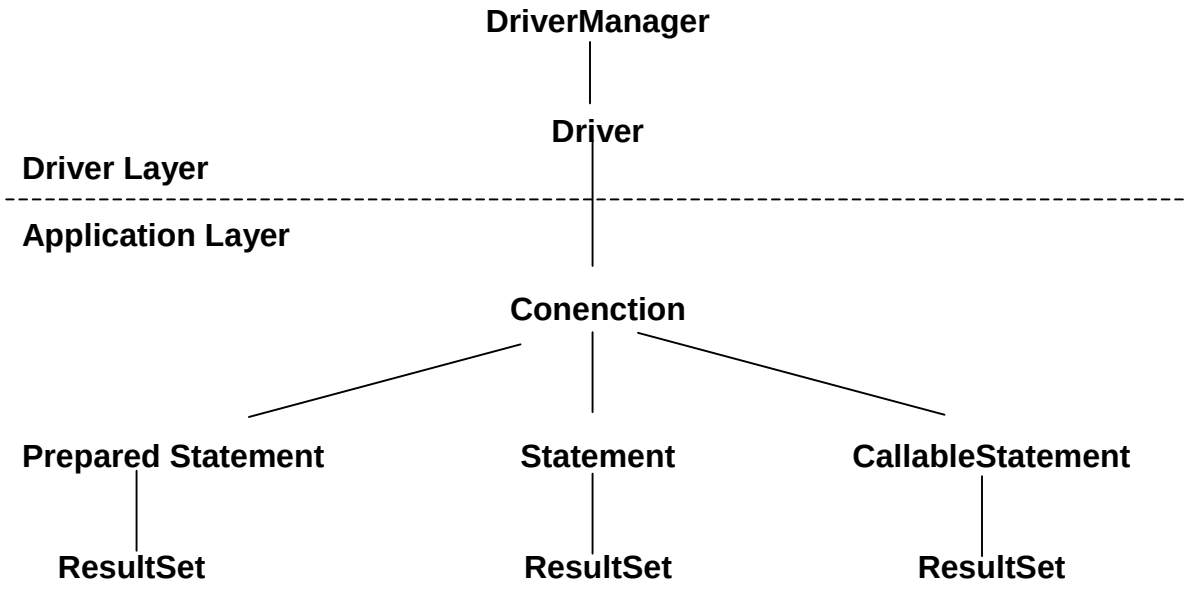

Figure: JDBC API Components

Fortunately the application developer need only use the standard API interface in order to guarantee JDBC compliance. The driver developer is responsible for developing code that interfaces to the database and supports the JDBC application level calls.

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 the Driver, Connection, Statement, and ResultSet. The driver interface implementation is where the connection to the database is made. In most applications, the Driver is accessed through the DriverManager class- providing one more layer of abstraction for the developer.

The Connection, Statement, and ResultSet interfaces are implemented by the driver vendor, but these interfaces represent the methods that the application developer will treat as real object classes and allow the developer to create statements and retrieve results. So the distinction in this section between Driver and Application layers is artificial –but it allows the developer to create the database applications without having to think about where the object are coming from or worry about what specific driver the application will use.

1.2.1 The Driver Layer

There is a one-one correspondence between the database and the JDBC driver. This approach is common in multi-tier designs. The Driver class is an interface implemented by the driver vendor. The other important class is the DriverManager class, which sits above the Driver and Application layers. The DriverManager is responsible for loading and unloading drivers and making connections through drivers. The DriverManager also provides features for logging and database login timeouts.

1.2.1.1 The Driver Interface

Every JDBC program must have at least one JDBC driver implementation. The Driver interface allows the DriverManager and JDBC Application layers to exist independently of the particular database used. A JDBC driver is an implementation of the Driver interface class. Drivers use a string to locate and access databases. The syntax of this string is very similar to a URL string. The purpose of JDBC URL string is to separate the application developer from the driver developer.

The driver vendor implements the Driver interface by creating methods for each of the following interface methods:

Signature: public interface java.sql.Driver

`Public abstract Connection connect (String url, Properties info) throws SQLException`

The driver implementation of this method should check the subprotocol name of the URL string passed for a match with this driver. If there is a match, the driver should then attempt to connect to the database using the information passed in the remainder of the URL. A successful database connection will return an instance of the driver's implementation of a Connection interface (object). The SQLException should be thrown only if the driver recognizes the URL subprotocol but cannot make the database connection. A null is returned if the URL does not match a URL, the driver expected. The username and password are included in a container class called Properties.

`Public abstract Driver PropertyInfo [] getPropertyInfo (String url, Properties info) throws SQLException`

If you are not aware of which properties to use when calling connect (), you can ask the Driver if the supplied properties are sufficient to establish a connection. If they are not an array of necessary properties is provided with the help of the DriverPropertyInfo class.

Public abstract Boolean acceptURL (string url) throws SQLException

It is also possible to explicitly "ask" the driver if a URL is valid.but note that the implementation of this method (typically) only checks if the subprotocol specified in the URL is valid, not wheather a connection can be made.

Public int getMajorVersion ()

Returns the drivers major version number. If the driver's version was at 4.3, this would return integer 4.

Public int getMinorVersion ()

Returns the drivers minor version number. If the driver's version was at 4.3, this would return integer 3.

Public Boolean jdbcComplaint ()

Returns whether or not the driver is a complete JDBC implementation. For legacy systems or lightweight solutions, it may not be possible, or necessary, to have to complete implementation.

The connect () method of driver is the most important method and is called by the DriverManager to obtain the connection object.as figure 15.5 previously showed, the connection object is the starting point of JDBC application layer. The Connection object is used to create Statement objects that perform queries.

The connect () method typically performs the following steps.

1. Checks to see if the URL string provided is valid.
2. Opens a tcp connection to the host and port number specified
3. Attempts to access the named database table (if any)
4. Returns an instance of a connection object.

1.2.1.2 The DriverManager class

The DriverManager class is really a utility class used to manage JDBC drivers. The class provides method to obtain a connection through a driver, register and de-register drivers, set up logging, and set login timeouts for database access. All of the methods in the DriverManager class listed below are static, may be, and may be referenced through the following class name.

Signature: `public class java.sql.DriverManager`

`Public static synchronized connection getConnection (String url, Properties info) throws SQLException`

This method (and the other getconnection () methods) attempt to return a reference to an object implementing the connection interface. The method sweeps through an internal collection of stored driver classes, passing the URL string and properties object info to each in turn. The first driver class that returns a connection is used. Info is a reference to a properties container object of tag/value pairs, typically username/password. This method allows several attempts to make an authorized connection for each driver in the collection.

`Public static synchronized connection getConnection (String url) throws SQLException`

This method calls getConnection (url, info) above with an empty properties object (info).

`Public static synchronized connection getConnection (string uql, string user, string password) throws SQLException`

This method creates a properties object (info), stores the user and password strings in it, and then calls getConnection (url, info) above.

`Public static synchronized void registerDriver (java.sql.Driver driver) throws SQLException`

This method stores the instance of the driver interface implementation into a collection of drivers, along with the programs current security context to identify where the driver came from.

`Public static void setLogStream (PrintStream out)` deprecated.

This method sets a logging/tracing PrintStream that is used by DriverManager.

`Public static void setLoginTimeout (int seconds)`

This method sets the permissible delay a driver should wait when attempting a database login. Drivers are registered with the DriverManager class either at initialization of the DriverManager class or when an instance of the driver is created.

When the DriverManager class is loaded, a section of static code (in the class) is run, and the class names of drivers listed in a java property named JDBC.drivers are loaded. This property can be used to define a list of colon seperated driver class names, such as:

```
jdbc.drivers=imaginary.sql.Driver:oracle.sql.Driver:
weblogic.sql.Driver
```

Each driver name is a class file name (including the package declaration) that the DriverManager will attempt to load through the current CLASSPATH. The DriverManager uses the following call to locate, load and link the named class:

```
Class.forName(driver);
```

If the jdbc.drivers property is empty (unspecified), then the application programmer must create an instance of driver class.

In both cases, the driver class implementation must explicitly register itself with the DriverManager by calling.

```
DriverManager.registerDriver (this);
```

1.2.2 The Application Layer

The application layer encompasses three interfaced that are implemented at the driver layer but are used by the application developer. In java, the interface provides a means of using a general name to indicate a specific object. The general name defines methods that must be implemented by the specific object. For the application developer, this means that the specific driver class implementation is irrelevant. Just coding to the standard JDBC, API's will be sufficient. This is of course, if the driver is JDBC complaint. Recall that this means the database at least supports ANSI SQL-2 entry level.

The three main interfaces are connection, statement and ResultSet. A connection method is obtained from the driver implementation through the drivermanager.getConnection () method call. Once a collection object is returned, the application developer may create a statement object to issue against the database.the result of a statement object to issue against the database. The result of a statement is a ResultSet object, which contains the result of the particular statement (if any).

1.2.2.1 Connection basics

The Connection interface represents a session with the database connection provided by the driver. Typical database connections include the ability to control changes made to actual data stored through transactions. A transaction is a set of operations that are completed in order. A commit action makes the operations store (to change) data in the database. A rollback action undoes the previous transaction before it has been committed. On creation, JDBC connections are in an auto-commit mode; there is no rollback possible. Therefore, after getting a connection object from the driver, the developer should consider setting auto-commit to false with setAutoCommit (Boolean b) method.

When auto-commit is disabled, the connection will support both connection.Commit () and connection.RollBack () method calls. The level of support for transaction isolation depends on the underlaying support for transaction in the support.

A portion of the connection interface definition follows:

Signature: `public interface connection`

`Statement createStatement() throws SQLException`

The connection object implementation will return an instance of an implementation of a statement object. The statement object is then used to issue queries.

`PraperedStatement prapareStatement(String sql) throws SQLException`

The connection object implementation will return an instance of a PraperedStatement object that. The driver may then send the statement to the database; if the database (driver) handles precompiled statements. Otherwise

the driver may wait until the PraparedStatement is executed by an execute method. An exception may be thrown if the driver and database do not implement precompiled statements.

CallableStatement prapareCall (String url) throws SQLException

The connection object implementation will return an instance of a CallableStatement. CallableStatements are optimized for handling stored procedures. The driver implementation may send the SQL string immediately when prapareCall () is complete or may wait until an execute () method occures.

Void setAutoCommit (Boolean autocommit) throws SQLException

Sets a flag in the driver implementation that enables commit/rollback (false) or makes all transactions commit immediately (true).

Void commit throws SQLException

Makes all changes made since the beginning of the current transaction (either the opening of the connection or since the last commit () or rollback ())

Void rollback () throws SQLException

Drops all changes made since the beginning of the current transaction.

The primary use of the connection interface is to create a statement

```
Connection msqlConn;
Statement stmt;

MsqlConn=drivermanager.getConnection (url);
Stmt=msqlConn.creatStatement();
```

This statement may be used to send SQL queries that return a single result set in a ResultSet object reference or a count of the number of records affected by the statement. Statements that need to be called a number of times with slight variations may be executed more effectively using a PraparedStatement. The connection interface is also used to create a CallableStatement whose primary purpose is to execute stored procedures.

TIP

The primary difference between Statement, PreparedStatement, and Callable Statement is that Statement does not permit any parameters within the SQL statement to be executed, PreparedStatement permits In parameters, and CallableStatement permits Inout and Out parameters .In parameters are parameters that are passed into an operation .Due parameters are parameters passed by reference; they are expected to return a result of the reference type. Inout parameters are Out parameters that contain an initial value that may change as a result of the operation. JDBC supports all three parameter types.

1.2.2.2 Statement Basics

A statement is the vehicle for sending SQL queries to the database and retrieving a set of results. Statements can be SQL updates, inserts, deletes, or queries (via Select). The Statement interface provides a number of methods designed to make the job of writing queries to the database easier. There are other methods to perform other operations with a Statement.

Signature: `public interface Statement`

`ResultSet executeQuery (String sql) throws SQLException`

Executes a single SQL query and returns the results in an object of type ResultSet.

`int executeUpdate (String sql) throws SQLException`

Executes a single SQL query that does not return a set of results, but a count of rows affected.

`boolean execute (Strinig sql) throws SQLException`

General SQL statements that may return multiple result sets and /or update counts. This method is most frequently used when you do not know what can be returned, probably because of a user entering the SQL statement directly. The getResultSet (), getUpdateCount (), and getMoreResults () methods are used to retreive the data returned.

`ResultSet getResultset() throws SQLException`

Returns the current data as the result of a statement execution as a ResultSet object. Note that if there are no results to be read or if the result is an update count, this method returns null. Also, note that the results are cleared.

`Int getUpdateCount () throws SQLException`

Returns the status of an Update, Insert, or Delete query or a stored procedure. The value returned is the number of rows affected. A –1 is returned if there is no update count or if the data returned is a result set. Once read, the update count is cleared.

`Boolean getMoreResults () throws SQLException`

Moves to the next result in a set of multiple results/update counts. This method returns true if the next result is a ResultSet object. This method will also close any previous ResultSet read.

Statements may or may not return a ResultSet object, depending on the Statement method used. The executeUpdate () method, for example, is used to execute SQL statements that do not expect a result (except a row count status):

```
int rowCount;
rowCount = stmt.executeUpdate (
“DELETE FROM customer WHERE CustomerID = ‘McG10233’”);
```

SQL statements that return a single set of results can use the executeQuery() method. This method returns a single ResultSet object. The object represents the row information returned as a result of the query.

```
Resultset results;
Results = stmt.executeQuery (“SELECT * FROM stock”);
```

SQL statement that execute stored procedures (or trigger a stored procedure) may return more than one set of results. The execute() method is used a general-purpose method that can return a single result set, a result count, or some combination thereof. The method returns a Boolean flag that is used to determine where there are more results. Because a result set could contain either data or the count of an operation that returns a row count, the getResultSet(), getMoreResults(), and getUpdateCount() methods are used.

The PreparedStatement interface extends interface extends the Statement interface. When there is a SQL, statement that requires repetition with minor variations, the PreparedStatement provides an efficient mechanism for passing a precompiled SQL statement that uses input parameters.

Signature: `public interface PreparedStatement extends Statement`

PreparedStatement parameters are used to pass data into a SQL statement, so they are considered IN parameters and are filled in by using setType methods.

The CallableStatement interface is used to execute SQL stored procedures. CallableStatement inherits from the PreparedStatement interface, so all of the execute and setType methods are available. Stored procedures have varying syntax among database vendors, so JDBC defines a standard way for all RDBMS to call stored procedures.

Signature: `public interface CallableStatement extends PreparedStatement`

The JDBC uses an escape syntax that allows parameters to be passed as In parameters and Out parameters. The syntax also allows a result to be returned; and if this syntax is used, the parameter must be registered as an Out parameter.

setType Methods

Method Signature	**Java Type**	**SQL Type From the database**
void setByte (int index, byte b)	byte	TINYINT
void setShort (int index, short s)	short	SMALLINT
void setInt (int index, int i)	int	INTEGER
void setLong (int index, long l)	long	BIGINT

getType methods

Method Signature	**Java Type**	**SQL Type From the database**
boolean getBoolean (int index)	boolean	BIT
byte getByte (int index)	byte	TINYINT

short getShort (int index)	short	SMALLINT
int getInt (int index)	int	INTEGER

ResultSet Basics

The ResultSet interface defines methods for accessing tables of the data generated as the result of executing a Statement. ReaultSet column values may be accessed in any order; they are indexed and may be selected by either the name or the number (numbered from one to n) of the first row of data returned. The next () method moves to the next row of data.

A partial look at the ResultSet interface follows:

Signature: `public interface ResultSet`

`boolean next () throws SQLException`

Positions the ResultSet to the next row; ResultSet row position is initially the first row of the result set.

`ResultSetMetaData getMetaData () throws SQLException`

Returns an object of the current result set; the number of columns, the type of each column, and properties of the results.

`void close () throws SQLException`

Normally a ResultSet is closed when another Statement is executed, but it may be desirable to release the resources earlier.

The 1.2JDK intorduces several methods with the JDBC2.0 API. With JDBC 2.0, additional capabilities are available that permit non-sequential reading of rows, as well as updating of rows, while reading. A partial look at the JDBC 2.0 methods of ResultSet follows:

`int getType () throws SQLException`

Returns the type of Result set, determining the manner in which you can read the results. Valid return values are TYPE_FORWARD-ONLY, TYPE-STATIC, TYPE_KEYSET, or TYPE_DYNAMIC .If TYPE_FORWARD_ONLY is returned

most of the remaining methods will throw a SQLException if they are attempted.

`boolean first () throws SQLException`

Positions the ResultSet at the first row. Return true if on a valid row, false otherwise.

`boolean last () throws SQLException`

Positions the ResultSet at the last row. Return true if on a valid row, false otherwise

`boolean previous () throws SQLException`

Positions the ResultSet at the previous row.Return true if on a valid row, false otherwise

`boolean absolute (int row) throws SQLException`

Positions the ResultSet at the designated row. If the row requested is negative positions ResultSet at row relative to end of set.

`boolean relative (Int row) throws SQLException`

Positions the ResultSet at the row, relative to the current position.

`boolean isFirst () throws SQLException`

JDBC 2.0 indicates whether the cursor is on the first row of the ResultSet.

`boolean isLast () throws SQLException`

JDBC 2.0 indicates whether the cursor is on the last row of the ResultSet.

ResultSetMetaData besides being able to read data from a ResultSet object, JDBC provides an interface to allow the developer to determine what type of data was returned. The ResultSetMetaData interface is similar to the DatabaseMetaData interface in the concept, but is specific to the current ResultSet. As with DatabaseMetaData, it is unlikely that many developers will use this interface, since most applications are written with an understanding of the database schema and column names and values. However, ResultSetMetaData is useful in dynamically determining the meta-data of a ResultSet returned from a stored procedure or from a user-supplied SQL statement.

The following code demonstrates the displaying of results with the help of ResultSetMetaData when the contents are unknown:

```
ResultSet results = stmt.executeQuery (sqlString)
ResultsetMetaData meta = results.getMetaData ();
Int columns = 0;
boolean first = true;
while (results.next ()) {
     If (first) {
          columns = meta.getColumnCount();
          for (int i=1; i<=columns; i++)  {
          System.out.print  (metadata.getColumnName(i) + "\t");
          }
          System.out.println();
          first=false;
          }
          for (Int i=; i<=columns; i++)  {
          System.out.print (results.getString (i)+ "\t");
          }
          System.out.println();
          }
```