JSP Overview

JSP is the latest Java technology for web application development, and is based on the servlet technology. While servlets are great in many ways, they are generally reserved for programmers. we look at the problems that JSP technology solves, the anatomy of a JSP page, the relationship between servlets and JSP, and how a JSP page is processed by the server.

In any web application, a program on the server processes requests and generates responses. In a simple one-page application, such as an online bulletin board, you don't need to be overly concerned about the design of this piece of code; all logic can be lumped together in a single program. But when the application grows into something bigger (spanning multiple pages, with more options and support for more types of clients) it's a different story. The way your site is designed is critical to how well it can be adapted to new requirements and continues to evolve. ***The good news is that JSP technology can be used in all kinds of web applications, from the simplest to the most complex.*** Therefore, this chapter also introduces the primary concepts in the design model recommended for web applications, and the different roles played by JSP and other Java technologies in this model.

The Problem with Servlets

In many Java servlet-based applications, processing the request and generating the response are both handled by a single servlet class.

A example servlet looks like this:

```
public class OrderServlet extends HttpServlet
{
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
 {
response.setContentType("text/html");
PrintWriter out = response.getWriter( );
if (isOrderInfoValid(request))
{
saveOrderInfo(request);
out.println("<html>");
out.println(" <head>");
out.println(" <title>Order Confirmation</title>");
```

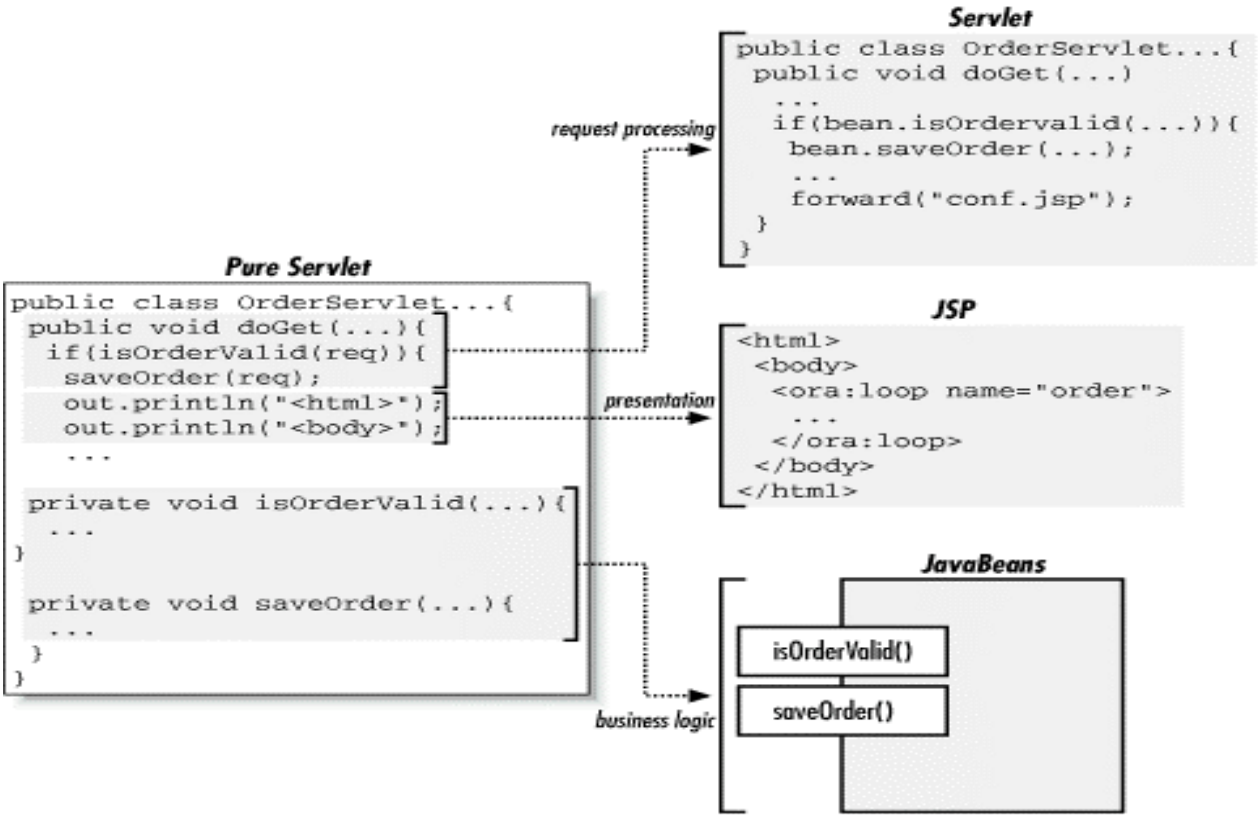
```
out.println(" </head>");
out.println(" <body>");
out.println(" <h1>Order Confirmation</h1>");
renderOrderInfo(request);
out.println(" </body>");
out.println("</html>");
}
…
```

If you're not a programmer, don't worry about all the details in this code. The point is that the servlet contains request processing and business logic (implemented by methods such as isOrderInfoValid( ) and saveOrderInfo( )) and also generates the response HTML code, embedded directly in the servlet code using println( ) calls. A more structured servlet application isolates different pieces of the processing in various reusable utility classes, and may also use a separate class library for generating the actual HTML elements in the response. ***But even so, the pure servlet-based approach still has a few problems***:

- Detailed Java programming knowledge is needed to develop and maintain all aspects of the application, since the processing code and the HTML elements are lumped together.
- Changing the look and feel of the application, or adding support for a new type of client, requires the servlet code to be updated and recompiled.
- It's hard to take advantage of web page development tools when designing the application interface.
- If such tools are used to develop the web page layout, the generated HTML must then be manually embedded into the servlet code, a process that is time-consuming, error-prone, and extremely boring.

Adding JSP to the puzzle lets you solve these problems by separating the request processing and business logic code from the presentation, Instead of embedding HTML in the code, you place all static HTML in JSP pages, just as in a regular web page, and add a few JSP elements to generate the dynamic parts of the page. The request processing can remain the domain of servlet programmers, and the business logic can be handled by JavaBeans and Enterprise JavaBeans (EJB) components.

Separation of request processing, business logic, and presentation



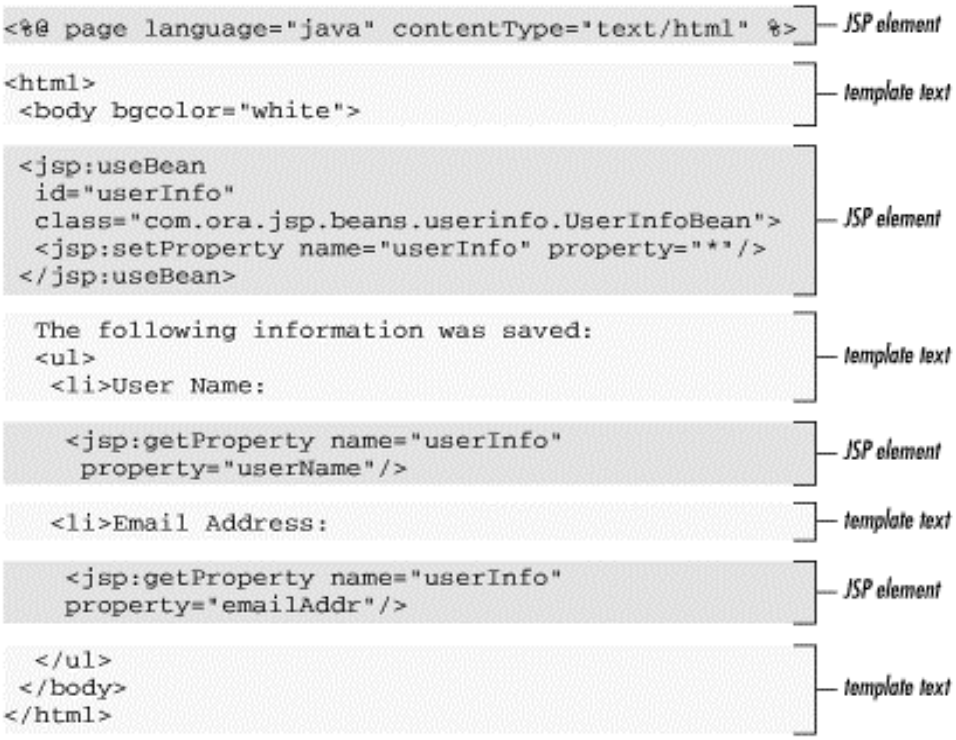
Separating the request processing and business logic from presentation makes it possible to divide the development tasks among people with different skills. Java programmers implement the request processing and business logic pieces, web page authors implement the user interface, and both groups can use best-of-breed development tools for the task at hand. The result is a much more productive development process. It also makes it possible to change different aspects of the application independently, such as changing the business rules without touching the user interface. This model has clear benefits even for a web page author without programming skills who is working alone. A page author can develop web applications with many dynamic features, using generic Java components provided by open source projects or commercial companies.

The Anatomy of a JSP Page

A JSP page is simply a regular web page with JSP elements for generating the parts of the page that differ for each request, as Everything in the page that is not a JSP element is called *template text* .

Template text can really be any text: HTML, WML, XML, or even plain text. Since HTML is by far the most common web page language in use today, most of the descriptions and examples in this book are HTML-based, but keep in mind that JSP has no dependency on HTML; it can be used with any markup language. Template text is always passed straight through to the browser.

Template text and JSP elements



When a JSP page request is processed, the template text and the dynamic content generated by the JSP elements are merged, and the result is sent as the response to the browser.

JSP Elements

There are three types of elements with Java Server Pages: ***directive*, *action*, and *scripting* elements**.

The directive elements, shown below, are used to specify information about the page itself that remains the same between page requests, for example, the scripting language used in the page, whether session tracking is required, and the name of a page that should be used to report errors, if any.

Element	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements
<%@ include ... %>	Includes a file during the translation phase
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

Action elements typically perform some action based on information that is required at the exact time the JSP page is requested by a client. An action element can, for instance, access parameters sent with the request to do a database lookup. It can also dynamically generate HTML, such as a table filled with information retrieved from an external system.
The JSP specification defines a few standard action elements, listed below, and includes a framework for developing custom action elements. A custom action element can be developed by a programmer to extend the JSP language.

Element	Description
<jsp:useBean>	Makes a JavaBeans component available in a page
<jsp:getProperty>	Gets a property value from a JavaBeans component and adds it to the response
<jsp:setProperty>	Sets a JavaBeans property value
<jsp:include>	Includes the response from a servlet or JSP page during the request processing phase
<jsp:forward>	Forwards the processing of a request to a servlet or JSP page
<jsp:param>	Adds a parameter value to a request handed off to another servlet or JSP page using <jsp:include> or <jsp:forward>
<jsp:plugin>	Generates HTML that contains the appropriate client browser-dependent elements (OBJECT or EMBED) needed to execute an Applet with the Java Plugin software

Scripting elements, allow you to add small pieces of code to a JSP page. Like actions, they are also executed when the page is requested

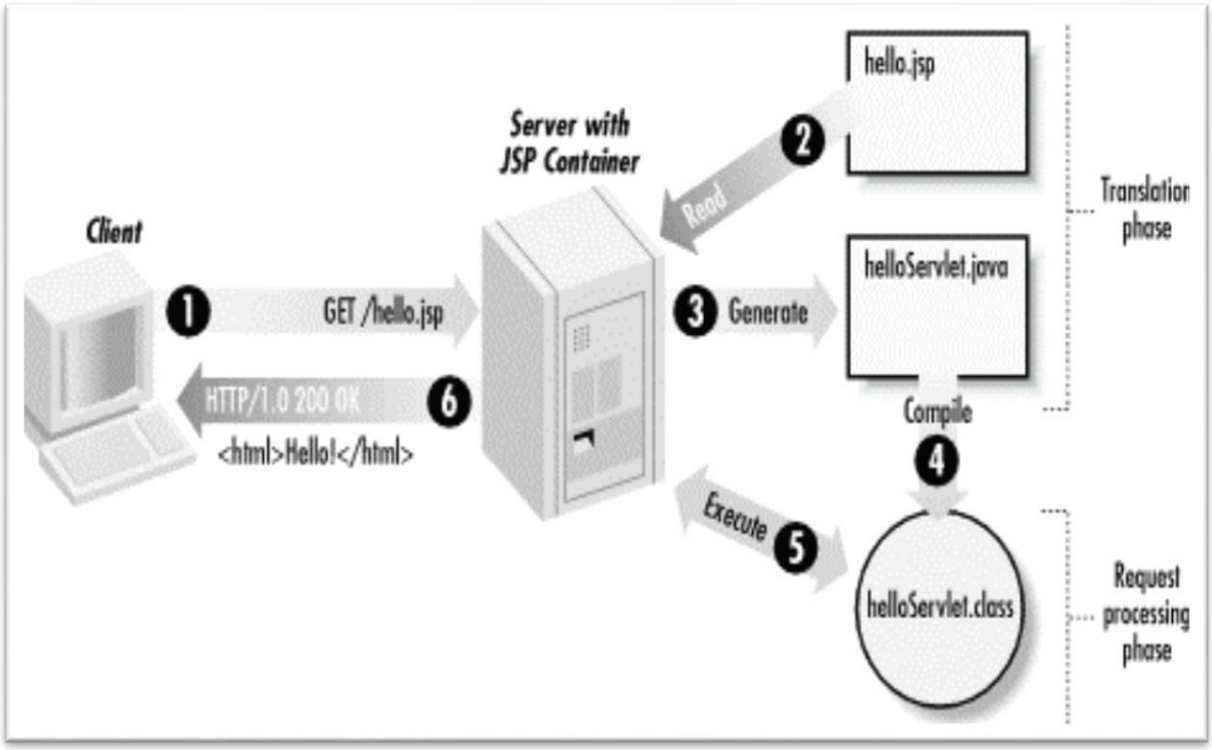
Element	Description
<% ... %>	Scriptlet, used to embed scripting code.
<%= ... %>	Expression, used to embed Java expressions when the result shall be added to the response. Also used as runtime action attribute values.
<%! ... %>	Declaration, used to declare instance variables and methods in the JSP page implementation class.

JSP Processing

A JSP page cannot be sent as-is to the browser; all JSP elements must first be processed by the server. This is done by turning the JSP page into a servlet, and then executing the servlet. Just as a web server needs a servlet container to provide an interface to servlets, the server needs a JSP container to process JSP pages. The JSP container is often implemented as a servlet configured to handle all requests for JSP pages. In fact, these two containers - a servlet container and a JSP container - are often combined into one package under the name *web container*. A JSP container is responsible for converting the JSP page into a servlet and compiling the servlet. ***These two steps form the translation phase.*** The JSP container automatically initiates the translation phase for a page when the first request for the page is received. The translation phase takes a bit of time, of course, so a user notices a slight delay the first time a JSP page is requested. The translation phase can also be initiated explicitly; this is referred to as *pre compilation* of a JSP page.

The JSP container is also responsible for invoking the JSP page implementation class to process each request and generate the response. ***This is called the request processing phase.***

The two phases are illustrated in



JSP page translation and processing phases

As long as the JSP page remains unchanged, any subsequent processing goes straight to the request processing phase. When the JSP page is modified, it goes through the translation phase again before entering the request processing phase.
So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. And, except for the translation phase, a JSP page is handled exactly like a regular servlet: it's loaded once and called repeatedly, until the server is shut down. By virtue of being an automatically generated servlet, ***a JSP page inherits all of the advantages of servlets: platform and vendor independence, integration, efficiency, scalability, robustness, and security.***

Hello World JSP Page

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World</h1>
It's <%= new java.util.Date( ).toString( ) %>.
</body>
</html>
```

This is as simple as it gets. A JSP page is a regular HTML page, except that it may also contain JSP elements like the highlighted element in this example. This element inserts the same Java code in the page as was used in the servlet to add the current date and time. If you compare this JSP page to the corresponding servlet, you see that the JSP page can be developed using any web page editor that allows you to insert extra, non-HTML elements. And the same tool can later be used to easily modify the layout. This is a great advantage over a servlet with embedded HTML.The JSP page is automatically turned into a servlet the first time it's requested.

The Equivalent Servlet code for Hello World JSP page

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

public class Helloworldservlet extends HttpServlet {

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException
{
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
     Date d=new Date( );
    String da = d.toString( );

     out.println("<HTML><BODY>");

    out.println( da );

     out.println("</BODY></HTML>");
 out.close( );
}
}
```

JSP Application Design with MVC

JSP technology can play a part in everything from the simplest web application, such as an online phone list or an employee vacation planner, to full-fledged enterprise applications, such as a human resource application or a sophisticated online shopping site. How large a part JSP plays differs in each case, of course. Here, we introduce a design model suitable for both simple and complex applications called ***Model-View-Controller (MVC).***The key point of using MVC is to separate components into three distinct units: ***the Model, the View, and the Controller***. In a server application, we commonly classify the parts of the application as: ***business logic, presentation, and request processing.***

Business logic is the term used for the manipulation of an application's data, i.e., customer, product, and order information. ***Presentation*** refers to how the application is displayed to the user, i.e., the position, font, and size. And finally, ***request processing*** is what ties the business logic and presentation parts together. In MVC terms, ***the Model corresponds to business logic and data, the View to the presentation logic, and the Controller to the request processing.***

Why use this design with JSP? The answer lies primarily in the first two elements. Remember that an application data structure and logic (the Model) is typically the most stable part of an application, while the presentation of that data (the View) changes fairly often. Just look at all the face-lifts that web sites have gone through to keep up with the latest fashion in web design. Yet, the data they present remains the same.
Another common example of why presentation should be separated from the business logic is that you may want to present the data in different languages or present different subsets of the data to internal and external users. Access to the data through new types of devices, such as cell phones and Personal Digital Assistants (PDAs), is the latest trend. Each client type requires its own presentation format. It should come as no surprise, then, that separating business logic from presentation makes it easier to evolve an application as the requirements change; new presentation interfaces can be developed without touching the business logic.

Generating Dynamic Content

JSP is all about generating dynamic content: content that differs based on user input, time of day, the state of an external system, or any other runtime conditions. JSP provides you with lots of tools for generating this content. We develop a page for displaying the current date and time, and look at the JSP directive element and how to use JavaBeans in a JSP page along the way. Next, we look at how to process user input in your JSP pages and make sure it has the appropriate format. We also look at how you can convert special Characters in the output, so they don't confuse the browser.

JSP pages should have the file extension *.jsp* , which tells the server that the page needs to be processed by the JSP container. Without this clue, the server is unable to distinguish a JSP page from any other type of file and sends it unprocessed to the browser. When working with JSP pages, you really just need a regular text editor such as Notepad on Windows or Emacs on Unix.

JSP Page Showing the Current Date and Time (date.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
The current time at the server is:
<ul>
<li>Date: <jsp:getProperty name="clock" property="date" />
<li>Month: <jsp:getProperty name="clock" property="month" />
<li>Year: <jsp:getProperty name="clock" property="year" />
<li>Hours: <jsp:getProperty name="clock" property="hours" />
<li>Minutes: <jsp:getProperty name="clock" property="minutes" />
</ul>
</body>
</html>
```

The *date.jsp* page displays the current date and time.

The HTML elements are used as-is, defining the layout of the page. If you use the View Source function in your browser, you notice that none of the JSP elements are visible in the page source. That's because the JSP elements are processed by the

server when the page is requested, and only the resulting output is sent to the browser.

Using JSP Directives

Directives are used to specify attributes of the page itself, primarily those that affect how the page is converted into a Java servlet.
There are three JSP directives:

page, include, and taglib.

JSP pages typically start with a page directive that specifies the scripting language and the content type for the page:

<%@ page language="java" contentType="text/html" %>

A JSP directive element starts with a directive-start identifier (<%@) followed by the directive name (e.g., page) and directive attributes, and ends with %>. A directive contains one or more attribute name/value pairs (e.g., language="java"). Note that JSP element and attribute names are case-sensitive, and in most cases the same is true for attribute values. For instance, the language attribute value must be java, not Java. All attribute values must also be enclosed in single or double quotes.

The language attribute specifies the scripting language used in the page. The content Type attribute specifies the type of content the page produces. The most common values are text/html for HTML content and text/plain for preformatted, plain text.

Using Template Text

Besides JSP elements, notice that the page shown below contains mostly regular HTML:

```
...
<html>
<body bgcolor="white">
...
The current time at the server is:
<ul>
<li>Date: ...
<li>Month: ...
```

Year: ...
Hours: ...
Minutes: ...

</body>
</html>

In JSP parlance, this is called *template text*. Everything that's not a JSP element, such as a directive, action, or scripting element, is template text. Template text is sent to the browser as-is. This means you can use JSP to generate any type of text-based output, such as XML, WML, or even plain text. The JSP container doesn't care what the template text is.

Using Scripting Elements

When you develop a JSP-based application, I recommend that you try to place all Java code in JavaBeans, in custom actions, or in regular Java classes. However, to tie all these components together, you sometimes need additional code embedded in the JSP pages themselves.

Implicit JSP Objects

When you use scripting elements in a JSP page, you always have access to a number of objects that the JSP container makes available. These are called ***implicit objects***.

Variable Name	Java Type
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
pageContext	javax.servlet.jsp.PageContext
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
out	javax.servlet.jsp.JspWriter
config	javax.servlet.ServletConfig
page	java.lang.Object
exception	java.lang.Throwable

Here is some more information about each of these implicit objects:

request

The request object is an instance of the class named javax.servlet.http.HttpServletRequest. This object provides methods that let you access all the information that's available about the current request, such as request parameters, attributes, headers, and cookies.

response

The response object represents the current response message. It's an instance of the javax.servlet.http.HttpServletResponse class, with methods for setting headers and the status code and for adding cookies. It also provides methods related to session tracking. These methods are the response methods you're most likely to use.

session

The session object allows you to access the client's session data, managed by the server. It's an instance of the javax.servlet.http.HttpSession class. Typically you do not need to directly access this object, since JSP also lets you access the session data through action elements.

application

The application object is another object that you typically access indirectly through action elements. It's an instance of the javax.servlet.ServletContext class. This object is used to hold references to other objects that more than one user may require access to, such as a database connection shared by all application users. It also contains log( ) methods that you can use to write messages to the container's log file.

Out

The out object is an instance of javax.servlet.jsp.JspWriter. You can use the print( ) and println( ) methods provided by this object to add text to the response message body.

exception

The exception object is available only in error pages and contains information about a runtime error.

Conditional Processing

In most web applications, you produce different output based on runtime conditions, such as the state of a bean or the value of a request header such as UserAgent .If the differences are not too great, you can use JSP scripting elements to control which parts of the JSP page are sent to the browser, generating alternative outputs from the same JSP page. However, if the outputs are completely different, I recommend using a separate JSP page for each alternative and passing control from one page to another.

Conditional Greeting Page (greeting.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
<% if (clock.getHours( ) < 12) { %>
Good morning!
<% } else if (clock.getHours( ) < 17) { %>
Good day!
<% } else { %>
Good evening!
<% } %>
</body>
</html>
```

The <jsp:useBean> action is first used to create a bean. Besides making the bean available to other actions, such as <jsp:getProperty> and <jsp:setProperty>, the <jsp:useBean> action also creates a Java variable that holds a reference to the bean. The name of the variable is the name specified by the id attribute, in this case clock. The clock bean is then used in four scriptlets, together forming a complete Java if statement with template text in the if and else blocks:

<% if (clock.getHours( ) < 12) { %>
An if statement, testing if it's before noon, with a block start brace.
<% } else if (clock.getHours( ) < 17) { %>

The if block end brace and an else-if statement, testing if it's before 5:00 P.M., with its block start brace.
<% } else { %>
The else-if block end brace, and a final else block start brace, handling the case when it's after 5:00 P.M.
<% } %>
The final else block end brace.
The JSP container combines the code segment in the four scriptlets with code for writing the template text to the response body. The end result is that when the first if statement is true, "Good morning!" is displayed;
when the second if statement is true, "Good day!" is displayed; and if none of the if statements is true, the
final else block is used, displaying, "Good evening!"

Displaying Values

Besides using scriplets for conditional output, one more way to employ scripting elements is by using a JSP *expression element* to insert values into the response. A JSP expression element can be used instead of the <jsp:getProperty> action in some places, but it is also useful to insert the value of any Java expression that can be treated as a String.
An expression starts with <%= and ends with %>. Note that the only syntax difference compared to a scriptlet is the equals sign (=) in the start identifier. An example is:
<%= userInfo.getUserName( ) %>
The result of the expression is written to the response body. One thing is important to note: as opposed to statements in a scriptlet, the code in an expression must not end with a semicolon. This is because the JSP container combines the expression code with code for writing the result to the response body. If the expression ends with a semicolon, the combined code will not be syntactically correct.

Using an Expression to Set an Attribute

In all our JSP action element examples so far, the attributes are set to literal string values. But in many cases, the value of an attribute is not known when you write the JSP page; instead, the value must be calculated when the JSP page is requested. For situations like this, you can use a JSP expression as an attribute value. This is called a *request-time attribute value*. Here is an example of how this can be used to set an attribute of a fictitious log entry bean:

```
<jsp:useBean id="logEntry" class="com.foo.LogEntryBean" />
<jsp:setProperty name="logEntry" property="entryTime"
value="<%= new java.util.Date( ) %>" />
...
```

This bean has a property named entryTime that holds a timestamp for a log entry, while other properties hold the information to be logged. To set the timestamp to the time when the JSP page is requested, a <jsp:setProperty> action with a request-time attribute value is used. The attribute value is represented by the same type of JSP expression as in the previous snippet, here an expression that creates a new java.util.Date object. The requesttime attribute is evaluated when the page is requested, and the corresponding attribute is set to the result of the expression. As you might have guessed, any property you set this way must have a Java type matching the result of the expression. In this case, the entryDate property must be of type java.util.Date.

Declaring Variables and Methods

We have used two of the three JSP scripting elements: scriptlets and expressions. There's one more, called a ***declaration element***, which is used to declare Java variables and methods in a JSP page. Java variables can be declared either within a method or outside the body of all methods, like this:

```
public class SomeClass {
// Instance variable
private String anInstanceVariable;
// Method
```

```
public void doSomething( ) {
String aLocalVariable;
}
}
```

A variable declared outside the body of all methods is called an *instance variable*. Its value can be accessed from any method in the class, and it keeps its value even when the method that sets it returns. A variable declared within the body of a method is called a *local variable*. A local variable can be accessed only from the method where it's declared. When the method returns, the local variable disappears.
When you declare a variable within a scriptlet element instead of in a JSP declaration block, the variable ends up as a local variable in the generated servlet's request processing method.

Using a Declaration Element (counter.jsp)

```
<%@ page language="java" contentType="text/html" %>
<%! int globalCounter = 0; %>
<html>
<head>
<title>A page with a counter</title>
</head>
<body bgcolor="white">
This page has been visited: <%= ++globalCounter %> times.
<p>
<% int localCounter = 0; %>
This counter never increases its value:
<%= ++localCounter %>
</body>
</html>
```

The JSP declaration element is right at the beginning of the page, starting with <%! and ending with %>. Note the exclamation point (!) in the start identifier; that's what makes it a declaration as opposed to a scriptlet.The declaration element declares an instance variable named globalCounter, shared by all requests for thepage. In the body section of the page, a JSP expression increments the variable's value. Next comes ascriptlet, enclosed by <% and %>, that declares a local variable named localCounter. The last scriptlet increments the value of the local variable.

When you run this example, the globalCounter value increases every time you load the page, but localCounter stays the same.

A JSP declaration element can also be used to declare a method that can then be used in scriptlets in the same page. The only harm this could cause is that your JSP pages end up containing too much code, making it hard to maintain the application.

Method Declaration and Use (color.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<%!
String randomColor( ) {
java.util.Random random = new java.util.Random( );
int red = (int) (random.nextFloat( ) * 255);
int green = (int) (random.nextFloat( ) * 255);
int blue = (int) (random.nextFloat( ) * 255);
return "#" +
Integer.toString(red, 16) +
Integer.toString(green, 16) +
Integer.toString(blue, 16);
}
%>
<h1>Random Color</h1>
<table bgcolor="<%= randomColor( ) %>" >
<tr><td width="100" height="100"> </td></tr>
</table>
</body>
</html>
```

The method named randomColor( ), declared between <%! and %>, returns a randomly generated String in a format that can be used as an HTML color value. This method is then called from an expression element to set the background color for a table. Every time you reload this page, you see a single table cell with a randomly selected color.

Error Handling and Debugging

When you develop any application that's more than a trivial example, errors are inevitable. A JSP-based application is no exception. There are many types of errors you will deal with. Simple syntax errors in the JSP pages are almost a given during the development phase. And even after you have fixed all the syntax errors, you may still have to figure out why the application doesn't work as you intended due to design mistakes. The application must also be designed to deal with problems that can occur when it's deployed for production use. Users can enter invalid values and try to use the application in ways you never imagined. External systems, such as databases, can fail or become unavailable due to network problems. Since a web application is the face of a company, making sure it behaves well, even when the users misbehave and the world around it falls apart, is extremely important for a positive customer perception. Proper design and testing is the only way to accomplish this goal. Unfortunately, many developers seem to forget the hard-learned lessons from traditional application development when designing web applications. For instance, a survey of 100 e-commerce managers, conducted by *InternetWeek* magazine (April 3, 2000 issue), shows that 50% of all web site problems were caused by application coding errors. That's the highest ranking reason in the survey, ahead of poor server performance (38%), poor service provider performance (35%), and poor network performance (22%).

Dealing with Syntax Errors

The first type of error you will encounter is the one you, or your co-workers, create by simple typos: in other words, syntax error. The JSP container needs every JSP element to be written exactly as it's defined in the specification in order to turn the JSP page into a valid servlet class. When it finds something that's not right, it will tell you. But how easy it is to understand what it tells you depends on the type of error, the JSP container implementation, and sometimes, on how fluent you are in computer gibberish.

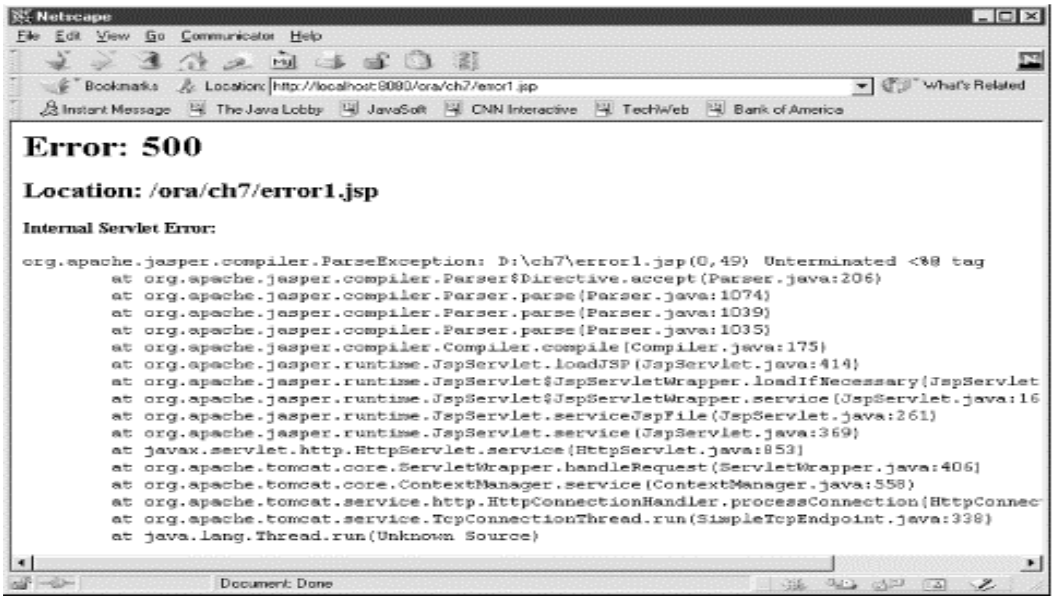
Element Syntax Errors

Let's first look at how Tomcat reports some typical syntax errors in JSP directives and action elements.

Improperly Terminated Directive (error1.jsp)

```
<%@ page language="java" contentType="text/html" >
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
The current time at the server is:
<ul>
<li>Date: <jsp:getProperty name="clock" property="date" />
<li>Month: <jsp:getProperty name="clock" property="month" />
<li>Year: <jsp:getProperty name="clock" property="year" />
<li>Hours: <jsp:getProperty name="clock" property="hours" />
<li>Minutes: <jsp:getProperty name="clock" property="minutes"
/>
</ul>
</body>
</html>
```

The syntax error here is that the page directive on the first line is not closed properly with %>; the percent sign is missing



Tomcat reports the error by sending an error message to the browser. This is the default behaviour for Tomcat, but it's not mandated by the JSP specification. The specification requires only that a response with the HTTP status code for a severe error (500) is returned, but how a JSP container reports the details is Vendor-specific. For instance, the error message can be written to a file instead of to the browser. If you use a container other than Tomcat, check the container documentation to see how it reports these types of errors.

Improperly Terminated Action (error2.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" >
The current time at the server is:
<ul>
<li>Date: <jsp:getProperty name="clock" property="date" />
<li>Month: <jsp:getProperty name="clock" property="month" />
<li>Year: <jsp:getProperty name="clock" property="year" />
<li>Hours: <jsp:getProperty name="clock" property="hours" />
<li>Minutes: <jsp:getProperty name="clock" property="minutes"
/>
</ul>
</body>
</html>
```

The syntax error here is almost the same as the "unterminated tag", but now it's the <jsp:useBean> action element that's not terminated properly.

Another common error is a typo in an attribute name, as shown below . In the first <jsp:getProperty> action, the name attribute is missing the e.

Mistyped Attribute Name (error3.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
```

```
The current time at the server is:
<ul>
<li>Date: <jsp:getProperty nam="clock" property="date" />
<li>Month: <jsp:getProperty name="clock" property="month" />
<li>Year: <jsp:getProperty name="clock" property="year" />
<li>Hours: <jsp:getProperty name="clock" property="hours" />
<li>Minutes: <jsp:getProperty name="clock" property="minutes"
/>
</ul>
</body>
</html>
```

Tomcat reports the problem like this:

```
D:\ch7\error3.jsp(7,10) getProperty: Mandatory attribute name
missing
```

In this case, the typo is in the name of a mandatory attribute, so Tomcat reports it as missing. If the typo is in the name of an optional attribute, Tomcat reports it as an invalid attribute name.

Scripting Syntax Errors

Unfortunately, syntax errors in scripting elements result in error messages that are much harder to interpret. This is because of the way the JSP container deals with scripting code when it converts a JSP page into a servlet. The container reads the JSP page and generates servlet code by replacing all JSP directives and actions with code that produces the appropriate result. To do this, it needs to analyze these types of elements in detail. If there's a syntax error in a directive or action element, it can easily tell which element is incorrect. Scripting elements, on the other hand, are more or less used as-is in the generated servlet code. A syntax error in scripting code is not discovered when the JSP page is read, but instead when the generated servlet is compiled. The compiler reports an error in terms of its location in the generated servlet code, with messages that don't always make sense to a JSP page author. Let's look at example to illustrate this. The last scriptlet, with a brace closing the last else block, is missing.

Missing End Brace (error4.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html>
<body bgcolor="white">
<jsp:useBean id="clock" class="java.util.Date" />
<% if (clock.getHours( ) < 12) { %>
Good morning!
<% } else if (clock.getHours( ) < 17) { %>
Good day!
<% } else { %>
Good evening!
</body>
</html>
```

Debugging a JSP-Based Application

For applications developed in compiled languages such as Java, C, or C++, a tool called a debugger is often used in this phase. A debugger steps through the program line by line or runs until it reaches a break point that you have defined, and lets you inspect the values of all variables in the program. With careful analysis of the program flow in runtime, you can discover why it works the way it does, and not the way you want it to.

There are debuggers for JSP as well, such as IBM's Visual Age for Java. This product lets you debug a JSP page exactly the same way as you would a program written in a more traditional programming language. But a real debugger is often overkill for JSP pages. If your pages are so complex that you feel you need a debugger, you may want to move code from the pages into JavaBeans or custom actions instead. These components can then be debugged with a standard Java debugger, which can be found in most Java Interactive Development Environments (IDEs). To debug JSP pages, another time-tested debugging approach is usually sufficient: simply add code to print variable values to the screen.

Sharing Data between JSP Pages, Requests, and Users

We can turn our attention to the JSP features and techniques needed to develop real applications. Any real application consists of more than a single page, and multiple pages often need access to the same information and server-side resources. When multiple pages are used to process the same request, for instance one page that retrieves the data the user asked for and another that displays it, there must be a way to pass data from one page to another.

In an application in which the user is asked to provide information in multiple steps, such as an online shopping application, there must be a way to collect the information received with each request and get access to the complete set when the user is ready. Other information and resources need to be shared among multiple pages, requests, and all users. Examples are information about currently logged-in users, database connection pool objects, and cache objects to avoid frequent database lookups.

You will see how using multiple pages to process a request leads to an application that's easier to maintain and expand, and learn about a JSP action that lets you pass control between the different pages.

Passing Control and Data Between Pages

As discussed earlier, one of the most fundamental features of JSP technology is that it allows for separation of request processing, business logic, and presentation, using what's known as the ***Model-View-Controller (MVC) model***. As you may recall, the roles of Model, View, and Controller can be assigned to different types of server-side components. In this part of the book, JSP pages are used for both the Controller and View roles and the Model role is played by either a bean or a JSP page. Using different JSP pages as Controller and View means that more than one page is used to process a request.

To make this happen, you need to be able to do two things:

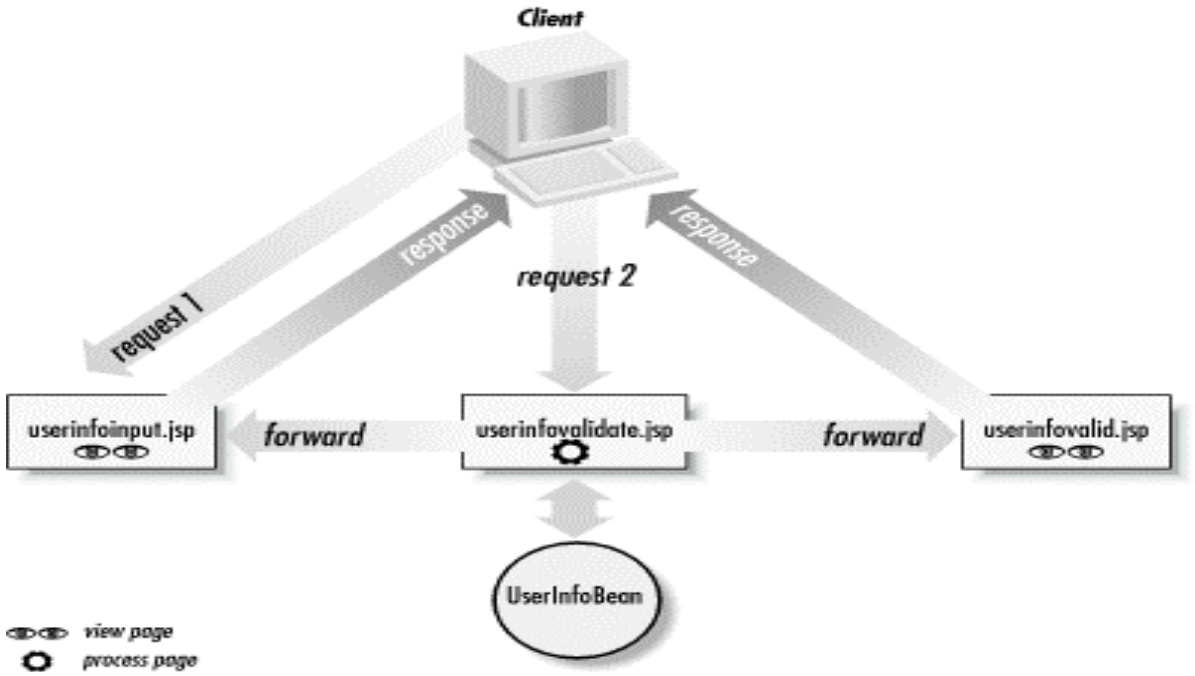
1. ***Pass control from one page to another.***
2. ***Pass data from one page to another.***

We can categorize the different aspects of the User Info example like this:

- ***Display the form for user input (presentation).***
- ***Validate the input (request processing and business logic).***
- ***Display the result of the validation (presentation).***

Let's use a separate JSP page for each aspect. The restructured application contains three JSP pages, as shown below

User Info application pages



Here's how it works. The *userinfoinput.jsp* page displays an input form. The user submits this form to *userinputvalidate.jsp* to validate the input. This page processes the request using the UserInfoBean and passes control (forwards) to either the *userinfoinput.jsp* page (if the input is invalid) or the *userinfovalid.jsp*page (if the input is valid). If valid, the *userinfovalid.jsp* page displays a "thank you" message. In this example, the UserInfoBean represents the Model, the *userinputvalidate.jsp* page the Controller, and *userinfoinput.jsp*

and *userinfovalid.jsp* represent the Views. If the validation rules change, a Java programmer can change the UserInfoBean implementation without touching any other part of the application. If the customer wants a different look, a page author can modify the View JSP pages without touching the request processing or business logic code.

Passing Control from One Page to Another

Before digging into the modified example pages, let's go through the basic mechanisms. the *userinfovalidate.jsp* page passes control to one of two other pages based on the result of the input validation. JSP supports this through the <jsp:forward> action:

```
<jsp:forward page="userinfovalid.jsp" />
```

This action stops processing one page and starts processing the page specified by the page attribute instead, called the *target page*. The control never returns to the original page.

The target page has access to all information about the request, including all request parameters. You can also add additional request parameters when you pass control to another page by using one or more nested
<jsp:param> action elements:

```
<jsp:forward page="userinfovalid.jsp" >
<jsp:param name="msg" value="Invalid email address" />
</jsp:forward>
```

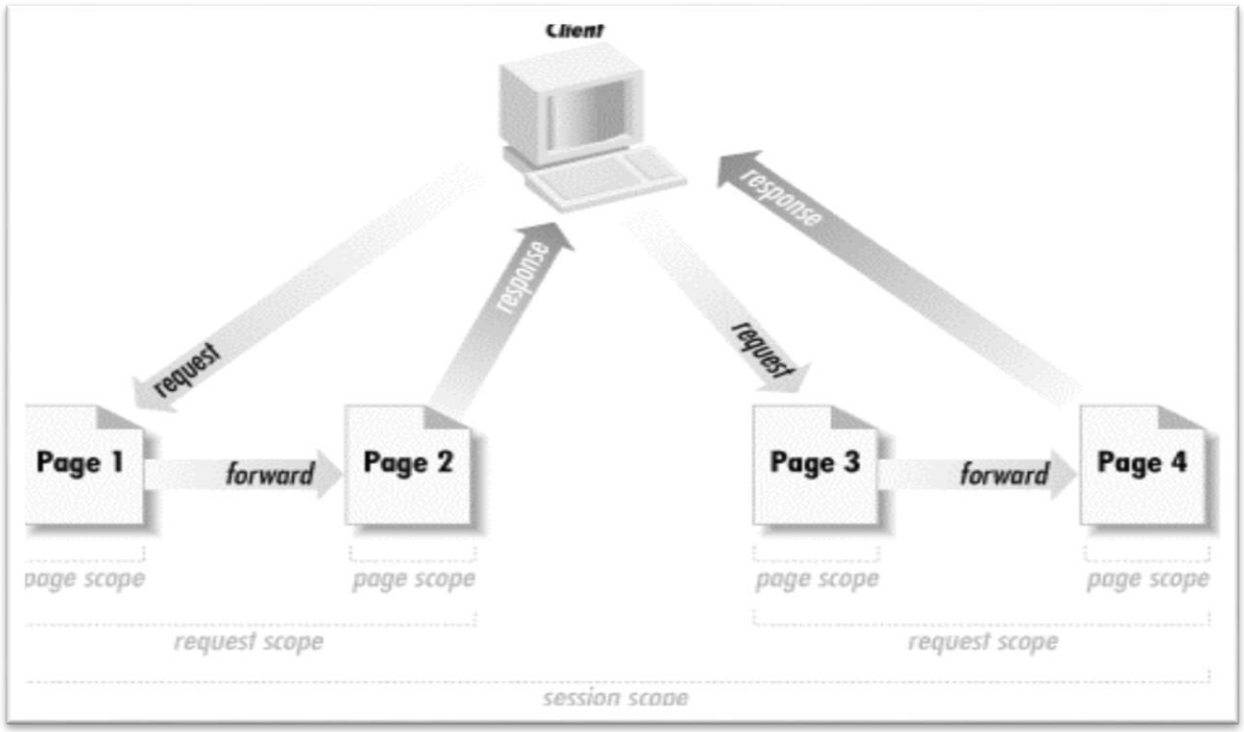
Parameters specified with <jsp:param> elements are added to the parameters received with the originalrequest. The target page, therefore, has access to both the original parameters and the new ones, and can access both types in the same way. If a parameter is added to the request using the name of a parameter that already exists, the new value is added to the list of values for the existing parameter.

The page attribute is interpreted relative to the location of the current page if it doesn't start with /. This is called a *page-relative path*. If the source and target page are located in the same directory, just use the name of the target page as the page attribute value, as in the previous example. You can also refer to a file in a different directory using notation like *../foo/bar.jsp* or */foo/bar.jsp*. When the page reference starts with /, it's interpreted relative to the top directory for the application's web page files. This is called a *contextrelative path*.

Passing Data from One Page to Another

JSP provides different *scopes* for sharing data objects between pages, requests, and users. The scope defines for how long the object is available and whether it's available only to one user or to all application users. The following scopes are defined:

- Page
- Request
- Session
- Application



Lifetime of objects in different scopes

Objects placed in the default scope, the ***page scope*** , are available only to actions and scriplets within one page. The ***request scope*** is for objects that need to be available to all pages processing the same request. The ***session scope*** is for objects shared by multiple requests by the same user, and the ***application scope*** is for objects shared by all users of the application.

The <jsp:useBean> action has a scope attribute that you use to specify in what scope the bean should be

placed. Here is an example:
<jsp:useBean id="userInfo" scope="request"
class="com.ora.jsp.beans.userinfo.UserInfoBean" />
The <jsp:useBean> action looks for a bean with the name specified by the id attribute in the specified scope.

If one already exists, it uses that one. If it cannot find one, it creates a new instance of the class specified by the class attribute and makes it available with the specified name within the specified scope. If you would like to perform an action only when the bean is created, place the elements in the body of the <jsp:useBean>
action:
<jsp:useBean id="userInfo" scope="request"
class="com.ora.jsp.beans.userinfo.UserInfoBean" >
<jsp:setProperty name="userInfo" property="*" />
</jsp:useBean>

In this example, the nested <jsp:setProperty> action sets all properties when the bean is created. If the bean already exists, the <jsp:useBean> action associates it with the name specified by the id attribute so it can be accessed by other actions and scripting code. In this case, the <jsp:setProperty> action is not executed.

All Together Now

At this point, you have seen the two mechanisms needed to let multiple pages process the same request:
passing control and passing data. These mechanisms allow you to employ the MVC design, using one page for request processing and business logic, and another for presentation. The <jsp:forward> action can be used to pass control between the pages, and information placed in the request scope is available to all pages processing the same request.

Page for Displaying Entry Form (userinfoinput.jsp)

```
<%@ page language="java" contentType="text/html" %>
<html><head>
<title>User Info Entry Form</title>
</head><body bgcolor="white">
```

```
<jsp:useBean
id="userInfo"
scope="request"
class="com.ora.jsp.beans.userinfo.UserInfoBean" />
<%-- Output list of values with invalid format, if any --%>
<font color="red">
<jsp:getProperty  name="userInfo"  property="propertyStatusMsg"
/>
</font>
<%-- Output form with submitted valid values --%>
<form action="userinfovalidate.jsp" method="post">
<table>
<tr>
<td>Name:</td>
<td><input type="text" name="userName"
value="<%= StringFormat.toHTMLString(userInfo.getUserName( ))
%>" >
</td>
</tr>
...
```

The rest of the example is the same as before.

The validation page, *userinfovalidate.jsp*

Input Validation Page (userinfovalidate.jsp)

```
<%@ page language="java" %>
<jsp:useBean
id="userInfo"
scope="request"
class="com.ora.jsp.beans.userinfo.UserInfoBean" >
<jsp:setProperty name="userInfo" property="*" />
</jsp:useBean>
<% if (userInfo.isValid( )) { %>
<jsp:forward page="userinfovalid.jsp" />
<% } else { %>
<jsp:forward page="userinfoinput.jsp" />
<% } %>
```

This is the request processing page, using the bean to perform the business logic. Note that there's no HTML at all in this page,

only a page directive specifying the scripting language, action elements, and scriptlets. This is typical of a request processing page: it doesn't produce a visible response message, it simply takes care of business and passes control to the appropriate presentation page.This example is relatively simple. We first create a new userInfo Bean named userInfo in the request scope and set its properties from the request parameters of the previous form. A scriptlet calls the bean's isValid( ) method to validate the properties and uses the <jsp:forward> action to pass control to the appropriate View page.

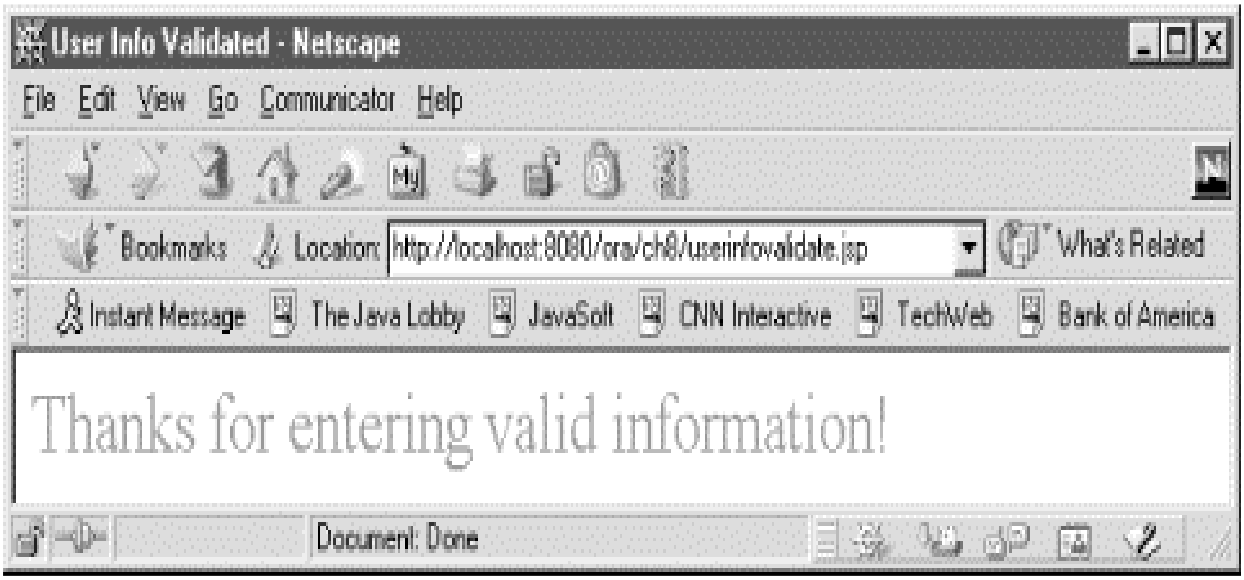
If the input is invalid, the *userinfoinput.jsp* page is used again. This time the <jsp:useBean> action finds the existing userInfo bean in the request scope, and its properties are used to show an error message and fill out the fields that were entered correctly, if any. If all input is valid, the control is passed to the *userinfovalid.jsp* page to present the "thank you" message.

Valid Input Message Page (userinfovalid.jsp)

```
<html>
<head>
<title>User Info Validated</title>
</head>
<body bgcolor="white">
<font color=green size=+3>
Thanks for entering valid information!
</font>
</body>
</html>
```

This page tells the user all input was correct. It consists only of template text, so this could have been a regular HTML file. Making it a JSP page allows you to add dynamic content later without changing the referring page, however. The results are shown below.

The valid input message page



Let's review how placing the bean in the request scope lets you access the same bean in all pages. The user first requests the *userinfoinput.jsp* page. A new instance of the userInfo bean is created in the request scope and used to generate the "enter all fields" status message. The user fills out the form and submits it as a new request to the *userinfovalidate.jsp* page. The previous bean is then out of scope, so this page creates a new userInfo bean in the request scope and sets all bean properties based on the form field values. If the input is invalid, the <jsp:forward> action passes the control back to the *userinfoinput.jsp* page. Note that we're still processing the same request that initially created the bean and set all the property values. Since the bean is saved in the request scope, the <jsp:useBean> action finds it and uses it to generate an appropriate error message and fill out the form with any valid values already entered.

Sharing Session and Application Data

HTTP is a stateless, request-response protocol. This means that the browser sends a request for a web resource, and the web server processes the request and returns a response. The server then forgets this transaction ever happened. So when the same browser sends a new request, the web server has no idea that this request is related to the previous one. This is fine if you're

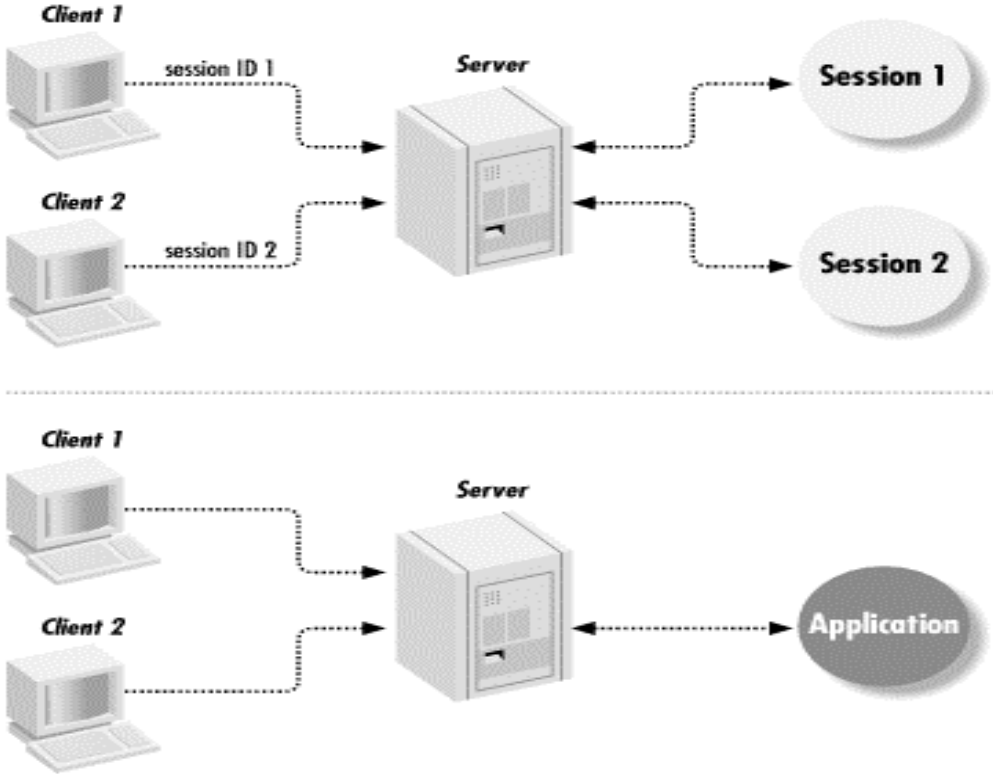
dealing with static files, but it's a problem in an interactive web application. In a travel agency application, for instance, it's important to remember the dates and destination entered to book the flight so the customer doesn't have to enter the same information again when it's time to make hotel and rental car reservations.

The way to solve this problem is to let the server send a piece of information to the browser that the browser then includes in all subsequent requests. This piece of information, called a *session ID*, is used by the server to recognize a set of requests from the same browser as related: in other words, as part of the same *session*. A session starts when the browser makes the first request for a JSP page in a particular application. The session can be ended explicitly by the application, or the JSP container can end it after a period of user inactivity. Thanks to the session ID, the server knows that all requests from the same browser are related. Information can therefore be saved on the server while processing one request and accessed later when another request is processed. The server uses the session ID to associate the requests with a *session object*, a temporary in memory storage area where servlets and JSP pages can store information.

The session ID can be transferred between the server and browser in a few different ways. The Servlet 2.2 API, which is the foundation for the JSP 1.1 specification, identifies three methods: using cookies, using encoded URLs, and using the session mechanism built into the Secure Socket Layer (SSL), the encryption technology used by HTTPS. SSL-based session tracking is currently not supported by any of the major servlet containers, but all of them support the cookie and URL rewriting techniques. JSP hides most of the details about how the session ID is transferred and how the session object is created and accessed, providing you with the session scope to handle session data at a convenient level of abstraction. Information saved in the session scope is available to all pages requested by the same browser during the lifetime of the session.

However, some information is needed by multiple pages independent of who the current user is. JSP supports access to this type of shared information through another scope, the application scope. Information saved in the application scope by one page can later be accessed by another page, even if the two pages were requested by different users. Examples of information

typically shared through the application scope are database connection pool objects, information about currently logged-in users, and cache objects to avoid frequent database lookups.



shows how the server provides access to the two scopes for different clients.

Counting Page Hits

A simple page counter bean can be used to illustrate how the scope affects the lifetime and reach of shared information. The difference between the two scopes becomes apparent when you place the bean in both the session and application scopes.

Page with Counter Beans (counter1.jsp)

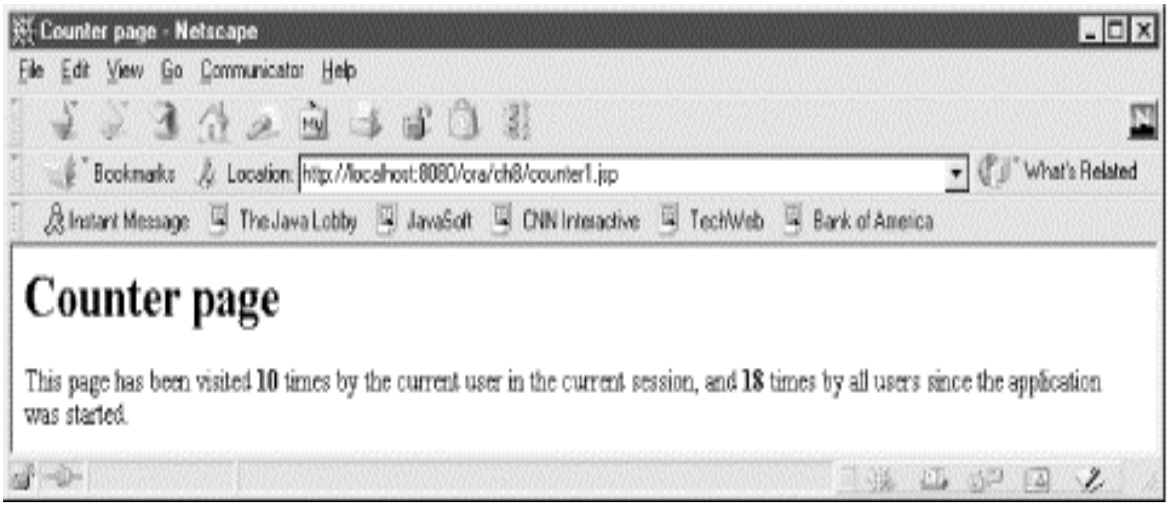
```
<%@ page language="java" contentType="text/html" %>
<html>
<head>
<title>Counter page</title>
</head>
<body bgcolor="white">
<jsp:useBean
id="sessionCounter"
scope="session"
class="com.ora.jsp.beans.counter.CounterBean"
/>
```

```
<jsp:useBean
id="applCounter"
scope="application"
class="com.ora.jsp.beans.counter.CounterBean"
/>
<% String uri = request.getRequestURI( ); %>
<h1>Counter page</h1>
This page has been visited <b>
<%= sessionCounter.getNextValue(uri) %>
</b> times by the current user in the current session, and <b>
<%= applCounter.getNextValue(uri) %>
</b> times by all users since the application was started.
</body>
</html>
```

The bean used in this example, the com.ora.jsp.beans.counter.CounterBean, keeps a separate counter for each page where it's used. It's a class with just one method: public int getNextValue(String uri); The method increments the counter for the page identified by the uri argument and returns the new value.two <jsp:useBean> actions are used to create one bean each for the session and application scopes. The bean placed in the session scope is found every time the same browser requests this page, and therefore counts hits per browser. The bean in the application scope, on the other hand, is shared by all users, so it counts the total number of hits for this page.

A scriptlet is used to ask the request object for the URI of the current page. The URI is then passed as an argument to the bean's getNextValue( ) method. A page is uniquely identified by its URI, so the bean uses the URI as a unique identifier to represent the counter it manages for each page.

A page with session and application page hit counters



As long as you use the same browser, the session and application counters stay in sync. But if you exit your browser and restart it, a new session is created when you access the first page. The session counter starts from 1 again but the application counter takes off from where it was at the end of the first session. Note that the bean described here keeps the counter values in memory only, so if you restart the server, both will start from 0 again.

Memory Usage Considerations

You should be aware that all objects you save in the application and session scopes take up memory in the server process. It's easy to calculate how much memory is used for application objects since you have full control over the number of objects you place there. But the total number of objects in the session scope depends on the number of concurrent sessions, so in addition to the size of each object, you also need to know how many concurrent users you have and how long a session lasts.

Let's look at an example. The CartBean used in this chapter is small. It stores only references to ProductBean instances, not copies of the beans. An object reference in Java is 8 bytes, so with three products in the cart we need 24 bytes. The java.util.Vector object used to hold the references adds some overhead, say 32 bytes. All in all, we need 56 bytes per shopping cart bean with three products. If this site has a modest number of customers, you may have 10 users shopping per hour. The default timeout for a session is 30 minutes, so let's say that at

any given moment, you have 10 active users and another 10 sessions that are not active but have not timed out yet. This gives a total of 20 sessions times 56 bytes per session, a total of 1,120 bytes. In other words, a bit more than 1 KB. That's nothing to worry about. Now let's say your site becomes extremely popular, with 2,000 customers per hour. Using the same method to calculate the number of concurrent sessions, you now have 4,000 sessions at 56 bytes, a total of roughly 220 KB - still nothing to worry about. However, if you store larger objects in each session, for instance the results of a database search, with an average of 10 KB per active session that corresponds to roughly 40 MB for 4,000 sessions. A lot more, but still not extreme, at least not for a site intended to handle this amount of Traffic. However, it should become apparent that with that many users, you have to be a bit more careful with how you use the session scope.

Here are some things you can do to keep the memory requirements under control:

- Place only those objects that really need to be unique for each session in the session scope.

- Set the timeout period for sessions to a lower value than the default.

- Provide a way to end the session explicitly. A good example is a logout function. Another possibility is to invalidate the session when something is completed. You can use the session.invalidate( ) method to invalidate a session and make all objects available for garbage collection