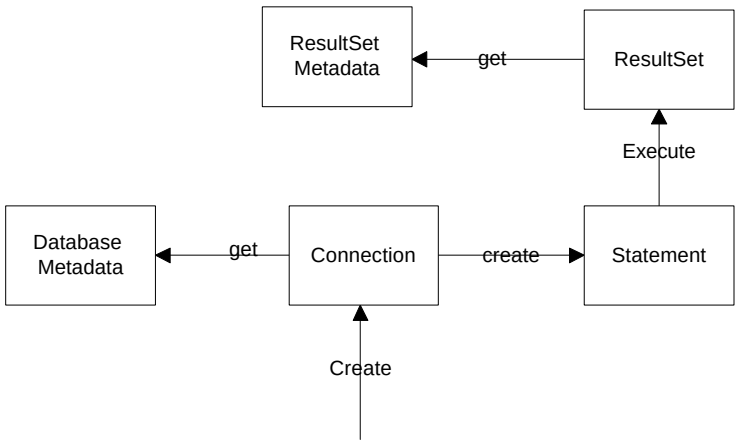
THE FAÇADE PATTERN

Frequently, as your programs evolve and develop, they grow in complexity. In fact, for all the excitement about using design patterns, these patterns sometimes generate so many classes that it is difficult to understand the program's flow. Furthermore, there may be a number of complicated subsystems, each of which has its own complex interface.

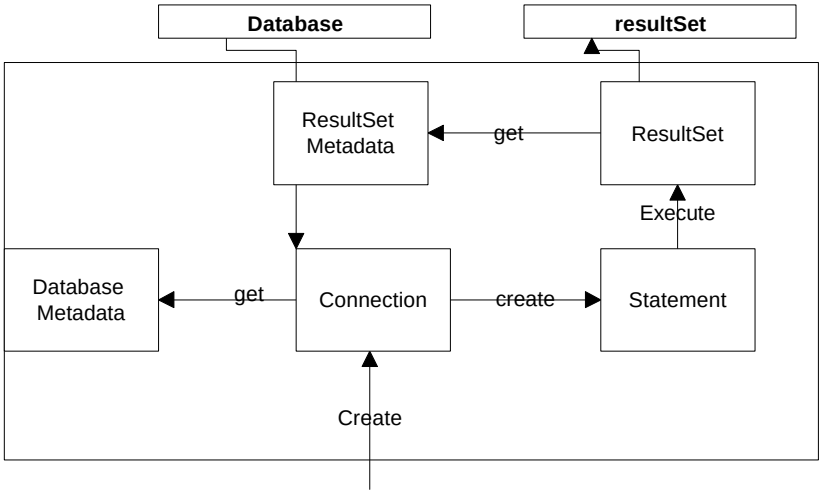
The Façade pattern allows you to simplify this complexity by providing a simplified interface to these subsystems. This simplification may in some cases reduce the flexibility of the underlying classes, but usually provides all the function needed for all but the most sophisticated users. These users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

Fortunately, we don't have to write a complex system to provide an example of where a Facade can be useful. Java provides a set of classes that connect to databases using an interface called JDBC. You can connect to any database for which the manufacturer has provided a JDBC connection class -- almost every database on he market. Some databases have direct connections using JDBC and a few allow connection to ODBC driver using the JDBC-ODBC bridge class.

These database classes in the java.sql package provide an excellent example of a set of quite low level classes that interact in a convoluted manner, as shown below.



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. Next, to issue a query, you compose the SQL query string and use the Connection to create a Statement class. By executing the statement, you obtain a ResultSet class, and to find out the names of the column rows in that 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.



resultSet class (note the lowercase "r"), we can build a much more usable system.

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 to find 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());}
```

If we want to list the names of the tables in the database, we then need to call the *getTables* method on the database metadata class, which returns a ResultSet object. Finally, to get the list of names we have to iterate through that object, making sure that we obtain only user table names, and exclude 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 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 in which we simply print out the infrequent errors and take no further action.

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

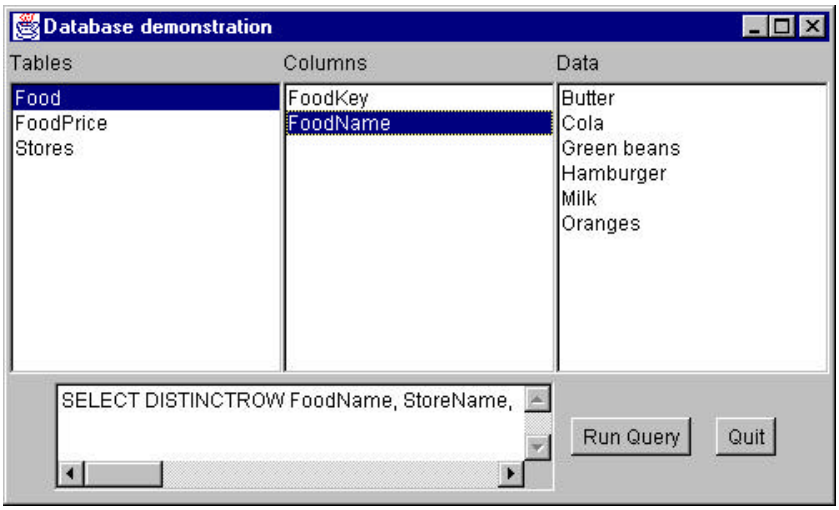
```
Class Database {
 public Database(String driver)()//constructor
 public void Open(String url, String cat);
 public String[] getTableNames();
 public String[] getColumnNames(String table);
 public String getColumnValue(String table,
             String columnName);
   public String getNextValue(String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



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 food prices sorted by store for oranges:

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 runs 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());
  }
}
```

Consequences of the Façade

The Façade pattern shields clients from complex 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 necessary.

In addition, the Façade allows you to make changes in the underlying subsystems without requiring changes in the client code, and reduces compilation dependencies.