84

Combined Java Web Example: Servlets, JDBC and Graphics

Java supports a very wide range of technologies through class libraries and containers. This module provides you with an example of the combination of some of those technologies - servlets, database access and graphics - in order to provide pictorial, dynamic web content without the need for any browser plugins.

84.1 Introduction to the requirement

This module builds up the combination of standard Java classes within a Servlet to provide a browsable graphic representation of a database table. In other words, it demonstrates the conversion of this table:

```
mysql> select * from prestatyn;
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
| month | avtemp | highest           | avnight | lowest             | avsun | avrain | id |
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
| JAN   |    7.6 | 17.6 on 10th 1971 |     2.6 | -12.2 on 25th 1963 |    47 |     60 |  1 |
| FEB   |    7.4 | 18.3 on 23rd 1990 |     2.2 | -11.1 on 16th 1969 |    66 |     43 |  2 |
| MAR   |    9.3 | 23.9 on 29th 1965 |     3.7 | -7.8 on 2nd 1965   |   105 |     51 |  3 |
| APR   |   11.3 | 26.6 on 16th 2003 |     5.1 | -2.7 on 2nd 1984   |   145 |     46 |  4 |
| MAY   |   14.9 | 28.1 on 31st 1978 |     7.8 | 1.2 on 2nd 1981    |   191 |     49 |  5 |
| JUN   |   17.5 | 31.6 on 18th 2000 |    10.8 | 3.9 on 4th 1975    |   184 |     54 |  6 |
| JUL   |   19.1 | 32.4 on 3rd 1976  |    12.9 | 5.6 on 17th 1965   |   171 |     46 |  7 |
| AUG   |     19 | 34.3 on 2nd 1990  |    12.8 | 4.4 on 31st 1964   |   159 |     59 |  8 |
| SEP   |   17.1 | 27.5 on 11th 2000 |    10.9 | 3.3 on 16th 1986   |   123 |     69 |  9 |
| OCT   |   14.3 | 25.5 on 8th 1995  |     8.5 | -1.2 on 30th 1988  |    94 |     70 | 10 |
| NOV   |   10.4 | 21.7 on 4th 1946* |     5.3 | -4.4 on 24th 1967  |    55 |     76 | 11 |
| DEC   |    8.5 | 16.7 on 10th 1994 |     3.7 | -13.5 on 12th 1981 |    35 |     68 | 12 |
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
12 rows in set (1.15 sec)

mysql>
```

into a table in HTML:

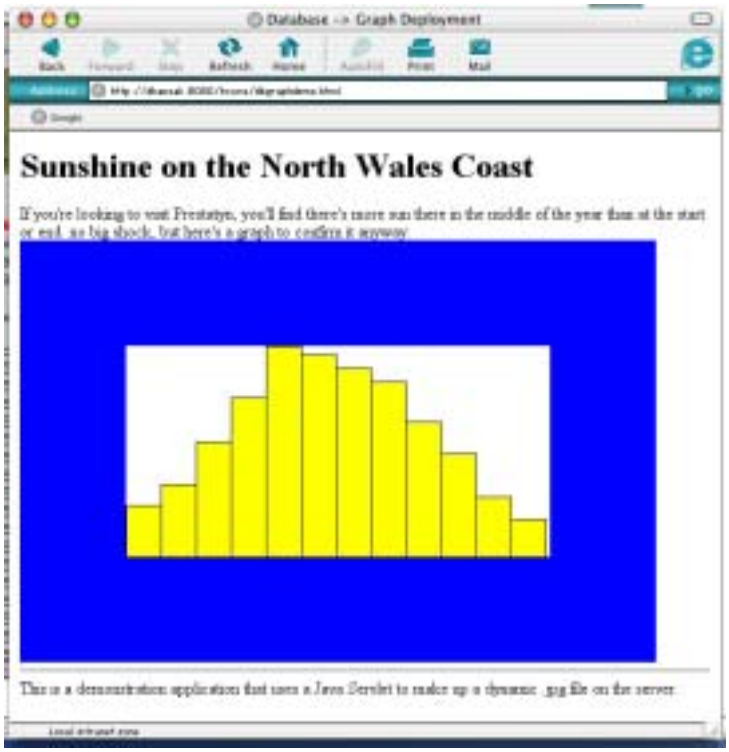


Figure 410 Sunshine in Prestatyn, viewed in a web page

84.2 Planning the application

It looks like a simple application to draw a graph of some numbers that are held in a database, and indeed it is, but it involves a number of technologies. For a more complex and new application, you'll be encouraged to use an application modelling

language such as UML (or something less formal) to get your strategy worked out before you start, and divide your application into separate classes to aid code development, testing and re-use.

In our application, we'll use

- JDBC to retrieve the data to be drawn from a relational database
- The Abstract Windowing Toolkit (awt) to draw our graphics
- The Image I/O classes in javax to output our image as a .jpg
- A servlet container to make our graphic generator web visible
- Coyote, Catalina and Tomcat in which to run the servlet

This module does not describe each of these technologies in detail; rather, it's concerned with the building up to the complete sample application and the combination of the technologies.

We'll build this demonstration up as a "shell" application. In other words, we'll use just a little of each technology to show how they all go together, and we'll leave you to split out the technologies into separate classes and expand on each of them to suit your particular database display needs.

84.3 Generating the image file

There's a big decision to make with regard to the deployment of our application - are we going to generate our graphics on the server computer, or within the browser on the client? If users connect over restricted connections (limited in bandwidth or speed) and you can be sure they'll have a Java enabled browser, then you should select client side graphics. If bandwidth is not a problem, or if you can't be sure of your users having Java on their local machines, you'll want to go for server side graphics.

Whilst this is a big deployment decision, the actual code used to generate your graphics will be encapsulate in its own class so that you can switch your deployment model later on, and with relative ease, if you choose.

Let's generate an image file on a server. You probably learnt Java Graphics using Applets, awt Frames, JFrames or other containers, but all of these are interactive components with event handlers, things that are quite alien to the server environment. They also generate their graphics on-screen. Again, something we certainly don't want to happen on the server. Solution: we'll use **`BufferedImage`** object which will allocate memory for the graphic within our application area:

```
  RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
```

The image is to be 600 pixels wide by 400 pixels tall, and is to use Red-Green-Blue technology rather than colour lookup maps. If you're producing .jpg files, this is a pretty good standard type to use, but there is no overloaded constructor to set a default type if you just give a width and height in pixels. Also note at this point that you'll want to make the image as large as it will be displayed in use, but no larger. Browsers can shrink images down to display them if necessary, but if they expand them to make them bigger all they can do is expand the dots. On the other hand, if you set up an image that's bigger than you'll ever use, you're liable to waste resources as you create it, and bandwidth as you transmit it.

Next, get the **`Graphics`** object from the **`BufferedImage`** object, and then use standard awt package methods to paint to the graphics. This paint code will work equally well for your **`BufferedImage`**, and for Frames and other interactive Graphics:

```
Graphics g = ((BufferedImage)myimg).createGraphics();
g.setColor(Color.blue);
g.fillRect(0,0,600,400);
g.setColor(Color.white);
```

```
    g.fillRect(100,100,400,200);
```

All that remains to do now is to save the image that we've created in memory. Later in our application, we'll be passing it out through our servlet. But for the moment, let's output it to file:

```
    ImageIO.write(myimg,"jpg",new File("image_1.jpg"));
```

And that's all there is to it. You can choose jpg and png formats, and give it the File object to which to write.

Here's the complete code:

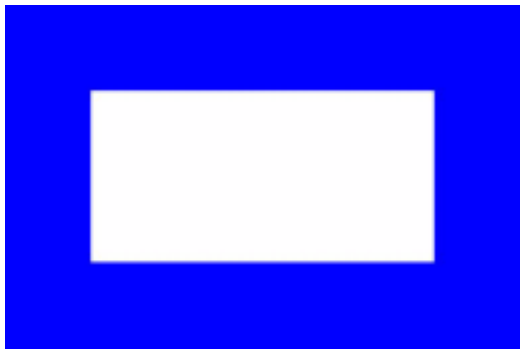
```
// Well House Consultants
// Im1 - Generating an image file on the server

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;

public class Im1  {

public static void main(String[] args) throws IOException

  {
    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    ImageIO.write(myimg,"jpg",new File("image_1.jpg"));

}

}
```

So what happens when we compile and run our java class?

```
[graham@dhansak java]$ javac Im1.java
[graham@dhansak java]$ java Im1
[graham@dhansak java]$
```

Is that really it? Yes, and that's good news since we'll be running this code on a server where we don't want any output. It's produced an output file called *image_1.jpg*, which looks like:

Figure 411 image_1.jpg - the output file from our Im1 class



84.4 Getting data to graph from an SQL database

The second step in our application is to vary the content of the graphic based on the content of a database. As a proof of concept, we're going to write our initial code heavily loaded with constants, and we're going to tell the code to ignore exceptions.

Let's connect to the database:

```
java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/test?user=trainee&password=abc123");
```

We're using MySQL here. It's a free license for testing, it runs easily on our training machines, and we know it pretty well because we run courses on it. MySQL also has the advantage for development of having one of the smaller subsets of SQL implemented; if you get your code running on MySQL, chances are that you can shift to another database engine very easily.

Having established the connection, the next step is to pull back the data we're going to draw. Let's run the query:

```
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
```

We don't know how much data will be returned; we could do a count query to find out and then set up an array, but much better to use an **`ArrayList`**:

```
ArrayList Values = new ArrayList();
while(r.next()) {
        Values.add(r.getString("avtemp")) ;
        }
```

In fact, our code is slightly more complex that this, as we've taken the opportunity to check through our data and find the maximum value so that we can work out how to scale our graph later. Object Orientation purists will shudder at this and suggest that we should have used a separate loop. Maybe; its a question of balance in the design. Here's the code including the hunt for the maximum:

```
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }
```

There's no change from our "Im1" class in how we set up the graphic, and how we draw the first few (constant) rectangles.

When we come to add in the bar chart information, we do so as follows:

```
  for (int k=0; k<Values.size(); k++) {
      int step = 400 / Values.size();
     int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
      g.setColor(Color.yellow);
      g.fillRect(100 + k * step, 300 - high, step, high);
      g.setColor(Color.black);
      g.drawRect(100 + k * step, 300 - high, step, high);
      }
```

The arithmetic here is converting from data (or world) coordinates which are in the range of 0 to the number of elements in the list in x, and from 0 to the maximum data value in the data retrieved in y to pixel numbers, which are in the range 100 to 500 in x and 300 down to 100 in y. Each rectangle is filled in yellow, and then outlined in black. These things are best seen in a diagram:

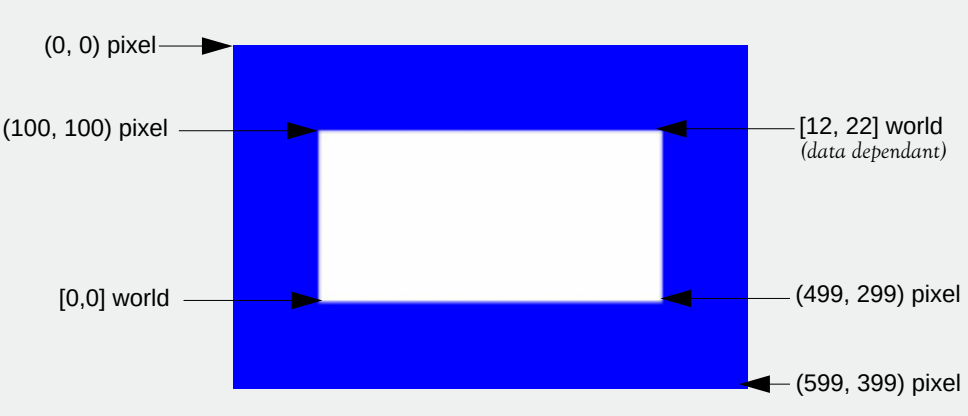
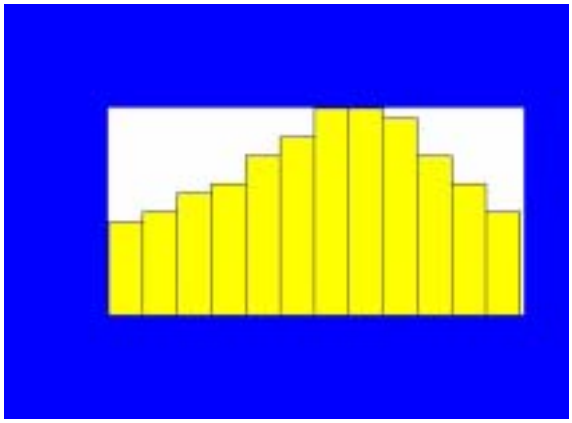


Figure 4.12 Planning out our bar chart – mapping real world data values onto pixel positions

Finally, we save our image to disk just as we did before. Just as we did before, we run our application and we see nothing at the command line, but an image file is output. Here's the data:

```
mysql> select * from lisbon;
+-----------+--------+----+
| month     | avtemp | id |
+-----------+--------+----+
| January   |     10 |  1 |
| February  |     11 |  2 |
| March     |     13 |  3 |
| April     |     14 |  4 |
| May       |     17 |  5 |
| June      |     19 |  6 |
| July      |     22 |  7 |
| August    |     22 |  8 |
| September |     21 |  9 |
| October   |     17 | 10 |
| November  |     14 | 11 |
| December  |     11 | 12 |
+-----------+--------+----+
12 rows in set (0.00 sec)

mysql>
```

Figure 413 image_2.jpg – a bar chart of the data in our SQL database



The complete source code at this stage:

```
// Well House Consultants, 2004.
// Generating an image file from data collected through JDBC

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;

public class Im2  {

public static void main(String[] args) throws Exception

  {

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/
test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }


    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
      int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
        g.setColor(Color.yellow);
```

```
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }
    ImageIO.write(myimg,"jpg",new File("image_2.jpg"));
}
}
```

84.5 Making the graphic available dynamically via the web

In order to make the graphic available via the web, we need to produce it within a servlet[1] and then run that servlet within a Java-enabled web server. For the purpose of this example, we'll deploy our servlet within the Tomcat server. This is the reference implementation of the Servlet definition, so if it works under Tomcat, it's supposed to work anywhere else. Tomcat is part of the Apache project. All the Java parts of the Apache project are now grouped under the Jakarta label. Tomcat uses the Catalina Java Runtime Environment internally, and its HTTP support is provided by Coyote. Sourcing, installation and configuration of these components is beyond the scope of this training module.

What do we change in our source code in order to run in Catalina rather than within the default JRE? Firstly, we have our class extend Servlet, and we write our main code within a **`doGet`** method rather than a main method:

```
    public class Im3 extends HttpServlet {
    public void doGet(HttpServletRequest incoming,
             HttpServletResponse outgoing) {
```

Our servlet needs to set the outgoing MIME type, and also get the output stream that it's to use. We no longer wish to write to a file on the server:

```
    outgoing.setContentType("image/jpeg");
    ServletOutputStream out = outgoing.getOutputStream();
```

The only other substantive change we make is to write our image to the **`OutputStream`**, and add in call to close that stream, thus:

```
    ImageIO.write(myimg,"jpg",out);
    out.close();
```

We've been careful in our demonstration code not to lengthen it with exception handling, but we do now need to a add a little. The servlet container doesn't allow for certain exceptions and we therefore must handle them within our class. Here's the complete code.

```
// Well House Consultants
// Generating an image file from data collected through JDBC
// Server based - Servlet container

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Im3 extends HttpServlet {
public void doGet(HttpServletRequest incoming,
```

[1] we could use a JSP if we preferred

```
    HttpServletResponse outgoing) {

try {
    outgoing.setContentType("image/jpeg");
    ServletOutputStream out = outgoing.getOutputStream();

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/
test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }
    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
      int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
        g.setColor(Color.yellow);
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }

    ImageIO.write(myimg,"jpg",out);

    out.close();
        }
    catch (Exception e) {}
    }
}
```

In order to deploy this as a web application, you'll need to upload the class to the web server, configure your web.xml, etc. We've deployed it within a web application called "towns" which has the following structure:

```
[graham@dhansak towns]$ ls -R
.:
WEB-INF  dbgraphdemo.html  index.html

./WEB-INF:
classes  web.xml

./WEB-INF/classes:
Im3.class  Im4.class  TinyServ.class
```

The directives to link the servlet's URL to the class are within the *web.xml* file. Here are the lines that do it:

```
<web-app>
    <servlet>
    <servlet-name>Im3</servlet-name>
    <servlet-class>Im3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Im3</servlet-name>
    <url-pattern>/im3</url-pattern>
  </servlet-mapping>
</web-app>
```

Here's our results via a browser

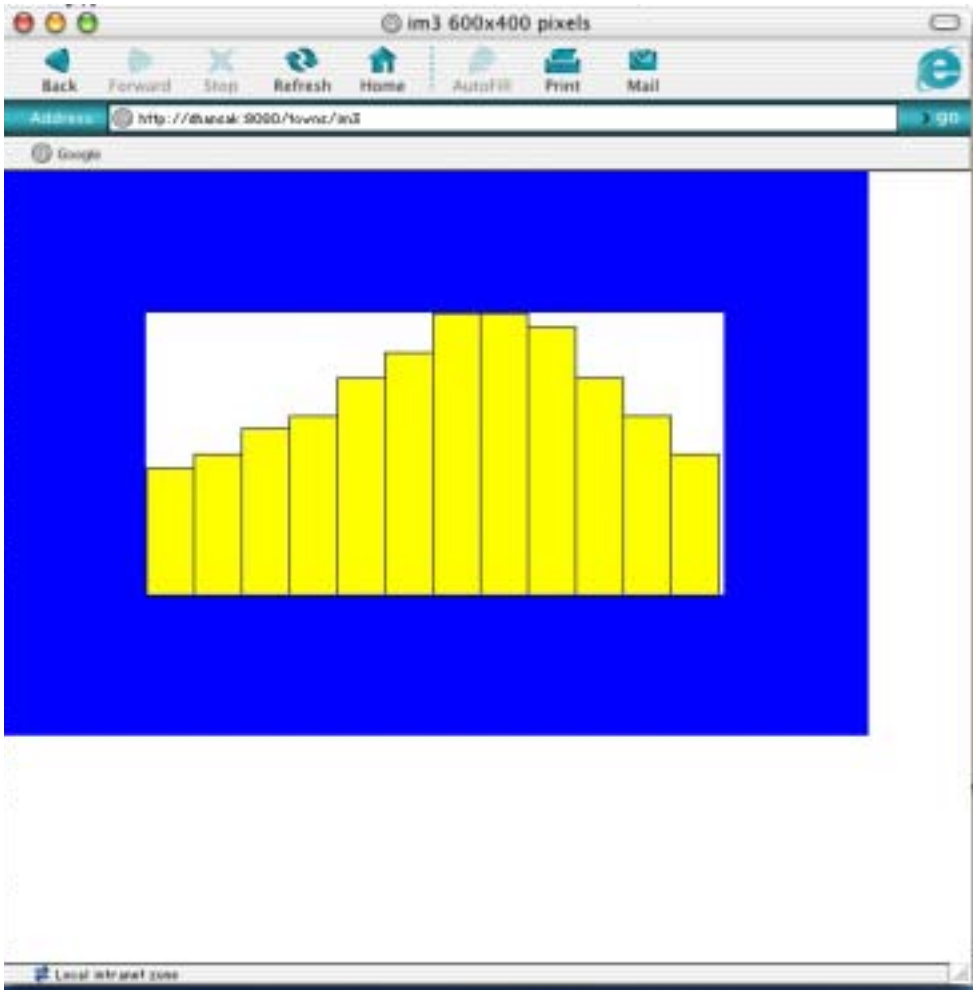


Figure 414 Bar chart of average temperature in Lisbon by month

84.6 Completing our sample web application

So far, we've generated an image on a server computer, we've sourced data from a relational database for that image, and we've provided a Servlet front end for it so that it's accessible via the web.

Let's complete this demonstration by incorporating our graphic in a more complete web application. We'll modify our class so that it can pick up the table name and column to be charted from a form, and we'll incorporate the resulting image within an HTML web page.

Here's the web page source:

```
<head>
<title>Database -> Graph Deployment</title>
</head>
<body bgcolor=white>
<h1>Sunshine on the North Wales Coast</h1>
If you're looking to visit Prestatyn, you'll find there's more sun there in the middle of the year than at
than the start or end. No big shock, but here's a graph to confirm it anyway:<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avsun"><br>
<hr>
This is a demonstration application that uses a Java Servlet to make up a dynamic .jpg file on the
server.
</body>
```

And here are the changes to the Java source. Firstly, we need to collect the name of the table and column to be charted:

```
String WantedTable = incoming.getParameter("table");
String WantedColumn = incoming.getParameter("column");
```

and then use those variables within our database access code:

```
java.sql.ResultSet r = s.executeQuery ("SELECT * from "+WantedTable);
```

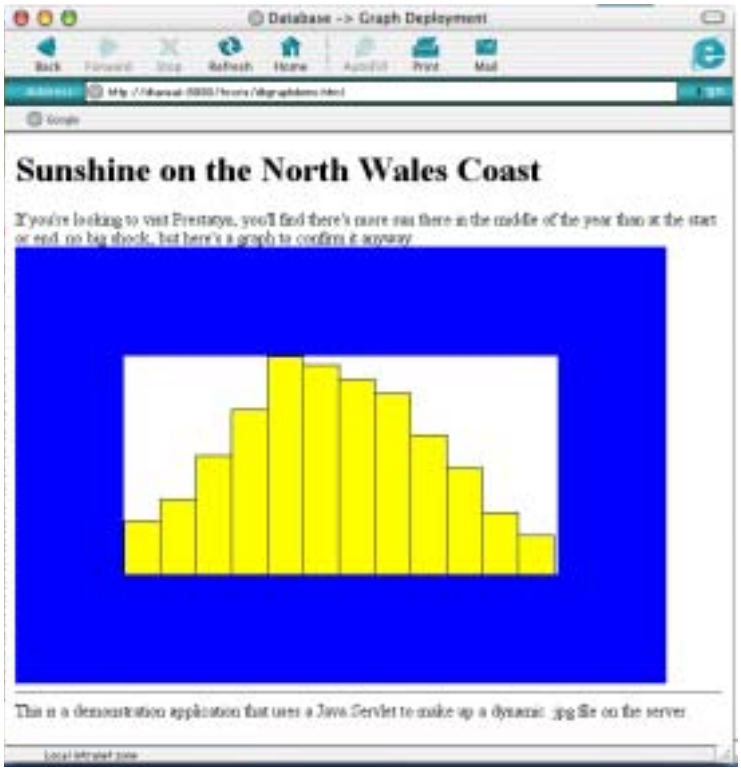
and

```
Values.add(current = r.getString(WantedColumn)) ;
```

That's all the changes we have made!

When we call up the html page, it in turn will call up the image with the parameters set. We get the result that we were aiming at, shown at the start of this module.

Figure 415 Our final result



Here's the complete source code with all the parts integrated:

```
// Well House Consultants
// Generating an image file from data collected through JDBC and HTTP parameters

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Im4 extends HttpServlet {

public void doGet(HttpServletRequest incoming,
    HttpServletResponse outgoing) {

        String WantedTable = incoming.getParameter("table");
        String WantedColumn = incoming.getParameter("column");

try {
    outgoing.setContentType("image/jpeg");
      ServletOutputStream out = outgoing.getOutputStream();

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from "+WantedTable);
ArrayList Values = new ArrayList();
float maxval = 0.0f;
while(r.next()) {
        String current;
        Values.add(current = r.getString(WantedColumn)) ;
        float temp;
        if ((temp = Float.parseFloat(current)) > maxval) maxval = temp;
        }

    RenderedImage myimg = new BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
        int high = (int)(200.0f * (Float.parseFloat((String)Values.get(k))) / maxval);
        g.setColor(Color.yellow);
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }

    ImageIO.write(myimg,"jpg",out);

    out.close();
        }
    catch (Exception e) {}

    }
}
```

84.7 Extending the application

You've seen the basic framework of an application in this module. It's a "how to" integration guide that gives you an overview. For your own application you'll want to provide much more for the user. Here are some areas:

a) You'll want to provide axes and scales on your graphics; the awt has a good set of font handlers, and there are 2D and 3D graphics libraries available too. You could go on to provide multiple graphs on one set of axes for instance
b) You should handle exceptions neatly and correctly. We have elected not to do so to keep the code lengths down.
c) You should structure each element of your application into separate classes so that each part is reusable elsewhere, each part is testable in it's own right, and each part can be maintained by separate staff if necessary
d) You'll want to enhance your HTML, provide forms and much more.

Just to show you how easily the extensions can be started, we've modified just our HTML to use the same servlet to produce multiple graphs

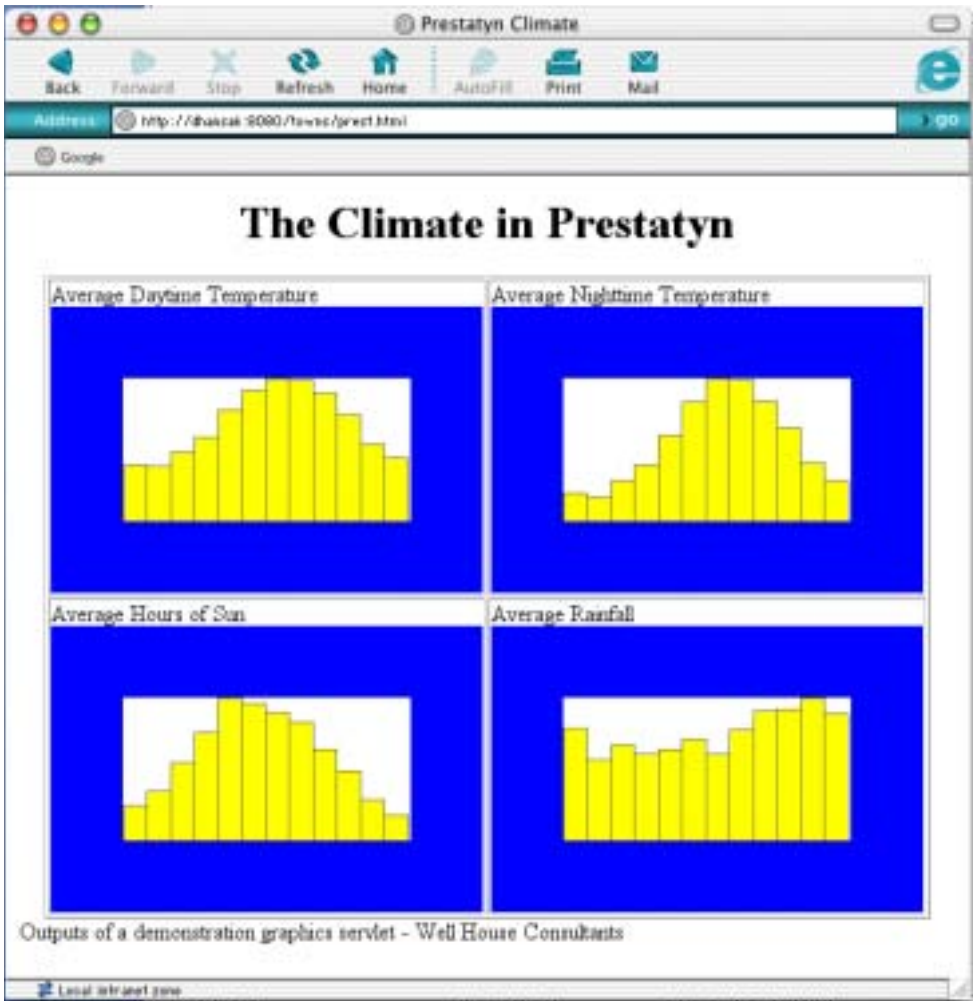


Figure 416 Sun, rain and temperature patterns

and here's the HTML:

```
<html>
<head>
<title>Prestatyn Climate</title>
</head>
<body bgcolor=white>
<center>
<h1>The Climate in Prestatyn</h1>
<table border=1>
<tr>
<td>Average Daytime Temperature<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avtemp" width=300></td>
<td>Average Nighttime Temperature<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avnight" width=300></td>
</tr>
<tr>
<td>Average Hours of Sun<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avsun" width=300></td>
<td>Average Rainfall<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avrain" width=300></td></tr>
</table>
</center>
Outputs of a demonstration graphics servlet - Well House Consultants
</body>
</html>
```

Exercise

84

Combined Java Web Example: Servlets, JDBC and Graphics

Java supports a very wide range of technologies through class libraries and containers. This module provides you with an example of the combination of some of those technologies - servlets, database access and graphics - in order to provide pictorial, dynamic web content without the need for any browser plugins.

84.1 Introduction to the requirement

This module builds up the combination of standard Java classes within a Servlet to provide a browsable graphic representation of a database table. In other words, it demonstrates the conversion of this table:

```
mysql> select * from prestatyn;
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
| month | avtemp | highest           | avnight | lowest             | avsun | avrain | id |
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
| JAN   |    7.6 | 17.6 on 10th 1971 |     2.6 | -12.2 on 25th 1963 |    47 |     60 |  1 |
| FEB   |    7.4 | 18.3 on 23rd 1990 |     2.2 | -11.1 on 16th 1969 |    66 |     43 |  2 |
| MAR   |    9.3 | 23.9 on 29th 1965 |     3.7 | -7.8 on 2nd 1965   |   105 |     51 |  3 |
| APR   |   11.3 | 26.6 on 16th 2003 |     5.1 | -2.7 on 2nd 1984   |   145 |     46 |  4 |
| MAY   |   14.9 | 28.1 on 31st 1978 |     7.8 | 1.2 on 2nd 1981    |   191 |     49 |  5 |
| JUN   |   17.5 | 31.6 on 18th 2000 |    10.8 | 3.9 on 4th 1975    |   184 |     54 |  6 |
| JUL   |   19.1 | 32.4 on 3rd 1976  |    12.9 | 5.6 on 17th 1965   |   171 |     46 |  7 |
| AUG   |     19 | 34.3 on 2nd 1990  |    12.8 | 4.4 on 31st 1964   |   159 |     59 |  8 |
| SEP   |   17.1 | 27.5 on 11th 2000 |    10.9 | 3.3 on 16th 1986   |   123 |     69 |  9 |
| OCT   |   14.3 | 25.5 on 8th 1995  |     8.5 | -1.2 on 30th 1988  |    94 |     70 | 10 |
| NOV   |   10.4 | 21.7 on 4th 1946* |     5.3 | -4.4 on 24th 1967  |    55 |     76 | 11 |
| DEC   |    8.5 | 16.7 on 10th 1994 |     3.7 | -13.5 on 12th 1981 |    35 |     68 | 12 |
+-------+--------+-------------------+---------+--------------------+-------+--------+----+
12 rows in set (1.15 sec)

mysql>
```

into a table in HTML:

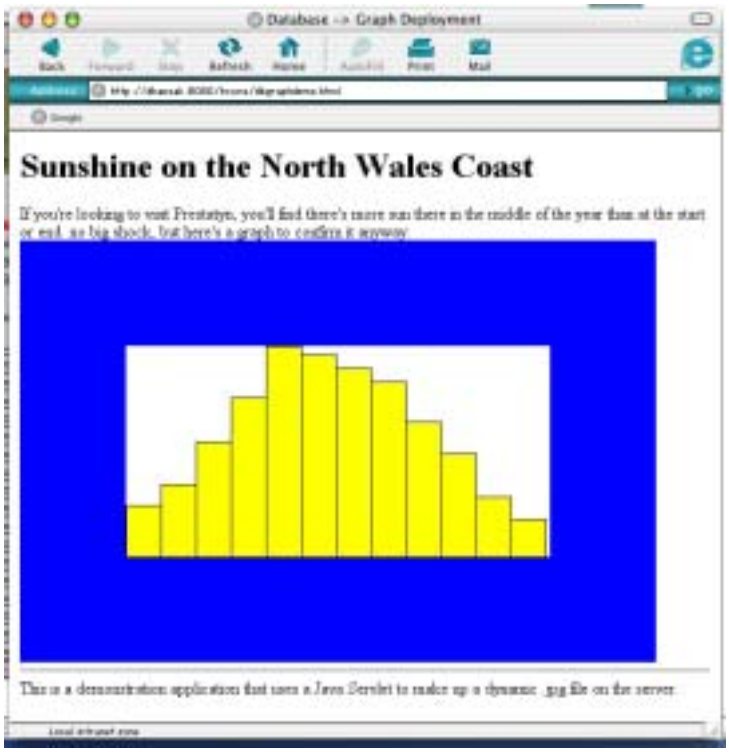


Figure 410 Sunshine in Prestatyn, viewed in a web page

84.2 Planning the application

It looks like a simple application to draw a graph of some numbers that are held in a database, and indeed it is, but it involves a number of technologies. For a more complex and new application, you'll be encouraged to use an application modelling

language such as UML (or something less formal) to get your strategy worked out before you start, and divide your application into separate classes to aid code development, testing and re-use.

In our application, we'll use

- JDBC to retrieve the data to be drawn from a relational database
- The Abstract Windowing Toolkit (awt) to draw our graphics
- The Image I/O classes in javax to output our image as a .jpg
- A servlet container to make our graphic generator web visible
- Coyote, Catalina and Tomcat in which to run the servlet

This module does not describe each of these technologies in detail; rather, it's concerned with the building up to the complete sample application and the combination of the technologies.

We'll build this demonstration up as a "shell" application. In other words, we'll use just a little of each technology to show how they all go together, and we'll leave you to split out the technologies into separate classes and expand on each of them to suit your particular database display needs.

84.3 Generating the image file

There's a big decision to make with regard to the deployment of our application - are we going to generate our graphics on the server computer, or within the browser on the client? If users connect over restricted connections (limited in bandwidth or speed) and you can be sure they'll have a Java enabled browser, then you should select client side graphics. If bandwidth is not a problem, or if you can't be sure of your users having Java on their local machines, you'll want to go for server side graphics.

Whilst this is a big deployment decision, the actual code used to generate your graphics will be encapsulate in its own class so that you can switch your deployment model later on, and with relative ease, if you choose.

Let's generate an image file on a server. You probably learnt Java Graphics using Applets, awt Frames, JFrames or other containers, but all of these are interactive components with event handlers, things that are quite alien to the server environment. They also generate their graphics on-screen. Again, something we certainly don't want to happen on the server. Solution: we'll use **`BufferedImage`** object which will allocate memory for the graphic within our application area:

```
  RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
```

The image is to be 600 pixels wide by 400 pixels tall, and is to use Red-Green-Blue technology rather than colour lookup maps. If you're producing .jpg files, this is a pretty good standard type to use, but there is no overloaded constructor to set a default type if you just give a width and height in pixels. Also note at this point that you'll want to make the image as large as it will be displayed in use, but no larger. Browsers can shrink images down to display them if necessary, but if they expand them to make them bigger all they can do is expand the dots. On the other hand, if you set up an image that's bigger than you'll ever use, you're liable to waste resources as you create it, and bandwidth as you transmit it.

Next, get the **`Graphics`** object from the **`BufferedImage`** object, and then use standard awt package methods to paint to the graphics. This paint code will work equally well for your **`BufferedImage`**, and for Frames and other interactive Graphics:

```
Graphics g = ((BufferedImage)myimg).createGraphics();
g.setColor(Color.blue);
g.fillRect(0,0,600,400);
g.setColor(Color.white);
```

```
    g.fillRect(100,100,400,200);
```

All that remains to do now is to save the image that we've created in memory. Later in our application, we'll be passing it out through our servlet. But for the moment, let's output it to file:

```
    ImageIO.write(myimg,"jpg",new File("image_1.jpg"));
```

And that's all there is to it. You can choose jpg and png formats, and give it the File object to which to write.

Here's the complete code:

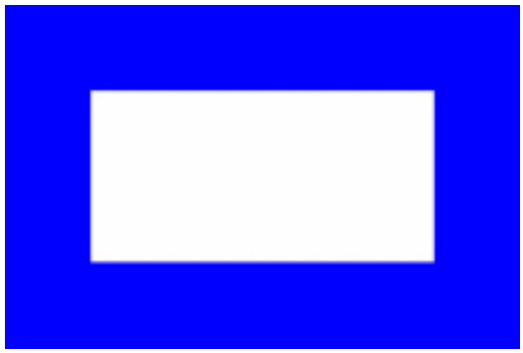
```
// Well House Consultants
// Im1 - Generating an image file on the server

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;

public class Im1  {

public static void main(String[] args) throws IOException

  {
    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    ImageIO.write(myimg,"jpg",new File("image_1.jpg"));

}

}
```

So what happens when we compile and run our java class?

```
[graham@dhansak java]$ javac Im1.java
[graham@dhansak java]$ java Im1
[graham@dhansak java]$
```

Is that really it? Yes, and that's good news since we'll be running this code on a server where we don't want any output. It's produced an output file called *image_1.jpg*, which looks like:

Figure 411 image_1.jpg - the output file from our Im1 class



84.4 Getting data to graph from an SQL database

The second step in our application is to vary the content of the graphic based on the content of a database. As a proof of concept, we're going to write our initial code heavily loaded with constants, and we're going to tell the code to ignore exceptions.

Let's connect to the database:

```
java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/test?user=trainee&password=abc123");
```

We're using MySQL here. It's a free license for testing, it runs easily on our training machines, and we know it pretty well because we run courses on it. MySQL also has the advantage for development of having one of the smaller subsets of SQL implemented; if you get your code running on MySQL, chances are that you can shift to another database engine very easily.

Having established the connection, the next step is to pull back the data we're going to draw. Let's run the query:

```
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
```

We don't know how much data will be returned; we could do a count query to find out and then set up an array, but much better to use an **`ArrayList`**:

```
ArrayList Values = new ArrayList();
while(r.next()) {
        Values.add(r.getString("avtemp")) ;
        }
```

In fact, our code is slightly more complex that this, as we've taken the opportunity to check through our data and find the maximum value so that we can work out how to scale our graph later. Object Orientation purists will shudder at this and suggest that we should have used a separate loop. Maybe; its a question of balance in the design. Here's the code including the hunt for the maximum:

```
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }
```

There's no change from our "Im1" class in how we set up the graphic, and how we draw the first few (constant) rectangles.

When we come to add in the bar chart information, we do so as follows:

```
  for (int k=0; k<Values.size(); k++) {
      int step = 400 / Values.size();
     int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
      g.setColor(Color.yellow);
      g.fillRect(100 + k * step, 300 - high, step, high);
      g.setColor(Color.black);
      g.drawRect(100 + k * step, 300 - high, step, high);
      }
```

The arithmetic here is converting from data (or world) coordinates which are in the range of 0 to the number of elements in the list in x, and from 0 to the maximum data value in the data retrieved in y to pixel numbers, which are in the range 100 to 500 in x and 300 down to 100 in y. Each rectangle is filled in yellow, and then outlined in black. These things are best seen in a diagram:

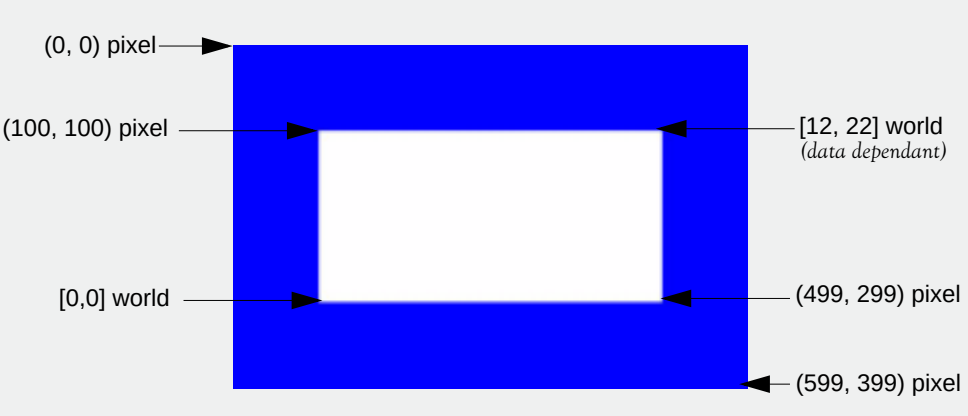
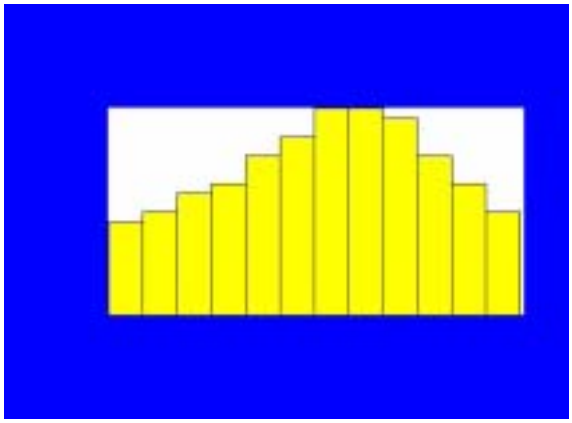


Figure 4.12 Planning out our bar chart – mapping real world data values onto pixel positions

Finally, we save our image to disk just as we did before. Just as we did before, we run our application and we see nothing at the command line, but an image file is output. Here's the data:

```
mysql> select * from lisbon;
+-----------+--------+----+
| month     | avtemp | id |
+-----------+--------+----+
| January   |     10 |  1 |
| February  |     11 |  2 |
| March     |     13 |  3 |
| April     |     14 |  4 |
| May       |     17 |  5 |
| June      |     19 |  6 |
| July      |     22 |  7 |
| August    |     22 |  8 |
| September |     21 |  9 |
| October   |     17 | 10 |
| November  |     14 | 11 |
| December  |     11 | 12 |
+-----------+--------+----+
12 rows in set (0.00 sec)

mysql>
```

Figure 413 image_2.jpg – a bar chart of the data in our SQL database



The complete source code at this stage:

```
// Well House Consultants, 2004.
// Generating an image file from data collected through JDBC

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;

public class Im2  {

public static void main(String[] args) throws Exception

  {

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/
test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }


    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
      int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
        g.setColor(Color.yellow);
```

```
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }
    ImageIO.write(myimg,"jpg",new File("image_2.jpg"));
}
}
```

84.5 Making the graphic available dynamically via the web

In order to make the graphic available via the web, we need to produce it within a servlet[1] and then run that servlet within a Java-enabled web server. For the purpose of this example, we'll deploy our servlet within the Tomcat server. This is the reference implementation of the Servlet definition, so if it works under Tomcat, it's supposed to work anywhere else. Tomcat is part of the Apache project. All the Java parts of the Apache project are now grouped under the Jakarta label. Tomcat uses the Catalina Java Runtime Environment internally, and its HTTP support is provided by Coyote. Sourcing, installation and configuration of these components is beyond the scope of this training module.

What do we change in our source code in order to run in Catalina rather than within the default JRE? Firstly, we have our class extend Servlet, and we write our main code within a **`doGet`** method rather than a main method:

```
    public class Im3 extends HttpServlet {
    public void doGet(HttpServletRequest incoming,
             HttpServletResponse outgoing) {
```

Our servlet needs to set the outgoing MIME type, and also get the output stream that it's to use. We no longer wish to write to a file on the server:

```
    outgoing.setContentType("image/jpeg");
    ServletOutputStream out = outgoing.getOutputStream();
```

The only other substantive change we make is to write our image to the **`OutputStream`**, and add in call to close that stream, thus:

```
    ImageIO.write(myimg,"jpg",out);
    out.close();
```

We've been careful in our demonstration code not to lengthen it with exception handling, but we do now need to a add a little. The servlet container doesn't allow for certain exceptions and we therefore must handle them within our class. Here's the complete code.

```
// Well House Consultants
// Generating an image file from data collected through JDBC
// Server based - Servlet container

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Im3 extends HttpServlet {
public void doGet(HttpServletRequest incoming,
```

[1] we could use a JSP if we preferred

```
    HttpServletResponse outgoing) {

try {
    outgoing.setContentType("image/jpeg");
    ServletOutputStream out = outgoing.getOutputStream();

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/
test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from lisbon");
ArrayList Values = new ArrayList();
int maxval = 0;
while(r.next()) {
        String current;
        Values.add(current = r.getString("avtemp")) ;
        int temp;
        if ((temp = Integer.parseInt(current)) > maxval) maxval = temp;
        }
    RenderedImage myimg = new
BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
      int high = 200 * (Integer.parseInt((String)Values.get(k))) / maxval;
        g.setColor(Color.yellow);
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }

    ImageIO.write(myimg,"jpg",out);

    out.close();
        }
    catch (Exception e) {}
    }
}
```

In order to deploy this as a web application, you'll need to upload the class to the web server, configure your web.xml, etc. We've deployed it within a web application called "towns" which has the following structure:

```
[graham@dhansak towns]$ ls -R
.:
WEB-INF  dbgraphdemo.html  index.html

./WEB-INF:
classes  web.xml

./WEB-INF/classes:
Im3.class  Im4.class  TinyServ.class
```

The directives to link the servlet's URL to the class are within the *web.xml* file. Here are the lines that do it:

```
<web-app>
    <servlet>
    <servlet-name>Im3</servlet-name>
    <servlet-class>Im3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Im3</servlet-name>
    <url-pattern>/im3</url-pattern>
  </servlet-mapping>
</web-app>
```

Here's our results via a browser

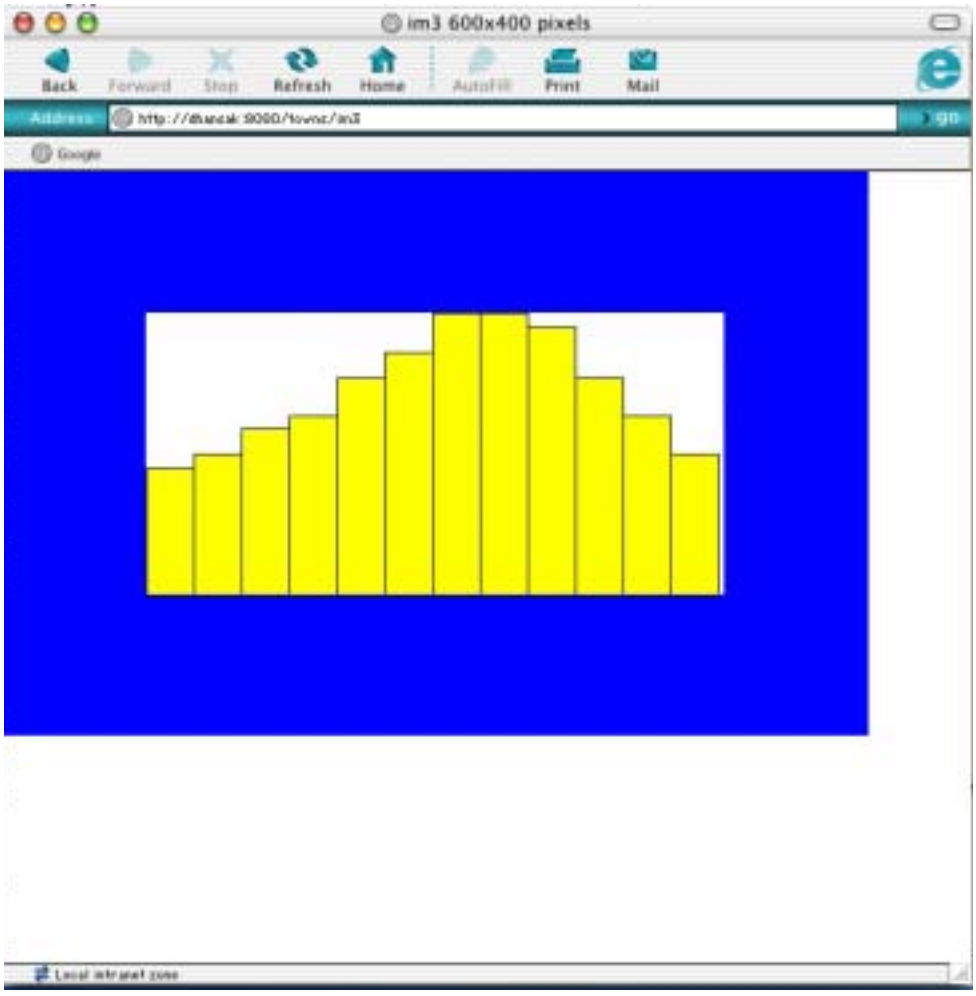


Figure 414 Bar chart of average temperature in Lisbon by month

84.6 Completing our sample web application

So far, we've generated an image on a server computer, we've sourced data from a relational database for that image, and we've provided a Servlet front end for it so that it's accessible via the web.

Let's complete this demonstration by incorporating our graphic in a more complete web application. We'll modify our class so that it can pick up the table name and column to be charted from a form, and we'll incorporate the resulting image within an HTML web page.

Here's the web page source:

```
<head>
<title>Database -> Graph Deployment</title>
</head>
<body bgcolor=white>
<h1>Sunshine on the North Wales Coast</h1>
If you're looking to visit Prestatyn, you'll find there's more sun there in the middle of the year than at
than the start or end. No big shock, but here's a graph to confirm it anyway:<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avsun"><br>
<hr>
This is a demonstration application that uses a Java Servlet to make up a dynamic .jpg file on the
server.
</body>
```

And here are the changes to the Java source. Firstly, we need to collect the name of the table and column to be charted:

```
String WantedTable = incoming.getParameter("table");
String WantedColumn = incoming.getParameter("column");
```

and then use those variables within our database access code:

```
java.sql.ResultSet r = s.executeQuery ("SELECT * from "+WantedTable);
```

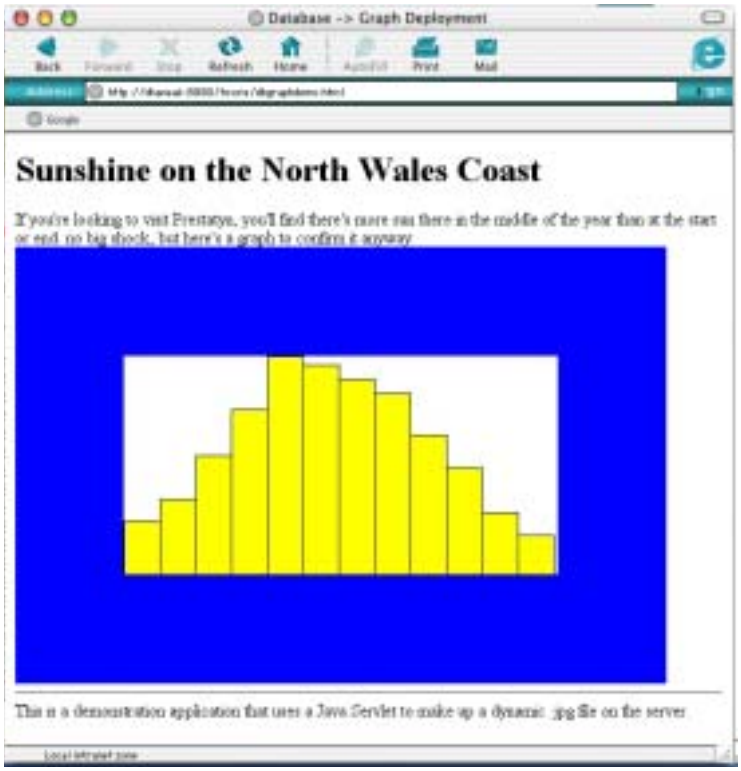
and

```
Values.add(current = r.getString(WantedColumn)) ;
```

That's all the changes we have made!

When we call up the html page, it in turn will call up the image with the parameters set. We get the result that we were aiming at, shown at the start of this module.

Figure 415 Our final result



Here's the complete source code with all the parts integrated:

```
// Well House Consultants
// Generating an image file from data collected through JDBC and HTTP parameters

import java.awt.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Im4 extends HttpServlet {

public void doGet(HttpServletRequest incoming,
    HttpServletResponse outgoing) {

        String WantedTable = incoming.getParameter("table");
        String WantedColumn = incoming.getParameter("column");

try {
    outgoing.setContentType("image/jpeg");
      ServletOutputStream out = outgoing.getOutputStream();

java.sql.Connection conn = null;
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
conn = java.sql.DriverManager.getConnection("jdbc:mysql://dhansak/test?user=trainee&password=abc123");
java.sql.Statement s = conn.createStatement();
java.sql.ResultSet r = s.executeQuery ("SELECT * from "+WantedTable);
ArrayList Values = new ArrayList();
float maxval = 0.0f;
while(r.next()) {
        String current;
        Values.add(current = r.getString(WantedColumn)) ;
        float temp;
        if ((temp = Float.parseFloat(current)) > maxval) maxval = temp;
        }

    RenderedImage myimg = new BufferedImage(600,400,BufferedImage.TYPE_INT_RGB);
    Graphics g = ((BufferedImage)myimg).createGraphics();
    g.setColor(Color.blue);
    g.fillRect(0,0,600,400);
    g.setColor(Color.white);
    g.fillRect(100,100,400,200);
    for (int k=0; k<Values.size(); k++) {
        int step = 400 / Values.size();
        int high = (int)(200.0f * (Float.parseFloat((String)Values.get(k))) / maxval);
        g.setColor(Color.yellow);
        g.fillRect(100 + k * step, 300 - high, step, high);
        g.setColor(Color.black);
        g.drawRect(100 + k * step, 300 - high, step, high);
        }

    ImageIO.write(myimg,"jpg",out);

    out.close();
        }
    catch (Exception e) {}

    }
}
```

84.7 Extending the application

You've seen the basic framework of an application in this module. It's a "how to" integration guide that gives you an overview. For your own application you'll want to provide much more for the user. Here are some areas:

a) You'll want to provide axes and scales on your graphics; the awt has a good set of font handlers, and there are 2D and 3D graphics libraries available too. You could go on to provide multiple graphs on one set of axes for instance
b) You should handle exceptions neatly and correctly. We have elected not to do so to keep the code lengths down.
c) You should structure each element of your application into separate classes so that each part is reusable elsewhere, each part is testable in it's own right, and each part can be maintained by separate staff if necessary
d) You'll want to enhance your HTML, provide forms and much more.

Just to show you how easily the extensions can be started, we've modified just our HTML to use the same servlet to produce multiple graphs

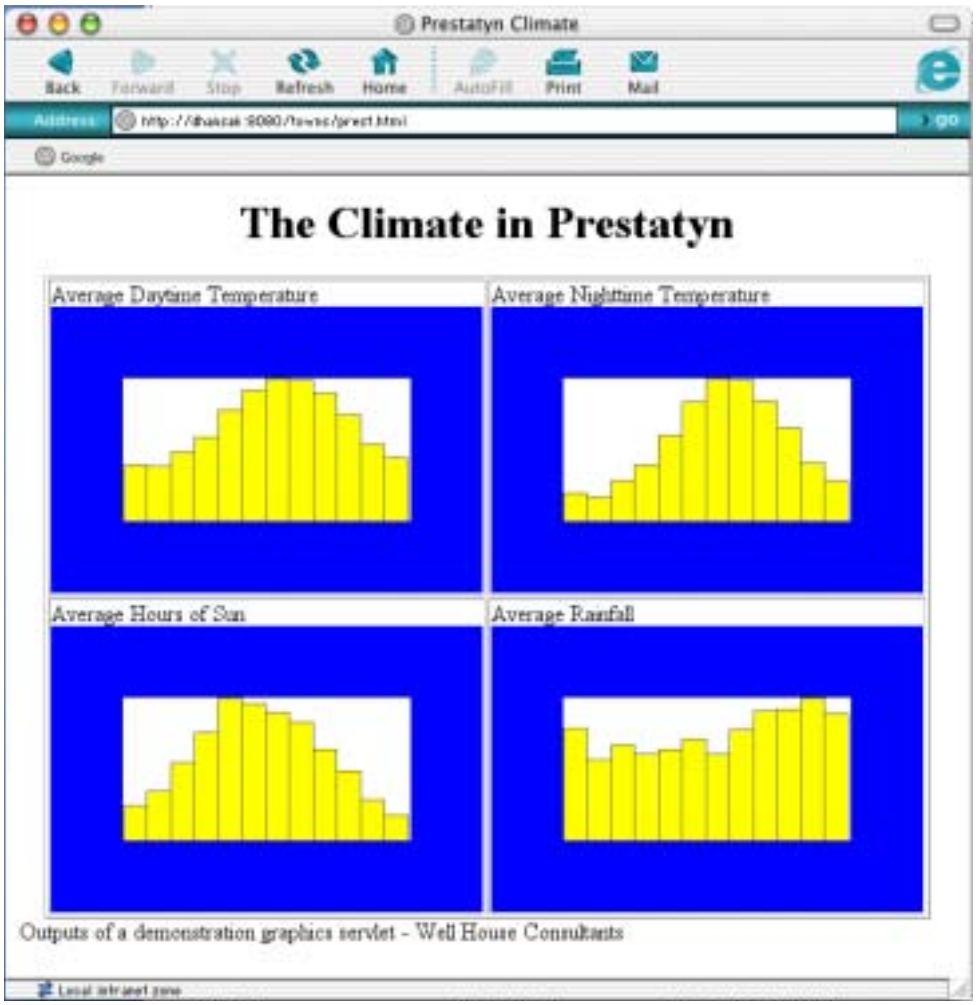


Figure 416 Sun, rain and temperature patterns

and here's the HTML:

```
<html>
<head>
<title>Prestatyn Climate</title>
</head>
<body bgcolor=white>
<center>
<h1>The Climate in Prestatyn</h1>
<table border=1>
<tr>
<td>Average Daytime Temperature<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avtemp" width=300></td>
<td>Average Nighttime Temperature<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avnight" width=300></td>
</tr>
<tr>
<td>Average Hours of Sun<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avsun" width=300></td>
<td>Average Rainfall<br>
<img src="http://dhansak:8080/towns/im4?table=prestatyn&column=avrain" width=300></td></tr>
</table>
</center>
Outputs of a demonstration graphics servlet - Well House Consultants
</body>
</html>
```

Exercise