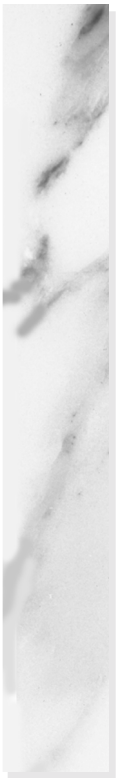
THE JSP ENGINE: UNDER THE HOOD



Topics in this Chapter:

- Behind the Scenes
- Multithreading and Persistence
- The Implicit Objects
- The JSP Lifecycle
- A JSP Compiled
- Performance Tuning the JSP

Chapter 4

The previous chapters addressed the background, syntax, and elements of JSP. Until this point many of the technical details of how the JSP engine works have been glossed over or avoided entirely. Developing good JSP applications involves at least a basic understanding of how the JSP engine works.

4.1 Behind the Scenes

When the JSP engine receives a request for a page it converts both static data and dynamic elements of a JSP page into Java code fragments. This translation is actually fairly straightforward.

The dynamic data contained within the JSP elements are already Java code, so these fragments can be used without modification. The static data gets wrapped up into `println()` methods. These Java code fragments are then sequentially put into a special wrapper class.

The JSP wrapper is created automatically by the JSP engine and handles most of the work involved in supporting JSP without the author's involvement. The wrapper usually extends the `javax.servlet.Servlet` class, which means that JSP actually get converted into a special form of Java Servlet

code. In many ways JSP could be considered a macro language for creating Java Servlets; JSP pages essentially provide a page-centric interface into the Java Servlet API.

The source code is then compiled into a fully functioning Java Servlet. This new Servlet created by the JSP engine deals with basic exception handling, I/O, threading, and a number of other network and protocol related tasks. It is actually this newly generated Servlet that handles requests and generates the output to clients requesting the JSP page.

Recompiling

The JSP engine could have been designed to recompile each page when a new request is received. Each request would generate its own Servlet to process the response. Fortunately JSP takes a more efficient approach.

JSP pages and Java Servlets create an instance once per page, rather than once per request. When a new request is received it simply creates a thread within the already generated Servlet. This means that the first request to a JSP page will generate a new Servlet, but subsequent requests will simply reuse the Servlet from the initial request.

CORE Note: Delay on first request

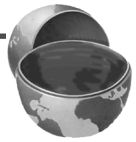
When a JSP page is first run through the JSP engine there may be a noticeable delay in receiving a response. These delays occur because the JSP engine needs to convert the JSP into Java code, compile it, and initialize it before responding to the first request.

Subsequent requests gain the advantage of using the already compiled Servlet. Requests after the initial request should be significantly faster in processing.

There are specific occurrences that can instruct the JSP engine when to recompile the JSP page. To manage this the JSP engine keeps a record of the JSP page source code and recompiles the page when the source code has been modified. Different implementations of JSP have different rules for when to compile, but all engines are required to recompile when the JSP source code changes.

Keep in mind that external resources to a JSP page, such as a JavaBean or an included JSP page, may not cause the page to recompile. Again, different JSP engines will have different rules on how and when to recompile the page.

CORE Note: The Precompile Protocol



As of JSP1.1, a means of precompiling JSP pages is defined in the specification. To precompile a specific JSP an HTTP request to the JSP must be made with the `jsp_precomile` *parameter set.*

For example, entering the URL `http://www.javadesktop.com/core-jsp/catalog.jsp?jsp_precompile="true"` *should compile this JSP, if it has not already been compiled or the JSP source code had changed.*

The Servlet-JSP Relationship

Because JSP pages are transformed into Java Servlets, JSP displays many of the same behaviors as Java Servlets. JSP inherits powerful advantages and several disadvantages from Java Servlets.

Java Servlets work by creating a single persistent application running in the JVM. New requests are actually handled by running a new thread through this persistent application. Each request to a JSP page is really a new thread in the corresponding Java Servlet.

The Java Servlet also provides the JSP developer with several built-in methods and objects. These provide a direct interface into the behavior of the Servlet and the JSP engine.

4.2 Multithreading and Persistence

JSP inherits multithreading and persistence from Java Servlets. Being persistent allows objects to be instantiated when the Servlet is first created, so the physical memory footprint of the JSP Servlet remains fairly constant between requests. Variables can be created in persistent space to allow the Servlet to perform caching, session tracking, and other functions not normally available in a stateless environment.

The JSP author is insulated from many of the issues involved with threaded programming. The JSP engine handles most of the work involved in creating, destroying, and managing threads. This frees the JSP author from many of the burdens of multithreaded programming. However, the JSP author needs to be aware of several aspects of multithreaded programming that affect JSP pages.

Threads can inadvertently cause harm to other threads. In these situations the JSP programmer needs to understand when and how to protect their pages from threading.

Persistence

Because the Servlet is created once and remains running as a constant instance, it allows persistent variables and objects to be created. Persistent variables and objects are shared between all threads of a single Servlet. Changes to these persistent objects are reflected in all threads.

From the perspective of the JSP author, all objects and variables created within declaration tags (`<%!...%>`) are persistent. Variables and objects created within a thread will not be persistent. Code inside of a scriptlet, expression, and action tags will be run within the new request thread and therefore will not create persistent variables or objects.

Having persistent objects allows the author to keep track of data between page requests. This allows in-memory objects to be used for caching, counters, session data, database connection pooling, and many other useful tasks.

Listing 4.1 shows a counter that uses persistent variables. When the page is first loaded the variable `counter` is created. Since the servlet remains running in memory the variable will remain until the servlet is restarted. Each time the page is requested the variable `counter` is incremented and displayed to the requestor. Each user should see a page count that is one higher than the last time the page was accessed.

Listing 4.1 counter.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">

<%!
  int counter;
%>

<HTML>

<STYLE>
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
```

Listing 4.1 counter.jsp (continued)

```
  font-size: 8pt; text-align: right;
}
</STYLE>

<BODY>
<DIV CLASS="pageFooter">
This page has been accessed
<% counter++;
   out.print(counter);
 %>
times since last restarted.
</DIV>
</BODY>

</HTML>
```

The Dangers of Threads

Unfortunately, object persistence also presents some potentially significant problems. To avoid these problems the JSP author needs to understand and steer away from these dangers.

The same factors that make persistence useful can also create a significant problem called a “Race Condition.” A Race Condition occurs when one thread is preparing to use data and a second thread modifies the data before the first thread has finished using the data.

Consider the above example (Listing 4.1) with two threads running. Take careful note of the value of the counter variable.

Thread 1 – UserA requests the page

Thread 2 – UserB requests the page

Thread 1 – `counter` increases by one

Thread 2 – `counter` increases by one

Thread 1 – `counter` is displayed for UserA

Thread 2 – `counter` is displayed for UserB

In this situation, UserA is actually viewing information that was intended for UserB. This is obviously not the expected result.

The problems caused by the above example are fairly trivial, UserA simply sees an incorrect page count. However, Race Conditions can just as easily result in very significant problems. Imagine if Race Condition occurred while billing a user for an online order.

It is good programming practice to resolve all Race Conditions, whether they appear trivial or not. Threads do not flow in a predictable order, so the results of a Race Condition may appear erratic. Race Conditions can be particularly difficult to spot and can turn from trivial to significant with very minor changes to the processing algorithms.

Thread Safety

In considering thread safety, it is important to first acknowledge that threading is a significant benefit to performance. Thread safety is almost always achieved by "disabling" threading for some portion of the code.

It is also important to understand that Race Conditions occur only with persistent variables. If all the variables and objects used by an application are created by threads, then there will be no Race Conditions. In these cases threading problems are not going to occur.

SingleThreadModel

The simplest method for gaining thread safety is also the least efficient. This is achieved by simply turning off threading for the entire page. Turning off threading is a poor option and should be avoided in most situations, because it avoids the potential problems by sacrificing many advantages.

JSP provides a means to turn off threading through the page directive attribute `isThreadSafe='false'`. This forces the page to be created under the `SingleThreadModel`, which allows only one request to be handled by the page at any time.

This option is still not 100% effective. Variables created in the `session` or `application` scope may still be affected by multiple instances.

synchronized()

A more practical and efficient manner of protecting variables from Race Conditions is to use Java's synchronized interface. This interface imposes a locking mechanism that allows only one thread at a time to process a particular block of code.

Entire methods can be synchronized and protected from Race Conditions, by using the `synchronized` keyword in the methods signature. This will protect all persistent variables accessed within the method. In this case only one thread can access the method at a time.

Blocks of code can also be synchronized by wrapping the code in a `synchronized()` block. In this case an argument is expected, which should represent the object to be locked.

CORE Note: Synchronized flag

Any object derived from `java.lang.Object` *can be used as the argument to the synchronized block. Every object has a special 'lock flag' that is used by* `synchronized()` *to manage threading (synchronized replaces both mutex and semaphore use in C programming). Primitive types in Java do not have this flag and therefore cannot be used as a lock for a synchronized block.*

In creating a synchronized block it is usually most efficient to use the object that most closely represents the data to be synchronized. For example, if a syncronized block is being written that modifies and writes to disk the `foo` *object it is best to use* `synchronized(foo)`*. Of course the* `this` *or* `page` *object can always be used, but that can create bottlenecks by locking the entire page each time the synchronized block is run.*

Listing 4.2 shows a new example of the counter page that uses a synchronized block. In this new example the two lines of code are grouped together within synchronized block. Only one thread is able to process the block at any given time. Each thread will lock the page object, increment the variable, display the variable, and then unlock the page object.

Listing 4.2 counter2.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">

<%!
  int counter;
%>

<HTML>

<STYLE>
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 8pt; text-align: right;
}
```

Listing 4.2 counter2.jsp (continued)

```
</STYLE>

<BODY>
<DIV CLASS="pageFooter">
This page has been accessed
<% synchronized (page) {
     counter++;
     out.print(counter);
   }
%>
times since last restarted.
</DIV>
</BODY>

</HTML>
```

4.3 The Implicit Objects

The Servlet also creates several objects to be used by the JSP engine. Many of these objects are exposed to the JSP developer and can be called directly without being explicitly declared.

The out Object

The major function of JSP is to describe data being sent to an output stream in response to a client request. This is output stream is exposed to the JSP author through the implicit `out` object.

The `out` object is an instantiation of a `javax.servlet.jsp.JspWriter` object. This object may represent a direct reference to the output stream, a filtered stream, or a nested `JspWriter` from another JSP. Output should never be sent directly to the output stream, because there may be several output streams during the lifecycle of the JSP.

The initial `JspWriter` object is instantiated differently depending on whether the page is buffered or not. By default, every JSP page has buffering turned on, which almost always improves performance. Buffering be easily turned off by using the `buffered='false'` attribute of the page directive.

A buffered `out` object collects and sends data in blocks, typically providing the best total throughput. With buffering the `PrintWriter` is created when the first block is sent, actually the first time that `flush()` is called.

With unbuffered output the `PrintWriter` object will be immediately created and referenced to the `out` object. In this situation, data sent to the `out` object is immediately sent to the output stream. The `PrintWriter` will be created using the default settings and header information determined by the server.

CORE Note: HTTP Headers and Buffering

HTTP uses response headers to both describe the server and define certain aspects of the data begin sent to the client. This might include the MIME content type of the page, new cookies, a forwarding URL, or other HTTP "actions."

JSP allows an author to change aspects of the response headers right up until the `OutputStream` *is created. Once the* `OutputStream` *is established the header information cannot be modified, as it is already sent to the client.*

In the case of a buffered `out` *object the* `OutputStream` *is not established until the first time that the buffer is flushed. When the buffer gets flushed depends largely on the* `autoFlush` *and* `bufferSize` *attributes of the page directive. It is usually best to set the header information before anything is sent to the* `out` *object.*

It is very difficult to set page headers with an unbuffered `out` *object. When an unbuffered page is created the* `OutputStream` *is established almost immediately.*

The sending headers after the `OutputStream` *has been established can result in a number of unexpected behaviors. Some headers will simply be ignored, others may generate exceptions such as* `IllegalStateException`.

The `JspWriter` object contains most of the same methods as the `java.io.PrintWriter` class. However, `JspWriter` has some additional methods designed to deal with buffering. Unlike the `PrintWriter` object, `JspWriter` throws `IOExceptions`. In JSP these exceptions need to be explicitly caught and dealt with. More about the `out` object is covered in Chapter 6.

CORE Note: autoFlush()

The default behavior of buffering in JSP is to automatically flush the buffer when it becomes full. However, there are cases where a JSP is actually talking directly to another application. In these cases the desired behavior might be to throw an exception if the buffer size is exceeded.

Setting the `autoFlush='false'` *attribute of the page directives will cause a buffer overflow to throw an exception.*

The request Object

Each time a client requests a page the JSP engine creates a new object to represent that request. This new object is an instance of `javax.servlet.http.HttpServletRequest` and is given parameters describing the request. This object is exposed to the JSP author through the `request` object.

Through the `request` object the JSP page is able to react to input received from the client. Request parameters are stored in special name/value pairs that can be retrieved using the `request.getParameter(name)` method.

The request object also provides methods to retrieve header information and cookie data. It provides means to identify both the client and the server—as previously seen in Chapter 2 (Listing 2.5 uses `request.getRequestURI()` and `request.getServerName()` to identify the server).

The `request` object is inherently limited to the request scope. Regardless of how the page directives have set the scope of the page, this object will always be recreated with each request. For each separate request from a client there will be a corresponding `request` object.

Additional information on using the request object and it methods will be discussed in Chapter 5.

The response Object

Just as the server creates the `request` object, it also creates an object to represent the response to the client. The object is an instance of `javax.servlet.http.HttpServletResponse` and is exposed to the JSP author as the `response` object.

The `response` object deals with the stream of data back to the client. The `out` object is very closely is related to the `response` object. The `response` object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP author can add new cookies or date stamps,

change the MIME content type of the page, or start "server-push" methods. The `response` object also contains enough information on the HTTP to be able to return HTTP status codes, such as forcing page redirects.

Additional information on using the `response` object and its methods will be discussed in Chapter 6.

The pageContext Object

The `pageContext` object is used to represent the entire JSP page. It is intended as a means to access information about the page while avoiding most of the implementation details.

This object stores references to the `request` and `response` objects for each request. The `application`, `config`, `session`, and `out` objects are derived by accessing attributes of this object. The `pageContext` object also contains information about the directives issued to the JSP page, including the buffering information, the `errorPageURL`, and page scope.

The `pageContext` object does more than just act as a data repository. It is this object that manages nested JSP pages, performing most of the work involved with the `forward` and `include` actions. The `pageContext` object also handles uncaught exceptions.

From the perspective of the JSP author this object is useful in deriving information about the current JSP page's environment. This can be particularly useful in creating components where behavior may be different based on the JSP `page` directives.

The session object

The `session` object is used to track information about a particular client while using stateless connection protocols, such as HTTP. Sessions can be used to store arbitrary information between client requests.

Each session should correspond to only one client and can exist throughout multiple requests. Sessions are often tracked by URL rewriting or cookies, but the method for tracking of the requesting client is not important to the `session` object.

The `session` object is an instance of `javax.servlet.http.HttpSession` and behaves exactly the same way that session objects behave under Java Servlets.

Additional information on the session object and its methods will be discussed in Chapter 7.

The application Object

The `application` object is direct wrapper around the `ServletContext` object for the generated Servlet. It has the same methods and interfaces that the `ServletContext` object does in programming Java Servlets.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the `jspDestroy()` method, the JSP page is recompiled, or the JVM crashes. Information stored in this object remains available to any object used within the JSP page.

The `application` object also provides a means for a JSP to communicate back to the server in a way that does not involve "requests." This can be useful for finding out information about the MIME type of a file, sending log information directly out to the servers log, or communicating with other servers.

The config Object

The `config` object is an instantiation of `javax.servlet.ServletConfig`. This object is a direct wrapper around the `ServletConfig` object for the generated servlet. It has the same methods and interfaces that the `ServletConfig` object does in programming Java Servlets.

This object allows the JSP author access to the initialization parameters for the Servlet or JSP engine. This can be useful in deriving standard global information, such as the paths or file locations.

The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page.

When the JSP page is first instantiated the `page` object is created by obtaining a reference to the `this` object. So, the `page` object is really a direct synonym for the `this` object.

However, during the JSP lifecycle, the `this` object may not refer to the page itself. Within the context of the JSP page, The `page` object will remain constant and will always represent the entire JSP page.

The exception Object

The error handling methods described in Chapter 3 utilizes this object. It is available only when the previous JSP page throws an uncaught exception and the `<%@ page errorPage="..." %>` tag was used.

The `exception` object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

4.4 The JSP lifecycle

The JSP engine uses three methods to manage the lifecycle of a JSP page and its generated servlet.

The heart of the JSP page is processed using a generated method called _jspService. This is created and managed by the JSP engine itself. _jspService should never be managed by the JSP author; doing so could cause disastrous results. The _jspService method represents the bulk of the JSP page, handling all requests and responses. In fact, new threads are effectively calling the _jspService method.

CORE Note: Reserved names

The JSP specification specifically reserves the methods and variables that begin with `jsp`, `_jsp`, `jspx`, *and* `_jspx`. *Methods and variables with these names may be accessible to the JSP author, however new methods and variables should not be created. The JSP engine expects to have control over these methods and variables, so changing them or creating new ones may result in erratic behavior.*

Two other methods, `jspInit()` and `jspDestroy()`, are designed to be overridden by the JSP author. In fact these methods do not exist unless specifically created by the JSP author. They play a special role in managing the lifecycle of a JSP page.

jspInit()

method signature: void jspInit()

`jspInit()` is a method that is run only once when the JSP page is first requested. `jspInit()`is guaranteed to be completely processed before any single request is handled. It is effectively the same as the `init()` method in Java Servlets and Java Applets.

`jspInit()` allows the JSP author a means to create or load objects that may be needed for every request. This can be useful for loading state information, creating database connection pools, and any task that only needs to happen once when the JSP page is first started.

jspDestroy()

method signature: void jspDestroy()

The server calls the `jspDestroy()` method when the Servlet is unloaded from the JVM. It is effectively the same as the `destroy()` method in Java Servlets and Java Applets.

Unlike `jspInit()` this method is not guaranteed to execute. The server will make its best effort attempt to run the method after each thread. Since this method occurs at the end of processing, there are situations—such as the server crashing—where `jspDestroy()` may not be executed.

`jspDestroy()` allows the JSP author a means to execute code just before the Servlet has finished. This is commonly used to free up resources or close connections that are still open. It can also be useful to store state information or other information that should be stored between instances.

JSP Lifecycle Overview

On first request or precompile `jspInit()` will be called, at which point the page is "running" waiting for requests. Now `_jspService` handles most transactions, picking up requests, running threads through, and generating responses. Finally, when a signal is received to shutdown the `jspDestroy()` method is called. The overall lifecycle of a JSP page is shown in Figure 4–1:

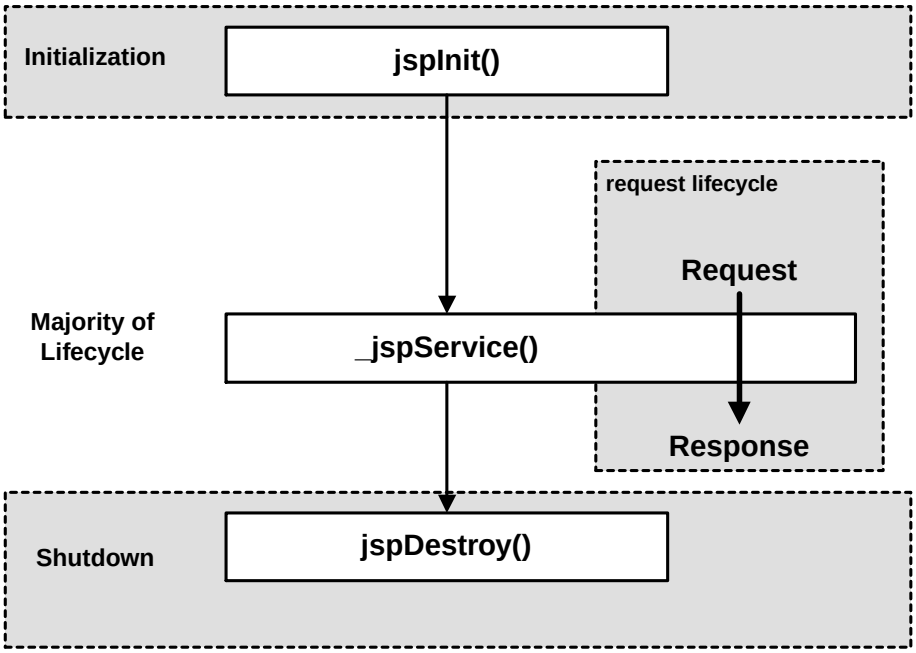


Figure 4–1 The JSP Lifecycle

The counter using jspInit() and jspDestroy()

The previous example of a page counter used a variable stored only in the memory of the running Servlet. It was never written to disk, so if the JSP page is restarted the variable would be reset.

The example shown in Listing 4.3 recovers the variable by using the `jspInit()` method to load the value of the variable when the page is first started. The example also uses the `jspDestroy()` method to write the value of variable to be recovered next time the JSP page is restarted.

Listing 4.3 counter3.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Final//EN">

<%@ page import="java.io.*" %>

<%!
  int counter = 0;

  public void jspInit() {
    try {
      FileInputStream countFile =
                      new FileInputStream ("counter.dat");
      DataInputStream countData =
                        new DataInputStream (countFile);
      counter = countData.readInt();
    }
    catch (FileNotFoundException ignore) {
 // No file indicates a new counter.
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void jspDestroy() {
    try {
      FileOutputStream countFile =
                     new FileOutputStream ("counter.dat");
      DataOutputStream countData =
                       new DataOutputStream (countFile);
      countData.writeInt(counter);
    }
    catch (IOException e) {
      e.printStackTrace();
    }
  }
     %>

<HTML>

<STYLE>
```

Listing 4.3 counter3.jsp (continued)

```
.pageFooter {
  position: absolute; top: 590px;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 8pt; text-align: right;
}
</STYLE>

<BODY>
<DIV CLASS="pageFooter">
This page has been accessed
<%

  synchronized(page) {
    counter++;
    out.print(counter);
  }
%>
times.
</DIV>
</BODY>

</HTML>
```

4.5 A JSP Compiled

Many implementations of JSP engines leave the Servlet source code that they create in a working directory. Under many engines this is an option that must be explicitly turned on, but it is usually a trivial task to enable.

Reading through the generated source code can be extremely useful in debugging problems that are not readily apparent in the JSP page. In addition, this source code can provide experienced Java developers with additional insight about the inner workings of the JSP implementation.

Listing 4.4 shows the compiled source code from the most recent counter. The source code in this listing is generated from Apache Jakarta Project's Tomcat v3.1; other JSP engines will probably produce slightly different results. The source code was also modified slightly to fit better on the page.

Listing 4.4 counter3_jsp.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.io.*;

public class _0002fcounter3_0002ejspcounter3_jsp_0
                                  extends HttpJspBase {

 // begin [file="/counter3.jsp";from=(4,3);to=(35,0)]

    int counter = 0;

    public void jspInit() {
      try {
        FileInputStream countFile =
                       new FileInputStream ("counter.dat");
        DataInputStream countData =
                         new DataInputStream (countFile);
        counter = countData.readInt();
      } catch (FileNotFoundException ignore) {
        // No file indicates a new counter.
      } catch (IOException e) {
        e.printStackTrace();
      }
    }

    public void jspDestroy() {
      try {
        FileOutputStream countFile =
```

Listing 4.4 counter3_jsp.java (continued)

```
                    new FileOutputStream ("counter.dat");
        DataOutputStream countData =
                        new DataOutputStream (countFile);
        countData.writeInt(counter);
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    // end

    static { }

    public _0002fcounter3_0002ejspcounter3_jsp_0( ) { }

    private static boolean _jspx_inited = false;

   public final void _jspx_init() throws JasperException {
}

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

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
```

Listing 4.4 counter3_jsp.java (continued)

```
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html");
           pageContext = _jspxFactory.getPageContext(this,
                                      request, response,
                                     "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

      // begin [file="/counter3.jsp";from=(0,0);to=(2,0)]
                out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//
DTD HTML 4.0 Final//EN\">\r\n\r\n");
      // end
      // begin [file="/counter3.jsp";from=(2,30);to=(4,0)]
                out.write("\r\n\r\n");
      // end
      // begin [file="/counter3.jsp";from=(35,2);to=(50,0)]

out.write("\r\n\r\n<HTML>\r\n\r\n<STYLE>\r\n.pageFooter
{\r\n  position: absolute; top: 590px;\r\n  font-family:
Arial, Helvetica, sans-serif; \r\n  font-size: 8pt; text-
align: right; \r\n}\r\n</STYLE>\r\n\r\n<BODY>\r\n<DIV
CLASS=\"pageFooter\">\r\nThis page has been accessed
\r\n");
      // end
      // begin [file="/counter3.jsp";from=(50,2);to=(55,0)]

                  synchronized(page) {
                    counter++;
                    out.print(counter);
                  }
      // end
```

Listing 4.4 counter3_jsp.java (continued)

```
      // begin [file="/counter3.jsp";from=(55,2);to=(62,0)]
                out.write(" \r\ntimes.\r\n</DIV>\r\n</
BODY>\r\n\r\n</HTML>\r\n\r\n");
      // end

        } catch (Exception ex) {

            if (out.getBufferSize() != 0)
                out.clear();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

4.6 Performance Tuning the Servlet

There are several aspects of Java programming that can have heavy impacts on the performance of a JSP page.

Some of these are not JSP specific, but simply good Java programming techniques overall. The ones listed are the most common efficiency mistakes.

Avoid appending with concatenation

In the course of development it is very easy to use the concatenation operator (+) to join `String` objects. For example:

```
String output;
output += “Item:  “ + item +  “  “;
output += “Price: “ + price + “  “;
println (output);
```

Unfortunately, it is easy to forget that the `String` object is immutable and is not designed to contain modifiable data. The `StringBuffer` object is

designed to perform manipulation of strings. Any time a `String` object is modified it really just creates a new `StringBuffer` and a series of new `String` objects, then all the `strings` are appended into the `StringBuffer`, and the old `String` is replaced with the results of `StringBuffer.toString()`.

While processing the above code there will be several new `String` and `StringBuffer` objects created. It's usually significantly more efficient to convert the `String` into a `StringBuffer` or even start out with a `StringBuffer` object. For example:

```
StringBuffer output;
output.append(new String("Item:  "));
output.append(item);
output.append(new String(" "));
output.append(new String("Price: "));
output.append(price)
output.append(new String(" "));
println (output.toString());
```

Use syncronize() carefully

While protecting objects from thread Race Conditions is extremely important, it is easy to create performance bottlenecks in the process of synchronizing.

Make sure that synchronized blocks contain the fewest lines of code possible. When many threads are waiting on a block of code, even reducing one line of code within the synchronized block can make a large difference.

Also, it is always good to synchronize on the most appropriate locking object as possible. Usually it is best to synchronize on the object that is threatened by the Race Condition. Avoid using the `this` or `page` object as the locking object as much as possible.

Overall it is obvious that the JSP engine is built to take advantage of the extremely powerful Java Servlet architecture. While threading is a significant enhancement, care needs to be taken to prevent the "Race Condition." Other techniques based on Java Servlet tuning can be used to fine-tune the performance of the JSP page and the underlying Java Servlet. Chapter 5 moves on to further discuss the `request` object and how it can be used to create dynamic Web applications.