

# *Notes from Well House Consultants*

*These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit*

*<http://www.wellho.net/net/whcotnl.html>  
for details.*

## 1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

## 1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

## 1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

## 1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

[graham@wellho.net](mailto:graham@wellho.net)

technical contact

[lisa@wellho.net](mailto:lisa@wellho.net)

administration contact

Our full postal address is

404 The Spa  
Melksham  
Wiltshire  
UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

# *JDBC – Relational Database Access*

Java accesses relational databases, such as Oracle and MySQL, through JDBC classes. A manager class which oversees the process is provided with the Java distribution, but you'll also need to obtain a driver class to talk to the specific database of your choice. Using MySQL as an example, this module takes you through the sourcing, configuration and use of JDBC so that you can access data in a relational database from within your Java application.

*Prerequisites:* In order to make the most of this module, trainees need prior knowledge of Java to an intermediate level and a little prior knowledge of SQL.

<i>Interfacing MySQL to Java</i> . . . . .	297
<i>Using JDBC to access other databases</i> . . . . .	298
<i>Using JDBC on the Web</i> . . . . .	300

As from Java 1.1, connectivity to relational databases has been provided through JDBC (Java Database Connectivity) in the `java.sql` package. Note that the Java distribution does not include a full relational database package, nor even the Java classes to access any particular database, which you will need to obtain and install in addition to your J2SE in order to have a Java front-ended relational database.

JDBC drivers are not necessarily available for all databases.

- They are available from various vendors for commercial databases such as Access and Oracle.
- Free or inexpensive databases such as PostgreSQL have limited drivers available at present.

#### Basic Techniques:

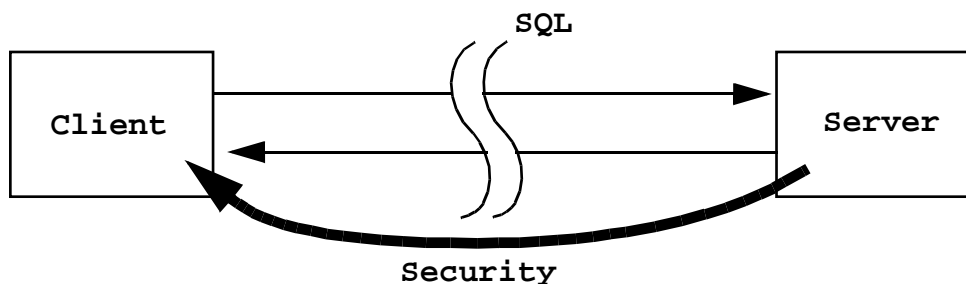
- Register the database driver
- Obtain a `Connection` object
- Use a `Statement` object to send a statement
- Retrieve the results in a `ResultSet` object

These basic techniques (and the `java.sql` package as a whole) are mostly interfaces rather than classes.

The `DriverManager` class (a true class, this one!) is responsible for managing all the classes available and providing the appropriate one for you. The `Class.forName` method will register with the `DriverManager` for you.

`DriverManager.getConnection` will obtain the connection object for you; we use a URL of the form:

```
jdbc:subprotocol://host:port/database
```



In a web environment, you'll note that the JDBC connection is made from the client using a TCP service and sockets. It follows that the specific driver class needs to be available to the browser and the security aspects of distributing that driver need to be considered.

The `getConnection()` method takes a user name and password as parameters. You should consider carefully where you obtain and store this information.

`createStatement()` does exactly that – it creates an object with which you can submit queries using either the `executeQuery()` method or for an update the `executeUpdate()` method.

Queries are formed and sent in SQL (Structured Query Language).

Finally, the `getResultSet()` method retrieves an object of type `ResultSet`. As ever, `get` methods allow you to collect data row-by-row or column-by-column.

As well as querying and updating data, facilities are available to learn about the structure of the database.

This is known as `MetaData`; the `DatabaseMetaData` interface allows you to retrieve this information using the `getMetaData` method of the `Connection` class.

Most database accesses will be either to obtain data or to update data, but you can go so far as using `SQL CREATE TABLE` statements to add tables and structure to your database (i.e. you have both the DDL and the DML elements available).

Although most queries are handled as one-offs, each query being committed to the

database as it is sent, you are able to send queries in sections, for example, atomic transactions which are combined when you `commit()`.

### 26.1 Interfacing MySQL to Java

Java interfaces to databases through JDBC (Java Database Connectivity). Your Java Runtime Environment should include the `java.sql` package by default, which is basically a driver manager class and does not include any drivers for any specific databases. You then source appropriate drivers from elsewhere; there's a list of suitable drivers on Sun's site:

<http://industry.java.sun.com/products/jdbc/drivers>

which currently lists almost 200 drivers, including several for MySQL.

In addition, you could interface MySQL to Java using the JDBC to ODBC bridge but ... don't; you'll be adding in an extra layer of conversions, and the drivers listed on the site above are all "type 4" drivers which means that they're written in native Java code.

Here is a complete working example, using drivers sourced via this web site.

```
public class jdbc1 {

public static void main(String [] args) {

    java.sql.Connection conn = null;

    System.out.println("SQL Test");

    try {
        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
        conn = java.sql.DriverManager.getConnection(

"jdbc:mysql://bhajee/test?user=jttest&password=");

    }
    catch (Exception e) {
        System.out.println(e);
        System.exit(0);
    }

    System.out.println("Connection established");

    try {
        java.sql.Statement s = conn.createStatement();
        java.sql.ResultSet r = s.executeQuery
        ("SELECT code, latitude, longitude FROM dist");
        while(r.next()) {
            System.out.println (
                r.getString("code") + " " +
                r.getString("latitude") + " " +
                r.getString("longitude") );
        }
    }
    catch (Exception e) {
        System.out.println(e);
        System.exit(0);
    }
}
}
```

And in operation:

```
$ java jdbc1
SQL Test
Connection established
java.sql.SQLException: General error: Table 'test.dist' doesn't exist
$ sql2.pl populate abc123
Table populated from text file
$ java jdbc1
SQL Test
Connection established
AB 3 57.8
AL 0.4 51.7
B 2 52.5
BA 2.4 51.4
BB 2.6 53.8
BD 1.9 53.8
BH 2.9 50.8
(etc)
Portugal -8.5 37
Denmark 14 55.5
Germany 9 50
Norway 10.5 59.8
Sweden 18 59
Austria 18 47
Italy 14 42
Italia 14 42
$
```

## 26.2 Using JDBC to access other databases

We've chosen MySQL as our example database for this section since we run it on our training machines, but just to show you how portable the code is, here's an example that uses the Oracle database:

```
package mypackage1;
import java.sql.*;
import java.util.*;

public class Bruce09
{
    public Bruce09()
    {
    }

    public static void main(String[] args)
    {
        Connection conn = null;
        System.out.println("SQL Test");
        try {
            DriverManager.registerDriver
            (new oracle.jdbc.driver.OracleDriver());
        }
        catch (Exception e) {
            System.out.println(e);
            System.exit(0);
        }
    }
}
```

```

try {
conn = java.sql.DriverManager.getConnection(
"jdbc:oracle:thin:username/password@machine.us.company.com:1234:dbSID");
}
catch (Exception e) {
System.out.println(e);
System.exit(0);
}

System.out.println("Connection established");

try {Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery
("Select display_name from kbi_periods");
while(r.next()) {
System.out.println (
r.getString("display_name") );

StringTokenizer Tok = new StringTokenizer(r.getString("display_name"), "-");
int n=0;
while (Tok.hasMoreElements())
System.out.println(" " + ++n + " : " +Tok.nextElement());
}
}
catch (Exception e) {
System.out.println(e);
System.exit(0);
}
}
}
}

```

The example extracts the **display\_name** field from the **kbi\_periods** table, and manipulates the data using utility classes such as the string tokenizer. Here's the data description to help you read the example in context:

```
SQL> desc kbi_periods
```

Name	Null?	Type
PERIOD_TYPE		VARCHAR2(10)
PERIOD		VARCHAR2(8)
PERIOD_START		DATE
PERIOD_END		DATE
DISPLAY_NAME		VARCHAR2(12)

```
SQL>
```

```
SQL> select * from kbi_periods;
```

PERIOD_TYP	PERIOD	PERIOD_ST	PERIOD_EN	DISPLAY_NAME
MONTH	2002-04	01-APR-02	30-APR-02	APR-02
MONTH	2002-05	01-MAY-02	31-MAY-02	MAY-02
MONTH	2002-06	01-JUN-02	30-JUN-02	JUN-02
MONTH	2002-07	01-JUL-02	31-JUL-02	JUL-02

MONTH	2002-08	01-AUG-02	31-AUG-02	AUG-02
MONTH	2002-09	01-SEP-02	30-SEP-02	SEP-02
MONTH	2002-10	01-OCT-02	31-OCT-02	OCT-02
MONTH	2002-11	01-NOV-02	30-NOV-02	NOV-02
MONTH	2002-12	01-DEC-02	31-DEC-02	DEC-02
MONTH	2003-01	01-JAN-03	31-JAN-03	JAN-03
MONTH	2003-02	01-FEB-03	28-FEB-03	FEB-03
MONTH	2003-03	01-MAR-03	31-MAR-03	MAR-03

12 rows selected.

SQL>

### 26.3 Using JDBC on the Web

These days, most users want to access data (including data held in their relational databases) from a browser, and Java provides you with tools to facilitate this. For example:

- Data is held in a relational database (such as MySQL), which is managed by an appropriate database engine (such as mysqld).
- A Java Class, using JDBC drivers, interfaces to the database engine allowing data to be transferred between that class and the database
- The Java class is then referenced by another class which extend the Servlet interface
- The Servlet is accessed by Tomcat, which provides the JRE conforming to the Servlet specification
- The Apache Web Server uses Tomcat to provide the Servlet capability via an Apache module interface
- The user who wants to access data in the database enters his request on a form on his browser, and when he submits the form data is accessed ...

Browser -> Apache Server -> Tomcat -> Servlet Class -> JDBC class -> Database Engine -> Database and then returned to the browser

Database -> Database Engine -> JDBC class -> Servlet Class -> Tomcat -> Apache Server -> Browser.

In order for a scheme such as this to work, you need to have a number of resources other than just your regular Java compiler available:

- A Web Server
- Java Servlet capability for the web server
- (Your Java environment, but we'll assume that as this is a Java course)
- Javax classes for handling Servlets and databases, from the Enterprise edition
- Appropriate JDBC driver classes
- A relational database, such as MySQL
- You will also need to have knowledge of (or access to) skills in:
  - HTML
  - Java
  - SQL

This may seem to be a daunting list of topics to learn (as indeed it is if you're new to all of them), but once you have acquired and tuned those skills, and configured and installed the appropriate software, you can write powerful applications quickly and efficiently.

We'll go on and look at two examples that use the mechanism and technologies we've just described to manage single database tables; each shows you a framework structure that you're free to learn from and develop to suit your own needs.

*Example: Managing a table of information about a population*

This example was written as a demonstration for a client who is involved in medical research. His organisation needs to keep records which includes peoples' names, dates of birth, and ethnic background amongst other information, so that the data can later be analysed and extracted for trend, etc.

With an application such as this one, there are usually a number of tables and many fields involved, and the experienced programmer will write general code to avoid much repetition; such code is, however, often hard to follow as you're learning, so this is very much a training example, and you'll find that the code would be long-winded to expand into a complete system [the following example shows you a solution to this problem].

The first file that we'll look at is a piece of pure HTML, and it includes the forms that the user starts with. To avoid too much code fragmentation, a single HTML page has forms for both inserting and searching for data from our population table:

```
<html>
<head>
<title>Servlet to JDBC demo - population for Biomedical research</title>
</head>
<body bgcolor=white>
<h1>Servlet to JDBC demo<BR>Biomedical research - table about a population
</h1>
Please complete any of the forms on this page to run the demonstration
programs that allow data entry and extraction for members of a population
who are included in a sample in use for biomedical research
<HR>
<H2>Enter new data</h2>
<form action="http://hawk:8080/nan/servlet/J850insert"><table border=1>
<tr><td>Full name of person</td><td><input name=fullname></td></tr>
<tr><td>Ethnic Origin</td><td><select name=origin>
<option value=Caucasian>Caucasian</option>
<option value=Chinese>Chinese</option>
<option value="West African">West African</option>
<option value=Iranian>Iranian</option>
<option value=Indian>Indian</option>
<option value=Other>Other</option>
</select><br>or enter: <input name=special></td></tr>
<tr><td>Date of Birth</td><td><input name=day size=2 value=16> /
/ <input name=month size=2 value=07> /
<input name=year size=4 value=1972></td></tr>
<tr><td>then ...</td><td><input type=submit value="Enter this data">
</td></tr></table>
</form>
<HR>
<H2>Select data</h2>
<form action="http://hawk:8080/nan/servlet/J850select"><table border=1>
<tr><td>Name</td><td><input name=name></td></tr>
<tr><td>Ethnic Origin</td><td><select name=origin>
<option value=Any>Any</option>
<option value=Caucasian>Caucasian</option>
<option value=Chinese>Chinese</option>
<option value="West African">West African</option>
<option value=Iranian>Iranian</option>
<option value=Indian>Indian</option>
</select><br>or enter: <input name=special></td></tr>
<tr><td>Year of Birth</td><td><input name=year size=4></td></tr>
```

```
<tr><td>then ...</td><td><input type=submit value="Search for this data">
</td></tr></table></form><hr>
</body>
</html>
```

The forms provided both refer to Servlets:

**J850insert**

and

**J850select**

J850insert includes input fields names full name, origin, special, day, month and year into which the data entry clerk can enter information about a new member being added to the population. J850select includes fewer fields, being the fields on which data can be selected.

When the user completes and submits the form, the data is added to the table – we'll look at the code that does this in a minute – and he gets a short report back:

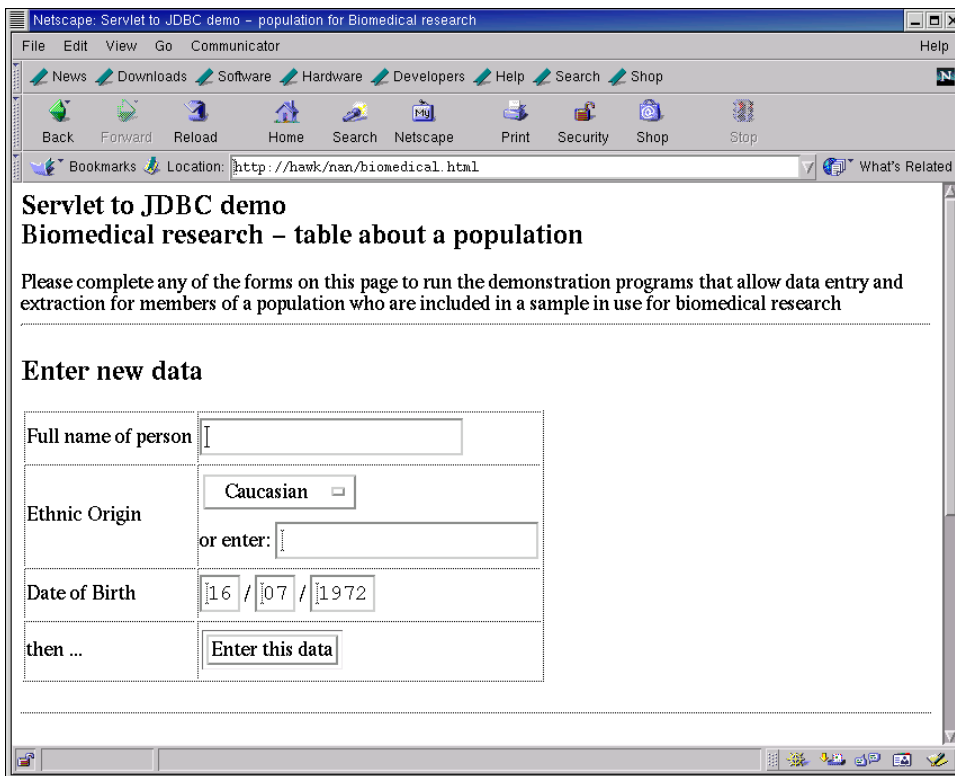


Figure 104 What the data entry will appear like on the browser.

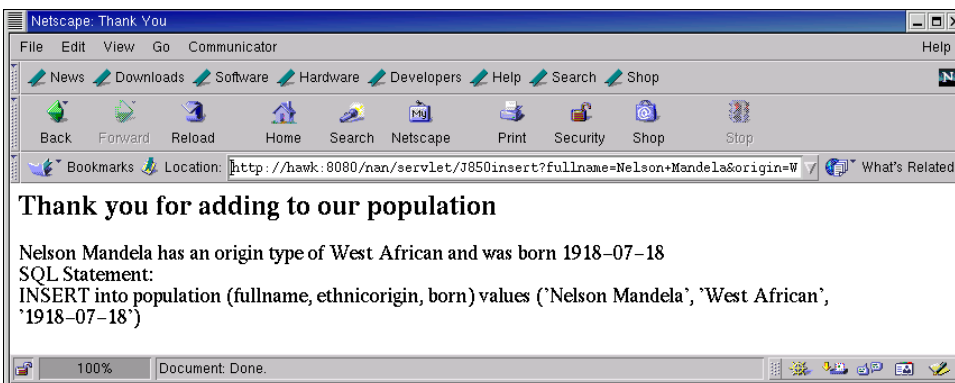


Figure 105 The resulting page after the form is filled in.

The J850 insert script is as follows:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/*
 * Inserting record into a population table
 */

public class J850insert extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {

        String called = request.getParameter("fullname");

        String origin = request.getParameter("special");
        if (origin.equals("")) {
            origin = request.getParameter("origin");
        }

        String day = request.getParameter("day");
        String month = request.getParameter("month");
        String year = request.getParameter("year");
        String born = year + "-" + month + "-" + day;

        // Now do the real work ....

        String SQLcommand = J850_jdbc.insert(called,origin,born);

        // After the work is done, send the reply out ....

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        out.println("<title>Thank You</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<body>");

        out.println("<h1>Thank you for adding to our population</h1>");
        out.println(called + " has an origin type of " + origin +
            " and was born " + born+"<BR>");
        out.println ("SQL Statement:<BR>"+SQLcommand);
        out.println("</body>");
        out.println("</html>");
    }
}
```

which is a fairly regular Java Servlet. Just note the call to the static insert method in

the J850\_jdbc class, to which the Servlet has passed the variable fields extracted from the form. The J850\_jdbc class itself is as follows:

```
public class J850_jdbc {

public static String insert(String name, String group, String born) {

    java.sql.Connection conn = linktodata();

    String todo = ("INSERT into population " +
        "(fullname, ethnicorigin, born) "+
        "values ('"+name+"', '"+group+"', '"+born+"')");

    try {
        java.sql.Statement s = conn.createStatement();
        int r = s.executeUpdate (todo);
    }
    catch (Exception e) {
        return ("Oh oops - code 003\n"+e);
    }

    return (todo);

}

public static String select(String name, String group, String yearborn) {

    java.sql.Connection conn = linktodata();

    StringBuffer reply = new StringBuffer("<table border=1>");

    StringBuffer conditions = new StringBuffer("where ");

    if (! group.equals("")) {
        conditions.append("ethnicorigin = '"+group+"'");
    }
    if (! name.equals("")) {
        if (conditions.length() > 6) {
            conditions.append(" && ");
        }
        conditions.append("fullname = '"+name+"'");
    }

    String selector = null;
    if (conditions.length() > 6) {
        selector = conditions.toString();
    } else {
        selector = "";
    }

    String todo = ("SELECT id, fullname, born, ethnicorigin "+
        " from population " + selector);

    try {
        java.sql.Statement s = conn.createStatement();
        java.sql.ResultSet r = s.executeQuery (todo);
        while(r.next()) {
            reply.append("<tr>");

```

```

        reply.append(tabit(r.getString("id")));
        reply.append(tabit(r.getString("ethnicorigin")));
        reply.append(tabit(r.getString("fullname")));
        reply.append(tabit(r.getString("born")));
        reply.append("</tr>");
    }
    reply.append("</table>");
}
catch (Exception e) {
    return ("Oh oops - code 003\n"+e);
}

return (reply.toString());

}

private static String tabit(String box) {
    return ("<td>"+box+"</td>");
}

private static java.sql.Connection linktodata () {

    java.sql.Connection conn = null;
    try {
        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    }
    catch (Exception e) {
        return conn;
        // return "Oh dear - code 001 and a quarter";
    }
    try {
        conn = java.sql.DriverManager.getConnection(
            "jdbc:mysql://bhajee/J850?user=jtest&password=");
    }
    catch (Exception e) {
        return conn;
        // return "Oh dear - code 001 and a half";
    }
    return conn;
}
}
}

```

This code contains the class that uses the JDBC API; to keep the example easy to follow, it's very specific in the information it uses. You'll find that we've hard-coded many items that would be better held in variables such as:

field names	( <b>fullname, ethnicorigin and born</b> )
Database	( <b>J850</b> )
Table name	( <b>Population</b> )
User name	( <b>Jtest</b> )
Password	(There is no password – leaky security!)
Host server for database	(implicitly <b>localhost</b> )
Database engine type	( <b>mysql</b> )

Since we're going to be making other connections to this same database, we've split out the common code into a private method called `linktodata` to avoid code duplication. Our insert method returns the actual SQL command that it used in its call to `executeUpdate` (used rather than `executeQuery`, since an Insert operation returns a count of the number of lines affected by the change, rather than data from tables). Again, you would not normally report back to your user in raw SQL, but it demonstrates the mechanisms for trainees "learning the ropes".

The initial form, together with the Java Servlet class and the Java JDBC access class, provide a complete mechanism for entering data into our simple SQL database; no other user-written files are required. What we haven't yet provided is any mechanism to access the data in our table. Let's look at that; a form has already been provided on the base of our initial HTML page (see Figure 106).

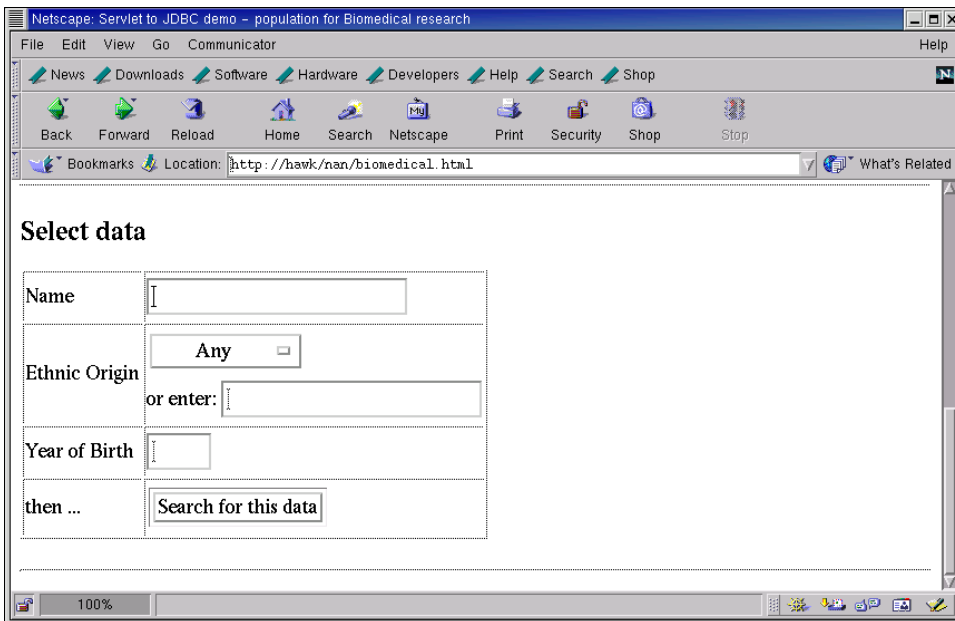
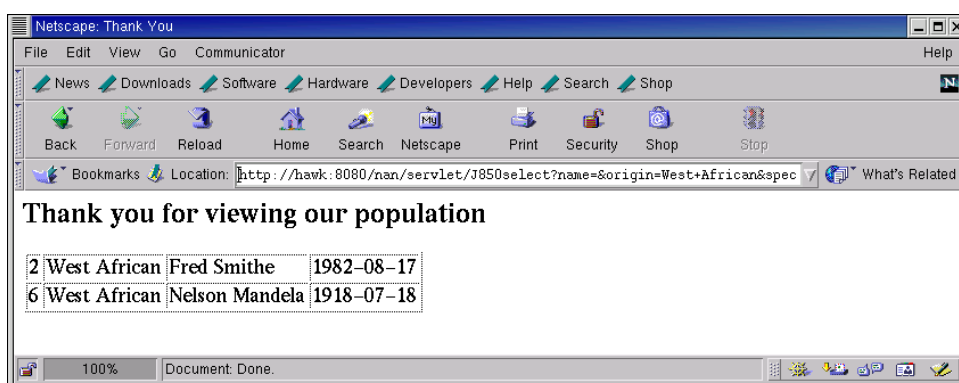


Figure 106 A form has been provided on the base of our initial HTML page

Completing and submitting this form, we get a thank you page (see Figure 107).

Figure 107 The thank you page after completing and submitting the form.



The only additional file here is the Servlet class `J850select`, which is called up from the HTML form and provides the servlet mechanism; it calls the `select` method in the same JDBC referencing class that we used to insert data – although that class has too many constants embedded in it for a real live application, it does at least encapsulate most of that information so that we could change the password or database engine or user name or some other value without the need to start changing other classes too!

Here's the source:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/*
 * Inserting record into a population table
 */

public class J850select extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {

        String called = request.getParameter("name");

        String origin = request.getParameter("special");
        if (origin.equals("")) {
            origin = request.getParameter("origin");
        }
        if (origin.equals("Any")) {
            origin = "";
        }
        String year = request.getParameter("year");

        // Now do the real work ....

        String matches = J850_jdbc.select(called,origin,year);

        // After the work is done, send the reply out ....

        response.setContentType("text/html");
    }
}
```

```

        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        out.println("<title>Thank You</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<body>");

        out.println("<h1>Thank you for viewing our population</h1>");
        out.println (matches);
        out.println("</body>");
        out.println("</html>");
    }
}

```

Once again, this is a straightforward wrapper that calls the JDBC referencing class from within the Servlet.

As an aside, you might like to note how we've written our search mechanism. Users will want to search by name, by year born, or by ethnic group; the `where` clause of our SQL request starts off empty (thus calling up all records), but if any entries are made into the selection form before it's submitted, then each such entry is combined (using an "and" operator so that everything must match) to allow us to become highly selective in the records that are chosen.

A final note on this example: The example assumes that the database and table already exist when data is first inserted or selected. Creating the table in a live application should be a one-off operation, and you really don't want to write a Servlet just to do it; much better to use the tools provided with the SQL engine. In our case that means the `mysql` program itself. In order to recreate an empty table before new courses (and also so that trainees can do this for themselves), we've provided an SQL script file:

```

# make_J850.sql - make up database for Java Servlet to JDBC demo
# Data is based on populations for a medical research group.

drop database if exists J850;
create database J850;
use J850;

drop table if exists population;
create table population (
    id          int primary key auto_increment,
    fullname    text,
    ethnicorigin text,
    born        date
);

# Make up a test record
insert into population (fullname, ethnicorigin, born)
values ("Graham Ellis", "Caucasian", "1954-07-16");

```

### *Example: A more general table management Servlet*

We've just completed above the study of quite a long example to show you the mechanisms of updating and querying a specific table. Code in that example is

repeated for each field, and the field names occur in multiple places in the code. Why? Just because that makes it easier to follow for newcomers – for maintenance, it's going to be a nightmare as the code grows.

Here's a further example, which is a single servlet class that generates the forms, and also calls both the insert and the select queries. It replaces three files from our previous example, and furthermore it only includes one explicit reference to each field name making it very easy to change in the future.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/*
 * General class for inserting and reporting on a table
 */

public class J850staff extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        String action = "Unselected";
        try {
            action = request.getParameter("action");
        } catch (Exception e) {
        }
        if (action == null) action = "Unselected";

        String [] fields = {"name","job","office","hometown","title"};
        String reply = "";

        if (action.equals("Unselected")) {
            reply = "Hello and Welcome";
        } else if (action.startsWith("Insert")) {
            reply = runInsert(fields,request);
        } else if (action.startsWith("Select")) {
            reply = runSelect(fields,request);
        } else {
            reply = "Corrupt form";
        }

        reply = reply + "<BR><HR><BR>" + getForms(fields);

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println(reply);
    }

    String runInsert (String [] fields,HttpServletRequest request) {

        StringBuffer fnames = new StringBuffer("");
        StringBuffer fvals = new StringBuffer("");
```

```

for (int i=0; i<fields.length; i++) {
    String called = request.getParameter(fields[i]);
    if (i>0) fnames.append(", ");
    fnames.append(fields[i]);
    if (i>0) fvals.append(", ");
    fvals.append("'" + called + "'");
}

J850sjdbc.insert(fnames.toString(), fvals.toString());

return ("Insertion Done");

}

String runSelect (String [] fields,HttpServletRequest request) {

    StringBuffer fselect = new StringBuffer("where ");

    for (int i=0; i<fields.length; i++) {
        String called = request.getParameter(fields[i]);

        if (! called.equals(""))
            {
                if (fselect.length() > 6) {
                    fselect.append(" && ");
                }
                fselect.append (fields[i] + " = '" +
                    called + "' ");
            }
    }
    if (fselect.length() <= 6) {
        fselect = new StringBuffer("");
    }

    return (J850sjdbc.select(fields, fselect.toString()));

}

String getForms (String [] fields) {

    StringBuffer insert = new StringBuffer("<table border=1>");

    for (int i=0; i<fields.length; i++) {
        insert.append("<tr><td>" + fields[i] + "</td><td><input name=\"" +
            fields[i] + "\"></td></tr>");
    }

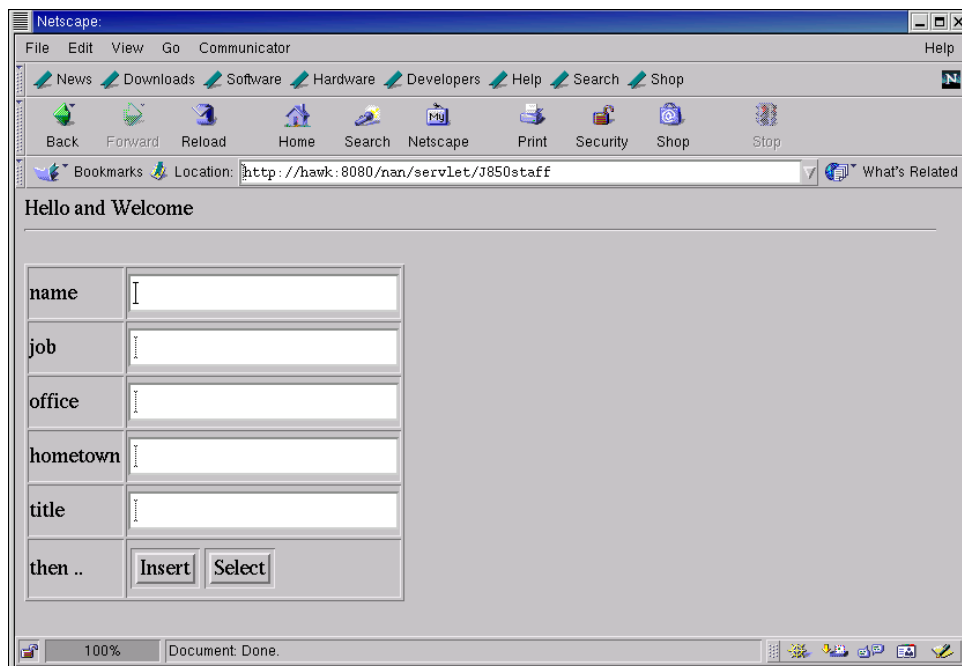
    insert.append("<tr><td>then ..</td><td>" +
        "<input type=submit name=action value=Insert>" +
        "<input type=submit name=action value=Select>" +
        "</td></tr></table>");

    return ("<form>" + insert.toString() + "</form>");
}
}

```

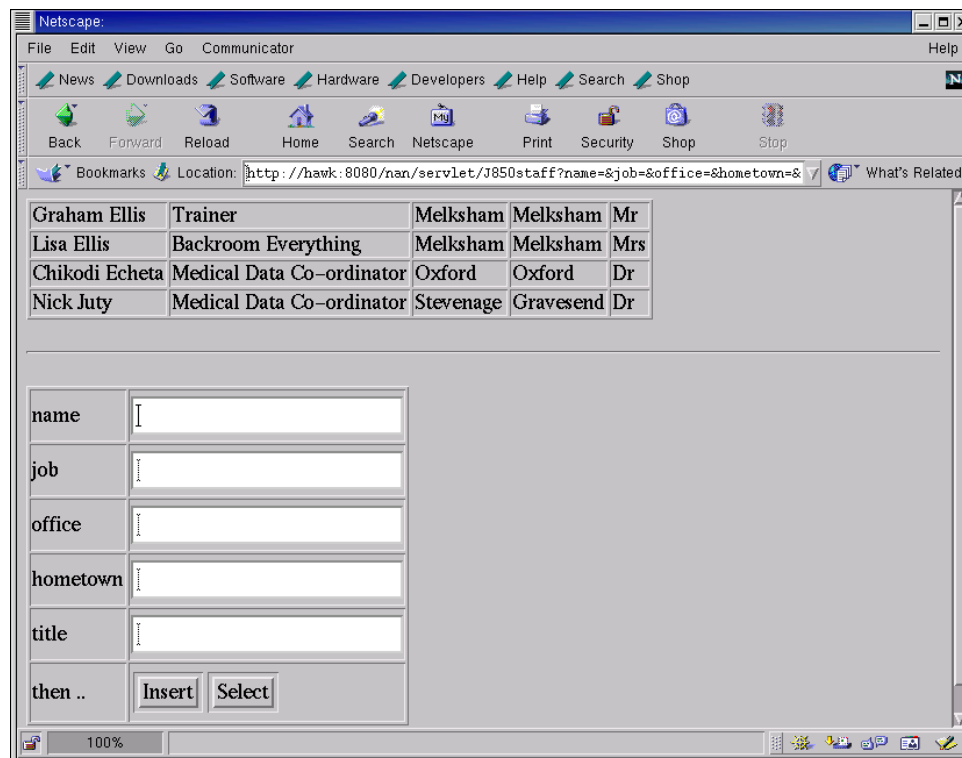
Let's see the form (see Figure 108.) that generates when we first run it (and note the URL is direct to the servlet!).

Figure 108 The form when first run.



The form here was generated by the Servlet itself, which starts off by checking which submit button was pressed, and if it was neither puts up the welcome form. Should the Insert or Select button be pressed, the resulting action is performed and the result reported prior to the generation of a new form - thus providing a mechanism through which a series of updates and enquiries can be made without having to keep following links back to the form.

Figure 109 Result of the completed form.



Here's the source code of that class that references the SQL database:

```
public class J850sjdbc {

public static String insert(String names, String values) {

    java.sql.Connection conn = linktodata();

    String todo = ("INSERT into staff " +
        "(" + names + ") values (" + values + ")");

    try {
        java.sql.Statement s = conn.createStatement();
        int r = s.executeUpdate (todo);
    }
    catch (Exception e) {
        return ("Oh oops - code 003\n"+e);
    }

    return (todo);

}

public static String select(String [] fields, String selector) {

    java.sql.Connection conn = linktodata();

    StringBuffer reply = new StringBuffer("<table border=1>");

    String todo = ("SELECT * "+
        " from staff " + selector);

    try {
        java.sql.Statement s = conn.createStatement();
        java.sql.ResultSet r = s.executeQuery (todo);
        while(r.next()) {
            reply.append("<tr>");
            for (int i=0;i<fields.length;i++) {
                reply.append(tabit(r.getString(fields[i])));
            }
            reply.append("</tr>");
        }
        reply.append("</table>");
    }
    catch (Exception e) {
        return ("Oh oops - code 003\n"+e);
    }

    return (reply.toString());

}

private static String tabit(String box) {
    return ("<td>"+box+"</td>");
}

private static java.sql.Connection linktodata () {
```

```

java.sql.Connection conn = null;
try {
    Class.forName("org.gjt.mm.mysql.Driver").newInstance();
}
catch (Exception e) {
    return conn;
    // return "Oh dear - code 001 and a quarter";
}
try {
    conn = java.sql.DriverManager.getConnection(
        "jdbc:mysql://bhajee/J850a?user=jtest&password=");
}
catch (Exception e) {
    return conn;
    // return "Oh dear - code 001 and a half";
}
return conn;
}
}

```

Perhaps the most major change in this example from the previous one is the use of loops to pass through each field of the database table in turn, collecting the field names in each case from an array of strings that is set up in just a single place in the code. Note the heavy use of `StringBuffers` to build up response strings (e.g. HTML tables, SQL commands) affecting every field.

Although there are many enhancements that can be made to this example, you'll find that it provides at least a starting point for many applications of Java, where you want to provide an Internet-(or intranet) to-database bridge.

### *Enhancing our examples*

Remember, "Design MATTERS". If you're in at the start of a project, you should give great attention to the overall structure and design of that project and plan to the best of your ability for current and foreseen future requirements from day one, and you should give great consideration to the design before you write even one line of code. If you jump in to coding on day one without a thorough understanding of your customer's requirement of your work, then you risk both wasting a huge amount of resources, and providing a poor or unacceptable solution. For larger projects, you may wish to consider using formal design methods such as UML, and design tools such as Rose to manage your designs.

Our examples in this section of the course are fit for the purpose for which they were written, which was to illustrate the mechanisms that you may use to interface SQL databases to the Web through Java, but for a production environment there will be many more aspects to consider. Here are a number of pointers and ideas for you to consider. Many of these would be necessary enhancements to our examples were you to wish to use them in a live situation:

- Provision of edit and delete capabilities. Did you notice that once we had inserted data into our table, we had no way of removing or correcting it?
- Provision of the ability to handle fields of different types; our example was restricted purely to table of textual strings.
- Limiting the number of records reported back and/or providing a paging mechanism; while our table remains short, reporting everything is fine ... but if you have a database of several megabytes it would not be.
- Requesting confirmation as data is entered (or updated). It's very clever of our table to provide a single form for both data entry and selection, but it's almost too clever; it would be very easy indeed for the user to make errors.

- A mechanism to allow different levels of access to different users, including a login capability, passwords, maintenance of state (i.e. Knowing who is who as users move on from one page to the next in a sequence), and also the provision of a mechanism to manage the users.
- You may wish to consider using the JSP interface as an alternative to using Servlets directly, especially if your database is a small part of a major web site which is maintained by a team using software tools such as Dreamweaver or Frontpage, and coming from a graphic arts background.
- The selection mechanism should be improved; our samples required an exact match to a field for a record to be supported, whereas a regular expression match might be more appropriate. This could be done using the Apache class (Java prior to 1.4), the built-in Regular Expressions introduced in Java 1.4, or the regular expression capabilities that are available in database engines such as MySQL. You might also wish to provide more advanced search capabilities such as "or" as well as "and", "not", and so on.
- The application could be enhanced to support multiple tables, and relate them to one another as data was entered, and also as the data is selected and reported.
- Rather than generating the complete HTML response page within your Java, you might wish to read in template response pages from files (or from another database table) and then replace markers in those templates with your results. This helps keep the HTML and Java separate, allowing for easier maintenance, especially if you have different experts in each of the languages on your staff.
- As a further step of generalisation of our example, you might wish to hold the field names in a configuration file rather than in the Java code itself, allowing the same applet to be used across a series of applications. As you start to provide enhancements such as this (the config file would include text descriptions of each field, field names, field types), and also standard profiles for the HTML, you're moving away from developing a specific application and towards a much more exciting content management system!

Remember: Design MATTERS. If you are about to embark on a project which uses the facilities described in this section, you need to talk to your user, listen and learn about his requirements, and consider how which of the enhancements we describe above might apply – and don't be too shocked if they all apply to you!

# *License*

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

### 3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log  
Original Version, Well House Consultants, 2004

Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_

*License Ends.*