JDBC provides a powerful, comprehensive interface for accessing databases from Java programs. For smaller projects, JDBC can seem overwhelming, driving some programmers to avoid using a database altogether. This article describes a simple wrapper library that makes basic database usage a snap. You'll find yourself wanting to use JDBC for every program you write.

It happens suddenly. You're hacking away on what you think is a little program, only to find that it has snowballed into something larger -- a project -- and now you've reached the point where you need to save some data.

The problem is, you hadn't intended for your program to get this far. It was just a little hack that you got carried away with. You didn't really make any provisions for saving or loading. What's more, your program is an applet, and applets can't save and load.

Or can they?

In fact, the JDBC API allows any Java program -- even an applet -- to connect to a relational database (RDBMS). Unfortunately, JDBC might be a little top-heavy for your little program. After all, an RDBMS is a powerful, complicated system when all you really want to do is save a little data.

In this article, we'll examine a simple abstraction layer which takes the bite out of using JDBC. For simple applications, it will let you save and load structured data in just a few lines of code. It won't handle sophisticated use of an RDBMS, but then, that's what we'd like to avoid, isn't it?

**The complexity of JDBC**

JDBC can be a complicated API to use. Not only must it support the entirety of the powerful SQL standard, but it must cleanly hide the variances between different database engines.

JDBC is also complicated because relational databases are built around SQL, and SQL was meant to be used by humans, not programs. Using JDBC directly is a little bit like programming in two different languages at the same time.

While these aspects of JDBC are not bad things, they do tend to conflict with our goal: quickly storing a small amount of data.

**A simplifying abstraction**

We will create a simple abstraction layer that lets you use JDBC without having to get into all the messy details. If you are already familiar with JDBC or relational databases, a first glance at our class list should be familiar:

- Database
- Table
- RowSet
- Row
We're not doing anything radical here; our data model is essentially the same as the relational model, but with annoying details (as well as powerful features) removed. Each class maps onto a basic RDBMS concept as well as a JDBC class. It is this mapping that allows our API to preserve its relational quality while still being much simpler to use.

The design of this API is based on assumptions about what our data-storage needs are. If you find that your program needs something a little different, feel free to change this abstraction to suit your situation. These classes should be thought of as a schema for simplifying your work, not as something set in stone.

If you aren't familiar with SQL or RDBMS technology, have no fear. Each of the next four sections will acquaint you with one of our classes, as well as the RDBMS feature onto which it maps.

The Database class

When using JDBC to make a connection to a database, you have to tell JDBC where to find the actual data. Because different database engines have different methods of being accessed and different syntaxes for describing these methods, there isn't a single way to specify a data source. In JDBC, a Uniform Resource Identifier (URI) string is used to specify a data source, and the construction of this string is database dependent.

The main purpose of the Database class is to encapsulate this string, as well as any username/password information that might be needed to make a connection.

Here's how you create a Database object:

```java
Database db =
   new Database("jdbc:postgresql://localhost/mito",
               "mito", "" );
```

The first argument to the constructor is the URI of the data source. In this case, I am using the PostgreSQL database engine, and I am accessing a database called mito on the local machine. Additionally, I am specifying my username, mito, and an empty password as the second and third arguments, respectively.

Once you have created a Database object, you can use it to access data, as we'll see in the next section.

The Table class

One of the simplifying assumptions made in our API is that when you read data from a row of a table, you get the entire row. Put another way, a row of a table is an atomic unit that is read and written as a single piece. This isn't especially efficient, but efficiency is not a primary concern in our approach.

The Table class allows you to read and write these row objects. The first thing you have to do is create a table object, which is as simple as knowing its name:

```java
Table table = db.getTable( "employee" );
```

The act of creating a Table object doesn't really do anything, except to let the object remember its name. To do something, we need to actually use the Table object. Here, we read a single row from a table:

```java
Row row = table.getRow( "id=101" );
```

Note that we've specified that we only want the row that has the value of the 'id' field set to '101'. By using the `getRow()` method, we've assumed that there is only a single row that meets this criterion. In other cases, we might expect multiple rows, in which case we want to use `getRows()`, as follows:

```java
RowSet rows = table.getRows( "id<103" );
```
In this case, the return value is a RowSet, rather than a Row. A RowSet is just a set of rows.

In the next two sections, we'll discuss the Row and RowSet classes.

The Row class

In our abstraction, a Row is a set of name/value pairs that represents a row of a table in the RDBMS. While RDBMS values can be of different types, a Row contains only one, the string type. This, again, is done to make things simpler -- we are assuming that you don't need any of the more powerful features provided by string typing.

Once you have a Row, it's easy to pull a value out of it, as we see in Listing 1.

```java
Row e = table.getRow( "id=100" );
String name = e.get( "name" );
System.out.println( "Employee name: "+name );
```

Note also that a Row is ordered, so that you can pull out name/value pairs by index. An example of this is given in Listing 2.

```java
Row e = table.getRow( "id=100" );
for (int i=0; i<e.length(); ++i) {
    String key = e.getKey( i );
    String value = e.get( i );
    System.out.println( key+" = "+value );
}
```

Of course, you can also change the value in a Row. This is necessary for altering the data in a database -- something we'll take a look at later on.

The RowSet class

Remember that some queries can return multiple Rows, in which case you'll get a RowSet. A RowSet is little more than a wrapper around a Vector. You can easily iterate through all the Rows in a RowSet, as seen in Listing 3.

```java
RowSet rs = table.get( "id<104" );
for (int i=0; i<rs.length(); ++i) {
    Row row = rs.get( i );
    // do something with the row....
}
```

A full example

Now that we've gone over each of the classes, let's take a look at a full example. In Listing 4, we're going to extract a set of records that meet a certain criterion, and print out their values.

```java
// First, get all rows meeting the criterion
RowSet rs = table.getRows( "id<103" );
// Iterate through the set
for (int i=0; i<rs.length(); ++i) {
    // Grab each row in turn
    Row row = rs.get( i );
    // Get and print the value of the "name" field
    String name = row.get( "name" );
    System.out.println( "Name: "+name );
}
```
Easy! In the next section, we'll see how to write data to the database.

**Modifying data**

As mentioned earlier, data that is read and written using our API comes and goes in entire rows. In order to write to the database, you have to create (or modify) a Row object, and then write that Row object to the database.

Writing to the database is done using the `putRow` method of the `Table` class. There are two variations on this method:

- `public void putRow( Row row )`
- `public void putRow( Row row, String conditions )`

These two variations correspond to the **INSERT** and **UPDATE** commands in SQL.

In the first variation, writing a row means inserting a brand new row into the table.

In the second variation, writing a row means modifying an existing row. The `conditions` parameter allows you to specify which row (or rows) you want to modify.

Let's look at an example of each of these techniques.

**Inserting a new row**

Inserting a new row is simple, because you do not need to specify a row or rows to be modified. You simply insert the row:

```
table.putRow( row );
```

You can create a Row from scratch, as we see in Listing 5.

```
// Create an empty row object
Row row = new Row();
// Fill it up with data
row.put( "id", "200" );
row.put( "name", "Joey Capellino" );
```

Or you can modify an existing row that had been read previously from the database, as in Listing 6.

```
// Grab a row from the database
Row row = table.getRow( someConditions );
// Change some or all of the fields
row.put( "name", "Joey Capellino" );
```

While it is common to create a Row from scratch when inserting and to use an existing Row when updating, you can actually do it any way you like.

**Updating an existing row**

As mentioned in the previous section, there is no restriction on how you create the Row that you use for updating. However, generally, you use a Row that you've just read from the database.
To describe this in detail, we'll use an example where we read the name of an employee, change the name, and write the change back to the database, as in Listing 7.

```java
Row row = table.getRow( "id=104" );
row.put( "name", newName );
table.putRow( row, "id=104" );
```

Note that we had to specify a condition in the call to `putRow()`. This is what makes the call an `update` rather than an `insert`.

Note also that this call will update all rows that match the criteria, not just one of them.

## Conclusion

In this article, we've taken a look at an API that provides a simplified interface to relational databases via the JDBC package. This abstraction retains many of the basic relational features of the JDBC interface, but simplifies them in such a way that makes it very easy to use. This simplification comes at the cost of efficiency, but that is not a surprising result when the goal is simplicity.

## Resources

- Download the full source for the classes discussed in this article:
  - [Database.java](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/)
  - [Table.java](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/)
  - [RowSet.java](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/)
  - [Row.java](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/)

- The [JDBC Learning Center](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/) contains links to a number of tutorials.

- Read the [JDBC Documentation Page](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/) for links to lots of useful documentation.


- The code in this article was tested against the [PostgreSQL](http://www-128.ibm.com/developerworks/java/library/j-jdbcwrap/) database engine.

- Josh Heidebrecht walks us through the latest JDBC revisions in "What's new in JDBC 3.0" (developerWorks, July 2001).

- Learn how to invoke legacy stored procedures through Enterprise JavaBeans in "Wrapping DBMS stored procedures as methods in session EJB components" (developerWorks, October 2000)

- Find out how the divide between object-oriented and data-oriented developers came to be in "The object-data divide and EJB" (developerWorks, April 2001). Then, read the follow-up story, "Overcoming the object-data divide on your EJB project" (developerWorks, April 2001) to find out
what you can do about it.


- Find more Java resources on the developerWorks Java technology zone.

About the author

Greg Travis is a freelance programmer living in New York City. His interest in computers can probably be traced back to that episode of "The Bionic Woman" where Jamie runs around trying to escape a building whose lights and doors are controlled by an evil artificial intelligence that mocks her through loudspeakers. He's a devout believer in the religious idea that, when a computer program works, it's a complete coincidence. He can be reached at mito@panix.com.