

19

Debugging JSP and Servlets

Debugging web-based applications using JSP and Servlets is hard work – harder, in many ways, than debugging other Java-based systems. Why?

- ❑ The separation between the client (typically a web browser such as Internet Explorer) and our server-based code causes newcomers conceptual difficulties, something not assisted by 'helpful' browser caching (HTTP headers related to caching are discussed in Appendix F).
- ❑ The inherent concurrency of a web application leads to many potential problems with synchronization and thread-safety.
- ❑ Java Web containers and application servers provide their very own directory structures and class-loading rules, which can be extremely confusing at first. This also makes using a conventional debugger such as `jdb` hard, as you in effect have to debug the server itself.
- ❑ With JSP in particular, there's a lot going on 'under the surface'. Despite the best intentions of JSP's designers, a thorough understanding of both Servlets and the JSP execution model (discussed in Chapters 3 and 4) is essential for effective debugging.
- ❑ Intermixing of various languages (JSP, Java, HTML, JavaScript) and their syntax and escaping rules can make reading and understanding the code very tricky.

The key to making it easy to debug your web applications is to design them well in the first place. A well-designed application (as described in Chapter 17) will be divided into a clearly demarcated multi-tier architecture. Because this will mean it has distinct, easily testable components, the application will avoid many of the potential pitfalls.

This chapter will not give you a simple, foolproof method of debugging your code, because there's no such thing. Debugging is as much a mental issue as a strict computational one. A bug is a part of a program that doesn't exactly do what it's supposed to – an error in our mapping between the system's requirements and its implementation. In a strict sense, debugging can be seen as locating and fixing the 'bad' code, but will only be truly effective when your application has been carefully designed in a modular and testable manner.

The easiest bug to fix is the one that never existed, and a lot of debugging frustration can be saved by taking more care and time over the design and development of the application. Chapter 17 discusses best practices in writing well-structured, maintainable (and hence debuggable) web applications with the Java code where it belongs, in JavaBeans and Servlets.

In this chapter we'll look in turn at:

- ❑ Understanding error messages you receive when running JSP pages
- ❑ Tools and techniques for debugging our web applications
- ❑ Ways of using debuggers with Java web applications
- ❑ Common problems that occur, and how we can address them

Debugging Techniques and Tools

Let's start by looking at some of the more common techniques that are used when debugging the web tier.

Understanding JSP Errors

Since JSP is in fact a layer on top of Servlets and combines different languages, bugs can come from any one of these layers and in any language. The first step in debugging is to understand the error, get an idea where it comes from, and know where to look for it.

Compilation Errors

The first errors you typically encounter are compilation errors. When you request a page for the first time, the JSP is first transformed into a Servlet, which is compiled to a `.class` file and used by the web container to create the page. In general, compiling the `.jsp` file into a Servlet doesn't often go wrong. When it does, the error message can be recognized in Tomcat 4.0 since it throws an `org.apache.jasper.compiler.ParseException` referring to the original `.jsp` file.

For example, if we start a page with:

```
<%@ page impot="java.sql.*" %>
```

(misspelling `impot` instead of `import`), we get this error message from Tomcat 4.0:



The error reporting comes from the JSP engine (Jasper is Tomcat's JSP engine) and can be different for other implementations. As you can see, the error message includes the filename, the line and column number at which the error was found (0, 0 in this case), and some more information about the type of error; other JSP engines may differ in the information they provide.

Most compilation errors come from the second stage, where the Java compiler transforms the .java file into a .classfile. In this case, the error report comes from the Java compiler. Since that compiler doesn't know anything about JSP, but is just compiling the automatically generated Servlet code, interpreting this error report is much more difficult.

How do you know it's a compilation error?

- It doesn't work at all; you only get a bunch of errors in your browser.
- There will probably be some information about the compilation error (see screenshot below).
- If you check the work directory the JSP engine uses to put its .java and .class files, you'll notice that it did generate a new .java file but no new .class file.
- Some containers allow you to compile the JSP off-line (like Tomcat's `jspc` command). This can also be used to trap compilation errors before putting the files on a server.

For example, if we include a line of the form:

```
<%= results %>
```

when the scripting variable `results` is not defined, the error in Tomcat 4.0 looks like this:



The error reporting shows the type of error, line number, and a piece of code where the error is located; but here the line number refers to the line in the generated Servlet; you'll have to go and dig out that code from the server's work directory to figure out where the error is in your JSP. Note that at present Tomcat conveniently hides the useful bit of the error message (Undefined variable: results) off the screen. Hopefully the formatting of the error messages will improve soon.

The most common compilation errors are just small Java syntax mistakes like missing semicolons, non-matching quotes, or typos in variable or class names, but sometimes it isn't easy to understand the messages. The following table gives you some examples of error messages encountered with Tomcat 4.0 and what they really mean. Again, this isn't always the same for every setup and can vary depending on which JSP engine and java compiler you use. I found that the JServ error reporting is easier to read than Tomcat, for example:

Java compiler error	Possible causes	Example
)' expected	When you put a ';' at the end of an expression.	<%=getValue(i);%> is translated into out.println(getvalue(i));
try without catch, or catch without try, with line numbers pointing to the end of the file	Missing { or }. The JSP engine puts all your code from the JSP page in one big try-catch block. When the curly brackets in your code are not balanced, this error is thrown.	try{ ... your code ... } catch (exception e) { ... }
useBean tag must begin and end in the same physical file	The / at the end of the <jsp:useBean> tag is missing.	<jsp:useBean id="theID" class="theClass" >

If it really isn't clear what the error means, it can be good to look at the generated `.java` file. (Some JSP engines require you to explicitly enable an option to retain the generated source files.) The directory where these files are stored depends on the JSP engine; for Tomcat, look inside the subdirectories of the `<CATALINA_HOME>\work` directory (`<TOMCAT_HOME>\work` on Tomcat 3.x). Overall, system generated code is difficult to read, but most JSP engines produce code that's easy to understand. Normally you have a standard header that stays more or less the same for all generated servlets. In the middle you find your entire HTML code translated into strings that are printed to the `out` stream. Scriptlets are just copied.

Some JSP engines (like that from ServletExec) don't copy the HTML code but use a function to extract the HTML portions from the original JSP file. Luckily, they also add comments to specify the link between the generated servlet code and the position in the JSP file.

When you start using a newly installed JSP setup or when you use new packages, a 'class not found' compilation error is very likely a missing `import` statement or a class loading problem. These are easy to detect, but it's not always easy to solve them. Whilst standardization is improving, every Servlet engine has its own particular way for you to specify classes to load; we'll return to this shortly.

Runtime Errors

When your code makes it through the compiler, the Servlet is run and, if you're unlucky, the runtime errors start to appear. The first type of runtime error is where an exception is thrown. Runtime errors aren't always 'all-or-nothing' like compilation errors. An error occurring might depend on the input parameters, the value of some variables, the user that's logged in, etc. If you specified an `errorPage`, the request is redirected to that page; otherwise, you get the JSP engine's default error. Normally this is a printout of the error type (`NullPointerException` being one of my favorites), error message, and a stack trace.

If the stacktrace shows '(Compiled code)' instead of line numbers, you should try to start your Servlet engine with JIT compilation disabled. You can do this by specifying `-Djava.compiler=NONE` as a start option for the web container.

We said that for compilation errors the line number might not be that informative, but with runtime errors it's about the only information shown, and in this case it almost always refers to the underlying Servlet and not to the JSP source.

We already mentioned that your JSP code is put inside one big `try - catch` block in the Servlet. Since this block catches `Exception`, we never get the compilation error about not catching a particular type of exception. This `try - catch` block catches all the `RuntimeExceptions` and the `Exceptions` we should have caught in our code like `SQLException`. When an exception is thrown, the output that was not yet flushed will be cleared; in most cases, this means that we get an empty page with the error message but hardly any information about whether the first part of the code did work out fine. There are two possibilities:

- ❑ Insert `out.flush()`; after crucial places in your code: everything before the `flush()` is sent to the browser. This might very likely result in a bad HTML page, but it helps locating the error.
- ❑ Temporarily enclose your entire `.jsp` file in a `try - catch` block yourself. Looking at the resulting HTML code gives you a very precise hint on where to look for the bug.

One particularly unintuitive message produced by Tomcat is `java.lang.IllegalStateException: Response has already been committed`. At face value, this means that an attempt has been made to forward the request after at least a partial response has been sent to the client; this is illegal, hence this error. What is less intuitive is that with some versions this error occurs when an exception within the page causes an error page to be accessed.

If the response to the client has already been 'committed' (in other words, the output buffer has been flushed) when the exception occurs, the attempt to forward control to the error page will fail in this manner. This often happens when an exception is thrown after a `<jsp:include>` action with `flush="true"` has been performed.

Another type of runtime error is where there are no exceptions thrown and, from a Java point of view, everything is working fine. However, the resulting file is not what it should be. It can be wrong data that's being displayed or incorrect HTML formatting. It is particularly for these types of errors that the debugging techniques we'll explain later can be helpful.

HTML and JavaScript Errors

When part of the HTML formatting or even JavaScript functions are dynamically generated, this can easily result in incorrect HTML pages. Some browsers are more susceptible than others to these errors, so you might not immediately notice that there is a problem. In most cases, the errors can be located by comparing the generated output to a static HTML page that has correct formatting.

There are several utilities available that can check if your HTML syntax is correct. Some, like NetMechanics (<http://www.netmechanics.com>), even allow you to have your page checked online. In general, these problems are not difficult but they might require some concentration for a more complex page. If you dynamically create JavaScript the complexity increases.

Other tools like SiteScope (<http://www.freshwater.com>) allow you to periodically request a specific URL and alert you if an error occurs or if something is missing on the page. Normally these tools are used to guarantee that a server is still running, but they could be just as well be used to find out if a certain JSP is still displaying the correct content.

Debugging Techniques for the Web Tier

So, our JSP runs after a fashion, but we're getting the wrong result or an exception is being thrown somewhere. What can be done to isolate the problem? Running a conventional debugger is certainly an option that we'll look at presently, but there are other, simpler approaches to take.

Eyeballing

As simple as it may seem, eyeballing or code walkthrough – just looking at the code – is one of the most important and most used techniques for finding bugs. In fact eyeballing is almost always part of the total debugging process. It happens quite often that we're using JDB-type debugging, and while stepping through the code we suddenly see the bug in one of the lines that is not processed yet.

This mechanism can be extended to what we may call 'peer eyeballing', the ability of someone not involved in the code development to instantly pick out an error you have been spending hours trying to find. We can become so consumed by the project that we can develop a forest-and-trees syndrome that only a fresh set of eyeballs can overcome. Formal inspection of the code can be a highly effective technique for complex or error-prone code.

In order to enhance the visual inspection, we should keep our code clean and adhere to good standards and practices whether widely accepted good practices or those set by our organization. (I can recommend *Essential Java Style* by Jeff Langr, from Prentice Hall, ISBN:0-13-085086-1.) Here are a few guidelines:

- ❑ Keep methods small and clear.
- ❑ Try to make the code self-explanatory. Use intuitive variable and method names.
- ❑ Use proper indentation. Whatever style we use for aligning braces, be consistent, and use plenty of whitespace. (Indenting JSP code neatly is particularly difficult.)
- ❑ A nice list of how not to do it can be found at <http://mindprod.com/unmain.html>.

The success of eyeballing depends very much on the expertise of our developer and how familiar they are with the code.

Although it appears that this is just a technique, there are also tools that can make the eyeballing easier. A good code editor has features like syntax coloring, bookmarks, and commands for locating matching braces. Use these options and take some time to learn or install the necessary shortcuts. (Unfortunately, the combination of HTML, JSP tags, Java code, and client-side JavaScript may well be too much for the syntax highlighter of your favorite editor.)

System.out.println()

Using the standard output or error stream is an obvious and often used debugging technique. It doesn't look very sophisticated at first, but its simplicity is also its power; it is often much simpler to insert a `System.out.println()` statement than to go through the rigmarole of starting up a debugger.

In most server components, it's very unlikely that the standard output will be used in the core processing of the object. This leaves it available for outputting debug messages:

- ❑ Since the `System` object is part of the core Java objects, it can be used everywhere without the need to install any extra classes. This includes Servlets, JSP, RMI, EJB's, ordinary Beans and classes, and standalone applications.
- ❑ Compared to stopping at breakpoints, writing to `System.out` doesn't interfere much with the normal execution flow of the application, which makes it very valuable when timing is crucial; for example, when debugging concurrency issues, or when stopping at a breakpoint could cause a timeout in other components.

`System.out.println()` is easy to use as a marker to test whether a certain piece of code is being executed or not. Of course, we can print out variable values as well, but then its inflexibility becomes more apparent. I personally use it mostly with `String` constants such as `System.out.println("debug: end of calculate method")`. If we print out variables appended to a `String` message like in `System.out.println("debug: loop counter=" + i)`; frequently, we're using a relatively expensive string concatenation just for debugging. Since the debugging overhead should be as low as possible, it might be better to use two statements if we are concerned about this:

```
System.out.print("debug: loop counter=");  
System.out.println(i);
```

The most important problem is that it isn't always clear where the information is printed. Depending on the application server, our messages might be written to a file, a terminal window on the server, or they might just vanish altogether. Later in the chapter, we'll look at a monitoring tool that can be used to get around this problem.

As an alternative to printing to `System.out`, debugging information of this sort can be directed into the JSP output stream itself using `out.println()` in a scriptlet. Another useful technique is to dump all the information available to the JSP – request parameters, cookies, session ID, etc. – in this way. Tomcat 4.0 also provides a 'request dumper' facility (enabled via a setting in `server.xml`) that provides similar functionality.

Logging

The Servlet API provides a simple way of outputting information by using the `log()` method. Both `GenericServlet` and `ServletContext` provide `log()` methods, the `GenericServlet` method prepending the Servlet's name to the log entry. In both cases, `log()` takes either a single `String` argument or a `String` and a `Throwable`. The file to which the logging information is sent depends on the Servlet container you're using and on its configuration.

While we can use this function to record debug information, strictly speaking the log isn't meant for this type of debugging purposes. The log should really be used to keep an eye on what happens in a production environment – when you start debugging, making small changes and requesting the pages again and again, it's not very convenient to keep opening the log files and scrolling to the bottom every time to see what happened.

The log files do give an indication of new emerging bugs or the frequency of problems. For that reason it's good to use the `log()` function in the `catch` clause of exceptions which should normally not occur.

Home-Made Logging Helper Classes

Creating your own `Debug` class, like in Alan R. Williamson's *Java Servlets By Example*, (Manning Publications, ISBN 188477766X) is a more flexible approach. A singleton class, or a class with only static functions, can be easily called from anywhere in your code. You can direct the output to whatever suits you best and there's the possibility to have a central switch to start and stop debugging mode.

The debug information can be rather easily directed to:

- A file
- A console-like window
- A network socket
- A buffer in memory
- The `out` `PrintWriter`

To do this you need to create your own `Debug` class that has an outline like the following:

```
public final class Debug {  
    private Debug() {}  
  
    public static void println(String logValue) {
```

```

        // put the code to output the
        // logValue to whatever you like here
        // (keep in mind that you can only use static class variables)
    }
}

```

The nice thing about the static `println()` method is that sending debug information is as easy as `Debug.println("my debug info");` and outputting information is not restricted to the Servlet environment, but is also possible in other classes and JavaBeans you want to use in your JSP pages.

Since you create this class yourself, you can keep extending the possibilities as you need them, such as:

- Printing the date and time, and session ID, when debugging in a situation where multiple clients are sending requests
- Printing out the stack trace when needed
- Dump variables of a specific bean by using introspection
- Using a threshold level in order to use different debug levels
- Showing which class/method the debugging info came from

A minor disadvantage of this technique is that it creates a tight coupling between all your code and the `Debug` class itself, which may limit the portability of your JavaBeans.

Instead of creating your own logging classes, you may prefer to use a ready-made logging library. The Apache Jakarta project's `log4j` is an open-source logging library that allows logging behavior to be controlled by editing a configuration file, without any need to recompile. Logging messages can be categorized, and categories organized in a hierarchy, allowing fine control of which log messages are output. See <http://jakarta.apache.org/log4j/> for more information.

Using Comments

Comments in your code can help the debugging process in various ways – I'm not only talking about the obvious advantage of having your code well documented. Comments can be used in lots of other ways in the debugging process.

Since we're mixing different languages, we also have different types of comments.

Java Comments

The Java single line (`// . . .`) and multiple line (`/* . . . */`) comments can be used to temporarily remove parts of your Java code. If the bug disappears, take a closer look at the code you just commented. The nice thing about the multi-line comment is that it can be used across HTML section in a JSP page so you can remove large parts very easily. The following code shows a part of a JSP file where we extract database data from a `ResultSet` object:

```

<TABLE>
  <% while (rs.next()) { %>

```

```

<TR>
  <TD><%=rs.getString("first_name")%></TD>
  <TD><%=rs.getString("last_name")%></TD>
  <TD><%=rs.getString("order_date")%></TD>
</TR>
<% } %>
</TABLE>

```

To check whether the error occurs when filling the table you can remove the retrieval of the data from the `ResultSet` by using the multi-line comments like this:

```

<TABLE>
<% /* while (rs.next()) { %>
  <TR>
    <TD><%=rs.getString("first_name")%></TD>
    <TD><%=rs.getString("last_name")%></TD>
    <TD><%=rs.getString("order_date")%></TD>
  </TR>
<% } */ %>
</TABLE>

```

The resulting HTML page will contain `<TABLE></TABLE>`, which will not ruin the normal HTML layout. Of course, you should pay attention as to where to put the comments, and to close the multi-line comment properly or you will get compilation errors again.

JSP Comments

JSP comments (`<%-- ... --%>`) can be used more or less like the Java multi-line comment but they should appear in the template portions of the JSP file. One important difference is that the code between the JSP comments isn't compiled into the Servlet, so it can be used to remove code that would produce an error when the JSP file is compiled into a Java file:

```

<TABLE>
<%-- <%while (rs.next()){%>
  <TR><TD><%=rs.getString("first_name")%></TD>
  <TD><%=rs.getString("last_name")%></TD>
  <TD><%=rs.getString("order_date")%></TD></TR>
<%}%> --%>
</TABLE>

```

HTML Comments

HTML comments (`<!-- ... -->`) are fundamentally different from Java and JSP comments. HTML comments are processed like any other code in the JSP file and included in the output, but the browser hides what is inside the comments. Therefore, HTML comments cannot be used to temporarily remove incorrect code like the previous types.

What these comments can be used for is to find out the value of a variable during processing, without having this output interfere with the page layout. In our previous example, it might be good to know the internal ID number of the client:

```

<TABLE>
<%while (rs.next()){%>
  <!--client ID: <%=rs.getString("client_id")%>-->
  <TR><TD><%=rs.getString("first_name")%></TD>

```

```
<TD><%=rs.getString("last_name")%></TD>  
<TD><%=rs.getString("order_date")%></TD></TR>  
<%}%>  
</TABLE>
```

The browser shows a page that's identical to the original one, but selecting View Source gives you the ID numbers right between the rest of the output. You should be careful about leaving too much debugging information like this in your final code – the HTML comment can be easily changed into a JSP comment to hide the debugging information in a production environment.

Remember that as well as always considering any input from a client suspect, you should view all text sent to a browser as being sent to someone with hostile purposes who will seek to exploit any potential security loopholes made evident.

Invoking Debuggers

The time may well come when these simple debugging tips and tricks fail us, and we have to resort to a 'conventional' debugger, and the tools we should use for this depend on several factors. Some companies standardize on a single product or vendor and expect us to use the tools they provide for debugging. A more important factor is the developer's attitude and habits.

Some developers like working with a slick editor that does a lot behind the scenes and lets us focus on the real programming issues. Others will argue that these editors mean that we lose control over vital parameters and that 'real programmers' prefer a simple text editor and use the command line to compile their classes. Most probably, both approaches are correct in their own way. The same attitudes determine the tools and techniques a developer will use for debugging.

Integrated Debuggers

Most of the popular Java IDE's (IBM's VisualAge for Java, JBuilder, Visual Café, Forté for Java, etc.) have debugging facilities for server-side Java that integrate nicely with the rest of the development environment. Other tools like JProbe offer additional debugging facilities. These debuggers can be helpful, and some of them offer extremely useful facilities; Forté for Java Internet Edition, for example, offers the ability to step through a JSP side-by-side with the Servlet generated from it. However, they're not always the perfect and complete solution they often seem to suggest:

- ❑ The debugging process is mostly reduced to setting breakpoints, stepping through the code, and inspecting variables at runtime. However, debugging is much more than that, and many problems (like concurrency bugs) cannot be caught by this type of debugging. In fact, integrated IDE debuggers often introduce problems that either mask real application bugs, or introduce their own complexities that make their results unreliable when compared with real runtime behavior.
- ❑ If we can run everything on our development machine, debugging is rather more straightforward, but this doesn't always resolve errors. Even if the debugger is capable of doing 'remote debugging' this is always more difficult. The virtual machines (VMs) on the different servers need to be started in some sort of 'debug mode'. This isn't always easy and sometimes even impossible. The effort and impact of restarting servers makes it much harder to switch to debug mode, so should only be used for errors that are difficult to find.

- ❑ The debugger is often linked to a specific application server and even to a certain platform. In OO-terminology: the lack of encapsulation creates a tight coupling between the development/debugging practices and the deployment possibilities. Independence and 'Write Once, Run Anywhere' is one of the most important features of Java.
- ❑ No matter how fancy and advanced our tool is, debugging a server-side application will never be easy. Saying "debugging is no problem because we use an advanced integrated debugger" is like saying "we don't have to bother with security anymore, because we use Java". In reality, keeping track of a distributed system with multiple threads on multiple systems will always be a complex task.
- ❑ Some of these tools aren't bug free themselves, and the bigger the tool, the more bugs they have. Even if the tool is working fine, some developers don't like to depend on them and feel that they don't have total control over what's happening.
- ❑ These tools can be rather expensive, especially when they force us to use specific Servlet engines or web servers.

Do-It-Yourself: Using JDB

Although it's more low-level and doesn't have a point-and-click interface, the standard Java debugger, JDB, also allows remote debugging. In a standard debugging session, an application is started with the `JDB` command instead of `java`; as a rule, we should try to debug our applications on our local machine whenever possible. Most server components can be started on our development computer, like Tomcat, or have a standalone counterpart that can be used to simulate the server version.

In the case of Tomcat, the server is started with a batch file or Unix shell script. Open a copy of this startup script in a text editor and look for the line where the server itself is started. In Tomcat 4.0 on a Windows platform, the file is `catalina.bat`, which contains these lines:

```

...

if not "%OS%" == "Windows_NT" goto noTitle
set _STARTJAVA=start "Catalina" "%JAVA_HOME%\bin\java"
set _RUNJAVA="%JAVA_HOME%\bin\java"
goto gotTitle
:noTitle
set _STARTJAVA=start "%JAVA_HOME%\bin\java"
set _RUNJAVA="%JAVA_HOME%\bin\java"
:gotTitle

...

%_RUNJAVA% %CATALINA_OPTS% -Dcatalina.home="%CATALINA_HOME%"
org.apache.catalina.startup.Bootstrap %2 %3 %4 %5 %6 %7 %8 %9

...
```

Changing `java` to `jdb`, we have an alternative startup script that starts the server in debug mode.

*In earlier versions of Tomcat, the file was instead called `tomcat.bat`. Note that this sort of experimentation with the Tomcat startup script is actually encouraged by the Tomcat developers; as the Tomcat 3.2 User Guide says "Do not hesitate, just do it." However, it would be prudent to make your changes to a **copy** of the file rather than to the original, indeed it is possible to assemble a good-sized collection of Tomcat startup scripts that each address a different debugging configuration.*

Other application servers allow us to specify a JVM startup path and additional options to achieve the same results. It might take some time to figure out how to start a particular Servlet engine in debug mode, but in most cases it is possible.

An important feature of `jdb` is that it allows us the option to connect to an already running JVM, which may be on another computer. This might be easier than the normal JDB use we just described, especially in a server environment. For details of how to do this, see the `jdb` documentation: the VM we want to debug must be started in debug mode, and we have to disable the JIT compiler by adding `-Djava.compiler=NONE` to the command line, and make a few extra classes available to the debugger. The JVM starts as usual but displays a password on the screen that we need to supply to `jdb` in order to connect to this VM.

This password is necessary to correctly identify the VM, since there can be more than one running on the same server. Also, because you can connect to this debug VM from another host, the password provides additional security. The biggest problem is that it's not always possible to start the JVM visually in order to get the password.

This is a very narrow interpretation of the debugging concept. As we said before, debugging is much more than setting breakpoints and stepping through the code. Most of the remarks mentioned, regarding the fancy integrated IDE's are also valid with this low level JDB approach. But if you are the kind of developer who edits their code with `notepad` or `vi` and uses the command line to compile their classes, you might prefer this JDB command-line technique.

Monitoring Tools

Monitoring tools, whether generic or developed especially for a particular application, can provide a valuable further weapon in our debugging arsenal. We'll look at a couple of potential tools.

Redirecting Standard Output

Most application servers and Servlet engines start the JVM without a visual presence, like a console window. However, even if we have the option of displaying a terminal window, (for example, by using `java` instead of `javaw`), the messages still show up on the server instead of on our PC.

Luckily, `System.setOut()` lets us change the `OutputStream` that is used by `System` at runtime. The following `Debug` class uses that technique to redirect the standard out to a `ServerSocket` at a specified port. The class extends `OutputStream` and registers itself as the standard output. It also starts a thread that listens to incoming connections. When a client connects, the standard out is redirected to that client socket. Since we keep a reference to the previous standard output, we can keep sending all messages to the old `OutputStream` as well. To keep things simple, this class only supports a single connection:

```
import java.io.*;
import java.util.*;
import java.net.*;

public class Debug extends OutputStream implements Runnable {

    // Instance variables
    int         port;           // The port we listen on
    ServerSocket server;       // ServerSocket listening on the port
    boolean     active;        // Are we actively awaiting connections?
```

```
Socket      client;        // Our debugging client
OutputStream clientStream; // The OutputStream to the client
Thread      listener;     // The thread used to listen for connections
PrintStream old;         // The original System.out

// Create a Debug OutputStream listening on the specified port

public Debug(int port) {
    this.port = port;

    try {
        server = new ServerSocket(port);
    } catch (IOException e) {
        System.out.println("could not create server");
    }
}
```

The `isActive()` method returns true if the debug stream is currently active:

```
public boolean isActive() {
    return active;
}
```

The `startServer()` method activates the debug stream by redirecting `System.out`:

```
public void startServer() {
    if (!active) {
        old = System.out;

        System.setOut(new PrintStream(this));

        active = true;
        listener = new Thread(this);

        listener.start();
        System.out.println("debug server started");
    }
}
```

The `stopServer()` method stops the debug stream by directing `System.out` back to the original stream:

```
public void stopServer() {
    active = false;

    System.setOut(old);
    System.out.println("debug server stopping");

    if (client != null) {
        try {
            client.close();
        } catch (IOException e) {}
    }
}
```

The `Debug` class implements `Runnable` and `run()` listens for socket connections and if no-one is already connected directs debugging output to the client:

```
public void run() {
    Socket localSocket = null;

    try {
        while (active) {
            localSocket = server.accept();

            if (client == null) {
                client = localSocket;
                clientStream = client.getOutputStream();

                new PrintStream(clientStream).println("Welcome to the Debug Server");
            } else {
                PrintWriter second = new PrintWriter(localSocket.getOutputStream());

                second.print("already connected");
                localSocket.close();
            }
        }

        System.out.println("debug server stopped");
    } catch (IOException e) {
        System.out.println("debug server crashed");
        System.out.println(e.getMessage());

        active = false;
    }
    finally {
        if (server != null) {
            try {
                server.close();
            } catch (IOException e) {}
        }
    }
}
```

A client is disconnected if communication with it goes wrong. This method is called from `write()` below in the event of an `IOException`:

```
protected void clearClient() {
    if (client != null) {
        try {
            client.close();
        } catch (IOException ioe) {}
    }

    client = null;
    clientStream = null;
}
```

The following method overrides `OutputStream.write()` to direct output to both any remote debugging client and the old `System.out`:

```
public void write(byte[] b) throws IOException {
    if (old != null) {
        old.write(b);
    }

    if (clientStream != null) {
        try {
            clientStream.write(b);
        } catch (IOException e) {
            clearClient();
        }
    }
}

public void write(byte[] b, int off, int len) throws IOException {
    if (old != null) {
        old.write(b, off, len);
    }

    if (clientStream != null) {
        try {
            clientStream.write(b, off, len);
        } catch (IOException e) {
            clearClient();
        }
    }
}

public void write(int b) throws IOException {
    if (old != null) {
        old.write(b);
    }

    if (clientStream != null) {
        try {
            clientStream.write(b);
        } catch (IOException e) {
            clearClient();
        }
    }
}
}
```

So how do we use this `Debug` class? We simply use `System.out.println()` to send debug messages. The nice thing is that there is no direct connection between the class we want to debug and this `Debug` server. We don't have to additionally import anything. The `Debug` server makes sure that these messages are sent to the developer's machine.

To use this class to debug servlets and JSP, we can create a `DebugServlet`:

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

```

public class DebugServlet extends HttpServlet {
    Debug debugger ;

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        String option = req.getParameter("option");
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<html><head><title>Debug servlet</title></head><body>");
        out.println("<h1>Debug servlet</h1>");
    }
}

```

Debugging is switched on and off like this:

```

    if ("socket".equals(option)) {
        if (debugger == null) {
            debugger = new Debug(9999);
        }
        debugger.startServer();
    }
    if ("closesocket".equals(option)) {
        if (debugger != null) {
            debugger.stopServer();
        }
    }
    String testValue = req.getParameter("test");
    if (testValue != null) {
        System.out.println("Debug test: " +testValue);
    }

    if ((debugger != null) && (debugger.isActive())) {
        out.print("<a href=\"telnet://\"");
        out.print(req.getServerName());
        out.println("\":9999\" target=\"_blank\"> Connect to the debugger </a><p>");

        out.print("<a href=\"");
        out.print(req.getRequestURI());
        out.println("?option=closesocket> Shut down debugger </a ><p>");
    } else {
        out.print("<a href=\"");
        out.print(req.getRequestURI());
        out.println("?option=socket\"> Start remote debugger </A><p>");
    }
    out.println("<form method=\"post\" >");
    out.println("Test <input type=\"text\" name=\"test\">");
    out.println("<input type=\"submit\">");
    out.println("</form> </body> </html>");
}
}

```

By specifying the `option` parameter, the Servlet controls starting and stopping of the remote debugger. The servlet itself shows the correct hyperlinks to perform these commands. We could create a 'debug client' as an application or an applet that opens a socket connection to our debug server. However, the only thing this client has to do is display to the user what was sent over the socket connection.

On most Operating Systems there is already a tool that does exactly that. It's called Telnet, and it's mostly used to control a server or interact with a certain service, but in our case, we just use it as an ultra thin client solution to connect to port 9999 and see the debug messages. To make this even more convenient, the Servlet presents us with a hyperlink that tries to open Telnet with the correct host and port. Any string that's entered into the form is printed to `System.out`, so this immediately allows us to check if everything is working, as is shown in the next screenshot.

Some versions of Windows have rather limited Telnet programs that lack the ability to scroll back up the screen; in this case, a third-party Telnet client may be of use.

The Telnet window not only shows the messages sent by Servlets or JSP, but also those from any other object or Bean that runs in the same VM:



This `Debug` class is already very powerful, but there are many potential enhancements:

- Add security like asking for a password or restricting access to certain IP addresses.
- Allow multiple Telnet sessions to connect to a single server.
- Allow the client to specify filters. When there are lots of debugging messages, it might be better to only send the messages that contain a specified 'filter' string.
- Add an optional timestamp to each message.
- Allow the client to enter certain commands, like asking for memory usage, or shutting down the debug server through the Telnet session.
- The JDBC `DriverManager` has a method `SetLogWriter` that works just like `System.setOut`. We could try to change the `Debug` class so that this JDBC log information is sent over the network instead of `System.out`.

Session Monitoring

The following servlet allows us to check out all objects stored in the session. We might also provide mechanisms to change the values of the objects or to delete them from the session:

```
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionMonitor extends HttpServlet {
```

`doGet()` processes a client request. An option is given to add a `TestBean` to the session. Then, if a Bean name was specified display that Bean's details, otherwise list all Beans in the session:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    HttpSession session = req.getSession(true);
    out.println("<HTML><HEAD><TITLE>session monitor</title></head>");
    out.println("<BODY><H1>SessionMonitor</H1>");
    out.println("This form allows us to add new string values to the ");
    out.println("current session to check out this servlet");
    out.println("<FORM>add string key <INPUT TYPE=\"text\" ");
    out.println("NAME=\"key\"><br/>");
    out.println("add string value<INPUT TYPE=\"text\" NAME=\"value\"><br/>");
    out.println("<INPUT TYPE=\"submit\"></FORM><P>");

    testInit(req,session);
    String beanName = req.getParameter("name");
    if (beanName == null) {
        showBeanList(req, session, out);
    } else {
        showSingleInstance(beanName, session, out);
    }
    out.println("</BODY></HTML>");
}
```

A `TestBean` is stored in the session with the specified name and first value:

```
private void testInit(HttpServletRequest req, HttpSession session) {
    String newKey = req.getParameter("key");
    String newValue = req.getParameter("value");
    if ((newKey != null) && (newValue != null)){
        TestBean test= new TestBean();
        test.setValue1(newValue);
        test.setValue2("fixed text");
        test.setValue3(newKey+"-->" +newValue);
        session.setAttribute(newKey, test);
    }
}
```

The `showBeanList()` method displays a list of beans stored in the session, and links back to this Servlet to give details of each one:

```
private void showBeanList(HttpServletRequest req, HttpSession session,
                          PrintWriter out) {
    String URI = req.getRequestURI();
    Enumeration names = session.getAttributeNames();

    while (names.hasMoreElements()) {
        String attributeName= (String) names.nextElement();
        out.print("<A HREF=");
        out.print(URI);
        out.print("?name=");
        out.print(attributeName);
        out.print(">");
        out.println(attributeName);
        out.print("</A><BR />");
    }
}
```

The following method displays the details of the bean stored in the session under the name `beanName`. Introspection is used to display each of its fields:

```
private void showSingleInstance(String beanName, HttpSession session,
                                PrintWriter out) {
    Object check = session.getAttribute(beanName);
    out.println("<H2> Checking object ");
    out.println(beanName);
    out.println("</H2><UL>");
    try {
        Class checkClass = check.getClass();
        Field[] fields = checkClass.getFields();
        for (int i = 0; i < fields.length; i++) {
            out.println("<LI>");
            out.println(fields[i].getName());
            out.println(" (");
            out.println(fields[i].getType().toString());
            out.println("): ");
            try {
                out.println(fields[i].get(check).toString());
            } catch (Exception e) {
                out.println(" ! Cannot be displayed !");
            }
        }
    } catch (NullPointerException e) {
        out.println("null pointer Exception");
    }
}
```

`TestBean` is a sample bean we can put into the session to test this Servlet:

```
private class TestBean {
    public String value1;
    public String value2;
    public String value3;
    public String getValue1() {
        return value1;
    }
    public void setValue1(String value) {
```

```

    value1 = value;
}
public String getValue2() {
    return value2;
}
public void setValue2(String value) {
    value2 = value;
}
public String getValue3() {
    return value3;
}
public void setValue3(String value) {
    value3 = value;
}
}
}

```

The application event listeners introduced in the new Servlet 2.3 specification introduce further possibilities for monitoring the status of an application. Running the Servlet gives us the following output:



Once we add a key and a string value, they are stored in the session and can be viewed by clicking on the key's name. For example, if we add a key named `shoppingCart` and give it the value `empty` we would see the following screen if we clicked on the appropriate link:



Avoiding Common Problems

Having reviewed the debugging tools and techniques that will be of use when dealing with Java-based web applications, let's look at some of the more specifically problematic areas that often cause problems.

Long Running Processes

Server-side Java processes typically run for a long time when compared to CGI scripts, as they don't start a new process with every request; therefore, some problems might never occur in an application/applet context, just because these are typically stopped and restarted quite often. This is obviously not something normally done on server-side applications.

One of the potential problems caused by long running processes is running out of system memory resources. Java's garbage collector takes care of many potential memory leaks; compared to C++ this saves the Java developer from some problems that are otherwise hard to debug. However, this doesn't mean that we can forget about memory issues altogether. The garbage collector destroys all objects that are no longer referenced, yet we can still have object instances that are no longer required but somehow still have a reference to them, and they cannot be cleaned. Since these objects can keep references to other objects, we might keep a lot of memory space occupied.

These objects are called 'loiterers'. (See <http://www.ddj.com/articles/2000/0002/00021/00021.htm> for some more details on memory leaks in Java.)

Firstly, we can use our OS to monitor the memory usage of the JVM. JDK 1.3 comes with a somewhat unfriendly profiler, or there are also tools like JProbe profiler (<http://www.sitraka.com>) and OptimizeIt (<http://www.optimizeit.com>) that give us a very detailed overview of how much memory our objects occupy and can help debugging these problems. The implementation of the garbage collector depends on the JVM. Some are better than others, which might lead to bugs or problems that only occur on a certain OS.

Most server-side applications use caching on several layers to speed up processing. In long running processes, it's important that this cache doesn't grow to unlimited size. Since there's no 'sizeof' in Java to measure how much memory an object uses, it's not easy to create a cache that is strictly memory limited.

Most caches simply allow a fixed number of objects, regardless of their size. Although not very sophisticated, this might give good results, provided we base our cache size on tests and not on some wild guess. Also, keep in mind that an object in the cache might keep a reference to other objects, preventing these instances from being garbage collected. In which case, a cache with a rather small number of objects can still occupy a lot of memory.

Even when resources are container managed it is important to hold on to resources for as little time as possible and to clean up other resources. We often see code like this:

```
try {
    FileInputStream in = new FileInputStream(filename);
    // Do something that can throw an IOException
    in.close();
} catch (IOException e) {
    // We don't have a handle here to the InputSteam
}
```

When an error occurs, the `InputStream` isn't properly closed. We might just rely on the garbage collector to close the file, but the correct code should declare the `FileInputStream` outside the `try` block and close the `FileInputStream` in a `finally` block. In any case, the OS will close the file when the program finishes. Therefore, this code might not be a big problem for normal applications. Server-side applications, however, are meant to run for days or even months without restarting, so proper cleanup becomes much more important.

The situation is even more complex in the case of JDBC connections, especially when we use connection pooling. Some JDBC drivers not only keep resources occupied when a connection isn't closed, but also with an open `Statement` or `ResultSet`. Normally we could expect the closing connection to take care of closing a `Statement` or a `ResultSet`, but this isn't always the case. Besides, when we use connection pooling, we don't close the connection at all, and therefore the garbage collector cannot help us in any way:

```
try {
    Connection conn = // Get a connection from the pool
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT COLUMNS FROM TABLE");
    // Get the values
    // or update the database
    rs.close();
    stmt.close();
    // Return connection to the pool
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

If we use JDBC code like this, there are several potential problems:

- ❑ If something goes wrong the `Connection` is never returned to the pool, but also the `ResultSet` and `Statement` aren't closed properly and could keep open resources on the database.
- ❑ If we don't use auto-commit, we should do a rollback in case of failure.
- ❑ Another thing that's often forgotten, we only use the first `SQLException`. `SQLExceptions` have a special feature that allows them to have several error messages linked to a single exception. We'll come back to this feature later. In our example, valuable information that's contained in the errors that are chained to the first exception is lost.

To avoid these problems, we should code it like this:

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
try {
    conn = // Get a connection from the pool
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT COLUMNS FROM TABLE");
    // Get the values
    // or update the database
    conn.commit();
} catch (SQLException e) {
```

If we're handling the exception at this level, deal with it. Otherwise let this method throw a `java.sql.SQLException` and get the next level up to deal with the consequences:

```

        System.out.println(e.getMessage());
        while ((e=e.getNextException()) != null) {
            System.out.println(" next " + e.getMessage());
        }
    } finally {

```

In the `finally` block, close the `ResultSet`, `Statement`, and `Connection`. Do each inside its own `try-catch` block, to help ensure that they all get called even if one of the `close()` calls throws an exception:

```

    try {
        if (rs != null) {
            rs.close();
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
        while ((e = e.getNextException ()) != null){
            System.out.println(" next " + e.getMessage());
        }
    }

    try {
        if (stmt != null) {
            stmt.close();
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
        while ((e = e.getNextException ()) != null){
            System.out.println(" next " + e.getMessage());
        }
    }

    try {
        if (conn != null) {
            conn.rollback();
            // Return the connection to the pool
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
        while ((e = e.getNextException ()) != null){
            System.out.println(" next " + e.getMessage());
        }
    }
}

```

Even this code isn't 100% bullet proof, but it certainly becomes ugly. OK, we might choose to ignore handling the exceptions that are thrown by the closing and rollback statements, but we still have more error handling than actual database code. Do we really write code like this everywhere we access the database? Probably not, unless we're a very disciplined programmer, but we ought to.

The difficulty with these memory- or resource-related bugs is that they only show up after a certain time and they are difficult to reproduce, especially on our development computer.

Robustness and Stability

Server-side applications should not fail, but when they do, they should do so gracefully with minimal impact on other components and the server. A server application crashing due to component failure can affect many users in addition to other applications running in the same VM.

Built-in range checking, memory management, and exception handling plus the absence of pointers prevent Java code from producing the types of system crashes associated with languages such as C++. It remains the developers' responsibility to follow the specifications for API's carefully to avoid threading issues and other programmatic failure.

Whilst the only real way around these issues is to design the system from the outset to avoid concurrency problems, we can use tools such as Apache JMeter (<http://java.apache.org/jmeter/>) that simulate a number of concurrent users requesting information – normally used to measure performance – to test the robustness of our setup under heavy load. Such tools will be covered in Chapter 20.

Stress testing the system can highlight weaknesses and look for performance bottlenecks. To provide stability at the hardware level, an application might be deployed on multiple servers. When one of them goes down, the others take over. Such a fail-over setup adds complexity to the total system.

Due to this complexity using a fail-over backup server might, curiously, introduce new bugs. For example, if we store `non-Serializable` objects in a session or if the application server doesn't support distributed sessions we could lose session state information while switching over to the backup server. (With distributed sessions, a session can be serialized and transferred bodily across to a different server.) The same problem can arise if we use multiple servers in parallel to provide better performance (known as load balancing) – the only solution here is to read your server documentation carefully.

Nesting Exceptions

In a typical setup, the interface to the user is made up of JSPs and Servlets. This is an appropriate place to use an error page to inform the user about the error. Exceptions move up the call stack to the point where they are caught. Eventually they reach the `try-catch` block around the body of the JSP-Servlet (the Servlet that's automatically created by the JSP engine) and the user is redirected to the error page.

In complex systems, the exception can originate from a distant low-level component. The end user shouldn't be bothered with these low-level details. A user should be shown an error like "user registration failed" instead of "SQLException: X03805: Primary key for table reg_usr is not unique", although for debugging purposes the latter is very useful.

In a clean setup, the higher-level components should catch exceptions (like `IOExceptions` or `SQLExceptions`) thrown by the components they use, and in turn, throw a higher-level exception (like `LoginException` or `EmployeeException`) themselves. If these higher-level components just pass on the low-level exceptions, all possible exceptions would have to be specified in the interface. This not only looks bad, but it also means our classes aren't properly encapsulated.

Of course, it would be ideal if we could somehow show the higher-level error messages to the end user, but still have the option of recording all the details of the original errors for debugging/maintenance purposes. To achieve this we can use nested (or chained) exceptions. Using nested exceptions means that an exception contains a handle to another exception (which might in turn have a reference to another, creating a chain of exceptions). This is a technique that's used in the JDK classes too, such as in `java.sql.SQLException` or `java.rmi.RemoteException`. Unfortunately, this isn't always implemented in the same way and it might have been better if this chaining mechanism were provided by the root exception object.

Exceptions normally have two constructors: the default constructor, and one where we can specify a `String` error message. In order to create nested exceptions, we make a third constructor that takes the message `String` and another object of type `Exception` that's stored in a member variable:

```
public class NestedException extends Exception {
    private Exception nested;

    public NestedException(){}

    public NestedException(String msg) {
        super(msg);
    }

    public NestedException(String msg, Exception nested) {
        super(msg);
        this.nested=nested;
    }

    public Exception getNestedException() {
        return nested;
    }
}
```

To use this technique, we must inherit our own custom exceptions from `NestedException` instead of from the root `Exception` class.

In our components that throw the higher-level exceptions and catch the lower-level ones, we typically use code like this:

```
public boolean login(String userid, String password) throws LoginException {
    try {
        // Here we have code to validate the login
        // say, by looking it up in the database
    } catch (SQLException e) {
        // Log the error
        throw new LoginException("Could not validate login", e);
        // Constructor type 3 from NestedException
    }
}
```

Since we inherited our `LoginException` from `NestedException`, we can use the `LoginException` information to show the user what happened, and recursively check all exceptions that are chained to this `LoginException` to provide a more detailed error message for the developer. Since the chaining mechanism isn't built into the standard exception object, probably the safest way to recursively get all the information from a `NestedException` is to override the `getMessage()` method:

```
public String getMessage() {
    StringBuffer msg = new StringBuffer (super.getMessage());
    if (nested !=null) {
        msg.append("\r\n"); // or <BR /> for display in browser
        msg.append(nested.getMessage());
    }
    return msg.toString();
}
```

The standard `SQLException` doesn't use this technique to show the messages of the contained exceptions automatically in their `getMessage()` method. As a result, most nested exceptions in `SQLExceptions` are never used and the information is just thrown away.

The article at <http://www.javaworld.com/javatips/jw-javatip91-2.html> gives a more detailed explanation of nested exceptions and shows how to get the stack trace from the nested exception, even if that exception is thrown on a remote RMI server.

Concurrency

Unlike some other technologies, web applications are inherently multi-threaded. Every request uses a unique thread that executes the `service()` method of a single Servlet instance. While this technique offers good performance and easy solutions to some specific problems, it also introduces a certain complexity. Developers not familiar with multi-threaded systems might miss code that is not thread-safe. The Java keyword `synchronized` is the key to making sure a piece of code is only executed by one thread at a time, and a thorough understanding of its use and related threading issues is crucial in writing thread-safe web applications.

We emphasized at the beginning of this chapter that a good design is the best way to reduce time spent debugging, and this applies particularly when considering concurrency. The way to prevent concurrency problems is to think about it very carefully before starting to code, rather than discovering the problems while debugging and trying to fix them at that stage.

Let's give an example of a Servlet that is not thread safe. We first create a very simple hit counter like this:

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

public class HitCounter extends HttpServlet {
    int hits = 0;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        hits++;

        out.println("<html><head><title>Run command</title></head>");
        out.println("<body><h1>You are visitor # ");
        out.println(hits);
        out.println("</h1></body></html>");
    }
}
```

At first we might see nothing wrong with this code. However, consider two requests at almost exactly the same moment. Assume the current number of hits is 78. The first thread starts executing the `doGet()` method but when it reaches line 11 (hits is now 79), the OS switches to the second request and this new thread processes the complete method and displays "You are visitor # 80". Immediately after that the first thread can continue and will also show "You are visitor # 80". The following table clearly shows the execution of the two concurrent requests:

Thread 1	Thread 2	Hits
res.setContentType("text...		78
hits++;		79
out.println("<html><head>...		79
	res.setContentType("text...	79
	hits++;	80
	out.println("<html><head>...	80
	out.println("<body><h1>...	80
	out.println(hits);	80
	out.println("</h1></body>...	80
out.println("<body><h1>...		80
out.println(hits);		80
out.println("</h1></body>...		80

There's never been a number 79 and we have two number 80s. Of course, if it's only a hit counter, we don't really care much about this. However, the same problem can occur in code that is more critical as well.

Think about the consequences if we want to give a prize for the 1000th visitor. We run the risk of having zero or two winners.

In our hit counter example, we might try to combine lines 11 and 15 to:

```
out.println(++hits);
```

The probability of still having the concurrency problem will be much lower, but it still exists. Besides, it isn't always possible to put the related lines that close together.

That's where we would need to synchronize to make sure no two threads will execute a single block of lines at the same time:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

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

    out.println("<html><head><title>Run command</title></head>");
    out.println("<body><h1>Number of hits: ");

    synchronized(this){
        hits++;
        out.println(hits);
    }

    out.println("</h1></body></html>");
}
```

However, synchronizing can have a serious impact on execution speed. First of all, the action of locking an object itself takes some time, but obviously, when a thread has to wait on another thread to finish a piece of code, that time is just wasted. That's why we had better not synchronize the complete `doGet()` method by using:

```
public synchronized void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
```

In this case, the chances of having one thread waiting for another would become much higher.

If it weren't for this performance penalty, the easiest solution would be to just synchronize every method. It's also not easy to decide what parts to synchronize. If the question, "What to synchronize?" could be answered easily, this would most probably be done automatically for you by the Java compiler. Instead, we need to carefully decide ourselves, as part of our system design, how we are going to prevent problems arising from concurrent access to shared objects.

Performance isn't the only issue in considering where to use synchronization. In fact, not designing your synchronization carefully can cause deadlock problems. Especially in a distributed environment, this can become quite complicated. If we have an RMI server that calls back to the original calling object and both methods are synchronized, we end up with two servers waiting for each other and the system hangs.

Where can we expect concurrency problems?

- ❑ Objects stored in the `ServletContext` (the JSP application scope). It should be obvious that these objects can be used by several threads at the same time.
- ❑ Objects stored in a session. Although at first, it might seem strange to have multiple concurrent requests using an object in a session. We might imagine a screen with several browser windows and a user clicking the mouse as fast as he can, but it's less spectacular. A single request for a page that uses frames can easily cause multiple requests that result in concurrent threads on the server. The fact that these multiple threads originate from a single user doesn't make it less problematic.
- ❑ To speed up our application, we often implement a caching mechanism. Instead of recreating certain objects every time we need them, we could put objects that take some time to be instantiated in a cache and use the same instance again later. An instance stored in such a cache can be retrieved and used by multiple requests at the same time. Ideally, the decision whether you're using a cache or not should be transparent to the programmer that uses the beans. Performance optimizations should be done near the end of the development cycle, once the code is working, though it is important to have eventual optimization in mind during the design phase. However, by simply starting to use a cache, new concurrency bugs can suddenly appear in code that always ran fine.
- ❑ Singleton patterns and static methods which access static variables. The reason to use a Singleton pattern and static methods is to have a single instance of an object available to the whole VM, so obviously, we can have concurrent requests.
- ❑ Servlet member variables like in the `HitCounter` example. Since there's normally only one instance of a Servlet and every request uses that same Servlet instance, every method that changes member variables should use synchronized blocks. The same is true for variable declarations between `<%! %>` JSP tags, that results in an instance variable in the generated Servlet. We often see code where people use `<%! %>` to declare variables, where a thread safe `<% %>` scriptlet (which generates a local variable within the `_jspService()` method) would give the same results. In fact, you hardly ever need to use `<%! %>` declarations in JSP.

Consider the final point above. We could write the following code to declare the `myPage` variable and instantiate it:

```
<%! String myPage; %>
<% myPage = request.getParameter ("target"); %>
<jsp:include page="<%=myPage%" />
```

However, a simpler, thread safe way of doing the same thing would be:

```
<% String myPage = request.getParameter ("target"); %>
<jsp:include page="<%=myPage%" />
```

This at first sight does exactly the same thing, and if you try it out, gives exactly the same results. However, with the first code, the requested page might be mixed up when two users try this at the same moment.

Once again, we must conclude that concurrency errors are hard to debug. The bugs are difficult to reproduce on a development server and almost impossible to isolate properly. It's very likely that they show up only when the application is being used at full speed. Step debugging (JDB-like) probably won't help very much in locating the problem since the timing of the different threads is so crucial for the error to occur. This is where logging is very important, as it may be the only way to locate these errors.

Some multi-threading issues like deadlocks and race conditions can be investigated with tools like JProbe ThreadAnalyzer (<http://www.sitraka.com>).

The Stress-test tools we mentioned before, like JMeter, can help us reproduce concurrency problems.

Class Loading

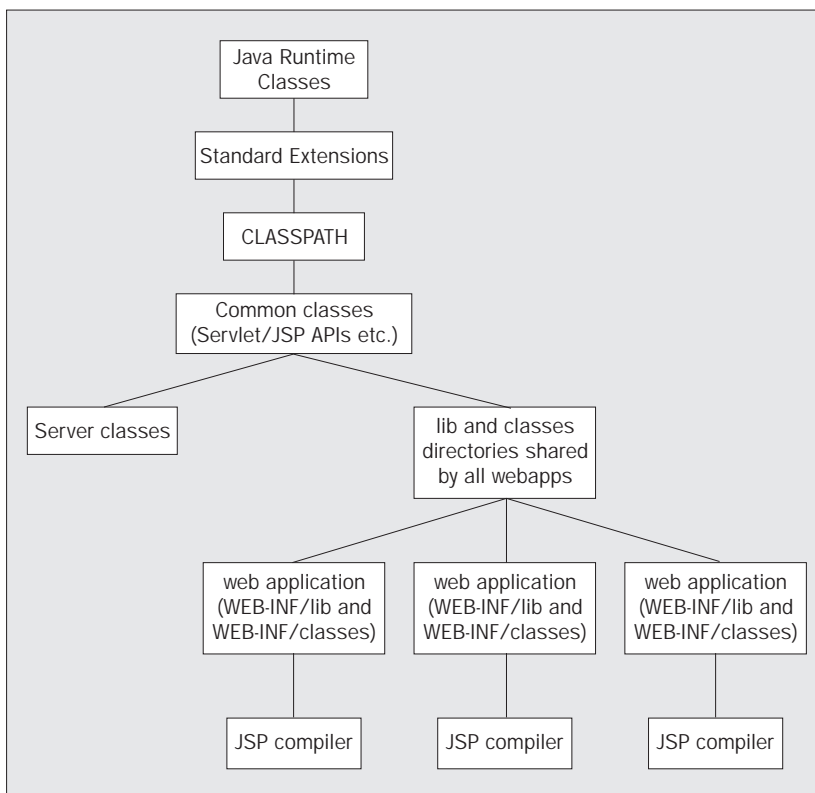
One final area that seems to cause particular difficulty when deploying JSP and Servlet-based applications is class loading: getting Tomcat to recognize the classes you need for your application. You know the classes are there; it looks as though Tomcat ought to be able to see them, but still you get the dreaded "Class java.wibble not found" message.

In Java, classes are loaded into the virtual machine by a `ClassLoader`, and Tomcat 4.0 uses a hierarchy of such `ClassLoaders` to load all relevant classes (including its own). The locations from which Tomcat 4.0 loads classes are many and various. The Tomcat documentation provides full details, but to summarize the relevant points for our purposes, the classes visible to web applications come from:

- ❑ The Java system classes, located in `rt.jar` and `i18n.jar` in the Java Runtime Environment's `jre/lib` directory.
- ❑ Any Java extensions found in the `jre/lib/ext` directory.
- ❑ All directories and JAR files listed in the `CLASSPATH` environment variable. Note that Tomcat 4.0's startup script, `catalina.bat`, deliberately **ignores** any system-wide `CLASSPATH` environment variable you may have set and constructs its own `CLASSPATH` containing only `bootstrap.jar` from the `$CATALINA_HOME/bin` directory.
- ❑ Any JAR files located in the `$CATALINA_HOME/lib` directory. This contains further Tomcat 4.0 components, and can be used to make available any sets of classes (like JDBC drivers) that are required by a number of web applications.

- ❑ Within each web application, the classes in its `WEB-INF/classes` directory and any JAR files in the `WEB-INF/lib` directory are made available to that application only.

That sounds quite straightforward, but to fully understand the implications remember that these `ClassLoaders` form an inverted tree-like hierarchy. The full picture, including Tomcat implementation classes not visible to web applications, looks like this:



There are therefore three things to bear in mind about how classes will be located:

- ❑ A class will be loaded by the highest `ClassLoader` that can locate it. Therefore, if a given JAR file is placed in both the `jre/lib/ext` directory and a web application's own `WEB-INF/lib` directory, its classes will be loaded from `jre/lib/ext`.
- ❑ When one class is trying to load another, this is possible if the class to be loaded and the loading class are in the same `ClassLoader`.
- ❑ However, a class cannot be loaded if it is only available in a `ClassLoader` lower down the hierarchy than the class attempting to load it.

This has important implications when using frameworks such as Struts (see Chapter 21) that need to load application-specific classes such as those implementing the Struts `Action` interface. If `struts.jar` is placed in the `$CATALINA_HOME/lib` directory, and the application-specific classes in that application's `WEB-INF/classes` or `WEB-INF/lib` directory, then the `Action` classes will not be found as they are in a lower `ClassLoader`. Therefore, `struts.jar` must **only** occur within that application's `WEB-INF/lib` directory.

In general, even if a JAR file is required by more than one application, it is best to make it available by placing a copy in *each* application's `WEB-INF/lib` directory.

Tomcat 4.0 Beta 1 had problems loading XML parsers other than the version of JAXP 1.1 release it came with, failing with a 'sealing violation' exception. This problem has been overcome, and the rules for using an XML parser within your web application are now that you should either:

- ❑ Place the JAR files for your choice of XML parser in the application's `WEB-INF/lib` directory
- ❑ If you want JAXP 1.1 to be available to every web application, move the JAXP 1.1 JAR files from the `$CATALINA_HOME\jasper` directory to the `$CATALINA_HOME\lib` directory

Alternatively, if you do not require JSP, you can place the JAR files for your choice of XML parser directly in `$CATALINA_HOME\lib`, but this is likely to cause the JSP engine to fail with 'sealing violation' exceptions.

Summary

In this chapter, we've covered:

- ❑ Understanding error messages you receive when running JSP pages
- ❑ Techniques for tracking down and isolating bugs, including logging, printing, and using comments
- ❑ Ways of using debuggers with Java web applications
- ❑ Building tools to help with debugging and monitoring applications
- ❑ Potential problem areas that you will often run into: robustness, concurrency, and class loading

Even after reading this chapter, debugging JSP/Servlet applications will still be hard. Hopefully some of the techniques and tools that we've shown will help you to get started, and you'll be able to avoid some of the more common pitfalls, but most importantly you'll need to acquire the special debugging mindset.

The next chapter looks at techniques for getting good performance from our web applications.

