HANDLING DATABASES MORE EFFECTIVELY

James W. Cooper

In our day-to-day work, we sometimes needed to store complicated relations between data in relational databases. Databases are used more than any other kind of structure in computing. You'll find them in payroll and employee records, in travel reservation systems and all through product manufacturing and marketing.

A database is a series of tables of information in some sort of file structure that allows you access these tables, select columns from them, sort them and select rows based on various criteria. Some columns in most database tables have indexes associated with them so they can be accessed as rapidly as possible. For example, suppose we decided to compile a database of food prices for some things we buy regularly at 3 local markets. We might just make a list of food items, their prices and the store where we found that price. But, if we wanted to sort these prices by item or look at items by store to reduce our shopping time, we might find it better to keep the food in one table, the stores in another table and the prices at each store for each item in another table. We see the food and store tables in Figure 1.

1	Apples
2	Oranges
3	Hamburger
4	Butter
5	Milk
6	Cola
7	Green beans

1	Stop and Shop
2	Village Market
3	Waldbaum's

Figure 1 - The food and store tables from our groceries database.

Rather than keeping the prices with the stores or with the food, we make a third table containing the price, the store key and the food item key. Being able to represent relations between row of different tables using keys is characteristic of *relational databases*, as shown in Figure 2.

Relkey	StoreKey	FoodKey	Price
1	1	1	0.27
2	2	1	0.29
3	3	1	0.33
4	1	2	0.36

Figure 2. Part of a longer relational table, showing the prices of each food in each store.

There is also a common language that all databases support for making queries about these tables, called Structured Query Language, or SQL. Finally, nearly all database vendors provide access to their databases through a common API called ODBC or Open Database Connectivity.

Databases in Java

Now, in Java we have a set of classes for connecting to databases using an interface similar to ODBC, called JDBC. You can connect to any database for which the manufacturer has provided a JDBC connection class. These JDBC classes are available for almost every database on the market. Some databases have direct connections using JDBC and a few allow connection to ODBC driver using the JDBC-ODBC bridge class. Then, you can pass standard SQL queries to the database without even knowing which one it is. (Of course there are some differences in SQL dialects between databases, so this is not quite as simple as we'd like.)

However, these database classes in the java.sql package provide an excellent example of a set of low level classes that interact in a convoluted manner, as shown in Figure 2.

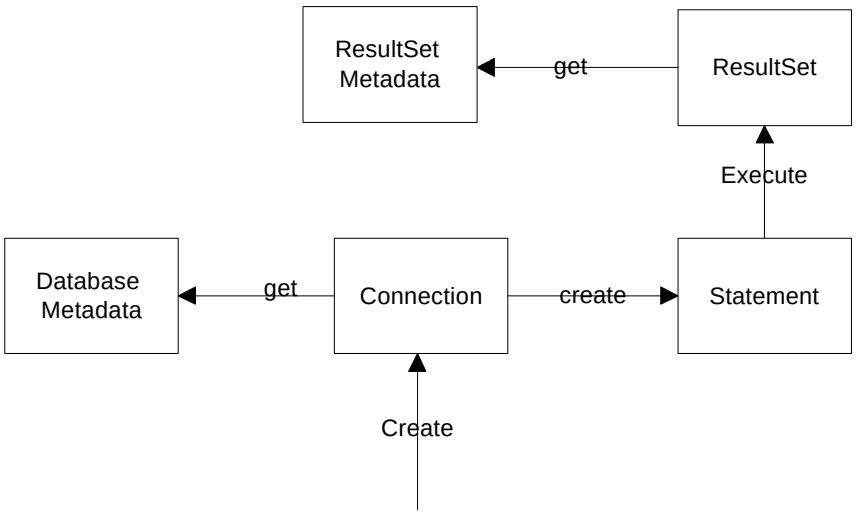

Figure 3. - The main JDBC classes.

To connect to a database, you use an instance of the Connection class. Then to find out the names of the database tables and fields, you need to get an instance of the DatabaseMetadata class from the Connection. Then, to issue a query, you compose the SQL query string and use the Connection to create a Statement class. You execute the statement, obtaining a ResultSet class, and to find out the names of the column rows in the ResultSet, you need to obtain an instance of the ResultsetMetadata class. Thus, it can be quite difficult to juggle all of these classes and since most of the calls to their methods throw Exceptions, the coding can be messy at least.

Fortunately, we can wrap the complexity of the JDBC's low level interface in some higher-level classes to make JDBC easier to use. Wrapping complex classes to provide a

simpler interface is a description of the Façade Design Pattern. This pattern allows you to simplify this complexity by providing a simplified interface to subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all but the most sophisticated users. These users can still, of course, access the underlying classes and methods.

Building the Façade Classes

Let's consider how we connect to a database. We first must load the database driver:

```
try{Class.forName(driver);}  //load the Bridge driver
    catch (Exception e)
    {System.out.println(e.getMessage());}
```

and then use the Connection class to connect to a database. We also obtain the database metadata for use in finding out more about the database:

```
try {con = DriverManager.getConnection(url);
    dma =con.getMetaData();   //get the meta data
    }
    catch (Exception e)
    {System.out.println(e.getMessage());}
```

Then if we want to list the names of the tables in the database, we need to call the *getTables* method on the database metadata class, which returns a ResultSet object. Then we have to iterate through that object to get the list of names, making sure that we obtain only user table names, and excluding internal system tables.

```
Vector tname = new Vector();
try {
      results = new resultSet(dma.getTables(catalog,
                        null, "%", types));
    }
    catch (Exception e) {System.out.println(e);}
    while (results.hasMoreElements())
        tname.addElement(
            results.getColumnValue("TABLE_NAME"));
```

This quickly becomes quite complex to manage, and we haven't even issued any queries yet.

One simplifying assumption we can make is that the exceptions that all these database class methods throw do not need complex exception handling. For the most part, the methods will work without error unless the network connection to the database fails. Thus, we can wrap all of these methods in classes where we simply print out the infrequent errors and take no further action.

This makes it possible to write two simple enclosing classes that contain all of the significant methods of the Connection, ResultSet, Statement and Metadata classes. These are the Database class:

```
Class Database {
 public Database(String driver)()   //constructor
 public void Open(String url, String cat);
 public String[] getTableNames();
 public String[] getColumnNames(String table);
 public String getColumnValue(String table,
```

```
            String columnName);
   public String getNextValue(String columnName);
   public resultSet Execute(String sql);
}
```

and the resultSet class:

```
class resultSet
{
   public resultSet(ResultSet rset) //constructor
   public String[] getMetaData();
   public boolean hasMoreElements();
   public String[] nextElement();
   public String getColumnValue(String columnName);
   public String getColumnValue(int i);
}
```

These 2 classes constitutes a Façade and are illustrated in Figure 4.

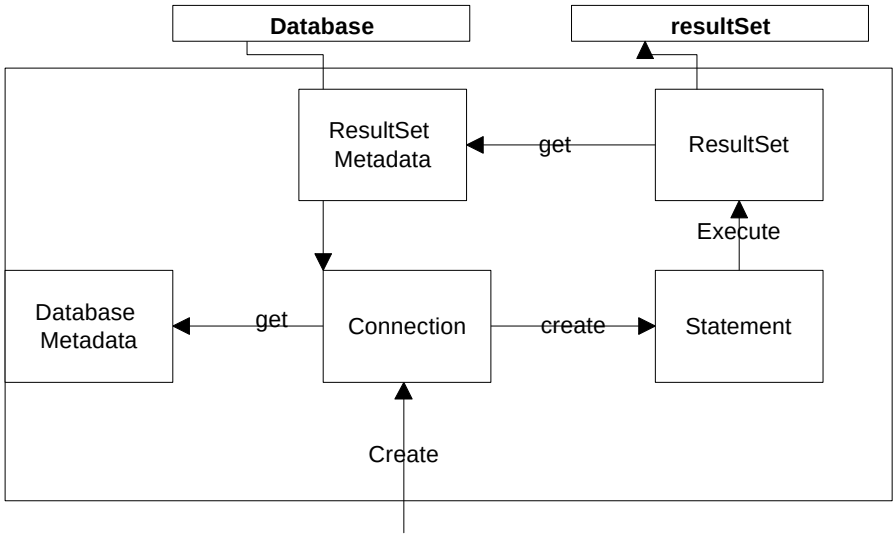

Figure 4 - A Façade which encloses the database classes in a simpler interface.

These simple classes allow us to write a program for opening a database, displaying its table names, column names and contents, and running a simple SQL query on the database.

The dbFrame.java program accesses a simple database containing food prices at 3 local markets. It is shown in Figure 5.

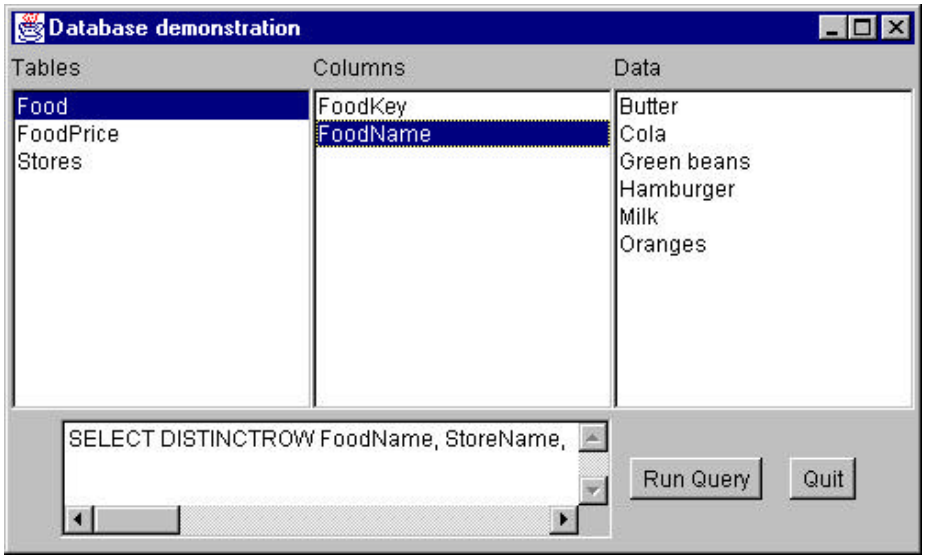

Figure 5. The dbFrame program.

Clicking on a table name shows you the column names and clicking on a column name shows you the contents of that column. If you click on Run Query, it displays the sorted food prices by store for oranges, as shown in Figure 6.

Figure 6- The results of a simple SQL query, sorting orange prices by store.

This program starts by connecting to the database and getting a list of the table names:

```
db = new Database("sun.jdbc.odbc.JdbcOdbcDriver");
db.Open("jdbc:odbc:Grocery prices", null);
String tnames[] = db.getTableNames();
loadList(Tables, tnames);
```

Then clicking on one of the lists causes a simple query for table column names or contents:

```
public void itemStateChanged(ItemEvent e)    {
//get list box selection
      Object obj = e.getSource();
      if (obj == Tables)
         showColumns();
      if (obj == Columns)
         showData();
}
```

```
//-----------------------------------
private void showColumns()  {
//display column names
String cnames[] =
      db.getColumnNames(Tables.getSelectedItem());
loadList(Columns, cnames);
}
//-----------------------------------
private void showData() {
//display column contents
String colname = Columns.getSelectedItem();
String colval =
      db.getColumnValue(Tables.getSelectedItem(),
            colname);
Data.removeAll();       //clear list box
colval = db.getNextValue(Columns.getSelectedItem());

while (colval.length()>0)   {
      //load list box
   Data.add(colval);
   colval = db.getNextValue(Columns.getSelectedItem());
  }
}
```

A Summary of the Façade

We've developed a Façade pattern, which shields clients from complex database subsystem components and provides a simpler programming interface for the general user. However, it does not prevent the advanced user from going to the deeper, more complex classes, when they find that necessary. In addition, we see that the Façade allows you to make changes in these underlying subsystems without requiring changes in the client code, and reduces compilation dependencies. You'll find the sample code, including an Access database at this magazine's web site. You will also need the JDBC-ODBC bridge from javasoft.sun.com, and may need the ODBC interface from Microsoft's site. Search for wx1350.exe. Needless to say, this example is based on a PC database, and won't run on other platforms.

References

1. George Reese, *Database Programming with JDBC and Java*, O'Reilly, 1997

2. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley: Reading, MA, 1995.

3. J. W. Cooper, *Principals of Object-Oriented Programming Using Java 1.1*, Coriolis/Ventana, 1997.

James W. Cooper is at work on his 13th book, Java Design Patterns, for Addison Wesley.