CSD Courseware

Web Technologies

Java Server Pages 1.2

Written & Complied by

F X Rohaan Pereira

Web Administrator

and

Syed Awase Khirni

Copyright © 2000 Centre for Software Development.

Table of Contents

S.No.	*Topic*	**Page**
1.	Introduction to JavaServer Pages	3
2.	The Comment Tag	12
3.	The Hidden Comment Tag	14
4.	The Declaration Tag	15
5.	The Expression Tag	16
6.	The Scriptlet Tag	17
7.	The JSP Forward Tag	19
8.	The Include Tag	20
9.	The Page Directive	21
10.	JavaBeans	22
11.	Putting JSP and JavaBeans Together	25
12.	JSP VS ASP Development	29
13.	JSP Implicit Objects	44
14.	JSP - Beans – Databases	54
15.	JSP – Interactions	59

Disclaimer : All material used in this Courseware is a compilation of various resources. This document is meant ONLY for use by staff & students of BITS,Pilani within its premises.

1.Introduction to Java Server Pages

Server-Side Scripting the Java Way

JavaServer Pages™ (JSP) is a web-scripting technology similar to Netscape server-side JavaScript (SSJS) or Microsoft Active Server Pages (ASP). However, it's more easily extensible than SSJS or ASP, and it isn't proprietary to any one vendor or any particular web server. Although the JSP specification has been managed by Sun Microsystems, any vendors can implement JSP in their own systems.

If you're working with Netscape technologies today or are planning to in the future, you should start learning about JSP, because Netscape will soon be introducing products that use it. They've already announced that the next version of Netscape Application Server (NAS) will use JSP as a presentation layer technology. In this article, I'll introduce you to JSP: its features, target users, and intended use. I'll compare JSP to some current Netscape technologies, such as SSJS and NAS's presentation markup language.

JSP, or Java Server Pages, is technology invented by Sun Microsystems which allows the easy creation and maintenance of server side HTML pages, which can be used as both a kind of Dynamic HTML and CGI replacement. Conceptually, JSP pages are similar to ASP pages, Cold Fusion Markup Pages, and Embeded perl scripts.

What makes JSP so attractive?

For one thing, JSP/Servlets are much higher performance than their CGI counterparts, due to the inprocess nature of the execution and memory sharing between scripts.
Since JSP uses Java as the scripting language, the powerful features of Java such as multi threading, structured exception handling and remote invocation make it the tool of choice for highly scalable, multi-tiered web sites, for eCommerce and other applications, where scalability, maintainability and portablility are important.

Additionally, JSP/Servlets allow the user to create their own utility, or business logic classes -- in Java --, and call them easily from within their JSP/HTML pages. An example of this might be an "HTML Generator" object, which the user can call to write his/her HTML pages for him. This Generator can operate like a Style sheet albeit much more powerful, and be portable to any browser since the action is done on the server side, making short work of normally tedious HTML code. From there the entire look and feel of a site can be easily switched by redefining one or two functions. Another example of a "worker class" could be encapsulating all of your SQL queries and logic into one (or more) classes that can be called from any web page. That way your French, Spanish, and Japanese web pages for online ordering all call the exact same routines for business logic... And if you switch databases you don't even have to edit the JSP files! Just change the methods in the Utility class and you are done.

Lastly, the use of "Worker Class" object as mentioned above, or the use of Java Beans within JSP pages, can make a complete separation of presentation (HTML) and Code (Java), so that the graphic designers and programmers do not interfere with each other while working on a dynamic web document.We will discuss this further in the next section.

What software do I need to run JSP pages?

You need what is known as a "Servlet Engine", which attaches to your web server. You also need a Java SDK, which you should install on your Server *before* installing the Servlet Engine. Note -- to run JSP pages, you *MUST* have the Java SDK, *NOT* just the JRE. JSP pages require the compiler to be present on the Server, therefore you MUST install the SDK. In contrast, Servlets do not require the compiler to be present on the server. Finally, the Servlet Engine must support JSP, preferably v 1.0 or higher. The most popular servlet engines right now seem to be JRUN and ServletExec. A number of development tools, such as the WebSphere Studio Page Designer, can be used to visually create a page containing dynamic contents based on the properties of Java beans

Some of the JSP Tools are listed below

JSP TOOLS

JSP Servlet Engines

Apache Jserv
Caucho
GNU JSP
Jakarta
Java Web Server (Sun)
Jrun (Live Software)
Orion Web Server
PolyJsp
ServletExec (New Atlanta)
SilverStream
SJSP
WebSphere Application Server (IBM)
WebLogic

JSP and Java Tools

Code Warrior
IBM WebSphere Studio (JSP)
IBM VisualAge for Java 2.0 (JSP)
Symantec Visual Café
Inprise JBuilder 3.5
Developer Tools Guide at JavaWorld
Oracle JDeveloper 3 (JSP)
Accessing COM objects from JSP using J-Integra pure Java-COM bridge

JSP and Servlet Hosting

The Adrenaline Group
Dan's Free Application Server
ebpcs.net
mycgiserver
Servlets.net
Tri Star Web
wantJAVA
WebAppCabaret

Other dynamic content models

Other models for deploying dynamic content abound for the Internet the strongest contender in times past being CGI. But there are others of note, CGI/Perl, PHP, Cold Fusion, JHTML, and of course, ASP.

Java Server Pages Vs Active Server Pages

JSP is the same basic concept as ASP - there are similarities:

- ASP uses ActiveX objects to provide external functionality and access to business objects. JSP uses JavaBeans.
- ASP focuses on MTS as its Transaction Server for making its solution scalable (but see http://www.esperanto.org.nz/espe-ranto.asp?inc=techtalk/MTSvsDCE.html). The focus of JSP is on EJB.
- ASP and JSP are essentially interpreted languages. ASP gets converted into pcode and JSP gets converted into Java Byte Code.
- ASP allows the creation of functions for seperating code, JSP uses Inner Classes or Declarations.

Differences:

- ASP uses Visual Basic Scripting or Java Script. JSP focused on Java (but allows others)
- ASP is a “product” (which Chillisoft has copied for other plat-forms), JSP is an almost specification. For many, it comes down to a question of personal preference. ASP has had the lead and many, many sites run ASP pages as it is a built in feature of Microsoft’s Internet Information Server. Many people further examine JSP for a number of reasons
- the Java language is more powerful than Visual Basic Script-ing/ JavaScript for server side work, Java is the chosen development language for server side development (and perhaps client side development)
- JSP allows the “scripting” language and the language for the development of the business logic (beans) to be the same but still interact with business logic embedded in other languages via CORBA (if necessary).
- CORBA and/or EJB play a large part in an organisation’s technology architecture
- An organisation focuses on Unix, AS400 or mainframe architectures and there is greater support on those platforms for the JSP model. ASP doesn’t have the support of ActiveX on non-Windows platforms.
- Greater choice in the vendors - there were 11 vendors who offered JSP implementations at the time of writing.

Conceptually, JSP pages are most similar to ASP pages, in that both JSP and ASP provide built in session tracking and management, and sophisticated database interactions. However, JSPs are built on Java, which is a much more advanced language than VB, and has better support for error trapping and multi threading. Also, JSP pages run on all platforms, not just IIS. Theoretically, ASP can run on Unix machines with certain software; practically speaking, most ASP sites depend on a plethora of ActiveX controls to provide the functionality and performance that VBA (the ASP scripting language) lacks. As an example, the ASP language is too weak to send mail by itself. It needs some kind of ActiveX control it can call to send mail for it. Since these ActiveX controls are usually not available on Unix, ASP is inherently non portable.

JSP vs CGI

CGI has one major disadvantage in relation to other environments avail-able
(including ASP and JSP) - performance and maintenance of state. 4
Both ASP and JSP (and many of the other products available for per-forming
similar functionality) hook directly into the web server(through DLLs or being dynamically linked in some other manner), and thus there is no need to go through the time consuming, resource intensive process of writing the request to a file or pipe, running a program and reading back the output. CGI web applications also tend to suffer from a lack of cohesiveness in their structure more than the other technologies as there is no focus with CGI technologies for tools that man-age a group of pages as an "application".

The other problem of CGI is that of state management. With the JSP/Servlet host running all of the time, the host can store state easily and make sure that it is provided to the JSP page in an easy, effective manner.With CGI applications, if libraries are not already written or other-wise available, they have to be written to manage all of the state management mechanisms.

JSP compared to Perl

Conceptually, JSP pages are most similar to Embeded perl pages, and Servlets are most similar to Perl as used in CGI programming. JSP is vastly superior to Perl, in that the Java language is much more advanced and rigorous, supports multi threading, and allows for sophisticated in memory data exchange among "CGI" Servlets and JSPs, which provides an incredible performance boost compared to Perl. JSP/Servles also provide a built in session management scheme, obviating the need for the complex and tortured code often necessary in Perl scripts to maintain state.

JSP compared to Servlets

A JSP page is ultimately compiled to a Servlet. Thus, a JSP page is a Servlet in a certain sense; however, JSPs can be more easily deployed, since they can live in any directory; also it is far easier to embed long HTML strings in a JSP than it is to output a bunch or out.println statements in a Servlet. Note that JSPs do not completely replace Servlets; often a few Servlets are still needed for application initialization and daemon support in the background; however, by and large, JSPs are geared to replace Servlets for all but the most specialized tasks.

Isn't Java too Slow?

Good question! But the answer is NO for several reasons. First of all most people's experience with Java's slowness has to do with running Applets or Java SWING applications; Java GUI programs are very slow. However, Java at its core is very fast, and, just as Unix Servers gain performance due to not having to deal with a Windowing system, so do Java Servlets/JSPs gain performance by not having to be burdened drawing graphics to the screen. It is really the Windowing system in Java that is slow, not Java itself, and this is is not present on the server side.
Beyond that, and where Java really BOOSTS performance, is that a JSP/Servlet is an in-process function call, which does not have the overhead of starting a new process for every HTTP request. Also Servlets and JSPs can easily communicate and store data in RAM, whereas other CGI implementations (Perl) typically must write state to an intermediate file or database in between CGI screens (eg, Shopping Cart) which *REALLY* slows things down.

Some Advantages of using JSP technology over other methods of dynamic content creation:

- **Separation of dynamic and static content**

This allows for the separation of application logic and Web page design,reducing the complexity of Web site development and making the site

easier to maintain.

- **Platform independence**

Because JSP technology is Java-based, it is platform independent. JSPs can run on any nearly any Web application server. JSPs can be developed on any platform and viewed by any browser because the output of a compiled JSP page is HTML.

- **Component reuse**

Using JavaBeans and Enterprise JavaBeans, JSPs leverage the inherent reusability offered by these technologies. This enables developers to share components with other developers or their client community, which can speed up Web site development.

- **Scripting and tags**

JSPs support both embedded JavaScript and tags. JavaScript is typically used to add page-level functionality to the JSP. Tags provide an easy way to embed and modify JavaBean properties and to specify other directives and actions.In short, JSP/Servlets can execute from 10s to 100s of times faster than their counterparts.

The future of JSP

The future of JSP is not entirely clearly mapped out. As of the time of writing, we are working with the JSP 1.0 Public Draft. This has introduced a number of things (particularly XML syntax and buffered output to help with redirection) and removed other items (particularly exten-siontags for HTML which has a number of developers wanting to stick with the 0.92 specification). There is an indication in the Public Draft that XML will become required in the 1.1 specification, and that JSPwill be tied more heavily into the Enterprise Java architecture (J2EE -Java2 Enterprise Edition) that Sun is working on. At present, there is no integration with any of the Enterprise features

JSP- The Technology

Java Server(TM) Pages is a simple, yet powerful technology for creating and maintaining dynamic-content web pages. Based on the Java programming language, Java Server Pages offers proven portability, open standards, and a mature re-usable component model.
The Java Server Pages architecture enables the separation of content generation from content presentation. This separation not only eases maintenance headaches, it also allows web team members to focus on their areas of expertise. Now, web page designers can concentrate on layout, and web application designers on programming, with minimal concern about impacting each other's work.
The rest of this document gives you the bigger picture:

- Portability
- Composition
- Processing
- Access Models

Portability:

JavaServer Pages files can be run on any web server or web-enabled application server that provides support for them. Dubbed the *JSP engine*, this support involves recognition, translation, and management of the JavaServer Page lifecycle and its interactions with associated components.
The JSP engine for a particular server might be built-in or might be provided through a 3rd-party add-on. As long as the server on which you plan to execute the JavaServer Pages supports the same specification level as that to which the file was written, no changes should be necessary as you move your files from server to server. Note, however, that instructions for the setup and configuration of the files may differ between files.
To date, there has been no upwards- or backwards-compatibility between JavaServer Pages specifications. A JavaServer Pages file written to the 0.92 specification can be run *only* on a server supporting JavaServer Pages 0.92. The same file could not run on a server supporting only JavaServer Pages 1.0 or JavaServer Pages 0.91.

Composition:

It was mentioned earlier that the JavaServer Pages architecture can include reusable Java components. The architecture also allows for the embedding of a scripting language directly into the JavaServer Pages file.

The components current supported include JavaBeans, and Servlets. Support for Enterprise Java Beans components will likely be added in a future release. As the default scripting language, JavaServer Pages use the Java programming language . This means that scripting on the server side can take advantage of the full set of capabilities that the Java programming language offers. Support for other scripting languages might become available in the future.

Processing:

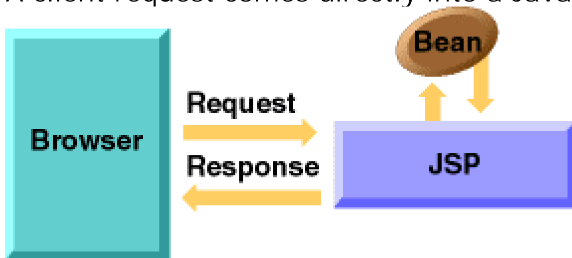
A JavaServer Pages file is essentially an HTML document with JSP scripting or tags. It may have associated components in the form of .class, .jar, or .ser files--or it may not. The use of components is not required.

The JavaServer Pages file has a *.jsp* extension to identify it to the server as a JavaServer Pages file. Before the page is served, the JavaServer Pages syntax is parsed and processed into a servlet on the server side. The servlet that is generated outputs real content in straight HTML for responding to the client. Because it is standard HTML, the dynamically generated response looks no different to the client browser than a static response.

Access Models:

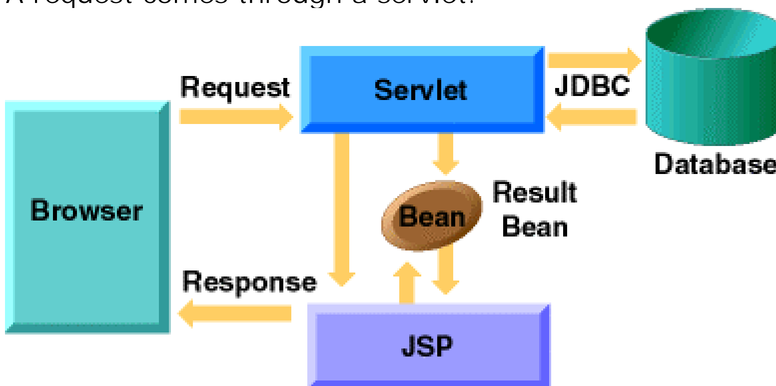
A JavaServer Pages file may be accessed in at least two different ways:

1. A client request comes directly into a JavaServer Page.



   In this scenario, suppose the page accesses reusable JavaBean components that perform particular well-defined computations like accessing a database. The result of the Bean's computations, called *result sets* are stored wihtin the Bean as properties. The page uses such Beans to generate dynamic content and present it back to the client.

2. A request comes through a servlet.



   The servlet generates the dynamic content. To handle the response to the client, the servlet creates a Bean and stores the dynamic content (sometimes called the *result set*) in the Bean. The servlet then invokes a JavaServer Page that will present the content along with the Bean containing the generated from the servlet.

   There are two APIs to support this model of request processing using JavaServer Pages. One API facilitates passing context between the invoking servlet and the JavaServer Page. The other API lets the invoking servlet specify which JavaServer Page to use.

In both of the above cases, the page could also contain any valid Java code. The JavaServer Pages architecture encourages separation of content from presentation--it does not mandate it.

How to Choose Between Access Models

With at least two access models, the question naturally arises "When does it make sense to have a JavaServer Page as the front-end to a servlet, as the back-end to a servlet, or use only the servlet? Here are some possible guidelines:

- If a graphical interface (GUI) is necessary to collect the request data-use a JavaServer Pages file.
- If the request and request parameters are otherwise available to the servlet, but the results of the servlet processing requires a graphical interface to present them use a JavaServer Pages file.
- If presentation layout is minimal (will not require very many println lines in your servlet code) and you don't need to make that presentation logic available to a customer or your webpage designer, then a Servlet might suffice.

What Is JSP?

JavaServer Pages are all about data-- how to move it, handle it and display it. JSP lets the programmer build dynamic web sites. For example, when you log on to your favorite on-line book store, the web site may greet you by name and suggest new titles of specific interest to you based on your past purchases. The web site content changes dynamically, in this example, based on who the user is. With JSP, web site design is not only a function of how the data is presented but also what data is used (and how it is used) based on current conditions.

Why Use JSP Instead of Pure Java?

JSP facilitates a division of labor between programmers and page designers.

As this tutorial progresses, you'll see that JSP looks a lot like HTML with embedded Java. The HTML paints the page, and the Java gets the work done, but that means the page designers and the programmers are constantly working on the same JSP script. It seems like the result would be chaos, but ...

If you let the page designers translate their vision into the JSP's HTML and let the programmers write their part in Java Beans that are executed from the JSP part of the script, then nobody steps on anybody. It's a powerful division of labor, allowing each group to concentrate on doing what it does best. Another advantage of this approach is that Java Beans are reusable, so it just makes good sense to separate them out into database calls, business rules, and other common routines.

So Learning JSP also Means Learning HTML and Java Beans

Don't panic! The HTML is pretty simple, and you'll see how to write it as we go along. Tons of reference books are available on the subject, but the best learning tool I've found is a two-page, plastic reference guide from Bar Charts , Inc. (They also make one for Java 1.2.)

Java Beans aren't all that big a deal either. If you can write an applet or a Java application, you can write a Bean. It's just got a bit more structure. We're not going to get there for quite a while, but Bean basics will be covered when we do.

How Does the JSP Process Work?

First we need to define a few terms that will keep popping up.

- A ***Web Server*** is software that accepts a request for a web file (HTML). When it gets that file, it sends the information back to the client machine's browser for display. I'm using the Apache web server.
- A ***Container*** is software that stores JSP files and servlets, converts JSP files into servlets, compiles servlets, and runs servlets, creating HTML. All this stuff happens when the web server gets a request for a file with a .jsp extension. I'm using the Tomcat container, but you should know that many alternatives exist. Some are free and some aren't. By the way, if you hear terms like JSP engine, don't worry about them. It's all part of the container.
- We'll also be talking about some software to accommodate connection pooling, but that can wait until a subsequent lesson when we're going after real data.

The user does something to cause a request for a JSP file to be sent to the web browser. A real-world example is a HTML form that allows the user to enter something and then take an action (like clicking a button) that fires off a request to the web server for a JSP file to be loaded into the web browser and displayed. A more simple example is to ask the browser for a URL. For instance: localhost/test/ray/hello1.jsp

The web server receives the request for a .jsp file, but it has no idea what to do with the .jsp extension. Remember, this web server isn't all that bright. It serves up HTML, and nothing else, so it forwards the request to the container.

The container looks for a servlet class with the requested file name in the appropriate package. If it finds the servlet, it runs the servlet, creating HTML that is immediately sent to the web browser (so, servlets create HTML). Important point: a JavaServer Page is converted to a Java servlet for compiling and processing. For now, you don't need to know anything at all about servlets, how to compile them, or how to run them to use JSP. All that stuff is done for you. Later, however, we'll have to get into the basics.

If the container doesn't find the servlet class, it looks for the same file name, but with a .jsp extension. If it finds a match, the JSP is converted to a servlet by the container's JSP engine, and the servlet is compiled. Then the container runs the servlet, creating HTML that is immediately sent to the web browser, just like before.
The web server gets the newly-created HTML from the container and forwards it to the client's browser for processing. The browser does its thing and the user sees the data he/she requested in the beginning.

Getting Set to Run JSP
You absolutely must establish a JSP environment (i.e., set up your own web server and container), so here are two ways to do it.

1. You may download the entire platform from Sun at http://developer.java.sun.com/developer . You'll have to register first, but it's the most non-invasive and thoroughly painless process I've encountered on the net. When you get there, you'll find two options. You may download the entire J2EE (30+ MB) or the JavaServer Web Development Kit (750KB). Your choice, but I urge you to settle for the Development Kit. It's quicker to download and the configuration issues are minimized. Either way, Sun gives you abundant help with installation and configuration.
2. Another way is to pick and choose your web server and container. A myriad of choices exist. Based on a recommendation, I downloaded:
   - Apache web server and its documentation from www.apache.org. Apache provides plenty of information on installation and configuration, so don't be intimidated.
   - Tomcat container from http://jakarta.apache.org. Since Tomcat is developed by the Apache folks, you may also get it from www.apache.org, just like the web server. As with the Apache web server, Tomcat's installation and configuration are well explained. If you are having a problem with setting up Tomcat, get help by checking out the Tomcat developer mailing list.

An Example: Hello1.jsp
Assuming you have installed and configured your environment, let's write some code to prove that everything works. Our first program will be short and sweet, but the fact it's saved in the container with a .jsp extension will prove that the process is alive and well.
As I mentioned above, a JSP script looks like HTML with Java embedded in it. It's more than that, but for now let's just look at it that way. As yet, however, we have no JSP code. So let's write some very basic HTML, and give it a .jsp extension. That way, the web server will know to send our URL request along to the container for processing. It's interesting to note that you don't need any JSP code, just the .jsp extension. Our example will be called Hello1.jsp.
Remember, **before you actually run the JSP file**, you have to:

- Write the JSP using an editor that doesn't put extra information on the page. I suggest using HomeSite from Allaire (you may download a thirty day free trial copy), but Notepad will work just fine.
- Save your JSP script in the container. (I used localhost\test\ray\Hello1.jsp)
- Start up your web server.
- Start up your container.
- Start up your browser.

Now, **write the JSP** source code.

```
1. <html>
2. <head>
3. <title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
4. </head>
5. <body>
6. Hello World
7. </body>
8. </html>
```

Line 3 puts a title on the top line of the browser. I chose to put the fully-qualified path there. It's my way of keeping track of what I' m looking at, but it probably shouldn't go into production.
Line 6 gets the job done. Sticking text into HTML will cause it to display on the browser window, so "Hello World" appears. Subsequent examples will build on this theme.

Save the Hello1.jsp file.
I used localhost\test\ray\Hello1.JSP, but you will have to determine your own path.

Run the Hello1.jsp file.
Enter the path you used to save the JSP on the URL line of your browser, and press the enter key. Again, I used localhost\test\ray\Hello1.jsp, but you will use your own path.

The **HTML output** (created by running the servlet) is:

```
<html>
<head>
<title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
</head>
<body>
Hello World
</body>
</html>
```

2.Introducing HTML and JSP Tags

Introduction
HTML is the language of the Internet. You give a browser a HTML file and it's going to do whatever that file tells it to do. Why? Because HTML is a language and, as such, it has structure, rules and syntax. In this case, the various parts of a page are defined by HTML tags.
HTML tags are devices used to identify various presentation elements in a HTML document. Let's take a look at a simple HTML file. So what do you think might be between the <title> and </title> tags? You have it: the page's title. Now you're about to become a HTML genius.

An Example of HTML
Above you learned that HTML instructions are identified by HTML tags, and in Lesson 1 you learned that JSP is just HTML with Java intermixed. So, how does the container software identify the Java code? Easy, by the JSP tags. But before we explore those, let's look at a really bare-bones HTML file. No surprises here, because it's very similar to the example in Lesson 1.

```
1. <html>
2. <head>
3. <title>This is the hidden title for your page.</title>
4. </head>
5. <body>
6. Hello World
7. </body>
8. </html>
```

Lines 1 & 8 identify the beginning and end of the HTML code. They also show us the standard format for beginning and ending tags. The difference is that the ending tag always has a "/" preceeding the tag text, as in </HTML>. Also, note that HTML is not case sensitive.

Lines 2 & 4 are actually tags within tags. The <head> tag pair allows you to enter a title and several meta tags which describe your document but remain unseen on the actual page.
Line 3, title, names your page. Note that the beginning and ending tags may appear on the same line. Note also that your title (or any other text) does not need to be enclosed in quotation marks. Another example of this appears on line 6.

Lines 5 & 7 demonstrate the beginning and end of the body of the page.

Line 6 displays the text 'Hello World' when the browser runs this file.

A good way to get acquainted with HTML is to look at the code that's behind your web page. To do this, click on your browser's View menu and then click on something like View Source, Page Source, or similar menu option. This will open another window which will almost always display the HTML.
There it is, the entire HTML short course. It's enough to get you going and you may be certain that subsequent lessons in *JSP: The Short Course* will continue to expand your HTML knowledge a bit at a time. If you want a reference, I recommend *HTML 4 for Dummies* by Ed Tittel, Natanya Pitts and Chelsea Valentine. It's an easy read and a good resource.

What Are JSP Tags?
Just as HTML tags are devices used to identify various presentation elements in HTML and JSP programs, JSP tags are devices used to identify various Java processing elements in a JSP program.
We'll cover most of the JSP tags before this short course ends but the following are the most commonly used tags. This list, by the way, is in the order they're presented during this tutorial.

1. Comment tag. *(Basic comments and examples.)*
2. Hidden comment tag. *(Why they exist and how to use them.)*
3. Declaration tag. *(Declaring page (static) variables.)*
4. Expression tag. *(JSP talk for System.out.println( ).)*
5. Scriptlet tag. *(Where the Java goes.)*
6. Forward tag. *(Transfers control to another program, file, etc.)*
7. Include tag. *(Lets you use common routines.)*
8. Page directive tag. *(Where the properties for a page are set.)*

3.The Comment Tag

Your Code Deserves Comment
Some programmers will tell you that well-written code is self documenting, but I just don't believe it. I guess it's all those years writing COBOL and getting calls in the middle of the night to fix whatever abend/crash that came along. If it weren't for the comments, the job would have been impossible. And if you think it's hard figuring out what someone else meant by some cute (but obscure) hunk of code, think of how embarrassing it is to look at your own code and wonder what on earth you were thinking.

So, comments are important, even critical, and JSP gives you two options. In this lesson, we explore the more basic of the them, the output comment which includes your comment as part of the generated HTML. Sometimes, however, your comments may be proprietary and you don't want them displayed. In this case, you may use the JSP hidden comment tag which is covered in the next lesson.

The Output Comment
Since JSP resides on an HTML page you may create comments using HTML. These comments, as in the two examples below, may include only HTML or be combined with Java to create something even more interesting.

1. `<!-- your comment -->`
2. `<!-- your comment and/or [<%= a Java expression %>] --> (Brackets are optional.)`

Please note that any text you put between the <!-- & --> comment tags will appear in the generated HTML.

<!-- your comment -->
In JSP, a comment may be generated in plain HTML and contain as many lines as you need. For instance:

```
<html>
<!--JSP name: Hello.jsp Written by Jane Programmer-->
<head> <title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
</head>
<body>
Hello World
</body>
</html>
```

When you run the JSP file, the comment portion of the generated HTML will look like this:

```
<!--JSP name: Hello.jsp Written by Jane Programmer-->
<!-- your comment and/or [<%= a Java expression %>] -->
```

JSP comments may also be a combination of HTML and Java, allowing dynamic information to be included in the comment. In the example below, the strange-looking Java code within the brackets is called a JSP expression (in a later lesson JSP expressions are covered in detail). Here's an example of a comment which combines HTML and a JSP expression:

```
<html>
<!--JSP name: Hello.jsp
Written by Jane Programmer. This page was executed on <%=(new java.util.Date( ))%>-->
<head>
<title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
</head>
<body>
Hello World
</body>
</html>
```

When you run the JSP, the comment in the generated HTML will look like this:

```
<!--JSP name: Hello.jsp Written by Jane Programmer. This page was executed on Wed Feb
2 10:14:19 PDT 2000-->
```

The Expression is Dynamic

Because the expression is dynamic, it is derived every time the JSP-generated HTML is run. So, in the example above, you will always get the current date and time. (If it were static, you would always get the same information, the date and time when the JSP was last converted to HTML.) For example, if I ran it again, I would get something like this:

```
<!--JSP name: Hello.jsp Written by Jane Programmer. This page was executed on Wed Feb
2 11:24:32 PDT 2000-->
```

Notice the time index changed by more than one hour.

4.The Hidden Comment Tag

What Is a Hidden Comment Tag?

A hidden comment tag is a comment which may be placed and viewed in the JSP code but is not included in the HTML returned from the container to the server. This essentially hides the comment from the view of a browser. Because they are ignored by the entire JSP process, hidden comments can't use JSP expressions. This is in contrast to output comments, discussed in Lesson 3, which do use JSP expressions to dynamically change generated HTML comments. Hence, hidden comments are not dynamic.

Hidden Comments are structured as follows:

```
<%-- your hidden comment--%>
```

Why Use a Hidden Comment?

Hidden comments are useful in many contexts. As mentioned in Lesson 3, hidden comments help protect proprietary information. Remember, if you write it, someone may steal it. A hidden comment allows you to include detailed information in the JSP source code that you might prefer to hide from the rest of the world. Hidden comments also help to document your code. While you may use output comments for this purpose, hidden comments may be more approrpiate in some situations. For example, hidden comments may be used to document blocks of logic, documentation that would otherwise only clutter up the generated HTML. Hidden comments may also be useful to you when you are in the middle of writing a program. They may be used as reminders to look up something or correct a problem. They are also especially handy for incremental testing.

An Example

In this example, a hidden comment is included in the JSP source code. Then the generated HTML returned from the container to the server shows the hidden comment is just that, hidden.

Here's the JSP source code:

```
<html>
<!-JSP name: Hello.JSP
Written by Jane Programmer on <%=(new java.util.Date( ))%>-->
<head>
<%-- Remember to replace the path from the title, and replace it with the JSP name--
%>
<title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
<body>
Hello World
</body>
</html>
After the JSP runs, the HTML will look like this:
<html>
<!-JSP name: Hello.JSP
Written by Jane Programmer on Tue Jul 18 10:56:20 PDT 2000-->
<head>
<title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
</head>
<body>
Hello World
</body>
</html>
```

5.The Declaration Tag

<%! declarations %>
A declaration tag declares a variable or method which may be used by any Java code being executed in the container for the current page. The scope declared with these tags is called *page scope*, meaning the variables or methods in a declaration tag may be used anywhere in a straightforward JSP page. This issue of scope may be quite tricky, however, such as when additional files are added when using an Include Directive. Scope issues are covered in a bit more detail later in this tutorial, and a great deal more information is available in the JSP Insider's reference material on JSP scripting (see Declarations). For now, the basic declaration tag will suit our purposes. The general form of this tag is shown in the section title above.

If declaration tags are tricky, why use them? While I prefer to declare variables and methods on an as-needed basis, and to limit their scope as narrowly as possible, sometimes a wider scope is required. When it is, you may rely on the declaration tag.

The All-Important Rules
A few simple rules must be followed for all declaration tags, including:

- Variables and methods are declared using standard Java syntax.
- Each declaration must be followed by a semicolon.
- Each variable or method must be declared before it's referenced.

Examples of the Declaration Tag
As may be imagined, declaration tags could refer to a wide variety of variables and methods. For the purposes of these examples, the radius of a circle as well as the height and width of an object are declared.

```
<%! int radius = 0 ; %>
<%! int height = 4, width = 7 ; %>
```

6.The Expression Tag

<%= Java expression %>
An expression tag is a short cut method to put data straight into the output buffer. Its format is shown above in the section title. Note, an expression tag never uses a semicolon.

The example below, from Lesson 3, demonstrates how an expression tag used in a comment could make the comment dynamic. Expression tags, may be used anywhere on JSP page. They are evaluated just like any other Java expression and the result of the evaluation is converted to a string. Then the string is inserted into the generated HTML.

Example 1
This example shows how a programmer could create a dynamic comment using an expression tag to insert the date and time when the page was executed. The JSP looks like this:

```
<html>
<!--JSP name: Hello.jsp
Written by Jane Programmer. This page was executed on <%=(new java.util.Date( ))%>-->
<head>
<title>c:\tomcat\webapps\test\ray\Hello1.jsp</title>
</head>
<body>
Hello World
</body>
</html>
Then the comment portion of the generated HTML looks like this
<!--JSP name: Hello.jsp Written by Jane Programmer. This page was executed on Wed Feb
2 10:14:19 PDT 2000-->
```

Example 2
Here's an example that also uses JSP declarations.

```
<html>
<head>
</head>
<body>
<%! int height = 4, width = 7 ; %>
The area of the rectangle is<%= height * width %>
</body>
</html>
```

The browser displays, 'The area of the rectangle is 28'.

The All-Important Rules
Two important points should be remembered when using JSP expression tags.

- No semicolons!
- Conform to the Java rules.

We've just learned one way to add dynamic content to our generated HTML.

7.The Scriptlet Tag

<% scriptlet %>
So far, we've covered declarations and expressions, each of which is limited to a very narrow range of activities. The scriptlet tag, however, is where the action is - the spot where the vast majority of your Java will be placed.

The All-Important Rules
The rules for this tag are a little more interesting.

- Semicolons are required to end statements!
- Normal Java language rules apply.
- A block statement may begin in one scriptlet and end in another scriptlet. This requires the first scriptlet to contain the opening brace of the block statement. Then the closing brace of the block statement must be included in one of the subsequent scriptlets. If the closing brace is missing, your JSP won't compile. This may sound a bit strange, but you'll find this ability to be really handy. We won't cover it in this lesson, but we'll use it a bit later in the course.

An Example
This example uses a scriptlet to generate Big Daddy's table. The example uses all the JSP tags presented so far, plus some Java comments.

```
1. <html>
2. <!--Big Daddy's JSP comment appears in the html. -->
3. <!--Big Daddy's JSP comment with an expression. Date = <%=(new java.util.Date(
   ))%>-->
4. <head>
5. <title>This is Big Daddy's JSP</title>
6. </head>
7. <%--DECLATRATIONS FOLLOW. Hidden comment does NOT appear in the html.--%>
8. <%!
9. // This java comment does NOT appear in the html.
10.       /* Neither does this one.*/
11.       String myString = "Hi Ho, Big Daddy" ;
12.       String lsTable = "" ;
13.       String lsInt1 = "" ;
14.       String lsInt2 = "" ;
15.       int i ;
16.       %>
17.       <body>
18.       <%--SCRIPTLET FOLLOWS. Hidden comment does NOT appear in the html.--%>
19.       <%
20.       lsTable = "<table border=1 >" ;
21.       lsTable += "<caption><i><b>This is the Big Daddy
   Table</b></i></caption><tr> <th>Even Numbers</th><th>Odd Numbers</th></tr>" ;
22.       for (i=0; i < 5; i+=2)
23.       {
24.       lsInt1 = String.valueOf( i );
25.       lsInt2 = String.valueOf( i + 1 );
26.       lsTable += "<TR><TD>" + myString + "-" + lsInt1 + "</TD><TD>" + myString
   + "-" + lsInt2 + "</tD></TR>" ;
27.       }
28.       lsTable += "</table>" ;
29.       %>
30.       <%--EXPRESSION FOLLOWS. Hidden comment does NOT appear in the html.--%>
31.       <%= lsTable %>
32.       </body>
33.       </html>
```

Line 2 is a standard output comment. It appears in the generated HTML.
Line 3 is a standard output comment with an embedded expression. When this JSP is run the comment appears in the generated HTML with the result of the expression.

Line 7 is a hidden comment. It does not appear in the generated HTML.
Line 8 begins the JSP declarations.
Line 9 is a Java comment using the // delimiter, so it does not appear in the generated HTML.
Line 10 is a Java comment using the /* - */ delimiters, so it does not appear in the generated HTML.
Lines 11 to 15 are standard Java variable declarations. I put them within declaration tags just to prove that this stuff really works. It does.
Line 16 ends the JSP declarations.
Line 18 is a hidden comment. It does not appear in the generated HTML.
Line 19 begins a scriptlet and line 29 ends it.
Line 30 is a hidden comment. It does not appear in the generated HTML.
Line 31 is a JSP expression. It displays the table we developed in the scriptlet in the generated HTML.

The Generated HTML

```
<html>
<!--Big Daddy's JSP comment appears in the html. -->
<!--Big Daddy's JSP comment with an expression. Date = Fri Jul 21 16:25:50 PDT 2000--
>
<head>
<title>This is Big Daddy's JSP</title>
</head>
<body>
<table border=1 ><caption><i><b>This is the Big Daddy Table</b> </i></caption><tr>
<th>Even Numbers</th><th>Odd Numbers< /th></tr><TR><TD>Hi Ho, Big Daddy-0</TD><TD>Hi
Ho, Big Daddy-1 </tD></TR><TR><TD>Hi Ho, Big Daddy-2</TD><TD>Hi Ho, Big Daddy-
3</tD></TR><TR><TD>Hi Ho, Big Daddy-4</TD><TD> Hi Ho, Big Daddy-5</tD></TR></table>
</body>
</html><<BR>
```

The Output

This is the Big Daddy Table

Even Numbers	*Odd Numbers*
Hi Ho, Big Daddy-0	Hi Ho, Big Daddy-1
Hi Ho, Big Daddy-2	Hi Ho, Big Daddy-3
Hi Ho, Big Daddy-4	Hi Ho, Big Daddy-5

8.The JSP Forward Tag

<jsp:forward page="relativeURL">
The purpose of the forward tag is to allow you to transfer control to a different location on the local server. The target may be another JSP page, a servlet, a CGI script, or even a static document. Simply insert the forward tag (with a relative URL) at the point in the code where you want the file added. That's it! Control is passed to the new program, or whatever, and the old one is forgotten. The forward tag's format is provided in the section title above.

Really Interesting Stuff To Know
When a forward tag is executed all processing stops in the current JSP page and the contents of the buffer are lost.

When the JSP container gets the request for a transfer to a new JSP page (as opposed to the other types of targets), it creates a new pageContext object, but the Request object and the Session object remain the same.

You may pass information (parameters) to the new page by using the <jsp:parm name="passingArg_1" and a value statement/>(note, this is a sub-tag in that it is always used in conjunction with another tag). The new program may access this information by using the getParameter ( ), getParameterValues ( ), and getParameterNames ( ) methods.

One excellent use for the jsp:forward tag is to handle exception processing. This is a good thing, because it ensures continuity.

Two Examples of the JSP Forward Tag
The first example simply passes control to a page called jspTest1.jsp. The second example also passes control plus information through the parameter sub-tag.

```
<!-- Pass control to jspTest1.jsp-->
<html>
<body>
<jsp:forward page="jspTest1.JSP" >
</body>
</html>
<!-- Pass control to jspTest1.JSP, passing arguments-->
<jsp:forward page="jspTest1.JSP" >
<jsp:parm name="passingArg_1" value="ABC" />
<jsp:parm name="passingArg_2" value= "XYZ" />
</jsp:forward>
</body>
</html>
```

9.The Include Tag

<%@ include file="relative.url"%>
The purpose of the include tag (format shown above) is to allow you to incorporate a file (JSP, HTML or text) into your JSP page. This facilitates code reuse because it lets you code and test your utilities. Then once they are bug-free, you may use and reuse the utilities in your own code or make them available to others for their use. Simply insert the include tag at the point in the code where you want the file added.

Note that the included *expression* is dynamic, meaning that your JSP will always display its current result.

Also note, this is the directive (or static) form of include. This means the file is included before the JSP page is compiled into a servlet. Effectively, the directive include causes the target file to be incorporated into the primary file before compiling. The action (or dynamic) form of include, where the file is included while the servlet is being processed, is not covered in this tutorial. For more information on directive (static) and action (dynamic) includes see JSP Insider's reference section (Including Files).

An Example of the JSP Include
In this example, two very simple utitlies, jspUtilTest2.jsp and jspUtilTest3.html, are called by the primary JSP file, jspTest1.jsp, using the include tag. Notice that the output of this example is dynamic - in this case, the date and time followed by some text.
This is the primary file, jspTest1.jsp.

```
<!-- jspTest1.jsp - accesses jspUtilTest2.jsp to display a date and jspUtilTest3.html
to add some text. -->
<html>
<body>
<%@ include file="jspUtilTest2.jsp" %> <%@ include file="jspUtilTest3.html" %>
</body>
</html>
```

The first utility is a dynamic JSP file which displays the date and time when the utility was called and generated the HTML.

```
<!-- jspUtilTest2.jsp is accessed by jspTest1, displaying the date and time. -->
<%= new java.util.Date() %>
```

The next utility adds simple HTML text.

```
<!-- jspUtilTest3.html - provides some html to jspTest1.jsp. -->
```
is a day to remember!

As a result the browser will display, "Tue Jul 18 10:56:20 PDT 2000 is a day to remember!"

10.The Page Directive

<%@ page att="val" %>
The page directive is simply a mechanism to tell the servlet engine about the page's properties. It controls what scripting language you'll be using, the size of the output buffer and so on. While this is important for you to know, you need to be cautious. I'll tell you why as we go through the various things you may do with it. As many page directives as you want are allowed, but only one for each attribute (except for import). You may place the directives anywhere, but I suggest that you put them at the top of your page unless some or all of them are set dynamically based on some condition(s). As an example, <%@ page buffer="1024k autoFlush="false" %> lets your JSP page store a one megabyte file before flushing the buffer to the client.

Syntax - One Argument at a Time : Here's a list of the page directive and most of its arguments.
<%@ page att="val" %>,
This format is the basis of the page directive statement and all of the following arguments are simply replacements for the att="val".

import="package.class"
The import argument allows you to specify what packages should be imported. For example, <%@ page import="java.util.*" %> lets you use the simple names from the Java package in your JSP program. Remember, this is the only page directive that may appear multiple times. Several packages, however, may be imported in one statement (i.e. import=" package1,package2,package3").

contentType="MIME-Type" and **charset="Character-Set"**
These arguments determine what MIME type and character set the JSP will use for output to the client's browser. For example, <%@ page contentType="text/plain" %> causes the browser to handle the output file as a text file, not HTML. The contentType default is text/html and the charset default is ISO-8859-1.

isThreadSafe="*true*|false"
The default value of true permits normal servlet processing, meaning the page may be run by several clients at a time. False, however, eliminates that flexibility by allowing only one request to run at a time. Be careful. Setting this to false may seriously impact your clients.

session="*true*|false"
This has a default value of true, and I suggest you leave it at that. The Session argument is a servlet issue, and that's well beyond the scope of this tutorial.

buffer="sizekb|none"
This argument specifies the output buffer size, the default being 8kb . A word of caution: entering "none" may cause a runtime exception if the JSP forward tag is used and the buffer isn't empty.

autoFlush="*true*|false"
The default value of true causes the buffer to be flushed when it is full. If the false value is used you cannot set buffer="none".

extends="package.class"
The extends argument provides the name of the superclass to be extended by the Java class in the JSP. Sun (and everybody else) recommends caution when using this argument because the JSP engine may be using a different superclass.

info="message"
This argument includes a string in the servlet. The string may then be retrieved via the getServletInfo method.

errorPage="url"
This is really great. The argument selects a URL page to display when an error was thrown but not caught. I guess it's there to protect us from ourselves, but I've never seen it used.

isErrorPage="true|*false*"
Refer to the errorPage="url" argument. This is its complement. The default value is false.

language="java"
Java is the default script language.

11.JavaBeans

What Is a JavaBean?
A JavaBean is a reusable software component that may be manipulated visually in a builder tool (the official definition). Let's break it down.

- A **software component** is encapsulated code that performs specific tasks. In plain talk, a JavaBean is a Java class with some extra rules. It may be either visual (to facilitate presentation) or non-visual (to facilitate application logic). This short course will focus on the JavaBean's usefulness in accessing data and enforcing business rules from a JSP script.
- A **reusable software component** can perform its magic for any application in the enterprise. If it exists, use it!
- A **builder tool** is a graphical environment that provides a host of programming aids to assist the programmer. It's usually called an 'Integrated Development Environment' (IDE for short). Examples are the visual components of Visual Basic, Access, Powerbuilder, JBuilder, VisualAge, etc. As you will soon see, a JavaBean is compiled to a class object. If it's written with the JavaBean pattern in mind any bean-loving API will know all about its properties. For this reason, we'll be following the design pattern and not simply instantiating the bean like any other Java program object.

I just mentioned that a JavaBean is a program. Great. A program is a class, right? So what do we do with a class? We instantiate it, creating an object that has all of the stuff that is in the class. In the case of a JavaBean that means we have properties (data), methods (functionality) and events. How we give it data to work with (setPropertyname( )), get data from it (getPropertyname( )), and use its exposed methods to get things done will all be explained in the remainder of this lesson.

Why Use JavaBeans with JSP
I suppose the better question might be: If all the functionality may be included in the JSP script, why on earth would I want to break out part of it? Isn't one program better than two?
Maybe, but here are the reasons I see for minimizing the JSP's Java scriptlets by using JavaBeans.

- **Separate the Work of the Page Designer and the Programmer**
  Early in the course we reviewed the need to control the number of people who are working concurrently on the JSP file itself. By moving the business rules and data acquisition/manipulation to JavaBean components, we allow each participant to focus on his/her area of expertise. The page designer does the HTML work while the programmer uses Java for all the behind-the-scenes stuff. The resulting overlap is minimal, so productivity is high.
- **Facilitate the Debugging Process**
  Debugging JSP is one of life's least pleasant experiences. Debugging a JavaBean, however, is far easier so why not take advantage of this happy circumstance by coding Java in JavaBeans?
- **JavaBeans Are Reusable Components**
  Since we constantly remind ourselves that everything we develop is part of the greater enterprise, it just makes good sense to develop reusable components whenever possible.
- **Granularity**
  JavaBeans were originally intended for small, compact programs, but no reason exists to limit either the size or complexity of the beans.
- **Portability**
  The JavaBean API, like its big brother, the Java API and the JSP API, is platform neutral. Write it once, run it anywhere.

Rules Of The Road
As the topic we're covering in this lesson is much larger in scope, the rules of the road are necessarily more lengthy and complex. This is the short course, though, so we'll keep it to a need-to-know basis. Where appropriate below, the code format is provided with the item which discusses it.

1. A JavaBean is a class, just like an applet or an application. All the usual Java rules apply.
2. A JavaBean needs to exist in a package.

package rayexamples ;

3. As always, the import statements follow the package declaration. The number of imports is unlimited.
   import java.io.Serializable ;
4. A JavaBean implements the serializable interface, as required from the JavaBean definition. The biggest reason for doing this is that the bean may easily be transferred and saved, even to disk. Don't worry about the other reasons. Just remember it's required, so we do it.
   public class bean1 implements Serializable {
5. Since our bean will be dealing with data (a name), declare it.
   String name ;
6. A bean has an empty constructor that is declared as public. "Empty" means it doesn't accept any input arguments as evidenced by the lack of anything between the parentheses. "Empty" does not mean that the method can't have code in it. In our case, we may instantiate the variable by giving it some content.
   ```
   public bean1( )
   {
   name = "World" ;
   }
   ```
7. All variables and methods used exclusively inside the class are private. They are protected from outside interference.
8. Each variable whose properties are provided and/or accessed outside the class are public, and each has the possibility of two public accessor methods. The prefixes "get" and "set" are expected to be appended to the variable name; the first letter of the property name is capitalized; and the bean accepts responsibility for processing them. Because the getter and setter methods are the bean's windows on the world, they are referred to as **exposed** methods.

- A **getter** method is always public, and it provides properties to an external source (like your JSP file).
  ```
  public String getName()
  {
   return name;
  }
  ```
- A **setter** method is always public, and it accepts properties from an external source (like your JSP file).
  ```
  public void setName(String name)
  {
  this.name = name;
  }
  ```
- An **is** method interrogates a boolean property. It is usually found in conditional statements, such as:
  ```
  if( isEndOfData( ) )
  {
  do something ;
  }
  ```
- In addition to the getter( ), setter( ), and is( ) methods, any method that needs to provide properties to an external source is simply declared as public. The example that follows illustrates this.

About Design Patterns

Let's revisit the definition of a JavaBean. A JavaBean is a reusable software component that may be manipulated visually in a builder tool. The part about the builder tool is what we need to focus on if we're going to consider the topic of the JavaBean design pattern. Here's why.

Any builder tool (IDEs like Jbuilder, VisualAge, etc.) uses the bean's .class file, not the .java file to get all that information about the bean. It needs to present you, the developer using a visual builder tool, with that bean's properties, methods and events available and ready for your use, and it needs to get that information from the compiled (.class) version of the bean. It does this by using **introspection**. In little words, introspection means the IDE can know all about the properties, methods and events as long as you stick to the naming conventions for the getter and setter

methods. If you don't have any getter or setter methods, you don't need to worry about it. That's all there is to it (at least for the short course).
So what do you do if you aren't using the JavaBean design pattern? You must implement the BeanInfo interface. Briefly, this means writing another Java program the IDE may use to get at the information it needs to recognize your bean's properties, methods and events. That's the bad news. The good news is that any bean-friendly IDE worth using will create the BeanInfo interface for you. This subject is out of the scope of this course. We'll just stick with using the JavaBean design pattern rules for naming conventions, whether we need them or not.

An Example
Our JavaBean example is another 'Hello World' variation. (You expected less?) The name of our Bean is Bean1.Java and its compiled name is Bean1.class, residing in the rayexamples package. It has one exposed variable (String name) and three exposed methods:

- **setName()** This is an accessor method that gets the property from the JSP (input).
- **getName()** This is an accessor method that sets the property in the JSP (output).
- **makeMeBig()** This is an accessor method that adds the word BIG to the output.

The purpose of our bean is to:

- Get an argument (name, like Joe or Betty).
- Use the getName() method to send back the name so it may be displayed as "Hello there, Joe" (or Betty) - or - to send it back so it may be displayed as "Hello there, World" if no argument is present.
- Use the makeMeBig () method to send back the name so it has the word "Big" added.

Bean1.Java
```
package rayexamples ;
import java.io.Serializable ;
public class bean1 implements Serializable
{
String name ;
/* The JavaBean's empty constructor */
public bean1()
{
name = "World" ;
}
/* THE SETTER METHOD - property returned to the JSP setProperty()*/
public void setName(String name)
{
this.name = name;
}
/* THE GETTER METHOD - property is set by the JSP setProperty().
If nothing is entered in the JSP, then name = World.
Otherwise, name is set to the name passed here from the JSP. */
public String getName()
{
return name;
}
/* THIS IS AN EXPOSED METHOD. It appends the word Big to the beginning of the name
string. This
kind of method may be used for enforcing business rules, formatting
the output, etc. */
public String makeMeBig()
{
name = "Big " + name ;
return name ;
}
}
```
That's the bean.

12. Putting JSP and JavaBeans Together

Finding and Instantiating a Bean into Your JSP

The <jsp:useBean> tag is JSP's gateway to using a JavaBean, and its most important functions are to provide an identifier (so you may reference the bean in your JSP page) and a scope (so you may control the bean's life span). Our example uses the following useBean tag:

<jsp:useBean id="myBean" scope="page" class="rayexamples.bean1" />

id=

This is the name by which you will reference the bean for as long as the bean exists. Our example calls it myBean.

scope=

This represents the life of the bean. It limits when and where your JSP page has access to what's going on inside the bean. Scope has four possible settings.

- **page (default)**
  The bean exists for the life of the page. This minimizes the bean's state, but that isn't a consideration for our example.
- **request**
  The bean's internal information is stored in the ServletRequest object and the bean will only be available for the current client request. This requires an understanding of the Java Servlet and is beyond the scope of this tutorial.
- **session**
  The bean is available to all of your JSP pages for the entire session unless, of course, you cause the bean to be destroyed.
- **application**
  The bean will remain until the application is reloaded or the server is restarted.

class=

This provides the names of the package and the class files. Our example tells the JSP engine that bean1.class is in the rayexamples package, so it knows where to look for it.

Passing Properties to a Bean

If you need to set and/or change a bean's property, you may do so either with the <jsp:setProperty> tag or by setting the property directly in the scriptlet using the bean's accessor functions. In either case, the bean must have been previously referenced.

If using the <jsp:setProperty> tag, two options are available, depending on the source of the information you wish to pass along to the bean's property.

<jsp:setProperty name="myBean" property="name" value="Sam" />

This loads the string "Sam" into the property "name".

<jsp:setProperty name="myBean" property="name" param="name" />

This gets the value to be loaded into the "name" property from the URL. One of our example URLs is http://localhost/examples/bean1.jsp?name=Betty-Boop. The jsp:setProperty's param="name" argument instructs the JSP to pass the String "Betty-Boop" from the URL to the bean's setName() method, where it will then be loaded into the name property.

Great, now you know how to use the <jsp:setProperty> tag. But where is it used? You have (surprise, surprise) two options.

Between the <jsp:useBean> & </jsp:useBean> tags

That way you know the bean's properties are initialized when the bean is instantiated. To do this, simply code the <jsp:useBean> tag, then code as many of the <jsp:setProperty> tags as you need and end it all with a </jsp:useBean> tag.

Outside the <jsp:useBean> tag

That way you may load the property any time you want, as many times as you want.

Getting Properties from a Bean

As with passing properties to a bean, you may use several ways to get them back.

- **accessor methods** : Using this method, <%= myBean.getName() %> will return a string containing the value of the property "name". You will see this in our example.
- **Ordinary exposed method** : As an example, myBean.makeMeBig() returns the property after completing its special processing.
- **<jsp:getProperty />** : This JSP tag requires two arguments, name= and property=. The first is the bean's id from the useBean tag and the second is the name of the property, as in <jsp:getProperty name="myBean" property="name" />.

Refresher: The JavaBean from Lesson 11

We're nearly ready to write a JSP page using the JavaBean created in Lesson 11. But first, let's review the bean's essentials. The name of our bean is Bean1.Java and its compiled name is Bean1.class. It resides in the rayexamples package. The bean has one exposed variable (String name) and three exposed methods. These include setName() (input from JSP), getName() (output to JSP) and makeMeBig() (adds the word 'Big' to the output).

The JSP page will use four ways to set the property "name" in the bean, including:

- The property's initial value (name = "World").
- The <jsp:setProperty> tag, passing a value to the bean's setName() method.
- The bean's setName() method directly.
- The <jsp:setProperty> tag, passing a parameter from the URL to the bean's setName() method.

Then the page will retrieve information from the bean in three ways, including:

- The bean's getName() method, sending back the name so it may be displayed.
- The <jsp:getProperty name="myBean" property="name" />.
- The makeMeBig() exposed method.

Here's the text of Bean1.java.

```
1. package rayexamples ;
2. import java.io.Serializable;
3. public class bean1 implements Serializable
4. {
5. String name ;
6. /* The JavaBean's empty constructor */
7. public bean1()
8. {
9. name = "World" ;
10.      }
11.      /* THE SETTER METHOD - property returned to the JSP setProperty()*/
12.      public void setName(String name)
13.      {
14.      this.name = name;
15.      }
16.      /* THE GETTER METHOD - property is set by the JSP setProperty().
17.      If nothing is entered in the JSP, then name = World.
18.      Otherwise, name is set to the name passed here from the JSP. */
19.      public String getName()
20.      {
21.      return name;
22.      }
23.      /* THIS IS AN EXPOSED METHOD. It appends the word Big to the beginning of
   the name string. This
24.      kind of method may be used for enforcing business rules, formatting
25.      the output, etc. */
26.      public String makeMeBig()
27.      {
28.      name = "Big " + name ;
29.      return name ;
30.      }
31.      }
```

Now that you remember the bean, let's write the JSP program.

The JSP Script

```
<html>
 <body>
 <!- Find and instantiate the bean, and refer to it as myBean -->
 <jsp:useBean id="myBean" scope="page" class= "rayexamples.bean1"/>
 STEP-1 Use the bean's default property "World"<BR>
 Hello there, <%= myBean.getName() %><BR>
 Hi there, <jsp:getProperty name="myBean" property="name" /><BR>
 Howdy, <%= myBean.makeMeBig() %><BR><BR>
 STEP-2 Set the name property to "Sam", using jsp:setProperty value="Sam"<BR>
 <jsp:setProperty name="myBean" property="name" value= "Sam"/>
 STEP-3 Use the bean's new property "Sam"<BR>
 Hello there, <%= myBean.getName() %><BR>
 Hi there, <jsp:getProperty name="myBean" property="name" /><BR>
 Howdy, <%=myBean.makeMeBig() %> <BR><BR>
 STEP-4 Set the name property to "Bernie", using setName()
 value= "Bernie"<BR>
 <% myBean.setName("Bernie");%>
 STEP-5 Use the bean's new property"Bernie"<BR>
         Hello there, <%=myBean.getName() %><BR>
 Hi there, <jsp:getProperty name="myBean" property="name" /><BR>
 Howdy, <%= myBean.makeMeBig()%> <BR><BR>
 STEP-6 Set the name property from the query string using jsp:setProperty
param="name"<BR>
 <jsp:setProperty name="myBean" property="name" param="name" />
 STEP-7 Use the bean's new property from the URL <BR>
 Hello there, <%= myBean.getName() %><BR>
 Hi there, <jsp:getProperty name="myBean" property="name" /><BR>
 Howdy, <%= myBean.makeMeBig() %> <BR><BR>
 </body>
</html>
```

The URL And Its Output

When you request the JSP page this is what your browser will show you.

STEP-1 Use the bean's default property "World"
Hello there, World
Hi there, World
Howdy, Big World

STEP-2 Set the name property to "Sam", using jsp:setProperty value="Sam"

STEP-3 Use the bean's new property "Sam"
Hello there, Sam
Hi there, Sam
Howdy, Big Sam

STEP-4 Set the name property to "Bernie", using setName(), value="Bernie"

STEP-5 Use the bean's new property "Bernie"
Hello there, Bernie
Hi there, Bernie
Howdy, Big Bernie

STEP-6 Set the name property from the query string using jsp:setProperty param="name"

STEP-7 Use the bean's new property from the URL
Hello there, Big Bernie

Hi there, Big Bernie
Howdy, Big Big Bernie

It's interesting to note that, in steps 6 and 7 when no external input is provided, the page refers to the output from step 5. In the following example, requesting the page with a parameter such as bean1.jsp?name=Betty-Boop gives a different result in steps 6 and 7.

STEP-1 Use the bean's default property "World"
Hello there, World
Hi there, World
Howdy, Big World

STEP-2 Set the name property to "Sam", using jsp:setProperty value="Sam"

STEP-3 Use the bean's new property "Sam"
Hello there, Sam
Hi there, Sam
Howdy, Big Sam

STEP-4 Set the name property to "Bernie", using setName(), value="Bernie"

STEP-5 Use the bean's new property "Bernie"
Hello there, Bernie
Hi there, Bernie
Howdy, Big Bernie

STEP-6 Set the name property from the query string using jsp:setProperty param="name"

STEP-7 Use the bean's new property from the URL
Hello there, Betty-Boop
Hi there, Betty-Boop
Howdy, Big Betty-Boop

13.JSP VS ASP Development

Which language is best for your dynamic web site? The winner of the battle between ASP and JSP is... Neither. Each of these languages has its strengths and weakness. Both languages are excellent choices for building dynamic web sites. Currently, the prime considerations for choosing between ASP or JSP should depend on the following criteria:

1. How large is your project?
2. Which expertise does your shop have?
3. How and where are you going to host your site?
4. How many concurrent users are you expecting?

Lets look at each of these criteria.

1. **How large is your project?**

   For large projects, I would say JSP is the clear winner. When used properly, you can logically assign your team resources easier in a JSP project. Some of your team can concentrate on building Java beans while other members concentrate on the presentation parts of your project. In addition, with the object-oriented nature of Java, it is easier to logically break apart the project among the project staff. On large projects, ASP tends to become spaghetti code rather easily and the maintenance issues can swamp you quickly. JSP is much cleaner and easier to maintain for large projects.

   For small projects, ASP tends to be better. This is due to three facts.

   1. ASP is very simple.
   2. There are fewer hassles in setting up and maintaining an ASP web site. The integration of ASP and Microsoft's IIS (Internet Information Web Server) is incredible. In addition, many excellent articles and books explain how to set up an ASP site. This makes setting up a web site based upon ASP a relatively simple task.

   Microsoft's ADO object. Using the ADO to data enable your project is simple and fast. It works with most data sources and is feature packed. This speed and power is passed on to you, which means you are that much more likely to finish that time critical project on time. If you are going to host your web site outside of your business then ASP is the clear winner. Currently it is much easier to host an ASP site over a JSP site. As of early June this year, I was able to find roughly 30 JSP hosting services worldwide. Out of this selection, I found only three hosting services that I was willing to consider for my web site. In fact, I ended up deciding to wait three months before making my final decision. When I examined ASP hosting services, I quickly lost count of all the possibilities. Over time, JSP hosting alternatives will improve but I don't see this changing this year. I must mention ASP had the same problem two years ago. The market place quickly caught up with demand and now hosting an ASP site is as easy as a normal HTML base site. I expect JSP to follow the same trend and by early next year we should see an explosion of sites that will host JSP.

   If you are going to host your own site, then this limitation is removed. Also if you are going to use a web server other than Microsoft's IIS, I strongly recommend JSP. This is due to the independent nature of JSP relative to a web server, and the ability to choose a JSP container (a container object is what the web server calls to process the JSP page) that best fits your needs. ASP, on the other hand, is really only meant to be used on Microsoft's web server. There are third party solutions to remove this limitation, but I prefer to bypass the issue altogether and use JSP, which is web server independent.

   If you are going to host your own site, then this limitation is removed. Also if you are going to use a web server other than Microsoft's IIS, I strongly recommend JSP. This is due to the independent nature of JSP relative to a web server, and the ability to choose a JSP container (a container object is what the web server calls to process the JSP page) that best fits your needs. ASP, on the other hand, is really only meant to be used on Microsoft's web server. There are third party solutions to remove this limitation, but I prefer to bypass the issue altogether and use JSP, which is web server independent.

   3. How many concurrent users are you expecting?

   As a web developer, I have concentrated on small sites where the number of concurrent users is under 500. At this size both ASP and JSP function well. However, when you begin

to scale your web sites to larger configurations, things change rapidly. Having used both ASP and JSP, I can clearly see JSP has much better support for larger web sites and for web farming solutions. If I were to deploy a web site designed for over 800 concurrent users, I would use JSP as my language of choice.

2. **Which expertise does your shop have?**

In other words, stay with what you are good at. I see no reason to drop ASP for JSP if your shop's expertise is in ASP. If your shop has expertise in neither language, I would lean towards ASP, since ASP has a easier learning curve over JSP/Servlets/Java. If your shop has experience in Java, then JSP would be the hands down winner.

Converting Between ASP and JSP

A series of annotated reference charts are provided to help programmers move efficiently between ASP and JSP development environments. Version 1.0 of this living document covers the following topics:

- implicit objects' commonly used functions and methods.
- scripting blocks .
- scripting comments .
- including files .
- file redirection .

Version 2.0 will expand the implicit objects section to include a complete listing of functions and methods. Then reference charts for VBScript and Java will be added in version 3.0. And, finally, in version 4.0, reference charts for ADO and JDBC will complete the set. General Notes and Definitions:

- Version 1.0 charts use ASP 2.0/IIS 4.0 and JSP 1.1/Tomcat. Where possible, exceptions and additions for ASP 3.0/IIS 5.0 are included.
- A container is the object processing the server page. For ASP, this container is called the ASP.DLL. For JSP, the container may be one of several possibilities. I currently use Tomcat as my reference container for JSP.
- Scriptlets are free-standing, server side, code snippets placed throughout a server page.
- JSP is case sensitive, so pay close attention to capitalization.
- For implicit objects, only specific usage notes are provided. These notes are not necessarily in the documentation. To find documentation for a particular function or attribute, look up the name of that function or attribute (eg. removeAttribute(String name)) at JSP Product Information.

Implicit Objects

Objects which must exist and form the core of ASP / JSP

Note: Due to spacing reasons I usually don't predicate functions and properties with their object. So, for example, in your code you would use application.setAttribute(String name,Object object) to set an application variable. Also for Version 1.0 of this chart I will only cover commonly used methods and properties of the implicit objects.

Implicit Objects : Objects which must exist and form the core of ASP / JSP

Functionality	ASP	JSP
Name of object	Application	application
Object type	N/A	javax.servlet.ServletContext
Storing an application variable	Application(String name) ="Your Data"	setAttribute(String name,Object object)*
Storing an application object	Set Application(String name) =Server.CreateObject(String name)	As Above
Retrieving an application	My_Variable =	getAttribute (String name)*

variable	Application(String name)	
Retrieving an application object	Set My_Object =Application(String name)	As Above
Removing an application variable or object	Contents.Remove(String name)	removeAttribute(String name)
*	* Note: You must use one of the object containers for primitive data types, for example Integer rather than int.	
Contents collection Note: This is the way to track of all the items added to the application object	Contents Code Example: <% Dim ls_write For Each Key in Application.Contents ls_write = Key + " : "+ Application(Key) Next %>	getAttributeNames() This returns an enumeration of string objects containing the names of attributes stored in the application object.
Lock and Unlock	Lock prevents other users from changing the application object's properties. Unlock releases the application to be free again.	There are no directly comparable methods in JSP's application object. However, JSP has great threading support, and with proper thread control/making your page thread safe, 1 can achieve similar results.
Determining which container you are using.	N/A	getServerInfo()
Determining servlet API version numbers	N/A	getMajorVersion() getMinorVersion()
Writing to the container's log file	N/A: See Response.AppendToLog(string)	log(String msg)
Determining a file's MIME type	N/A	String= getMimeType(String file)
Finding a file's real path	N/A See ASP Server.MapPath(Path)	String =getRealPath(String virtualpath)
Finding a URL to a resource/file	N/A	URL = getResource(String path)

Config Object :This object stores servlet configuration data, but is rarely used.

Functionality	ASP	JSP
Name of object	ASP doesn't have a similar object.	Config
Object type	N/A	javax.servlet.ServletConfig
Name of servlet	N/A	getServletName
Reference to Servlet context	N/A	getServletContext()
Returns the names of the servlet's initialization parameters	N/A	getInitParameterNames()

Get value of initialization parameter	N/A	getInitParameter(String name)

Error Object : This object contains data about any error that has occurred in the script.

Functionality	ASP	JSP
Name of object	ASPError	Exception
Object type	N/A	java.lang.Throwable
Special notes	A new ASP 3.0 / IIS 5.0 object. You call Server.GetLastError method to receive the ASPError object.	You only have access to this object if you declare your JSP page as an error page. To do this you use the declaration: <%@ page isErrorPage="true" %>
Error message	Description ()	getMessage()
Full error	ASPDescription()	toString()
Error trace	N/A	printStackTrace(out)
Error position	Line Column	N/A

Out : An object used to write and control the output buffer from the server to the browser.

Functionality	ASP	JSP
Name of object	Response	Out
Object type	N/A	javax.servlet.jsp.JspWriter
Special notes	ASP uses the Response Object to perform most of these functions	
Writing data to output buffer	Write *variant*	print(object or primitive data type)
Writing binary data	BinaryWrite *data*	You need to access the Java OutputStream class and use its binary write methods.Quick Example: ServletOutputStream Output = response.getOutputStream();Output.write(Btye[] buffer);
Clear out buffer	Clear	clearBuffer()
Send current buffer to client	Flush	flush()
Stop processing the current page	End	close()This is different than end. It closes off the current output stream. The JSP page is still free to finish its processing.

Page Object The current servlet page representation of the JSP page being executed.

Functionality	ASP	JSP
Name of object	ASP doesn't have an object like this one.	page
Object type	N/A	java.lang.Object

Special notes		This object is rarely directly used, since when using Java as your scripting language, you already have access to the whole page object through the "this" keyword. This object is present for giving other scripting languages access to your JSP pages methods and attributes.

PageContext Object: This object provides access to all the other implicit objects and their attributes. In the processing a single JSP page, you might call this object the glue that holds everything together. It is also the object ultimately responsible for managing scope and transferring control from one page to another page. This object will be more fully explained in Version 2.0 of this document.

Functionality	ASP	JSP
Name of object	ASP doesn't have an object like this one.	pageContext
Object type	N/A	javax.servlet.jsp.PageContext

Request Object : An object to receive information from the client (browser).

Functionality	ASP	JSP
Name of object	Request	request
Object type	N/A	Subclass of:javax.servlet.ServletRequestUsually:javax.servlet.HttpServletRequest
Certification details	ClientCertificate(Key[Field])	N/A
Cookie details	Cookies(cookie)[(key).attribute]	cookie[]=getCookies()
Getting form data	string = Form(element)[(index)]For example:mydata= Request.Form("date")	string = getParameter(Name)Enum = getParameterNames()string[] = getParameterValues(name)For example:ls_form = request.getParameter("date");
Getting query data	QueryString(element)[(index)|.Count]	getParameter(Name)getQueryString() (entire query string)
HTTP headers sent by the client	ServerVariables (server environment var)For example: ServerVariables (ALL_RAW) returns to you all the headers in raw format	getHeaderNames()getHeader(name) getHeaders(name)getIntHeader(name)getDateHeader(name)

Response Object: An object to send information to the browser. ASP and JSP treat the response objects slightly differently. ASP only has the Response object for controlling the output to the client. JSP splits the functionality into two objects. The Response object in JSP is the actually object being sent back to the client. JSP also uses the out object for the functionality to write to the output buffer. Most of the calls you will use as a JSP programmer will be found in the out implicit object.

Functionality	ASP	JSP
Name of object	Response	response
Object type	N/A	Subclass of: javax.servlet.ServletResponseUsually:javax.servlet.HttpServletResponse

Buffer page output	Buffer = True/False	By default, JSP buffers the first 8k of output. After this limit has been reached, it flushes the output. You use the page directive to change the buffer size and state. For example, to turn buffering off: <%@ page buffer= "none" %>
Enable/Disable proxy server caching	CacheControl =Private/Public	setHeader("Pragma","no-cache")setHeader("Cache-Control","no-cache")
Adding cookies	Cookies(cookie)[(key).attribute] = value	addCookie(cookie)
Adding an HTML header	AddHeader Name,Value	setHeader(Name,Value)
Redirect client to a new page	Redirect *URL*	sendRedirect(Absolute URL)(see also encodeRedirectURL() if you are using URL rewriting as your session management.)
Send error to client	N/AYou can use VbScript to raise an error, but it isn't as clean.	sendError(int code,String msg)
Encode an URL	N/AFunction of the server object.See Server.URLencode.	encodeURL(name)Note, if you are using URL rewriting as your session management, then this function also appends the session ID for this purpose.
Set output MIME type	ContentType = "MIME TYPE"	setContentType("MIME TYPE")

Server Object :This object provides access to methods and properties on the server.

Functionality	ASP	JSP
Name of object	Server	JSP doesn't have a Server object. The functions found in the ASP server object are distributed across the other JSP implicit objects.
Object type	N/A	N/A
Creating an object on the server.	CreateObject(Object id)	Use standard Java syntax to create objects.
HTML encode a string	HTMLEncode(String)	N/A
Finding a files real path	MapPath(Path **)**	N/Asee JSP application.getRealPath(virtualpath)
Encode an URL	URLEncode(String)	N/Asee JSP response.encodeURL(name)
Forwarding control to new page	(IIS 5.0 / ASP 3.0) Transfer	N/A see **file redirection**
Timeout for server side scripts	ScriptTimeout = Seconds	N/AThis is a feature that would be container specific in JSP. Currently Tomcat doesn't have a timeout feature.

Session Object : An object to share information, for one user, across multiple pages, while visiting a web site. A session object is a method of retaining state for a normally stateless HTTP web site.

Functionality	**ASP**	**JSP**
Name of object	Session	session
Object type	N/A	Javax.servlet.http.HttpSession
Special notes	ASP manages Session by using cookies only.	JSP has two methods of Session management. 1. Using cookies 2. URL rewriting
How to close a session and release its resources	Abandon	invalidate()
Storing a session variable	Session (String name) ="Your Data"	setAttribute(String name,Object object)*
Storing a session object	Set Session (String name) = Server.CreateObject(String name)	As Above
Retrieving a session variable	My_Variable = Session(String name)	getAttribute (String name)*
Retrieving a session object	Set My_Object = Session(String name)	As Above
Removing a session variable or object	Contents.Remove(String name)	removeAttribute(String name)
*	* Note: You must use one of the object containers for primitive data types, for example Integer rather than int.	
Contents collection	Contents	getAttributeNames()This returns an Enumeration of String objects containing the names of attributes stored in the application object.
The session ID	SessionID	string =getId()
Setting the timeout period	Timeout(Minutes)	setMaxInactiveInterval(int interval in seconds)
Getting the timeout period	N/A	int =getMaxInactiveInterval()
Getting a location identifier	LCID(Locale ID)	see Request Object getLocale
Disabling the session	A Directive Command<%@ EnableSessionState = False%>	A Directive Command<%@ page session="false"%>

Scripting Blocks

Scriptlet declaration : How the server side script is separated from client side script.

ASP	JSP
<% Your Server Side Script %>	<% Your Server Side Script %>

Expression :A shortcut method to put data straight into the output buffer.

ASP	JSP
<%= Your_Variable %>	<%= Your_Variable %>

Declaration : How to declare variables and functions/methods which can then be used by any scriptlet or expression on this page.

ASP	JSP
<% Your Function %>	<%! Your Function %>
Note: This means your variable or function is positional. That is you must place your function or variable code in the script before its first use. Also, ASP doesn't have a similar concept of the class variable.	• A declaration is a complete logical unit of work. • Declarations don't produce any results into the output buffer. • Declarations are initialized when the JSP page is initialized. • Code within declarations is made available to other declarations, expressions and scriptlets. • Variables created within a declaration block become instance variables. Since JSP Containers tend to only make one instance of a page to share requests against, this can mean that all current instances of the JSP page have access to the same variable. In other words, if your page is currently being accessed at the same time by 10 users, those 10 users are sharing one variable. For all practical purposes you are creating a class variable. If you want to avoid this, you can create variables within your scriptlet block. Variables declared within a scriptlet block are local to that script block. You can also set <%@ page isThreadSafe="false" %> to get around this issue. However, using the isThreadSafe property has serious performance issues on high traffic sites. If you want to use your variable as a class variable we recommend that you declare it as such and don't count on this behavior as being the rule. • You can also declare class variables and functions in a declaration block.

Directive : How do you tell the container how to perform special processing of the page. In effect directives provide information for the compilation/translation phase of the server page.

ASP	JSP
<%@ Your Directive %>	<%@ Your Directive %>
Example of setting the scripting language: <%@ LANGUAGE="VBSCRIPT" %> Please note many of the directives found in JSP are not directives in ASP. Rather you will find what you might think of a directive is actually a property of one of the ASP objects. Setting these properties will have global effects on the page. For example to turn page buffering on you would write the following code: <%response.buffer=true%>	• Directives are messages to the JSP container. • Directives don't produce any results into the output buffer. • Directives are processed when the JSP page is initialized. Example of setting the scripting language: <%@ page language= "java"%> Example to turn page buffering on : <%@ page buffer="64k" autoFlush= "true" %>

Actions : Actions are XML based tags. These tags make it possible to build simple HTML like tags for web developers to use to perform complicated or repetitive tasks.

For example: Instead of using a scriptlet to build a report table, you can build special Java classes to handle building the report. You then build a simple action tag to interface with these report classes. So, all your web site designer needs is a one line call to your custom tag. A tag the user interfaces with might look like this :

```
<report:build SQL="select * from employee" style="simple"/>
```

This style of code would be easier to maintain than a complicated scriptlet.

ASP	JSP
N/A	<jsp:Action> or <yourtag:YourAction>
N/A	• These tags are case sensitive. • There are a standard set of actions which must be supported by all containers. For example you will always be able to use the <jsp:forward> action. • Attributes must be quoted (see example). • Tags must have a closing delimiter. • You can build new actions. • You can extend existing actions. • Actions can put data into the output buffer. • Actions can access, modify and create objects on the current server page. Example to access a JavaBean <jsp:useBean id="myBean" class="beans.htmlBean" /> Note, when you hear the term tag library, it refers to a custom built collection of actions.

Scripting Comments

Comment Type	ASP	JSP
In line script comment. Note: These comments are based upon the scripting language used.	VbScript uses: a single quote ' <% 'Your Comment %>	Java uses: // for a single line /* */ for multiple lines. <%//my comment %> or <% /* my comment */ %>
Special Comments	N/A	JSP Comments <%-- your comment --%> These comments are only visible from the original JSP file. This form of comment is not processed by the container and is not passed onto the servlet. Also this comment type doesn't nest.

Including Files

Static Includes:

How to include files before the page has been processed.

A fundamental difference exists between the way files are included in ASP and JSP. ASP doesn't directly support include files, rather including files in an ASP application is a web server process. After the web server combines all the files to be included it then sends the file off to be processed by the ASP DLL. In a JSP application the web server doesn't preprocess the JSP page but rather hands the JSP pages straight off to the JSP container. The JSP container then performs the work of including files.

ASP	JSP
Command Format: <!--#include file="Your File.asp" --> <!--#include virtual ="/Your File.asp"--> Usage Notes: The keyword "Virtual" indicates you are using a full virtual path from a virtual directory in your Web site. When you use the keyword "File", you are using a relative path from the directory containing the document.	**Command Format:** You can include a file using a directive: <%@ include file="Your File" %> or in tag format <jsp:directive.include file="Your File" %>
• You can include HTML or additional ASP scripting elements. • The files are combined into a single source code and then translated into a servlet. • You cannot use scripting conditional logic to make the include statements conditional, since the include directive is preprocessed before scripting code executes to process the conditional. • The files are combined into a single source code and then ASP logic is performed.	• The file can be passed as either a relative path or an absolute path. • You can include HTML or additional JSP scripting elements. • You cannot use scripting conditional logic to make the include statements conditional, since the include directive is preprocessed before scripting code executes to process the conditional. • The files are combined into a single source code and then translated into a servlet. • The JSP container will only recombine the include files and the main page when the main page has been modified. So if you change an include file but not the page that references it, no new servlet will be generated. • The included page shares all local information and shares the same pageContext Object. • Faster than using the jsp:include action.This is due to the static include being preprocessed during page translation and compiled into a servlet.

Dynamic Includes: How to include files while the page is being processed.

ASP	JSP
Command Format: IIS 5.0 / ASP 3.0 Introduces Server.Execute. **Usage Notes:**This function will let you execute an ASP page from within another ASP page	**Command Format:** <jsp:include page="Your File" flush="true"> If you desire to pass additional information to the included page use the following format: <jsp:include page="Your File" flush="true"> <jsp: param name = "Name" value ="Data" /> </ jsp:include>**Usage Notes:**You can pass as many parameters as you need using this method. The flush property is used to tell the page to flush the current output buffer of the current page before the action of the including the new file. Currently you can only set this property to true. Since the previous page's buffer has been flushed and you cannot forward to another page.
	• You may include a text file(HTML), a CGI script, a servlet or another JSP page. You can include dynamic real time content. • The included page receives a new pageContext object. • The include page shares the request and session objects as the original page. • You can dynamically pass additional information to the

	include page by using parameters. • The processing of the included file happens during the request of the current page this means: a) you can dynamically change what your include processes. b) your latest include file will always be processed. c) you can not only include contents of a file but you can also include dynamic output. • Restricts one from setting response headers and the response code.

File Redirection
Browser Redirection:

ASP	JSP
response.redirect("to_File.asp")	response.sendRedirect(Is_Absolute_Url)
• The file may be passed as either a relative path or an absolute path. • Once the header has been sent down to the user, you cannot redirect to a different page. • This command has the server send a HTTP response to the browser asking it to load a new URL	• This only works with an absolute URL. (Major Bummer). • You can only redirect to a new page provided no output of the original page has been sent to the browser. • I have read that sendRedirect actually, creates a new thread for the new pages redirection, while the old thread and page keeps executing until it finishes at end of the original page. I haven't seen this behavior yet and don't know if it is part of the JSP specs or a by product of a JSP containers implementation.

Server Redirection:

ASP	JSP
In IIS 5.0 / ASP 3.0 You can use Server.Transfer. **Usage Notes:** Server.Transfer ends execution of the current asp page without clearing the output. The new page has access to same set objects (Application, Request, Response, Server, Session) as the starting file. To the browser it will appear you have the originally requested page not the page to which you are transferred.	Using an action: <jsp:forward page="home/Default.jsp" / > OR with parameters <jsp:forward page="home/Default.jsp" > <jsp:param name="source" value="entry"/> </jsp:forward> **Usage Notes:** You may pass as many parameters as you need using this method by using the param tag. The forward action ends execution of the current JSP page and removes any existing buffered output. The new page has access to following objects (Application, Request, Session) as the starting file. A new pageContext object is generated for the page. To the browser it will appear you have the originally requested page not the page to which you are transferred.
	• You can forward to a text file(HTML), a CGI script, a servlet or another JSP page. • You can only forward to a new page provided no output of the original page has been sent to the browser.

Browser Redirection:
How to have your client's browser request a new page.
response.sendRedirect(Is_Absolute_Url)
- This only works with an absolute URL.

- You can only redirect to a new page, provided no output of the original page has been sent to the browser.
- Several listserv postings have indicated that sendRedirect actually creates a new thread for the new page's redirection, while the old thread keeps executing to the end of the original page. I haven't seen this behavior yet, and I don't know if it is part of the JSP specs or a by-product of a JSP container's implementation.

Example:
```
<%  String ls_Absolute_Url = ("http://" + request.getHeader("host") +
"/home/Default.jsp");
        /* redirection by browser request */
        response.sendRedirect(ls_Absolute_Url);
 %>
```
Server Redirection: How to have the server sent to a new page.

<jsp:forward page= URL / >
- You can forward to a text file (HTML), a CGI script, a servlet or another JSP page.
- You can only forward to a new page, provided no output of the original page has been sent to the browser.
- You may pass as many parameters as you need with this method by using the param tag.
- The forward action ends execution of the current JSP page and removes any existing buffered output. The new page has access to application, request, and session objects as the starting file. A new pageContext object is generated for the page. To the browser, it will appear you have the originally requested page, not the page to which you are transferred.

Example:
```
<jsp:forward page="home/Default.jsp" >
       <jsp:param name="source" value="entry"/>
</jsp:forward>
```

Scriptlet declaration:
How the server side script is separated from client side script.
<% Your Server Side Script %>
Example of scriptlet : The following scriptlet determines the absolute URL of a home page and then sends a request to the client browser to open up the home page.
```
 <%  String ls_Absolute_Url = ("http://" + request.getHeader("host") +
"/home/Default.jsp");
       /* redirection by browser request */
        response.sendRedirect(ls_Absolute_Url);
  %>
```

Expression:
A shortcut method to put data straight into the output buffer.
<%= Your_Variable %>
Example:
```
<%= "test" %>
or
<% String ls_data = "winter"; %>
<Br> the season is <%= ls_data %>
```

Declaration:
How to declare variables and functions/methods which can then be used by any scriptlet or expression on the current page.

<%! Your Function %>
- A declaration is a complete logical unit of work.
- Declarations don't produce any results into the output buffer.
- Declarations are initialized when the JSP page is initialized.
- Code within declarations is made available to other declarations, expressions and scriptlets.

- Variables created within a declaration block become instance variables. Since JSP containers tend to only make one instance of a page to share requests against, this can mean that all current instances of the JSP page have access to the same variable. In other words, if your page is currently being accessed at the same time by 10 users, those 10 users are sharing one variable. For all practical purposes, you are creating a class variable. If you want to avoid this, you can create variables within your scriptlet block. Variables declared within a scriptlet block are local to that script block. You can also set
<%@ page isThreadSafe="false" %> to get around this issue. However, using the isThreadSafe property has serious performance issues on high traffic sites. If you want to use your variable as a class variable, we recommend that you declare it as such and don't count on this behavior as being the rule.
- You can also declare class variables and functions in a declaration block.

Example:
```
<%! public int determineArea(int ai_len, int ai_width)
       {    return ( ai_len * ai_width);
       }
%>
<% int li_area;
      out.print("the area of your room is :" );
      out.print(determineArea(10,15));
%>
```

Directive:
How to tell the container how to perform special processing of the page. In effect, directives provide information for the compilation/translation phase of the server page.

<%@ Your Directive %>
- Directives are messages to the JSP container.
- Directives don't produce any results into the output buffer.
- Directives are processed when the JSP page is initialized.

Examples:
1) setting the scripting language:
```
<%@ page language= "java"%>

2) turn page buffering on :
<%@ page  buffer="64k"
         autoFlush= "true"
%>
```

Actions
Actions are XML-based tags. These tags make it possible to build simple HTML-like tags for web developers to use to perform complicated or repetitive tasks.

<jsp:Action> or <yourtag:YourAction>
- These tags are case sensitive.
- A standard set of actions exist which must be supported by all containers. For example, you will always be able to use the <jsp:forward> action.
- Attributes must be quoted (see example).
- Tags must have a closing delimiter.
- You can build new actions.
- You can extend existing actions.
- Actions can put data into the output buffer.
- Actions can access, modify and create objects on the current server page.

Example: How to access a JavaBean
<jsp:useBean id="myBean" class="beans.htmlBean" />
Example: Building your own action.
Instead of using a scriptlet to build a report table, you can build special Java classes to handle building the report. You then build a simple action tag to interface with these report classes. So, all

your web site designer needs is a one line call to your custom tag. A tag with which the user interfaces might look like this :

```
<report:build SQL="select * from employee" style="simple"/>
```

This style of code would be easier to maintainthan acomplicated scriptlet.

Note, when you hear the term tag library, it refers to a custom built collection of actions.

JSP File Redirection.

How you transfer control from one JSP page to another JSP page.

Browser Redirection: How to have your client's browser request a new page.

response.sendRedirect(ls_Absolute_Url)

- This only works with an absolute URL.
- You can only redirect to a new page, provided no output of the original page has been sent to the browser.
- Several listserv postings have indicated that sendRedirect actually creates a new thread for the new page's redirection, while the old thread keeps executing to the end of the original page. I haven't seen this behavior yet, and I don't know if it is part of the JSP specs or a by-product of a JSP container's implementation.

Example:

```
<%  String ls_Absolute_Url = ("http://" + request.getHeader("host") +
"/home/Default.jsp");
        /* redirection by browser request */
        response.sendRedirect(ls_Absolute_Url);
 %>
```

Server Redirection

How to have the server sent to a new page. Through Browser Redirection and Server Redirection

<jsp:forward page= URL / >

- You can forward to a text file (HTML), a CGI script, a servlet or another JSP page.
- You can only forward to a new page, provided no output of the original page has been sent to the browser.
- You may pass as many parameters as you need with this method by using the param tag.
- The forward action ends execution of the current JSP page and removes any existing buffered output. The new page has access to application, request, and session objects as the starting file. A new pageContext object is generated for the page. To the browser, it will appear you have the originally requested page, not the page to which you are transferred.

Example:

```
<jsp:forward page="home/Default.jsp" >
       <jsp:param name="source" value="entry"/>
</jsp:forward>
```

Including Files

How you can combine several pages to form a single new page in JSP. Using Static Includes and Dynamic Includes

Static Includes

Including the file before the JSP page is compiled into a servlet.

<%@ include file=File %> or <jsp:directive.include file=File %>

- The file may be passed as either a relative path or an absolute path.
- You may include HTML or additional JSP scripting elements.
- You cannot use scripting conditional logic to make the include statements conditional, since the include directive is preprocessed before scripting code executes to process the conditional.
- The files are combined into a single source code and then translated into a servlet.
- The JSP container will only recombine the include files and the main page when the main page has been modified. So, if you change an include file but not the page that references it, no new servlet will be generated.

- The included page shares all local information and the same pageContext Object.
- Static includes are faster than dynamic includes (jsp:include). This is due to the static include being preprocessed during page translation and compiled into a servlet.
- In a JSP application the web server doesn't preprocess the JSP page, but rather hands the JSP pages straight off to the JSP container. The JSP container then performs the work of including files.

Example:

You may include a file using a directive:

```
<%@ include file="inc_header.jsp" %>
```

or in tag format

```
<jsp:directive.include file="inc_header.jsp" %>
```

Dynamic Includes

Including the file while the servlet is being processed.

<jsp:include page="Your File" flush="true">

- You may pass as many parameters as you need using this method.
- The flush property is used to tell the page to flush the current output buffer of the current page before the action of the including the new file. Currently, you can only set this property to true since the previous page's buffer has been flushed and you cannot forward to another page.
- You may include a text file (HTML), a CGI script, a servlet or another JSP page. You may also include dynamic real time content.
- The included page receives a new pageContext object.
- The included page shares the request and session objects with the original page.
- You may dynamically pass additional information to the included page by using parameters.
- The processing of the included file happens during the request of the current page. This means:
    a. You may dynamically change what your include processes.
    b. Your latest include file will always be processed.
    c. You can not only include contents of a file, but also dynamic output.
- Dynamic includes restricts you from setting response headers and the response code.

Example:

```
<jsp:include page="inc_header.jsp" flush="true">
```

If you desire to pass additional information to the included page, use the following format:

```
<jsp:include page="inc_header.jsp" flush="true">
      <jsp: param name = "Name" value ="Data" />
</ jsp:include>
```

14.JSP Implicit Objects

Application Object
This document does not replace the Java docs for javax.servlet.ServletContext. For documentation purposes, I am using Tomcat 3.1 as my reference container. Depending on the container you use, you can get slightly different results from some of the functions.

Application Object
An appliction object shares information among all users of a currently active JSP application. This object is also used to communicate with the servlet container running the current JSP page. It's up to the container to define how it determines a "web application." Usually, a "web application is defined as a specific set of files under a predefined directory. The container defines one application object per Java Virtual Machine. This means that every page within a JSP web application uses the same Java Virtual Machine. It is possible set up a JSP web application as a distributed website running across several Java Virtual Machines. When this is the case, the application object can no longer be used as a centralize data storage location for your web site (since you now have more than one application object). It is recommended you use a database to centrally store information when using a distributed setup.
Methods / Properties
The methods/properties described include the following:

- getAttribute
- getAttributeNames
- getMajorVersion/getMinorVersion
- getMimeType
- getRealPath
- getResource
- getServerInfo
- log
- removeAttribute
- setAttribute

Method/Property	**Object = getAttribute (String name)**
Purpose	Retrieves data which has been stashed away in the application object.

Example	<%
	// first, let's set up an application variable
	String ls_test = "Data to store in Application";
	application.setAttribute("Testing", ls_test)
	// now lets retrieve it
	Object lobj_getdata = application.getAttribute("Testing");
	/* Notice we have to type the object since the getAttribute returns an Object */
	String ls_getdata = (String)lobj_getdata;
	%>

Notes	• getAttribute will return a null value if the named property doesn't exist within the application object. • Primitive data types require one extra step of conversion from the primitive object container class to the primitive type. For example: <% Integer li_test = (Integer)application.getAttribute("MyData"); int li_primitive = li_test.intValue(); %> • Your servlet container may store data in the application object. Check with with your container's documentation.

Method/Property	Enumeration = getAttributeNames()
Purpose	Returns a list of all the names of the attributes currently stored in your application object.

Example	<%
	String ls_name = "";
	Enumeration enum_app = application.getAttributeNames();
	for(;enum_app.hasMoreElements();)
	 {
	        ls_name = enum_app.nextElement().toString() + "
	";
	        out.print(ls_name);
	 }
	%>

Method/Property	int = getMajorVersion() int = getMinorVersion()
Purpose	Determining servlet API version numbers

Example	<%
	   int li_major = application.getMajorVersion() ;
	   int li_minor = application.getMinorVersion() ;
	%>

Notes	In Tomcat 3.1, this returns 2 for the major version and 2 for the minor version

Method/Property	String= getMimeType(String file)
Purpose	Determining a file's MIME type.
Example	<% String ls_mime = application.getMimeType("C:\\Temp\\test.txt"); %> Notice I used the escape character \\ , "C:\Temp\test.txt" would give an error since \T represents an unknown escape character. Remember in Java "\" initiates an escape sequence to print special characters and to put a "\" into a string you must use "\\". In this example a mime type of 'text/plain' is returned.
Notes	• Returns null if the file's mime type is unknown.

Method/Property	String =getRealPath(String virtualpath)
Purpose	Finding a file's real path.
Example	<% String ls_file = application.getRealPath(request.getRequestURI()) ; %> In this example, request.getRequestURI() returns the currently requested URL. In this example my requested URL is: /Test/Examples/Bean_Example.jsp Then getRealPath translates it into the real hard path on my machine. The real path ends up being: C:\Tomcat\webapps\Test\Examples\Bean_Example.jsp
Notes	

Method/Property	URL = getResource(String path) throws java.net.MalformedURLException
Purpose	Finding a URL to a resource/file
Example	<% java.net.URL l_URL = application.getResource(ls_file); %>

Notes	• getResource lets you access resources from your script which were otherwise unavailable. For example, you can load resources from remote systems, files stored in a database or files stored in a .war file • Returns null if no resource is mapped to the path. • Requesting a .jsp page returns the JSP source code, not the compiled servlet.

Method/Property	**String = getServerInfo()**
Purpose	Determining which container you are using. Function requires that a server returns at least the server name and server version number.
Example	<% String ls_server = application.getServerInfo() ; %> In Tomcat this returns the following string: Tomcat Web Server/3.1 (JSP 1.1; Servlet 2.2; Java 1.2.2; Windows NT 4.0 x86; java.vendor=Sun Microsystems Inc.)
Notes	• Container may return additional information within (). See above for an example of what additional information Tomcat returns.

Method/Property	**log(String msg)** **log(String message, Throwable throwable)**
Purpose	Sending a message to the container's log file.

Example	<%
	try
	{
	String ls_bad_data = request.getParameterValues("None")[0] ;
	}
	catch(Exception e)
	{
	application.log("Drat an Error",e);
	}
	%>

Notes	This code causes an error since my request parameter doesn't exist. This writes the error to the Tomcat servlet.log file found in the log directory. The above code causes the following error line to the log: Context log path="/lift" :Drat an Error Depending on the exception, a stack trace may also be passed into the log file.

Method/Property	**removeAttribute(String name)**
Purpose	Removing an application variable or object

Example	<%
	// This adds a "Testing" object to the application object
	String ls_test = "Data to store in Application";
	application.setAttribute("Testing", ls_test);
	
	// Now this removes "Testing" from the application object
	application.removeAttribute("Testing");
	%>

Notes	• After you remove an object, any calls to retrieve it will return null.

Method/Property	setAttribute(String name,Object object)
Purpose	Storing data into the application object.
Example	<% String ls_test = "Data to store in Application"; application.setAttribute("Testing", ls_test); %>
Notes	• You may not use java.*, javax.*, and sun.* as attribute names.

Session Object
This document does not replace the Java docs for javax.servlet.http.HttpSession. Instead, this document adds helpful code examples and extended explanations.

Overview
This discussion of the session object is broken down into three primary sections, including: Session Description , Session Methods & Examples and Session Management Example

Session Object Description
The session object is used to share information for one user across multiple pages while visiting a web site. In other words, a session object is a way of retaining state for a normally stateless HTTP web site. By default, all JSP pages have access to the implicit session object. The basic way a JSP container tracks a session is as follows:

- The client makes a request to the server and the server sends the client a cookie with session / current browser id information.
- When the same client sends back new requests to the server, the server may now validate the cookie to see if that user has a session. If there is a match, you may use the appropriate session for a given client.
- If the client opens a new browser window, then a new session is created for the new browser instance.
- Now things get more confusing. If you open up a new browser from within your current browser then those two browsers will share the same session. However, in some browsers, you may set the preferences so new browser windows are opened up with new process id's. When this is the case (and its rare) these new browser windows will have a new session created for the client.

When using the session object consider the following points:

When not to use the session object.
The session is a powerful object, giving your web site capabilities that match traditional programs. Sometimes, however, you may not want to use the session object. Session objects are unnecessary when:

- You are not tracking information across your web pages.
- You don't have objects which you want to be persistent across several pages.

If you don't want a JSP page to have access to a session object, you may use the page directive to prevent the page from interacting with the session object (eg, <%@ page session="false" %>). The advantages of not using the session object are that your pages will run faster and use less memory in the JVM. If you try to access the session object in a page which isn't participating with sessions, you will receive a fatal error from your JSP container.

Issues of storing objects in your session object.
While using the session object adds some overhead to your JSP application, this overhead is more than made up by the speed you gain when re-accessing commonly used objects. This is because it is much faster to reference an object that exists, rather than rebuilding the object from scratch. As with everything, however, you must balance several considerations when storing objects in the session object. You must be aware of your memory usage. Each user has at least one session (possibly more), so when your web site is fully loaded your memory usage may be impacted in a negative manner. For example, if you store 512k per user in the session object and you expect

1000 users to be online at the same time, this translates into 1/2 of gigabyte of memory required **just** for your session variables (don't forget to add the memory required for your normal processing on top of that). Another important consideration is that we aren't necessarily talking about concurrent users, since a session has a lifetime that may exceed the time the user is actually accessing the web site. Session variables are a great method to speed up your application but only when you have more than enough memory to meet your needs!

How JSP tracks your session object.
You may manage your session in two ways, using cookies or URL rewriting. By default, JSP uses cookies to manage sessions. All JSP containers support this method of session management and using it is a very easy affair.

URL rewriting.
When cookies are not available, then an alternate method called URL rewriting is available to manage your session. The methods of implementing URL rewriting will vary among the various JSP containers. Some containers may offer automatic URL rewriting while others force you to implement a manual solution. In URL rewriting, the session id is written as part of your URL's query string parameter. Two methods in the response explicit object exist to help the programmer use URL rewriting. These methods are response.encodeRedirectURL() and response.encodeURL(). It is interesting to note these two functions will do nothing if cookies are enabled (since then the cookie method of session management is used by the container) or your Container doesn't support URL rewriting. In using URL rewriting, you must make sure every time you move from page to page you encode the session into the URL string. You will find a simple example of URL rewriting at the end of this page .

Synchronization of objects stored in the session object.
Objects stored in the session object may have unique synchronization needs, since multiple pages can access the same object stored in a session object at the same time. It is up to the author of the JSP page to ensure synchronization where it's required by the stored session objects!

Distributable web site issues.
Additional requirements are placed upon objects stored in a session object when your JSP application is marked as distributable. In this case, any object stored in the session object must implement the serializable interface. You will receive an IllegalArgumentException when this requirement isn't met.

More about memory usage.
When your session object expires, all the memory being used by the session object then becomes available for the garbage collector to clean up. But remember the way the garbage collector works. Just because an object is eligible to be cleaned up by the garbage collector doesn't mean it goes away. It still lives in memory until the garbage collector runs. This means memory is not freed up as soon as your session object times out or gets invalidated by the programmer.

Special events for objects.
JSP provides a method for objects to react when being placed into or being retrieved out of the session object. If your object being stored in the session implements the HttpSessionBindingListener interface, the session object will trigger the appropriate event in the object when the conditions are met. One very practical use of this is to allow your objects to automatically save themselves to the database when the user's session ends!

Security and recently deprecated methods.
Earlier versions of JSP allowed easier access to session objects not associated with the current user. As this poses a major security risk, many of these functions have been deprecated. Consider User A who logs onto the system. A JSP page has access to User A. This page also tracks other session variables from other users. This means User A might be able to get information about the other users and the data they are accessing. Because of this potential security risk, Sun has deprecated (as of Servlet API version 2.1) many of the methods which would have given you more generic

access to other session objects. While this is good from a security point of view, it also adds some limitations on session manipulation. For example, re-attaching a user to a pre-existing session or determining the number of live sessions has become more difficult in JSP 1.1.

When you don't have access to the session object.
Sometimes in your JSP page you will not have direct access to your session object. This is due to the timing and the way your JSP page is constructed. Implicit objects, including the session object, are defined in the _jspService method. This means two things. First, implicit objects are not instance level objects of the JSP servlet. Second, methods and other objects outside the service method don't have direct access to the implicit objects. So if you are using a method built within a declaration block, you must pass in your session object as a parameter if you want access to your session object.

Setting the default timeout period.
When deploying web applications your container class will have a method to control your default session timeout. In Tomcat your web.xml file has a statement which controls the default session timeout. The tag is as follows:

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

The timeout period is measured in minutes. The above example sets the default session timeout to 15 minutes.

Known problems
A strange behavior has been found, especially for sessions with a short timeout period. While your session has a firm timeout in theory, in practice your session might live longer than its timeout period. To date, why this behavior exists or if it is container specific is unclear. This has been observed in Tomcat 3.1, but the author is unaware if this bug has been tested for other containers.

Methods/Properties
The methods/properties described for the session object include the following: getAttribute, getAttributeNames, getCreationTime, getId, getLastAccessedTime, getMaxInactiveInterval, getValueNames, invalidate, isNew, putValue, removeAttribute, removeValue, setAttribute, setMaxInactiveInterval.

Method/Property	Object = getAttribute (String name)
Purpose	Retrieves data which has been stashed away in the session object.

Example	<%
	// first, let's set up an session variable
	String ls_test = "Data to store in session";
	session.setAttribute("Testing", ls_test);
	// now lets retrieve it
	Object lobj_getdata = session.getAttribute("Testing");
	/* Notice we have to type the object since the getAttribute returns an Object */
	String ls_getdata = (String)lobj_getdata;
	%>

Notes	• getAttribute will return a null value if the named property doesn't exist within the session object. • Primitive data types require one extra step of conversion from the primitive object container class to the primitive type. For example: <% Integer li_test = (Integer)session.getAttribute("MyData"); int li_primitive = li_test.intValue(); %> • If you have an invalidated session then this function throws an

	IllegalStateException.

Method/Property	Enumeration = getAttributeNames()
Purpose	Returns a list of the objects which are stored in the session object.

Example	<%@ page import="java.util.*" %>
	<%
	String ls_validate = "This is our original session";
	session.setAttribute("Validate", ls_validate);
	String ls_name = "";
	Enumeration enum_app = session.getAttributeNames();
	for(;enum_app.hasMoreElements();)
	 {
	  ls_name = "
	" + enum_app.nextElement().toString() + ":" ;
	  ls_name += session.getAttribute("Validate").toString();
	  out.print(ls_name);
	 }
	%>

Notes	• If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	long = getCreationTime()
Purpose	Returns when this session object was created. The number is in milliseconds since 1/1/1970 GMT.
Example	See example from getLastAccessedTime()
Notes	• If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	String = getId()
Purpose	Returns the unique ID for the session object.
Example	Our Session id is : <%= session.getId() %> In Tomcat the above example returns something like the following. Our Session id is : To1010mC6152742502460762At
Notes	• Each new session gets its own unique id number. • Each JSP container will return its own version of an unique id. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	long = getLastAccessedTime()
Purpose	Returns when the session object was last accessed by a user.

Example	<%
	 long ll_lasttime = session.getLastAccessedTime();
	 long ll_created = session.getCreationTime();
	 long ll_time_used = (ll_lasttime - ll_created) / 60000 ;
	%>
	
	You have been using this web site for <%= ll_time_used %> minutes.

Notes	• Similar to getCreationTime in that the returned number is in milliseconds since 1/1/1970 GMT. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	int = getMaxInactiveInterval()
Purpose	Returns how many seconds of inactivity must occur before the session timeout occurs.
Example	<% int li_inactive = session.getMaxInactiveInterval(); %>
Notes	• A negative return value means that the session will never timeout.

Method/Property	String[] = getValueNames()
Purpose	Gets a list of objects stored in the session object.
Example	See getAttributeNames instead.
Notes	• Deprecated as of Servlet version 2.2. You should use getAttributeNames() instead. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = invalidate()
Purpose	Invalidates the session and removes all objects stored within the session.
Example	<% session.invalidate(); %>
Notes	• Just because a session is invalidated doesn't mean the garbage collection will take place immediately. • Typically, you manually invalidate a session when the user submits a logoff request from your site. Since logging off from web site is an action the user must initiate, you should not count on log off requests from every user. A pseudo solution some sites implement is to perform a check to see if a user is relogging in. If so, it will invalidate the previous session. All these design issues revolve around the nature of Web applications and you should think about your needs early in your project. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	boolean = isNew()
Purpose	Returns "true" if a session object has been created but not accessed by client. Returns "false" if a session object has been used by the client.
Example	<% boolean lb_test = session.isNew() ; if (lb_test) out.print("we have a new session"); %>
Notes	• If you are using cookies to track your session, and if the client has disable cookies, then this function will always return true. This is because a new session would be created upon each access of the page. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = putValue(String name,Object value)
Purpose	Stores an object into the session object.
Example	See setAttribute instead.
Notes	• Deprecated as of Servlet version 2.2. You should use setAttribute instead. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = removeAttribute(String name)
Purpose	Removes the named object from the session object. If the object doesn't exist then nothing happens.

Example	<%
	// This adds a "Testing" object to the application object
	String ls_test = "Data to store in Application";
	session.setAttribute("Testing", ls_test);
	
	// Now this removes "Testing" from the application object
	session.removeAttribute("Testing");
	%>

Notes	• If the removed object implements HttpSessionBindingListener then the container calls HttpSessionBindingListener.valueUnbound. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = removeValue(String name)
Purpose	Removes the named object from the session object.
Example	See removeAttribute.
Notes	• Deprecated as of Servlet version 2.2. You should use removeAttribute instead. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = setAttribute(String name,Object value)
Purpose	Stores an object into the session object.
Example	<% String ls_validate = "This is our original session"; session.setAttribute("Validate", ls_validate); %>
Notes	• If the object being stored in the session object implements HttpSessionBindingListener, then the container calls HttpSessionBindingListener.valueBound. • If you have an invalidated session then this function throws an IllegalStateException.

Method/Property	void = setMaxInactiveInterval(int interval)
Purpose	Set the time duration (in seconds) of how long the session object is kept alive between client requests.
Example	<% session.setMaxInactiveInterval(5) ; %>
Notes	• If you have an invalidated session then this function throws an IllegalStateException.

Session Management Programming Example:
Here is an example of how to use URL rewriting.

- This example uses two pages called StartPage.jsp and NewPage.jsp
- To properly test this example disable cookie use in your browser's perferences.
- This function illustrates that your must encode all your URL requests or when you move to a new page you will lose your session.
- URL rewriting only works if your JSP container supports this feature. If your container doesn't support URL rewriting then you must implement your own manual version of URL rewriting. Tomcat 3.1 currently doesn't support URL rewriting.

StartPage.jsp

```
<%@ page session="true" %>
<%
// first we will put some test data into the bean to show
// we can get it back out again in another page!
String ls_validate = "This is our original session";
session.setAttribute("Validate", ls_validate);
// Now we will build a string containing the URL for the next page.
// We are encoding session information into the URL string, so as you
// move from page to page, your the Container will know what session
// has been used. Without this information, the container will
// start a new session for each page request.
String ls_session_id = session.getId();
String ls_encode_url = response.encodeURL("NewPage.jsp");
String ls_normal_url = "NewPage.jsp";
%>
<Html><Head></Head><Body>
Sample URL with encoded information: <%= ls_encode_url %> <BR>
Our Session id is : <%= ls_session_id %><BR><BR>
<ahref='<%= ls_encode_url %>'> Move to new page and keep session alive</a><BR><BR>
<ahref= '<%= ls_normal_url %>'> Move to new page without encoding the URL</a>
</Body></Html>
```

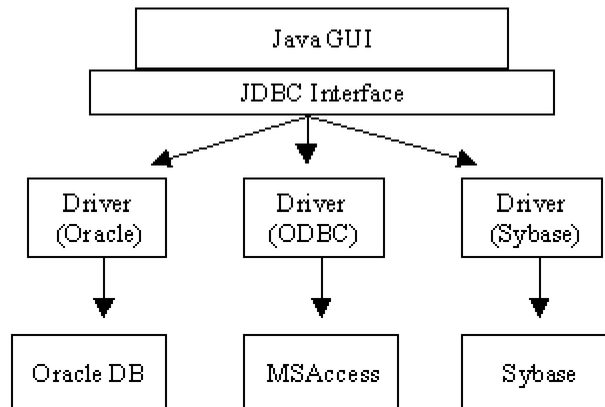
NewPage.jsp

```
<Html><Head></Head><Body>
<%@ page session="true" %>
The Current Session id is: <%= session.getId() %>

Checking the value stored in our validate attribute:
<%=session.getAttribute("Validate")%>
</Body></Html>
```

15.JSP - Beans - Databases

JDBC Overview
The "Write Once, Run Anywhere™" Java™ 2 Platform is a safe, flexible, and complete cross-platform solution for developing robust Java applications for the Internet and corporate intranets. The open and extensible Java Platform APIs are a set of essential interfaces that enable developers to build their Java applications and applets. The Java 2 Platform provides uniform, industry-standard, seamless connectivity and inter operability with enterprise information assets.
The JDBC™ API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases. The JDBC API provides a call-level API for SQL-based database access. JDBC technology allows developers using the Java programming language to exploit "Write Once, Run Anywhere" capabilities for applications that require access to enterprise data.
Figure:



What is Possible with JDBC?
The JDBC API, makes following things possible:
[a] Loading JDBC Driver
[b] Establishing the DB Connection
[c] Sending SQL Statements
[d] Processing the Result Sets

JDBC API Architecture
The application developers can access databases via the JDBC API using the vendor specific drivers as explained below. Basically there four types of driver categories are available currently in the IT industry.
[a]. JDBC-ODBC Bridge (Type 1)
[b]. A native-API partly Java technology-enabled driver (Type 2)
[c]. Pure Java Driver for Database Middleware (Type 3)
[d]. Direct-to-Database Pure Java Driver (Type 4)

The Advantages and Disadvantages of Driver Types
[a]. JDBC-ODBC Bridge (Type 1)
This driver type provides JDBC access via ODBC drivers. To use JDBC-ODBC Bridge , the ODBC binary code must loaded on each client machine, which uses JDBC-ODBC bridge. Sun provides a JDBC-ODBC Bridge driver. Provides JDBC access via one or more Open Database Connectivity (ODBC) drivers. ODBC, which predates JDBC, is widely used by developers to connect to databases in a non-Java environment.

Advantages: A good approach for training institutions for learning JDBC. This may be useful for corporates that already have ODBC drivers installed on each client machine — typically the case for Windows-based machines running productivity applications.

Disadvantages: Does not support all the features of Java JDBC API Specification. It is not suitable for for large-scale applications. Performance suffers because there's some overhead associated with translation work to go from JDBC to ODBC. The viable functionality is limited by the ODBC driver.

[b]. A native-API partly Java technology-enabled driver (Type 2)
This type of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Like JDBC-ODBC Bridge driver, this driver also requires some binary code to be loaded on each client machine, i.e each client machine need to be configured.
Advantages: The performance is better than JDBC-ODBC Bridge. The Type 2 driver contains compiled code that's optimized for the back-end database server's operating system.
Disadvantages: User needs to make sure the JDBC driver of the database vendor is loaded onto each client machine. On each client machine must have compiled code for every operating system that the application will run on. This is more suitable for intranets.

[c]. Pure Java Driver for Database Middleware (Type 3)
This type driver types translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.
Advantages: Does not need to configure each and every client machine. Gives you the better performance than Types 1 and Type 2. Can be used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them. This is only on server side configuration needed, if server side components are optimized for good performance then this is a good choice.
Disadvantages: Needs some database-specific code on the middleware server. If the middleware must run on different platforms, a Type 4 driver might be more effective.

[d]. Direct-to-Database Pure Java Driver (Type 4)
This type of drivers converts JDBC calls into the network protocol used directly by DBMSs, allowing a direct call from the client machine to the DBMS server. This driver converts JDBC calls into packets that are sent over the network in the proprietary format used by the specific database. Allows a direct call from the client machine to the database.
Advantages: Better performance than Types 1, Type 2, Type 3. Does not need to install any special software on client. On server side the needs installation and configuration.
Disadvantages: Not optimized for server operating system, so the driver can't take advantage of operating system features. (The driver is optimized for the database and can take advantage of the database vendor's functionality.) User needs a different driver for each different database.
The choice of JDBC driver type really depends upon the architecture of your existing systems and which drivers are available for your particular database. Sun maintains a list of the types and vendors of JDBC drivers at http://splash.javasoft.com/jdbc/jdbc.drivers.html.
JSP - JDBC : In this topic we are covering the following topics

- 1. Setting Up a Database
- 2. Creating Tables from JSP
- 3. Inserting Data in to Tables using JSP
- 4. Retrieving Values from Result Sets from JSP
- 5. Updating the Values using JSP

1. Setting Up a Database
Here we are assuming that the MS-Access database is already is installed in the local machine or the on the web server , where you are testing this JSP's.

Database Table Schema
List of Columns in Titles Table

Filed Name	Data Type	Filed Name	Data Type
Title_id	INTEGER	Price	FLOAT
Title_name	VARCHAR(50)	Quantity	INTEGER
Rating	VARCHAR(5)	Type_id	INTEGER
Category_id	INTEGER	Total	INTEGER

List of Columns in Categories Table

Filed Name	Data Type
category_id	INTEGER
category_name	VARCHAR(50)

List of Columns in Types Table

Filed Name	Data Type
Type_id	INTEGER
Type_name	VARCHAR(50)

2. Creating Tables from JSP

The following simple CreateJsp.jsp will illustrates how to create database tables in to MS-Access Database.

```
<HTML><HEAD><TITLE>My Titles Page</TITLE></HEAD>
<%@ page language="java" import="java.sql.*" %>
<BODY>
<H1>Creating Titles, Categories, Titles Tables </H1>
<TABLE BORDER="2" width="450"><TR>,TD>
<%
         try {
         // Load the Driver class file
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         // Make a connection to the ODBC datasource sample
         Connection con = DriverManager.getConnection("jdbc:odbc:sample", "", "");
         // Creating the statement Object
         Statement statement = con.createStatement();
         // Use the created statement to CREATE the database table Create Titles Table
         statement.executeUpdate("CREATE TABLE Titles " + "(title_id INTEGER, title_name
VARCHAR(50), " +
        "rating VARCHAR(5), price FLOAT, quantity INTEGER, " + "type_id INTEGER, category_id
INTEGER)");
        // Creating a database Categories table using statement
       statement.executeUpdate("CREATE TABLE Categories" + "(category_id INTEGER,
category_name ARCHAR(50))");
       // Creating a database Types table using statement
       statement.executeUpdate("CREATE TABLE Types" + "(type_id INTEGER, type_name
VARCHAR(50))");
%>
<H1>Successfully Created Titles, Categories, Titles Tables </H1>
<%
     }
    catch (SQLException sqle) {
      System.err.println(sqle.getMessage());
    }
    catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe.getMessage());
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
    }
    finally {
      try {
        if ( con != null ) {
          // Close the connection no matter what
          con.close();
        }
      }
      catch (SQLException sqle) {
        System.err.println(sqle.getMessage());
      }
    }
%>
<TD></TR></TABLE></BODY></HTML>
```

3. Inserting Data in to Tables using JSP

The following simple InsertJsp.jsp will illustrates how to insert data into database tables in to MS-Access Database.

```
<HTML><HEAD><TITLE>My Titles Page</TITLE></HEAD>
<%@ page language="java" import="java.sql.*" %>
<BODY>
<H1>Inserting data into Types, Titles, Categories, Tables </H1>
<TABLE BORDER="2" width="450"><TR><TD>
```

```
<%
        // Load the Driver class file
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       // Make a connection to the ODBC datasource sample
       connection con = DriverManager.getConnection("jdbc:odbc:sample", "", "");
       // Create the statement
       Statement statement = con.createStatement();
       // Use the created statement to INSERT DATA into the database tables.
       // Insert Data into the Types Table
       statement.executeUpdate("INSERT INTO Types " + "VALUES (0, 'VHS')");
       statement.executeUpdate("INSERT INTO Types " + "VALUES (1, 'DVD')");
       // Insert Data into the Categories Table
       statement.executeUpdate("INSERT INTO Categories " +"VALUES (0, 'Action Movies')");
       statement.executeUpdate("INSERT INTO Categories " +"VALUES (1, 'Comedy Movies')");
       statement.executeUpdate("INSERT INTO Categories " +"VALUES (2, 'SCI-FI')");
       // Insert Data into the Titles Table
       statement.executeUpdate("INSERT INTO Titles " +"VALUES (0, 'Riders of the Last Arc', " +"'PG',
18.95, 40, 0, 4)");
       statement.executeUpdate("INSERT INTO Titles " +"VALUES (1, 'BIG BUSINESS', " +"'R', 19.95, 12, 1,
0)");
       statement.executeUpdate("INSERT INTO Titles " +"VALUES (3, 'GodZilla', " + "'PG', 19.95, 9, 1, 1)");
 %>
<H1>Successfully Inserted Data into Titles, Categories, Titles Tables </H1>
<%
     }
    catch (SQLException sqle) {
      System.err.println(sqle.getMessage());
    }
    catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe.getMessage());
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
    }
    finally {
      try {
        if ( con != null ) {
          // Close the connection no matter what
          con.close();
        }
      }
      catch (SQLException sqle) {
        System.err.println(sqle.getMessage());
      }
    }
%>
<TD></TR></TABLE></BODY></HTML>
```

4. Retrieving Values from Result Set using JSP

The following simple RetrieveJsp.jsp will illustrates how to retrieve data from MS-Access Database tables .

```
<HTML><HEAD><TITLE>My Title Listings</Title></Head>
<%@ page language="java" import="java.sql.*" %>
<Body>
<H1>My Title Listings</h1>
<TABLE BORDER="1" WIDTH="400">
<TR>
     <TD><B>Title ID</B></TD><TD><B>Title Name</B></TD>
     <TD><B>Rating</B></TD><TD><B>Category</B></TD>
</TR>
<%
       // Load the Driver class file
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       // Make a connection to the ODBC datasource sample
       Connection con = DriverManager.getConnection("jdbc:odbc:sample", "", "");
       // Create the statement
       Statement statement = con.createStatement();
       // Creating the Result Set object
      ResultSet rSet = stmt.executeQuery("select * from Ttitles");
      // Iterating the Result Set
    if (rSet != null) {
      while (rSet.next()) {
      String id = rSet.getString("Title_id");
      String name = rSet.getString("Title_Name");
      String rating = rSet.getString("Rating");
      String category = rSet.getString("Category_id");
%>
<tr>
      <td><%= id %></td>
```

```
      <td><%= name %></td>
      <td><%= rating %> </td>
      <td><%= category %> </td>
</tr>
<%
      }
   }
    stmt.close();
    con.close();
%>
</TABLE></BODY></HTML>
```

5. Updating the Values using JSP

The following simple UpdateJsp.jsp will illustrates how to update MS-Access Database tables .

```
<HTML><HEAD><TITLE>My Titles Page</TITLE></HEAD>
<%@ page language="java" import="java.sql.*" %>
<BODY>
<H1>Updating Titles Table </H1>
<TABLE BORDER="2" width="450"><TR>,TD>
<%
         try {                    // Load the Driver class file
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         // Make a connection to the ODBC datasource sample
         Connection con = DriverManager.getConnection("jdbc:odbc:sample", "", "");
        // Create the statement
        Statement statement = con.createStatement();
        // Use the created statement to UPDATE DATA in the database tables.
        // Update the Quantity of "The Last Emperor"
       statement.executeUpdate("UPDATE Titles " +"SET quantity = 5 " +"WHERE title_name = 'BIG
BUSINESS'");
%>
<H1>Successfully Updated  Titles Table </H1>
<%
     }
    catch (SQLException sqle) {
      System.err.println(sqle.getMessage());
    }
    catch (ClassNotFoundException cnfe) {
      System.err.println(cnfe.getMessage());
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
    }
    finally {
      try {
        if ( con != null ) {
          // Close the connection no matter what
          con.close();
        }
      }
      catch (SQLException sqle) {
        System.err.println(sqle.getMessage());
      }
    }
%>
<TD></TR></TABLE></BODY></HTML>
```

16.JSP - Interactions

1. Invoking a JSP by URL
2. Invoking JSP from JSP
3. Calling a JSP from a Servlet
4. Calling Servlet from JSP

There are a different ways of that a JSP can use to interact with the Web environment. A JSP will use a JavaBean object to present dynamic content. However, a JSP can also invoke another JSP page by URL, by including another JSP or HTML page in the include directive, or by calling a servlet. This following sections describes the details of JSP interactions.

1. Invoking a JSP by URL (from browser)
A JSP can be invoked by browser URL address. The following code snippet gives you the details.
To invoke a JSP by URL, use the syntax:
http://servername/path/filename.jsp

For example, to invoke the Date.jsp, use this URL:
http://localhost:8080/jsphome/welcome .jsp
welcome.jsp

```
<HTML><HEAD><TITLE>Welcome to JSP </TITLE></HEAD>
<BODY>
<H1>Hell, How are you? </H1>
</BODY>
</HTML>
```

2. Invoking JSP from JSP
The JSP can also be invoked from <FORM> action within the HTML page other JSP page. For example the following code will execute the welcome.jsp.
To invoke a JSP file from another JSP file, you can, specify the URL of the second JSP file on the FORM ACTION attribute:

```
<FORM action="/called .jsp">
welcome.jsp
<HTML><HEAD><TITLE>Welcome to JSP </TITLE></HEAD>
<BODY>
<FORM ACTION="/called.jsp">
<INPUT TYPE=submit value="Call JSP" >
</FORM>
</BODY>
</HTML>
called.jsp
<HTML><HEAD><TITLE>Welcome to JSP </TITLE></HEAD>
<BODY>
<H1>Hell, How are you? </H1>
</BODY>
</HTML>
We can also invoke JSP frm <A HREF> tag of HTML.Specify the URL of the second JSP
file in an anchor tag HREF attribute:
<a href="JSP_URL"> reference-text </a>
welcome.jsp
<HTML><HEAD><TITLE>Welcome to JSP </TITLE></HEAD>
<BODY>
<FORM ACTION="/called.jsp">
<INPUT TYPE=submit value="Call JSP" >
</FORM>
<a href="called.jsp"> Call JSP Page </a>
</BODY>
</HTML>
```

We can also use to javax.servlet.htt.RequestDispatcher , forward() method to invoke second JSP file, which is similar to jsp:forward tag.
Example:
```
<HTML><BODY>
<H2> JSP to JSP </H2>
<HR>
<jsp:forward page="/called.jsp" />
<HR>
</BODY></HTML>
```

3. Calling a JSP from a Servlet
Here is code snippet to call .jsp from a servlet using HttpServletResponse.sendRedirect() method.
Example
```
import javax.servlet.http.*;
public class DisplayServlet extends HttpServlet {
     public void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
          // Redirect to the DateDisplay  JSP page
          res. sendRedirect("/called .jsp");
     }
}
```
Here is code snippet to call .jsp from a servlet using RequestDispatcher class.
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class TestDispatchServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws
IOException,ServletException  {
         ServletContext sc   =  getServletContext();
         RequestDispatcher rd;
         boolean b = true;
         if(b) {
          rd = sc.getRequestDispatcher("called.jsp");
         rd.forward(req,res);
      }
   }
}
```

4. Calling Servlet from JSP
You can invoke a servlet from a JSP either as an action on a form, or directly through the jsp:include or jsp:forward tags.
Form action
<FORM METHOD="POST" ACTION="/servlet/ServletName">
JSP include tag
You can include the output of a servlet in a JSP using the jsp.include tag:
```
<jsp:include page="/servlet/ServletName" />
Example
<HTML><BODY>
<H2> JSP to Servlet </H2>
<HR>
<jsp:include page="/servlet/MyServlet"  />
<HR>
<H2>End of servlet include</H2>
</HTML></BODY>
```
JSP forward tag
You can forward processing from a JSP to a servlet using the jsp.forward tag:
<jsp:forward page="/servlet/MyServlet" />
Example:
```
<HTML><BODY>
<H2> JSP to Servlet </H2>
<HR>
```

```
<jsp:forward page="/servlet/MyServlet" />
<HR>
<H2>End of servlet include</H2>
</HTML></BODY>
```

When you run this JSP, the output of the processing servlet replaces the output of the JSP. All output of the JSP will lost.

Building a JSP Form Handler :WEB FORMS AND FORM HANDLERS

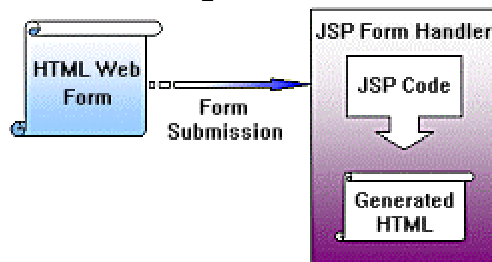
Web forms are the cornerstone of meaningful interaction with Web site visitors. From simple guest book pages to data entry forms on an intranet, you can use Web forms written in plain HTML to gather information from site visitors and bring that data back to the Web server for processing. Once that information has been passed to the server, you can do just about anything with it: validate it, store it in a database, write it to a file, or process it and give feedback to the user. You do all of this server-side processing with a form handler, which is a special Web page containing server-side code that you create to read and process the information submitted from the form.

Form Handler Basics

In JSP, form handlers are simply pages that contain JSP code and possibly some HTML code. Its purpose is to process information passed in from a Web form and then take some action based upon the processing. For example, a form handler might accept input from a site visitor, then attempt to store the information in a database. If the operation succeeds, the form handler could output a confirmation message, and if it fails it could output an error message. You can generally build a form handler in one of two ways:

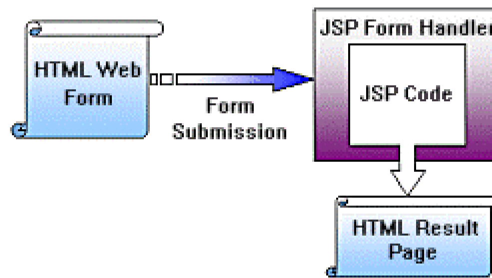
Handler plus feedback. With this approach, your form handler is also responsible for returning HTML and providing feedback or other information after it's done processing the form input. Refer to figure 1 for a simple interaction diagram. This approach is well suited for reporting the success or failure of some data entry operation by the site visitor. However, it does force you to mix your HTML output and presentation logic with your business logic, which in some cases can decrease maintainability.

Figure - JSP form handler generates its own HTML response



Handler with redirect. Using this approach, your form handler will only do server-side processing and will not output any HTML back to the user. Refer to figure 2 for a simple interaction diagram. Instead, when it's done processing, it will redirect to a new HTML or JSP page that will give some feedback to the site visitor. This approach can be useful in Web applications where, after processing a form, you need to send visitors to another part of the application instead of giving explicit feedback. Such an approach ultimately separates the business processes that may be implemented in your form handler from the presentation logic of the form results, which can make your applications more maintainable. In addition, if you find yourself reusing forms and their handlers frequently, such as you might with a standard login form, separating the form handler from the resulting output will help you reuse your code.

Figure - JSP form handler redirects to an HTML or JSP result page



Retrieving Information from a Form

Regardless of the server platform used, information is sent from a Web browser back to a Web server through HTTP using one of two methods: GET or POST. You've probably visited Web sites and seen a bunch of cryptic information tacked on after a question mark at the end of the URL. All of that cryptic information after the question mark gets sent back to the Web server using the GET method, and is stored in a special variable called QUERY_STRING which is accessible to Web server programs. The query string can contain several name/value pairs, such as color=blue. For example, the URL http://myserver.com/thing.html?color=blue&number=3 will arrive in QUERY_STRING as color=blue&number=3. Similarly, the POST method, which is typically used to send back information entered in a Web form, also sends data back to the server. However, it comes back to the server's standard input buffer instead of the QUERY_STRING variable.

In a traditional CGI application, you would have to read the server variables and parse out the name/value pairs yourself before you could use them. JSP has a built in object, called request, that parses out these values for you and makes them available through its getParameter method. For example, to retrieve the variable color, you would call request.getParameter("color").

EXAMPLE: BUILDING AN EMPLOYEE INFORMATION FORM

Now that you know how Web forms and form handlers work, let's put that knowledge to use by building a simple Web form and its form handler using HTML and JSP. Suppose that we are asked to build a simple form that new employees can use to enter their personal information, such as their Social Security number, phone number, and so on. Additionally, we must check to make sure the information is complete, and if so, store it in a database. We can build the form with standard HTML and the form handler using JSP, and the following sections will show how to build the code, step-by-step. You can view the complete code for this example at any time.

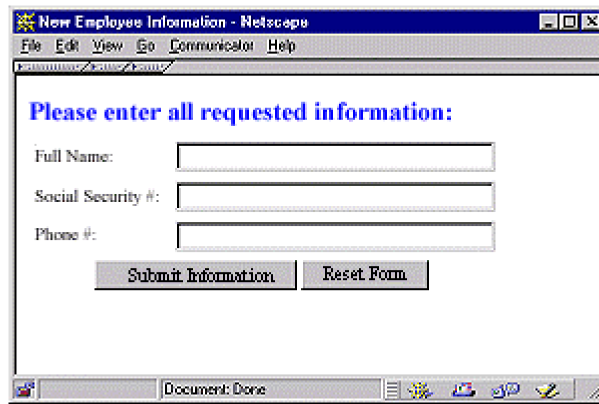
We'll design this mini-application in the following stages:

1. Build the Web form in HTML
2. Build the JSP form handler
    - 2a. Validate the user input sent from the form
    - 2b. If some fields were left blank, set the URL for redirection indicating an error
    - 2c. If no fields were left blank, store the data in the database using JDBC
    - 2d. Set the URL for redirection indicating success
    - 2e. Redirect to the URL specified in step 2b or 2d
3. Build the result page

Step 1: Build the Web Form

Building the Web form is a simple, straightforward process that requires no JSP coding. You simply create an HTML page, add a <FORM> tag, add input elements, and close the form tag. Figure 3 shows the form as it would appear in a Web browser. You can also review the HTML source code.

Figure - The Employee Information Web Form



Step 2: Build the Form Handler

Before diving into each of the steps involved in creating the form handler, you need to become familiar with the request and response objects that are available to you in JSP. Recall that earlier you learned that information sent from a Web form is passed to the Web server using the HTTP GET or POST method, and the request object takes care of parsing out the name/value pairs that correspond to each element in the form. We'll use the getParameter method of the request object to retrieve all of the form entries that a site visitor submits. Once we've processed the form, we'll want to redirect the user to a page indicating success, if they filled in all of the fields, or failure if they forgot to enter some information. To do this, we'll use the response object's sendRedirect method to redirect site visitors to a different page that displays results after they submit the form. You'll learn how to do all of this in the remaining steps.

Step 2a: Validate User Input

You might want to run validation checks on the input fields of a Web form for a number of reasons. You can check to make sure that site visitors enter valid phone numbers, complete e-mail addresses and so on, or you can simply check to make sure they didn't leave any fields blank, as we'll do for this example. You can run these checks using either client-side JavaScript or server-side JSP code. For the purposes of this example, we'll run the validation checks on the server using JSP.

First, we'll need to retrieve the form values from the request object.

```
// Retrieve and store the values submitted with the form on the page EmpForm.html
String fullName = request.getParameter("FullName");
String ssn = request.getParameter("SSN");
String phone = request.getParameter("Phone");
```

Next, we want to check to make sure that none of the three values are empty. For our purposes, this means that they should not be null, meaning the value was never passed over from the form, and they should not equal the empty string, "", meaning the user didn't type anything in one of the input boxes on the form. We could embed the code to do this right after the assignment statements shown above. However, with JSP you can define reusable methods on a page and call them from anywhere on the page. The Script Library in NetObjects ScriptBuilder 2.01 contains a reusable JSP method called isBlank that checks for values which are equal to the empty string or null. Since we'll need to call it three times -- once for each form input variable -- it seems like an ideal use of a reusable, page-level JSP method.

In general, you declare methods using `<SCRIPT RUNAT="server"></SCRIPT> tags, and call them from the many sets of scriptlet tags, <% %> your JSP pages may have.`

```
<SCRIPT RUNAT="server">
public boolean isBlank(String s) {
boolean isValid = true;
int i = 0;

/* if s is not null or empty string, loop through to see if all characters are spaces
*/
if (s != null && s != "") {
/* loop through the entire string testing for spaces */
```

```
for (i = 0; i < s.length(); i++) {
if (s.charAt(i) != ' ') {
isValid = false;
break;
}
} // end for loop
} // end if

return isValid;
} // end isBlank()
</SCRIPT>
```

Additionally, we can package up the validation code that makes use of the isBlank method into a method itself, called validateFields. Even though we won't be reusing validateFields, packaging up that code into a method will make the rest of the code easier to follow and understand.

```
private String validateFields(String fullName, String ssn, String phone) {
String errorStr = "";

// If fullName is blank, add it to errorStr
if (isBlank(fullName))
errorStr = "Full Name";

// If ssn is blank, add it to errorStr
if (isBlank(ssn)) {
// If there's already an error in errorStr, add a comma
if (!isBlank(errorStr))
errorStr += ", ";
errorStr += "Social Security Number";
}

// If phone is blank, add it to errorStr
if (isBlank(phone)) {
// If there's already an error in errorStr, add a comma
if (!isBlank(errorStr))
errorStr += ", ";
errorStr += "Phone Number";
}

return errorStr;
} // end validateFields()
```

Finally, we can make use of the validateFields method in the page by calling it and passing the three form variables we assigned in step 2a.1 above.

```
<%
// Validate the three form fields
err = validateFields(fullName, ssn, phone);
%>
```

Step 2b: If an error occurred, set the URL for redirection

At this point, we can check the err variable set by the validation code, and if it equals true we can take action to inform the user that an error has occurred. For this example, we could redirect to a page that will report the error by using the response.sendRedirect method. sendRedirect accepts one string argument which represents the URL of the page that should be sent back to the browser immediately. You can send any valid URL, which means that you can also include information in the query string portion of the URL. We'll take advantage of this to send the error message to our error page so that it can print the message out to the application user.

However, we'll make the actual call to response.sendRedirect later in the page. For now, we'll just assign the URL that should be called to a temporary variable and call sendRedirect after all of the error checking is done. Remember that we're going to redirect whether or not there is an error. The

only difference is what message will be displayed. In general, when the final action of a form handler is to redirect to some URL, you should conditionally assign the redirection URLs to temporary string variables and pass one to the call to response.sendRedirect as last command on the page . This will help you debug your code should a problem arise, and will help make the code more maintainable. If you need to change the URL of the result page, you'll only have to change it in one place, not two places.

```
<%
// Assign the messages that will be sent to the result page conditionally
if (err != "") {
result = "Error";
message = "You left the following fields blank: " + err + "<BR>";
message += "Please use the back button on your browser and try again.";
}
%>
```

Step 2c: Store the Data if No Error Occurred

If all of the data entry fields passed the validation checks, we're ready to store the data. For this example, we'll embed database access code right into the JSP page using the Java JDBC API. It's important to note that we could also make use of a JavaBean for storing the data. Accessing the database directly from the form handler will bind the form handler to the specifics of the database so that if the database structure changes, we'll have to change the form handler too. Instead, we could create a slightly more maintainable application by separating the database access logic out into a JavaBean, and just send the form data to the JavaBean for storage. This way, if the database structure changes, we only have to change the JavaBean code and the form handler doesn't have to know anything about the change. However, there are some cases where it's simpler and perfectly acceptable to access a database directly inside a JSP page. Because this example application is relatively small and simple, we'll perform the database access right in the page.

Accessing a database in JSP works just like accessing a database using JDBC from any other Java application, applet or servlet. We need to import the java.sql.* classes at the top of the form handler using an import directive of the form: <%@ import="java.sql.*" %>. Once the libraries are imported, we can use standard JDBC calls to connect to and insert information into the Employee database. Connecting to the database and performing the insert are two distinct operations, so to make the code more readable and easier to maintain in the future, we'll create a method in the page for each operation.

This example comes with a Microsoft Access 97 database that contains a table called Employee with three fields: FullName, SSN, Phone. The script will use the JDBC-ODBC bridge and the JDBC-ODBC driver that comes with the JDK 1.1 to connect to the database. It will then pass in the form parameters that correspond to the fields in the database and perform a SQL INSERT to add a new record to the database. If anything goes wrong, the handler will send the error to the result page, Result.jsp. The doConnect and doInsert methods which implement the database functionality are shown below.

```
public Connection doConnect(String connectString) {
Connection dbconn = null;

// Attempt to load the driver manager for the JDBC-ODBC bridge
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (java.lang.ClassNotFoundException ex) { }

// Attempt to get a connection to the ODBC data source indicated in the connectString
argument
try {
dbconn = DriverManager.getConnection(connectString, "", "");
} catch (SQLException ex) { }

return dbconn;
```

```
} // end doConnect()

public String doInsert(Connection dbconn,
String tableName,
String fullName,
String ssn,
String phone) {

String err = null;

// Attempt to insert a record into the table indicated by tableName
try {
Statement stmt = dbconn.createStatement();
stmt.executeUpdate("INSERT INTO " + tableName + " (FirstName, LastName, Address,
City) " +
"VALUES ('Joe','" + fullName + "','" + ssn + "','" + phone + "')");
} catch (SQLException ex) {
err = "Insert: " + ex.getMessage();
}

return err;
} // end doInsert()
```

We'll call these methods in the body of the JSP form handler, but we should only call it if Step 2b, the validation checks, completed without an error. So we must modify the if statement from Step 2b by adding an else statement that performs the database connection and insert operations. The example assumes that we have configured an ODBC data source name (DSN) called "SBEmp" that points to the Access database. According to the JDBC protocol, the database connection URL needed to connect to this database is jdbc:odbc:SBEmp. The new code required to connect to the SBEmp data source and insert a record into the Employee table is highlighted in **bold** below.

```
// Assign the messages that will be sent to the result page conditionally
if (err != "") {
result = "Error";
message = "You left the following fields blank: " + err + "<BR>";
message += "Please use the back button on your browser and try again.";
} else {
// Get a connection to the database indicated by connectString
dbconn = doConnect("jdbc:odbc:SBEmp");
// If we retrieved a database connection, do the insert
if (dbconn != null) {
err = doInsert(dbconn, "Employee", fullName, ssn, phone);
} else { // Otherwise, report the connection error
result = "Error";
message = "Error connecting to database: " + err;
}
}
```

Step 2d: Set the URL to a Confirmation Page : If all operations succeeded up to this point, we should send the site visitor to a page that tells him or her that the operation succeeded. Rather than create two separate pages -- one for reporting an error and one for reporting success -- we can just use one result page and send a different message in the query string of the URL. The result page will print whatever message we've sent it so it doesn't have to worry about whether or not the result was an error or success. Again, just as with the case where an error was encountered, for now we'll only set the proper URL and messages and in a temporary string variables and will perform the actual redirect at the end of the page.

To add the code for the success case, we need to add an if statement to the code we created in step 2c that checks the value of the err variable that's returned by the doInsert method. If the doInsert method failed, err will contain an error message. If it's empty, then we'll know the error succeeded. So let's change the code from Step 2c one more time to add a new error message if the insert operation failed, and a success message if the insert succeded. The new code is highlighted below in **bold** text.

```
// Assign the messages that will be sent to the result page conditionally
if (err != "") {
result = "Error";
message = "You left the following fields blank: " + err + "<BR>";
message += "Please use the back button on your browser and try again.";
} else {
// Get a connection to the database indicated by connectString
dbconn = doConnect(connectString);
// If we retrieved a database connection, do the insert
if (dbconn != null) {
err = doInsert(dbconn, tableName, fullName, ssn, phone);
// If the insert failed, report the insert error
if (err != null) {
result = "Error";
message = "Error inserting record into database: " + err;
} else { // Otherwise, the insert succeeded so report success!
result = "Success";
message = "Your information has been stored!";
}
} else { // Otherwise, report the connection error
result = "Error";
message = "Error connecting to database at " + connectString + ": " + err;
}
}
```

Step 2e: Redirect the Site Visitor

At this point, we'll have finished the validation checks and generated the messages we'll need to pass on to the result page. All that remains is to redirect the user to the result page by calling response.sendRedirect.

```
<%
// Redirect and send the result and message to the result page.
response.sendRedirect("Result.jsp?result=" + result + "&message=" +
java.net.URLEncoder.encode(message));
%>
```

Notice that we're invoking a class method of the java.net.URLEncoder class. This is necessary because only certain characters are legal within a URL string. Characters such as spaces, which our messages contain, and some punctuation are illegal and must be encoded in order to be passed as part of a valid URL. The encode method of the URLEncoder class will encode the message variable so that it can be passed in the URL. We don't have to worry about decoding it when it gets to Result.jsp because the request object will automatically decode the variables for us.

Step 3: Building the Result Page

Finally, we'll build the result page. This page is very simple, and writes out some information with static HTML and some with dynamically generated HTML through JSP. All that we need to do is write a small section of code to write out the response sent by the form handler by way of the URL's query string.

```
<HTML><HEAD>
<!-- This title is written out dynamically using JSP -->
<TITLE><%= request.getParameter("result") %></TITLE></HEAD>

<BODY BGCOLOR="white">
<BR>
<!-- The string below is written out dynamically using JSP -->

<CENTER><B>
<%
// Make sure that the parameter was passed before writing it out
if (request.getParameter("message") != null && request.getParameter("message") != "")
out.println(request.getParameter("message"));
```

```
else // if the parameter wasn't passed, print a default error message
out.println("An error has occurred on the server. Please contact your system
administrator.");
%>
</B></CENTER>
</BODY></HTML>
```

In Result.jsp, we're dynamically generating the title of the page and some of the page content. Notice the special scriptlet tag, <%=, that we've used to output the title string. You can output the result of a valid Java expression between the special output scriptlet tags, <%= %>. These tags instruct the server to simply write out the result of the expression as a string, and they automatically handle the conversion of most of literal data types such as int, float, etc. as well.