1. INTRODUCCIÓN

1.1. ¿Qué es JSP?

Java Server Pages (JSP, para abreviar) es una tecnología basada en Java que simplifica el desarrollo de páginas web con contenido dinámico. Con JSP, los diseñadores web pueden incorporar elementos dinámicos dentro de la página utilizando tanto porciones de código Java incrustadas, como unas cuantas etiquetas. Así, las páginas JSP tienen el aspecto de una página tradicional HTML, a la que se le ha introducido parte de código Java junto con unas etiquetas. De esta forma, cuando una página es solicitada por un usuario y procesada por un servidor HTTP, el código HTML pasará directamente al usuario, mientras que las porciones de código Java serán ejecutadas en el servidor cuando la solicitud haya sido recibida, para generar el contenido dinámico de la página. Cuando el usuario acceda al código de la página que le llega sólo verá HTML, sin poder acceder al código JSP subyacente.

1.2. Evolución tecnológica

El hecho de que una página contenga contenido dinámico, exige que el servidor web realice algún trabajo de procesado que en el caso de páginas estáticas no era necesario. La forma de realizar este procesado ha ido cambiando con el tiempo, y así veremos ahora un pequeño resumen de cómo ha ido evolucionando.

Inicialmente la generación de contenido dinámico se hacía fuera de los servidores. Cuando llegaba una solicitud, esta era procesada en el servidor, y cuando era necesario se llamaba a un proceso, fuera del servidor, que generaba el contenido dinámico y lo devolvía al servidor. Los modelos basados en el *Common Gateway Interface* (CGI) seguían esta idea. Sus principales problemas eran: necesidad de *overhead* para la comunicación entre proceso y servidor, consumo de recursos de sistema por parte del proceso...

Posteriormente se optó por sistemas que introdujeran porciones de código de lenguaje tradicionales incrustados dentro de la página usando etiquetas, de forma que el estilo de la sintaxis fuera más consistente y sencillo. Este modelo lo han seguido entre otros: *Microsoft Active Server Pages (ASP), Server Side JavaScript (SSJS), Java Server Pages (JSP)* y otros.

En JSP el lenguaje que se utilizará para la generación de contenido dinámico es, típicamente, Java, y provee un conjunto de etiquetas que interactúan con objetos Java en el servidor de forma que no es estrictamente necesario que aparezca código Java en la página.

1.3. Beneficios

JSP ofrece varios beneficios como sistema de generación de contenido dinámico. Al estar basado en Java, presenta las ventajas que este lenguaje ofrece con respecto a la portabilidad entre plataformas y las derivadas de la orientación a objetos de este lenguaje.

Realización.

Las peticiones de páginas JSP son normalmente implementadas mediante *servlets*, de forma que el contenedor *servlet*, al que llamaremos contenedor JSP, maneja múltiples solicitudes a la vez, requiriendo menor *overhead*, y por tanto requiriendo menos recursos. Esto hace que JSP sea mucho más eficiente que otros modelos como los programas CGI .

Componentes reutilizables.

Esta característica deriva de la orientación a objetos de Java. JSP permite implementar contenido dinámico incluyendo código Java directamente en la página. Sin embargo, también ofrece una serie de etiquetas que le permiten actuar sobre objetos Java residentes en el servidor. Estos objetos se comportan como cajas negras a las que la página accede sin tener que conocer como funcionan internamente, y por tanto, liberando al creador de la página de la programación en Java. Estos objetos, además podrán ser reutilizados sin más que conocer su funcionalidad.

Separación de presentación e implementación.

Esta ventaja proviene directamente de la anterior. El hecho de que la implementación del programa puede ser llevada a cabo por los objetos Java, podemos separar lo que es la presentación en sí, en la página, y el código encargado de generar la información necesaria que aparecerá en la página.

División de labor.

La separación de presentación e implementación permitirá desligar las labores encargadas de desarrollar ambas. Así, alguien que no sepa nada de Java, podría encargarse de la parte de la página relacionada con la presentación, le bastaría conocer las propiedades que les ofrece un conjunto de objetos, y así accediendo a ellos conseguirían la información necesaria. Del mismo modo, un programador Java, siguiendo una serie de normas a la hora de crear los objetos, se encargaría de crear el código que generará la información dinámica, despreocupándose de los problemas de presentación de la página.

2 FUNDAMENTOS

2.1 Escribiendo tu primera JSP.

En primer lugar hay que decir que el contenedor JSP acepta las etiquetas HTML, de forma que la solicitud a un contenedor JSP local del siguiente fichero, al que habremos dado su correspondiente extensión (normalmente .jsp), sería perfectamente procesado, y por lo tanto sería un fichero JSP válido.

```
<HTML>
<BODY>
¡Hola, mundo!
</BODY>
</HTML>
```

Veamos ahora un ejemplo más interesante:

```
<HTML>
<BODY>
<% String visitante = request.getParameter("nombre");

if (visitante==null) visitante = "mundo"; %>
¡Hola, <%= visitante%>!

</BODY>
</HTML>
```

Sin entrar en detalles, este fichero JSP declara una variable Java *String* llamada *visitante*, y luego la intenta inicializar con un parámetro que busca en la solicitud HTTP. En caso de no encontrarlo, le da el valor *mundo*, y posteriormente inserta el valor de esta variable en la salida HTML de esta página.

Asumiremos que el URL para esta página JSP es **http://localhost:8100/Hola.jsp** y así, si un navegador buscara esta URL mediante una solicitud *GET* el contenedor JSP procesaría esta página y respondería con lo siguiente.

¡Hola, mundo!

En efecto, al no encontrar ningún parámetro en la solicitud, le daría a la variable *visitante* el valor *mundo*. Si, por el contrario, accediéramos a dicha página, mediante un *GET* a **http://localhost:8100/Hola.jsp?nombre=Toni**, estaríamos pasando un valor al parámetro nombre de valor *Toni*, y así la variable visitante valdría *Toni*. Por lo tanto, la respuesta a esta petición sería:

¡Hola, Toni!

En el código JSP del ejemplo anterior, hemos puesto en negrita el código puramente JSP. Como ya comentamos anteriormente, este código es ejecutado en el lado del servidor, generándose el contenido dinámico, de forma que el usuario al ver el código de la respuesta únicamente vería:

<HTML>
<BODY>
¡Hola, Toni!
</BODY>
</HTML>

Con este simple ejemplo ya hemos visto la característica central de las páginas con contenido dinámico: dependiendo del valor de los parámetros en la solicitud, el resultado de la misma es distinto. Otro aspecto importante es el hecho de que el navegador no tiene constancia del código JSP de la página. Este código se ejecuta en el contenedor JSP, generando contenido HTML, y devolviéndolo completo al servidor HTTP, que lo devuelve al navegador sin dejar ninguna constancia de toda esta labor. Por último, es interesante reseñar que en este ejemplo el contenido dinámico se ha obtenido mediante la inclusión en la página de *scripting*, puro lenguaje Java.

En el capítulo anterior hablamos de que uno de los beneficios que ofrece JSP es la posibilidad de usar objetos Java, a los que llamaremos *Javabean* en el servidor, permitiendo la separación entre las labores de presentación e implementación. Veamos cómo podemos hacer esto, para ello supongamos que tenemos el siguiente *bean*.

```
public class Holabean{

String nombre;

public Holabean(){

this.nombre="mundo";

}

public String getNombre(){

return nombre;

}
```

public void setNombre(String nombre){

this.nombre=nombre;

}

}

En principio, no se trata más que de una clase Java normal y corriente, pero el hecho de que haya cumplido una serie de normas la convierten en un *bean*, accesible desde una página JSP que lo llame. Así, y sin entrar en detalles, una página JSP como la que sigue estaría accediendo a dicho *bean*.

<HTML>
<BODY>
<jsp:usebean id="hola" class="Holabean"/>
<jsp:setProperty name="hola" property="nombre" param="nombre"/>
¡Hola ***<jsp:getProperty name="hola" property="nombre"/>****!*
</BODY>
</HTML>

En esta página, primero se declara que se va a acceder a *Holabean* al que se le llamará *hola*. Después se accede en escritura a la propiedad *nombre* de dicho *bean* asignándole el valor que venga en el parámetro *nombre* de la solicitud. Por último se accede a la misma propiedad, pero ahora en lectura, para conocer cuál es su valor e introducirlo en la salida de la página.

2.2 Formato de las etiquetas.

Una vez que hemos visto unos ejemplos que nos muestran la forma que toman los documentos JSP, llega el momento de clasificar las etiquetas que se utilizan en este lenguaje. Estas etiquetas quedan englobadas de dos tipos de formato: las etiquetas orientadas a *scripting* y las basadas en XML.

1.− Orientadas a *scripting*.

Son etiquetas derivadas del ASP (*Microsoft Active Server Page*) y toman el formato de empiezar por ´<%´ y terminar por ´%>´. Un carácter especial puede aparecer después del ´<%´ inicial, tal como: ¡ , = , @ y sirven para añadir significado a la etiqueta.

<%! double radio=7.8; %>
<%@page include file="copyright.html" %>

2.− Basadas en XML..

El segundo tipo de etiqueta viene de la sintaxis utilizada en XML y sus convenciones. Esta sintaxis es muy parecida a la de HTML, pero añadiendo unas pocas reglas tales como diferenciar entre mayúsculas y minúsculas.

Empiezan por < y terminan por />.

<jsp:usebean id="login" class="com.taglib.wdjsp.fundamentals"/>

Al contrario que las anteriores, estas etiquetas pueden tener un cuerpo, y toman el siguiente formato:

<jsp:usebean id="login" class="com.taglib.wdjsp.fundamentals">
<jsp:setProperty name="login" property="group" value="admin"/>
</jsp:usebean>

En este caso, el cuerpo de la etiqueta es otra etiqueta. Esto es un caso muy frecuente cuando al notificar el uso de un *bean* inicializamos sus propiedades.

JSP soporta el anidamiento de etiquetas una dentro de otra. Así, cualquier etiqueta JSP puede estar dentro de las etiquetas HTML del documento, permitiendo generar contenido dinámico de las propias etiquetas. Igualmente, JSP permite contener, bajo ciertas restricciones, etiquetas JSP dentro de otras etiquetas JSP. Aquí tenemos un ejemplo:

<jsp:setProperty name="login" property="visitantes" value="<% visitasPrevias + 1%>"/>

2.3. Ejecutando JSP

Con lo que hemos visto, deberíamos tener una idea general de qué es lo que JSP puede hacer. Ahora veremos cómo lo hace.

2.3.1. Requisitos para JSP.

Lo primero que necesitamos para trabajar con JSP es un servidor web. Cuando hablamos de servidor, nos referimos tanto a una parte *hardware* como a una parte *sofware*. La parte *hardware* es normalmente un ordenador accesible desde Internet o desde una red corporativa, mientras que la parte *sofware* es un servidor HTTP ejecutándose en el *hardware*.

Además del servidor HTTP, es necesario un *sofware* que implemente un contenedor JSP. Muchos servidores incluyen este contenedor. En otros, será necesario incluir este contenedor. Actualmente, la mayoría de proveedores de servidores HTTP ofrecen API (*Application Programming Interfaces*) para incorporar la generación de contenido dinámico a la funcionalidad HTTP.

Una vez que tenemos el servidor instalado y configurado, añadiremos los ficheros JSP a la jerarquía normal de documentos del servidor, y se distinguirán del resto por su extensión, normalmente .jsp aunque no siempre es esta. Las clases Java, ya compiladas, que son referidas por nuestros documentos JSP necesitan ser instaladas en el path de las clases Java que utiliza el contenedor Java.

2.3.2. ¿Cómo trabaja JSP?..

El proceso de ejecución de un documento JSP empieza con la solicitud del mismo. Estas solicitudes están indicadas por el URL que emplea una extensión especial, que ya dijimos que generalmente era *.jsp* pero podría ser otra.

Servlets

La mayoría de las implementaciones de JSP están basadas en los *servlets*. Por ello, el primer paso para comprender cómo trabaja JSP, es comprender cómo trabajan los *servlets*.

Los *servlet*s son programas basados en Java, análogos a los programas CGI, implementados mediante un contenedor *servlet* asociado a un servidor HTTP. El fundamento de los *servlets* es el siguiente; un conjunto de URLs son configurados para ser administrados por el contenedor *servlet*, de forma que siempre que llegue una solicitud para una de estos URLs en el servidor, este lo envía al contenedor *servlet* para que lo procese.

La forma de enviarlo es creando un objeto Java que empaquete todos los datos de la solicitud. Un objeto Java también es creado representando la respuesta. Ambos objetos tendrán sus métodos de acceso, de esta forma, el contenedor *servlet* accede a los datos de la solicitud para realizar las operaciones necesarias sobre los mismos y así construir la respuesta. El código HTML generado como respuesta (no hay que olvidar que el contenido dinámico generado es código HTML) es escrito en la cadena de salida asociada al objeto respuesta, y este objeto es enviado al servidor HTTP, el cual la devuelve al navegador que hizo la solicitud en primer lugar. En el caso de que existan múltiples solicitudes para un *servlet*, están son administradas ejecutando cada llamada a los métodos de los *servlet*s en diferentes threads.

JavaServer Pages.

El componente principal de una implementación de JSP basada en *servlets* es un *servlet* especial llamado compilador de página. El contenedor está configurado para llamar a este *servlet* siempre que llega una solicitud a una página JSP. Es este compilador de página y su clase Java asociada el que vuelve al contenedor *servlet* en un contenedor JSP.

El procedimiento es el que sigue. Cuando llega al servidor HTTP una solicitud de una página JSP, esta solicitud es enviada al contenedor JSP, el cual invoca al compilador de página para que se encargue de la misma. El compilador analiza el contenido de la página buscando etiquetas JSP, traduciendo su contenido en el código Java equivalente que, al ser ejecutado, generará el contenido dinámico. Mezclando el contenido estático de la página original junto con el código Java del contenido dinámico, se generará un *servlet* con sus métodos de servicio. Una vez que todo el código del *servlet* ha sido construido, el compilador de página llama al compilador Java para compilar este código y añadir el fichero de clase Java resultante al directorio apropiado en el path de las clases del contenedor JSP. Todo este proceso sólo se realiza la primera vez que se solicita una página JSP, el resto de solicitudes son remitidas directamente al *servlet* compilado. Así cuando se llama a una página JSP el compilador de página invoca a este *servlet* para generar la respuesta para la solicitud original.

Resumiendo, podemos decir que las solicitudes del navegador llegan al servidor HTTP y las páginas JSP son enviadas al *servlet* compilador de páginas que corre en el contenedor JSP. Si el *servlet* para la página actual está actualizado lo genera y compila, cargándolo en el contenedor *servlet*. En caso contrario el control es transferido al *servlet* de la página JSP que se encarga de manejar la solicitud generando la respuesta y enviándola al servidor HTTP el cual la remitirá al navegador.

3. Programando JSP *script*.

3.1 Lenguajes de *scripting*.

Por defecto, el lenguaje *scripting* para JSP es Java. En el tema anterior, vimos cómo podíamos insertar dentro de una página JSP porciones de código Java con el uso de determinadas etiquetas. A estas porciones de código Java es a lo que llamamos *scripts*. Teniendo en cuenta que JSP es un lenguaje orientado a componentes, la opción de usar *scripts* parece menos aconsejable que el uso de *Beans*, sin embargo en muchos casos no son soluciones enfrentadas y además nos pueden resultar muy útiles. Por ello en este tema pretendemos estudiar las posibilidades que nos ofrecen los *scripts* cuando trabajamos con JSP.

3.2 Etiquetas JSP. (Ir al índice)

JSP ofrece cuatro tipos principales de etiquetas:

- − Directivas: estas etiquetas servirán para dar al contenedor JSP de instrucciones específica de la página sobre cómo el documento va a ser procesado. Las directivas no afectan al manejo de solicitudes individuales, sino que afectan a las propiedades globales de la página JSP.

- − Elementos *scripting*: se utilizan para insertar instrucciones de programación escritas en el lenguaje *scripting* de la página (generalmente Java), las cuales serán ejecutadas cada vez que la página sea procesada por una solicitud.
- − Comentarios:se usan para añadir cadenas de comentarios a la página JSP. Existen varias formas de introducir comentarios en JSP. Algunos de ellos se verán en la salida, otros sólo se verán en el documento original JSP o en el código fuente del *servlet* en el que la página es traducida.
- − Acciones: al igual que los *scripting* son procesadas cada vez que la página es solicitada. Las acciones pueden transferir el control entre páginas, especificar *applets* e interactuar con componentes *JavaBeans* en el servidor.

3.3 Directivas. (Ir al índice)

Las directivas son usadas para llevar información del procesado especial al contenedor JSP. Por ejemplo, se pueden usar para especificar el lenguaje *scripting* para la página, incluir contenidos de otra página. Las directivas no producen directamente ninguna salida visible para los usuarios cuando la página es solicitada, en lugar de eso, generan efectos laterales que modifican el modo en el que el contenedor JSP procesa la página.

3.3.1. Directiva *page*.

La directiva *page* soporta un amplio abanico de atributos, lo cual la convierte en una de las directivas más complicadas. La sintaxis básica de esta directiva es:

<%@ page atributo1="valor1" atributo2="valor2"...%>

Veamos los **atributos de la directivia *page***:

- − *info*: permite añadir información en forma de cadena de caracteres a la página que resume su función. No tiene ninguna restricción con respecto al tamaño de la cadena y no es obligatoria.

<%@ page info="Mi página particular, Copyright 2000 por Antonio Galán" %>

- − *language*: especifica el lenguaje *scripting* que se va a utilizar en los elementos *scripting* de la página. Por defecto, este lenguaje es el Java ya que todos los contenedores JSP deben soportar Java.

<%@ page language="java" %>

- − *extends*: identifica la superclase que será usada por el contenedor JSP cuando la página JSP sea traducida a un *servlet* Java. Su uso es muy poco común.

<%@ page extends="com.librería.miPaginaJsp" %>

- − *import*: sirve para especificar el conjunto de clases Java que serán utilizadas en la página. Para especificar correctamente una clase hay que especificar correctamente el paquete que la contiene.

<%@ page import="java.util.List" %>

Aparte de estos atributos, existen varios más, que se pueden consultar en el manual de JSP que proporciona ***Sun Microsystem***.

3.3.2. Directiva *include*.

La directiva *include* permite al creador de una página JSP incluir el contenido de un fichero en otro. El fichero

incluido será identificado por su URL, y la directiva tiene el efecto de colocarlo en nuestra página en el punto en el que aparece la directiva, antes de que la página sea traducida a código fuente y compilado en un *servlet*.

<%@ include file="localURL" %>

Una página JSP puede incluir otra JSP siempre y cuando esta utilice el mismo lenguaje *scripting* que aquella.

3.3.3. Directiva *taglib*.

Se usa para informar al contenedor JSP que la página utiliza una o más librerías de etiquetas de cliente. Su utilización es como sigue:

<%@ taglib uri="libreriaEtiquetaURI" prefix="prefijoEtiqueta" %>

Un funcionamiento más detallado de esta directiva puede verse en el manual de JSP que ***Sun Microsystem*** proporciona.

3.4 Elementos de *scripting.* (Ir al índice)

Al contrario que las directivas, los elementos *scripting* permiten insertar directamente código que generan salida para que la vea el usuario. Existen tres tipos de elementos *scripting*: declaraciones, *scriptlets* y expresiones.

3.4.1 Declaraciones.

Se usan para definir variables y métodos específicos de una página JSP. Estas variables y métodos pueden ser referenciados por otros elementos *scripting* en la misma página. La sintaxis para las declaraciones es:

<%! declaracion(s) %>

Declaración de variables.

Las variables definidas con las declaraciones se convierten en variables de objeto en la clase *servlet* en la que la página es traducida y compilada. Así, si declaramos:

<%! private int x=0, y=0; private String unidades="m"; %>

Estaremos creando tres variables de objeto en el *servlet* creado para la página JSP que pueden ser referenciadas por cualquier elemento *scripting* de la página, incluso aquellos que aparezcan antes de la declaración.

Es importante recordar cuando declaramos variables de objeto JSP, la posibilidad de que múltiples *threads* accedan simultáneamente a la página JSP, representando varias solicitudes de la página. Así, si un elemento *scripting* de la página modifica el valor de una variable de objeto, todas las referencias posteriores a esa variable usarán el nuevo valor. Si deseas crear variables cuyo valor sea local al proceso de una única solicitud, tendrás que hacerlo en un scriplet.

Estas declaraciones son útiles para declarar variables de clase mediante el uso de *static*, si bien, como el contenedor JSP suele crear un único objeto de la clase *servlet* para representar una página particular, no hay muchas diferencias entre declarar variables objeto o variables de clase.

Declaración de métodos.

Los métodos definidos mediante las declaraciones se convierten en métodos de la clase *servlet* en la que la página JSP es compilada. Todo lo dicho sobre la declaración de variables puede ser aplicado para la declaración de métodos.

3.4.2. Expresiones.

Las expresiones están explícitamente proyectadas para generar salida. Su sintaxis es la siguiente:

<%= expresion %>

La expresión debe ser una expresión válida y completa en cualquiera de los lenguajes *scripting* que hayan sido especificados para la página. El efecto de este elemento es evaluar la expresión especificada y sustituir el resultado en la salida de la página en el lugar donde está el elemento en cuestión. Algunas expresiones válidas son:

<%= Math.PI %>

*<%= 4*5 %>*

Si la página incluyera la declaración de un método llamado *factorial(int numero)* que devolviera el factorial del entero que se le pasa como argumento, podríamos escribir la siguiente expresión:

<%= factorial(3) %>

Estos tres ejemplos devuelven valores numéricos, pero **NO HAY RESTRICCIONES** para los tipos de valores que puede devolver una expresión, así puede devolver números, caracteres, *booleans*... Todos los resultados de las expresiones son convertidas a cadena de caracteres antes de ser añadidas a la salida de la página.

3.4.3. *Scriptlets.*

Un *scriptlet* puede contener sentencias del lenguaje *scripting* válido para la página que son evaluadas por sus efectos laterales ya que no generan automáticamente salida como las expresiones. Su sintaxis es la siguiente:

<% scriptlet %>

Un *scriptlet* puede dejar abierto uno o más bloques de sentencias, los cuales deben ser cerrados en *scriptlets* posteriores en la misma página. En el caso de que el lenguaje *scripting* sea Java, los bloques de sentencias son abiertas mediante "{" y cerradas mediante el "}".

Un *scriptlet* de una página se ejecutará para cada solicitud recibida de esa página. Así, si en un *scriptlet* se crea un objeto, se creará un objeto por cada solicitud de esa página, y se podrá acceder a él, al igual que a cualquier variable definida en un *scriptlet*, en subsiguientes *scriptlet* y expresiones en la misma página.

Es posible controlar el alcance de una variable aprovechándose de la posibilidad que ofrece JSP de dejar bloques de código abierto a lo largo de muchos *scriptlets*. Veamos esto con un ejemplo.

<html>
<body>
<h1>Alerta Intruso</h1>
* Entrada no autorizada, cursando reconocedores... *
<% GameGrid grid=GameGrid.getGameGrid();

{ Reconocedor r1 = new Reconocedor(new Coordinates(grid,0,0));
Reconocedor r2 = new Reconocedor(new Coordinates(grid,100,100));
r1.encuentraPrograma("Toni");
r2.encuentraPrograma("Toni"); %>

<h2>Estado</h2>
<ul>
*<li>Primer Reconocedor:<%= **r1.informeEstado() %>***
*<li>Segundo Reconocedor:<%= **r2.informeEstado()%>***
*
*
<% } %>
*Nivel de alerta:<% **grid.nivelAlerta() %>***
</body>
</html>

En este ejemplo, los objetos *r1* y *r2* creados en el primer *scriptlet* sólo son accesibles desde que son creados hasta el penúltimo *scriptlet* donde se cierra el bloque en el que se hallan. Esto ocurre debido a cómo es traducido el contenido de una página JSP en el código fuente del *servlet*. El contenido estático, como el código HTML, es traducido en sentencias Java que imprimen ese texto como salida del *servlet*. Igualmente las expresiones son traducidas en sentencias Java que evalúan la expresión, convierten el resultado en una cadena, e imprimen ese valor como salida del *servlet*. Con los *scriptlets* ocurre que no hace ninguna traducción y son directamente insertados en el código del *servlet*.

ELEMENTOS *SCRIPTING* EN JSP		
Tipos de *scripting*		
Sintaxis	¿Qué hacen?	
Declaraciones	<%! declaración %>	Definen variables y métodos del *servlet*. Pueden ser accesibles desde cualquier punto de la página. No generan salida.
Expresiones	< %= expresión %>	Evalúan la expresión y envían el valor resultante a la salida. Generan salida.
Scriptlets	<% *scriptlet* %>	Hacen uso de métodos y variables definidos anteriormente en otro *scriptlet*o en cualquier parte de la página en una declaración. Pueden generar salida.

3.5 Control de flujo.

Esta capacidad de los *scriptlets* de introducir bloques de sentencia sin cerrarlos puede ser usada en las páginas JSP para afectar al flujo de control a través los diversos elementos, estáticos o dinámicos, que gobiernan la salida de la página.

3.5.1. Condiciones.

Podemos usar las sentencias *if*, con las cláusulas opcionales *else if* y *else* para introducir condiciones en el código de la página. Veámoslo con un ejemplo:

<% if (x<0) { %>
* Lo siento, no puedo calcular el factorial de un número negativo. *
<% } else if (x>20) { %>

* Lo siento, los argumentos mayores de 20 pueden causar un error de "overflow" *
<% } else { %>
* ***<%= x%>***! = ***<%= fact(x) %>*** ***
<% } %>

En este ejemplo vemos que el uso de sentencias de condición en los *scriptlet* permite gobernar que código estático HTML va a ser pasado a la salida de la página.

3.5.2. Iteración.

Del mismo modo podemos utilizar en los *scriptlets* las sentencias de iteración *for*, *while* y *do/while*, para añadir contenido iterativo a una página JSP, veámoslo con un ejemplo donde se crea una tabla.

<table>
<tr><th>x</th><th><I>x</I>! </th></tr>
<% for (long x=0; x<=19; x++) { %>

<tr><td> ***<%= x %>*** *</td><td>* ***<%= fact(x) %>*** *</td></tr>*

<% } %>
</table>

3.6 Comentarios.

Los comentarios son líneas de texto que se incluyen en el código de un programa para aclarar qué es lo que se pretende hacer. En JSP tenemos tres maneras diferentes de introducir comentarios.

3.6.1. Comentarios contenidos.

Son comentarios que aparecen en la salida de la página. Su sintaxis es la siguiente:

<!−−comment −−>

Podemos incluir contenido dinámico en estos comentarios.

<!−−El factorial de 20 es ***<%= fact(20) %>*** *−−>*

3.6.2. Comentarios JSP.

Los comentarios JSP son independientes del tipo de contenido producido por la página. También son independientes del lenguaje *scripting* usado por la página. Estos comentarios pueden ser vistos únicamente examinando el fichero JSP, y su sintaxis es:

<%−− comment −−>

El cuerpo de este comentario es ignorado por el contenedor JSP. Cuando la página es compilada en un *servlet*, cualquier cosa que aparezca entre estos dos delimitadores es obviada mientras se traduce la página en el código fuente del *servlet*.

3.6.3. Comentarios del lenguaje *scripting*.

Podemos utilizar los *scriptlets* para introducir comentarios del tipo del lenguaje *scripting* que se esté utilizando. Para el caso más corriente de lenguaje Java, la sintaxis quedaría:

<% / comment */ %>*

Estos comentarios no aparecerán en la salida de la página, pero aparecerán en el código fuente generado para el servlet.

Una expresión JSP que contenga sólo un comentario, pero ninguna expresión en lenguaje *scripting*, no será válida y nos dará un error de compilación.

Java nos ofrece una segunda forma de poner un comentario mediante los caracteres // que toma por comentario todo lo que hay desde este delimitador hasta el fin de la línea.

Dos al cuadrado es igual a ***<% = 2*2 %>*** *// ¡¡Vaya ejemplo estúpido!!*

TIPOS DE COMENTARIOS EN JSP		
Tipos de comentarios		
Sintaxis	*Scope* (dónde aparecen)	
Contenidos	<!−− comentario −−>	Forman parte de la salida.
Del lenguaje *scripting*	< %−− comentario −− >	Sólo aparecen en el código JSP (los ignora el contenedor JSP).
De JSP	< % /* comentario */ >	Aparecen en el código JSP y en el *servlet.*

4.1 Objetos implícitos.

El contenedor JSP ofrece un conjunto de objetos internos a la página del autor. Se les llaman objetos implícitos porque su disponibilidad en una página JSP es automática; el creador de una página puede asumir que estos objetos están presentes y accesibles mediante elementos *scripting* JSP. Más concretamente, estos objetos serán automáticamente asignados a nombres de variables específicas en el lenguaje *scripting* de la página. Además cada objeto implícito debe unirse a la correspondiente API en forma de una clase Java específica o definición de interface. Las clases correspondientes se pueden consultar en el manual que provee ***Sun Microsystems***.

Más concretamente, estos objetos serán automáticamente asignado a nombres de variables específicos en el lenguaje *scripting* de la página.

JSP ofrece nueve objetos implícitos agrupados en cuatro categorías principales:

- − Objetos relacionados con el *servlet* de la página:
  page, config.
- − Objetos relacionados con la salida o entrada de la página:
  request, response, out.
- − Objetos que proporcionan información del contexto dentro del cual la página está siendo procesada:
  session, application, pageContext.
- − Objetos resultantes de errores:
  exception.

Independientemente de esta clasificación, cuatro de estos objetos implícitos: *request, session, application* y *pageContext* tienen algo en común: la capacidad de almacenar y devolver valores de atributos. Así, mediante

el acceso en lectura y escritura de estos atributos, estos objetos son capaces de transferir información entre páginas JSP y *servlets* como un simple mecanismo de almacenamiento de datos. Para las clases e interfaces de estos cuatro objetos existen una serie de métodos estándar para el manejo de los atributos nombrados anteriormente, y aparecen en la tabla siguiente:

Métodos de manejo de atributos.

Método

Descripción

setAttribute(key, value)

Asocia el valor de un atributo con una clave (key).

getAttributeNames()

Devuelve los nombres de todos los atributos asociados con la sesión.

getAttribute(key)

Devuelve el valor del atributo asociado a la clave (key).

removeAttribute(key)

Elimina el valor del atributo asociado a la clave (key).

4.1.1 Objetos relacionados con el *servlet*.

Los dos objetos implícitos en esta categoría están fundamentados en la implementación de las páginas JSP en un *servlet*. El objeto implícito *page* representa el el *servlet* mismo, mientras que el objeto implícito *config* almacena los parámetros de inicialización del *servlet*, en caso de que exista alguno.

Objeto *page*.

Representa un objeto de la clase *servlet* en la que la página ha sido traducida, y como tal, puede ser usado para llamar a cualquiera de los métodos definidos para esta clase. Es un objeto raramente utilizado y por lo tanto no diremos nada más de él.

Objeto *config*.

Es un objeto que almacena datos de configuración del *servlet* en el que la página JSP es compilada. Este objeto tampoco se utiliza mucho por lo que no entraremos más en él.

4.1.2 Objetos relacionados con la salida o la entrada de la página.

Estos objetos están enfocados en la entrada y salida de una página JSP. Más concretamente, el objeto *request* representa los datos que vienen en la página, mientras que *response* representa su resultado. El objeto *out* representa la actual cadena de salida asociada con *response*, en la que el contenido de la página es escrito.

Objeto *request*.

Representa la solicitud que lanzó el proceso de la actual página. Para las solicitudes HTTP, este objeto proporciona acceso a toda la información asociada con una solicitud, incluyendo su fuente, el URL solicitado, y cualquier cabecera, *cookie* o parámetro asociado con la solicitud.

Es uno de los cuatro objetos que soportan atributos mediante los siguientes métodos:

Métodos para acceder a los parámetros del *request*

Método

Descripción

getParameterNames()

Devuelve el nombre de todos los parámetros de la solicitud.

getParameter(name)

Devuelve el primer valor de un parámetro de la solicitud.

getParameterValues(name)

Devuelve todos los valores de los parámetros de la solicitud.

Existen más métodos asociados a este objeto, y se pueden ver en el manual de ***Sun Microsystems***.

Objeto *response*.

Representa el la respuesta que será enviada al usuario como resultado del procesado de la página JSP. Existen muchos métodos asociados a este objeto y de nuevo remito al manual.

Objeto *out*.

Representa la cadena de salida de la página. Es un objeto particularmente interesante ya que nos permite acceder a la salida que ve el usuario. Hereda todos los métodos estándar *write()* proporcionado por *java.io.Writer* y también implementa todos los métodos *print()* y *println()* definidos por *java.io.PrintWriter*.

Un ejemplo de uso del objeto *out* sería el siguiente.

```
<P>Contando huevos
<% int contados=0;

while (carton.tengaOtro()) {
contados++;
out.print(".");
}

%>
<BR>
Hay <%= contados %> huevos.
```

El *scriplet* en este ejemplo va contando huevos y a medida que cuenta uno, accede al objeto *out* con el método *print()* para escribir un punto en la cadena de salida de la página.

Una importante ventaja de este objeto es la posibilidad de generar salida sin tener que salir del cuerpo de un *scriptlet*. Hasta ahora no era posible generar salida desde un *scriptlet*, pero accediendo al objeto implícito *out*, y más concretamente a sus métodos, esto ya será posible.

De nuevo existen numerosos métodos más de los vistos que pueden ser consultados en el manual.

4.1.3.− Objetos contextuales. (Ir al índice)

Estos son objetos que permite a la página acceder al contexto en el cual está siendo procesada. El objeto *session* representa el contexto para la solicitud a la cual la página está respondiendo, mientras que *application*, es un objeto que representa el contexto referente al servidor en el que la página se está ejecutando. Por último, el objeto *pageContext* está centrado en el contexto de la página en sí mismo.

Objeto *session.*

Representa una sesión actual de usuario. Todas las solicitudes hechas por un usuario que son parte de una única serie de interacciones con el servidor, se considera que son parte de una sesión. Mientras que nuevas peticiones de ese usuario sigan siendo recibidas por el servidor, la sesión persiste. Si, sin embargo, pasa un periodo de tiempo sin ninguna solicitud desde el usuario, la sesión termina.

Los métodos asociados a este objeto nos permitirán acceder a: el identificador de la sesión, la hora en la que la sesión empezó, la hora a la cual se recibió la última petición,...

Junto con el objeto *exception*, son los únicos que no están automáticamente disponibles para todas las páginas JSP. El acceso a este objeto está restringido a las páginas que participan en la administración de sesión. Esto se indica por medio del atributo *session* de la directiva *page*. Cierto es, que por defecto, todas las páginas participan en la administración de sesión, pero si el atributo *session* de la directiva *page* está fijado como *null*, no se podrá acceder a el objeto *session*.

Objeto *application.*

Representa la aplicación a la que la página JSP pertenece, referente al servidor en el que la página está siendo ejecutada.

Los métodos asociados a este objeto, nos permitirán acceder a diversa información referente al contenedor *servlet*, a los recursos del servidor representados como nombres de ficheros y URLs...

La lista de métodos se puede consultar en el manual.

Objeto *pageContext.*

Este objeto proporciona acceso a todos los otros objetos implícitos. Para los objetos que soportan atributos, el objeto *pageContext* proporciona métodos para acceder a dichos atributos. Además implementa métodos para transferir el control desde la página actual a otra página.

Los métodos para acceder a los otros objetos son los siguientes:

Métodos de la clase *pageContext* para el acceso a objetos JSP

Método

Descripción

getPage()

Devuelve el objeto *servlet* de la página actual

getRequest()

Devuelve el objeto request que inició el proceso de la página actual.

getResponse()

Devuelve el objeto *response* de la página actual.

getOut()

Devuelve el objeto *out* de la página actual.

getSession()

Devuelve el objeto *session* asociado a la página actua

getServletConfig()

Devuelve el objeto *config* del *servlet* actual.

getServletContext()

Devuelve el objeto *application* asociado al *servlet* de la página.

getException()

Para errores de página devuelve la *exception* pasada a la página.

Los métodos para transferir el control serán:

Métodos de la clase *pageContext* para transferir el control.

Método

Descripción

forward(path)

Envía el proceso a otro URL local.

include(path)

Incluye a la salida del proceso otro URL local.

Además existen otros métodos relacionados con el acceso a los atributos y que podemos consultar en el manual.

4.1.4 Objetos para el manejo de errores.

Esta categoría tiene un único objeto implícito, *exception*, y proporciona métodos para el manejo de errores dentro de JSP.

Objeto *exception.*

Junto con el atributo *session* son los únicos objetos implícitos que no son automáticamente accesibles por cualquier página JSP. Sólo aquellas que hayan sido designadas como *error pages* usando el atributo *isErrorPage* de la directiva *page*.

Los métodos asociados a este objeto nos permitirán acceder a la información referente a los posibles errores que hayan podido surgir.

Métodos asociados al objeto *exception.*

Método

Descripción

getMessage()

Devuelve un mensaje descriptivo del error asociado a la excepción.

toString()

Devuelve un *string* combinando el nombre de la clase de la excepción con su mensaje de error.

4.2 Acciones. (Ir al índice)

Las acciones eran la cuarta y última categoría principal de las etiquetas JSP. Las otras tres eran las directivas, elementos *scripting* y los comentarios. Las acciones tienen tres características principales. La primera es que nos permiten transferir el control entre páginas. La segunda es que soportan las especificaciones para los *applets* de Java de una manera independiente del navegador. Por último, las acciones permiten a las páginas JSP interactuar con los objetos *JavaBean* que residen en el servidor.

4.2.1 *forward.*

La acción *<jsp:forward>* se usa para transferir el control de una página JSP a otra localización del servidor local. La sintaxis es la siguiente:

<jsp:forward page="localURL" />

El atributo *page* se usa para especificar la localización a la que se le transferirá el control, de forma que si en una página JSP se llega a una acción *forward* cualquier contenido generado por la página actual será descartado, y el proceso de la solicitud empezará de nuevo en la localización alternativa.

Con la acción *forward* tenemos la posibilidad de dar valores de atributo en tiempo de respuesta mediante el uso de expresiones.

*<jsp:forward page=´ <%= "**mensaje"+ codigoEstado + ".html" %**> ´ />*

Cada vez que la página sea procesada, esta expresión será evaluada por el contenedor JSP.

Para el caso de que el control sea transferido a otra página JSP, el contenedor JSP asignará automáticamente un nuevo objeto *pageContext* a esta página. Los objetos *request* y *session*, sin embargo, serán los mismos para ambas páginas. El objeto *application* puede o no puede ser el mismo dependiendo de si ambas páginas pertenecen a la misma aplicación.

Como el objeto *request* es igual para ambas páginas, es posible especificar nuevos parámetros que sean disponibles para la página a la que se le transfiere el control a través del uso de *<jsp:param>* dentro de cuerpo de la acción *<jsp:forward>*.

<jsp:forward page="localURL">
<jsp:param name="parametro1" value="valorParametro1" />
<jsp:param name="parametro2" value="valorParametro2" />
.....
<jsp:param name="parametroN" value="valorParametroN" />
</jsp:forward>

4.2.2 *include.*

Esta acción permitirá incorporar el contenido generado por otro documento local a la salida de la página actual.

<jsp:include page="localURL" flush="true"/>

En este caso, al llegar a una acción *include* en nuestra página JSP, el control es transferido a la página señalada por el atributo *page*, se incluye el contenido generado por esta página y se devuelve el control a la página principal justo después de donde apareció la etiqueta *<jsp:include>*.

Esta acción también permite dar valores al atributo *page* en tiempo de respuesta. Al igual que con las páginas que se acceden vía *<jsp:forward>*, a las páginas JSP procesadas a partir de un *<jsp:include>* se les asignará un nuevo objeto pageContext, pero compartirán los mismos objetos *request* y *session*. Igualmente al objeto *request* también se le pueden añadir nuevos parámetros mediante la etiqueta *<jsp:param>*.

<jsp:include page="localURL" flush="true">
<jsp:param name="parametro1" value="valorParametro1" />
<jsp:param name="parametro2" value="valorParametro2" />
<jsp:param name="parametroN" value="valorParametroN" />
</jsp:include>

4.2.1 *plug−in.*

La acción *<jsp:plugin>* se usa para generar HTML específico del navegador para especificar *applets* Java. Debido a que JSP está centrado en el lado del servidor, no entraremos en los detalles de esta acción que está orientada al lado del cliente.

4.2.1 Etiquetas de *Beans.*

JSP proporciona tres diferentes acciones para interactuar con *JavaBeans* en residentes en el servidor *<jsp:useBean>*, *<jsp:setProperty>*, *<jsp:getProperty>*. Estas acciones son muy importantes pues nos

permiten realizar un diseño centrado en los componentes, y por lo tanto separar la lógica de presentación de la lógica de aplicación. Los dos próximos temas se centrarán en los *beans* y en estas acciones.

(Ir al índice)

5. USANDO COMPONENTES DE JSP

El uso de los *scriptlets* y de las expresiones vistos en el tema anterior nos permitirán añadir contenido dinámico a una página web. Sin embargo, el uso de estos elementos, carece de la elegancia de una separación entre la presentación y la implementación ya que exige conocimientos de Java a los diseñadores. JSP proporciona un método centrado en los componentes, de diseño de páginas web dinámicas. JSP nos permite interactuar con objetos Java sin tener que saber nada de Java, simplemente con el uso de etiquetas.

5.1 El modelo de componentes de JSP.

El modelo de componentes de JSP está centrado en unos componentes *software* llamados *JavaBeans*. Antes de entrar en detalle en estos *JavaBeans* vamos a ver un poco en qué consiste la arquitectura basada en componentes.

En una arquitectura basada en componentes, estos componentes son elementos *software* autocontenidos y reutilizables. De cara al exterior se comportan como cajas negras de la que conocemos qué hace pero no cómo lo hace. Esta característica, convierte a los componentes en elementos reutilizables no específicos de ninguna página en concreto.

La principal ventaja de este tipo de arquitectura es la separación entre los aspectos meramente de presentación y los relacionados con la implementación. Los diseñadores de páginas web pueden realizar páginas de contenido dinámico, sin tener que tener un conocimiento exhaustivo del lenguaje *scripting* que esté utilizando la página (normalmente Java). Mientras que un programador sólo tendrá que ir desarrollando componentes, indicando cómo se puede acceder a ellos. Como vemos, una arquitectura basada en componentes permitirá la división de labores en el trabajo de desarrollo de una página web de contenido dinámico.

5.2 Fundamentos de los *JavaBeans.*

Los *JavaBeans* son **componentes *software* escritos en Java que deben cumplir una serie de especificaciones detalladas en la *JavaBean API***. Usando una serie de etiquetas que nos proporciona JSP, podremos acceder a estos *Beans* y por lo tanto a las utilidades que nos ofrecen sin tener que conocer nada de Java.

Contenedor *Bean.*

El contenedor *Bean* es una aplicación, lenguaje de programación o entorno que permite a los diseñadores llamar a los *Beans*, configurarlos y acceder a su información y comportamiento. Es este contenedor el que permitirá el uso de los *Beans* en una página JSP.

Propiedades de los *Beans.*

Los *Beans* están caracterizados por una serie de atributos a los que llamamos propiedades. El contenedor *Bean* nos permitirá acceder a ellos en tiempo de ejecución para controlar el comportamiento del *Bean*. Estas propiedades serán el único mecanismo que el contenedor *Bean* usa para exponer el *Bean* al exterior.

Quizá se vea todo esto más claro con un ejemplo. Supongamos que tenemos un pequeño programa en Java que ha cumplido todos los requisitos para ser un *Bean*. Este programa servirá para dar la predicción de la

temperatura en los próximos cinco días: máxima temperatura pronosticada y mínima temperatura pronosticada. Cada uno de estos datos será una propiedad de nuestro *Bean* al que llamaremos *PredicBean*: *maxTemp* y *minTemp* Otra propiedad será el código postal de la zona del país de la que queremos conocer los pronósticos, *codigoPostal*.

La naturaleza de las propiedades (*property*) dependerá de cada *Bean* y así encontraremos propiedades a las que sólo podremos acceder en lectura, o sólo en escritura, o incluso en lectura y escritura. En nuestro ejemplo, es lógico pensar que las propiedades *maxTemp* y *minTemp* serán configuradas como de sólo lectura, mientras que *codigoPostal* será configurada o como de sólo escritura, o como de escritura y lectura.

Propiedades *trigger* y *linked*.

Algunas propiedades son usadas para lanzar el comportamiento de un *Bean*. Son propiedades tales que al acceder a ellas (bien sea en lectura o en escritura) provocan que el *Bean* realice una operación.

En el ejemplo *PredicBean* una propiedad *trigger* sería *codigoPostal*, al modificarla, el *Bean* respondería hallando la predicción (no nos importa cómo lo hace) para la zona de ese código postal. Además, diremos que las propiedades *maxTemp* y *minTemp* junto con la propiedad *codigoPostal* son propiedades *linked*, ya que al modificar esta última, se modifican las dos primeras.

Propiedades *indexed*.

Son propiedades que almacenan una colección de valores. La forma de acceder a ellas será a partir de un índice, de ahí que se llamen propiedades *indexed*. En nuestro caso, como la predicción es la de los próximos cinco días, ambas propiedades relativas a la temperatura serán propiedades *indexed* cuyo índice irá de 0 a 4, apuntando a cada uno de los cinco días.

Con respecto a las propiedades *indexed* hay que decir que no todos los contenedores *Bean* proporcionan mecanismos que nos permitan trabajar con estas propiedades mediante etiquetas normales. Esto no significa que no podamos utilizarlas, sino que para poder hacerlo tendremos que recurrir a los *scriptlets* JSP, las expresiones JSP o las etiquetas de cliente.

Tipos de datos de las propiedades.

Las propiedades de los *Beans* pueden almacenar cualquier tipo de dato primitivo de Java como *int* o *double*, así como objetos Java tales que *Strings* o *Dates*. Incluso pueden almacenar objetos definidos por el usuario e incluso otros *Bean*.

Con los *scriptlets* JSP y las expresiones JSP referenciamos las propiedades por el tipo de dato Java. Si una propiedad almacena un entero, nosotros le enviaremos enteros y leeremos enteros. Sin embargo, con las etiquetas *Bean*, trataremos a las propiedades como si almacenaran un *string*. Cuando accedemos a una propiedad en escritura, le pasamos un *string*, y cuando leemos el contenido, estamos leyendo un *string*.

Independientemente del tipo de dato que almacene una propiedad de un *Bean*, cuando accedemos a ellos mediante etiquetas *Bean* ESTAMOS ACCEDIENDO A DATOS DE TIPO *string*.

El contenedor JSP realiza automáticamente todas las conversiones de tipo necesarias. Cuando accedemos en escritura a una propiedad que almacena un entero, el contenedor JSP realiza las llamadas Java necesarias para convertir las series de caracteres numéricos que le damos, en un valor entero.

Una opción inteligente que puede tomar el diseñador de *Beans* es controlar el valor de las propiedades, aceptando valores *string* para propiedades no *string* y realizar ellos mismos la conversión. Esto será obligatorio para valores que no correspondan a un tipo primitivo Java.

5.3 Etiquetas *Beans* de JSP.

Una vez que hemos entendido en qué consiste la arquitectura basada en componentes es el momento de entrar en detalle en los *JavaBeans* y más concretamente en las etiquetas JSP que nos permiten acceder a las propiedades de estos. Estas etiquetas se conocen como Etiquetas *Bean* de JSP.

5.3.1 Accediendo a los componentes JSP.

Lo primero que tenemos que hacer para poder utilizar un *Bean* en la página JSP, es decirle a ésta dónde encontrar el fichero Java *class* que define al *Bean* y asignarle un nombre. No hay que perder de vista que un *Bean* no es más que una clase Java que por el hecho de cumplir una serie de especificaciones el contenedor *Bean* permitirá a la página JSP acceder a ellos, o más bien, a los valores almacenados en sus propiedades, mediante una serie de etiquetas similares al resto de etiquetas de JSP. Veamos cuál es la etiqueta que nos permitirá hacer esto.

La etiqueta: *<jsp:useBean>*

La etiqueta *<jsp:useBean>* dice a la página que queremos hacer a un *Bean* disponible para la misma. Esta etiqueta aparece de dos formas; la primera en forma de una etiqueta vacía:

<jsp:useBean id="nombre del *Bean*" *class*="nombre de la clase del *Bean*" / >

La segunda en forma de dos etiquetas, una de comienzo y otra de final, que encierran a un cuerpo que puede ser usado para especificar información adicional de configuración:

<jsp:useBean id="nombre del *Bean*" *class*="nombre de la clase del *Bean*">

código de inicialización

</ jsp:useBean>

Atributos de la etiqueta: *<jsp:useBean>*

El atributo *id.*

El atributo *id* sirve para especificar el nombre que vamos a utilizar en la página JSP para referirnos al *Bean*. Debe cumplir que:

- El nombre debe ser único en la página.
- El primer caracter debe ser una letra.
- No puede contener espacios, sólo letras, números y subguiones (_).
- Es sensible a las minúsculas y mayúsculas. Suelen empezar por minúscula.

El atributo *class.*

Este atributo especifica el nombre de la clase del *JavaBean*. Hay que permitir que la página tenga acceso a esa clase, para ello o bien al referirnos a esta clase lo hacemos con su nombre completo, incluyendo el nombre de los paquetes (*package*) de clase y el nombre de la clase en cuestión. Por ejemplo, si la clase de nuestro *Bean*,

llamada *NuestroBean* pertenece al *package libreriaBeans* que a su vez pertenece al *package componentes*, su nombre completo sería:

componentes.libreriaBeans.NuestroBean

Otra forma posible de referirnos a la clase del *Bean* sin tener que dar el nombre completo, es utilizar el atributo *import* de la directiva *page*. Así, si declaramos:

<% @page import="componentes.libreriaBeans.NuestroBean.class" %>

Entonces podremos referirnos a la clase *NuestroBean* en el resto de la página sin tener que referenciar los paquetes a los que pertenece.

Veamos ahora algunos aspectos interesantes relativos a este atributo. Las páginas JSP debemos incluirlas en algún directorio específico del servidor. Igual ocurrirá con las clases que utilicen estas páginas JSP, si no especificamos al servidor que haga otra cosa, irá a buscarlas a un directorio específico, y por lo tanto las clases deberán estar en ese directorio.

Otro aspecto a recalcar es que las clases que metamos en el directorio en cuestión tendrán que estar ya compiladas y por lo tanto con su extensión correspondiente *.class*.

Por último, reseñar que el hecho de que podemos utilizar varios objetos de una misma clase sin más que utilizar *id* distintos para cada uno de ellos.

El cuerpo de la etiqueta.

Ya vimos que se trata de algo opcional cuando usamos la etiqueta *<jsp:useBean>*. Básicamente su misión consiste en inicializar el *Bean*, y más concretamente sus propiedades.

Como resumen de esta importantísima etiqueta, diremos que si escribimos lo siguiente:

<jsp:useBean id="pronostico" class="componentes.LibreriaBeans.PronosticoBean" />

Estaremos creando un objeto del *Bean* dado por la clase *PronosticoBean* asignándole un identificador llamado *pronostico*.

Atributos de la etiqueta *<jsp:useBean>*	
Atributo	
Descripción	
id	Le da un nombre al objeto de la clase *Bean* que servirá para referenciarlo a lo largo de la vida del *Bean*.
class	Especifica la clase Java que implementa al *Bean*
scope	Especifica el alcance que tendrá el *Bean*. Puede tomar los valores: *page, session, request, application*

La etiqueta: *<jsp:getProperty>*

Una vez que hemos informado a la página que vamos a utilizar un *Bean* querremos acceder a sus propiedades. Para ello utilizaremos la etiqueta *<jsp:getProperty>*, que tiene la siguiente sintaxis:

<jsp:getProperty name="nombre del Bean" property="nombre de la propiedad" />

Con esta etiqueta, accederemos en lectura a la propiedad indicada por el atributo *property* del *Bean* dado en el atributo *name*. El valor dado en el atributo *name* debe coincidir con el señalado en el atributo *id* de la etiqueta *<jsp:useBean>* en la que se declara el *Bean*.

La etiqueta *<jsp:useBean>* genera contenido en la página de salida. Recordemos que aunque las propiedades de los *Beans* pueden almacenar cualquier tipo de datos, al acceder a ellas en lectura mediante etiquetas *Bean*, estas nos devolverán un *string*.

Supongamos que tenemos una página JSP que hace uso del *Bean mihora*, el cual posee una propiedad de sólo lectura llamada *hora* que nos devuelve la hora actual. La forma de acceder a este *Bean* es como sigue.

<jsp:useBean id="mihora" class="componentes.LibreriaBeans.HoraBean">
<html>
<body>
El Bean dice que la hora actual es:
< jsp:getProperty name="mihora" property="hora" />
<html>
<body>

Este código JSP generaría el siguiente código HTML en el lado del usuario.

<html>
<body>
El Bean dice que la hora actual es: 11:41 am
<html>
<body>

Un aspecto muy importante de las etiquetas *Bean* es que podemos usarlas para controlar atributos de HTML, y así, si contáramos con un *Bean* con una propiedad *randomcolor* que al acceder a ella, nos devolviera un código RGB aleatorio (una forma de determinar un color es mediante el código RGB), sería totalmente válido hacer el siguiente acceso a esa propiedad:

*<body bgcolor="**<jsp:getProperty name="color" property="randomcolor" />**" >*

Atributos de la etiqueta *<jsp:getProperty>*	
Atributo	
Descripción	
name	Nombre con el que referenciamos al *Bean* al que pretendemos acceder. Debe coincidir con el atributo *id* de la etiqueta *<jsp:useBean>* en la que se declaró el *Bean*.
property	Especifica la propiedad del *Bean* a la que queremos acceder en lectura.

La etiqueta: *<jsp:setProperty>*

Esta etiqueta nos servirá para modificar el valor de las propiedades de los *Beans*. Para ello utilizaremos la etiqueta *<jsp:setProperty>*, que tiene la siguiente sintaxis:

<jsp:setProperty name="nombre del Bean" property="nombre de la propiedad" value="valor que pasamos" />

Los atributos *name* y *property* tienen el mismo significado que para la etiqueta *<jsp:getProperty>*. El atributo *value* es texto que pasamos a la propiedad, y también puede ser calculado en tiempo de ejecución mediante

expresiones JSP. Veamos esto con dos ejemplos igualmente válidos.

<jsp:setProperty name="usuario" property="horastrabajo" value="30" />

*<jsp:setProperty name="usuario" property="horastrabajo" value="<%= 15 * 2 %>" />*

Una vez que un *Bean* ha sido introducido en una página JSP, la etiqueta *<jsp:setProperty>* puede ser utilizada en cualquier punto del documento, y en tiempo de ejecución se evaluarán secuencialmente. Es importante tener en cuenta que modificar el valor de una propiedad puede dar lugar a una serie de acciones en el *Bean*, esto tiene que ver con lo que vimos sobre propiedades *trigger*, por ello hemos de ser cuidadosos y conocer en detalle cómo el *Bean* reacciona ante accesos en escritura de cualquiera de sus propiedades.

Atributos de la etiqueta *<jsp:setProperty>*	
Atributo	
Descripción	
name	Nombre con el que referenciamos al *Bean* al que pretendemos acceder. Debe coincidir con el atributo *id* de la etiqueta *<jsp:useBean>* en la que se declaró el *Bean*.
property	Especifica la propiedad del *Bean* a la que queremos acceder en lectura.
value	Es texto que queremos pasarle como valor a la propiedad. Puede ser directamente texto, o incluso una expresión JSP que se evaluará en tiempo de ejecución.
param	Sirve también para pasarle un valor a la propiedad, pero el valor lo toma de los parámetros que se le pasan a la página desde la solicitud. Es equivalente a *<%= request.getParameter("nombre del parámetro") %>*

Propiedades *indexed.*

Ya hemos hablado de la posibilidad de que alguna propiedad almacenara un colección de valores. También dijimos que **las etiquetas *Bean* no pueden manejar propiedades indexadas**. Esto no significa que no podamos contar con ellas, sino que para su manejo tendremos que hacer uso de los *scriptlets*, expresiones o etiquetas de cliente JSP.

En este caso, accederemos al *Bean* no como un *Bean* sino como el objeto que creamos con la etiqueta *<jsp:useBean>*. Veamos esto con un ejemplo. Supongamos una clase *Bean*, llamada *EquipoBean*, que entre sus propiedades cuenta con una indexada llamada *jugadores* que representa a los once jugadores del equipo inicial. Esta propiedad será tanto de escritura como de lectura. Inicialmente declararemos al *Bean*:

<jsp:useBean id="equipo" class="EquipoBean" />

Con esto he creado un objeto llamado *equipo* de la clase *EquipoBean*. Doy por hecho que se ha definido la clase de forma que el contenedor *Bean* la reconoce como tal, y por esto, y porque la propiedad indexada *jugadores* es de escritura y lectura, la clase *EquipoBean* tendrá definida dos métodos que serán: (ya veremos como esto es así)

getJugadores() : de acceso en lectura a la lista de once jugadores.
setJugadores() : de acceso en escritura a la lista de once jugadores.

Veamos ahora un ejemplo de cómo acceder en lectura a esta propiedad.

<html>
<body>
<jsp:useBean id="equipo" class="EquipoBean" />

*El equipo inicial es:
*
<UL>
<% for(int indice=0; indice<11; indice++) { %>
*<li>**<%= equipo.getJugadores(indice)%>***
<% } %>
</UL>
</body>
</html>

Sin embargo ya veremos como se pueden diseñar los *Beans* para trabajar más fácilmente con las propiedades indexadas sin tener que hacer uso de los *scriptlets*. Para ello los *Beans* incluirán propiedades que nos permitirán tratar las propiedades indexadas cómo si fuera un *string* simple mediante la separación con comas u otro delimitador.

5.3.2 Inicializando *Beans.*

Cuando se explicó la sintaxis de la etiqueta *<jsp:useBean>* vimos cómo existían dos posibilidades. En una de ellas la etiqueta tenía un cuerpo que dijimos se utilizaba para inicializar el *Bean*.

<jsp:useBean id="nombre del *Bean*" *class*="nombre de la clase del *Bean*">

código de inicialización

</ jsp:useBean>

Esta inicialización ocurre sólamente la primera vez que el *Bean* es creado. Por defecto, esta inicialización tendrá lugar cada vez que se accede a la página ya que un *Bean* se crea por cada solicitud. Sin embargo ya veremos como un *Bean* puede ser almacenado y retomado desde el entorno del servidor web, en cuyo caso no será necesario que se reinicialice.

El proceso de inicialización suele consistir en dar valores a las propiedades mediante la etiqueta *<jsp:setProperty>*. Cualquier comando que forma parte del cuerpo es procesado inmediatamente después de que el *Bean* es creado y antes de que sea disponible para el resto de la página. Por ejemplo:

<jsp:useBean id="nombre del Bean" class="nombre de la clase del Bean">

<jsp:setProperty name="nombre del Bean" property="nombre de la propiedad" value="valor">

</ jsp:useBean>

En el cuerpo de inicialización del *Bean* podemos incluir etiquetas propias de HTML o bien *scriptlets* JSP. Este HTML será mostrado como parte de la página sólo si el *Bean* es creado, si ya existiera no se mostraría.

Inicializando *Beans* desde la solicitud de la página.

Una importante característica de la etiqueta *<jsp:setProperty>* es la posibilidad de acceso en escritura a las propiedades en tiempo de ejecución. Esto nos va a permitir inicializar los *Beans* dinámicamente gracias a datos de usuario o sucesos relativos a la solicitud de la página. Así, cumpliendo una serie de reglas será posible usar un formulario HTML para permitir a los usuarios de una forma cómoda dar valores a las propiedades.

Veamos cómo podemos dar valores a propiedades de un *Bean* a través de un formulario mediante un ejemplo

simple. Queremos calcular el la cantidad anual de dinero que hemos de pagar al solicitar un préstamo. Para ello tenemos que conocer la cantidad prestada, el interés anual y el número de años a pagar. En primer lugar crearemos el formulario que pide todos estos datos.

Prestamo.html

```
<html>
<body>
<form action="ResultadoPrestamo.jsp">
Cantidad de dinero pedida:<input type="text" name="prestamo">
Tasa de interés:<input type="text" name="tasaInteres">
Años de pago:<input type="text" name="anosPago">
<input type="submit" value="Calcula">
</form>
</body>
</html>
```

Este formulario hará una llamada a la página JSP *ResultadoPrestamo.jsp* pasándole como parámetros: *prestamo, tasaInteres* y *anosPago*. Si, por ejemplo, si se hubiera rellenado un formulario como el que sigue:

Cantidad de dinero pedida: Tasa de interés: Años de pago:

Es como si hubiéramos hecho la siguiente solicitud:

http://.../ResultadoPrestamo.jsp?prestamo=100000&name=5&anosPago=10

Así, pasaremos los datos obtenidos en el formulario a la página JSP *ResultadoPrestamo*. En esta página JSP, y usando la arquitectura basada en componentes, llamaremos a un *Bean* que se encargará de hacer los cálculos. Nosotros no entraremos en cómo hace estos cálculos el *Bean* y nos limitaremos a conocer las propiedades que posee.

Tabla de propiedades de *PrestamoBean*			
Nombre	**Tipo de acceso**	**Tipo Java**	**Ejemplo**
prestamo	lectura/escritura	*double*	100.5
tasaInteres	lectura/escritura	*double*	.1
anosPago	lectura/escritura	*int*	10
pagoAnual	lectura	*String*	100.5

La propiedad *pagoAnual* es una propiedad *linked* a las otras, de forma que *PrestamoBean* nos devolverá la cantidad de dinero anual que hemos de pagar mediante esa propiedad. Veamos cómo sería la página JSP que manejara este formulario y llamara a *PrestamoBean* para realizar los cálculos.Es interesante fijarse en cómo pasa los valores a las propiedades.

ResultadoPrestamo.jsp

```
<html>
<body>
<%@ page import="componentes.LibreriaBeans.*" %>
```

<jsp:useBean id="prestamo" class="PrestamoBean" />
<jsp:setProperty name="prestamo" property="prestamo" param="prestamo" />
<jsp:setProperty name="prestamo" property="tasaInteres" param="tasaInteres" />
<jsp:setProperty name="prestamo" property="anosPago" param="anosPago" />
La cantidad de dinero que tienes que pagar durante los próximo/s ***<jsp:setProperty name="prestamo" property="anosPago" />*** *año/s es de* ***<jsp:setProperty name="prestamo" property="pagoAnual" />*** *pesetas.*
</body>
</html>

Vemos que hemos utilizado el atributo *param=* de la etiqueta *<jsp:setProperty>* para pasarle los valores al *Bean*. Este atributo es equivalente al *scriptlet* JSP *<%= request.getParameter()>*.

En este caso hemos llamado igual a las propiedades que a los parámetros pasados desde la solicitud. ESTO NO ES OBLIGATORIO, pero permite una forma de pasar los parámetros más fácil y cómoda, consistente en omitir el parámetro:

<jsp:setProperty name="prestamo" property="prestamo" />
<jsp:setProperty name="prestamo" property="tasaInteres" />
<jsp:setProperty name="prestamo" property="anosPago" />

Incluso si, como es nuestro caso, tenemos varios campos del formulario que se "mapean" directamente sobre propiedades *Bean* podemos hacer uso del caracter ´*´.

<jsp:setProperty name="prestamo" property="" />*

El uso del asterisco indica que toda aquella propiedad del *Bean* cuyo nombre se corresponda con el nombre de un parámetro de la solicitud, tome el valor de este.

5.3.3 Controlando el alcance de los *Beans.*

Ya sabemos que cada vez que una página es solicitada, se crea un objeto *Bean*. Sin embargo, un *Bean* puede continuar existiendo más allá del alcance de la solicitud de una página. Estos *Beans* se almacenan en el entorno del servidor, y son reutilizados en varias solicitudes de la misma página. Esto nos permitirá crear un *Bean* una vez, y acceder a él a través de la visita de un usuario a nuestro sitio, y así cualquier modificación de una propiedad permanecerá mientras el *Bean* exista.

El alcance de un *Bean* se controla gracias al atributo *scope* de la etiqueta *<jsp:useBean>*. La tabla siguiente muestra los posibles valores que puede tomar este atributo.

Valores del atributo *scope.*		
Valor	*Accesibilidad*	*Tiempo de vida*
page	Página actual sólamente.	Hasta que la página sea mostrada o el control pasado(*forward*) a otra página.
request	Página actual y cualquier página incluida (*include*) o a la que se le pasa (*forward*) el control.	Hasta que la solicitud haya sido completamente procesada y la respuesta haya sido enviada al usuario.
session	La actual solicitud y cualquiera de las siguientes solicitudes desde la misma ventana del navegador.	La vida de la sesión del usuario.
application		La vida de la aplicación.

	La actual solicitud y futuras solicitudes que sean parte de la misma aplicación web.	

No entraré más en detalle sobre este aspecto de los *Beans*. Para profundizar más se puede consultar el manual de usuario de ***Sun Microsystem***.

En este capítulo se pretende ayudar a crear sus propios *Beans* para usarlos como componentes así como enseñar cómo se implementan.

6. DESARROLLO DE COMPONENTES DE JSP

6.1 ¿Qué hace a un *Bean* ser un *Bean*?

Ya en en capítulo anterior vimos cómo los *Beans* no eran más que clases Java que cumplían una serie de requisitos para que el contenedor *Bean* los reconociese como un *Bean*. Así, las propiedades no eran más que los métodos de la clase, y vimos como era posible acceder a estos métodos como se acceden a los métodos de las clases Java, y así para tratar con propiedades indexadas, una posibilidad era acceder a ellas como un método:

*<li>**<%= equipo.getJugadores(indice)%>***

En definitiva, un *Bean* no es más que una clase simple Java que sigue una serie de normas simples en el nombrado de sus métodos y unas convenciones de diseño perfiladas por las especificaciones de los *JavaBeans*. Si una clase sigue estas normas y la tratamos como un *Bean*, entonces es un *Bean*.

6.1.1 Convenciones *Bean*.

Las convenciones *JavaBean* son las que nos capacitarán a usar los *Bean* ya que son las que permiten al contenedor *Bean* analizar una clase Java como un *Bean* e interpretar sus métodos como propiedades.

La API *JavaBean*.

Si seguimos las convenciones especificadas por la API *JavaBean*, permitiremos al contenedor JSP interactuar con los *Beans* a un nivel programático, incluso aunque la aplicación que lo contiene no tenga una comprensión real de lo que el *Bean* hace o cómo trabaja. Para JSP, nos interesará los aspectos de la API que dictan cómo accederemos a las propiedades.

Los *Beans* son objetos.

Si hemos dicho que un *Bean* es una clase Java, entonces una instancia de un *Bean* no es más que un objeto Java, y por tanto, siempre tendremos la opción de acceder a sus métodos a través de código Java en otras clases o a través de elementos *scripting* JSP.

Una instancia de un *Bean* es un objeto Java, y por tanto: SIEMPRE PODREMOS ACCEDER A SUS MÉTODOS A TRAVES DE ELEMENTOS *SCRIPTING*.

Normas en el nombrado de las clases de los *Beans*.

Hasta ahora, a las hora de nombrar clases *Beans* hemos introducido la palabra *Bean* en su nombre, como: *predicBean, HoraBean* y *EquipoBean*. Este es un método común que permitirá distinguir aquellas clases que son creadas siguiendo las reglas de los *Bean*. Sin embargo, esto no es obligatorio.

No es obligatorio introducir la palabra *Bean* en el nombre de la clase.

6.1.2 Constructores *Bean.*

La primera regla es que el *Bean* debe implementar un constructor que no tome ningún argumento. Este será el constructor que el contenedor JSP utilice para instanciar el *Bean* a través de la etiqueta *<jsp:useBean>*. En el caso de que la clase no especifique ningún constructor, entonces se asumirá un constructor sin argumentos por defecto.

6.1.3 Definiendo una propiedad de un *Bean.*

Un aspecto importante de los *Beans* son sus propiedades y sobre todo la forma de acceder a ellas; es lo que llamamos como métodos de acceso, tanto en escritura como en lectura. La forma de definir propiedades de los *Beans* consiste en crear métodos *public* con el nombre de la propiedad que deseamos definir con el prefijo *get* o *set* dependiendo si es el acceso en lectura o escritura respectivamente. Los métodos *getter* o de acceso en lectura deberían devolver el tipo apropiado de dato, mientras que el correspondiente método *setter* debería ser declarado *void* y aceptar un argumento del tipo apropiado.

```
public void setPropertyName(ProperType value);

public PropertyType getPropertyName();
```

Es obligatorio declarar *public* a los métodos de acceso a las propiedades.

Vamos a ver con un ejemplo todo esto que estamos diciendo. Para ello crearemos el *Bean HoraActualBean* que nos permitirá mostrar la hora actual en nuestra página.

HoraActualBean.java

```
package com.taglib.wjsp.components;
import java.util.*;

public class HoraActualBean{

private int horas;
private int minutos;

public HoraActualBean(){

Calendar now=Calendar.getInstance();
this.horas=now.get(Calendar.HOUR_OF_DAY);
this.minutos=now.get(Calendar.MINUTE);
}

public int getHoras(){

return horas;

}

public int getMinutos(){
```

return minutos;

}

}

Ya está creado nuestro *Bean*. Hemos creado dos propiedades a las que sólo se puede acceder en lectura, y como hemos seguido las reglas *JavaBean* para llamar a los métodos de acceso, hemos definido dos propiedades que son accesibles a través de etiquetas *Bean* JSP. Por lo tanto es totalmente válido lo siguiente:

<jsp:useBean id="hora" class="HoraActualBean" /> *<html>*
<body>
Son las ***<jsp:getProperty name="hora" property="horas">*** *y* ***<jsp:getProperty name="hora" property="minutos">*** *minutos </body>*
</html>

Normas en el nombrado de las propiedades.

Generalmente los nombres de propiedades suelen comenzar con minúsculas y en mayúscula la primera letra de cada palabra en el nombre de la propiedad. Sin embargo, a la hora de llamar a los nombres de los métodos de acceso, se les antepone el prefijo *set* o *get* seguido por el nombre de la propiedad pero comenzando en mayúsculas.

6.1.4 Propiedades indexadas.

Ya hemos hablado de los problemas que plantean a las etiquetas *Bean* el uso de propiedades indexadas. También vimos cómo tratar este tipo de propiedades, consistente en tratar al *Bean* como la clase Java que es y acceder a sus propiedades como los métodos que son.

Tenemos dos opciones a la hora de definir una propiedad indexada. La primera consiste en crear un método de acceso que devuelva el conjunto entero de propiedades como un *array*. La forma de declararlo sería la siguiente:

public TipoPropiedad[] getPropiedad()

La segunda opción es que el acceso no sea a la lista entera, sino sólo a uno de los elementos del conjunto, el indicado por un índice:

public TipoPropiedad getPropiedad(int indice)

En principio es conveniente declarar ambas formas ya que no representa mucho más trabajo y añade la posibilidad de un trato más flexible de las propiedades. Evidentemente, para el acceso en escritura tendríamos también dos posibilidades:

public void setPropiedad(int indice, TipoPropiedad valor)
public void setPropiedad(TipoPropiedad[])

Un método particularmente útil y que generalmente es implementado y reconocido por los contenedores *Bean* es: *size()* que nos servirá para determinar el tamaño de una propiedad indexada:

public int getPropiedadSize()

Como las etiquetas *Bean* JSP manejan exclusivamente propiedades escalares, será preciso valerse de los elementos *scripting* JSP para tratar con este tipo de propiedades.

6.1.5 Propiedades *booleans*.

Una norma a la hora de declarar propiedades que almacenan valores de "verdadero" o "falso" en sus métodos *getter* es anteponerle la palabra *is*:

public boolean isProperty();

Los métodos de acceso *setter* de una propiedad *boolean*, sin embargo, no se diferencian de los métodos *setter* para otras propiedades.

public void setProperty(boolean b);

6.1.6 Conversión de tipo JSP.

Una propiedad de un componente JSP no está limitada a valores *String*, pero es importante comprender que todos los valores de propiedades a las que se accede mediante una etiqueta *<jsp:getProperty>* serán convertido en un *String*.

Un método *getter* no necesitará devolver un *String* explícitamente, sin embargo, el contenedor JSP convertirá automáticamente el valor de retorno en un *String*.

Para los tipos primitivos Java, la conversión es manejada por los métodos mostrados en la siguiente tabla:

Conversión de tipo para *<jsp:getProperty>*	
Tipo de la propiedad Conversión a *String*	
boolean	*java.lang.Boolean.toString(boolean)*
byte	*java.lang.Byte.toString(byte)*
char	*java.lang.Character.toString(char)*
double	*java.lang.Double.toString(double)*
int	*java.lang.Integer.toString(int)*
float	*java.lang.Float.toString(float)*
long	*java.lang.Long.toString(long)*

Del mismo modo, todos los métodos de acceso *setter* accedidos con una etiqueta *<jsp:setProperty>* será automáticamente convertido de un *String* al tipo nativo apropiado por el contenedor JSP. Esto es llevado a cabo mediante los métodos siguientes:

Conversión de tipo para *<jsp:setProperty>*	
Tipo de la propiedad Conversión desde *String*	
boolean ó *Boolean*	*java.lang.Boolean.valueOf(String)*
byte ó *Byte*	*java.lang.Byte.valueOf(String)*
char ó *Character*	*java.lang.Character.valueOf(String)*

double ó *Double*	*java.lang.Double.valueOf(String)*
int ó *Integer*	*java.lang.Integer.valueOf(String)*
float ó *Float*	*java.lang.Float.valueOf(String)*
long ó *Long*	*java.lang.Long.valueOf(String)*

Las propiedades no están restringidas a tipos primitivos. Si utilizáramos objetos, será preciso, si es que no existe el método Java que lo haga, que declaremos métodos para pasar el objeto a *String* y viceversa.

Manejando propiedades con valores nulos.

Los métodos *getter* para tipos primitivos Java como *int* o *double* no pueden devolver un valor *null*, esto sólo es válido para métodos que devuelven objetos. Sin embargo hay situaciones en las que el valor de una propiedad realmente está indefinido. ¿Qué podemos hacer?. Una opción es crear un método que devuelva un valor *boolean* que diga si el valor que almacena la propiedad es válido. Así, si definimos un método de lectura *getLecturaTemperatura* y otro *isValidaLectuTemper* que devuelva *true* en caso de que el valor almacenado por la propiedad *LecturaTemperatura* sea válido, nos bastará acceder al valor *boolean* para saber si el valor almacenado por la propiedad *LecturaTemperatura* es válido.

6.1.7 Configurando *Beans*.

Muchas veces un *Bean* requerirá ser configurado en tiempo de ejecución por la página que lo está inicializando antes de que realice adecuadamente su tarea. Una forma de hacer esto es aprovechando el cuerpo de la etiqueta <*jsp:useBean*>:

<*jsp:useBean id*="nombre del *Bean*" *class*="nombre de la clase del *Bean*">

código de inicialización

</ jsp:useBean>

Otra posibilidad es acceder en escritura a la propiedad en cualquier punto de la página pero siempre antes de que sea accedida en lectura.

Incluso podríamos definir un constructor de *Bean* que tome argumentos. En este caso no podríamos acceder al *Bean* mediante las etiquetas *Bean*, sino mediante *scriptlets* JSP.

<% Termometro t= new Termometro(78); %>
El termometro marca una temperatura de
< t.getTemperatura() %> *grados.*

Otra técnica bastante útil es proporcionar un método simple que maneje todos los pasos de la configuración. Este método puede ser llamado por el constructor que toma argumentos para usar fuera de las etiquetas *Bean* y también por los métodos de acceso a la propiedad una vez que todas las propiedades necesarias han sido configuradas. Para el ejemplo anterior, vamos a ver como pondremos dos constructores para la clas *Termometro*, además de un método *init()* que manejaría cualquier configuración interna. El constructor sin argumentos nos servirá para poder acceder mediante etiquetas *Bean*, llamando al constructor, que toma una temperatura inicial como argumento por defecto.

TermometroBean.java

public class TermometroBean{

```
private int temp;
private int maxTemp;
private int minTemp;
private int tipoFuel;

public TermometroBean(){

this(75);

}

public int TermometroBean(int temp){

this.temp=temp;
init();

}

public void setTemp(int temp){

this.temp=temp;
init();

}

public int getTemp(){

return temp;

}

private void init(){

maxTemp=this.temp+10;
minTemp=this.temp−15;
if (maxTemp> 150)

tipoFuel=Fuels.DILITHEUM;

else

TipoFuel=Fuels.NATURALGAS;

}

}
```

6.2 Ejemplos. (Ir al índice)

6.2.1 Ejemplo 1. Media de dos números.

Vamos a ver un ejemplo muy simple consistente en una página JSP que utiliza un *Bean* para que le calcule la

media de un par de números que le pasa.

En primer lugar construiré el *Bean* cumpliendo todas las normas.

MediaBean

```
public class MediaBean{

private double numero1;
private double numero2;
private double media;

public double getMedia(){

media=(numero1+numero2)/2;
return media;

}

public void setnumero1(int num1){

numero1=num1;

}

public double getnumero1(){

return numero1;

}

public double getnumero2(){

return numero2;

}

public void setnumero2(int num2){

numero2=num2;

}

}
```

Vemos como se trata de un ejemplo extremadamente sencillo. Consta de tres propiedades, dos de escritura y lectura, *numero1* y *numero2*, y otra de sólo lectura, *media*.

Una página JSP que hace uso de este *Bean* podría ser la siguiente:

Media.jsp

<jsp:useBean id="media" class="MediaBean">

<jsp:setProperty name="media" property="numero1" value="32" />
<jsp:setProperty name="media" property="numero2" value="5" />

</jsp:useBean>
<html>
<body>
*La media de los números:***<jsp:getProperty name="media" property="numero1"/>**
y ***<jsp:getProperty name="media" property="numero2"/>*** *vale:* ***<jsp:getProperty name="media" property="media"/>***
</html>
</body>

6.2.2 Ejemplo 2. Uso de propiedades indexadas. Media de cinco números.

En este caso vamos a ver cómo se van a tratar propiedades indexadas. Vamos a utilizar un *Bean* que toma un *array* de cinco *doubles* para calcularles la media.

EstadBean.java

```
import java.util.*;

public class EstadBean {

private double[] numeros=new double[5];

public void setNumeros(double[] numeros){

this.numeros=numeros;

}

public double getMedia(){

double sum=0;
for (int i=0;i<numeros.length ;i++ )

sum+=numeros[i];

return sum/numeros.length;

}

public double[] getNumeros(){

return numeros;

}

}
```

En cuanto a la página JSP, quedaría como sigue:

Media.JSP

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

<jsp:useBean id="estad" class="EstadBean">

<%double[] num=new double[5];
num={5,2,4,6,8};
estad.setNumeros(num);

</jsp:useBean>
<HTML>
<BODY>
La media de los siguientes números:
<%
double[] numeros=estad.getNumeros();
for(int i=0;i<numeros.length;i++){

if(i!=numeros.length)

out.print(numeros[i] + ",");

else

out.println(""+numeros[i]);

}
%>
<br>VALE:<%= estad.getMedia()%>
</BODY>
</HTML>
```

Vemos que, por el hecho de utilizar propiedades indexadas, vamos a acceder al *Bean* como si fuera una clase, accediendo a las propiedades como métodos de una clase Java normal. A pesar de todo, hemos utilizado la etiqueta
< *jsp:useBean*> y, a efectos de nuestra página JSP, un objeto *estad* de la clase *EstadBean* ha sido creado.

6.2.3 Ejemplo 3. Valores pasados como parámetros de un formulario HTML.

Este ejemplo es muy parecido al anterior, la única diferencia estriba en el hecho de que los valores de los números a los que vamos a hacerle la media son pasados a través de un formulario HTML. En este caso el *Bean* va a seguir siendo el mismo que en el ejemplo anterior. Este es un claro ejemplo de la arquitectura orientada a componentes, el *Bean* es reutilizable por otra página JSP.

EstadBean.java

```
import java.util.*;

public class EstadBean {
```

```
private double[] numeros=new double[5];

public void setNumeros(double[] numeros){

this.numeros=numeros;

}

public double getMedia(){

double sum=0;
for (int i=0;i<numeros.length ;i++ )

sum+=numeros[i];

return sum/numeros.length;

}

public double[] getNumeros(){

return numeros;

}

}
```

El formulario HTML sería extraordinariamente sencillo:

Formulario.html

```
<HTML>
<BODY>
<FORM ACTION="Media2.JSP">
Primer número:<INPUT TYPE=TEXT NAME="num1"><BR>
Segundo número:<INPUT TYPE=TEXT NAME="num2"><BR>
Tercer número:<INPUT TYPE=TEXT NAME="num3"><BR>
Cuarto número:<INPUT TYPE=TEXT NAME="num4"><BR>
Quinto número:<INPUT TYPE=TEXT NAME="num5"><BR><BR>
<INPUT TYPE=SUBMIT VALUE="ENVIAR">
</FORM>
</BODY>
</HTML>
```

En cuanto a la página JSP en sí, va a presentar varios cambios respecto a la página *Media.JSP* del apartado anterior. Estos cambios van a derivar del hecho de cómo le son pasado los números, mediante parámetros en la solicitud. Para obtenerlos nos apoyaremos en el objeto implícito *request*, y más concretamente en su propiedad *getParameter* que nos da el parámetro en cuestión pero en forma de *String*. Por ello tendremos que pasar este *String* a un tipo *double*. Veamos ahora la página *Media2.JSP* que va a utilizar este *Bean*:

Media2.JSP

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

<jsp:useBean id="estad" class="EstadBean">

<%double[] num=new double[5];
num[0]=Double.valueOf(request.getParameter("num1")).doubleValue();
num[1]=Double.valueOf(request.getParameter("num2")).doubleValue();
num[2]=Double.valueOf(request.getParameter("num3")).doubleValue();
num[3]=Double.valueOf(request.getParameter("num4")).doubleValue();
num[4]=Double.valueOf(request.getParameter("num5")).doubleValue();
estad.setNumeros(num);

</jsp:useBean>
<HTML>
<BODY>
La media de los siguientes números:
<%
double[] numeros=estad.getNumeros();
for(int i=0;i<numeros.length;i++){

if(i!=numeros.length)

out.print(numeros[i] + ",");

else

out.println(""+numeros[i]);

}
%>
<br>VALE:<%= estad.getMedia()%>
</BODY>
</HTML>
```