Reflections on Java, Beans, and relational databases

Here's how to get to your relational database with the new 1.1 APIs

Summary
The 1.1 version of the Java Development Kit (JDK) contains new APIs for database access (JDBC) and components (JavaBeans). Together, these two APIs allow for the development of generic database code. A single class to access any JDBC database -- with the code specific to each individual application isolated in separate components. This means you won't have to update any database code with each minor change in a database's structure. (3,200 words)

By Sandor Spruit

A relational database basically consists of a set of interconnected tables, where each table stores one sort of data relevant to the application. An address book database might, for example, have tables for people, addresses, phone numbers, and so on. Each of these entities would be stored in the database as series of strings, integers, and other primitive types. The definition of the database tables would describe how the relevant information on each type of entity is to be stored in a table's columns. You could, for example, have a "person" table with columns meant to contain strings to store "first name" and "last name." Each table should have one or more columns containing a value to uniquely identify each row. These identifiers or "keys" are used to connect the information in different tables. You could, for example, assign a unique "person_number" to each person in the "person" table and use the same numbers in an extra column of the "address" table. You could then link people to addresses by matching values in both "person_number" columns.

Relational database systems have been around since the 1970s, and nowadays they are the predominant way to store large amounts of data. Hence, Java software tools are needed to access the databases built using these systems.

There are, basically, two problems that need to be solved before a relational database can be used from within a Java application. First, you need some basic middleware to establish the connection with the database, send SQL queries to it, and so forth. Second, you would like to manipulate the results just as you do any other piece of information in Java -- as objects. The former already has been solved by Sun and several database vendors; the latter is left for us to work on.

Sun has been cooperating with many software companies to define a large number of APIs for common programming tasks. The API for Java database connectivity (JDBC) was among the first JDK 1.1 APIs to stabilize, and there are numerous implementations of it available from various sources. Some of these are 100 percent pure Java. Others use a mixture of Java and native code to connect to, for example, existing ODBC data sources (see Figure 1). The JavaSoft people have put an extensive overview of available JDBC drivers on their Web site at http://splash.javasoft.com/jdbc/jdbc.drivers.html.
The pros and cons of each of these implementations obviously depend heavily on your particular set-up and environment, and therefore I will not discuss each of them individually. There are just too many drivers from too many different vendors to get all this information into one article. Furthermore, most of the Java files available for downloading on the 'Net contain detailed release notes describing the installation procedures. Therefore, I will assume throughout the rest of this article that you have at least some Java development environment up and running, and that you have successfully installed and tested either a JDBC driver or a driver plus Sun's JDBC/ODBC bridge. In other words: I will assume you are ready to develop some JDBC-based software.

The JDBC API

The JDBC API comes as a single Java package (java.sql) containing a series of classes. These classes provide just the type of middleware needed to handle a relational database. Basically, they allow you to establish a connection to a database and send queries. You can process the results of these queries, retrieve meta-information on your database, and handle the exceptions that might occur in the midst of all this.

A simple JDBC example

Let us, just briefly, look at a how a simple example query can be performed using Java JDBC classes. The extremely small database in Table 1 suffices to demonstrate the basic principles. The code in Listing 1 is about the smallest possible piece of Java code that allows you to execute a single SQL query against a relational database.

<table>
<thead>
<tr>
<th>Person#</th>
<th>First name</th>
<th>Last name</th>
</tr>
</thead>
<tbody>
<tr>
<td>43674</td>
<td>Sandor</td>
<td>Spruit</td>
</tr>
<tr>
<td>90329</td>
<td>John</td>
<td>Doe</td>
</tr>
<tr>
<td>65435</td>
<td>Donald</td>
<td>Duck</td>
</tr>
</tbody>
</table>

Table 1: A simple "Sample" database with three rows and three columns from a "Person" table

```java
String url = "jdbc:odbc:sample";
String query = "SELECT * FROM PERSON";
boolean more;
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con = DriverManager.getConnection(url,"sandor","guest");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (more = rs.next())
    {
        int number = rs.getInt("PERSON#");
        String firstName = rs.getString("FIRST_NAME");
        String lastName = rs.getString("LAST_NAME");
        System.out.println(number + " " + firstName + " " + lastName);
    }
}
rs.close();
```
If you have ever written some Java yourself, you'll see that this code is fairly straightforward. It loads Sun's JDBC/ODBC driver, establishes a connection to the database pointed to by jdbc:odbc:sample, and executes the simple SELECT query against it. Unless an SQLException is caught, it loops through the ResultSet to extract the rows from the database one at a time and display the results on the screen.

Now, what is wrong with this code? I mean, all programmers with an urge to produce elegant code will feel very uncomfortable writing this sort of code. I think there are two things that are fundamentally wrong with the sort of code presented in Listing 1.

- This code can only be written using a considerable amount of database meta-information, and this information is all hard-coded into the program. If you want to fetch a number, for example, you have to know in advance whether you will actually receive an integer, a float, or a double. It is difficult -- if not impossible -- to write a class that can handle any database; each minor adjustment to the database forces you to carefully check and modify the code.
- The database information is always delivered as a single RecordSet instance, which is not a real object. The RecordSet class -- not unlike other database class wrappers -- is more like a pointer or a cursor, providing access to database information via methods. The instances of RecordSet do not actually contain information, they merely represent the means to get to it. This is exactly why you have to work your way through the RecordSet -- advancing a cursor using ResultSet.next() -- while calling other RecordSet methods to get to some real information. In fact, the JDBC classes do only deliver such sets of loosely coupled fields. These are of little use, even if you know all the inner details of the database's structure, because in Java, real objects provide the predominant way to store and process information.

Thus, ideally you would have to find some elegant way to extract the records and fields from a database one by one (via a RecordSet) and stuff the information you get into fresh objects.

The primary key (pun intended) to such a solution is the similarity between a relational and an object-oriented model of a certain data set. The definitions of classes and tables partially serve similar purposes, while objects and records share some properties too. You can readily think of records as sets of values to initialize an object's data members. However, you need to call a class constructor in order to create the fresh objects mentioned above, once you have fetched an arbitrary record from an arbitrary table. You could easily create objects from records, if only you could automatically deliver each record to the right constructor.

When you are developing a small application, it is possible to deliver each record to some constructor that creates an object from it. You can always use an Object reference to manipulate any value extracted from the database, because any object you obtain using a RecordSet is an extended java.lang.Object in the end. It may be possible to
define one BigClass implementing the features you would otherwise put into separate classes. You can use references to an Object to manipulate any value extracted from the database, because any object you obtain using a RecordSet is an extended java.lang.Object in the end. The BigClass implementation could use the Java instanceof operator to determine the runtime type of the database information and a huge switch to jump to the right code.

You could also define a similar BigClass with several constructors, each having a slightly different signature. You could use BigClass(int,int), BigClass(int,float), and so on, depending on the sort of information you get while looping through your RecordSet. (This implementation would undoubtedly include a lot of nasty casting.) However, neither of these suggestions would really solve anything, because the relations between the records and constructors would still have to be hard-coded into your program. For generic database code to work with anything but toy applications, we obviously need some different way to automatically connect tables and constructors.

This is where a nice Java feature comes to our rescue. The code fragment in Listing 2 creates a new instance of any Java class, given nothing but a class name. This means we can use names of classes and tables to find out which constructor can handle a record fetched from a table. It is easy to obtain a complete list of table names from a database using standard JDBC classes, so we can put this little Java trick to good use! You simply develop a Java class per database table, where the class name matches the table name. Whenever your program fetches a record from a table, it creates an object by passing the table name to Class.forName(). It's that easy!

```java
Class c = Class.forName("Person");
Person p = (Person)c.newInstance();
System.out.println("... just created a " + c.getName());
```

Listing 2: A simple Class.forName() example

However, there is a complication. The forName() method will call the void constructor for the given class, so we can not pass our RecordSet variable directly to the constructor. We need an initialization method with a ResultSet parameter that extracts one record from it, and uses it to assign the values to the object’s data members. It may be wise to introduce a superclass as the common ancestor to all the classes linked to a table, because they all need it. In fact, this class plays a key role in querying the database, as I will demonstrate shortly.

**Querying the database**

It is all very well to create objects from records, but you still need SQL statements to query a database, right? You can not construct these statements without having some insight into the database structure, so it seems we are back where we started. The SQL statements would still have to be hand-coded, even if we can automatically match table and class names. This means you would still have to manually edit these statements every time you update your database structure. Notice, however, that we can overcome this hurdle using the approach outlined above once more. After all, there is little you need to know to query a database table. You will normally query a database table using names and values for those fields that are part of either the primary key or an index. In other words, you can extract records (or objects) from a table if someone supplies you with suitable values for all the right fields. A quick look at the JDBC specification reveals
that DatabaseMetaData objects can be used just to retrieve lists of table names (see above) but also to obtain lists of primary key and index fields. Similar problem, similar solution.

A relational database can be queried using a relatively small piece of code, by feeding it a series of proper (name, value) pairs. You can match all the names in the pairs with the field names of the primary keys and the indices. Whenever you find a complete primary key or an index in the list of names, you can use the corresponding values to build an SQL statement, execute it to get a RecordSet, and turn it into objects via the Class.forName() construct. Easy.

This idea requires instances of any class associated with a database table to have methods that allow access to its data members as (name, value) pairs, but such methods can be implemented perfectly well by the common ancestor we introduced in the previous section. Listings 3 and 4 present this method in pseudo-code.

Open the database connection
Retrieve a list of user defined tables
For each table
{
    Check whether there is a corresponding class file
    if (it is available)
    {
        load the class file
        Retrieve lists of key fields and indices for this table
        Store these lists in hashtables for easy access
    }
    else throw an exception
}

Listing 3: Pseudocode to initialize a database connection

Take an object A containing a series of (name, value) pairs
For each table T
{
    For each (name, value) pair
    {
        if (name matches primary_key_field or index_field)
            store a reference to both name and value
    }
    if all key_fields were found
        create a query string using key names and values
    else if all index_fields were found
        create a query string using index names and values
    execute the query to obtain a ResultSet
    For each record in the ResultSet
    {
        Create an object of the class associated with table T
        Initialize the object using the record's contents
        Add the object to the results, e.g., attach it to A
    }
}

Listing 4: Pseudocode describing a database query
Okay, now that we can relate database table to classes and create objects from records, let us shift into fifth gear. We have only used a tiny Java feature so far. What happens if we bring in the heavy artillery? Do I hear the trumpets announcing the arrival of the cavalry?

**The sophisticated approach: Java reflection and JavaBeans**

The introduction of the Java 1.1 Development Kit (JDK) brought us many powerful new features to play with, such as new user interface classes. Two of the new JDK 1.1 APIs in particular deserve our special attention: the reflection facility (the java.lang.reflect package) and the JavaBeans component API (the java.beans package). These two APIs will help us build a sophisticated database class, because they allow us to work with meta-information about classes. They are the "heavy artillery" that can help solve the problems in developing a generic database class.

The class `Class`, with its `forName()` and `newInstance()` methods, is just a simple example of the power of reflection. The point is that `forName()`'s String parameter is not necessarily some character sequence appearing in your source code. You can load and instantiate any class, given its name as any character sequence taken from anywhere. In the case of our database class, we can derive the class name directly from the table name obtained from the database itself. This means the name of the Java class associated with a database table does not have to appear anywhere in our source code. Consequently, you won't have to update your source code when a table's name changes or when a table is added. You'll just have to make sure a class with the new name is available on your system.

The idea of using meta-information about classes can be exploited much further. The reflection classes are meant to obtain, store, and process information about the classes in your Java code at run time. Their instances can be used like any object in Java, allowing you to tinker with classes, types, return types, method references, and arguments as easily as with strings and integers. This reflection concept may seem quite useless at first -- because you have immediate access to all the information you need on classes, methods, and parameters in your own code. You are, after all, the one who wrote the source code in the first place. So what is new? The answer is that reflection will work on the compiled code in Java class files, not at the source code level. It doesn't matter whether you have access to source files, because you can obtain useful information directly from compiled classes. You can use reflection to use other people's code from within your own applications, because you can examine aspects of the inner workings of their compiled code! The JavaBeans API is, in fact, meant to do just that: allow for the construction of applications using classes from completely different programmers and vendors.

The JavaBeans specification prescribes a set of conventions for the names of class members, to make sure the name of the methods systematically describe their function. Any Java class that adheres to the rules can be interrogated by a Bean Introspector instance (using reflection) to reveal important aspects of its behavior -- like the sort of events the class will respond to and the events it may generate. Any class that adheres to these rules is effectively a Bean, and thus a component. In theory, this means you can collect a series of Beans from various sources and bind them together at run time, when they appear to fire and consume the same sort of events.

You may be wondering by now what all of this has to do with querying databases. There is another series of naming conventions -- apart from those on event handling -- that apply to the data members of a class. These conventions are crucial to our database
endeavors. Let us look at an example Bean to illustrate how Java features like reflection and Beans can serve our purposes.

**An example Bean**

The example Translation Bean (see below) has one constructor and two methods to manipulate a single property named "language." The point is, you can check out the code of a class to learn about its constructors, methods, and properties, and so can the Bean Introspector!

```java
public class Translation extends Object
{
    int language;

    public Translation()
    {
    }

    public int getLanguage()
    {
        return (language);
    }

    public void setLanguage(int language)
    {
        this.language = language;
    }
}
```

*Listing 5: A very simple example Bean*

A Bean Introspector can supply arrays of PropertyDescriptor instances containing type information for any Bean's properties, defined by the very presence of the sort of get/set methods shown in the example. You can invoke the methods too -- using reflection -- to read or write these properties.

The reflection facilities provide us with better means to check the integrity of an otherwise loose connection between classes and database tables. The fact that some class is available with a name matching the name of a table does not guarantee that its internals are okay. A class associated with a table obviously should have members to have its instances store all the table's columns. A class may have the right name, but the initialization code could be missing, for all we know. It may have the right name but have members with either the wrong name or the wrong type. Why not use the combination of JDBC's DatabaseMetaData and the reflection facilities to check whether they all match!

Need a recipe?

Invoke some JDBC calls to obtain all the necessary details on the nature of the database information, check your system for classes with the right names, and use reflection to compare the properties of the tables and classes. It's a piece of cake really! ;)}
A conclusion and a hint

The combination of JDBC, reflection, and JavaBeans makes it relatively easy to fetch records from a relational database and use them to initialize complete components -- not just objects. In fact, you won't have to modify your database for this to work. You'll only have to make sure the classes adhere to the Bean specification, and that their properties match those of tables. To give two hints of other neat tricks that can make life even more fun, Beans can bring their own user interface components, and the Bean specification includes things called Customizers. You can introduce extra classes specifically meant to view, edit, or customize an instance of a Bean class. (For more on customizing Beans, see Mark Johnson's current JavaBeans column, "Double Shot, Half Decaf, Skinny Latte": Customize your Java.)

So, why not write Customizer classes for our database classes? Your application could fetch stuff from your relational database, obtain a fresh instance by instantiating the right class, and bring in the associated GUI components or Customizers to view or edit the data. All done using generic code that could handle any database. A versatile database viewer/editor written in Java is right around the corner!

Note: This article is all about the retrieval of information. I'm sure you can figure out the storage part yourself. ;)

About the author

Sandor Spruit has a BS in Electrical Engineering from the Hogeschool Utrecht (1992) and an MSc. in Computer Science from the University of Twente (1995). He is currently working at Utrecht University in the Department of Computer & Humanities, developing tools to create educational software (in general) and a flexible bilingual dictionary for educational environments (in particular).