Objective Viewpoint

DATABASES
Spring 2001 - 7.3

JDBC - Java Database Connectivity

by *Kevin Henry*

What is JDBC?

Java Database Connectivity (**JDBC**) is a programming framework for Java developers writing programs that access information stored in databases, spreadsheets, and flat files. JDBC is commonly used to connect a user program to a "behind the scenes" database, regardless of what database management software is used to control the database. In this way, JDBC is cross-platform [1]. This article will provide an introduction and sample code that demonstrates database access from Java programs that use the classes of the JDBC API, which is available for free download from Sun's site [3].

A database that another program links to is called a data source. Many data sources, including products produced by Microsoft and Oracle, already use a standard called Open Database Connectivity (ODBC). Many legacy C and Perl programs use ODBC to connect to data sources. ODBC consolidated much of the commonality between database management systems. JDBC builds on this feature, and increases the level of abstraction. JDBC-ODBC bridges have been created to allow Java programs to connect to ODBC-enabled database software [1].

This article assumes that readers already have a data source established and are moderately familiar with the Structured Query Language (SQL), the command language for adding records, retrieving records, and other basic database manipulations. See Hoffman's tutorial on SQL if you are a beginner or need some refreshing [2].

Using a JDBC driver

Regardless of data source location, platform, or driver (Oracle, Microsoft, etc.), JDBC makes connecting to a data source less difficult by providing a collection of classes that abstract details of the database interaction. Software engineering with JDBC is also conducive to module reuse. Programs can easily be ported to a different infrastructure for which you have data stored (whatever platform you choose to use

in the future) with only a driver substitution.

As long as you stick with the more popular database platforms (Oracle, Informix, Microsoft, MySQL, etc.), there is almost certainly a JDBC driver written to let your programs connect and manipulate data. You can download a specific JDBC driver from the manufacturer of your database management system (DBMS) or from a third party (in the case of less popular open source products) [5]. The JDBC driver for your database will come with specific instructions to make the class files of the driver available to the Java Virtual Machine, which your program is going to run. JDBC drivers use Java's built-in `DriverManager` to open and access a database from within your Java program.

To begin connecting to a data source, you first need to instantiate an object of your JDBC driver. This essentially requires only one line of code, a command to the `DriverManager`, telling the Java Virtual Machine to load the bytecode of your driver into memory, where its methods will be available to your program. The `String` parameter below is the fully qualified class name of the driver you are using for your platform combination:

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

Connecting to your database

To actually manipulate your database, you need to get an object of the `Connection` class from your driver. At the very least, your driver will need a URL for the database and parameters for access control, which usually involves standard password authentication for a database account.

As you may already be aware, the Uniform Resource Locator (URL) standard is good for much more than telling your browser where to find a web page:

```
http://www.vusports.com/index.html
```

The URL for our example driver and database looks like this:

```
jdbc:mysql://db_server:3306/contacts/
```

Even though these two URLs look different, they are actually the same in form: the protocol for connection, machine host name and optional port number, and the relative path of the resource. Your JDBC driver will come with instructions detailing how to form the URL for your database. It will look similar to our example.

You will want to control access to your data, unless security is not an issue. The standard least common denominator for authentication to a database is a pair of strings, an account and a password. The account name and password you give the driver should have meaning within your DBMS, where permissions

should have been established to govern access privileges.

Our example JDBC driver uses an object of the `Properties` class to pass information through the `DriverManager`, which yields a `Connection` object:

```
Properties props = new Properties();
props.setProperty("user", "contacts");
props.setProperty("password", "blackbook");
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/contacts/", props);
```

Now that we have a `Connection` object, we can easily pass commands through it to the database, taking advantage of the abstraction layers provided by JDBC.

Structuring statements

Databases are composed of tables, which in turn are composed of rows. Each database table has a set of rows that define what data types are in each record. Records are also stored as rows of the database table with one row per record. We use the data source connection created in the last section to execute a command to the database.

We write commands to be executed by the DBMS on a database using SQL. The syntax of a SQL statement, or query, usually consists of an action keyword, a target table name, and some parameters. For example:

```
INSERT INTO songs VALUES (
   "Jesus Jones", "Right Here, Right Now");
INSERT INTO songs VALUES (
   "Def Leppard", "Hysteria");
```

These SQL queries each added a row of data to table "songs" in the database. Naturally, the order of the values being inserted into the table must match the order of the corresponding columns of the table, and the data types of the new values must match the data types of the corresponding columns. For more information about the supported data types in your DBMS, consult your reference material.

To execute an SQL statement using a `Connection` object, you first need to create a `Statement` object, which will execute the query contained in a `String`.

```
Statement stmt = con.createStatement();
String query = ... // define query
stmt.executeQuery(query);
```

Example: Parsing a text file into a database table

In the course of modernizing a record keeping system, you encounter a flat file of data that was created long before the rise of the modern relational database. Rather than type all the data from the flat file into the DBMS, you may want to create a program that reads in the text file, inserting each row into a database table, which has been created to model the original flat file structure.

In this case, we examine a very simple text file. There are only a few rows and columns, but the principle here can be applied and scaled to larger problems. There are only a few steps:

- Open a connection to the database.
- Loop until the end of the file:
  - Read a line of text from the flat file.
  - Parse the line of text into the columns of the table.
  - Execute a SQL statement to insert the record.

Here is the code of the example program:

```
import java.io.*;
import java.sql.*;
import java.util.*;

public class TextToDatabaseTable {
   private static final String DB = "contacts",
                               TABLE_NAME = "records",
                               HOST = "jdbc:mysql://db_lhost:3306/",
                               ACCOUNT = "account",
                               PASSWORD = "nevermind",
                               DRIVER = "org.gjt.mm.mysql.Driver",
                               FILENAME = "records.txt";

   public static void main (String[] args) {
      try {

         // connect to db
         Properties props = new Properties();
         props.setProperty("user", ACCOUNT);
         props.setProperty("password", PASSWORD);

         Class.forName(DRIVER).newInstance();
         Connection con = DriverManager.getConnection(
```

```
            HOST + DB, props);
         Statement stmt = con.createStatement();

         // open text file
         BufferedReader in = new BufferedReader(
                                 new FileReader(FILENAME));

         // read and parse a line
         String line = in.readLine();
         while(line != null) {

            StringTokenizer tk = new StringTokenizer(line);
            String first = tk.nextToken(),
                   last = tk.nextToken(),
                   email = tk.nextToken(),
                   phone = tk.nextToken();

            // execute SQL insert statement
            String query = "INSERT INTO " + TABLE_NAME;
            query += " VALUES(" + quote(first) + ", ";
            query += quote(last) + ", ";
            query += quote(email) + ", ";
            query += quote(phone) + ");";
            stmt.executeQuery(query);

            // prepare to process next line
            line = in.readLine();
         }
         in.close();
      }

      catch( Exception e) {
         e.printStackTrace();
      }
   }

   // protect data with quotes
   private static String quote(String include) {
      return("\"" + include + "\"");
   }
}
```

Processing a result set

Perhaps even more often than inserting data, you will want to retrieve existing information from your database and use it in your Java program. The usual way to implement this is with another type of SQL query, which selects a set of rows and columns from your database and appears very much like a table. The rows and columns of your result set will be a subset of the tables you queried, where certain fields match your parameters. For example:

```
SELECT title FROM songs WHERE artist="Def Leppard";
```

This query returns:

title
Hysteria

The boxed portion above is a sample result set from a particular database program. In a Java program, this SQL statement can be executed in the same way as in the insert example, but additionally, we must capture the results in a `ResultSet` object.

```
Statement stmt = con.createStatement();
String query = "SELECT FROM junk;"; // define query
ResultSet answers = stmt.executeQuery(query);
```

The JDBC version of a query result set has a cursor that initially points to the row just before the first row. To advance the cursor, use the `next()` method. If you know the names of the columns from your result set, you can refer to them by name. You can also refer to the columns by number, starting with 1. Usually you will want to get access all of the rows of your result set, using a loop as in the following code segment:

```
while(answers.next()) {
   String name = answers.getString("name");
   int number = answers.getInt("number");
   // do something interesting
}
```

All database tables have **meta data** that describe the names and data types of each column; result sets are the same way. You can use the ResultSetMetaData class to get the column count and the names of the columns, like so:

```
ResultSetMetaData meta = answers.getMetaData();
String[] colNames = new String[meta.getColumnCount()];
for (int col = 0; col < colNames.length; col++)
   colNames[col] = meta.getColumnName(col + 1);
```

Example: Printing a database table

We choose to write a simple software tool to show the rows and columns of a database table.

In this case, we are going to query a database table for all its records, and display the result set to the command line. We could also have created a graphical front end made of Java Swing components.

Notice that we do not know anything except the URL and authentication information to the database table we are going to display. Everything else is determined from the `ResultSet` and its meta data.

Comments in the code explain the actions of the program. Here is the code of the example program:

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

public class DatabaseTableViewer {
   private static final String DB = "contacts",
                               TABLE_NAME = "records",
                               HOST = "jdbc:mysql://db_host:3306/",
                               ACCOUNT = "account",
                               PASSWORD = "nevermind",
                               DRIVER = "org.gjt.mm.mysql.Driver";

   public static void main (String[] args) {
      try {

         // authentication properties
         Properties props = new Properties();
         props.setProperty("user", ACCOUNT);
         props.setProperty("password", PASSWORD);

         // load driver and prepare to access
                Class.forName(DRIVER).newInstance();
         Connection con = DriverManager.getConnection(
            HOST + DB, props);
         Statement stmt = con.createStatement();
```

```
      // execute select query
      String query = "SELECT * FROM " + TABLE_NAME + ";";
      ResultSet table = stmt.executeQuery(query);

      // determine properties of table
      ResultSetMetaData meta = table.getMetaData();
      String[] colNames = new String[meta.getColumnCount()];
      Vector[] cells = new Vector[colNames.length];
      for( int col = 0; col < colNames.length; col++) {
         colNames[col] = meta.getColumnName(col + 1);
         cells[col] = new Vector();
      }

      // hold data from result set
      while(table.next()) {
         for(int col = 0; col < colNames.length; col++) {
            Object cell = table.getObject(colNames[col]);
            cells[col].add(cell);
         }
      }

      // print column headings
      for(int col = 0; col < colNames.length; col++)
         System.out.print(colNames[col].toUpperCase() + "\t");
      System.out.println();

      // print data row-wise
      while(!cells[0].isEmpty()) {
         for(int col = 0; col < colNames.length; col++)
            System.out.print(cells[col].remove(0).toString()
                             + "\t");
         System.out.println();
      }
   }

   // exit more gently
   catch(Exception e) {
      e.printStackTrace();
   }
  }
}
```

Conclusions

In this article, you saw a quick introduction to manipulating databases with JDBC. More advanced features of JDBC require a greater knowledge of databases. See the references for more articles about JDBC and its applications [4]. As a Java programmer, JDBC is a good tool to have in your arsenal.

I encourage you to copy the code in this article to your own computer. With this article and documentation for another JDBC driver, you are on your way to creating data source-driven Java programs. Experiment with this code, and adapt it to connect to data sources available to you.

References

1 ComputerWorld. Java Database Connectivity by Carol Sliwa -- http://www.computerworld.com/cwi/story/0,1199,NAV47-68-85-98_STO43545,00.html

2 Hoffman. SQL tutorial -- http://w3.one.net/~jhoffman/sqltut.htm

3 Sun Microsystems. JDBC Data Access API -- http://java.sun.com/products/jdbc/index.html

4 Sun Microsystems. JDBC Data Access API: Articles and Success Stories -- http://java.sun.com/products/jdbc/articles.html

5 Sun Microsystems. JDBC Data Access API: Drivers -- http://industry.java.sun.com/products/jdbc/drivers

---