Making Databases Easier for Your Users

James W. Cooper

In an early column , I showed how you can create a couple of useful classes called Database and Results, which simplify using JDBC methods. They allow you to connect to databases and get the results of queries. You can use classes like these in any system you develop, and the code will be the easier to read for it. This month, I want to extend that work and show convenient ways to build query objects and simple classes that can allow your users to create his own database tables.

We start by redefining our database classes. Our base class will be the AbstractDatabase class which has the following methods. We show the constructor in detail. The remaining methods are available in the sample code.

```
public class AbstractDatabase {
    protected Connection con;  //connection saved here
    public AbstractDatabase(String driver) throws
                               ClassNotFoundException {
        types = new String[1];
        types[0] = "TABLES";         //initialize type array
        Class.forName(driver);
    }
    //-----------------------------------
    public void Open(String url, String cat) throws SQLException {
        catalog = cat;
        con = DriverManager.getConnection(url);
        dma =con.getMetaData();   //get the meta data
    }
    //-----------------------------------
    public void Open(String strURL, String strUser,
           String strPassword , String strCatalog) throws SQLException
    public String[] getTableNames() throws SQLException
    public String[] getTableMetaData() throws SQLException
    public String[] getColumnMetaData(String tablename)
                        throws SQLException
    public String[] getColumnNames(String table) throws SQLException
    public String getColumnValue(String table, String columnName)
                        throws SQLException
    public String getNextValue(String columnName) throws SQLException
    public void setAutoCommit(boolean a) throws SQLException {
    public void commit() throws SQLException {
    public void rollback() throws SQLException {
}
```

The reaseon we start with a class we call AbstractDatabase is that there may be some details that vary for each specific database we connect to. For example, for DB2, the method getTableNames returns only the tables which do not begin with “SYS” since these are internal tables of no interest to the user.

Our Results class is a somewhat simpler version of the ResultSet class which has somewhat more convenient methods. We only show the constructor in detail here.

```
public class Results {
```

```
    private ResultSet rs;
    private ResultSetMetaData rsmd;
    private Statement stmt;
    private int numCols;
    private boolean opened;
//Creates an easier to use class based on the
//java.sql.ResultSet class
    public Results(ResultSet rset) throws SQLException {
        rs = rset;
        opened = false;
     //get the meta data and column count at once
        rsmd = rs.getMetaData();
        opened= true;
        numCols = rsmd.getColumnCount();
     }
    //---------------------------------
    public void close() throws SQLException
    public String[] getMetaData() throws SQLException {
    public boolean hasMoreElements() throws SQLException  {
    public String[] getRow() throws SQLException {
    public String getColumnValue(String columnName) {
    public String getColumnValue(int i)  {
}
```

Queries

Now we can make up an SqlQuery class that actually issues the queries and returns our Results class. This allows us to deal exclusively with these higher level classes.

```
//base class for defining SQL query.
public class SqlQuery {
    protected String sql;
    protected Connection con;
    protected Statement stmt;
    // constructor
    public SqlQuery(String sqlquery, AbstractDatabase db)
                        throws SQLException {
        sql = sqlquery;
        con=  db.getConnection ();
        stmt = con.createStatement();
    }
    //Sets a new query string to reuse exisiting object
    public void setQueryString(String qry) {
        sql = qry;
    }
    //gets the SQL string back
     public String getQueryString() {
        return sql;
    }
    // executes an SQL query that returns results
    public Results Execute() throws SQLException {
        Results rset = new Results(stmt.executeQuery (sql));
        rset.setStatement(stmt); //copy in so it can be closed
        return rset;
    }
    // executes an SQL statement that does not return results
    public int executeUpdate() throws SQLException {
```

```
        int count = stmt.executeUpdate (sql);
        stmt.close ();
        return count;
    }
    //Closes the SQL statement
    //You must close the statement when you close the Results object
    public void close() throws SQLException {
        stmt.close();
    }
}
```

With these simple classes, you can connect to a database and execute queries in just a few lines of code. Here is the code for connecting to the database:

```
    public ShowGroceries(String dbName)
      throws SQLException, ClassNotFoundException{
        super("Database display");
        db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
        db.Open("jdbc:odbc:Grocery prices", null);
        setGUI();
    }
```

and here is the code to execute a simple query and display it:

```
String sql="Select FoodName from Food order by  FoodName";
            try {
                SqlQuery qry = new SqlQuery(sql, db);
                Results rs = qry.Execute ();

                while (rs.hasMoreElements ()) {
                    jlist.add (rs.getColumnValue (1));
                }
            }
            catch (SQLException e) {
                System.out.println("SQL error:"+e.getMessage());
            }
```

The display of our basic ShowGroceries application is show in Figure 1

Figure 1 – The ShowGroceries application. The list box is filled when you click on the button marked “Groceries.”

It is important that you close the underlying statement object whenever you are done with that particular query. If you don’t you will soon run out of database resources. Since the statement object starts in our SqlQuery class, but generates results in our Results class,

we include a setStatement method in the Results class so you can close the statement and the ResultSet objects at the same time:

```
//closes the current Results and statement object
    public void close() throws SQLException  {
        if (rs != null)
            rs.close();    //close the results
        if (stmt != null)
            stmt.close();  //and close the statement
        opened = false;
    }
```

Building a More Flexible Application

Now, let's suppose that you have constructed a complete application system built on these classes for querying which stores have which groceries at which prices. Then you proudly hand the application to a friend who needs something quite similar. However, your friend quickly finds that while your application is good for what it does, it does not completely fill her needs, because she needs to tabulate the vitamin content of each food item for a special study. So, while she needs all you have written, including your databse design, she also needs to add tables for vitamin content to the database schema you have designed. Further, she gets the data from various surveys and needs to able to write code to add those data to the database herself.

The question, then, is how can you preserve your database design and still provide a way for a user to add and load her own tables and generate her own queries.

Our solution is to create a generic DbTable class where your user can create a table and define the fields and create them at a high level. The SQL statements we need to generate has the form

```
CREATE TABLE Vitamins
      (VITAKEY INTEGER NOT NULL,
      VITANAME VARCHAR(50) NOT NULL,
      PRIMARY KEY(Vitakey));
```

and

```
CREATE UNIQUE INDEX
      INX_Vitaname_Vitamins
      ON Vitamins(Vitaname);
```

We design the high level methods we want to implement as the following simple code:

```
//create table object
        DbTable vitamins = new DbTable("Vitamins", db);
        vitamins.delete ();  //remove any prior table

 //add the fields
        vitamins.addField ("Vitakey");
        vitamins.addField ("Vitaname", 50);
        vitamins.setPrimaryKey ("Vitakey");
        vitamins.createTable ();

 //and the index
        vitamins.addIndex ("Vitaname", true);
```

Thus, the createTable method will generate the SQL statement CREATE TABLE we showed above. All the DbTable class needs to do is to keep track of the field names and types as we add them and then use those to generate the create statement.

We start with an abstract class which holds the name of the field and its length and has an abstract method to generate the SQL string needed to create the table.

```
//a small class holding info about
//fields to be created by DBTable
abstract class FieldName {
    private String name;
    private int length;

    public FieldName(String nm) {
        name = nm.toUpperCase ();
        length = 4;    //default
    }
    public void setLength(int len) {
        length = len;
    }
    public int getLength() {
        return length;
    }
    public String getName() {
        return name;
    }
    public abstract String getSqlString();
}
```

Then, we create subclasses for String and integer fields which implement the getSqlString method. Here is the integer version

```
class IntField extends FieldName {
    public IntField(String name) {
        super(name);
    }
    public String getSqlString() {
        return getName()+" INTEGER NOT NULL";
    }
}
```

and here the string version:

```
class StringField extends FieldName {
    public StringField(String name, int size) {
        super(name);
        setLength(size);
    }
    public String getSqlString() {
        return getName()+" VARCHAR(" +
            new Integer(getLength()).toString () +
            ") NOT NULL";
    }
}
```

Note that we did not declare any of these classes as public, since they are only used within the package containing our DbTable class. Here we use two simple methods to generate our create table string:

```
//Creates the table using the field names you have added
    public void createTable() throws SQLException {
        if (! exists() ) {
            String sql = "CREATE TABLE " +
                  tableName +" (" ;
            sql += getFieldString();
            if (primaryKey.length () > 0) {
                sql += ", PRIMARY KEY(" +
                  primaryKey +")";
            }
            if(tableSpace.length() > 0)
                sql += ") IN " + tableSpace+ ";";
            else
                sql += ");";
            SqlQuery qry = new SqlQuery(sql, db);
            System.out.println(sql);
            qry.executeUpdate ();
            tableCreated = true;
        }
    }
    //---------------------------------
    //generates the field string from the field class
    //and follows it with a comma unless last
    private String getFieldString() {
        String sql = "";
        for (int i=0; i< tblNames.size (); i++) {
            FieldName fld = (FieldName)tblNames.elementAt (i);
            sql+= fld.getSqlString ();
            if (i < (tblNames.size()-1))
                sql+=", ";
        }
        return sql;
    }
```

Finally, for each indexed field we use the following method to create an index

```
//Adds in indexed column to the table.
    public void addIndex(String indexField, boolean unique)
            throws SQLException {
        String sql = "CREATE ";
        if (unique)
            sql+=" UNIQUE ";
        sql+= "INDEX INX_"+indexField+"_"+tableName;
        sql+= " ON " + tableName+"("+indexField+");";
        SqlQuery qry = new SqlQuery(sql, db);
        System.out.println(sql);
        qry.executeUpdate();
    }
```

Adding Rows to Our Table

Once we have created a new table, we need a way to add data to it from our program without having to understand too much of the internal workings of whatever database we are using. The easiest way to add rows to a table is to add all of the values of the fields and then use a PreparedStatement to do the adding. Prepared statements are SQL statements that are defined in advance with variables in them, and which can be

precompiled by the database to gain better performance. The SQL syntax for a prepared INSERT statement is

```
INSERT INTO Vitamins (VITAKEY, VITANAME)VALUES (?, ?);
```

where the values for the name and key are added when the prepared statement is executed. We can simplify how we use prepared statements by creating a PreparedQuery class:

```
import java.sql.*;
//A prepared statement query type. Precompiles the query
public class PreparedQuery extends SqlQuery {
    private PreparedStatement prep;

    //Creates the prepared query.
    public PreparedQuery(String sql,
            AbstractDatabase db) throws SQLException {
        super(sql, db);
        prep = con.prepareStatement(sql);
    }
    //---------------------------------------
    //Sets one of the query parameters
    //from a String.
   public void setString(int index, String s)
                  throws SQLException     {
        prep.setString(index, s);
    }
//---------------------------------------
    //Sets one of the query parameters
    //from an integer.
    public void setInteger(int index, int s)
                  throws SQLException  {
        prep.setInt(index, s);
    }
//---------------------------------------
    //Executes the prepared query
    //with the current parameters
    public Results Execute() throws SQLException {
        Results rs = new Results(prep.executeQuery ());
        return rs;
    }
//---------------------------------------
    //Executes the prepared query statement
    //with the current parameters
    public int executeUpdate() throws SQLException {
        return prep.executeUpdate();
    }
//---------------------------------------
    public void close() throws SQLException {
        prep.close();
   }
}
```

Then we add an openTable method to our DbTable class which creates the prepared statement after the table is complete:

```
public void openTable() throws SQLException {
        String query="INSERT INTO " +tableName+" (" ;
        query += getNames() +")";
```

```
        query += "VALUES (";
        for (int i=0; i< tblNames.size (); i++) {
            query += "?";
            if (i < (tblNames.size()-1))
                query +=", ";
        }
        query +=");";
        System.out.println(query);
        prep = new PreparedQuery(query, db);

    }
```

and we can add rows as follows, where the indexes are the order that we added the original columns in.

```
  private void addRow() throws SQLException {
        vitamins.openTable();
        vitamins.setPreparedTerm(1, 1);     //Vitakey
        vitamins.setPreparedTerm("B12", 2); //Vitaname
        vitamins.addRow();
    }
```

Conclusions

In this column, I've shown how you can build query and table classes that can make it easy for you and your users to manipulate databases at a high level. I've tried this code on both Microsoft Access using the odbc-jdbc bridge driver and on IBM DB2 6.1. The only difference that DB2 imposes is that you have to define the name of a table space to create the tables in. This is often called "USERSPACE1" and you can make this a default value by subclassing DbTable. If there is a user space name it is added to the CREATE TABLE SQL statement.

These classes are more advanced examples of the Façade design pattern, where we enclose a relative complex series of classes in a simpler framework. And these classes certainly do make handling databases a lot easier.