

# *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

# *Servlets in More Detail*

If you're writing a substantive web application, you're likely to have a number of visitors to your site at the same time, with each of them calling up a series of screens. In this module, we examine parameter handling, session control, and the persistence of servlets from one call to the next.

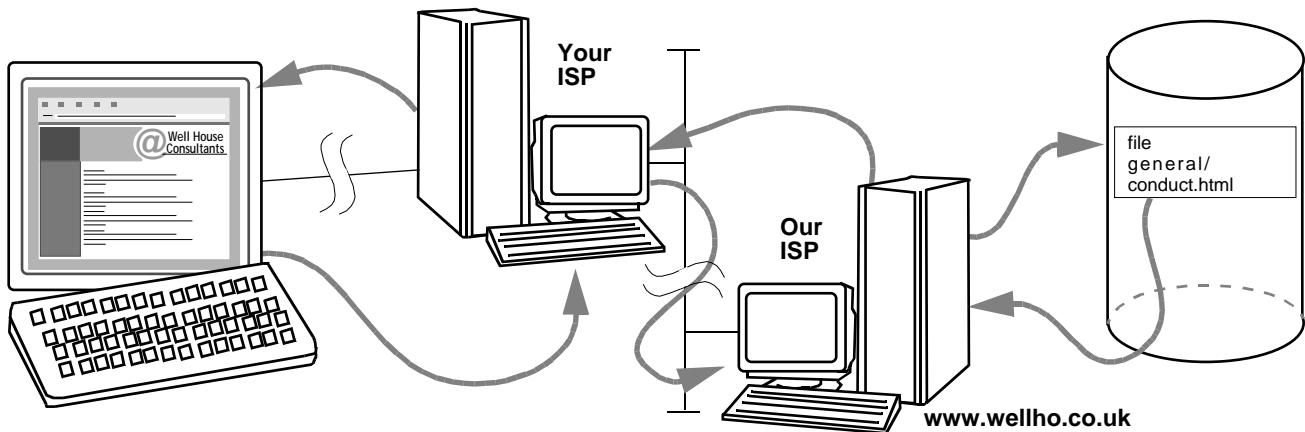
<i>Introduction</i> . . . . .	4
<i>A first servlet.</i> . . . . .	8
<i>Reading from a form</i> . . . . .	10
<i>The life of a servlet</i> . . . . .	11
<i>Maintaining State</i> . . . . .	12
<i>Programming techniques</i> . . . . .	14
<i>Other facilities of servlets.</i> . . . . .	16
<i>Sessions in Servlets</i> . . . . .	17

## 2.1 Introduction

### What is a Servlet?

Servlets are Java classes which run on a Java virtual machine built into a web server, at the request of web users visiting that server. They are one of a number of methods available to web developers to provide server side executable content, such as search engine and central database access capabilities.

Servlets can handle multiple requests concurrently and so they can support co-operative applications such as conferencing.



### How do Servlets fit into the scheme of things?

Figure 1 Using a hypertext link

A Web user<sup>1</sup> selects most pages using a hypertext link (an `a href=` HTML tag) in his web page.

Upon clicking on the link, a connection is made with the named (or current) server, using HTTP (the HyperText Transfer Protocol). The page actually requested is sent up this link, using the specifics of the HTTP protocol.<sup>2</sup>

The Web server takes the complete HTTP request. The majority of such requests are for static pages – documents which are only changed when the author of the Web pages has something new to say.

If the request is such a static one, the Web server retrieves the text from the disk, adds in a number of headers, and returns the page to the browser. The connection is then closed.

If you want to see this activity in action, you can use a telnet program.

Figure 2 Telnet to see demo file

```
seal% telnet magnet 80
Trying 192.168.200.225...
Connected to magnet.
Escape character is '^]'.
GET /tmp/demo HTTP/1.0

HTTP/1.0 200 OK
Date: Sun, 26 Mar 2000 19:48:46 GMT
Server: Apache/1.0.3
Content-type: text/plain
Content-length: 100
Last-modified: Sun, 26 Mar 2000 19:48:24 GMT

This is a demo file provided for the Module number 601 of Well House Consultant's training material
Connection closed by foreign host.
seal%
```

<sup>1</sup> normally running a browser, but Web users might also be robots such as search engines

<sup>2</sup> full protocol details may be found in RFC 2068

The Web might have started as a simple reference resource of static pages, but times have moved on and a significant minority of pages you'll be requesting as a user are now unique. If you visit a search engine, how many other users have entered the keywords "Beluga", "Onions" and "Monroe"? Probably none, so the search engine doesn't have a ready-made page for you.

Of course, the first thing that you need to be able to do is actually tell the Web server that you're interested in Belugas, etc, and so you need a form. The form may be just an ordinary page which is retrieved by your browser and displayed as we saw previously, but it does include a URL to which the completed form must be submitted, and a mechanism through which the user can enter varying data.

When a form has been completed, the user will select a submit button and an HTTP connection will be made to the server using the same protocol as was used for the unchanging form itself. There are, though, two differences:

- a) The page requested from the server will be in a specially named pseudo-directory (or, sometimes, the connection will be on a different port number) to indicate that this is not just a request for a static page, but rather a request to the server to do something more interesting.
- b) Data entered into the form will be included as a part of the HTTP request so that the server has your varying data (Beluga, Onion and Monroe) on which to operate.

For Servlets, the special directory is often a top-level directory called "servlet" although it's possible for systems to be configured differently by the administrator. Indeed, on our courses we usually use "servletrunner" on port 8080. If you run your own server, it's practical for you to hide all of this from the user so that he doesn't know what's a regular page and what's a servlet, or even to have every page on the site be a servlet!

Examples of Servlet URLs:

`http://www.wellho.co.uk/servlet/lookfor`

`http://www.wellho.co.uk:8080/lookfor`

as compared to a "normal" url:

`http://www.wellho.co.uk/examples/lookfor.html`

The big difference when you call up a Web server to run a servlet is that the server doesn't just pass a file back; it runs a program that you have referenced and passes back the results of running that program. That program is the servlet.

As it's running on your Web server computer, the servlet can look up any data that it's authorised to read on that system, including data which is outside the areas normally visible to web visitors. It can also write back (if authorised) to disk files, and even contact other systems on its local network (or the Internet as a whole) for further information. The most common use of this last is for database lookups.

The output from the Web server that's sent back to the user will be received back by his browser, so it needs to be in the form of an HTML page, and you'll find servlets writing out a very wide range of tags. Replies are often tabular, and so you'll find yourself writing a lot of tables; although tables are hard to manage from a text editor, you'll find that they're easy from within a program! Replies also often include further forms; many sites take the user through a series of forms for the more complex requests, or decide what boxes should be on the second form having completed entry of the first form.

Two important things to note:

a) The administrator of the Web server and the author of the servlets both need to be aware of security aspects. Servlets provide a mechanism through which a visitor from the Worldwide Web can run code on your server computer.

b) Unless major security errors are made, the visitor to your site cannot get a copy of your servlet code. All he ever sees is the result of running the servlet. By contrast,

if the visitor calls up an applet he will be sent a copy of the appropriate class file, which is in a published format, and he'll be able to work out to some extent what the applet does and how it does it if he's a geek. HTML pages are even more visible – you only have to be one-tenth geek to read those!

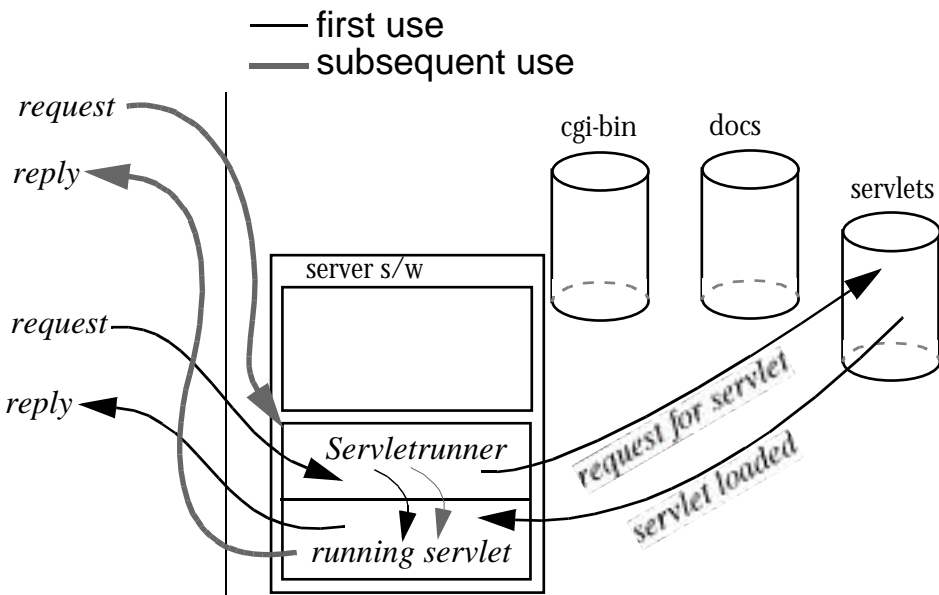


Figure 3 Running a servlet

*What alternatives are there to servlets?*

Lots.

Really, a servlet comprises two parts. The first part is the interface model which connects the incoming HTTP request to the program in some way, and deals with the returning of information. The second part is the program itself, which will be written in some programming language or other.

The most common alternative interface methods are CGI and ASP.

CGI (The Common Gateway Interface) is supported by nearly all Web servers. It provides for requests made to a specified directory to cause a new copy of the specified server side program to be run, and have passed to it all the information that the user entered onto the form, plus a whole raft of additional information such as the name and IP address of the machine making the request, what the previous page called up at that location was, etc. Output from a CGI program takes the form of a short header back to the Web server, followed by the reply page itself which is normally in HTML.

ASP (Active Server Pages) use a different model to CGI and servlets. A request for an "asp" page causes the server to look up files in the document tree on the server (and not a separate program tree). The server then parses the ASP page as it's passed back to the browser; this parsing takes certain tags which tell the server that there's server side executable content and actually runs that code on the server during the parsing, substituting the results into the page.

Servlets are written in the Java language. Using alternative interface methods, though, you have a wide choice of other languages. The most common language used under CGI is Perl<sup>1</sup> and a whole series of other languages are used. Indeed, under CGI any language that can read input from "the keyboard" and "the environment" and can write back to "the screen" can be used. To Perl you can add C, C++, Python, Fortran, Shell languages, etc, etc.

<sup>1</sup> CGI and Perl used together account for more than half of all the server side executable programs written

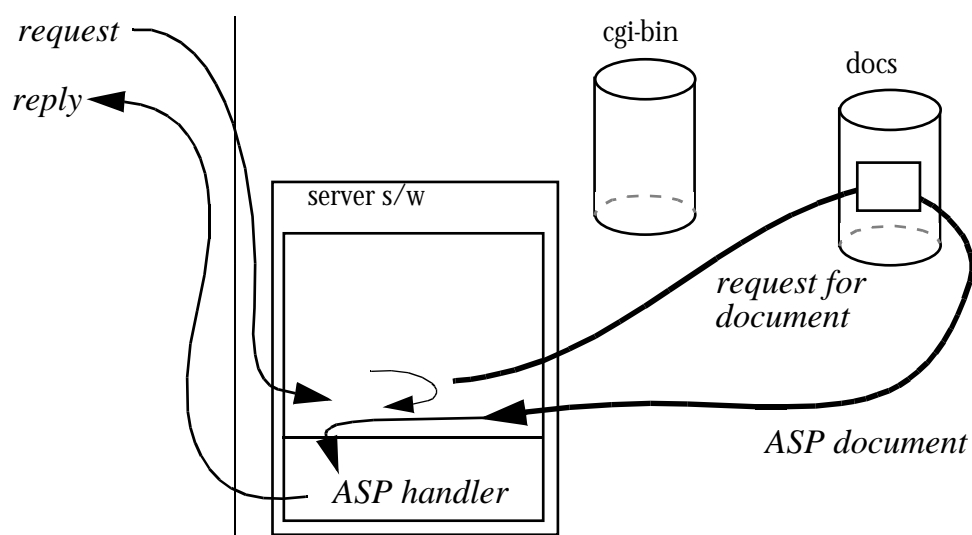
Using ASP, choices are more limited as the languages have to be built in to the ASP server. The two languages that are most visible to new ASP users are VBScript and JavaScript<sup>1</sup> but support is also there for "PerlScript" – a variant of Perl which is an excellent choice for meatier work.

If you like the approach of ASP in that it keeps some of the script within the document, you might also like to consider JSP or Java Server Pages. JSP is actually an additional layer on top of servlets, and is usually implemented using servlets.

Why use servlets and Java? Perhaps because you already know Java, or because your company's standardising on Java and wants to use the same language server and client side. Perhaps because the rich features of Java make it a good language choice for your particular server side task. But the servlet technology does have other advantages too. For example, servlets do stay running between invocations and there's no need to reload each time – one of the problems of using CGI or ASP on a very heavily used site.

You've probably gathered in this section that there's no one ideal server side technology. You may also come across other technologies, each of which have their places, such as modPerl, specialist servers and PHP.

Figure 4 Using ASP



### Servlet Engines

The Web server market is fragmented. There are a lot of web server products out there, ranging from personal Web servers that will run on your PC through to heavy commercial engines.

Of the wide range of servers available, Apache (an open source server) is the most commonly used. It's free, robust, and has a security model that's more than adequate for most users, including ISPs. Within the Apache server, Tomcat (or JServ on an older installation) will be used as your servlet engine.

It's unwise to test new servlet code on your company's production Web server; production and development should be kept well apart. And even if you have a development server, you're likely to want to do basic testing of servlets on your own system; you don't want to be copying files around, you want to be the only member of the development team who can use the very early test versions, etc.

As part of the JDK (Java Development Kit) Sun provide JSDK (the Java Server Development Kit), which includes "servletrunner". Servletrunner is a server in its own right, but it doesn't provide for handling regular pages – just servlets. You'll typically

<sup>1</sup> caution - JavaScript is NOT Java!

run it on your own machine, and it defaults to port 8080. Although it's running on your own machine, there is of course nothing to prevent external connections if you want to let your colleagues run your servlets for testing or see your concept.

### Java Class Structure

Servlets are written by implementing the servlet interface, which includes a number of methods such as `doGet` and `doPost`; requests to whichever servlet engine you're running will then cause the appropriate method to be run.

More commonly, you'll extend the *HttpServlet* class which declares all the necessary methods and leaves you implementing only those which you need. With Java's single inheritance, this alternative is not viable for code which runs as both an Applet and a Servlet since you'll already be subclassing applets.

### Data Interface

Two objects are provided to you by the basic servlet, allowing you to access data from and return data to the user:

- An **HttpServletRequest** object, which includes the initial information sent from the client to the server
- An **HttpServletResponse** object, which you use with the result of your work within the Servlet and which is returned to the user.

## 2.2 A first servlet

Here's a first example of a servlet:

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

public class HelloServlet extends HttpServlet {

    public void doGet(HttpServletRequest incoming,
        HttpServletResponse outgoing)
        throws ServletException, IOException {

        outgoing.setContentType("text/html");

        PrintWriter out = outgoing.getWriter();
        out.println("<html><head><title>Hello!</title></head>");
        out.println("<body bgcolor=\"white\"><h1>Hello Servlet World</h1>");
        out.println("</body></html>");
        out.close();
    }
}
```

Let's look through that in some detail:

```
public void doGet(HttpServletRequest incoming,
    HttpServletResponse outgoing)
    throws ServletException, IOException {
```

**doGet** is the servlet method that's called when you follow a link to a servlet using an "a href=" type link, or if you follow a link from an image map, or if you complete a form using "method=get".

Note that **doGet** is not a static method; you may have multiple instances of the object available if appropriate, and even a simple servlet like this can run multithreaded.

You must catch `ServletExceptions` and `IOExceptions`; since you're now programming across a network this is more important even than it was with file IO – so much can go wrong. Network connections can break, servers and clients may fail to respond, etc.

```
outgoing.setContentType("text/html");
```

If you've written Web pages before, they probably have names ending in `.htm` or `.html`, which is a signal to the Web server that these pages are written in HyperText Markup Language. The Web server will pass on this information (as part of a header) to the browser, which in turn knows how to interpret all the various tags. Other data types are also available, for example `.txt` files will be taken as plain text by the Web server; it also knows what to do with files like `.gif`, `.jpg` and a host of others.

It's not so easy with a servlet. Although most output pages will be written in HTML, some won't be. Servlets could produce plain text, images (that's a natural for Java) and even sound files, and the web server needs to tell the browser. It does so by setting the content type, even if you feel that it's obvious, from the content itself. Note that the content type must be set before any content is written.

```
PrintWriter out = outgoing.getWriter();
```

You need an output stream on which to write your output and this line of code gets you an output stream which is connected to your `HttpServletResponse` object. You can then use your `PrintWriter` object to reply to the client.

```
out.println("<html><head><title>Hello!</title></head>");
out.println("<body bgcolor=\"white\"><h1>Hello Servlet World</h1>");
out.println("</body></html>");
```

These three lines are simply generating the response and will become much more complex in later examples. At the moment, our first servlet does nothing more than send out a flat page to demonstrate the servlet interface to you.

Please note that any errors in the HTML you generate like this will get sent through to the user's browser, which will then do its best to interpret what it has received. Remember that there are a number of versions of HTML, and that you should ensure that you use a subset appropriate to the type and release of browsers that your users will have available.

```
out.close();
```

Flush the output buffers and close the `PrintWriter`. Although you may get away without the close on some systems, you should always provide this line or its equivalent!

To complete this example:

- [i] A web page that calls up the servlet
- [ii] A Screen capture of the servlet in operation]

## 2.3 Reading from a form

`getQueryString`

In order to access the data that a user has entered onto a form, you may use the `getQueryString` method which returns you a raw string of data from the client. You then need to parse that data stream to extract appropriate information.

Forms comprise a number of fields. Each field has a name and a value, and these are encoded into the query string in the form:

```
name=value&name2=value2&name3=value3 . . . . .
```

To parse this data, you'll probably want to use the `stringTokenizer` class, splitting your incoming data at each & character. You'll then split the name from the value at the = sign.

It is quite possible for there to be several fields returned with the same name, so you cannot use a Hashtable to hold the data that's returned without some careful thought. This is especially important if you use "select multiple" input elements.

It's also possible that users may wish to include & and = characters in their input. Oh dear ... this could cause a problem to deciphering programs. Fortunately, the designers of HTTP were aware of the possible problem and took note. Browsers encode certain characters into three character sequences: the first character is a %, and that's followed by two hex digits. It's quite common to see a response string that includes sequences like %2D (which really means an "=").

Characters encoded in this way include: + & = [ ] %

As a further twist, all space characters are encoded into + characters, so that if a user enters

```
Beluga, Onion, Monroe
```

the servlet will see something like

```
lookfor=Beluga,+Onion,+Monroe
```

### *At a higher level*

If you don't want to do your own interpretation from the query string, there are a number of alternative methods available:

<code>getParameter</code>	returns the value (String) of a named parameter
<code>getParameterValues</code>	returns an array of values for a named parameter
<code>getParameterNames</code>	returns any enumerations that will return the names of all parameters

The `ServletRequest` object does not allow you to mix these higher level calls with `getQueryString`.

### *Get v Post*

The original design of the Web was as a tool to provide flat pages of information and, initially, forms were to be kept simple. The default interface used ("the `GET` method") provided up to 1024 bytes of user input (including field names), and that input was echoed back by browsers into the "location" line. Most Web users are familiar with this sort of thing if they visit search engines.

We still use the `GET` method much of the time; it remains the only method available if we use "`a href=`" links (to which we can add our own data after a `?` character if we wish!). It's also the default for forms and it's used by image maps. But we need an alternative too.

The `POST` method can be used as an alternative to the `GET` method to handle input from a form; the 1k limit is swept away, and the "posted" data does not get echoed as part of the URL so it means that passwords and hidden fields can be a little

more secure.

Posted requests are handled by a servlet method called `doPost` (called in the same way as `doGet`). The posted data can be accessed through the same high level methods as we studied under `doGet`, or at the lower level you can:

```
BufferedReader inpost = incoming.getReader();
```

and then read in raw text, or

```
ServletInputStream instr = incoming.getInputStream();
```

if you're expecting binary data.

General advice is to use the **POST** method in preference to the **GET** method for forms, and you should normally use the higher level access methods. There will be exceptions, for example, you may be using a servlet to upload files from a client. I myself upload images to my server in this way rather than through FTP.

Although rarely used, HTTP and servlets also support **PUT** and **DELETE** methods, allowing data to be put onto the server, and allowing server files to be deleted. Just from this short description, you'll understand why they're not often provided for the general public to use on the World Wide Web!

## 2.4 The life of a servlet

The simple servlets that we've looked at so far have simply run, but you may have noticed during practicals that the first use of each servlet seemed slower than subsequent uses, or that the `servletrunner` log contains differing information.

When a servlet is first called, the server loads it and runs the servlet's `init` method. The running of this method will be completed before any of the service methods are started.

Once a servlet has been initialised, service methods such as `doGet` and `doPost` are run as required.

When all service methods have been completed, and/or after a certain timeout, your servlet may be destroyed (using the `destroy` method).

### *Initialisation*

The `init` method in the base class may provide all you need for a simple servlet, but you'll wish to extend it for more complex servlets. You might want to establish connections to databases (and throw errors if they're not available), populate hash tables (if your servlet provides a translation facility), etc. You do still need to call the `init` method in the base class, so a typical `init` method may look like:

```
public void init(ServletConfig cf)
    throws ServletException {
    super.init (cf)
    // Your initialisation code comes here; e.g.
    dclookup = new hashtable();
    // then code to load the hashtable from disk?
}
```

A further method called `getInitParameter` is also available. It takes a string as its parameter and returns a string, but its implementation beyond that is server specific. You might use it to initialise a servlet to provide a hashed lookup to a specific text file, but remember that you'll be sacrificing portability if you use the facility.

### *Destruction*

The `destroy` method in the base class is all that's needed in "simple" cases, but if you've added resource in during your `init` method (or during subsequent methods), you'll want to release it here.

```
public void destroy() {
    dclookup = null;
}
```

If there might be long-running operations in process at **destroy** time (for example, if your **init** process started additional threads), you need to keep track of these and service thread requests, and provide a clean shutdown facility. This is something of a specialist topic, beyond what's needed by most servlet authors.

### *Example procedure - take an application of ours, modify, re-upload*

You may be feeling a bit overwhelmed by all the files, changes, modifications around. Here's a sample procedure you can follow for practise: Download and rename a class file (and make a minor change to the code) to create a new web application.

```
1. ftp 192.168.200.190
log in as trainee, password abc123
2. cd /usr/local/tomcat/webapps
3. cd octj/WEB-INF/classes
4. get Tempconv.java
5. Quit
6. Edit and save as [yourname.java]
repeat 1 2 3
7. put [yourname.java]
8. quit
9. telnet 192.168.200.190 (trainee, abc123)
10. repeat 2 and 3
11. javac [yourname.java]
12. cd ..
13. modify the file web.xml to refelect the new servlet
14. Restart the web app via the tomcat manager
15. exit
16 browse via http://192.168.200.190/octj/[yourname]
```

If you are not familiar with Tomcat configuration, the tutor will assist you with steps 12 to 14.

## 2.5 Maintaining State

When you visit your local supermarket, you don't have to wait for their previous client to leave the store before you can enter. And you'll want to have the same flexibility with your servlets.

Imagine that I'm using a servlet to order my groceries. I'll be completing a number of forms and calling on the server to "**doGet**" or "**doPost**" from time-to-time. But most of the time, I'll be filling in the next form, selecting whether I want Cheddar or Cheshire, walking over to the bathroom to see how much soap we have in the cupboard, etc.

When I do come back, I want to carry on with my own order, even though I'm starting a fresh run of (probably) **doPost**. I certainly don't want to rake over the order of another more recent arrival in the supermarket, who's organising a party and who has ordered 48 cans of beer and loads of nibbles! (Nor does he want my soap and cheese.)

We need sessions – or "shopping carts" – and we need to keep track of who is using which. There are a number of methods of doing this.

## Session Objects

An `HttpSession` object is part of the `HttpServletRequest` that's passed in to the `doGet` or `doPost` methods. The `getSession` method returns you the actual session object:

```
HttpSession who = in.getSession(true);
```

(the boolean parameter being set to *true* tells the servletrunner to create a new session if one doesn't exist).

Once created, sessions have unique IDs which can be used for user tracking.

Within a session object, you'll typically want to manage an object of type `ShoppingCart`; you can do so by setting up a shopping cart and associating it with the session object. This snippet of code opens up a session (creating a new one if necessary), and assigns a shopping cart to that session (once again, creating it if necessary). Data is then added to that shopping cart from a parameter on the form.

```
HttpSession who = in.getSession(true);
ShoppingCart trolley = (ShoppingCart)
    who.getValue(who.getId());
if (trolley == null) {
    trolley = new ShoppingCart();
    who.putValue(who.getId(), trolley);
}
String cheese = in.getParameter("flavour");
String howmuch = in.getParameter("weight");
trolley.add(cheese,howmuch);
```

Once you've finished with a session (for example, after the user has visited the checkout in your store), you'll want to clear out the session. You can use the `invalidate` method to do this. Some servers automatically invalidate sessions after a certain time, others may need to be explicitly cleared.

Session objects provide a useful and convenient way of tracking users, but they won't always work for you in the manner that we've shown above. Internally, the "Cookie" facility of browsers is used, and some users will not accept cookies, or are running mature browsers that don't support cookies; you need to use an alternative method for sites that are required to support such users.

### *Rewriting URLs*

If cookies aren't available, you may use URL rewriting. This technique (not supported by `ServletRunner`) requires you to encode your session ID within links to subsequent pages. If the user fills in a form or follows a link based on the re-written URL, the servlet recognises the session ID and uses it to retrieve the appropriate `HttpSession` object.

### *Hidden fields*

If you have a series of forms following on from one another, this one will work no matter what the browser or the server, and whether or not the user is accepting cookies. Personally, I use it all the time for some quite major Web applications, with users visiting from Internet Explorer, Netscape, AOL, Compuserve and others.

The scheme is similar to URL re-writing ...

As a user enters the site (calls up the servlet for the first time), he can be clearly identified because the form he's completed does not include a field called (say) "missionid". This is the trigger for a SessionID and shopping cart to be created for him, and might also be the trigger to validate our user, his password, etc, before letting him proceed further into the site.

All subsequent pages sent out to the user during his session include a hidden field, named "missionid", and a unique reference to the SessionID (use the `toString` method) and allow us to identify who the user is when he comes back to us.

Whilst it is convenient for us to use the session facility provided in servlets to provide this capability, for more secure sites we've provided our own facility which includes a "channel hopping" type capability. The contents of the `missionid` field change between each and every page which blocks meaningful backtracking and disables anyone walking up to the machine later from entering the system through intermediate cached pages.

## 2.6 Programming techniques

This subsection isn't servlet specific – it describes techniques and tips that you'll want to use whether you're writing Java Servlets or scripts in Perl to run via CGI.

### *Webifying output*

You're generating response pages from within your program, and those response pages are in HTML. Your code may look like this:

```
PrintWriter out = outgoing.getWriter();
out.println("<html><head><title>Hello!</title></head>");
out.println("<body bgcolor=\"white\"><h1>");
out.println(heading);
out.println("</h1>");
out.println(boding);
out.println("</body></html>");
out.close();
```

where **heading** and **boding** are strings containing the title and text of your response page.

It may work for you. Or it may not.

Is there any chance of the **boding** variable containing a "<" character? If so, will it be an HTML tag, or is it an unfortunate co-incidence? Might you be displaying a piece of program such as:

```
if ( j3 < h4 ) {....
```

Oops ... looks like a tag!

You need to encode characters:

& to **&amp;**

< to **&lt;**

at the very least!

You should also be aware of line breaks and white space on your output. Take a carefully formatted report<sup>1</sup> and the report looks good. Include it within a web page and it's a disaster! Multiple spaces become single spaces, lines are folded and the web page is of no practical use.

Solutions? A quick and dirty solution is to use the **<pre>** and **</pre>** tag pair in your HTML; this stands for "preformatted" and means that white spaces are not squished. It also flips the display into a fixed width font for that section of the text so that columns are maintained.

If you're working on a public visible, prestige page, then **<pre>** is too crude a tool, and you'll want to format your data into tables. A lot of work, but of course in Java you can write a class just once and use it in many applications; you might also consider Swing.

### *Keeping code and pages separate*

Good Java programmers are hard to come by and expensive, but they're not necessarily the world's best graphic artists. So who's going to write and maintain the user interface within your servlet application?

Programmer? Graphic specialist? Which of them understands the code?

Much better to use separate files and a merging capability. Cascaded style sheets may help, or you may choose to write a template of each type of response page in HTML and have your Java servlet search in the page (just once in `init`?) and then complete it with different values each time the servlet requires it. This is a scheme we use in our own Web page work – the page designer writes HTML into which keys such as `%name%` are inserted ... that one represents where a person's name is to be substituted. Page design can be done independently of the servlet code; the designer may temporarily make a few substitutions to check that the page looks OK with longer names, but that's the limit of any compromise.

Similarly, servlets can be designed without the need to be working around huge amounts of HTML embedded within the Java code. This is one of the major design intents of JSP, which allows you to provide a document which makes calls to Java classes in order for it to "complete the blanks".

### *State Diagrams*

You need to design and maintain the flow of your users through the various pages served by your servlet(s).

<sup>1</sup> run a `dir` command on DOS or an `ls -l` on Unix or Linux

## 2.7 Other facilities of servlets

### *Multiuser Servlets*

Typically, servlets are capable of handling multiple client requests concurrently. If your servlet modifies data on the server (or in a database ...), you'll need to consider the question "What if several users are running me at the same time?".

In some circumstances, the answer is to make use of the threading capabilities of Java, and to synchronize blocks and methods to objects. These are excellent facilities of Java and allow you to write a superb, heavy-traffic capable servlet, but they do involve you in considerable additional thought and testing (and some extra programming) as you develop your servlet.

If you declare your class:

```
implements SingleThreadModel
```

then your server will only run one service method (**doGet**, **doPost**) instance at a time.

A word of warning: Although only one **doGet** or **doPost** can be running at a time, the single thread model does not block out all other users while user #1 is between requests (i.e. while he's completing the next form), so the servlet programmer may not leave data lying around and pick it up next time through without considering who's who. This makes common sense when you think of it; users will often leave a site in the middle of a sequence of forms and you can't afford to have your site locked out.

### *Servlet Descriptions*

The **HttpServlet** class, which you've been extending to provide your own servlets, includes a method **getServletInfo()** which returns a string describing the servlet. By default it's null.

There are a number of tools available which get descriptive information from servlets and can display it; if you're writing a number of servlets, it might be worthwhile including such a method in your template:

```
public String getServletInfo() {
    String Author = "Well House Consultants";
    String Version = "V1.2";
    String Name = "Servdemo";
    String Desc = "A Demonstration Servlet";

    String say = "Author: "+Author+"\n";
    say = say + "Version: "+Version+"\n";
    say = say + "Name: "+Name+"\n";
    say = say + "Description: "+Desc;

    return say;
}
```

(Yes, I know I should have used a StringBuffer!!)

## Cookies

As well as their use in tracking sessions, the servlet interface includes a number of other methods useful for the setting, retrieving and examination of cookies.

### 2.8 Sessions in Servlets

The following examples show the newer session tracking API in use within Servlets.

The first time a user runs the "Barman" servlet, it sets up a session for him and prompts for his name. On subsequent visits, it addresses him by name straight away, and also keeps a tally of how many times he's visited.

The "Landlord" servlet does the same as the Barman, but in addition it maintains a static vector of sessions so that it can report on everyone in the bar and how much they've had to drink.

Note: "Barman" keeps the users apart, which is typically what you would do with a shopping cart ... "Landlord" provides a management information service too ...

```

::::::::::::::::::
Barman.java
::::::::::::::::::
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Simple Session Tracking
 */
public class Barman extends HttpServlet {

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

        HttpSession session = request.getSession(true);
        Integer count = (Integer)session.getAttribute("mycounter");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        if (count == null) {
            count = new Integer(0);
            out.print("<h1>Welcome. Please enter your name</h1>");
            out.print("<form action=/nan/servlet/Barman>");
            out.print("<input name=whoyouare>");
            out.print("</form>");
        } else {
            String wanted = request.getParameter("whoyouare");
            if (wanted != null) {
                session.setAttribute("who",wanted);
            } else {
                wanted = (String)session.getAttribute("who");
            }
        }
    }
}

```

```

        count = new Integer(count.intValue() + 1);
        out.print("<h1>Welcome back "+wanted+"</h1>");
        out.println("This is your visit no. "+count+"<br>");
    }
    session.setAttribute("mycounter",count);
    out.println("</body>");
    out.println("</html>");
}
}

```

```

::::::::::::::::::

```

```

Landlord.java

```

```

::::::::::::::::::

```

```

import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Simple Session Tracking
 * Also reports on all users
 */

public class Landlord extends HttpServlet {
    public static Vector People;

    public void init(ServletConfig cc) throws ServletException {
        super.init(cc);
        People = new Vector();
    }

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

        HttpSession session = request.getSession(true);
        Integer count = (Integer)session.getAttribute("mycounter");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body bgcolor=\"white\">");

        if (count == null) {
            count = new Integer(0);
            out.print("<h1>Welcome. Please enter your name</h1>");
            out.print("<form action=/nan/servlet/Landlord>");
            out.print("<input name=whoyouare>");
            out.print("</form>");
            People.addElement(session);
        } else {
            String wanted = request.getParameter("whoyouare");
            if (wanted != null) {
                session.setAttribute("who",wanted);
            } else {
                wanted = (String)session.getAttribute("who");
            }
        }
    }
}

```

```
count = new Integer(count.intValue() + 1);
out.print("<h1>Welcome back "+wanted+"</h1>");
out.println("This is your visit no. "+count+"<BR>");
out.println("<BR><H2>Those here present ...</h2>");

for (int i=0; i<People.size();i++) {
    HttpSession present = (HttpSession)(People.elementAt(i));
    String ere = (String)present.getAttribute("who");
    out.print(ere+ " is here and has drunk ");
    int ii = ((Integer)present.getAttribute("mycounter")).intValue();
    out.print(" " + ii + " previously<br>");
}
session.setAttribute("mycounter",count);
out.println("</body>");
out.println("</html>");
}
}
```



**Exercise**

---

# *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.*