JournalDev

# JSP Tutorial

JAVA SERVER PAGES

Pankaj Kumar

# Table of Content

# 1. Getting Started with JSP

In this section, we will look into the basics of JSP, advantages of JSP over Servlets, Life Cycle of JSP, JSP API interfaces and Classes and where can we put JSP files in the web application.

We will also look into the JSP Comments, Scriptlets, Directives, Expression, Declaration and JSP attributes in brief detail. You should have good understanding of these topics to get a hold of Java Server Pages technology.

## A. What is JSP?

**JSP** (Java Server Pages) is server side technology to create dynamic java web application. JSP can be thought as an extension to servlet technology because it provides features to easily create user views.

JSP Page consists of HTML code and provide option to include java code for dynamic content. Since web applications contain a lot of user screens, JSPs are used a lot in web applications. To bridge the gap between java code and HTML in JSP, it provides additional features such as JSP Tags, Expression Language, Custom tags. This makes it easy to understand and helps a web developer to quickly develop JSP pages.

## B. Advantages of JSP over Servlets

- We can generate HTML response from servlets also but the process is cumbersome and error prone, when it comes to writing a complex HTML response, writing in a servlet will be a nightmare. JSP helps in this situation and provide us flexibility to write normal HTML page and include our java code only where it's required.
- JSP provides additional features such as tag libraries, expression language, custom tags that helps in faster development of user views.
- JSP pages are easy to deploy, we just need to replace the modified page in the server and container takes care of the deployment. For servlets, we need to recompile and deploy whole project again.

Actually Servlet and JSPs complement each other. We should use Servlet as server side controller and to communicate with model classes whereas JSPs should be used for presentation layer.

## C. Life cycle of JSP Page

JSP life cycle is also managed by container. Usually every web container that contains servlet container also contains JSP container for managing JSP pages.

JSP pages life cycle phases are:

- **Translation** – JSP pages doesn't look like normal java classes, actually JSP container parse the JSP pages and translate them to generate corresponding servlet source code. If JSP file name is home.jsp, usually it's named as home_jsp.java.
- **Compilation** – If the translation is successful, then container compiles the generated servlet source file to generate class file.
- **Class Loading** – Once JSP is compiled as servlet class, its lifecycle is similar to servlet and it gets loaded into memory.
- **Instance Creation** – After JSP class is loaded into memory, its object is instantiated by the container.

- **Initialization** – The JSP class is then initialized and it transforms from a normal class to servlet. After initialization, ServletConfig and ServletContext objects become accessible to JSP class.
- **Request Processing** – For every client request, a new thread is spawned with ServletRequest and ServletResponse to process and generate the HTML response.
- **Destroy** – Last phase of JSP life cycle where it's unloaded into memory.

## D. Life cycle methods of JSP

JSP lifecycle methods are:

1. **jspInit()** declared in JspPage interface. This method is called only once in JSP lifecycle to initialize config params.
2. **_jspService(HttpServletRequest request, HttpServletResponse response)** declared in HttpJspPage interface and response for handling client requests.
3. **jspDestroy()** declared in JspPage interface to unload the JSP from memory.

## E. Simple JSP Example with Eclipse and Tomcat

We can use Eclipse IDE for building dynamic web project with JSPs and use Tomcat to run it. Please refer Java Web Applications article to learn how we can easily create JSPs in Eclipse and run it in tomcat.

A simple JSP page example is:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>First JSP</title>
```

```
</head>
<%@ page import="java.util.Date" %>
<body>
<h3>Hi Pankaj</h3><br>
<strong>Current Time is</strong>: <%=new Date() %>

</body>
</html>
```

If you have a simple JSP that uses only JRE classes, you are not required to put it as WAR file. Just create a directory in the tomcat webapps folder and place your JSP file in the newly created directory. For example, if your JSP is located at *apache-tomcat/webapps/test/home.jsp*, then you can access it in browser with URL *http://localhost:8080/test/home.jsp*. If your host and port is different, then you need to make changes in URL accordingly.

## F. JSP Files location in Web Application WAR File

We can place JSP files at any location in the WAR file, however if we put it inside the WEB-INF directory, we won't be able to access it directly from client.

We can configure JSP just like servlets in web.xml, for example if I have a JSP page like below inside WEB-INF directory:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Test JSP</title>
</head>
<body>
Test JSP Page inside WEB-INF folder.<br>
Init Param "test" value =<%=config.getInitParameter("test") %><br>
HashCode of this object=<%=this.hashCode() %>
```

```
</body>
</html>
```

And I configure it in web.xml configuration as:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>FirstJSP</display-name>

  <servlet>
  <servlet-name>Test</servlet-name>
  <jsp-file>/WEB-INF/test.jsp</jsp-file>
  <init-param>
    <param-name>test</param-name>
    <param-value>Test Value</param-value>
  </init-param>
  </servlet>

  <servlet-mapping>
  <servlet-name>Test</servlet-name>
  <url-pattern>/Test.do</url-pattern>
  </servlet-mapping>

  <servlet>
  <servlet-name>Test1</servlet-name>
  <jsp-file>/WEB-INF/test.jsp</jsp-file>
  </servlet>

  <servlet-mapping>
  <servlet-name>Test1</servlet-name>
  <url-pattern>/Test1.do</url-pattern>
  </servlet-mapping>
</web-app>
```

Then I can access it with both the URLs
*http://localhost:8080/FirstJSP/Test.do* and
*http://localhost:8080/FirstJSP/Test1.do*

Notice that container will create two instances in this case and both will have their own servlet config objects, you can confirm this by visiting these URLs in browser.

For Test.do URI, you will get response like below.

```
Test JSP Page inside WEB-INF folder.
```

```
Init Param "test" value =Test Value
HashCode of this object=1839060256
```

For Test1.do URI, you will get response like below.

```
Test JSP Page inside WEB-INF folder.
Init Param "test" value =null
HashCode of this object=38139054
```

Notice the init param value in second case is null because it's not defined for the second servlet, also notice the hashcode is different. If you will make further requests, the hashcode value will not change because the requests are processed by spawning a new thread by the container.

Did you noticed the use of **config** variable in above JSP but there is no variable declared, it's because it's one of the 9 implicit objects available in JSP page, read more about them at **JSP Implicit Objects** article.

## G. JSP API Interfaces and Classes

All the core JSP interfaces and classes are defined in *javax.servlet.jsp* package. Expression Language API interfaces are classes are part of *javax.servlet.jsp.el* package. JSP Tag Libraries interfaces and classes are defined in *javax.servlet.jsp.tagext* package.

Here we will look into interfaces and classes of Core JSP API.

- **JspPage Interface**

JspPage interface extends Servlet interface and declares jspInit() and jspDestroy() life cycle methods of the JSP pages.

- **HttpJspPage Interface**

HttpJspPage interface describes the interaction that a JSP Page Implementation Class must satisfy when using the HTTP protocol. This interface declares the service method of JSP page for HTTP protocol as *public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException*.

- **JspWriter abstract Class**

Similar to PrintWriter in servlets with additional facility of buffering support. This is one of the implicit variables in a JSP page with name “out”. This class extends java.io.Writer and container provide their own implementation for this abstract class and use it while translating JSP page to Servlet. We can get it’s object using PageContext.getOut() method. Apache Tomcat concrete class for JspWriter is org.apache.jasper.runtime.JspWriterImpl.

- **JspContext abstract Class**

JspContext serves as the base class for the PageContext class and abstracts all information that is not specific to servlets. The JspContext provides mechanism to obtain the JspWriter for output, mechanism to work with attributes and API to manage the various scoped namespaces.

- **PageContext abstract Class**

PageContext extends JspContext to provide useful context information when JSP is used for web applications. A PageContext instance provides access to all the namespaces associated with a JSP page, provides access to several page attributes, as well as a layer above the implementation details. Implicit objects are added to the pageContext automatically.

- **JspFactory abstract Class**

The JspFactory is an abstract class that defines a number of factory methods available to a JSP page at runtime for the purposes of creating instances of various interfaces and classes used to support the JSP implementation.

- **JspEngineInfo abstract Class**

The JspEngineInfo is an abstract class that provides information on the current JSP engine.

- **ErrorData final Class**

Contains information about an error, for error pages.

- **JspException Class**

A generic exception known to the JSP container, similar to ServletException. If JSP pages throw JspException then error page mechanism is used to present error information to user.

- **JspTagException Class**

Exception to be used by a Tag Handler to indicate some unrecoverable error.

- **SkipPageException Class**

Exception to indicate the calling page must cease evaluation. Thrown by a simple tag handler to indicate that the remainder of the page must not be evaluated. This exception should not be thrown manually in a JSP page.

## H. JSP Comments

Since JSP is built on top of HTML, we can write comments in JSP file like html comments as

**<-- This is HTML Comment -->**

These comments are sent to the client and we can look it with view source option of browsers.

We can put comments in JSP files as:

**<%-- This is JSP Comment -- %>**

This comment is suitable for developers to provide code level comments because these are not sent in the client response.

## I. JSP Scriptlets

Scriptlet tags are the easiest way to put java code in a JSP page. A scriptlet tag starts with <% and ends with %>.

Any code written inside the scriptlet tags go into the ***_jspService()*** method.

**For example:**

```
<%
Date d = new Date();
System.out.println("Current Date="+d);
%>
```

## J. JSP Expression

Since most of the times we print dynamic data in JSP page using ***out.print()*** method, there is a shortcut to do this through JSP Expressions. JSP Expression starts with **<%=** and ends with **%>.**

`<% out.print("Pankaj"); %> can be written using JSP Expression as <%= "Pankaj" %>`

Notice that anything between **<%= %>** is sent as parameter to ***out.print()*** method. Also notice that scriptlets can contain multiple java statements and always ends with semicolon (;) but expression doesn't end with semicolon.

## K. JSP Directives

JSP Directives are used to give special instructions to the container while JSP page is getting translated to servlet source code. JSP directives starts with **<%@** and ends with **%>**

For example, in above JSP Example, I am using ***page*** directive to to instruct container JSP translator to import the Date class.

## L. JSP Declaration

JSP Declarations are used to declare member methods and variables of servlet class. JSP Declarations starts **with <%!** and ends with **%>**.

For example we can create an int variable in JSP at class level as ***<%! public static int count=0; %>***

## M. JSP transformed Servlet Source Code and Class File location in Tomcat

Once JSP files are translated to Servlet source code, the source code (.java) and compiled classes both are place in **Tomcat/work/Catalina/localhost/FirstJSP/org/apache/jsp** directory. If the JSP files are inside other directories of application, the directory structure is maintained.

For JSPs inside WEB-INF directory, its source and class files are inside **Tomcat/work/Catalina/localhost/FirstJSP/org/apache/jsp/WEB_002dINF** directory.

Here is the source code generated for above test.jsp page.

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/7.0.32
 * Generated at: 2015-04-21 03:40:59 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp.WEB_002dINF;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class test_jsp extends
org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final javax.servlet.jsp.JspFactory _jspxFactory =
          javax.servlet.jsp.JspFactory.getDefaultFactory();

  private static java.util.Map<java.lang.String,java.lang.Long>
_jspx_dependants;

  private javax.el.ExpressionFactory _el_expressionfactory;
  private org.apache.tomcat.InstanceManager _jsp_instancemanager;
  public java.util.Map<java.lang.String,java.lang.Long> getDependants()
{
    return _jspx_dependants;
  }

  public void _jspInit() {
    _el_expressionfactory =
_jspxFactory.getJspApplicationContext(getServletConfig().getServletCont
ext()).getExpressionFactory();
    _jsp_instancemanager =
org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(get
ServletConfig());
  }

  public void _jspDestroy() {
  }

  public void _jspService(final javax.servlet.http.HttpServletRequest
request, final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html; charset=US-ASCII");
      pageContext = _jspxFactory.getPageContext(this, request,
response,
                null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("\n");
      out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\" \"http://www.w3.org/TR/html4/loose.dtd\">\n");
      out.write("<html>\n");
      out.write("<head>\n");
      out.write("<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=US-ASCII\">\n");
      out.write("<title>Test JSP</title>\n");
      out.write("</head>\n");
      out.write("<body>\n");
      out.write("Test JSP Page inside WEB-INF folder.<br>\n");
      out.write("Init Param \"test\" value =");
      out.print(config.getInitParameter("test") );
      out.write("<br>\n");
      out.write("HashCode of this object=");
      out.print(this.hashCode() );
      out.write("\n");
      out.write("</body>\n");
      out.write("</html>");
    } catch (java.lang.Throwable t) {
      if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          try { out.clearBuffer(); } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

Notice following points in above servlet code;

- The package of class starts with org.apache.jsp and if JSPs are inside other folders, it includes directory hierarchy too. Usually we don’t care about it.
- The generated servlet class is final and can’t be extended.
- It extends ***org.apache.jasper.runtime.HttpJspBase*** that is similar to HttpServlet except that it’s internal to Tomcat JSP Translator implementation. HttpJspBase extends HttpServlet and implements HttpJspPage interface.
- Notice the local variables at the start of _jspService() method implementation, they are automatically added by JSP translator and available for use in service methods, i.e. in scriptlets.

As a java programmer, sometimes it helps to look into the generated source for debugging purposes.

## N. JSP init parameters

We can define init parameters for the JSP page as shown in above example and we can retrieve them in JSP using **config** implicit object, we will look into implicit objects in JSP in more detail in future posts.

## O. Overriding JSP init() method

We can override JSP init method for creating resources to be used by JSP **service()** method using JSP Declaration tags, we can override **jspInit()** and **jspDestroy()** or any other methods also. However we should never override **_jspService()** method because anything we write in JSP goes into service method.

## P. Attributes in a JSP

Apart from standard servlet attributes with request, session and context scope, in JSP we have another scope for attributes, i.e. Page Scope that we can get from *pageContext* object. We will look its importance in custom tags tutorial. For normal JSP programming, we don't need to worry about page scope.

That's all for getting started with JSP technology, I hope it will help you in understanding the basic concepts of JSPs and help you in getting started.

# 2. JSP Implicit Objects

**JSP implicit objects** are created by container while translating JSP page to Servlet source to help developers. We can use these objects directly in **scriptlets** that goes in service method, however we can't use them in JSP Declaration because that code will go at class level.

We have 9 implicit objects that we can directly use in JSP page. Seven of them are declared as local variable at the start of ***_jspService()*** method whereas two of them are part of ***_jspService()*** method argument that we can use.

## A. out Object

JSP out implicit object is instance of javax.servlet.jsp.JspWriter implementation and it's used to output content to be sent in client response. This is one of the most used JSP implicit object and that's why we have JSP Expression to easily invoke ***out.print()*** method.

## B. request Object

JSP request implicit object is instance of ***javax.servlet.http.HttpServletRequest*** implementation and it's one of the argument of JSP service method. We can use request object to get the request parameters, cookies, request attributes, session, header information and other details about client request.

## C. response Object

JSP response implicit object is instance of ***javax.servlet.http.HttpServletResponse*** implementation and comes as argument of service method. We can response object to set content type, character encoding, header information in response, adding cookies to response and redirecting the request to other resource.

## D. config Object

JSP config implicit object is instance of ***javax.servlet.ServletConfig*** implementation and used to get the JSP init params configured in deployment descriptor.

## E. application Object

JSP application implicit object is instance of ***javax.servlet.ServletContext*** implementation and it's used to get the context information and attributes in JSP. We can use it to get the RequestDispatcher object in JSP to forward the request to another resource or to include the response from another resource in the JSP.

## F. session Object

JSP session implicit object is instance of ***javax.servlet.http.HttpSession*** implementation. Whenever we request a JSP page, container automatically creates a session for the JSP in the service method.

Since session management is heavy process, so if we don't want session to be created for JSP, we can use page directive to not create the session for JSP using **<%@** page session="false" **%>.** This is very helpful when our login page or the index page is a JSP page and we don't need any user session there.

## G. pageContext Object

JSP pageContext implicit object is instance of ***javax.servlet.jsp.PageContext*** abstract class implementation. We can use pageContext to get and set attributes with different scopes and to forward request to other resources. pageContext object also hold reference to other implicit object.

## H. page Object

JSP page implicit object is instance of ***java.lang.Object*** class and represents the current JSP page. *page* object provide reference to the generated servlet class. This object is very rarely used.

## I. exception Object

JSP exception implicit object is instance of ***java.lang.Throwable*** class and used to provide exception details in JSP error pages. We can’t use this object in normal JSP pages and it’s available only in JSP error pages.

## J. JSP Implicit Objects Example

Here is a simple JSP page that’s showing usage of JSP implicit objects.

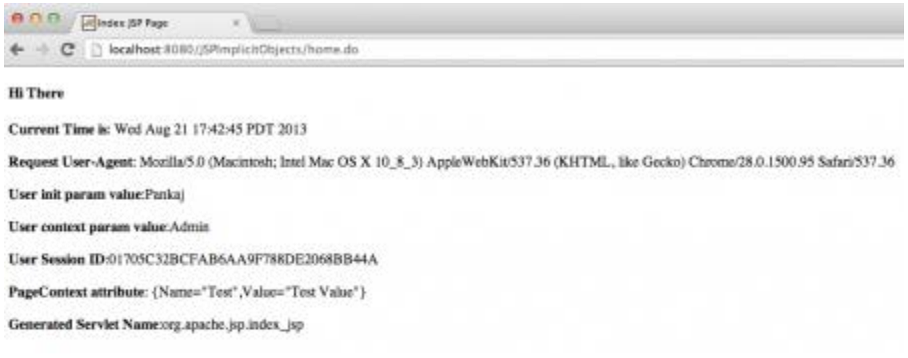
```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<%@ page import="java.util.Date" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Index JSP Page</title>
</head>
<body>
<%-- out object example --%>
<h4>Hi There</h4>
<strong>Current Time is</strong>: <% out.print(new Date()); %><br><br>

<%-- request object example --%>
<strong>Request User-Agent</strong>: <%=request.getHeader("User-Agent")
%><br><br>

<%-- response object example --%>
<%response.addCookie(new Cookie("Test","Value")); %>

<%-- config object example --%>
```

```
<strong>User init param
value</strong>:<%=config.getInitParameter("User") %><br><br>

<%-- application object example --%>
<strong>User context param
value</strong>:<%=application.getInitParameter("User") %><br><br>

<%-- session object example --%>
<strong>User Session ID</strong>:<%=session.getId() %><br><br>

<%-- pageContext object example --%>
<% pageContext.setAttribute("Test", "Test Value"); %>
<strong>PageContext attribute</strong>:
{Name="Test",Value="<%=pageContext.getAttribute("Test") %>"}<br><br>

<%-- page object example --%>
<strong>Generated Servlet Name</strong>:<%=page.getClass().getName() %>

</body>
</html>
```

Deployment descriptor for the dynamic web project is:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>JSPImplicitObjects</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <context-param>
  <param-name>User</param-name>
  <param-value>Admin</param-value>
  </context-param>

  <servlet>
  <servlet-name>home</servlet-name>
  <jsp-file>/index.jsp</jsp-file>
  <init-param>
    <param-name>User</param-name>
    <param-value>Pankaj</param-value>
  </init-param>
  </servlet>
  <servlet-mapping>
  <servlet-name>home</servlet-name>
  <url-pattern>/home.do</url-pattern>
  <url-pattern>/home.jsp</url-pattern>
  </servlet-mapping>
</web-app>
```

When I access above JSP in browser, I get response like below image.



**Hi There**

**Current Time is:** Wed Aug 21 17:42:45 PDT 2013

**Request User-Agent:** Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/28.0.1500.95 Safari/537.36

**User init param value:**Pankaj

**User context param value:**Admin

**User Session ID:**01705C32BCFAB6AA9F788DE2068BB44A

**PageContext attribute:** {Name="Test",Value="Test Value"}

**Generated Servlet Name:**org.apache.jsp.index_jsp

Let’s look at the generated servlet class code for better understanding about these objects.

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/7.0.32
 * Generated at: 2013-08-22 00:42:24 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.util.Date;

public final class index_jsp extends
org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final javax.servlet.jsp.JspFactory _jspxFactory =
          javax.servlet.jsp.JspFactory.getDefaultFactory();

  private static java.util.Map<java.lang.String,java.lang.Long>
_jspx_dependants;

  private javax.el.ExpressionFactory _el_expressionfactory;
  private org.apache.tomcat.InstanceManager _jsp_instancemanager;

  public java.util.Map<java.lang.String,java.lang.Long> getDependants()
{
    return _jspx_dependants;
  }

  public void _jspInit() {
    _el_expressionfactory =
_jspxFactory.getJspApplicationContext(getServletConfig().getServletCont
ext()).getExpressionFactory();
    _jsp_instancemanager =
org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(get
ServletConfig());
  }

  public void _jspDestroy() {
  }

  public void _jspService(final javax.servlet.http.HttpServletRequest
request, final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html; charset=US-ASCII");
      pageContext = _jspxFactory.getPageContext(this, request,
response,
                  null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("\n");
      out.write("\n");
      out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\" \"http://www.w3.org/TR/html4/loose.dtd\">\n");
      out.write("<html>\n");
      out.write("<head>\n");
      out.write("<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=US-ASCII\">\n");
      out.write("<title>Index JSP Page</title>\n");
      out.write("</head>\n");
      out.write("<body>\n");
      out.write("\n");
      out.write("<h4>Hi There</h4>\n");
      out.write("<strong>Current Time is</strong>: ");
 out.print(new Date());
      out.write("<br><br>\n");
      out.write("\n");
      out.write("\n");
      out.write("<strong>Request User-Agent</strong>: ");
      out.print(request.getHeader("User-Agent") );
      out.write("<br><br>\n");
      out.write("\n");
      out.write('\n');
response.addCookie(new Cookie("Test","Value"));
      out.write('\n');
      out.write('\n');
      out.write("\n");
      out.write("<strong>User init param value</strong>:");
      out.print(config.getInitParameter("User") );
      out.write("<br><br>\n");
      out.write("\n");
      out.write("\n");
      out.write("<strong>User context param value</strong>:");
      out.print(application.getInitParameter("User") );
      out.write("<br><br>\n");
      out.write("\n");
      out.write("\n");
      out.write("<strong>User Session ID</strong>:");
      out.print(session.getId() );
      out.write("<br><br>\n");
      out.write("\n");
      out.write('\n');
 pageContext.setAttribute("Test", "Test Value");
      out.write("\n");
      out.write("<strong>PageContext attribute</strong>:
{Name=\"Test\",Value=\"");
      out.print(pageContext.getAttribute("Test") );
      out.write("\"}<br><br>\n");
      out.write("\n");
      out.write("\n");
      out.write("<strong>Generated Servlet Name</strong>:");
      out.print(page.getClass().getName() );
      out.write("\n");
      out.write("\n");
      out.write("\n");
      out.write("</body>\n");
      out.write("</html>");
    } catch (java.lang.Throwable t) {
      if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          try { out.clearBuffer(); } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

That’s all for JSP implicit objects, take some time to go through with each of these objects because they are used a lot in JSP programming.

# 3. JSP Directives

JSP Directives are used to give special instruction to container for translation of JSP to Servlet code. JSP Directives are placed between <%@ %>.

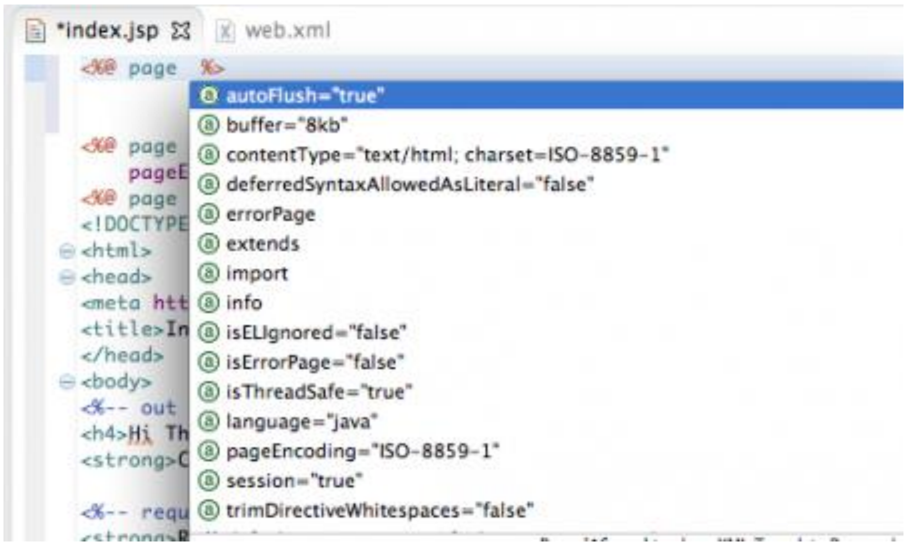
JSP provides three directives for us to use;

1. page directive
2. include directive
3. taglib directive

Every directive has a set of attributes to provide specific type of instructions. So usually a JSP directive will look like <%@ directive attribute="value" %>.

## A. JSP page directive

**page** directive provide attributes that gets applied to the entire JSP page. **page** directive has a lot of attributes that we will look now. We can define multiple attributes in a single page directive or we can have multiple page directives in a single JSP page.



```
*index.jsp   web.xml
<%@ page  %>
<%@ page
    pageE
<%@ page
<!DOCTYPE
<html>
<head>
<meta htt
<title>In
</head>
<body>
<%-- out
<h4>Hi Th
<strong>C

<%-- requ
```

```
autoFlush="true"
buffer="8kb"
contentType="text/html; charset=ISO-8859-1"
deferredSyntaxAllowedAsLiteral="false"
errorPage
extends
import
info
isELIgnored="false"
isErrorPage="false"
isThreadSafe="true"
language="java"
pageEncoding="ISO-8859-1"
session="true"
trimDirectiveWhitespaces="false"
```

1. **import attribute**: This is one of the most used page directive attribute. It’s used to instruct container to import other java classes, interfaces, enums etc. while generating servlet code. This is similar to import statements in java classes, interfaces. An example of import page directive usage is:

   ```
   <%@ page import="java.util.Date,java.util.List,java.io.*" %>
   ```

2. **contentType attribute**: This attribute is used to set the content type and character set of the response. The default value of contentType attribute is”text/html; charset=ISO-8859-1″. We can use it like below.

   ```
   <%@ page contentType="text/html; charset=US-ASCII" %>
   ```

3. **pageEncoding attribute**: We can set response encoding type with this page directive attribute, its default value is “ISO-8859-1″.

   ```
   <%@ page pageEncoding="US-ASCII" %>
   ```

4. **extends attribute**: This attribute is used to define the super class of the generated servlet code. This is very rarely used and we can use it if we have extended HttpServlet and overridden some of it’s implementations. For example;

   ```
   <%@ page extends="org.apache.jasper.runtime.HttpJspBase" %>
   ```

5. **info attribute**: We can use this attribute to set the servlet description and we can retrieve it using Servlet interface getServletInfo() method. For example;

   ```
   <%@ page info="Home Page JSP" %>
   ```

6. **buffer attribute**: We know that JspWriter has buffering capabilities, we can use this attribute to set the buffer size in KB to handle output generated by JSP page. Default value of buffer attribute is 8kb. We can define 16 KB buffer size as;

   ```
   <%@ page buffer="16kb" %>
   ```

7. **language attribute**: language attribute is added to specify the scripting language used in JSP page. It’s default value is “java” and this is the

only value it can have. May be in future, JSPs provide support to include other scripting languages like C++ or PHP too.

<%@ page language="java" %>

8. **isELIgnored attribute**: We can ignore the Expression Language (EL) in JSP using this page directive attribute. Its datatype is **Java Enum** and default value is *false*, so EL is enabled by default. We can instruct container to ignore EL using below directive;

   <%@ page isELIgnored="true" %>

9. **isThreadSafe attribute**: We can use this attribute to implement SingleThreadModel interface in generated servlet. It's an Enum with default value as true. If we set its value to false, the generated servlet will implement SingleThreadModel and eventually we will lose all the benefits of servlet **multi-threading** features. You should never set its value to false.

   <%@ page isThreadSafe="false" %>

10. **errorPage attribute**: This attribute is used to set the error page for the JSP, if the JSP throws exception, the request is redirected to the error handler defined in this attribute. Its datatype is URI. For example;

    <%@ page errorPage="errorHandler.jsp" %>

11. **isErrorPage attribute**: This attribute is used to declare that current JSP page is an error page. It's of type Enum and default value is false. If we are creating an error handler JSP page for our application, we have to use this attribute to let container know that it's an error page. JSP implicit attribute exception is available only to the error page JSPs. For example;

    <%@ page isErrorPage="true" %>

12. **autoFlush attribute**: autoFlush attribute is to control the buffer output. Its default value is true and output is flushed automatically when buffer is full. If we set it to false, the buffer will not be flushed automatically and if it's full, we will get exception for buffer overflow. We can use

this attribute when we want to make sure that JSP response is sent in full or none. For example;

<%@ page autoFlush="false" %>

13. **session attribute**: By default JSP page creates a session but sometimes we don't need session in JSP page. We can use this attribute to indicate compiler to not create session by default. Its default value is true and session is created. To disable the session creation, we can use it like below.

    <%@ page session ="false" %>

14. **trimDirectiveWhitespaces attribute**: This attribute was added in JSP 2.1 and used to strip out extra white spaces from JSP page output. Its default value is false. It helps in reducing the generated code size, notice the generate servlet code keeping this attribute value as true and false. You will notice no out.write("\n") when it's true.

    <%@ page trimDirectiveWhitespaces ="true" %>

## B. JSP include directive

JSP include directive is used to include the contents of another file to the current JSP page. The included file can be HTML, JSP, text files etc. Include directive is very helpful in creating templates for user views and break the pages into header, footer, sidebar sections.

We can include any resource in the JSP page like below.

<%@ include file="test.html" %>

The file attribute value should be the relative URI of the resource from the current JSP page.

## C. JSP taglib directive

JSP taglib directive is used to define a tag library with prefix that we can use in JSP, we will look into more details in JSP Custom Tags tutorial.

We can define JSP tag libraries in like below;

```
<%@ taglib uri="/WEB-INF/c.tld" prefix="c"%>
```

That's all for JSP directives, taglibs are used a lot in JSP programming, so you should spend some time to understand it programmatically.

# 4. JSP Expression Language (EL)

Most of the times we use JSP for view purposes and all the business logic is present in servlet code or model classes. When we receive client request in servlet, we process it and then add attributes in request/session/context scope to be retrieved in JSP code. We also use request params, headers, cookies and init params in JSP to create response views.

We saw in earlier section, how we can use **scriptlets** and **JSP expressions** to retrieve attributes and parameters in JSP with java code and use it for view purpose. But for web designers, java code is hard to understand and that's why JSP Specs 2.0 introduced **Expression Language** (EL) through which we can get attributes and parameters easily using HTML like tags.

Expression language syntax is ${name} and we will see how we can use them in JSP code.

## A. JSP EL Implicit Objects

JSP Expression Language provides many implicit objects that we can use to get attributes from different scopes and parameter values. The list is given below.

| JSP EL Implicit Objects | Type | Description |
|---|---|---|
| pageScope | Map | A map that contains the attributes set with page scope. |
| requestScope | Map | Used to get the attribute value with request scope. |
| sessionScope | Map | Used to get the attribute value with session scope. |
| applicationScope | Map | Used to get the attributes value from application scope. |
| Param | Map | Used to get the request parameter value, returns a single value |

|  |  |  |
|---|---|---|
| paramValues | Map | Used to get the request param values in an array, useful when request parameter contain multiple values. |
| Header | Map | Used to get request header information |
| headerValues | Map | Used to get header values in an array. |
| Cookie | Map | Used to get the cookie value in the JSP |
| initParam | Map | Used to get the context init params, we can't use it for servlet init params |
| pageContext | Map | Same as JSP implicit pageContext object, used to get the request, session references etc. example usage is getting request HTTP Method name. |

Note that these implicit objects are different from **JSP implicit objects** and can be used only with JSP EL.

## B. JSP EL Operators

Let's look at EL Operators and understand how they are interpreted and how to use them.

1. **EL Property Access Operator or Dot (.) Operator**

   JSP EL Dot operator is used to get the attribute values.

   ${firstObj.secondObj}

   In above expression, firstObj can be EL implicit object or an attribute in page, request, session or application scope. For example,

   ${requestScope.employee.address}

   Note that except the last part of the EL, all the objects should be either Map or Java Bean, so in above example requestScope is a Map and employee should be a Java Bean or Map. If scope is not provided, the JSP EL looks into page, request, session and application scope to find the named attribute.

2. **JSP EL [] Operator or Collection Access Operator**

   [] operator is more powerful than dot operator. We can use it to get data from List and Array too.

   Some examples;

   ${myList[1]} and ${myList["1″]} are same, we can provide List or Array index as String literal also.

   ${myMap[expr]} – if the parameter inside [] is not String, it's evaluated as an EL.

   ${myMap[myList[1]]} – [] can be nested.

   ${requestScope["foo.bar"]} – we can't use dot operator when attribute names have dots.

3. **JSP EL Arithmetic Operators**

   Arithmetic operators are provided for simple calculations in EL expressions. They are +, -, *, / or div, % or mod.

4. **JSP EL Logical Operators**

   They are && (and), || (or) and ! (not).

5. **JSP EL Relational Operators**

   They are == (eq), != (ne), < (lt), > (gt), <= (le) and >= (ge).

## C. JSP EL Operator Precedence

JSP EL expressions are evaluated from left to right. JSP EL Operator precedence is listed in below table from highest to lowest.

| **JSP EL Operator Precedence from Highest to Lowest** |
|---|
| [ ] . |
| () – Used to change the precedence of operators. |
| – (unary) not ! empty |
| * / div % mod |
| + – (binary) |
| < > <= >= lt gt le ge |
| == != eq ne |
| && and |
| \|\| or |
| ? : |

## D. JSP EL Reserve Words

| | | | | |
|---|---|---|---|---|
| and | or | not | eq | ne |
| lt | gt | le | ge | true |
| false | null | instanceof | empty | div,mod |

Above are the reserved words, don't use them as identifier in JSPs.

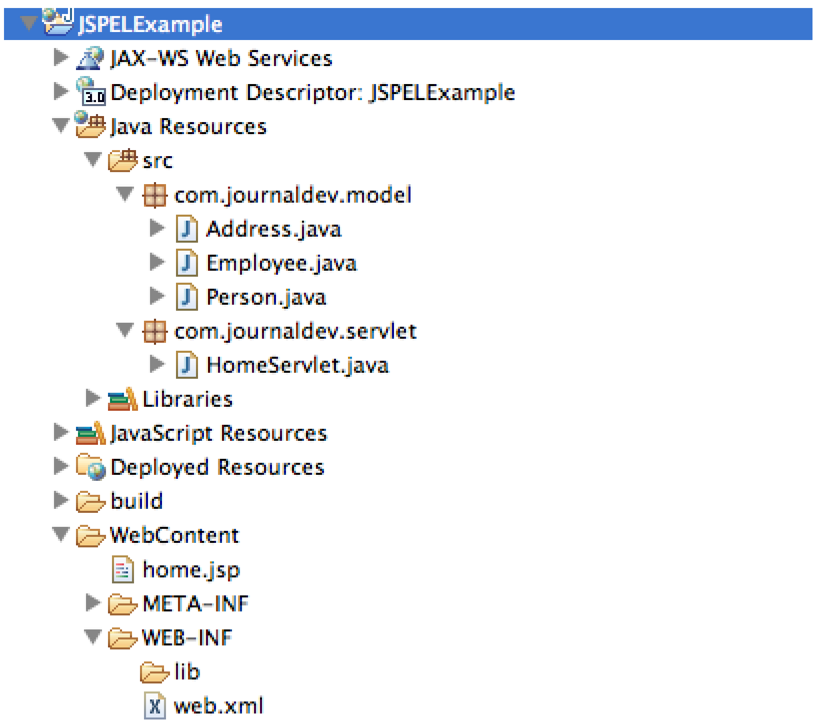
## E. JSP EL Important Points

1. EL expressions are always within curly braces prefixed with $ sign, for example ${expr}
2. We can disable EL expression in JSP by setting **JSP page directive** isELIgnored attribute value to TRUE.

3. JSP EL can be used to get attributes, header, cookies, init params etc, but we can't set the values.
4. JSP EL implicit objects are different from JSP implicit objects except pageContext, don't get confused.
5. JSP EL pageContext implicit object is provided to get additional properties from request, response etc, for example getting HTTP request method.
6. JSP EL is NULL friendly, if given attribute is not found or expression returns null, it doesn't throw any exception. For arithmetic operations, EL treats null as 0 and for logical operations, EL treats null as false.
7. The [] operator is more powerful than dot operator because we can access list and array data too, it can be nested and argument to [] is evaluated when it's not string literal.
8. If you are using Tomcat, the EL expressions are evaluated using org.apache.jasper.runtime.PageContextImpl.proprietaryEvaluate() method.
9. We can use EL functions to call method from a java class, more on this in custom tags post in near future.

## F. JSP EL Example

Let's see EL usage with a simple application. We will set some attributes in different scopes and use EL to retrieve them and show in JSP page. Our project structure will be like below image.



I have defined some model classes that we will use – Person interface, Employee implementing Person and Address used in Employee.

```
package com.journaldev.model;

public interface Person {

    public String getName();
    public void setName(String nm);
}
package com.journaldev.model;

public class Address {

    private String address;

    public Address() {
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String toString(){
        return "Address="+address;
    }
}


package com.journaldev.model;

public class Employee implements Person {

    private String name;
    private int id;
    private Address address;

    public Employee() {
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public void setName(String nm) {
        this.name = nm;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public String toString(){
        return "ID="+id+",Name="+name+",Address="+address;
    }

}
```

Notice that Employee and Address are java beans with no-args constructor and getter-setter methods for properties. I have also provided implementation of toString() method that we will use in JSP page.

Now let’s see the code of a simple servlet that will set some attributes.

```
package com.journaldev.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.journaldev.model.Address;
import com.journaldev.model.Employee;
import com.journaldev.model.Person;

@WebServlet("/HomeServlet")
public class HomeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        //Setting some attributes
        Person person = new Employee();
        person.setName("Pankaj");

        request.setAttribute("person", person);

        Employee emp = new Employee();
        Address add = new Address();
        add.setAddress("India");
        emp.setAddress(add);
        emp.setId(1);
        emp.setName("Pankaj Kumar");

        HttpSession session = request.getSession();
        session.setAttribute("employee", emp);

        response.addCookie(new Cookie("User.Cookie","Tomcat User"));
        getServletContext().setAttribute("User.Cookie","Tomcat User");

        RequestDispatcher rd =
getServletContext().getRequestDispatcher("/home.jsp");
```

```
        rd.forward(request, response);
    }

}
```

## Let’s define some context init params in web.xml

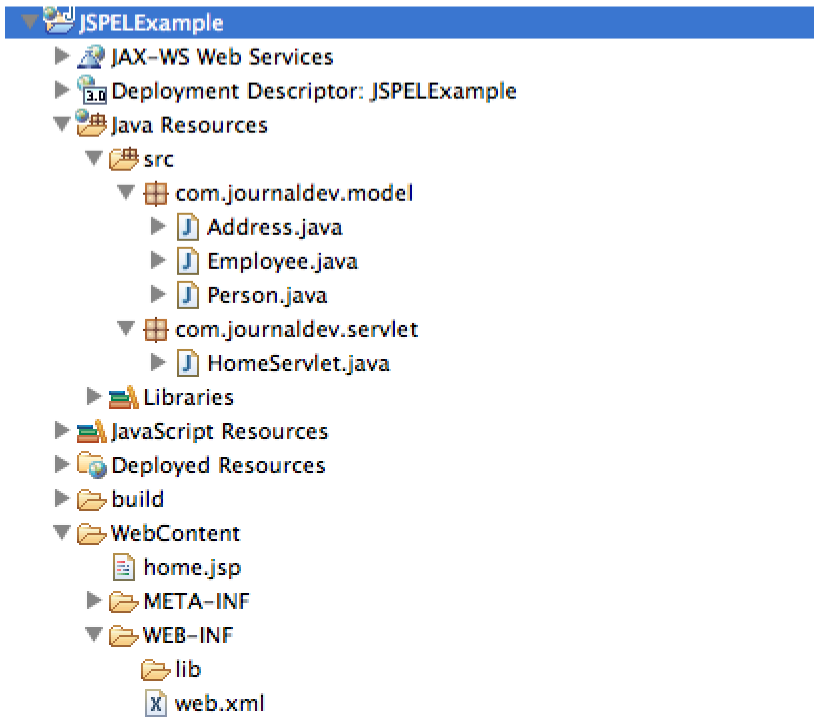
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>JSPELExample</display-name>

  <context-param>
  <param-name>AppID</param-name>
  <param-value>123</param-value>
  </context-param>

</web-app>
```

## JSP code using EL to create views:

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII" import="java.util.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>JSP EL Example Home</title>
</head>
<body>
<%   List<String> names = new ArrayList<String>();
    names.add("Pankaj");names.add("David");
    pageContext.setAttribute("names", names);
%>
<strong>Simple . EL Example:</strong> ${requestScope.person}
<br><br>
<strong>Simple . EL Example without scope:</strong> ${person}
<br><br>
<strong>Simple [] Example:</strong> ${applicationScope["User.Cookie"]}
<br><br>
<strong>Multiples . EL Example:</strong>
${sessionScope.employee.address.address}
<br><br>
<strong>List EL Example:</strong> ${names[1]}
<br><br>
<strong>Header information EL Example:</strong> ${header["Accept-
Encoding"]}
```

```
<br><br>
<strong>Cookie EL Example:</strong> ${cookie["User.Cookie"].value}
<br><br>
<strong>pageContext EL Example:</strong> HTTP Method is
${pageContext.request.method}
<br><br>
<strong>Context param EL Example:</strong> ${initParam.AppID}
<br><br>
<strong>Arithmetic Operator EL Example:</strong> ${initParam.AppID +
200}
<br><br>
<strong>Relational Operator EL Example:</strong> ${initParam.AppID <
200}
<br><br>
<strong>Arithmetic Operator EL Example:</strong> ${initParam.AppID +
200}
<br><br>

</body>
</html>
```

When we send request for above servlet, we get output like below image.



```
JSPELExample
  JAX-WS Web Services
  Deployment Descriptor: JSPELExample
  Java Resources
    src
      com.journaldev.model
        Address.java
        Employee.java
        Person.java
      com.journaldev.servlet
        HomeServlet.java
    Libraries
  JavaScript Resources
  Deployed Resources
  build
  WebContent
    home.jsp
    META-INF
    WEB-INF
      lib
      web.xml
```

That’s all for JSP Expression Language, we will look into EL Functions in custom tags post. We will look into JSP standard action tags in next post.

# 5. JSTL Tutorial with Examples – JSTL Core Tags

Earlier we saw how we can use JSP EL to write JSP code like HTML but their functionality is very limited. For example, we can't loop through a collection using EL or action elements and we can't escape HTML tags to show them like text in client side.

**JSP Standard Tag Library** (**JSTL**) is the standard tag library that provides tags to control the JSP page behavior, iteration and control statements, internationalization tags, and SQL tags.

**JSTL** is part of the Java EE API and included in most servlet containers. But to use JSTL in our JSP pages, we need to download the JSTL jars for your servlet container. Most of the times, you can find them in the example projects and you can use them. You need to include these libraries in the project **WEB-INF/lib** directory. These jars are container specific, for example in Tomcat, we need to include jstl.jar and standard.jar jar files in project build path. If they are not present in the container lib directory, you should include them into your application. If you have maven project, below dependencies should be added in pom.xml file or else you will get following error in JSP pages –

**eclipse Cannot find the tag library descriptor for "http://java.sun.com/jsp/jstl/ core"**

```
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
</dependency>
```

Based on the JSTL functions, they are categorized into five types.

1. **Core Tags**: Core tags provide support for iteration, conditional logic, catch exception, url, forward or redirect response etc. To use JSTL core tags, we should include it in the JSP page like below.

   <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

   In this article, we will look into important JSTL core tags.

2. **Formatting and Localization Tags**: These tags are provided for formatting of Numbers, Dates and i18n support through locales and resource bundles. We can include these tags in JSP with below syntax:

   <%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

3. **SQL Tags**: JSTL SQL Tags provide support for interaction with relational databases such as Oracle, MySql etc. Using SQL tags we can run database queries, we include it in JSP with below syntax:

   <%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>

4. **XML Tags**: XML tags are used to work with XML documents such as parsing XML, transforming XML data and XPath expressions evaluation. Syntax to include XML tags in JSP page is:

   <%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>

5. **JSTL Functions Tags**: JSTL tags provide a number of functions that we can use to perform common operation, most of them are for String manipulation such as String Concatenation, Split String etc. Syntax to include JSTL functions in JSP page is:

   <%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

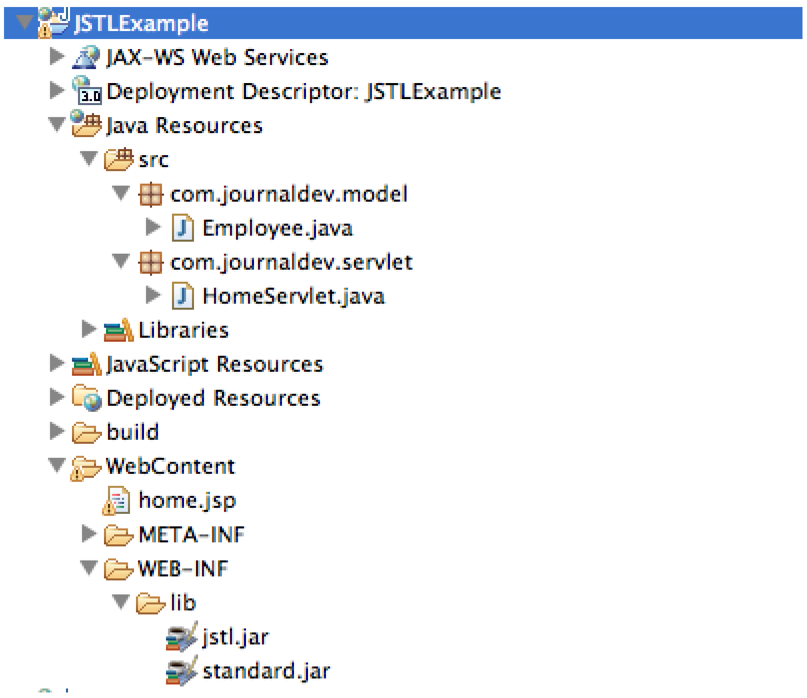
Note that all the JSTL standard tags URI starts with **http://java.sun.com/jsp/jstl/** and we can use any prefix we want but it’s best practice to use the prefix defined above because everybody uses them, so it will not create any confusion.

# A. JSTL Core Tags

JSTL Core Tags are listed in the below table.

| Tag | Description |
|---|---|
| <c:out> | To write something in JSP page, we can use EL also with this tag |
| <c:import> | Same as <jsp:include> or include directive |
| <c:redirect> | redirect request to another resource |
| <c:set> | To set the variable value in given scope. |
| <c:remove> | To remove the variable from given scope |
| <c:catch> | To catch the exception and wrap it into an object. |
| <c:if> | Simple conditional logic, used with EL and we can use it to process the exception from <c:catch> |
| <c:choose> | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <c:when> and <c:otherwise> |
| <c:when> | Subtag of <c:choose> that includes its body if its condition evalutes to 'true'. |
| <c:otherwise> | Subtag of <c:choose> that includes its body if its condition evalutes to 'false'. |
| <c:forEach> | for iteration over a collection |
| <c:forTokens> | for iteration over tokens separated by a delimiter. |
| <c:param> | used with <c:import> to pass parameters |
| <c:url> | to create a URL with optional query string parameters |

Let's see some of the core tags usage with a simple web application. Our project will include a Java Bean and we will create a list of objects and set some attributes that will be used in the JSP. JSP page will show how to iterate over a collection, using conditional logic with EL and some other common usage.



```
JSTLExample
    JAX-WS Web Services
    Deployment Descriptor: JSTLExample
    Java Resources
        src
            com.journaldev.model
                Employee.java
            com.journaldev.servlet
                HomeServlet.java
        Libraries
    JavaScript Resources
    Deployed Resources
    build
    WebContent
        home.jsp
        META-INF
        WEB-INF
            lib
                jstl.jar
                standard.jar
```

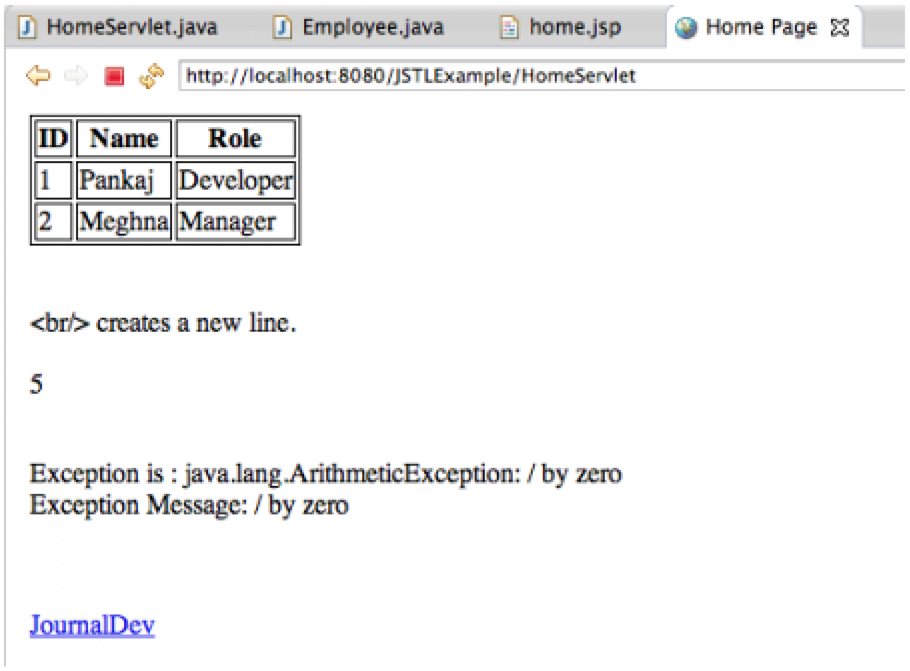
```
package com.journaldev.model;

public class Employee {

    private int id;
    private String name;
    private String role;
    public Employee() {
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }

}
```

```
package com.journaldev.servlet;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.journaldev.model.Employee;

@WebServlet("/HomeServlet")
public class HomeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        List<Employee> empList = new ArrayList<Employee>();
        Employee emp1 = new Employee();
        emp1.setId(1);
emp1.setName("Pankaj");emp1.setRole("Developer");
        Employee emp2 = new Employee();
        emp2.setId(2); emp2.setName("Meghna");emp2.setRole("Manager");
        empList.add(emp1);empList.add(emp2);
        request.setAttribute("empList", empList);

        request.setAttribute("htmlTagData", "<br/> creates a new
line.");
        request.setAttribute("url", "http://www.journaldev.com");
        RequestDispatcher rd =
getServletContext().getRequestDispatcher("/home.jsp");
        rd.forward(request, response);
    }

}
```

```
<%@ page language="java" contentType="text/html; charset=US-ASCII"
    pageEncoding="US-ASCII"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
<title>Home Page</title>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<style>
table,th,td
{
```

```
border:1px solid black;
}
</style>
</head>
<body>
<%-- Using JSTL forEach and out to loop a list and display items in
table --%>
<table>
<tbody>
<tr><th>ID</th><th>Name</th><th>Role</th></tr>
<c:forEach items="${requestScope.empList}" var="emp">
<tr><td><c:out value="${emp.id}"></c:out></td>
<td><c:out value="${emp.name}"></c:out></td>
<td><c:out value="${emp.role}"></c:out></td></tr>
</c:forEach>
</tbody>
</table>
<br><br>
<%-- simple c:if and c:out example with HTML escaping --%>
<c:if test="${requestScope.htmlTagData ne null }">
<c:out value="${requestScope.htmlTagData}" escapeXml="true"></c:out>
</c:if>
<br><br>
<%-- c:set example to set variable value --%>
<c:set var="id" value="5" scope="request"></c:set>
<c:out value="${requestScope.id }" ></c:out>
<br><br>
<%-- c:catch example --%>
<c:catch var ="exception">
   <% int x = 5/0;%>
</c:catch>

<c:if test = "${exception ne null}">
   <p>Exception is : ${exception} <br />
   Exception Message: ${exception.message}</p>
</c:if>
<br><br>
<%-- c:url example --%>
<a href="<c:url value="${requestScope.url}”></c:url>">JournalDev</a>
</body>
</html>
```

Now when we run application with URL http://localhost:8080/JSTLExample/HomeServlet, we get response as in below image.



| ID | Name | Role |
|---|---|---|
| 1 | Pankaj | Developer |
| 2 | Meghna | Manager |

```
<br/> creates a new line.

5

Exception is : java.lang.ArithmeticException: / by zero
Exception Message: / by zero

JournalDev
```

In above example, we are using c:catch to catch the exception within the JSP service method, it’s different from the JSP Exception Handling with error pages configurations.

That’s all for a quick roundup of JSTL and example of core tags usage, we will look into custom tags in future post.

## Copyright Notice

## References

1. http://www.journaldev.com/2021/jsp-example-tutorial-for-beginners
2. http://www.journaldev.com/2038/jsp-implicit-objects-with-examples
3. http://www.journaldev.com/2044/jsp-directives-page-include-and-taglib-example
4. http://www.journaldev.com/2064/jsp-expression-language-el-example-tutorial
5. http://www.journaldev.com/2090/jstl-tutorial-with-examples-jstl-core-tags