An Introduction to Basic JDBC

Alan P Sexton

<aps@cs.bham.ac.uk>

Copyright © 2005 Alan P. Sexton

Revision History

Revision 1.0	27 July 2001	aps

This is the first version of a JDBC tutorial. Currently this talks only about using an Access database through a JDBC-ODBC bridge. Later versions shall cover using PostgreSQL on Solaris, Linux and Windows. I assume the use of the Standard Edition JDK 1.3.1.

Table of Contents

1. Basic JDBC

JDBC (the Java Database Connection) is the standard method of accessing databases from Java. Sun developed the JDBC library after considering Microsoft's ODBC. Their aims were to get something similar but easier to learn and use: ODBC is complex because it has a few very complex calls. JDBC has split up this complexity into many more calls, but with each of them being relatively simple.

Accessing a database using JDBC involves a number of steps:

1. Get a *Connection* object connected to a database
2. Get a *Statement* object from an open *Connection*object
3. Get a ResultSet from a *Statement*'s query execution
4. Process the rows from the *ResultSet*

While the above is the standard pattern, there are variations: Instead of a *Statement*, you can get a *PreparedStatement* (which allows a query to be pre-compiled for extra performance when executed repetitively). An update query may only return a count of the number of rows updated, inserted or deleted instead of a *ResultSet*. There are calls to get information about the database and about *ResultSet* objects (e.g. the number, names and types of columns). There are calls to support transactional updates to the database.

1.1. Packages

The main package used is

```
import java.sql.* ;
```

This is in the Standard Edition JDK. There is an extension package from the Enterprise Edition JDK. However, this is rarely necessary and even then only for the use of some advanced features such as connection pooling:

```
import javax.sql.* ;
```

1.2. Drivers

Individual database management systems require a JDBC driver to be accessed via JDBC. Sun provides, as part of the Standard Edition JDK, a JDBC-ODBC bridge driver, which lets you connect to any ODBC database. This driver is intended for test and study purposes and is not recommended for commercial use (it is slow, missing some features and is somewhat buggy). Most DBMS vendors supply their own drivers with their products and there are many third party drivers available (both commercial and free)

The first step in accessing the database is to load a driver. First your driver should be installed on the system (usually this requires having the driver jar file available and its path name in your classpath. If you are using the JDBC-ODBC bridge driver, then you have to register your database with ODBC (on windows, use the "ODBC data sources" tool in the control panel). The class name of the JDBC-ODBC bridge driver is *sun.jdbc.odbc.JdbcOdbcDriver*. When a driver is loaded, it registers itself with Driverman-ager which is then used to get the Connection.

There are a number of alternative ways to do the actual loading::

1. Use *new* to explicitly load the Driver class. This hard codes the driver and (indirectly) the name of the database into your pro-gram and is not recommended as changing the driver or the database or even the name or location of the database will usually require recompiling the program.

2. Class.forName takes a string class name and loads the necessary class dynamically at runtime. This is a safe method that works well in all Java environments although it still requires extra coding to avoid hard coding the class name into the pro-gram.

3. The System class has a static Property list. If this has a Property *jdbc.drivers* set to a ':' separated list of driver class names, then all of these drivers will be loaded and registered automatically. Since there is support for loading property lists from files easily in Java, this is a convenient mechanism to set up a whole set of drivers. When a connection is requested, all loaded drivers are checked to see which one can handle the request and an appropriate one is chosen. Unfortunately, support for us-ing this approach in servlet servers is patchy so we will stay with method 2 above but use the properties file method to load the database url and the driver name at runtime:

   ```
   Properties props = new Properties() ;
   FileInputStream in = new FileInputStream("Database.Properties") ;
   props.load(in) ;

   String drivers = props.getProperty("jdbc.drivers") ;
   Class.forName(drivers) ;
   ```

The Database.Properties file contents look like this:

```
# Default JDBC driver and database specification
jdbc.drivers      =  sun.jdbc.odbc.JdbcOdbcDriver
database.Shop  =  jdbc:odbc:Shop
```

1.3. Getting a Connection

To get a connection, we need to specify a url for the actual database we wish to use. The form of this url is specific to the driver we are using. With the driver loaded, we can use the properties file above to get the database url. Using the Sun JDBC-ODBC bridge driver, the url of the database is *jdb:odbc:xxx* where xxx is the ODBC data source name registered for your database. (The name of the property we use is unimportant)

```
String database = props.getProperty("database.Shop") ;
Connection con = DriverManager.getConnection(database) ;
```

Warning

Microsoft Access, accessed through the JDBC-ODBC bridge driver, is not thread safe. This means that multiple different concurrent connections to Access will not necessarily work in a correct manner. In consequence, no application should have multiple Connection objects open at the same time and, therefore, use of Microsoft Access for Web Database work is severely limited (although it is still fine for educational purposes).

1.4. Using Statements

A Statement is obtained from a Connection:

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

Once you have a Statement, you can use it to execute, and control the execution of, various kinds of SQL queries.

- Use *stmt.executeUpdate* with a string argument containing the text of an SQL update query (INSERT, DELETE or UPDATE). This returns an integer count of the number of rows updated.

- Use *stmt.executeQuery* with a string argument containing the text of an SQL SELECT query. This returns a *ResultSet* object which is used to access the rows of the query results.

- You can use *stmt.execute* to execute an arbitrary SQL statement which may be of any type, but extracting the results, whether an integer or a *ResultSet*, is less convenient. This is usually only used where you want a generalized access to the database that allows programmatic generation of queries.

```
int count = stmt.executeUpdate("INSERT INTO Customers " +
                               "(CustomerFirstName, CustomerLastName, CustomerAddress) "
                               "VALUES ('Tony', 'Blair', '10 Downing Street, London')") ;
ResultSet rs = stmt.executeQuery("SELECT * FROM Customers") ;
// do something with count and RS
```

Note

The syntax of the SQL string passed as an argument must match the syntax of the database being used. In particular, appropriate quoting of special characters must be used. For example, if a name, *O'Neill*, is to be inserted, it has to be entered as

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Customers" +
                                 "WHERE CustomerLastName = 'O''Neill'") ;
```

1.5. Working with ResultSets

If you do not know exactly the table structure (the schema) of the *ResultSet*, you can obtain it via a *ResultSetMetaData* object.

```
ResultSetMetaData rsmd = rs.getMetaData() ;
int colCount = rsmd.getColumnCount() ;

for (int i = 1 ; i <= colCount ; i++)
{
 if (i > 1)
  out.print(", ");
 out.print(rsmd.getColumnLabel(i)) ;
}
out.println() ;
```

Once a *ResultSet* has been obtained, you can step through it to obtain its rows, or, more specifically, the fields of its rows:

```
while (rs.next())
{
 for (int i = 1 ; i <= colCount ; i++)
 {
  if (i > 1)
   out.print(", ");
  out.print(rs.getObject(i)) ;
 }
 out.println() ;
}
```

Note that the column numbers start at 1, not 0 as in Java arrays. More conveniently, although somewhat less efficiently, there is a *getObject* method for *ResultSet* which takes a String argument containing the column name. There are also *getxxx* methods that take the *String* name of the column instead of the column number. Thus the above code could have been written:

```
while (rs.next())
{
 out.println( rs.getObject("CustomerID")   + ", " +
   rs.getObject("CustomerFirstName") + ", " +
   rs.getObject("CustomerLastName")  + ", " +
   rs.getObject("CustomerAddress") ) ;
}
```

Instead of *getObject*, you can use type specific methods, *getInt*, *getString*, etc. However, these have a major disadvantage: if the field is of primitive type such as int, float etc., then if the field is actually null in the database, then there is no value that can be returned that is indistinguishable from some valid value. There is a mechanism for finding out whether the last value obtained was really null or not: *wasNull*, but this must be called immediately after the *getXxx* method and before the next such call. If you use *getObject*, then if the field was null then the object value returned will be null so you can pass this value around and check for it at your convenience. Note also that printing is the most common thing to do with retrieved values, and passing a null to print will print the string "null". Thus for many cases no extra processing of nulls will be necessary.

1.6. Prepared Statements

Rather than *Statement* objects, *PreparedStatement* objects can be used. This have the advantages over plain *Statement* objects of:

- For repetitive queries that are very similar except for some parameter values, they are considerably more efficient because the

SQL is compiled once and then executed many times, with the parameter values substituted in each execution

- The mechanism for inserting parameter values takes care of all necessary special character quoting in the correct manner for the connected database

The *PreparedStatement* has its SQL text set when it is constructed. The parameters are specified as '?' characters. After creation, the parameters can be cleared using *clearParameters* and set using *setInt*, *setString*, etc. methods (parameter positions start at 1) and the statement can then be executed using *execute*, *executeUpdate* or *executeQuery* methods as for *Statement* and with the same return types but with no arguments (as the SQL text has already been set when the statement was created):

```
PreparedStatement pstmt = con.prepareStatement(
                              "INSERT INTO Customers " +
                              "(CustomerFirstName, CustomerLastName, CustomerAddress) "+
                              "VALUES (?, ?, ?)") ;

pstmt.clearParameters() ;
pstmt.setString(1, "Joan") ;
pstmt.setString(2, "D'Arc") ;
pstmt.setString(3, "Tower of London") ;
count = pstmt.executeUpdate() ;
System.out.println ("\nInserted " + count + " record successfully\n") ;

pstmt.clearParameters() ;
pstmt.setString(1, "John") ;
pstmt.setString(2, "D'Orc") ;
pstmt.setString(3, "Houses of Parliament, London") ;
count = pstmt.executeUpdate() ;
System.out.println ("\nInserted " + count + " record successfully\n") ;
```

1.7. Transactions

Transactions are a mechanism to group operations together so that either all of them complete together successfully or none of them do. This avoids database consistency problems that can occur if some groups of operations are only partly completed. Think of a bank transfer that requires withdrawing money from one account to deposit in another. If the withdraw is completed but the deposit fails then the customer is likely to be very unhappy. If the deposit succeeds but the withdraw fails then the bank is likely to be very unhappy. Actually, transactions handle other aspects of consistency as well. For example, ensuring that a second transaction sees the database as if either the first transaction has completely finished or as if it has not started yet but *not* as if some of the first transaction's operations have completed but not others - even if both transactions are running simultaneously.

When a *Connection* is obtained, by default its AutoCommit property is set to true. This means that every query execution is committed immediately after it is executed and before the next one is executed. To enable grouping of operations in transactions, you have to switch the AutoCommit property off:

```
con.setAutoCommit(false) ;
```

Now you have to obtain new statement objects from the connection (the old ones won't work), and query or update as usual. When all operations that you want to group together have completed, you *must* commit the updates to the database:

```
con.commit() ;
```

At this point you can continue with more operations which will be grouped into a new transaction or you can switch AutoCommit back on:

```
con.setAutoCommit(true) ;
```

If anything goes wrong during a transaction (e.g. an Exception is thrown or an error means that you cannot complete your group of operations) then you have to undo all operations in your transaction so far:

```
con.rollBack() ;
```

Note

If the database or the machine crashes, rollBack will (essentially) be called for you automatically to clean up uncommitted transactions when the database is restarted.

You should make every effort to minimise the length of time that you have open transactions running as they hold expensive resources and, in particular, locks in the database system which may stop any other competing transactions from proceeding. In many cases, careful design may obviate many needs to use anything other than the standard AutoCommit mode in all except a very few cases.

Getting all this working correctly requires careful attention to your Exception handling. You must embed a transaction in a try clause so that any exception will trigger a rollback. If you do have to handle an exception (and therefore rollback) in a method that normally closes open connections before returning, make sure that this does not create a loop hole that allows the method to return without closing the connection. You can use a *finally* block to ensure that this is handled correctly.

Finally, note that when you modify the AutoCommit status of a connection, all operations by any thread using that connection object are run in the same transaction. Therefore you have to be very careful about sharing connection objects between different threads (particularly important in servlet and JSP code). The simple rule is that you can share without problems a connection which has been set to auto commit. Do not share non auto committing connections unless you use some other mechanism to make sure that you don't end up merging different transactions into one - with consequences for committing and roll backs. An appropriate mechanism would be connection pooling: an advanced topic that is not discussed in this tutorial.

1.8. Exceptions and Warnings

SQLExceptions, thrown when something goes wrong in JDBC processing, has a slight peculiarity compared to other types of exceptions. When an *SQLException* is caught, it may have a chain of further *SQLExceptions* attached to it. This is because a number of problems may occur as part of one overall problem and you may need information about all of them to handle it correctly. To process the chain of exceptions, you should have code similar to the following:

```
try
{

 // do some JDBC calls
}
catch (SQLException e)
{
 do

System.out.println(e.getMessage()) ;
 while ((e = e.getNextException()) != null) ;
}

catch (Exception e)

{

 // handle non-SQLException exceptions

}
```

There is another object related to exceptions that can give information about various kinds of problems: this is an *SQLWarning*. They are not exceptions and do not interrupt the normal flow of control but a *Connection*, *Statement* or *ResultSet* object can be queried for warnings after they have been used. The most common type of warning is a *Data Truncation* error. Otherwise many programmers don't query them at all.

1.9. Sample Database

The database we use for the sample program and the exercise is a (seriously) simplified version of a Microsoft Access database suitable for handling on-line book orders. We omit details that are necessary in practice but do not add anything qualitative to the programming exercises. Thus, for example, there is no information kept about authors, ISBN, delivery charges or options, etc.

The following is a picture of the relationship structure showing the tables, columns and attributes:

By way of explanation:

- A Customer can have many Orders.
- An Order can have many OrderDetail records.
- Each OrderDetail record refers to precisely one Book.
- All the primary keys ID fields (in each case the first column name in the corresponding table in the figure above) are AutoNum generated integers and therefore the primary key of each table does not need to be included when inserting new records.
- Referential integrity rules are maintained on all foreign keys (ID fields which not in the first position in the tables of the figure above). Hence, for example, it will cause an error to try to insert an Order record with an Orders.CustomerID for which there is no corresponding Customers record.
- OrderDetail.Quantity is an integer.
- Books.BookPrice is a double (it should really be Currency which is MSAcess's equivalent to SQL's Numeric, which in turn is handled in Java via the java.math.BigDecimal class. However, since that adds an extra level of complication that does not contribute anything significant to the exercise, it is left as double here).
- *CustomerFirstName*, *CustomerLastName CustomerAddress*, and *BookName* are Strings.
- Orders.OrderDate is a Date/Time object in MSAccess which is handled as a JDBC TIMESTAMP object which in turn is handled as a *java.sql.Timestamp* object.

Timestamp objects can be set to the current time as follows:

```
long millisecs = System.currentTimeMillis() ;
Timestamp ts = new java.sql.Timestamp(millisecs) ;
```

They can be set to a specific value using the *valueOf* method with a string argument:

```
ts = Timestamp.valueOf("2001-07-06 14:25:29.9") ;
```

As should be expected, *ResultSet* has the usual *getTimestamp*, *putTimestamp* and *updateTimestamp* methods and *PreparedStatement* has a *setTimestamp* method.

BigDecimal works rather similarly, see the online API documentation for details.

1.10. Sample JDBC Program

The following program shows a full program using each of the features discussed above: Customer.java [files/Customer.java] You also need the Database.Properties [files/Database.Properties] file and the Shop.mdb [files/Shop.mdb] database in the same directory. Finally, you need to create an entry in your *ODBC data sources* (from the control panel) for Shop.mdb

1.11. JDBC Exercises

1. List books application

   a. Write an application that takes a string from the command line and lists book details for books whose titles start with this string.

2. Add books to an order application

   a. Write an application that takes a customer firstname, a customer lastname and a book title prefix from the command line.

   b. It should check that the customer exists in in database and exit with an error message otherwise.

   c. If there is no open order for that customer then a new open order is created.

   d. All books that match the book title prefix should be added to the open order.

   e. Print to *System.out* a listing of the customer's details and the customer's open order.

   f. (Optional) Rather than simply adding new OrderDetail records for each matching book to the existing open order, simply increment the Quantity for OrderDetail records that already refer to the book(s) being added and only add new OrderDetail records for books that do not already appear in the open order