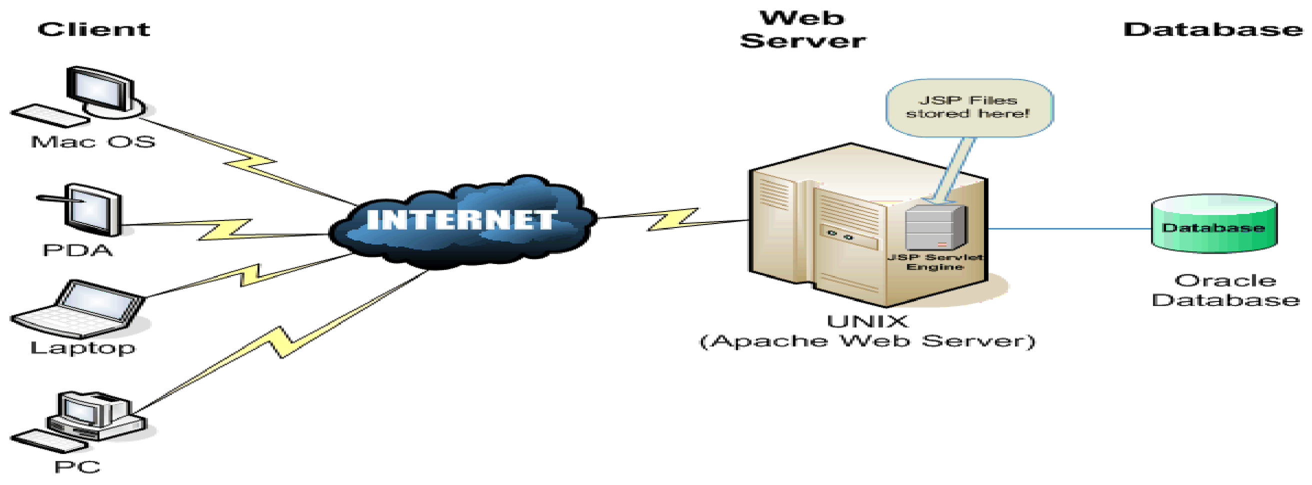


# JSP Tutorials

Kaji

## JSP Introduction

In this JSP tutorial, you will learn about JSP, usage of JSP, process of development, independency of layers and simplification of process.

## Usage of JSP:

- JSP is widely used for developing dynamic web sites.
- JSP is used for creating database driven web applications because it provides superior server side scripting support.

Some of the reasons for the popularity of JSP are

### Simplifies the process of development:

It allows programmers to insert the Java code directly into the JSP file, making the development process easier. Although JSP files are HTML files, they use special tags containing the Java source code which provides a dynamic ability.

### Portability:

The Java feature of 'write once, run anywhere' is applicable to JSP. JSP is platform independent, making it portable across any platform and therefore, mulit-platform. It is possible for the programmer to take a JSP file and move it to another platform, JSP Servlet engine or web server.

### Because of Efficiency :

As soon as the request is received, the JSP pages gets loaded into the web servers' memory. The following transactions are then carried out within a minimal time period, handling JSP pages with efficiency.

### Reusability:

JSP allow component reuse by using JavaBeans and EJBs.

### Robust:

JSP offers a robust platform for web development

## Independency of Layers:

There is a clear separation between presentation and implementation layers. The HTML on the web browser of the client is displayed as a presentation layer. The JSP on the server is displayed in the implementation layer. Programmers who want to work with HTML can work independently without the work of the Java developers', working in the implementation layer, being affected. Java Server Pages, or JSP, is the way to separate the look of the web page from the corresponding content.

Integration of JSP with Other source like JDBC, Servlet and so on:

The combination of JDBC and JSP works very well. JSP is used for generating dynamic web pages. It is essential that data in these systems be maintained efficiently and securely. The programmer must have a good database and database connectivity. This is achieved by JDBC through its excellent database connectivity in heterogeneous database system. This database system is used for the integration of heterogeneous database management systems presenting a single query interface system.

## Simplification of Process:

The JSP language has a simple development and maintenance process. A JSP file that has the extension .jsp is converted into a servlet .java which is dynamically compiled, loaded and executed. Only when there is a change in a JSP file, the Conversion, compilation, and loading process is then performed.

## What is JSP?

JSP is java server pages, which is dynamic web pages and used in to build dynamic websites. JSP provides developer to do less hard work with better work output. JSP works with http protocols; it provides tag and tools to make JSP pages. To run jsp, we need web server it can be tomcat provided by apache, it can be jRun, jBoss(Redhat), weblogic (BEA) , or websphere(IBM)

## JSP Running platform

JSP can be run on any platform. Most of web server supports all platforms like linux, Microsoft windows, sun soloris, Macintosh, FreeBSD. JSP takes, advantage of java platform independent

## .What is a JSP File?

JSP is dynamic file which like as HTML file. Html file is static and can not get data from database or fly data. JSP can be interactive pages and communicate with database and more controllable by programmer. JSP is executed on web server with help of java compiler. JSP can contain text, images, hyperlinks and all can be in HTML page. It is saved by extension of .jsp.

## ASP, JSP, HTML Difference

ASP active server pages is given by Microsoft technologies, it needs IIS (Internet Information Services) to run ASP. Mostly it is run on Windows operating systems. If need to run on others operating system rather than windows, needs extra tools.
JSP is almost same but provided by sun Microsystems. JSP is first compiled and then make SERVLET. All ASP, JSP take request from browser and process this request by server (web server) and response back to browser. When web server get request, it compile this code (JSP or ASP) and output from this code

response back to particular request. HTML having static content and request doesn't go to server for process. It process almost on clients side on client's browser (Internet Explorer or firefox browser).

### What can do with JSP?

JSP is used for dynamic programming, mean we can communicate with database, insert, update, select, alter, and delete records from database with JSP. Logic building with JSP, to print data, e.g., if we want to print good morning, good afternoon, good evening at particular slot of time, then we need to make a logic of time 12 am to 12 pm is good morning, 12pm after 4 pm is good after noon, and 4pm to 12 am is good evening. This can be done by if conditions to check current time and if time comes in any slot then print this message.
Interact with user, with the help of forms like enquiry form, or submitting user's data to server.
Provide session to individual users. Increase security and privacy of user. JSP code can not view at client or browser. Web server response browser with plain HTML page; send no JSP code, so user can not get code or see JSP code.
JSP code can be put inside HTML code. Both code work together, JSP code be embedded with scriptlet <% inside this jsp code %>

## JSP Tags

Another important syntax element of JSP are tags. JSP tags do not use **<%**, but just the **<** character. A JSP tag is somewhat like an HTML tag. JSP tags can have a "start tag", a "tag body" and an "end tag". The start and end tag both use the tag name, enclosed in < and > characters. The end starts with a / character after the <character. The tag names have an embedded colon character : in them, the part before the colon describes the type of the tag. For instance:

```
<some:tag>
body
</some:tag>
```

If the tag does not require a body, the start and end can be conveniently merged together, as

```
<some:tag/>
```

Here by closing the start tag with a /> instead of > character, we are ending the tag immediately, and without a body. (This syntax convention is the the same as XML.)

Tags can be of two types: loaded from an external tag library, or predefined tags. Predefined tags start with **jsp:** characters. For instance, jsp:include is a predefined tag that is used to include other pages.

We have already seen the include directive. jsp:include is similar. But instead of loading the text of the included file in the original file, it actually calls the included target at run-time (the way a browser would call the included target. In practice, this is actually a simulated request rather than a full round-trip between the browser and the server). Following is an example of jsp:include usage

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<jsp:include page="hello.jsp"/>
</BODY>
</HTML>
```

## JSP Sessions

On a typical web site, a visitor might visit several pages and perform several interactions.

If you are programming the site, it is very helpful to be able to associate some data with each visitor. For this purpose, "session"s can be used in JSP.

A session is an object associated with a visitor. Data can be put in the session and retrieved from it, much like a Hashtable. A different set of data is kept for each visitor to the site.

Here is a set of pages that put a user's name in the session, and display it elsewhere. Try out installing and using these.

First we have a form, let us call it GetName.html

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

The target of the form is "SaveName.jsp", which saves the user's name in the session. Note the variable "session". This is another variable that is normally made available in JSPs, just like out and request variables. (In the @page directive, you can indicate that you do not need sessions, in which case the "session" variable will not be made available.)

```
<%
  String name = request.getParameter( "username" );
  session.setAttribute( "theName", name );
%>
<HTML>
<BODY>
<A HREF="NextPage.jsp">Continue</A>
</BODY>
</HTML>
```

The SaveName.jsp saves the user's name in the session, and puts a link to another page, NextPage.jsp.

NextPage.jsp shows how to retrieve the saved name.

```
<HTML>
<BODY>
Hello, <%= session.getAttribute( "theName" ) %>
</BODY>
</HTML>
```

## JSP Directives

We have been fully qualifying the java.util.Date in the examples in the below display current time and date program . Perhaps you wondered why we don't just import java.util.*;

It is possible to use "import" statements in JSPs, but the syntax is a little different from normal Java. Try the following example:

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%
   System.out.println( "Evaluating date now" );
   Date date = new Date();
%>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

The first line in the above example is called a "directive". A JSP "directive" starts with <%@ characters.

This one is a "page directive". The page directive can contain the list of all imported packages. To import more than one item, separate the package names by commas, e.g.

```
<%@ page import="java.util.*,java.text.*" %>
```

There are a number of JSP directives, besides the page directive. Besides the page directives, the other most useful directives are include and taglib. We will be covering taglib separately.

The include directive is used to physically include the contents of another file. The included file can be HTML or JSP or anything else -- the result is as if the original JSP file actually contained the included text. To see this directive in action, create a new JSP

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<%@ include file="hello.jsp" %>
</BODY>
</HTML>
```

View this JSP in your browser, and you will see your original hello.jsp get included in the new JSP.

## JSP Declarations

The JSP you write turns into a class definition. All the scriptlets you write are placed inside a single method of this class.
You can also add variable and method declarations to this class. You can then use these variables and methods from your scriptlets and expressions.

To add a declaration, you must use the **<%!** and **%>** sequences to enclose your declarations, as shown below.

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%!
   Date theDate = new Date();
   Date getDate()
   {
      System.out.println( "In getDate() method" );
  return theDate;
   }
%>
Hello!  The time is now <%= getDate() %>
</BODY>
</HTML>
```

The example has been created a little contrived, to show variable and method declarations.

Here we are declaring a Date variable theDate, and the method getDate.  Both of these are available now in our scriptlets and expressions.

## Jsp Programs

### First program of adding two numbers in JSP

```
<%@ page language="java"%>
<html>
<head>
<title>Add number program in JSP</title>
</head>

<body>
<%
 int a=5;
 int b=2;

 int result=a+b;

 out.print("Additon of a and b :"+result);
%>
</body>
</html>
```

**Second program of subtraction two numbers in JSP**

```
<%@ page language="java"%>
<html>
<head>
<title>Subtraction numbers program in JSP</title>
</head>

<body>
<%
 int a=5;
 int b=2;

 int result=a-b;

 out.print("Subtraction of a and b :"+result);
%>
</body>
</html>
```

**Third program of print even numbers in JSP**

```
<%@ page language="java"%>
<html>
<head>
<title>Even number program in JSP</title>
</head>

<body>
<%
 for(int i=0;i<=10;i++)
 {
   if((i%2)==0)
   {
     out.print("Even number  :"+i);
     out.print("<br>");
   }
 }
%>
</body>
</html>
```

**Forth program of print odd numbers in JSP**

```
<%@ page language="java"%>
<html>
<head>
<title>Odd number program in JSP</title>
</head>

<body>
<%
 for(int i=0;i<=10;i++)
 {
```

```
if((i%2)!=0)
   {
     out.print("Odd number  :"+i);
     out.print("<br>");
   }
 }
%>
</body>
</html>
```

*****Session program**

**page1.jsp**

```
-------------------------------------------------
<%@ page language="java" %>
<html>
<head>
<title>page1</title>
</head>
<body>
<form method="post" action="savesession.jsp">
Name: </b><input type="text" name="name" value=""><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

The above code of the JSP page1.jsp that takes the input from user.

**savesession.jsp**

```
-------------------------------------------------

<%@ page language="java" %>
<%
String name=request.getParameter("name");
session.setAttribute("name",name);
%>

<html>
<head>
<title>Name Saved</title>
</head>
<body>
<p><a href="showsession.jsp">Next Page to view the session value</a><p>
</body>
</html>
```

The savesession.jsp save the name into session.

**showsession.jsp**

```
--------------------------------------------------
<%@ page language="java" %>
<%
String name=(String) session.getAttribute("name");
%>
<html>
<head>
<title>Show value</title>
</head>
<body>
Name is: <%=username%><p>

</body>
</html>
```

# Session Management in JSP

As we know that the Http protocol is a stateless protocol, that means that it can't persist the data. Http treats each request as a new request so every time you will send a request you will be considered as a new user. It is not reliable when we are doing any type of transactions or any other related work where persistence of the information is necessary. To remove these obstacles we use session management. In session management whenever a request comes for any resource, a unique token is generated by the server and transmitted to the client by the response object and stored on the client machine as a cookie. We can also say that the process of managing the state of a web based client is through the use of session IDs. Session IDs are used to uniquely identify a client browser, while the server side processes are used to associate the session ID with a level of access. Thus, once a client has successfully authenticated to the web applicatiion, the session ID can be used as a stored authentication voucher so that the client does not have to retype their login information with each page request. Now whenever a request goes from this client again the ID or token will also be passed through the request object so that the server can understand from where the request is coming. Session management can be achieved by using the following thing.

1. **Cookies:** cookies are small bits of textual information that a web server sends to a browser and that browsers returns the cookie when it visits the same site again. In cookie the information is stored in the form of a name, value pair. By default the cookie is generated. If the user doesn't want to use cookies then it can disable them.

2. **URL rewriting:** In URL rewriting we append some extra information on the end of each URL that identifies the session. This URL rewriting can be used where a cookie is disabled. It is a good practice to use URL rewriting. In this session ID information is embedded in the URL, which is recieved by theapplication through Http GET requests when the client clicks on the links embedded with a page.

3. **Hidden form fields:** In hidden form fields the html entry will be like this : <input type ="hidden" name = "name" value="">. This means that when you submit the form, the specified name and value will be get included in get or post method. In this session ID information would be embedded within the form as a hidden field and submitted with the Http POST command.

In JSP we have been provided a implicit object session so we don't need to create a object of session explicitly as we do in Servlets. In Jsp the session is by default true. The session is defined inside the directive <%@ page session = "true/false" %>. If we don't declare it inside the jsp page then session will  be available to the page, as it is default by true.
For the convenience to understand the concept of session management we have made one program.

**The code of the program is given below:**

index.html

```
<html>
        <head>
                <title>Welcome to the first program of jsp</title>
        </head>
        <body>
        <form method = "post" action = "FirstPageOfSession.jsp">
  <font size = 6>Enter your name<input type = "text" name = "name"></font><br><br>
  <font size = 6>Enter your password<input type="password" name = "pwd" >
                </font><br><br>
                <input type = "submit" name = "submit" value = "submit" >
        </form>
        </body>
</html>
```

FirstPageOfSession.jsp

```
<%
    String name = request.getParameter("name");
        String password = request.getParameter("pwd");
        if(name.equals("Williams") && password.equals("abcde"))
        {
                session.setAttribute("username",name);
                response.sendRedirect("NextPageAfterFirst.jsp");
        }
        else
        {
                response.sendRedirect("SessionManagement.html");
        }
        %>
```

NextPageAfterFirst.jsp

```
<html>
        <head>
                <title>Welcome in In the program of URL rewriting</title>
        </head>
        <body>
<font size = 6>Hello</font> <%= session.getAttribute("username") %>
        </body>
</html>
```

SessionManagement.html

```
<html>
        <head>
                <title>Welcome in SessionManagement </title>
        </head>
        <body>
<font size = 6>Hello</font> <%= session.getAttribute("username") %>
        </body>
</html>
```

## Program to display current Time and Date

```
 <%@page contentType="text/html" import="java.util.*" %>
<!--  http://www.java-samples.com/jsp  -->
<html>
<body>
<p>&nbsp;</p>
<div align="center">
<center>
<table border="0" cellpadding="0" cellspacing="0" width="460" bgcolor="#EEFFCA">
<tr>
<td width="100%"><font size="6" color="#008000">&nbsp;Date Example</font></td>
</tr>
<tr>
<td width="100%"><b>&nbsp;Current Date and time is:&nbsp; <font color="#FF0000">
<%= new java.util.Date() %>
</font></b></td>
</tr>
</table>
</center>
</div>
</body>
</html>
```

## JSP Life Cycle

JSP’s life cycle can be grouped into following phases.

**1. JSP Page Translation:**

A java servlet file is generated from the JSP source file. This is the first step in its tedious multiple phase life cycle. In the translation phase, the container validates the syntactic correctness of the JSP pages and tag files. The container interprets the standard directives and actions, and the custom actions referencing tag libraries used in the page.

**2. JSP Page Compilation:**

The generated java servlet file is compiled into a java servlet class.

Note: The translation of a JSP source page into its implementation class can happen at any time between initial deployment of the JSP page into the JSP container and the receipt and processing of a client request for the target JSP page.

**3. Class Loading:**

The java servlet class that was compiled from the JSP source is loaded into the container.

**4. Execution phase:**

In the execution phase the container manages one or more instances of this class in response to requests and other events.
The interface JspPage contains jspInit() and jspDestroy(). The JSP specification has provided a special interface HttpJspPage for JSP pages serving HTTP requests and this interface contains _jspService().

**5. Initialization:**

jspInit() method is called immediately after the instance was created. It is called only once during JSP life cycle.

**6. _jspService() execution:**

This method is called for every request of this JSP during its life cycle. This is where it serves the purpose of creation. Oops! it has to pass through all the above steps to reach this phase. It passes the request and the response objects. _jspService() cannot be overridden.

**7. jspDestroy() execution:**

This method is called when this JSP is destroyed. With this call the servlet serves its purpose and submits itself to heaven (garbage collection). This is the end of jsp life cycle.

jspInit(), _jspService() and jspDestroy() are called the life cycle methods of the JSP.

## Scope in JSP

**<%@ page contentType="text/html; charset=iso-8859-1" language="java" import="java.sql.*" errorPage="errorpage.jsp" scope="page|request|session|application" %>**

One of the most powerful features of JSP is that a JSP page can access, create, and modify data objects on the server. You can then make these objects visible to JSP pages. When an object is created, it defines or defaults to a given scope. The container creates some of these objects, and the JSP designer creates others.

The *scope* of an object describes how widely it's available and who has access to it. For example, if an object is defined to have page scope, then it's available only for the duration of the current request on that page before being destroyed by the container. In this case, only the current page has access to this data, and no one else can read it. At the other end of the scale, if an object has application scope, then any page may use the data because it lasts for the duration of the application, which means until the container is switched off.

### Page Scope

Objects with *page scope* are accessible only within the page in which they're created. The data is valid only during the processing of the current response; once the response is sent back to the browser, the data is no longer valid. If the request is forwarded to another page or the browser makes another request as a result of a redirect, the data is also lost.

### Request Scope

Objects with *request scope* are accessible from pages processing the same request in which they were created. Once the container has processed the request, the data is released. Even if the request is forwarded to another page, the data is still available though not if a redirect is required.

### Session Scope

Objects with *session scope* are accessible from pages processing requests that are in the same session as the one in which they were created. A *session* is the time users spend using the application, which ends when they close their browser, when they go to another Web site, or when the application designer wants (after a logout, for instance). So, for example, when users log in, their username could be stored in the session and displayed on every page they access. This data lasts until they leave the Web site or log out.

### Application Scope

Objects with *application scope* are accessible from JSP pages that reside in the same application. This creates a global object that's available to all pages.

Application scope uses a single namespace, which means all your pages should be careful not to duplicate the names of application scope objects or change the values when they're likely to be read by another page (this is called *thread safety*). Application scope variables are typically created and populated when an application starts and then used as read-only for the rest of the application.

# JSP Implicit Objects

Implicit objects in jsp are the objects that are created by the container automatically and the container makes them available to the developers, the developer do not need to create them explicitly. Since these objects are created automatically by the container and are accessed using standard variables; hence, they are called implicit objects. The implicit objects are parsed by the container and inserted into the generated servlet code. They are available only within the jspService method and not in any declaration. Implicit objects are used for different purposes. Our own methods (user defined methods) can't access them as they are local to the service method and are created at the conversion time of a jsp into a servlet. But we can pass them to our own method if we wish to use them locally in those functions.

There are nine implicit objects. Here is the list of all the implicit objects:

| Object | Class |
|---|---|
| application | javax.servlet.ServletContext |
| config | javax.servlet.ServletConfig |
| exception | java.lang.Throwable |
| out | javax.servlet.jsp.JspWriter |
| page | java.lang.Object |
| PageContext | javax.servlet.jsp.PageContext |
| request | javax.servlet.ServletRequest |
| response | javax.servlet.ServletResponse |
| session | javax.servlet.http.HttpSession |

- **Application:** These objects has an application scope. These objects are available at the widest context level, that allows to share the same information between the JSP page's servlet and any Web components with in the same application.
- **Config:** These object has a page scope and is an instance of javax.servlet.ServletConfig class. Config object allows to pass the initialization data to a JSP page's servlet. Parameters of this objects can be set in the deployment descriptor (web.xml) inside the element <jsp-file>. The method getInitParameter() is used to access the initialization parameters.
- **Exception:** This object has a page scope and is an instance of java.lang.Throwable class. This object allows the exception data to be accessed only by designated JSP "error pages."
- **Out:** This object allows us to access the servlet's output stream and has a page scope. Out object is an instance of javax.servlet.jsp

  .JspWriter class. It provides the output stream that enable access to the servlet's output stream.
- **Page:** This object has a page scope and is an instance of the JSP page's servlet class that processes the current request. Page object represents the current page that is used to call the methods defined by the translated servlet class. First type cast the servlet before accessing any method of the servlet through the page.
- **Pagecontext:** PageContext has a page scope. Pagecontext is the context for the JSP page

## Overview

Implicit objects will be automatically instantiated under specific variable names. Furthermore,each object must adhere to a specific Java class or interface definition.

We have already met one of these objects already - the out object in which we used the println() method to add text to the output stream.

| Object | Class or Interface | Description |
|---|---|---|
| page | jsp.HttpJspPage | Page's servlet instance |
| config | ServletConfig | Servlet configuration information |
| pageContext | jsp.pageContext | Provides access to all the namespaces associated with a JSP page and access to several page attributes |
| request | http.HttpServletRequest | Data included with the HTTP Request |
| response | http.HttpServletResponse | HTTP Response data, e.g. cookies |
| out | jsp.JspWriter | Output stream for page context |
| session | http.HttpSession | User specific session data |
| application | ServletContext | Data shared by all application pages |

The first three are rarely used. The application, session and request implicit objects have the additional ability to hold arbitrary values. By setting and getting attribute values these objects are able to share information between several JSP pages.

# JSP General Overview

## Introduction to JavaServer Pages

JavaServer Pages(TM) is a technology specified by Sun Microsystems as a convenient way of generating dynamic content in pages that are output by a Web application (an application running on a Web server).

This technology, which is closely coupled with Java servlet technology, allows you to include Java code snippets and calls to external Java components within the HTML code (or other markup code, such as XML) of your Web pages. JavaServer Pages (JSP) technology works nicely as a front-end for business logic and dynamic functionality in JavaBeans and Enterprise JavaBeans (EJBs).

JSP code is distinct from other Web scripting code, such as JavaScript, in a Web page. Anything that you can include in a normal HTML page can be included in a JSP page as well.

In a typical scenario for a database application, a JSP page will call a component such as a JavaBean or Enterprise JavaBean, and the bean will directly or indirectly access the database, generally through JDBC or perhaps SQLJ.

A JSP page is translated into a Java servlet before being executed (typically on demand, but sometimes in advance), and it processes HTTP requests and generates responses similarly to any other servlet. JSP technology offers a more convenient way to code the servlet.

Furthermore, JSP pages are fully interoperable with servlets--JSP pages can include output from a servlet or forward to a servlet, and servlets can include output from a JSP page or forward to a JSP page.

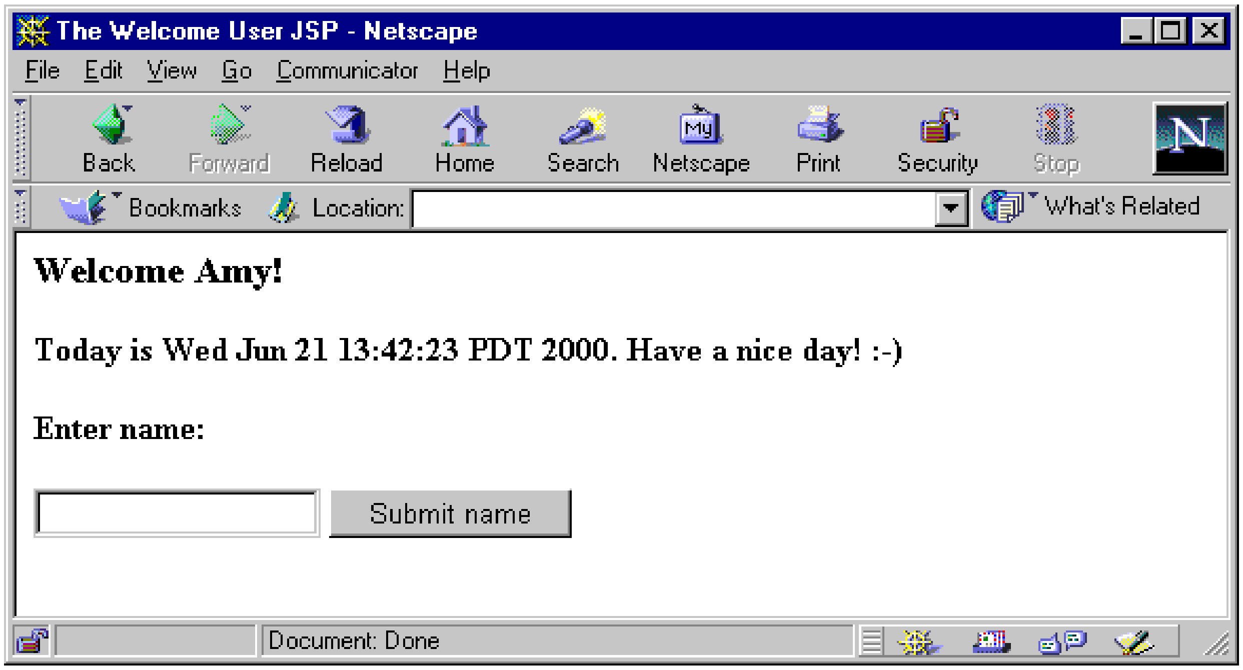
### What a JSP Page Looks Like

Here is an example of a simple JSP page. (For an explanation of JSP syntax elements used here, see "Overview of JSP Syntax Elements".)

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

In a JSP page, Java elements are set off by tags such as <% and %>, as in the preceding example. In this example, Java snippets get the user name from an HTTP request object, print the user name, and get the current date.

This JSP page will produce the following output if the user inputs the name "Amy":


The Welcome User JSP - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: What's Related
Submit name
Document: Done

Welcome Amy!

Today is Wed Jun 21 13:42:23 PDT 2000. Have a nice day! :-)

Enter name:

Text description of the illustration welcoamy.gif

## Convenience of JSP Coding Versus Servlet Coding

Combining Java code and Java calls into an HTML page is more convenient than using straight Java code in a servlet. JSP syntax gives you a shortcut for coding dynamic Web pages, typically requiring much less code than Java servlet syntax. Following is an example contrasting servlet code and JSP code.

## Servlet Code

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

public class Hello extends HttpServlet
{
  public void doGet(HttpServletRequest rq, HttpServletResponse rsp)
  {
    rsp.setContentType("text/html");
    try {
      PrintWriter out = rsp.getWriter();
      out.println("<HTML>");
      out.println("<HEAD><TITLE>Welcome</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<H3>Welcome!</H3>");
      out.println("<P>Today is "+new java.util.Date()+".</P>");
      out.println("</BODY>");
      out.println("</HTML>");
    } catch (IOException ioe)
    {
      // (error processing)
    }
  }
}
```

### JSP Code

```
<HTML>
<HEAD><TITLE>Welcome</TITLE></HEAD>
<BODY>
<H3>Welcome!</H3>
<P>Today is <%= new java.util.Date() %>.</P>
</BODY>
</HTML>
```

Note how much simpler JSP syntax is. Among other things, it saves Java overhead such as package imports and try...catch blocks.

Additionally, the JSP translator automatically handles a significant amount of servlet coding overhead for you in the .java file that it outputs, such as directly or indirectly implementing the standard javax.servlet.jsp.HttpJspPage interface and adding code to acquire an HTTP session.

Also note that because the HTML of a JSP page is not embedded within Java print statements as is the case in servlet code, you can use HTML authoring tools to create JSP pages.

## Separation of Business Logic from Page Presentation: Calling JavaBeans

JSP technology allows separating the development efforts between the HTML code that determines static page presentation, and the Java code that processes business logic and presents dynamic content. It therefore becomes much easier to split maintenance responsibilities between presentation and layout specialists who may be proficient in HTML but not Java, and code specialists who may be proficient in Java but not HTML.

In a typical JSP page, most Java code and business logic will *not* be within snippets embedded in the JSP page--instead, it will be in JavaBeans or Enterprise JavaBeans that are invoked from the JSP page.

JSP technology offers the following syntax for defining and creating an instance of a JavaBeans class:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This example creates an instance, pageBean, of the mybeans.NameBean class (the scope parameter will be explained later in this chapter).

Later in the page, you can use this bean instance, as in the following example:

```
Hello <%= pageBean.getNewName() %> !
```

(This prints "Hello Julie !", for example, if the name "Julie" is in the newName attribute of pageBean, which might occur through user input.)

The separation of business logic from page presentation allows convenient division of responsibilities between the Java expert who is responsible for the business logic and dynamic content--this developer owns and maintains the code for the NameBean class--and the HTML expert who is responsible for the static presentation and layout of the Web page that the application user sees--this developer owns and maintains the code in the .jsp file for this JSP page.

Tags used with JavaBeans--useBean to declare the JavaBean instance and getProperty and setProperty to access bean properties--are further discussed in "JSP Actions and the <jsp: > Tag Set".

## JSP Pages and Alternative Markup Languages

JavaServer Pages technology is typically used for dynamic HTML output, but the Sun Microsystems *JavaServer Pages Specification, Version 1.1* also supports additional types of structured, text-based document output. A JSP translator does not process text outside of JSP elements, so any text that is appropriate for Web pages in general is typically appropriate for a JSP page as well.

A JSP page takes information from an HTTP request and accesses information from a data server (such as through a SQL database query). It combines and processes this information and incorporates it as appropriate into an HTTP response with dynamic content. The content can be formatted as HTML, DHTML, XHTML, or XML, for example.

For information about XML support, see "XML-Alternative Syntax". You can also refer to the *Oracle9iAS Containers for J2EE JSP Tag Libraries and Utilities Reference* for information about the JML transform tag.

## JSP Execution

This section provides a top-level look at how a JSP is run, including on-demand translation (the first time a JSP page is run), the role of the *JSP container* and the servlet container, and error processing.

A JSP container is an entity that translates, executes, and processes JSP pages and delivers requests to them.

The exact make-up of a JSP container varies from implementation to implementation, but it will consist of a servlet or collection of servlets. The JSP container, therefore, is executed by a servlet container.

A JSP container may be incorporated into a Web server if the Web server is written in Java, or the container may be otherwise associated with and used by the Web server.

## JSP Pages and On-Demand Translation

Presuming the typical on-demand translation scenario, a JSP page is usually executed as follows:

1. The user requests the JSP page through a URL ending with a .jsp file name.
2. Upon noting the .jsp file name extension in the URL, the servlet container of the Web server invokes the JSP container.
3. The JSP container locates the JSP page and translates it if this is the first time it has been requested. Translation includes producing servlet code in a .java file and then compiling the .java file to produce a servlet .class file.

   The servlet class generated by the JSP translator subclasses a class (provided by the JSP container) that implements the javax.servlet.jsp.HttpJspPage interface. The servlet class is referred to as the *page implementation class*. This document will refer to instances of page implementation classes as *JSP page instances*.

   Translating a JSP page into a servlet automatically incorporates standard servlet programming overhead into the generated servlet code, such as implementing the HttpJspPage interface and generating code for its service method.

4. The JSP container triggers instantiation and execution of the page implementation class.

The servlet (JSP page instance) will then process the HTTP request, generate an HTTP response, and pass the response back to the client.

**Note:**

The preceding steps are loosely described for purposes of this discussion. As mentioned earlier, each vendor decides how to implement its JSP container, but it will consist of a servlet or collection of servlets. For example, there may be a front-end servlet that locates the JSP page, a translation servlet that handles translation and compilation, and a wrapper servlet class that is subclassed by each page implementation class (because a translated page is not a pure servlet and cannot be run directly by the servlet container). A servlet container is required to run each of these components.

## Requesting a JSP Page

A JSP page can be requested either directly--through a URL--or indirectly--through another Web page or servlet.

### Directly Request a JSP Page

As with a servlet or HTML page, the end-user can request a JSP page directly by URL. For example, assume you have a HelloWorld JSP page that is located under the myapp application root directory in the Web server, as follows:

myapp/dir1/HelloWorld.jsp

If it uses port 8080 of the Web server, you can request it with the following URL:

http://*hostname*:8080/myapp/dir1/HelloWorld.jsp

(The application root directory is specified in the servlet context of the application.)

The first time the end-user requests HelloWorld.jsp, the JSP container triggers both translation and execution of the page. With subsequent requests, the JSP container triggers page execution only; the translation step is no longer necessary.

### Indirectly Requesting a JSP Page

JSP pages, like servlets, can also be executed indirectly--linked from a regular HTML page or referenced from another JSP page or from a servlet.

When invoking one JSP page from a JSP statement in another JSP page, the path can be either relative to the application root--known as *context-relative* or *application-relative*--or relative to the invoking page--known as *page-relative*. An application-relative path starts with "/"; a page-relative path does not.

Be aware that, typically, neither of these paths is the same path as used in a URL or HTML link. Continuing the example in the preceding section, the path in an HTML link is the same as in the direct URL request, as follows:

```
<a href="/myapp/dir1/HelloWorld.jsp" /a>
```

The application-relative path in a JSP statement is:

```
<jsp:include page="/dir1/HelloWorld.jsp" flush="true" />
```

The page-relative path to invoke HelloWorld.jsp from a JSP page in the same directory is:

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

("JSP Actions and the <jsp: > Tag Set" discusses the jsp:include and jsp:forward statements.)

## Overview of JSP Syntax Elements

You have seen a simple example of JSP syntax in "What a JSP Page Looks Like"; now here is a top-level list of syntax categories and topics:

- *directives*--These convey information regarding the JSP page as a whole.
- *scripting elements*--These are Java coding elements such as declarations, expressions, scriptlets, and comments.
- *objects* and *scopes*--JSP objects can be created either explicitly or implicitly and are accessible within a given scope, such as from anywhere in the JSP page or the session.
- *actions*--These create objects or affect the output stream in the JSP response (or both).

This section introduces each category, including basic syntax and a few examples. For more information, see the Sun Microsystems *JavaServer Pages Specification, Version 1.1*.

---

**Notes:**

There are XML-compatible alternatives to the syntax for JSP directives, declarations, expressions, and scriptlets. See "XML-Alternative Syntax".

---

## Directives

Directives provide instruction to the JSP container regarding the entire JSP page. This information is used in translating or executing the page. The basic syntax is as follows:

```
<%@ directive attribute1="value1" attribute2="value2"... %>
```

The JSP 1.1 specification supports the following directives:

- page--Use this directive to specify any of a number of page-dependent attributes, such as the scripting language to use, a class to extend, a package to import, an error page to use, or the JSP page output buffer size. For example:

  ```
  <%@ page language="java" import="packages.mypackage" errorPage="boof.jsp" %>
  ```

or, to set the JSP page output buffer size to 20kb (the default is 8kb):

```
<%@ page buffer="20kb" %>
```

or, to unbuffer the page:

```
<%@ page buffer="none" %>
```

---

**Notes:**

- A JSP page using an error page must be buffered. Forwarding to an error page clears the buffer (not outputting it to the browser).
- For the Oracle JSP container, java is the default language setting. It is good programming practice to set it explicitly, however.

---

- include--Use this directive to specify a resource that contains text or code to be inserted into the JSP page when it is translated. Specify the path of the resource relative to the URL specification of the JSP page.

  Example:

  ```
  <%@ include file="/jsp/userinfopage.jsp" %>
  ```

  The `include` directive can specify either a page-relative or context-relative location. (See "Requesting a JSP Page" for related discussion.)

---

**Notes:**

- The `include` directive, referred to as a "static include", is comparable in nature to the `jsp:include` action discussed later in this chapter, but takes effect at JSP translation time instead of request time. See "Static Includes Versus Dynamic Includes".
- The `include` directive can be used only between pages in the same servlet context.

---

- `taglib`--Use this directive to specify a library of custom JSP tags that will be used in the JSP page. Vendors can extend JSP functionality with their own sets of tags. This directive indicates the location of a *tag library description* file and a prefix to distinguish use of tags from that library.

Example:

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

Later in the page, use the `oracust` prefix whenever you want to use one of the tags in the library (presume this library includes a tag `dbaseAccess`):

```
<oracust:dbaseAccess ... >
...
</oracust:dbaseAccess>
```

As you can see, this example uses XML-style start-tag and end-tag syntax.

JSP tag libraries and tag library description files are introduced later in this chapter, in "Tag Libraries", and discussed in detail in Chapter 7, "JSP Tag Libraries".

## Scripting Elements

JSP scripting elements include the following categories of Java code snippets that can appear in a JSP page:

- *declarations*--These are statements declaring methods or member variables that will be used in the JSP page.

  A JSP declaration uses standard Java syntax within the `<%!...%>` declaration tags to declare a member variable or method. This will result in a corresponding declaration in the generated servlet code. For example:

  ```
  <%! double f1=0.0; %>
  ```

  This example declares a member variable, `f1`. In the servlet class code generated by the JSP translator, `f1` will be declared at the class top level.

  **Note:**

  Method variables, as opposed to member variables, are declared within JSP scriptlets as described below. See "Method Variable Declarations Versus Member Variable Declarations" for more information.

- *expressions*--These are Java expressions that are evaluated, converted into string values as appropriate, and displayed where they are encountered on the page.

  A JSP expression does *not* end in a semi-colon, and is contained within `<%=...%>` tags.

**Example:**

```
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
```

---

**Note:**

A JSP expression in a request-time attribute, such as in a `jsp:setProperty` statement, need not be converted to a string value.

---

- *scriptlets*--These are portions of Java code intermixed within the markup language of the page.

  A scriptlet, or code fragment, may consist of anything from a partial line to multiple lines of Java code. You can use them within the HTML code of a JSP page to set up conditional branches or a loop, for example.

  A JSP scriptlet is contained within `<%...%>` scriptlet tags, using normal Java syntax.

  **Example 1:**

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
<% } else { %>
   Hello <%= pageBean.getNewName() %>.
<% } %>
```

  Three one-line JSP scriptlets are intermixed with two lines of HTML (one of which includes a JSP expression, which does *not* require a semi-colon). Note that JSP syntax allows HTML code to be the code that is conditionally executed within the `if` and `else` branches (inside the Java brackets set out in the scriptlets).

  The preceding example assumes the use of a JavaBean instance, `pageBean`.

  **Example 2:**

```
<% if (pageBean.getNewName().equals("")) { %>
   I don't know you.
   <% empmgr.unknownemployee();
} else { %>
   Hello <%= pageBean.getNewName() %>.
   <% empmgr.knownemployee();
} %>
```

  This example adds more Java code to the scriptlets. It assumes the use of a JavaBean instance, `pageBean`, and assumes that some object, `empmgr`, was previously instantiated and has methods to execute appropriate functionality for a known employee or an unknown employee.

---

**Note:**

Use a JSP scriptlet to declare method variables, as opposed to member variables, as in the following example:

```
<% double f2=0.0; %>
```

This scriptlet declares a method variable, `f2`. In the servlet class code generated by the JSP translator, `f2` will be declared as a variable within the service method of the servlet.

Member variables are declared in JSP declarations as described above.

For a comparative discussion, see "Method Variable Declarations Versus Member Variable Declarations".

---

- ***comments***--These are developer comments embedded within the JSP code, similar to comments embedded within any Java code.

  Comments are contained within `<%--...--%>` tags.

  **Example:**

  ```
  <%-- Execute the following branch if no user name is entered. --%>
  ```

## JSP Objects and Scopes

In this document, the term *JSP object* refers to a Java class instance declared within or accessible to a JSP page. JSP objects can be either:

- ***explicit***--Explicit objects are declared and created within the code of your JSP page, accessible to that page and other pages according to the `scope` setting you choose.

or:

- ***implicit***--Implicit objects are created by the underlying JSP mechanism and accessible to Java scriptlets or expressions in JSP pages according to the inherent `scope` setting of the particular object type.

Scopes are discussed below, in "Object Scopes".

### Explicit Objects

Explicit objects are typically JavaBean instances declared and created in `jsp:useBean` action statements. The `jsp:useBean` statement and other action statements are described in "JSP Actions and the <jsp: > Tag Set", but an example is also shown here:

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

This statement defines an instance, `pageBean`, of the `NameBean` class that is in the `mybeans` package. The scope parameter is discussed in "Object Scopes" below.

You can also create objects within Java scriptlets or declarations, just as you would create Java class instances in any Java program.

## Object Scopes

Objects in a JSP page, whether explicit or implicit, are accessible within a particular *scope*. In the case of explicit objects, such as a JavaBean instance created in a `jsp:useBean` action statement, you can explicitly set the scope with the following syntax (as in the example in the preceding section, "Explicit Objects"):

`scope="`*`scopevalue`*`"`

**There are four possible scopes:**

- `scope="page"`--The object is accessible only from within the JSP page where it was created.

  Note that when the user refreshes the page while executing a JSP page, new instances will be created of all page-scope objects.

- `scope="request"`--The object is accessible from any JSP page servicing the same HTTP request that is serviced by the JSP page that created the object.
- `scope="session"`--The object is accessible from any JSP page sharing the same HTTP session as the JSP page that created the object.
- `scope="application"`--The object is accessible from any JSP page used in the same Web application (within any single Java virtual machine) as the JSP page that created the object.

## Implicit Objects

JSP technology makes available to any JSP page a set of *implicit objects*. These are Java class instances that are created automatically by the JSP mechanism and that allow interaction with the underlying servlet environment.

- `page`

  This is an instance of the JSP page implementation class that was created when the page was translated, and that implements the interface `javax.servlet.jsp.HttpJspPage`; `page` is synonymous with `this` within a JSP page.

- `request`

  This represents an HTTP request and is an instance of a class that implements the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface.

- `response`

  This represents an HTTP response and is an instance of a class that implements the `javax.servlet.http.HttpServletResponse` interface, which extends the `javax.servlet.ServletResponse` interface.

The `response` and `request` objects for a particular request are associated with each other.

- `pageContext`

  This represents the *page context* of a JSP page, which is provided for storage and access of all `page` scope objects of a JSP page instance. A `pageContext` object is an instance of the `javax.servlet.jsp.PageContext` class.

  The `pageContext` object has `page` scope, making it accessible only to the JSP page instance with which it is associated.

- `session`

  This represents an HTTP session and is an instance of the `javax.servlet.http.HttpSession` class.

- `application`

  This represents the servlet context for the Web application and is an instance of the `javax.servlet.ServletContext` class.

  The `application` object is accessible from any JSP page instance running as part of any instance of the application within a single JVM. (The programmer should be aware of the server architecture regarding use of JVMs.)

- `out`

  This is an object that is used to write content to the output stream of a JSP page instance. It is an instance of the `javax.servlet.jsp.JspWriter` class, which extends the `java.io.Writer` class.

  The `out` object is associated with the `response` object for a particular request.

- `config`

  This represents the servlet configuration for a JSP page and is an instance of a class that implements the `javax.servlet.ServletConfig` interface. Generally speaking, servlet containers use `ServletConfig` instances to provide information to servlets during initialization. Part of this information is the appropriate `ServletContext` instance.

- `exception` (JSP error pages only)

  This implicit object applies only to JSP error pages--these are pages to which processing is forwarded when an exception is thrown from another JSP page; they must have the `page` directive `isErrorPage` attribute set to `true`.

  The implicit `exception` object is a `java.lang.Exception` instance that represents the uncaught exception that was thrown from another JSP page and that resulted in the current error page being invoked.

  The `exception` object is accessible only from the JSP error page instance to which processing was forwarded when the exception was encountered.

  For an example of JSP error processing and use of the `exception` object, see "JSP Runtime Error Processing".

### Using an Implicit Object

Any of the implicit objects discussed in the preceding section might be useful. The following example uses the `request` object to retrieve and display the value of the `username` parameter from the HTTP request:

```
<H3> Welcome <%= request.getParameter("username") %> ! <H3>
```

## JSP Actions and the <jsp: > Tag Set

JSP action elements result in some sort of action occurring while the JSP page is being executed, such as instantiating a Java object and making it available to the page. Such actions may include the following:

- creating a JavaBean instance and accessing its properties
- forwarding execution to another HTML page, JSP page, or servlet
- including an external resource in the JSP page

Action elements use a set of standard JSP tags that begin with "`<jsp:`" syntax. Although the tags described earlier in this chapter that begin with "`<%`" syntax are sufficient to code a JSP page, the "`<jsp:`" tags provide additional functionality and convenience.

Action elements also use syntax similar to that of XML statements, with similar "begin" and "end" tags such as in the following example:

```
<jsp:sampletag attr1="value1" attr2="value2" ... attrN="valueN">
...body...
</jsp:sampletag>
```

or, where there is no body, the action statement is terminated with an empty tag:

```
<jsp:sampletag attr1="value1", ..., attrN="valueN" />
```

The JSP specification includes the following standard action tags, which are introduced and briefly discussed here:

- `jsp:useBean`

  The `jsp:useBean` action creates an instance of a specified JavaBean class, gives the instance a specified name, and defines the scope within which it is accessible (such as from anywhere within the current JSP page instance).

  **Example:**

  ```
  <jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
  ```

  This example creates a page-scoped instance `pageBean` of the `mybeans.NameBean` class. This instance is accessible only from the JSP page instance that creates it.

- `jsp:setProperty`

  The `jsp:setProperty` action sets one or more bean properties. The bean must have been previously specified in a `useBean` action. You can directly specify a value for a specified property, or take the value for a specified property from an associated HTTP request parameter, or iterate through a series of properties and values from the HTTP request parameters.

  The following example sets the `user` property of the `pageBean` instance (defined in the preceding `useBean` example) to a value of "Smith":

  ```
  <jsp:setProperty name="pageBean" property="user" value="Smith" />
  ```

  The following example sets the `user` property of the `pageBean` instance according to the value set for a parameter called `username` in the HTTP request:

  ```
  <jsp:setProperty name="pageBean" property="user" param="username" />
  ```

  If the bean property and request parameter have the same name (`user`), you can simply set the property as follows:

  ```
  <jsp:setProperty name="pageBean" property="user" />
  ```

  The following example results in iteration over the HTTP request parameters, matching bean property names with request parameter names and setting bean property values according to the corresponding request parameter values:

  ```
  <jsp:setProperty name="pageBean" property="*" />
  ```

  **Important:**

  For `property="*"`, the JSP 1.1 specification does not stipulate the order in which properties are set. If order matters, and if you want to ensure that your JSP page is portable, you should use a separate `jsp:setProperty` statement for each property.

  Also, if you use separate `jsp:setProperty` statements, then the Oracle JSP translator can generate the corresponding `setXXX()` methods directly. In this case, introspection only occurs during translation. There will be no need to introspect the bean during runtime, which would be somewhat more costly.

- `jsp:getProperty`

  The `jsp:getProperty` action reads a bean property value, converts it to a Java string, and places the string value into the implicit `out` object so that it can be displayed as output. The bean must have been previously specified in a `jsp:useBean` action. For the string conversion, primitive types are converted directly and object types are converted using the `toString()` method specified in the `java.lang.Object` class.

The following example puts the value of the `user` property of the `pageBean` bean into the `out` object:

```
<jsp:getProperty name="pageBean" property="user" />
```

- `jsp:param`

  You can use the `jsp:param` action in conjunction with `jsp:include`, `jsp:forward`, or `jsp:plugin` actions (described below).

  For `jsp:forward` and `jsp:include` statements, a `jsp:param` action optionally provides key/value pairs for parameter values in the HTTP request object. New parameters and values specified with this action are added to the request object, with new values taking precedence over old.

  The following example sets the request object parameter `username` to a value of `Smith`:

  ```
  <jsp:param name="username" value="Smith" />
  ```

---

**Note:**

The `jsp:param` tag is not supported for `jsp:include` or `jsp:forward` in the JSP 1.0 specification.

---

- `jsp:include`

  The `jsp:include` action inserts additional static or dynamic resources into the page at request time as the page is displayed. Specify the resource with a relative URL (either page-relative or application-relative).

  As of the Sun Microsystems *JavaServer Pages Specification, Version 1.1*, you must set `flush` to `true`, which results in the buffer being flushed to the browser when a `jsp:include` action is executed. (The `flush` attribute is mandatory, but a setting of `false` is currently invalid.)

  You can also have an action body with `jsp:param` settings, as shown in the second example.

  **Examples:**

  ```
  <jsp:include page="/templates/userinfopage.jsp" flush="true" />
  ```

or:

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" >
   <jsp:param name="username" value="Smith" />
   <jsp:param name="userempno" value="9876" />
</jsp:include>
```

Note that the following syntax would work as an alternative to the preceding:

```
<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876"
flush="true" />
```

---

**Notes:**

- The `jsp:include` action, known as a "dynamic include", is similar in nature to the `include` directive discussed earlier in this chapter, but takes effect at request time instead of translation time. See "Static Includes Versus Dynamic Includes".
- The `jsp:include` action can be used only between pages in the same servlet context.

---

- `jsp:forward`

  The `jsp:forward` action effectively terminates execution of the current page, discards its output, and dispatches a new page--either an HTML page, a JSP page, or a servlet.

  The JSP page must be buffered to use a `jsp:forward` action; you cannot set `buffer="none"`. The action will clear the buffer, not outputting contents to the browser.

  As with `jsp:include`, you can also have an action body with `jsp:param` settings, as shown in the second example.

  **Examples:**

  ```
  <jsp:forward page="/templates/userinfopage.jsp" />
  ```

  or:

  ```
  <jsp:forward page="/templates/userinfopage.jsp" >
     <jsp:param name="username" value="Smith" />
     <jsp:param name="userempno" value="9876" />
  </jsp:forward>
  ```

---

**Notes:**

- The difference between the `jsp:forward` examples here and the `jsp:include` examples earlier is that the `jsp:include` examples insert `userinfopage.jsp` within the output of the current page; the `jsp:forward` examples stop executing the current page and display `userinfopage.jsp` instead.
- The `jsp:forward` action can be used only between pages in the same servlet context.
- The `jsp:forward` action results in the original `request` object being forwarded to the target page. As an alternative, if you do not want the `request` object forwarded, you can use the `sendRedirect(String)` method specified in the standard `javax.servlet.http.HttpServletResponse` interface. This sends a temporary redirect response to the client using the specified redirect-location URL. You can specify a relative URL; the servlet container will convert the relative URL to an absolute URL.

---

- `jsp:plugin`

The `jsp:plugin` action results in the execution of a specified applet or JavaBean in the client browser, preceded by a download of Java plugin software if necessary.

Specify configuration information, such as the applet to run and the codebase, using `jsp:plugin` attributes. The JSP container might provide a default URL for the download, but you can also specify attribute `nspluginurl="`*`url`*`"` (for a Netscape browser) or `iepluginurl="`*`url`*`"` (for an Internet Explorer browser).

Use nested `jsp:param` actions within `<jsp:params>` and `</jsp:params>` start and end tags to specify parameters to the applet or JavaBean. (Note that these `jsp:params` start and end tags are *not* necessary when using `jsp:param` in a `jsp:include` or `jsp:forward` action.)

Use `<jsp:fallback>` and `</jsp:fallback>` start and end tags to delimit alternative text to execute if the plugin cannot run.

The following example, from the *Sun Microsystems JavaServer Pages Specification, Version 1.1*, shows the use of an applet plugin:

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
   <jsp:params>
      <jsp:param name="molecule" value="molecules/benzene.mol" />
   </jsp:params>
   <jsp:fallback>
      <p> Unable to start the plugin. </p>
   </jsp:fallback>
</jsp:plugin>
```

Many additional parameters--such as `ARCHIVE`, `HEIGHT`, `NAME`, `TITLE`, and `WIDTH`--are allowed in the `jsp:plugin` action statement as well. Use of these parameters is according to the general HTML specification.

## Tag Libraries

In addition to the standard JSP tags discussed previously in this section, the JSP 1.1 specification lets vendors define their own *tag libraries* and also lets vendors implement a framework allowing customers to define their own tag libraries.

A tag library defines a collection of custom tags and can be thought of as a JSP sub-language. Developers can use tag libraries directly, in manually coding a JSP page, but they might also be used automatically by Java development tools. A standard tag library must be portable between different JSP container implementations.

Import a tag library into a JSP page using the `taglib` directive, introduced in "Directives".

Key concepts of standard JavaServer Pages support for JSP tag libraries include the following topics:

- **tag handlers**

  A *tag handler* describes the semantics of the action that results from use of a custom tag. A tag handler is an instance of a Java class that implements either the `Tag` or `BodyTag` interface (depending on whether the tag uses a body between a start tag and an end tag) in the standard `javax.servlet.jsp.tagext` package.

- **scripting variables**

  Custom tag actions can create server-side objects available for use by the tag itself or by other scripting elements such as scriptlets. This is accomplished by creating or updating *scripting variables*.

  Details regarding scripting variables that a custom tag defines must be specified in a subclass of the standard `javax.servlet.jsp.tagext.TagExtraInfo` abstract class. This document refers to such a subclass as a *tag-extra-info class*. The JSP container uses instances of these classes during translation.

- **tag library description files**

  A *tag library description* (TLD) file is an XML document that contains information about a tag library and about individual tags of the library. The file name of a TLD has the `.tld` extension.

  A JSP container uses the TLD file in determining what action to take when it encounters a tag from the library.

- **use of `web.xml` for tag libraries**

  The Sun Microsystems *Java Servlet Specification, Version 2.2* describes a standard deployment descriptor for servlets--the `web.xml` file. JSP applications can use this file in specifying the location of a JSP tag library description file.

  For JSP tag libraries, the `web.xml` file can include a `taglib` element and two subelements: `taglib-uri` and `taglib-location`.

# JSP Directives

**Directive tag:**

The *directive* tag gives special information about the page to JSP Engine. This changes the way JSP Engine processes the page. Using directive tag, user can import packages, define error handling pages or session information of JSP page.

General notation of *directive* tag is as follows:

There are three types of *directive* tag.

- page
- Include
- Tag Lib

Syntax and usage of *directive* tag

**Page directive:**

General syntax for the *page* directive is

<%@ page optional attribute ... %>

There are many optional attributes available for page directive. Each of these attributes are used to give special processing information to the JSP Engine changing the way the JSP Engine processes the page. Some of the optional attributes available for page directive are:

- language
- extends
- import
- session
- buffer
- autoFlush
- isThreadSafe
- info
- errorPage
- IsErrorPage
- contentType

Syntax and usage of some of the optional attributes available for *page* directive discussed below.

**language:**

This attribute is used to denote the language used by a file. Language denotes the scripting language used in scriptlets, declarations, and expressions in the JSP page and any included files.

Syntax of language attribute available for page directive is

```
<%@ page language = "lang" %>
```

In the above statement page and language are keywords and one places whatever language the file uses inside " ".

For example if one wants to mention the language as java which is generally mentioned for all it is done as shown below:

```
<%@ page language = "java" %>
```

**extends:**

This is used to signify the fully qualified name of the Super class of the Java class used by the JSP engine for the translated Servlet.

Syntax of extends attribute available for page directive is

```
<%@ page extends = "package.class"%>
```

In the above statement page and extends are keywords.

**import:**

The import attribute is used to import all the classes in a java package into the current JSP page. With this facility, the JSP page can use other java classes.

Syntax of import attribute available for page directive is

```
<%@ page import = "java.util.*" %>
```

In the above statement page and import are keywords.

If there are many Java packages that the JSP page wants to import, the programmer can use import more than once in a JSP page or separate the Java packages with commas, as shown below:

```
<%@ page import="{package.class | package.*}, ..." %>
```

**session:**

The session attribute, when set to true, sets the page to make use of sessions.

**NOTE:** by default, the session attribute value is set to true therefore, all JSP pages have session data available. If the user sets the session attribute to false, it should be performed in this section. When the session attribution is set to false, the user cannot use the session object, or a <jsp:useBean> element with scope=session in the JSP page which, if used, would give error.

Syntax of session attribute available for page directive is

```
<%@ page session="true|false" %>
```

In the above statement page and session are keywords. And either true or false value can be setted and by default the value is true.

**buffer:**

If a programmer likes to control the use of buffered output for a JSP page then the buffer attribute can be made use of for achieving this.

Syntax of buffer attribute available for page directive is

```
<%@ page buffer = "none|8kb|sizekb" %>
```

In the above statement page and buffer are keywords. The size of buffer size is mentioned in kilobytes. This is used by the out object to handle output sent from the compiled JSP page to the client web browser. The default value is 8kb. If a user specifies a buffer size then the output is buffered with at least the size mentioned by the user.

For example one can specify as:

```
<%@ page buffer = "none" %>
```

## JSP Page Directive

In this JSP tutorial, you will learn about JSP page directive, attributes of the page directive along with syntax, examples and explanations.

**autoFlush:**

autoFlush attribute is used to specify whether or not to automatically flush out the output buffer when it is full. Syntax of autoFlush attribute available for page directive is written as:

```
<%@ page autoFlush = "true|false" %>
```

In the above example, page and autoFlush are keywords. True or false value can be set to autoFlush attribute, by default, its value is true . This means, the buffer will be flushed automatically when it is full. When the autoflush attribute is set to false, then an exception is thrown when the output buffer gets full and results in overflow.

**NOTE:** The user should not to set the autoflush to false when the buffer attribute is set to none.

### isThreadSafe:

isThreadSafe attribute is used to set whether the generated Servlet handles multiple requests or single requests, depending on the set value of true or false. isThreadSafe attribute is used to set and ensure whether thread safety is implemented in the JSP page.

Syntax of isThreadSafe attribute available for page directive is:

```
<%@ page isThreadSafe="true|false" %>
```

In the above statement, page and isThreadSafe are keywords. A true or false value can be set and, by default, the value is set to true. It implies that the JSP container can handle or send multiple concurrent client requests to the JSP page by starting a new thread. If the value of this attribute is set to false, then the JSP container sends client requests only one at a time to the JSP page.

### info:

Programmers make use of info attribute to place the information or documentation for a page. Details such as: author, version, copyright and date are placed in this attribute. This is actually text string that is written as input text in the compiled JSP page.
Syntax of info attribute available for page directive is:

```
<%@ page info = "text" %>
```

In the above statement, page and info are keywords. The details of documentation or information are placed inside " ". This is actually text string written as input text in the compiled JSP page.

**For example:**

```
<%@ page info = "exforsys.com example,2006" %>
```

The above text, exforsys.com example, 2006, is a text string written as text in the compiled JSP page.

### errorPage:

If the programmer wants to place errors in a different page then the URL to the error page can be mentioned in this attribute as errorPage.

Syntax of errorPage attribute available for page directive is as below:

```
<%@ page errorPage = "relativeURL" %>
```

In the above statement, page and errorPage are keywords.

For example, the user specifies the attribute errorpage is:

```
<%@ page errorPage = "/handleerr/testerr.jsp" %>
```

### isErrorPage:

isErrorPage attribute is used to specify whether or not a JSP page displays an error page by setting the value as true or false. By default, the value is set to false, meaning that the user cannot make use of the exception object in the JSP page. If the value is set to true, then it means that the user can make use of the exception object in the JSP page.

Syntax of isErrorPage attribute available for page directive is:

```
<%@ page isErrorPage="true|false" %>
```

In the above statement, page and isErrorPage are keywords. By default, the value is false.

### contentType:

contentType attribute is used to set the mime type and character set of the JSP. The user can make use of any MIME type or character set valid for the JSP container.

Syntax of contentType attribute available for page directive is:

```
<%@ page contentType="mimeType [; charset=characterSet]" %>
```

In the above statement, page and contentType are keywords. The default, MIMEtype is text/html, and the default character set is ISO-8859-1.

For example, the user specifies the attribute contentType is:

```
<%@ page contentType="text/html;charset=ISO-8859-1" %>
```

## JSP Directive Tag and Scriptlet tag

In this JSP tutorial, you will learn about two types of Directive tag namely Include and Tag Lib and also Scriptlet tag used in Java Server Pages along with syntax, usage, example and explanation for each of the tag.

### Include directive:

Include directive is a type of directive tag. If a programmer wants to include contents of a file inside another, then the Include directive is used. The file included can be either static ( HTML file) or

dynamic (i.e., another tag file). The include directive is used to include the contents of other file in the current tag file.

Syntax of Include directive is:

```
<%@ include file = "xxx.html/xx.tagf"%>
```

In the above statement, include and file are keywords. This includes either .html (static file) or .tagf file (dynamic file) between " " in the include directive.

**For instance:**

```
<%@ include file = "include/exforsys.html"%>
```

Above example shows how to include a static resource called exforsys.html in an include directive. The above example includes the html from exforsys.html found in the include directory into the current JPS page.

**For example:**

The user includes a dynamic file in the include directive:

```
<%@ include file="exforsys.tagf" %>
```

The above example shows how to include a dynamic resource called exforsys.tagf in the include directive.

## Tag Lib directive:

A user can customize actions from their tag file using the Tag Lib directive. The Tag Lib directive is a collection of custom tags.

The syntax of Tag Lib directive is:

```
<%@ taglib uri = "tag library URI" prefix = "tag Prefix" %>
```

In the above, taglib and uri are both keywords. The URI, which can be absolute or relative, uniquely identifies the tag library descriptor and is enclosed with " " in tag library URI. The tag prefix included inside " " is a string that will become the prefix to distinguish a custom action.

If the custom tag has a content body, then the format for using a custom tag in taglib directive is as follows:

```
<prefix:tagName>body</prefix:tagName>
```

If the custom tag does not have content body, then the format for using a custom tag in taglib directive is as follows:

```
<prefix:tagName/>
```

## Scriptlet tag:

Scriptlet tag is a type tag used for inserting Java Code into JSP.

Notation of the Scriptlet tag is:

<% %>

To begin the Scriptlet tag, the user must add <% . Inside the Scriptlet tag, the user can add any valid Scriptlet, meaning any valid Java Code. User can write the code in between the Scriptlet tag accesses any variable or bean declared. The Scriptlet tag ends with the notation %>.

General syntax of Scriptlet Tag:

```
<%!
    statement1;
    statement2;     //Valid Java Code
    ..........;
    ..........;
%>              //end of Scriptlet tag
```

**For instance**

```
<%
    for (int j = 0; j <5; j++)
    {
       out.print(j);
    }
%>
```

In the above example, the Java code embeds inside the tag <% and %>.

## JSP Action tags

In this section let us see about Action tag which is another type of tag, forward action tag and useBean action tag used in Java Server pages with syntax, usage, example and explanation in detail.

## Action Tag:

Action tag is used to transfer the control between pages and is also used to enable the use of server side JavaBeans. Instead of using Java code, the programmer uses special JSP action tags to either link to a Java Bean set its properties, or get its properties.

**General syntax of Action Tag:**

```
<jsp:action attributes />
```

In the above statement jsp is a keyword.

There are many action tags that are available for Java Server Pages. The most commonly used action tags are three of them and they are namely:

- include
- forward
- useBean

Syntax, attributes and usage of above three action tags are provided in brief.

## include action tag:

include is a type of directive tag. include tag has the same concept as that of the include directive. The include tag is used to include either static or dynamic content, wherever required.

**General syntax of include action Tag:**

```
<jsp:include page="{relativeURL | <%= expression %>}" flush="true" />
```

In the above statement, the words jsp, include, flush and page are keywords. The include action tag includes a static file or it is used to send a request to a dynamic file. If the included file is a static file, the contents are included as such in the called JSP file. In contrast, if the included file is dynamic, then a request sends to the dynamic file. Once the include action is completed, the result is sent back.

If the file is dynamic one can use the syntax as below:

```
<jsp:include page="{relativeURL | <%= expression %>}" flush="true" />
<jsp:param name="parameterName" value="{parameterValue | <%= expression %>}" />
</jsp:include>
```

The file is dynamic and extra attribute clause passed is jsp:param clause, and this is used to pass the name and value of a parameter to the dynamic file. In the above, the words jsp, include, flush, page, param, name and value are keywords. In the above syntax for dynamic file inclusion, the name attribute specifies the parameter name and its case-sensitive literal string. The value attribute specifies the parameter value, taking either a case-sensitive literal string or an expression that is evaluated at request time.

In the syntax of include action Tag we relative URL placed within " " denotes the location of the file to be included or denotes the pathname to be included. This can also be an expression which is taken automatically as string, denoting the relative URL.

**NOTE:** the pathname must not contain the protocol name, port number, or domain name. Both absolute and relative representations can be made for denoting the URL.

The flush attribute can only take on value "true" as mentioned in syntax notation.

**NOTE:** Do not to include the value if false for the flush attribute.

The syntax used for including dynamic file had <jsp:param> clause allows the user to pass one or more name/value pairs as parameters to an included file. This is the syntax for dynamic file, meaning that the included file is a dynamic file (i.e. either a JSP file or a servlet, or other dynamic file). Also, the user can pass more than one parameter to the included file by using more than one <jsp:param> clause.

**For example:**

An example to include a static file exforsys.html in the include action tag:

```
<jsp:include page="exforsys.html" flush="true" />
```

Let us see an example of dynamic file inclusion in the include action tag which is done as below:

```
<jsp:include page="example/test.jsp">
<jsp:include name="username" value="exforsys" />


</jsp:include>
```

Let us see about the other two commonly used action tags namely forward action tag and useBean action tag with syntax, usage, example and explanation in detail

## forward action tag:

The forward action tag is used to transfer control to a static or dynamic resource. The static or dynamic resource to which control has to be transferred is represented as a URL. The user can have the target file as an HTML file, another JSP file, or a servlet. Any of the above is permitted.

**NOTE:** The target file must be in the same application context as the forwarding JSP file.

**General syntax of forward action Tag:**

```
  <jsp:forward page="{relativeURL | <%= expression %>}" />
&nbsp;
```

Another syntax representation for forward action Tag is as below:

```
<jsp:forward page ="{relativeURL | <%= expression %>}" />
<jsp:param   name ="parameterName"
        value="parameterValue" | <%= expression %>}" />


</jsp:forward>
```

In the above statement, jsp, forward, page, param, name and value are all keywords. The relative URL of the file to which the request is forwarded is represented as a String, an expression or as absolute or relative representation to the current JSP file denoting the pathname.

**NOTE:** Do not include protocol name, port number, or domain name in the pathname. The file can be a JSP file or a servlet, or any other dynamic file.

If the user follows the second syntax representation of forward action Tag, then the target file should be dynamic where jsp:param> clause is used. This allows the sending of one or more name/value pairs as parameters to a dynamic file. The user can pass more than one parameter to the target file by using more than one <jsp:param> clause. In the above second representation, the name attribute is used to specify the parameter name and its case-sensitive literal string. The value attribute can be specified by the parameter value taking either a case-sensitive literal string or an expression that is evaluated at request time.

**For example:**

```
<jsp:forward page ="exforsys.html" />
<jsp:forward page ="/example/test" />
<jsp:param   name ="username"
        value="exforsys" />
</jsp:forward>
```

## useBean action tag:

The useBean action tag is the most commonly used tag because of its powerful features. It allows a JSP to create an instance or receive an instance of a Java Bean. It is used for creating or instantiating a bean with a specific name and scope. This is first performed by locating the instance of the bean.

**General syntax of useBean action Tag:**

```
<jsp:useBean
    id="beanInstanceName"
    scope="page|request|session|application"
    {
      class="package.class" |
      type="package.class" |
      class="package.class" type="package.class" |
      beanName="{package.class | <%= expression %>}"
      type="package.class"
    }
    { /> |
    > other elements
    </jsp:useBean>
}
```

The <jsp:useBean locates the instance of the bean. If the bean is not present, then the <jsp:useBean> instantiates it from a class.

In the above syntax of useBean action Tag there are several attributes mentioned. They are:

**id:**

This refers to a variable that is case sensitive, identifying the bean within scope attribute defined.

**scope:**

This refers to the scope of the existence of bean. The default scope is page.

**class:**

The class name is mentioned and using this bean is Instantiated. The data type to which bean must be assigned is mentioned in type.

**beanName:**

When this is specified, the bean is instantiated by the java.beans.Beans.instantiate method.

## JSP Request Object

In this JSP tutorial, you will learn about JSP request object, JSP request object methods, getParameter(String name), getParameterNames(), getParameterValues(String name), getQueryString(), getRequestURI(), getServletPath(), setAttribute(String,Object), removeAttribute(String)

request object in JSP is used to get the values that the client passes to the web server during an HTTP request. The request object is used to take the value from the client's web browser and pass it to the server. This is performed using an HTTP request such as: headers, cookies or arguments. The class or the interface name of the object request is http.httpservletrequest. The object request is written: Javax.servlet.http.httpservletrequest.

### Methods of request Object:

There are numerous methods available for request Object. Some of them are:

- getCookies()
- getHeader(String name)
- getHeaderNames()
- getAttribute(String name)
- getAttributeNames()
- getMethod()
- getParameter(String name)
- getParameterNames()
- getParameterValues(String name)
- getQueryString()
- getRequestURI()
- getServletPath()
- setAttribute(String,Object)
- removeAttribute(String)

Below is a list of the usage with syntax and examples with explanation of each.

### getCookies():

The getCookies() method of request object returns all cookies sent with the request information by the client. The cookies are returned as an array of Cookie Objects. We will see in detail about JSP cookies in the coming sections.  General syntax of getHeader() of request object is request.getHeader("String").

getHeader()request object returned value is a string.

For instance String exfosys = request.getHeader("exforsys")

The above would retrieve the value of the HTTP header whose name is exforsys in JSP.

### getHeader(String name):

The method getHeader(String name) of request object is used to return the value of the requested header. The returned value of header is a string.  General syntax of getHeader() of request object is request.getHeader("String")

In the above the returned value is a String.

For instance:  String exfosys = request.getHeader("exforsys")

The above retrieves the value of the HTTP header named exforsys in JSP.

**getHeaderNames():**

The method getHeaderNames() of request object returns all the header names in the request. This method is used to find available headers. The value returned is an enumerator of all header names. General syntax of getHeaderNames() of request object is as follows: request.getHeaderNames();

In the above the returned value is an enumerator.

**For example:**  Enumeration exforsys = request.getHeaderNames();

The above returns all header names under the enumerator exforsys.

**getAttribute(String name):**

The method getAttribute() of request object is used to return the value of the attribute. The getAttribute() method returns the objects associated with the attribute. When the attribute is not present, then a null value is returned. If the attribute is present then the return value is the object associated with the attribute.  General syntax of getAttribute() of request object is request.getAttribute()

In the above the returned value is an object.

**For example:**  Object exforsys = request.getAttribute("test");

The above retrieves the object stored in the request test and returns the object in exforsys.

**getAttributeNames():**

The method getAttribute() of request object is used to return the object associated with the particular given attribute. If the user wants to get names of all the attributes associated with the current session, then the request object method getAttributeNames() can be used. The returned value is an enumerator of all attribute names.  General syntax of getAttributeNames() of request object is request.getAttributeNames()

**For example:**  Enumeration exforsys = request.getAttributeNames();

The above returns all attribute names of the current session under the enumerator: exforsys.

**getMethod():**

The getMethod() of request object is used to return the methods GET, POST, or PUT corresponding to the requested HTTP method used.  General syntax of getMethod() of request object is request.getMethod()

**For example:**

```
if (request.getMethod().equals("POST"))
{
.........
.......
}
```

In the above example, the method returned by the request.getMethod is compared with POST Method and if the returned method from request.getMethod() equals POST then the statement in if block executes.

**getParameter(String name):**

getParameter() method of request object is used to return the value of a requested parameter. The returned value of a parameter is a string. If the requested parameter does not exist, then a null value is returned. If the requested parameter exists, then the value of the requested parameter is returned as a string.  General syntax of getParameter() of request object is request.getParameter(String name)

The returned value by the above statement is a string.

**For example:** String exforsys = request.getParameter("test");

The above example returns the value of the parameter test passed to the getParameter() method of the request object in the string exforsys. If the given parameter test does not exist then a null value is assigned to the string exforsys.

**getParameterNames():**

The getParameterNames() method of request object is used to return the names of the parameters given in the current request. The names of parameters returned are enumeration of string objects. General syntax of getParameterNames() of request object is  request.getParameterNames()

Value returned from the above statement getParameterNames() method is enumeration of string objects.

```
Enumeration exforsys = request.getParameterNames();
```

The above statement returns the names of the parameters in the current request as an enumeration of string object.

**getParameterValues(String name):**

The getParameter(String name) method of request object was used to return the value of a requested given parameter. The returned value of the parameter is a string. If there are a number of values of parameter to be returned, then the method getParameterValues(String name) of request object can be used by the programmer. The getParameterValues(String name) method of request object is used to return all the values of a given parameter's request. The returned values of parameter is a array of string objects. If the requested parameter is found, then the values associated with it are returned as array of string object. If the requested given parameter is not found, then null value is returned by the method.  General syntax of getParameterValues of request object is request.getParameterValues(String name)

The returned value from the above method getParameterValues() is array of string objects.

**For example:** `String[] vegetables = request.getParameterValues("vegetable");`

The above example returns a value of parameter vegetable passed to the method getParameterValues() of request object and the returned values are array of string of vegetables.

## getQueryString():

The getQueryString() method of request object is used to return the query string from the request. From this method, the returned value is a string.  General syntax of getQueryString() of request object is request.getQueryString()

Value returned from the above method is a string.

**For example:** String exforsys=request.getQueryString();
out.println("Result is"+exforsys);

The above example returns a string exforsys from the method getQueryString() of request object. The value is returned and the string is printed in second statement using out.println statement.

## getRequestURI():

The getRequestURI() method of request object is used for returning the URL of the current JSP page. Value returned is a URL denoting path from the protocol name up to query string.  General syntax of getRequestURI() of request object is request.getRequestURI()

The above method returns a URL.

**For example:** out.println("URI Requested is " + request.getRequestURI());

Output of the above statement would be: URI Requested is /Jsp/test.jsp

### getServletPath():

The getServletPath() method of request object is used to return the part of request URL that calls the servlet. General syntax of getServletPath() of request object is request.getServletPath()

The above method returns a URL that calls the servlet.

**For example:** out.println("Path of Servlet is " + request.getServletPath());

The output of the above statement would be: Path of Servlet is/test.jsp

### setAttribute(String,Object):

The setAttribute method of request object is used to set object to the named attribute. If the attribute does not exist, then it is created and assigned to the object. General syntax of setAttribute of request object is request.setAttribute(String, object)

In the above statement the object is assigned with named string given in parameter.

**For example:** request.setAttribute("username", "exforsys");

The above example assigns the value exforsys to username.

### removeAttribute(String):

The removeAttribute method of request object is used to remove the object bound with specified name from the corresponding session. If there is no object bound with specified name then the method simply remains and performs no function. General syntax of removeAttribute of request object is request.removeAttribute(String);

## JSP Implicit and Session Objects

In this JPS tutorial, you will learn how to program using JSP, JSP expressions and Implicit Objects, JSP Session Object, methods of session object, getAttribute(String name), getAttributeNames and isNew().

### JSP expressions:

If a programmer wants to insert data into an HTML page, then this is achieved by making use of the JSP expression.

**General syntax:**

The general syntax of JSP expression is as follows:

```
<%= expression %>
```

The expression is enclosed between the tags <%= %>

For example, if the programmer wishes to add 10 and 20 and display the result, then the JSP expression written would be as follows:

```
<%= 10+20 %>
```

**implicit Objects:**

Implicit Objects in JSP are objects that are automatically available in JSP. Implicit Objects are Java objects that the JSP Container provides to a developer to access them in their program using JavaBeans and Servlets. These objects are called implicit objects because they are automatically instantiated.

There are many implicit objects available. Some of them are:

**request:** The class or the interface name of the object request is http.httpservletrequest. The object request is of type Javax.servlet.http.httpservletrequest. This denotes the data included with the HTTP Request. The client first makes a request that is then passed to the server. The requested object is used to take the value from client's web browser and pass it to the server. This is performed using HTTP request like headers, cookies and arguments.

**response:** This denotes the HTTP Response data. The result or the information from a request is denoted by this object. This is in contrast to the request object. The class or the interface name of the object response is http.HttpServletResponse. The object response is of type Javax.servlet.http. >httpservletresponse. Generally, the object response is used with cookies. The response object is also used with HTTP Headers.

**Session:** This denotes the data associated with a specific session of user. The class or the interface name of the object Session is http.HttpSession. The object Session is of type Javax.servlet.http.httpsession. The previous two objects, request and response, are used to pass information from web browser to server and from server to web browser respectively. The Session Object provides the connection or association between the client and the server. The main use of Session Objects is for maintaining states when there are multiple page requests. This will be explained in further detail in following sections.

**Out:** This denotes the Output stream in the context of page. The class or the interface name of the Out object is jsp.JspWriter. The Out object is written: Javax.servlet.jsp.JspWriter

**PageContext:** This is used to access page attributes and also to access all the namespaces associated with a JSP page. The class or the interface name of the object PageContext is jsp.pageContext. The object PageContext is written: Javax.servlet.jsp.pagecontext

**Application:** This is used to share the data with all application pages. The class or the interface name of the Application object is ServletContext. The Application object is written: Javax.servlet.http.ServletContext

**Config:** This is used to get information regarding the Servlet configuration, stored in the Config object. The class or the interface name of the Config object is ServletConfig. The object Config is written Javax.servlet.http.ServletConfig

**Page:** The Page object denotes the JSP page, used for calling any instance of a Page's servlet. The class or the interface name of the Page object is jsp.HttpJspPage. The Page object is written: Java.lang.Object

The most commonly used implicit objects are request, response and session objects

## JSP Session Object

The previous section listed implicit objects available in JSP and detailed each of the methods of request object and response object. This section details the Session Object and the methods available with syntax and explanation on each.
Session Object denotes the data associated with a specific session of user. The class or the interface name of the object session is http.HttpSession. The object session is written as:

Javax.servlet.http.httpsession.

The previous two objects, request and response, are used to pass information from web browser to server and from server to web browser respectively.
The Session Object provides the connection or association between the client and the server. The main use of Session Objects is to maintain states when there are multiple page requests.
The main feature of session object is to navigate between multiple pages in a application where variables are stored for the entire user session. The session objects do not lose the variables and the value remains for the user' session. The concept of maintenance of sessions can be performed by cookies or URL rewriting. A detailed approach of session handling will be discusses in coming sections.

### Methods of session Object:

There are numerous methods available for session Object. Some are:

- getAttribute(String name)
- getAttributeNames
- isNew()
- getCreationTime
- getId
- invalidate()
- getLastAccessedTime
- getMaxInactiveInterval
- removeAttribute(String name)
- setAttribute(String, object)

Detailed usage with syntax and example with explanation of each of these methods.

### getAttribute(String name):

The getAttribute method of session object is used to return the object with the specified name given in parameter. If there is no object then a null value is returned.
General syntax of getAttribute of session object is as follows:

```
session.getAttribute(String name)
```

The value returned is an object of the corresponding name given as string in parameter. The returned value from the getAttribute() method is an object written: java.lang.Object.

**For example:**

```
String exforsys = (String) session.getAttribute("name");
```

In the above statement, the value returned by the method getAttribute of session object is the object of name given in parameter of type java.lang. Object and this is typecast to String data type and is assigned to the string exforsys.

getAttributeNames:

The getAttributeNames method of session object is used to retrieve all attribute names associated with the current session. The name of each object of the current session is returned. The value returned by this method is an enumeration of objects that contains all the unique names stored in the session object.

General Syntax:

```
session.getAttributeNames()
```

The returned value by this method getAttributeNames() is Enumeration of object.

**For example:**

```
exforsys = session.getAttributeNames( )
```

The above statement returns enumeration of objects, which contains all the unique names stored in the current session object in the enumeration object exforsys.

**isNew():**

The isNew() method of session object returns a true value if the session is new. If the session is not new, then a false value is returned. The session is marked as new if the server has created the session, but the client has not yet acknowledged the session. If a client has not yet chosen the session, i.e., the client switched off the cookie by choice, then the session is considered new. Then the isNew() method returns true value until the client joins the session. Thus, the isNew() method session object returns a Boolean value of true of false.

General syntax of isNew() of session object is as follows:

```
session.isNew()
```

The returned value from the above method isNew() is Boolean

**JSP Response Object**

In this JSP tutorial, you will learn about JSP Response object, Methods of response Object, setContentType(), addCookie(Cookie cookie), containsHeader(String name), setHeader(String name, String value), sendRedirect(String) and sendError(int status_code).

<p>The response object denotes the HTTP Response data. The result or the information of a request is denoted with this object. The response object handles the output of the client. This contrasts with the request object. The class or the interface name of the response object is http.HttpServletResponse.

The response object is written: Javax.servlet.http.httpservletresponse.

The response object is generally used by cookies.

The response object is also used with HTTP Headers.

## Methods of response Object:

There are numerous methods available for response object. Some of them are:

- setContentType()
- addCookie(Cookie cookie)
- addHeader(String name, String value)
- containsHeader(String name)
- setHeader(String name, String value)
- sendRedirect(String)
- sendError(int status_code)

List below details the usage with syntax, example and explanation of each of these methods.

### setContentType():

setContentType() method of response object is used to set the MIME type and character encoding for the page.

General syntax of setContentType() of response object is as follows:

```
        response.setContentType();
```

**For example:**

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

The above statement is used to set the content type as text/html dynamically.

### addCookie(Cookie cookie):

addCookie() method of response object is used to add the specified cookie to the response. The addcookie() method is used to write a cookie to the response. If the user wants to add more than one cookie, then using this method by calling it as many times as the user wants will add cookies.

General syntax of addCookie() of response object is as follows:

```
        response.addCookie(Cookie cookie)
```

**For example:**

```
        response.addCookie(Cookie exforsys);
```

The above statement adds the specified cookie exforsys to the response.

### addHeader(String name, String value):

addHeader() method of response object is used to write the header as a pair of name and value to the response. If the header is already present, then value is added to the existing header values.

General syntax of addHeader() of response object is as follows:

```
response.addHeader(String name, String value)
```

Here the value of string is given as second parameter and this gets assigned to the header given in first parameter as string name.

**For example:**

```
response.addHeader("Author", "Exforsys");
```

The output of above statement is as below:

```
Author: Exforsys
```

**containsHeader(String name):**

containsHeader() method of response object is used to check whether the response already includes the header given as parameter. If the named response header is set then it returns a true value. If the named response header is not set, the value is returned as false. Thus, the containsHeader method is used to test the presence of a header before setting its value. The return value from this method is a Boolean value of true or false.

General syntax of containsHeader() of response object is as follows:

```
response.containsHeader(String name)
```

Return value of the above containsHeader() method is a Boolean value true or false.

**setHeader(String name, String value):**

setHeader method of response object is used to create an HTTP Header with the name and value given as string. If the header is already present, then the original value is replaced by the current value given as parameter in this method.

General syntax of setHeader of response object is as follows:

```
response.setHeader(String name, String value)
```

**For example:**

```
response.setHeader("Content_Type","text/html");
```

The above statement would give output as

```
Content_Type: text/html
```

**sendRedirect(String):**

sendRedirect method of response object is used to send a redirect response to the client temporarily by making use of redirect location URL given in parameter. Thus the sendRedirect method of the response object enables one to forward a request to a new target. But one must note that if the JSP executing has already sent page content to the client, then the sendRedirect() method of response object will not work and will fail.

General syntax of sendRedirect of response object is as follows:

```
response.sendRedirect(String)
```

In the above the URL is given as string.

**For example:**

```
response.sendRedirect("http://xxx.test.com/error.html");
```

The above statement would redirect response to the error.html URL mentioned in string in Parameter of the method sendRedirect() of response object.

**sendError(int status_code):**

sendError method of response object is used to send an error response to the client containing the specified status code given in parameter.

General syntax of sendError of response object is as follows:

```
response.sendError(int status_code)
```

## JSP Architecture

In this JSP tutorial, you will learn about JSP Architecture, page-centric approach, dispatcher approach and steps in execution of a JSP file.

JSP is a high-end technology that helps developers insert java code in HTML pages by making use of special JSP tags.
The JSP are HTML pages but do not automatically have .html as file extension. JSP files have .jsp as extension. The following steps takes place in execution of a JSP file.

- JSP files are compiled by JSP engine into a servlet. This step creates the .jsp file as a Java servlet source file.

- Once this is processed, the source file above is compiled into a class file.

- The engine then makes use of the compiled servlet from the above process and executes requests.

Out of the two processes, the first two take time to produce a compiled servlet. This is performed only once unless modification in the source file is required. Once the compiled servlet is completed, the execution of requests is performed at a faster speed.

There are two methods for using JSP technology:

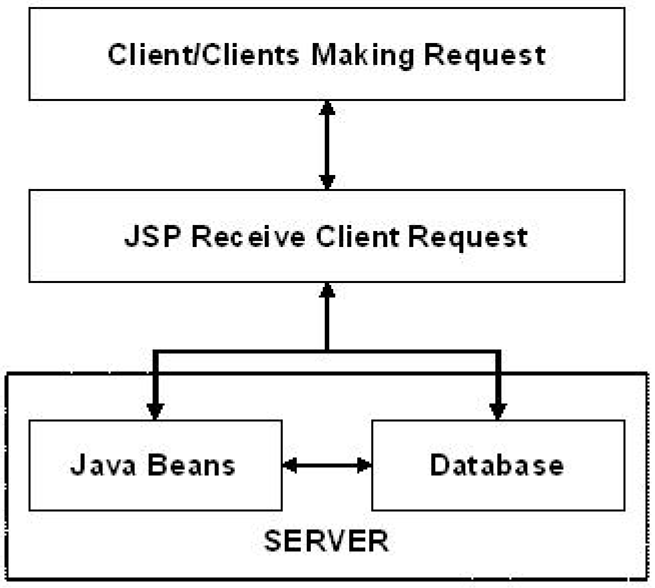
- Page-Centric Approach
- Dispatcher Approach

**Page-Centric Approach:**

The page-centric approach is also called Client-Server approach. The basic idea of Client-Server approach is that the application lies on the client side and services the requests connecting to the server application. JSP using this approach-processes as follows:

Client makes a request. The JSP page takes the request from client and processes it. The JSP have access to the database by using Java Beans. The requests are processed and the serviced results are sent back to client by JSP.

The above approach has the advantage of simplifying the process but its disadvantage is when the number of clients increases, the process becomes difficult to handle.



**Dispatcher Approach:**

This is also called N-tier approach, where the server side of the above architecture is divided into multiple tiers, using JSP as a controller, passing requests to Java Beans.

JSP is popular because of its processing ability. Processing is distinctly divided between Presentation and Front Components. The popular JSP Architecture is the *Model View Controller* (MVC) model. In this MVC model, the request is sent by the browser to the controller or the servlet. This request is instantiated by the servlet as a Java Bean by JSP. The main aspect is JSP are compiled into servlets at the back end and the front end tasks are not interrupted. The servlet engine takes up the responsibility of compiling JSP Servlet and producing the final JSP servlet class for usage. The front end presentation modules are handled by JSP for viewing and the manipulation of data is handled by Java Bean and passed back to JSP when needed. The Presentation part of the MVC Model has no processing logic. It performs the task of extracting beans or objects that may have been initially created by the controller. It also extracts the dynamic content within for insertion within its static templates. The Application Manager in the MVC Model is the Controller that processes HTTP requests. They are not responsible for presentation tasks. That can be either servlets or JSP. They take the task of managing the application state, security, and presentation uniformity and thus, have a single point of entry.

This explains the approach and the process of execution of a request.

The following steps execute a JSP request from the time the request is received from client until it is processed and sent back to client.

**Step1:**
**Request from Client:**

A JSP page has the extension as .jsp. The client request is made on the web browser by going to the .jsp extension JSP file.

**Step2:**
**Request Sent To Server:**

The request received from client (web browser) is sent to the server side.

**Step3:**
**JSP Servlet Engine:**

Since the extension of request made has .jsp extension, it indicates that it is a JSP file and therefore the web server passes the request received to the JSP Servlet Engine.

**Step4:**
**Process of generating servlet:**

JSP files are compiled by the JSP engine into a servlet. This step creates the .jsp file as a Java servlet source file.

**Step5:**
**Process of making class file:**

The source file from Step4 is compiled into a class file.

**Step6:**
Instantiation of Servlet:

The instantiation of servlet is performed using the init and service methods. In these methods, the jspInit() method is defined by the developer and the jspService method is generated by the JSP engine.

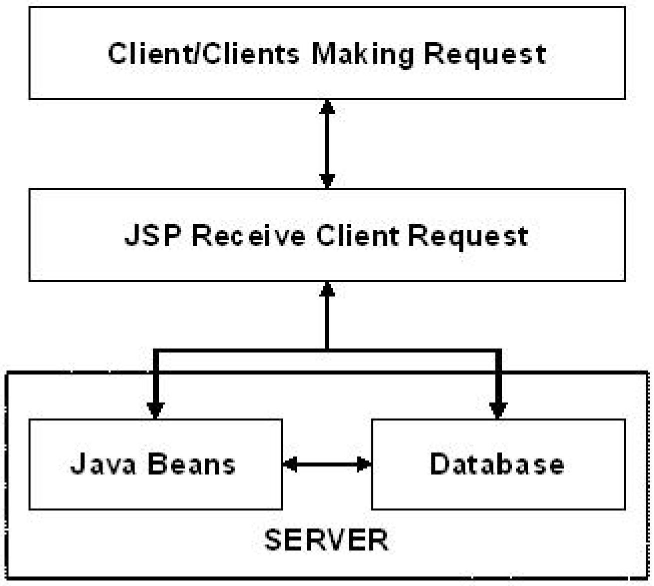
**Step7:**
Output HTML:

The request received from client is executed and the output is sent as HTML.

**Step8:**

**Client Receives the Output:**

The Client Receives the Output and thus the result namely the HTML gets displays in the client browser.



## Working with JSP Sessions

In this JSP tutorial, you will learn about JSP session object methods, getCreationTime, getLastAccessedTime, getId, invalidate(), getMaxInactiveInterval, setMaxInactiveInterval(), removeAttribute(String name) and setAttribute(String, object).

This section details the syntax, usage, example and explanation of more session object methods, such as:

- getCreationTime
- getLastAccessedTime
- getId
- invalidate()
- getMaxInactiveInterval
- setMaxInactiveInterval()
- removeAttribute(String name)
- setAttribute(String, object)

## getCreationTime:

The getCreationTime method of session object is used to return the session created time. The returned time value would be in milliseconds, the time value is midnight January 1, 1970 GMT.

General syntax of getCreationTime of session object is as follows:

```
session.getCreationTime()
```

The above method returns value of time in milliseconds.

**For example:**

```
out.println("Created Time of Session is: " + session.getCreationTime());
```

The above statement would display

```
Created Time of Session is: 974154073972
```

The above output depicts the creation time of session since January 1, 1970 GMT in milliseconds.

## getLastAccessedTime:

The getLastAccessedTime method of session object is used to return the latest time of the client request associated with the session. By using this method, it is possible to determine the last time the session was accessed before the current request. The returned time value would be in milliseconds and the time value is since midnight January 1, 1970 GMT.

General syntax of getLastAccessedTime of session object is as follows:

```
session.getLastAccessedTime()
```

The above method returns the value of time in milliseconds.

**For example:**

```
out.println("Last Accessed Time of Session is: " +
session.getLastAccessedTime());
```

The above statement would display:

```
Last Accessed Time of Session is: 953044321813
```

The above output depicts the last time the session was accessed before the current request since January 1, 1970 GMT in milliseconds.

## getId:

The getID method of session object is used to return the unique identifier associated with the session.

General syntax of getID of session object is as follows:

```
session.getId()
```

**For example:**

```
    out.println("The Session ID is: " + session.getId());
```

The above statement would display

```
    The Session ID is: A1BQWTBBBBBBKSY2HJKQBBB
```

The above statement denotes the unique identifier associated with the current session.

**invalidate():**

invalidate method of session object is used to discard the session and releases any objects stored as attributes. This method helps to reduce memory overhead and achieves improvement in performance. It is always a good practice to explicitly remove or invalidate sessions using session.invalidate() method.

General syntax of invalidate of session object is as follows:

```
    session.invalidate()
```

**getMaxInactiveInterval:**

The getMaxInactiveInterval method of session object is used to return the maximum amount of time the JRun keeps the session open between client accesses. This returns the maximum amount of time, in seconds, that a session can be inactive before it is deleted. The returned value of time is in seconds. Thus, by using this method the user determines how long it will take a session for it to time out. The default timeout period for sessions defined by the servlet container is determined using the getMaxInactiveInterval method. The returned value from this method is seconds and thus, an integer.

General syntax of getMaxInactiveInterval of session object is as follows:

```
    session.getMaxInactiveInterval()
```

The above method returns value of time in seconds.

**For example:**

```
    out.println("Maximum Inactive Interval of Session in Seconds is : "
    +session.getMaxInactiveInterval());
```

The above statement would display

```
    Maximum Inactive Interval of Session in Seconds is : 2000
```

The above denotes the time in seconds.

**setMaxInactiveInterval():**

This is another setMaxInactiveInterval() method that a developer can use to set the timeout explicitly for each session. A user can use this method to set the default timeout.

General syntax of setMaxInactiveInterval of session object is as follows:

```
    session.setMaxInactiveInterval(time)
```

In the above the time given in parameter is in seconds.

**For example:**

```
session.setMaxInactiveInterval(600);
```

In the above statement, the inactivity period for the session would be set to 10minutes.The parameter 600 given in the method would be in seconds.

## removeAttribute(String name):

The removeAttribute method of session object is used to remove the attribute and value from the session.

General syntax of removeAttribute of session object is as follows:

```
session.removeAttribute(String)
```

**For example:**

```
session.removeAttribute("exforsys");
```

Example exforsys given in parameter of removeAttribute method is a string.

## setAttribute(String, object):

The setAttribute method of session object is used to set the object to the named attribute. This method is used to write an attribute and value to the session. If the attribute does not exist, then it is created and then the object is associated with this.

General syntax of setAttribute of session object is as follows:

```
session.setAttribute(String, object)
```

**For example:**

```
String exforsys = request.getParameter("test");
session.setAttribute("test", exforsys);
```

In the above example, the first parameter passed to the method setAttribute test denotes a string and the second parameter exforsys denotes the object. By passing these to the setattribute method, the object exforsys is set with the string test.

## JSP Out Object

In this JSP tutorial, you will learn about out object and its methods viz. clear, clearBuffer, flush, isAutoFlush, getBufferSize, getRemaining, newLine, print and println.

out object denotes the Output stream in the context of page. The class or the interface name of the object out is jsp.JspWriter.

The out object is written as:

```
Javax.servlet.jsp.JspWriter
```

The object that write to the JSP's output stream is defined by the out object.

## Methods of out Object:

There are numerous methods available for out Object, such as:

- clear
- clearBuffer
- flush
- isAutoFlush
- getBufferSize
- getRemaining
- newLine
- print
- println

Detailed below is the usage with syntax, example and explanation of above methods.

### clear:

As the name implies, the clear method of out object is used to clear the output buffer. This method does not write any contents to the client. An exception is thrown by this method if the buffer was flushed.

General syntax of clear method of out object is as follows:

```
out.clear();
```

### clearBuffer:

The clearBuffer method of out object is used to clear the output buffer. This method does not write any contents to the client.

The only difference between the clear method of out object and clearBuffer method is

- clear method throws an exception when the buffer is flushed.
- clearBuffer method does not throw an exception when the buffer is flushed.

General syntax of clearBuffer method of out object is as follows:

```
out.clearBuffer();
```

**For example**

```
if (out.getBufferSize() != 0)
out.clearBuffer();
```

**out.clearBuffer();**

**For example**

```
if (out.getBufferSize() != 0)
out.clearBuffer();
```

**flush:**

Two methods of out object, clear and clearBuffer are used to clear the output buffer without writing any contents to the client. If you wish to flush the buffer and write contents to the client, then you can use the flush method of out object. The flush method of out object is used to flush the buffer by writing the contents to the client.

General syntax of flush method of out object is as follows:

```
out.flush();
```

**For example:**

```
out.flush();
jsp:forward page="exforsys.jsp"
```

The above example will output only the first line of text. This text will be the only thing viewed in the browser. In the above example, no forwarding will take place. This is because of the out.flush() method. This method will flush the JspWriter buffer. Once the content is written to the browser, a forward cannot be invoked. Trying this throws the exception IllegalStateException. The user can handle this exception by catching it using the catch block. Handling Exceptions will be covered in detail in a later section of this tutorial.

isAutoFlush:

The isAutoFlush method of out object returns a true value if the output buffer is automatically flushed.

General syntax of isAutoFlush method of out object is as follows:

```
out.isAutoFlush
```

The returned value from this method is Boolean value of true or false.

## getBufferSize:

The getBufferSize method of out object is used to return the size of the buffer. The returned value of the size of the buffer is in bytes. If the output is not buffered, then the getBufferSize method returns a 0 byte.

General syntax of getBufferSize method of out object is as follows:

```
out.getBufferSize()
```

The return value from the above method is in bytes.

**For example:**

```
if (out.getBufferSize() != 0)
out.getBufferSize()
```

## getRemaining:

The getRemaining method of out object is used to return the number of empty bytes in the buffer.

General syntax of getRemaining method of out object is as follows:

```
out.getRemaining();
```

## newLine:

As the name implies, the newLine method of out object is used to write a newline character to the output.

The general syntax of newLine method of out object is as follows:

```
out.newLine();
```

**For example:**

If a JSP program has the following statements in sequence

```
out.write("Welcome!!!");
out.newLine();
out.write("To Exforsys Training");
out.newLine();
```

Then the output would be as follows:

```
Welcome!!!
```

```
To Exforsys Training
```

The newLine method of out object gives a new line to the output and thus, the parameter given in third out.write statement "To Exforsys Training" is written in the next line.

print:

The print method of out object writes the value to the output without a newline character.

General syntax of print method of out object is as follows:

```
out.print();
```

**For example:**

If a JSP program has the following statements in sequence

```
out.print("Welcome!!!");
out.print("To Exforsys Training");
```

would give output as

```
Welcome!!! To Exforsys Training
```

This is because the print method of out object writes the output without a newline character.

**println:**

The println method of out object is used to write the value to the output, including the newline character.

General syntax of println method of out object is as follows:

```
out.println();
```

**For example:**

```
out.println("The Output is:" + ex);
```

Here if the value of ex is 5 then the output is

```
The Output is:5
```

Here the string concatenation operator + is used along with println method of out object.

**JSP Application Object**

In this JSP tutorial, you will learn about application object, the methods available in application object, getAttribute(String name), getAttributeNames, setAttribute(String objName, Object object), removeAttribute(String objName), getMajorVersion(), getMinorVersion(), getServerInfo(), getInitParameter(String name), getInitParameterNames, getResourceAsStream(Path) and log(Message)

Application Object is used to share the data with all application pages. Thus, all users share information of a given application using the Application object. The Application object is accessed by any JSP present in the application. The class or the interface name of the object application is ServletContext.

The application object is written as:

```
Javax.servlet.http.ServletContext
```

Associated with the Application object are methods that provide details about the Container of JSP and utility methods.

## Methods of Application Object:

There are numerous methods available for Application object. Some of the methods of Application object are:

- getAttribute(String name)
- getAttributeNames
- setAttribute(String objName, Object object)
- removeAttribute(String objName)
- getMajorVersion()
- getMinorVersion()
- getServerInfo()
- getInitParameter(String name)
- getInitParameterNames
- getResourceAsStream(Path)
- log(Message)

Detailed below is the usage of Application object along with syntax, example and explanation for some of the above methods.

## getAttribute(String name):

The method getAttribute of Application object is used to return the attribute with the specified name. It returns the object given in parameter with name. If the object with name given in parameter of this getAttribute does not exist, then null value is returned.

General syntax of getAttribute method of Application object is as follows:

```
application.getAttribute(String name);
```

**For example:**

```
application.getAttribute("exforsys");
```

The above statement returns the object exforsys.

## getAttributeNames:

The method getAttributeNames of Application object is used to return the attribute names available within the application. The names of attributes returned are an Enumeration.

General syntax of getAttributeNames method of Application object is as follows:

```
application.getAttributeNames();
```

**For example:**

```
Enumeration exforsys;
```

```
exforsys=application.getAttributeNames();
```

The above example returns the attribute names available within the current application as enumeration in exforsys.

## setAttribute(String objName, Object object):

The method setAttribute of Application object is used to store the object with the given object name in the application.

General syntax of setAttribute method of Application object is as follows:

```
application.setAttribute(String objName, Object object);
```

The above syntax stores the objname mentioned in String in the corresponding object mentioned as Object in the parameter of the setAttribute method.

**For example:**

```
application.setAttribute("exvar", "Exforsys");
```

In the above example, the object exvar is stored with the object name Exforsys in the application.

## removeAttribute(String objName):

The method removeAttribute of Application object is used to remove the name of the object mentioned in parameter of this method from the object of the application.

General syntax of removeAttribute method of Application object is as follows:

```
application.removeAttribute(String objName);
```

**For example:**

```
application.setAttribute("password",password);
application.removeAttribute("password");
```

The above statement removes the name from the object password of the application.

## getMajorVersion():

The method getMajorVersion of Application object is used to return the major version of the Servlet

API for the JSP Container.

General syntax of getMajorVersion method of Application object is as follows:

```
application.getMajorVersion();
```

The returned value from the above method is an integer denoting the major version of the Servlet API.

**For example:**

```
out.println("Major Version:"+application.getMajorVersion());
```

```
Major Version:2
```

The above statement gives 2 as the major version of the Servlet API in use for the Application object.

### getMinorVersion():

The method getMinorVersion of Application object is used to return the minor version of the Servlet API for the JSP Container.

General syntax of getMinorVersion method of Application object is as follows:

```
application.getMinorVersion();
```

The returned value from the above method is an integer denoting the minor version of the Servlet API.

**For example:**

```
out.println("Minor Version:"+application.getMinorVersion());
```

```
Minor Version:1
```

The above gives 1 as the minor version of the Servlet API in use for the Application object.

### getServerInfo():

The method getServerInfo of Application object is used to return the name and version number of the JRun servlet engine. Information about the JSP Container, such as, the name and product version, are returned by the method getServerInfo of Application object.

General syntax of getServerInfo method of Application object is as follows:

```
application.getServerInfo();
```

**For example:**

```
out.println("Server Information:"+application.getServerInfo());
```

### getInitParameter(String name):

The method getInitParameter of Application object is used to return the value of an initialization parameter. If the parameter does not exist, then null value is returned.

General syntax of getInitParameter method of Application object is as follows:

```
application.getInitParameter(String name);
```

**For example:**

```
String exforsys = application.getInitParameter("eURL");
```

In the above, the value of initialization parameter eURL is retrieved and stored in string exforsys.

**getInitParameterNames:**

The method getInitParameterNames of Application object is used to return the name of each initialization parameter. The returned value is an enumeration. General syntax of getInitParameterNames method of Application object is as follows:

```
application.getInitParameterNames();
```

The returned value from the above method is an enumeration.

**For example:**

```
Enumeration e;
e=application.getInitParameterNames();
```

**getResourceAsStream(Path):**

The method getResourceAsStream of Application object is used to translate the resource URL mentioned as parameter in the method into an input stream to read. General syntax of getResourceAsStream method of Application object is as follows:

```
application.getResourceAsStream(Path);
```

**For example:**

```
InputStream stream = application.getResourceAsStream("/exforsys.txt");
```

The above example translates the URL /exforsys.txt mentioned in the parameter of getResourceAsStream method into an input stream to read.

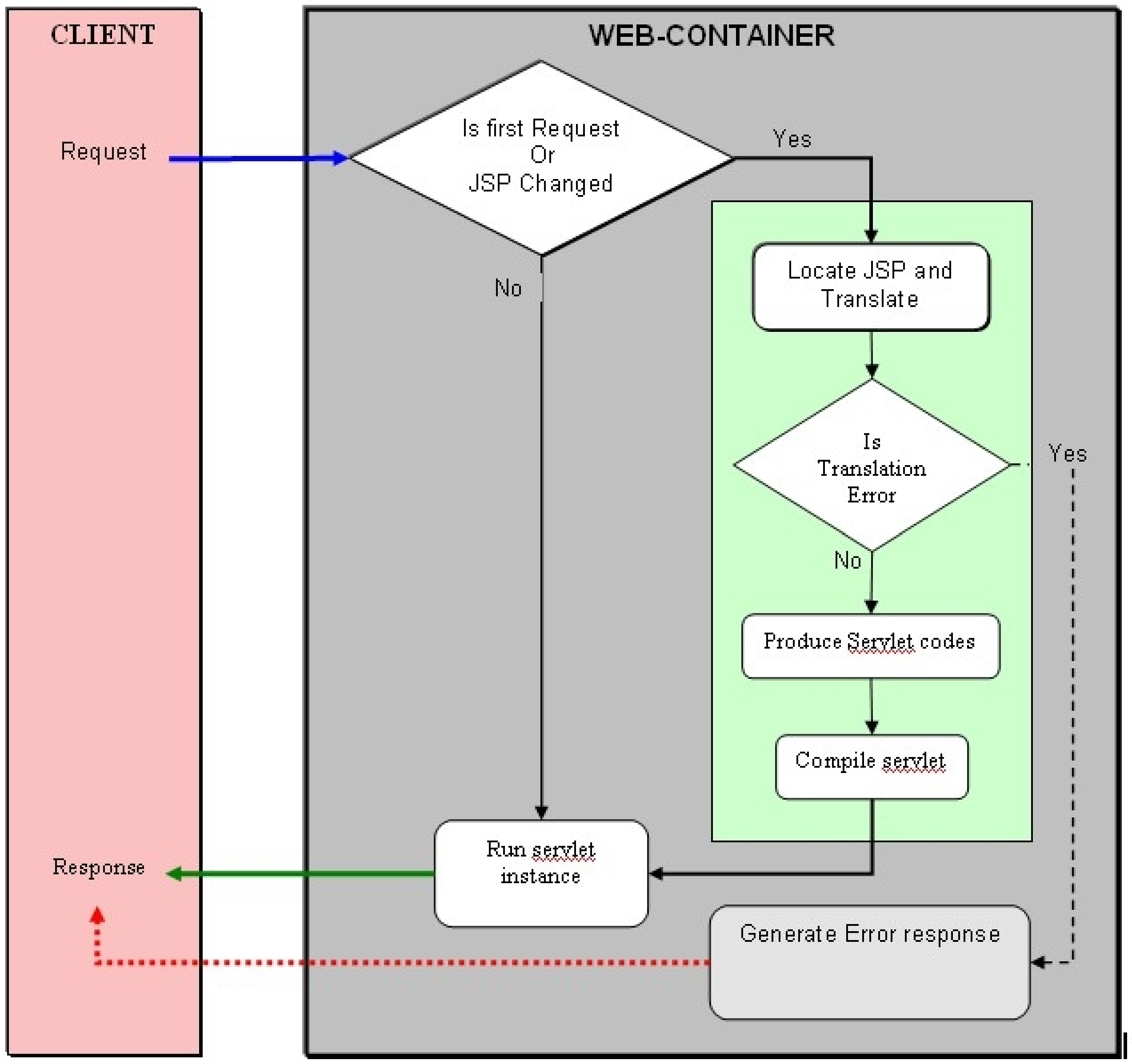
**log(Message):**

The method log of Application object is used to write a text string to the JSP Container’s default log file.

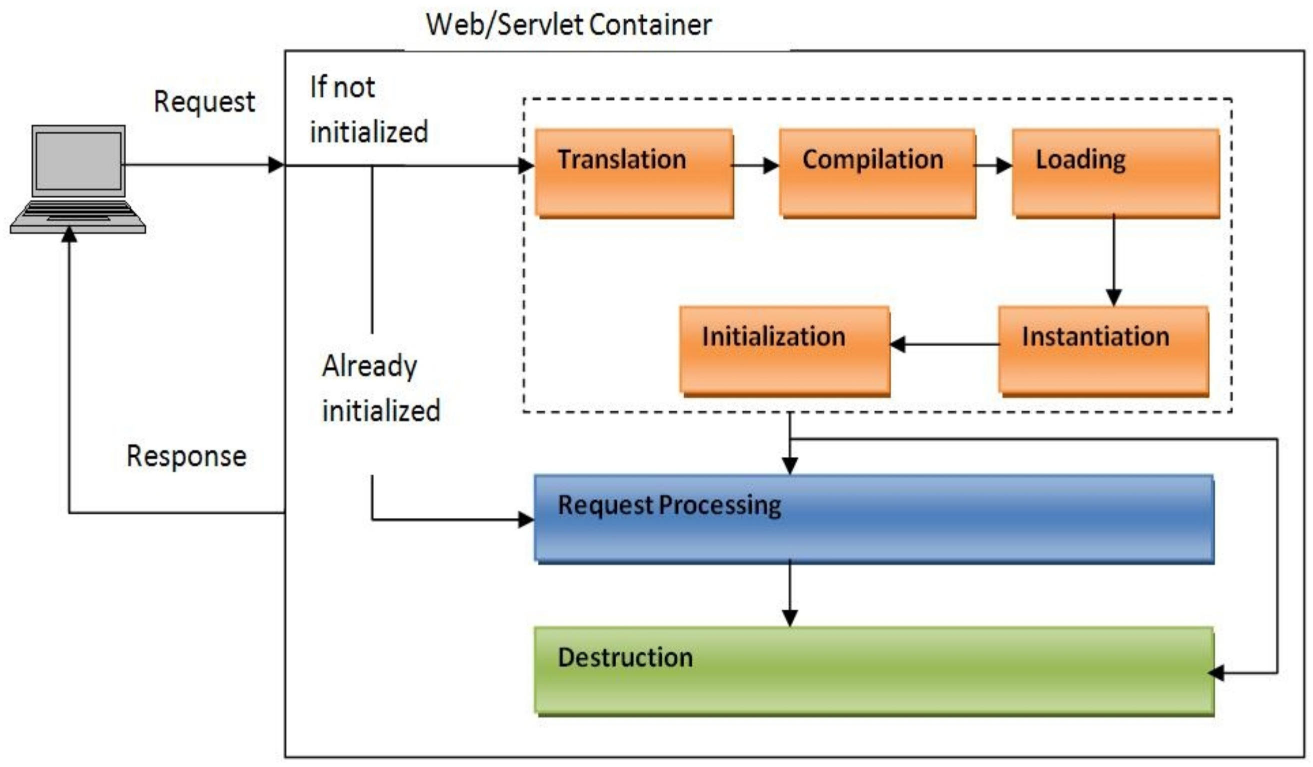
General syntax of log method of Application object is as follows:

```
application.log(Message);
```

# JSP Life Cycle



## Other View


Web/Servlet Container
Request
If not initialized
Translation
Compilation
Loading
Instantiation
Initialization
Already initialized
Response
Request Processing
Destruction