Tema 7
JSP + JSTL

JSP (**JavaServer Pages**) es una tecnología que permite incluir código Java en páginas web. El denominado *contenedor JSP* es el encargado de tomar la página, sustituir el código Java que contiene por el resultado de su ejecución, y enviarla al cliente. Así, se pueden diseñar fácilmente páginas con partes fijas y partes variables. Ejemplo sencillo de página JSP:

primera.jsp

```
<html>
<head>
<title>Mi primera página JSP</title>
</head>
<body>
        <h2> Hoy es: <%= new java.util.Date() %> </h2>
       <h2> Te conectas desde: <%= request.getRemoteAddr()  %> </h2>
</body>
</html>
```

Para ejecutar la página basta con colocarla en una aplicación web. Puede ir en cualquier directorio de la zona web en el que se colocaría normalmente un HTML.

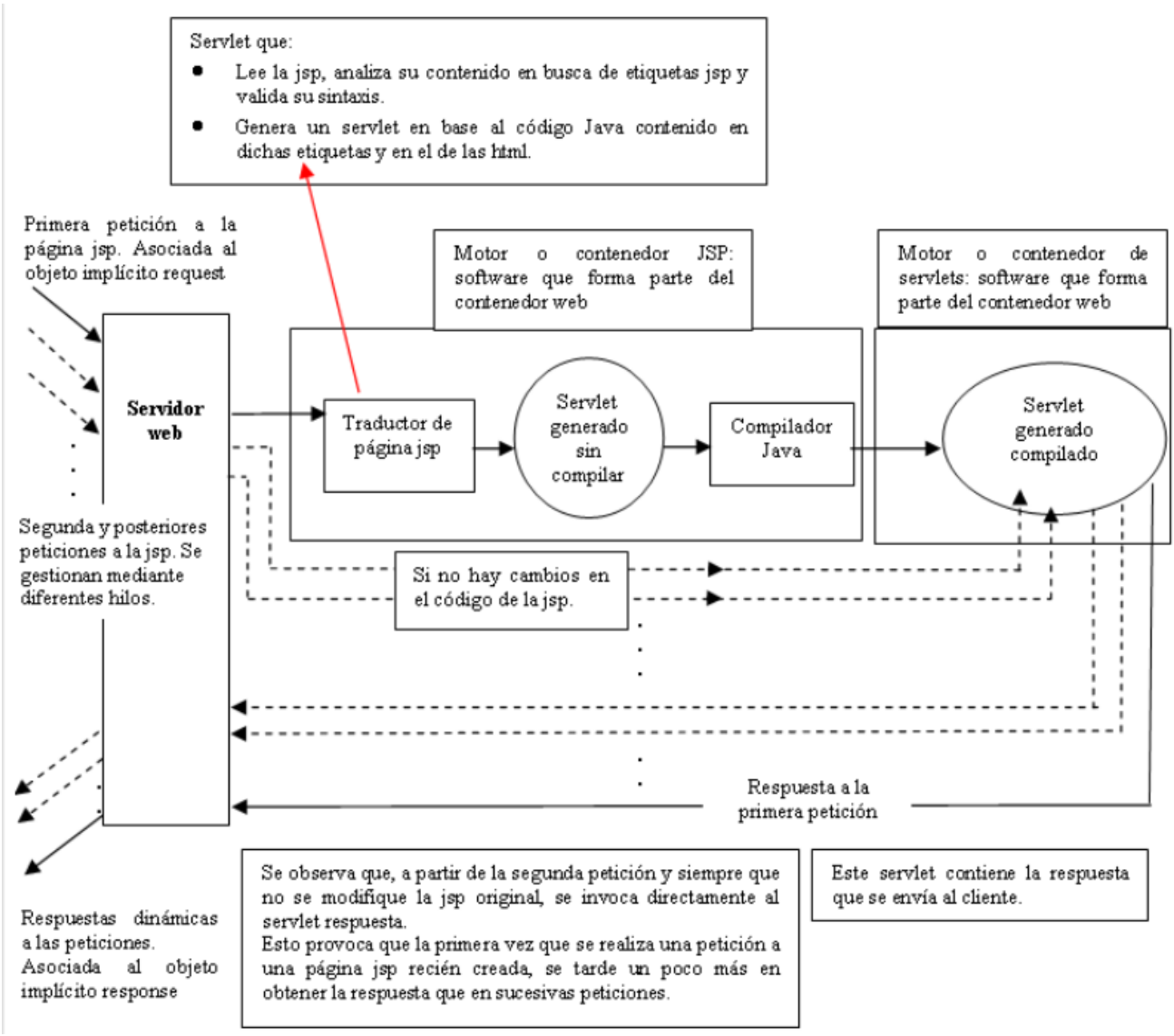
Aunque JSP y servlets parecen tecnologías distintas, en realidad el servidor web traduce el JSP a servlet, lo compila y las siguientes peticiones de la página JSP se realizan al servlet.

Por ello, aunque JSPs y servlets ofrecen la misma funcionalidad, sus características los hacen apropiados para distinto tipo de tareas.

- Los JSP se utilizan para dibujar la salida HTML. Un servlet que realice la misma función debe incluir gran cantidad de sentencias out.println().
- Los servlets son mejores en tareas que generen poca salida: tareas de inicialización, acceso a BBDDs, control del flujo de la información (redireccionamiento), control de atributos de request, sesión, etc

Una aplicación web suele combinar ambas tecnologías: los servlets para el procesamiento de información y los JSP para “dibujar” los datos que se enviarán al cliente.

Petición de una pagina JSP (traducción de JSP a Servlet)



La primera vez que se solicita una página JSP, el servidor genera el servlet equivalente, lo compila y lo ejecuta. Para las siguientes solicitudes, solo es necesario ejecutar el código compilado.
El servlet generado a partir del jsp tiene un método ***_jspService*** que es el equivalente al *service* de los servlets. Este método genera HTML, mediante instrucciones *println*, tras ejecutar el código Java insertado en la página.

El servlet asociado a una página jsp es un servlet por lo siguiente: implementa la interface HttpJspPage, que es subinterface de JspPage, que, a su vez, es subinterface de javax.servlet.Servlet.
Si se consulta la API, se observa que :

- La interface *HttpJspPage* declara el método *_jspService(HttpServletRequest request, HttpServletResponse response)*
- La interface *JspPage* declara los métodos *jspInit*() y *jspDestroy*().
- Estos tres métodos constituyen el ciclo de vida de una jsp y son similares al *init*(..), *service*(..) y *destroy*() del ciclo de vida de un servlet.

La anterior página ***primera.jsp*** podría generar un servlet con estructura similar al siguiente:

primera.jsp

```
<html>
<head>
<title>Mi primera página JSP</title>
</head>
<body>
        <h2> Hoy es: <%= new java.util.Date() %> </h2>
       <h2> Te conectas desde: <%= request.getRemoteAddr()  %> </h2>
</body>
</html>
```

Servlet generado

```
public final class primera_jsp extends org.apache.jasper.runtime.HttpJspBase….
{
          ……..
          ……
          public void _jspService(HttpServletRequest request,   HttpServletResponse  response)
                              throws java.io.IOException, ServletException {
                ……
                ……
                JspWriter out = null;
                response.setContentType("text/html;ISO-8859-1");
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Mi primera pagina JSP</title>");
                out.println("</head>");
                out.write("<body>\n");
                out.write("\t<h2> Hoy es: ");
                out.print( new java.util.Date() );
                out.write(" </h2>\n");
                out.write("        <h2> Te conectas desde: ");
                out.print( request.getRemoteAddr()  );
                out.write(" </h2>\n");
                out.write("</body>\n");   out.println("</html>");
                ……
                 …….
        }
}
```

Elementos de JSP

Existen tres tipos de elementos JSP que podemos insertar en una página web:

- **Código**: para "incrustar" cualquier código Java (declaraciones de variables y/o métodos, expresiones, sentencias), que será ejecutado por el contenedor JSP.
- **Directivas**: permiten controlar distintos parámetros del servlet resultante de la traducción del JSP
- **Acciones**: sobre todo para alterar el flujo secuencial de ejecución de la página (p.ej. redirecciones), aunque tienen usos variados.

Se pueden poner comentarios en una página JSP entre los símbolos <%-- y --%>. El contenedor JSP ignorará todo lo contenido entre ambos. Dentro de los fragmentos de código Java también se pueden colocar comentarios habituales de Java(// /* */)

Inserción de código en páginas JSP

Hay tres formas de insertar código Java en una página JSP: expresiones, scriptlets, declaraciones

Expresiones de la forma **<%= *expresión* %>**

La expresión se evalúa, se convierte a *String* y se escribe en la salida (el objeto *out*). La forma de traducir una expresión a código de servlet es imprimiéndola en out (mediante una sentencia *out.write(*expresion*))* o similar.

Ejemplos:

- Esta página ha sido generada en este momento: *<%=new Java.util.Date() %>*
- *Debes <%= x+y-z %> euros*
- *La suma es: <%= miClase.sumar(3,2) %>*
- *El color de pelo del pato <%= pato1.getNombre( ) %> es <%= pato1.getColor( ) %>*

Scriptlets de la forma **<% código %>:**

Permiten ejecutar código arbitrario Java, cuyo resultado no es necesario enviar a la salida.
Si desde un *scriptlet* se desea escribir en dicha salida, bastará con usar el objeto predefinido **out**
El código de 1 scriptlet se ejecuta dentro del método ***_jspService*** del servlet generado.

Ejemplo de scriptlet con **if** para hacer que ciertas partes de código HTML aparezcan o no:

```
<%   java.util.Calendar ahora = java.util.Calendar.getInstance();
     int hora = ahora.get(java.util.Calendar.HOUR_OF_DAY);
%>
<b> Hola mundo, <i>
<% if ((hora>20)||(hora<6)) { %>
       buenas noches
<% }
  else if ((hora>=6)&&(hora<=12)) { %>
            buenos días
<%     }
       else { %>
            buenas tardes
<%     } %>
</i> </b>
```

Declaraciones de la forma **<%! código %>**

Permiten definir variables o métodos
Se insertan como atributos del servlet generado y por tanto, conservan su valor entre sucesivas llamadas a la página de distintos clientes. Esto nos permite, por ejemplo, crear un contador de accesos a la página:

```
<%! private int accesos = 0; %>
<h1> Visitas: <%= ++accesos %> </h1>
```

Así pues, las variables declaradas en un jsp con la notación <%! %> tienen ámbito de contexto y su valor será compartido por los distintos clientes del jsp
Mientras que aquellas declaradas de manera simple (<% int x; %>), tendrán ámbito de página: se crean y destruyen por cada acceso a la página. Se utilizan , por tanto, para cálculos intermedios.

Comentarios de la forma **<%- - Comentario - -%>**

Los comentarios serán ignorados por el intérprete jsp

Es más habitual incluir los comentarios propios de Java dentro de un scriptlet:

- <% // comentario línea %>
- <% /* comentario varias líneas */ %>

Objetos implícitos de JSP

Son variables instanciadas de manera automática en el servlet generado a partir del JSP. Los objetos predefinidos en JSP son los siguientes:

Objeto	Significado
request	Objeto HttpServletRequest asociado con la petición del cliente
response	Objeto HttpServletResponse asociado con la respuesta al cliente
out	Writer empleado para enviar la salida al cliente. La salida de los JSP emplea un *buffer* que permite que se envíen cabeceras HTTP o códigos de estado aunque ya se haya empezado a escribir en la salida En JSP out no es exactamente un PrintWriter sino un objeto de la clase especial JspWriter.
session	Objeto HttpSession asociado con la petición actual. En JSP, las sesiones se crean automáticamente, de modo que este objeto está instanciado en todas las páginas jsp, aunque no se cree explícitamente una sesión.
application	Objeto de tipo ServletContext, es decir, el entorno o app web al que pertenece la página jsp. Es común a todos los servlets de la aplicación web.
config	Objeto de tipo ServletConfig, para leer parámetros de inicialización del fichero xml
page	Referencia al propio servlet (equivale a **this**). Aquí no tiene demasiado sentido utilizarla. Está pensada para el caso en que se utilizara un lenguaje de programación distinto.
~~**pageContext**~~	~~Representa el entorno de la página. Se puede utilizar para:~~ ~~• Acceder y establecer atributos en diferentes ámbitos (page, request, session, application)~~ ~~• Acceder a los objetos request, response y session, así como al JspWriter out~~ ~~• Incluir contenidos de otras URLS: void include(String relativeURL)~~ ~~• Redireccionar a otras URL: void forward(String relativeURL)~~
exception	Representa un error producido en la aplicación. Solo es accesible si la página se ha designado como página de error (mediante la directiva page isErrorPage).

Directivas de página (@)

Las *directivas* influyen en la estructura que tendrá el servlet generado a partir de la página JSP.

La sintaxis de una directiva genérica es:

```
<%@ directiva atributo="valor" atributo1=”…” atributo2=”…”  %>
```

Los nombres de los atributos son case-sensitive.

Veremos las directivas principales:

- **page**: Empleada para transmitir información de interés acerca del jsp al contenedor web. Por ejemplo, el lenguaje de script que va a emplear, clases a importar, cómo se van a gestionar los errores, etc.
- **include**: sirve para incluir código en la página antes de que se realice la compilación del JSP.

Directiva **page**

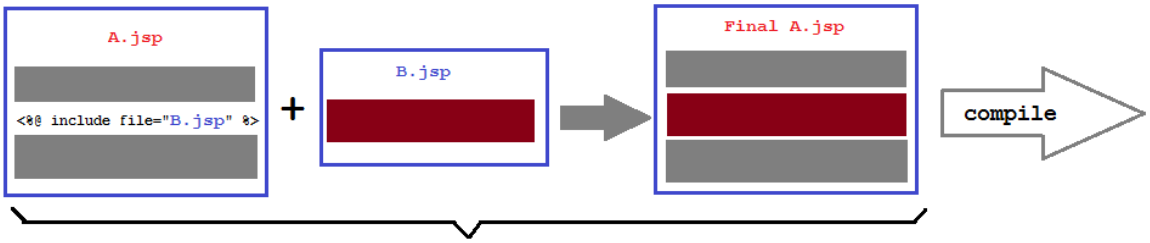
Directiva **page**. ATRIBUTOS		
Atributo	**Significado**	**Ejemplo**
import	Equivalente a una sentencia import de Java. Si son varios, se separan por comas	<%@ page import="java.util.Date, …." %>
contentType	Genera una cabecera HTTP Content-Type	<%@ page contentType="text/html" %>
pageEncoding	Define el juego de caracteres que usa la página. El valor por defecto es ISO-8859-1	<%@page contentType="text/html" pageEncoding="UTF-8" %>
isThreadSafe	Si es false, el servlet generado implementará el interface SingleThreadModel .Esto crea un único hilo para todas las peticiones → hasta no procesar 1 petición, no se admitirá otra. Suele dejarse el valor por defecto, que es true (permitir varios hilos simultáneos)	
session	Si es false, no se crea un objeto session de manera automática. Suele dejarse el valor por defecto: true	
buffer	Define el tamaño del *buffer* de salida o none si no se desea *buffer.* El buffer de salida es un contenedor donde se almacena la respuesta antes de ser enviada al cliente. Su valor por defecto es 8 kb	<%@ page buffer="64kb" %>
autoflush	Si es true (valor por defecto), el buffer se envía automáticamente a la salida al llenarse. Si es false, al llenarse el buffer se genera una excepción.	
info	Define una cadena con información sobre el jsp, que puede obtenerse a través del método getServletInfo de la clase GenericServlet	<%@ page info="carro de la compra" %>
errorPage	Qué página JSP debe procesar los errores no capturados en la página actual. Si no se define, el contenedor web se encarga de mostrar los mensajes de error.	<%@ page errorPage="error.jsp" %>
isErrorPage	Si es true, indica que esta página actúa como página de error para otro JSP. Por defecto es false.	

La directiva **include**

Permite insertar código en la página antes de que ésta se transforme en un servlet (al traducirla)
De este modo se pueden reutilizar fragmentos de código JSP o HTML.
Su sintaxis es:

```
<%@ include file="fichero" %>
```

Equivale a reemplazar la directiva con el contenido del fichero incluido.



Como el código se incluye en el servlet generado, los fragmentos de código incluidos pueden tener efecto sobre la página actual. Se suele utilizar esta directiva para definir constantes, generar encabezados o pies de página que contendrán todas las jsps de una aplicación web, …

El contenedor JSP no detecta de manera automática los cambios en los ficheros incluidos, de manera que si cambia uno de ellos puede ser necesario forzar la recompilación de las páginas JSP que los incluyan.
Si queremos que cada vez que se referencie la página incluida se vuelva a recargar su contenido lo haremos con la acción <jsp:include page="*nombre del archivo*" flush="true" />

Acciones

Permiten incluir elementos externos en la página jsp (applets, otras páginas jsp, servlets, javabeans, etc), así como redirigir la petición a otra página JSP. Utilizan nomenclatura xml.

- La acción **<jsp:include>** permite insertar la *salida* de otro recurso (p.e, por otra página jsp), no el código (Ver diferencia con la directiva **include)**
- La acción **<jsp:forward>** sirve para redirigir la petición a otra página JSP.

La acción **<jsp:include>**

Esta acción incluye en una página la salida generada por otra perteneciente a la misma aplicación web. La petición se redirige a la página incluida, y la respuesta que genera se incluye en la generada por la principal. Su sintaxis es:

```
<jsp:include page="URL relativa" flush="true|false"/>
```

El atributo page es la URL del recurso a incluir.
El atributo flush por defecto es false. Si se cambia a true, especifica que el flujo de salida de la página principal deberá ser enviado al cliente antes de enviar el de la página incluida. No suele utilizarse.

Esta acción presenta la ventaja sobre la directiva <%@include %> de que cambios en la página incluida no obligan a recompilar la "principal".
La página incluida solo tiene acceso al JspWriter de la "principal" y no puede generar cabeceras (por ejemplo, no podría crear *cookies*).

La petición que se le pasa a la página incluida es la original, pero se le pueden agregar parámetros adicionales, mediante la etiqueta **<jsp:param**> (se acceden con getParameter) Por ejemplo:

```
<jsp:include page="cabecera.jsp">
        <jsp:param name="color" value="YELLOW" />
        <jsp:param name="tamanio" value="16" />
        . . . . . .
</jsp:include>
```

Es posible utilizar expresiones en cualquiera de los atributos. Por ejemplo:

```
<%   String url = "cabecera.jsp";
        String valor1 = "YELLOW";
        String valor2 = "16";
%>

<jsp:include page="<% = url    %>">
        <jsp:param name="color" value="<%= valor1 %>" />
        <jsp:param name="tamanio" value="<%= valor2 %>" />
        . . . . . .
</jsp:include>
```

La acción **<jsp:forward>**

Esta acción se utiliza para redirigir la petición hacia otra página JSP, Servlet o página HTML que esté en la misma aplicación web que la actual. Un ejemplo de su sintaxis básica es:

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

La salida generada hasta el momento por la página actual se descarta (se borra el buffer), y todas las líneas que aparezcan a continuación de esta acción no se consideran.

Al igual que en el caso de <jsp:include>, se pueden añadir parámetros a la petición original para que los reciba la nueva página JSP:

```
<jsp:forward page="principal.jsp">
        <jsp:param name="privilegios" value="root" />
</jsp:forward>
```

JavaBeans

Los javabeans (también llamados beans) son clases Java corrientes que se almacenan en un servidor web (tipo la clase Pato), que deben cumplir una serie de requisitos y que se comportan como cajas negras: sólo es necesario conocer qué hacen y decidir si interesa o no utilizarlas desde las páginas jsp.
Pueden considerarse componentes software que ofrecen a los desarrolladores de aplicaciones web una serie de utilidades sin necesidad de conocer el modo en que se obtienen (pensemos que han sido desarrollados por otro equipo).

Ventajas del uso de javabeans:
- Pueden ser utilizados por personas sin amplios conocimientos de Java
- Facilitan la separación entre la lógica de presentación de datos de una aplicación y la lógica de negocio, es decir, entre el mostrar contenidos y el generarlos.

Como norma general, no conviene llenar de scriptlets una jsp, sino trasladar toda la parte lógica Java que se necesite a uno o varios beans.Estos beans procesarán los datos para que los muestre la página jsp. Desde la jsp, simplemente se invocará al bean del modo adecuado y se obtendrá lo deseado.

Además, esta forma de trabajo permite reutilizar beans de otras aplicaciones (no necesariamente web).

Requerimientos de la clase asociada a un bean:
- Debe ser pública
- Debe tener, al menos, un constructor público sin argumentos. Como ya sabes, ésto se consigue definiendo uno explícitamente o no definiendo ninguno
- Debe tener métodos públicos de tipo *setXxx* y *getXxx* que permitan la inicialización y el acceso a las variables de instancia del bean (preferiblemente privadas). No son necesarios todos.
  Estos métodos son fundamentales porque definen las propiedades del bean. Sus nombres son lo que viene a continuación del *set* o *get,* cambiando la mayúscula por su correspondiente minúscula.
  * Ejemplo: Si un bean tiene los métodos *setNombre, getNombre, setApellidos, getApellidos, getLugarNacimiento*, dicho bean tiene tres propiedades: nombre y apellidos, que son de lectura y escritura, y *lugarNacimiento* de lectura.

Aparte de métodos setters y getters asociados a propiedades, el bean puede tener otros métodos a los que también podrá accederse desde la jsp

Ubicación de los JavaBeans: por ser clases Java, deben estar situados en la misma zona que los servlets.

Ejemplo de clase de tipo Bean

```
public class Aficion{
    private String nombre;
    private int riesgo;

    public void setNombre(String nombre){
            this.nombre = nombre ;
    }
    public void setRiesgo(int riesgo){
            this.riesgo=riesgo ;
    }
    public String getNombre( ){
            return nombre ;
    }
    public int getRiesgo( ){
            return riesgo ;
    }
    public String getComentarioAficion( ){
            if (nombre.equals("Tumbismo"))
                    return "Eres el monstruo del sofa";
            else if (nombre.equals("Buceo"))
                    return "Cuidado con los pulpos";
            else if (nombre.equals("Parapente"))
                    return "Cuidado no te estrelles";
            else return "Aficion saludable";
    }
    public String despedida( ){
            return "Adios";
    }
}
```

JSTL

En JSP es posible definir librerías de etiquetas personalizadas. Estas etiquetas no son más que clases Java que heredan de determinadas clases y que se agrupan en librerías mediante un archivo descriptor TLD.
No nos vamos a encargar de construir librerías de etiquetas, pero sí vamos a utilizar una muy común: JSTL

JSTL es un componente dentro de la especificación J2EE . Se trata de un conjunto de librerías de etiquetas y estándares que amplía el conjunto básico de etiquetas JSP. Consta de los siguientes grupos de etiquetas:

- **core**: Con las funciones básicas para escribir, como bucles, condicionales, entrada/salida.
- **xml**: Para procesamiento de xml.
- **fmt**: Comprende la internacionalización y formato de valores como de moneda y fechas.
- **sql**: Comprende el acceso a base de datos.

Además JSTL define un nuevo lenguaje de expresiones llamado **EL** (Expression Language), adoptado por JSP 2.0. En EL, las expresiones están delimitadas por ${ }.
Nos centraremos en la librería ***core*** y en el ***EL***

En las expresiones (EL), podemos usar:

- los operadores típicos +, -, *, /, %, >, >=, <, <=, ==, !=, &&, ||, !
- El operador empty, que nos servirá para comparar a la vez con null y con cadena vacía

El *EL* tiene acceso a todos los objetos implícitos de los jsp (request, session, application,out,…), a los que se añaden los objetos **pageContext, pageScope, requestScope, sessionScope, applicationScope, param, paramValues , header, headerValues, cookie**

Para acceder a un atributo de un objeto, podemos usar los operadores '.' Y '[]', de la forma:
objeto.atributo ó **objeto["atributo"]**

Ejemplo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"  prefix="c"  %>
<c:out  value="${1+2+3}" /> <br/>
<%= request.getParameter("parametro") %>    equivale a      <c:out value="${param.parametro}"  />
<%  application.setAttribute("atributo","valor");  %>
<%= application.getAttribute("atributo")  %>   equivale a       <c:out value="${applicationScope.atributo}" />
```

En el ejemplo vemos que con <%= request.getParameter("parametro") %> se muestra null si el parámetro no está definido, mientras que con <c:out value="${param.parametro}" /> se muestra la cadena vacía.
En las últimas versiones de JSTL, se pueden embeber directamente expresiones EL, de forma que se hace innecesario el uso de <c:out> para mostrarlas.

Ejemplo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"  prefix="c"  %>
Valor del parámetro 'direccion':      ${param.direccion}
Valor del atributo de sesión 'usuario':          ${sessionScope.usuario}
```

La librería **core** de JSTL

En las páginas que la usen tendremos que incluir la directiva:

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

Esta librería implementa acciones de propósito general, como mostrar información, crear y modificar variables de distinto ámbito, tratar excepciones, etc. Veamos algunas acciones

<c:out>

Muestra en la salida html el resultado de evaluar su atributo value. Ejemplos:

<c:out value="1+2+3" />	Equivale a	"1+2+3"
<c:out value="${1+2+3}" />	Equivale a	${1+2+3}
<c:out value="${param.nombreParametro}" />	Equivale a	${param.nombreParametro}
<c:out value="${sessionScope.atribDeSesion} " />	Equivale a	${sessionScope.atribDeSesion}

<c:set>

Guarda información en una variable y tiene los siguientes atributos

- var: Nombre de la variable
- scope: Ámbito (page, request, session, application)

```
<c:set var="variableDePagina" scope="page" >
        Esta información se guarda en la página
</c:set>
<c:set var="variableDeSesion" scope="session" >
        Esta información se guarda en la sesión actual
</c:set>
<c:set var="variableDeAplicacion" scope="application">
        Esta información se guarda en la aplicación
</c:set>
<br>${variableDePagina}
<br>${variableDeSesion}
<br>${variableDeAplicacion}
```

<c:if> *(No válido para if-else)*

Procesa el cuerpo de la etiqueta si la condición es cierta. La condición se indica en el atributo *test*

```
<c:if test="${applicationScope.totalVisits == 1000000}" var="visitas">
                Eres el visitante 1.000.000. Enhorabuena!
</c:if>
```

<c:choose>

Procesa condiciones múltiples. Equivale a la sentencia ***switch***. (O a una sentencia if-else si sólo contiene un bloque *<c:when>* + un bloque *<c:otherwise>*)

```
<c:choose>
        <c:when test="${item.tipo == 'book'}">
        ...
        </c:when>
        <c:when test="${item.tipo == 'electronics'}">
        ...
        </c:when>
        <c:when test="${item.tipo == 'toy'}">
        ...
        </c:when>
        <c:otherwise>
        ...
        </c:otherwise>
</c:choose>
```

<c:forEach>

Se utiliza para iterar sobre una colección (lista, mapa, conjunto,….)
Consta de los siguientes atributos:

- **items**: nombre de la colección sobre la que iterar
- **var**: nombre simbólico de la variable donde se guarda el elemento en curso
- **varStatus**: guarda el estado de la iteración. Se trata de un objeto con varias propiedades interesantes acerca de la iteración actual
    - **index**: Indice del elemento en curso (comienza en 0)
    - **count**: Número de iteración (comienza en 1)
    - **first**: Booleano que indica si es la primera iteración
    - **last**: Booleano que indica si es la última iteración

```
<table>
<c:forEach items="${sessionScope.arrLibros}" var="libro">
    <tr>
      <td>${libro.titulo}</td>
      <td>${libro.autor.nombre}</td>
      <td>${libro.autor.nacionalidad}</td>
    </tr>
</c:forEach>
</table>
```

Recorre arrayList de objetos Libro de la sesión

```
<ul>
<c:forEach items="${mapaPaises}" var="pais" >
        <li> Country name: ${pais.key}  - Capital: ${pais.value}  </li>
</c:forEach>
</ul>
```

Recorrido de HashMap (Pais → Capital) existente en el ámbito +cercano

```
<c:forEach items="${arrLibros}" var="libro" varStatus="estado">
            <a href="${libro.isbn}"> ${libro.titulo}</a>
             ${ estado.last ? '' : '<hr/>'}
</c:forEach>
```

Recorrido de array de Libros, haciendo uso de varStatus para controlar en qué iteración estamos

<c:redirect>

Redirige a la dirección especificada en el atributo url.
El navegador generará una nueva petición (equivale a ***response.sendRedirect()***)

```
<c:if test="${param.clave!='damocles'}">
        <c:redirect url="login.jsp" />
</c:if>
```

<c:catch>

Lo utilizaremos para capturar excepciones sin que se aborte la ejecución de la página al producirse un error.
El atributo *var* guarda el nombre de la variable que almacenará información de la excepción (null si no se ha generado la excepción)

```
<c:catch  var="error01">
        <%=  Integer.parseInt(request.getParameter("parametro"))  %>
</c:catch>
<c:if  test="${error01!=null}">
        Se produjo un error: ${error01}
</c:if>

<form>
        <input  type="hidden" name="parametro" value="prueba"  />
        <input  type="submit" name="submit" value="Enviar 'prueba'"  />
</form>
<form>
        <input  type="hidden" name="parametro" value="1234"  />
        <input  type="submit" name="submit" value="Enviar 1234"  />
</form>
<form>
        <input  type="submit" name="submit" value="No enviar parametro"  />
</form>
```