



# Programación con Java Server Pages

Autor: Antonio J. Martín

## ÍNDICE

---

<b>1.</b>	<b>INTRODUCCIÓN.....</b>	<b>5</b>
1.1.	¿QUÉ ES UNA PÁGINA JSP? .....	5
1.2.	VENTAJAS E INCONVENIENTES DE LAS PÁGINAS JSP .....	7
1.3.	CICLO DE VIDA DE UNA PÁGINA JSP .....	8
	EJERCICIO 1.....	9
<b>2.</b>	<b>COMPONENTES DE UNA PÁGINA JSP.....</b>	<b>10</b>
2.1.	BLOQUES DE TEXTO ESTÁTICO.....	11
2.2.	ELEMENTOS DE SCRIPT .....	11
2.2.1.	<i>scriptlets.</i> .....	11
2.2.2.	<i>expresiones.</i> .....	11
2.2.3.	<i>declaraciones.</i> .....	11
2.3.	OBJETOS IMPLÍCITOS JSP .....	12
2.3.1.	<i>Objetos request, response, session y application</i> .....	13
	EJERCICIO 2.....	14
2.3.2.	<i>Objeto config</i> .....	15
2.3.3.	<i>Objeto PageContext</i> .....	15
2.3.4.	<i>Objeto page</i> .....	16
2.3.5.	<i>Objeto exception</i> .....	16
2.4.	DIRECTIVAS.....	17
2.4.1.	<i>Directiva page</i> .....	17
	EJERCICIO 3.....	19
	EJERCICIO 4.....	19
2.4.2.	<i>Directiva include.</i> .....	20
2.4.3.	<i>Directiva taglib</i> .....	21
2.5.	ACCIONES JSP .....	21
2.5.1.	<i>Sintaxis de una acción JSP</i> .....	22
2.5.2.	<i>Acciones estándar JSP</i> .....	22
2.5.2.1.	<i>Acción forward</i> .....	22

2.5.2.2.	Acción include .....	23
2.5.2.3.	Acción param .....	23
2.5.2.4.	Acción useBean .....	24
2.5.2.5.	Acción setProperty .....	25
2.5.2.6.	Acción getProperty .....	28
	EJERCICIO 5.....	29
<b>3.</b>	<b>EL LENGUAJE EL.....</b>	<b>36</b>
3.1.	EXPRESIONES EL .....	37
3.2.	ACCESO A OBJETOS MEDIANTE EXPRESIONES EL .....	37
3.3.	OBJETOS IMPLÍCITOS EL .....	38
3.4.	OPERADORES EL.....	40
<b>4.</b>	<b>LA LIBRERÍA DE ACCIONES JSTL.....</b>	<b>40</b>
4.1.	INSTALACIÓN DE JSTL .....	41
4.2.	UTILIZACIÓN DE JSTL EN UNA PÁGINA JSP .....	42
4.3.	ESTUDIO DE LA PRINCIPALES ACCIONES DEL CORE DE JSTL.....	42
4.3.1.	<i>Acciones de propósito general.....</i>	<i>43</i>
4.3.1.1.	out.....	43
4.3.1.2.	set.....	43
4.3.1.3.	remove .....	44
4.3.1.4.	redirect .....	44
4.3.2.	<i>Acciones de control de flujo .....</i>	<i>44</i>
4.3.2.1.	if.....	44
4.3.2.2.	choose .....	44
4.3.2.3.	foreach .....	45
4.3.2.4.	fortokens.....	46
	EJERCICIO 6.....	46
<b>5.</b>	<b>CREACIÓN DE ACCIONES JSP PERSONALIZADAS.....</b>	<b>49</b>
5.1.	IMPLEMENTACIÓN DE LA CLASE MANEJADORA .....	50
5.1.1.	<i>Ciclo de vida básico de una acción.....</i>	<i>50</i>
5.1.2.	<i>Escritura en la página de respuesta.....</i>	<i>51</i>

5.2.	CREACIÓN DE UN ARCHIVO DE LIBRERÍA .....	51
5.2.1.	<i>Etiquetas para la definición de una librería de acciones.....</i>	<i>51</i>
5.3.	UTILIZACIÓN DE ACCIONES PERSONALIZADAS EN UNA PÁGINA JSP .....	53
5.4.	ATRIBUTOS EN ACCIONES JSP.....	54
5.5.	ITERACIÓN SOBRE EL CUERPO DE UNA ACCIÓN .....	56
5.6.	MANIPULACIÓN DEL CUERPO DE LA ACCIÓN .....	58
	EJERCICIO 7.....	62

# 1. INTRODUCCIÓN

---

Al igual que los servlets las Java Server Pages, o páginas JSP como se les llama habitualmente, son un tipo de componente que forma parte de la capa intermedia de una aplicación Web J2EE, concretamente, dentro de la subcapa de presentación.

Aunque se trate de un componente que, aparentemente, realiza las mismas funciones que un servlet, esto es, capturar datos del cliente y generar dinámicamente respuestas, la tecnología JSP no representa ningún elemento redundante en la programación del servidor ni tampoco tiene como objetivo reemplazar a los servlets. Más al contrario, y como veremos más adelante, ambas tecnologías se complementan perfectamente además de resultar imprescindible la utilización de ambas para el desarrollo de una aplicación Web estructurada.

## 1.1. ¿QUÉ ES UNA PÁGINA JSP?

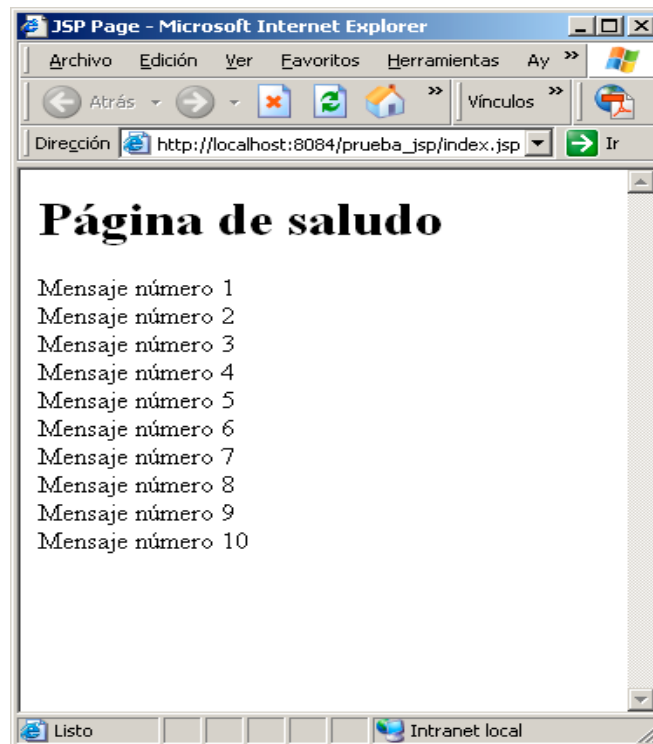
Una página JSP es un archivo de texto con extensión .jsp en el que se combinan bloques de texto HTML (o cualquier otro lenguaje de marcado) con código Java de servidor. Estos archivos forman parte de la aplicación Web J2EE, situándose en el directorio raíz de la misma o en algún subdirectorio de éste.

El siguiente listado corresponde a una sencilla página JSP de ejemplo:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Página de saludo</h1>
    <%for(int i=1;i<=10;i++){%>
      Mensaje número <%=i%>
      <br/>
    <%}%>
  </body>
</html>
```

Dejando al margen las dos primeras líneas, que en su momento veremos que significan, podemos observar que la mayor parte de la página está formada por texto HTML que constituye la página de respuesta que será enviada al cliente cuando la página sea solicitada.

Por otro lado, distinguimos en el interior de ésta los bloques de código Java encerrados entre los símbolos <% y %>. Este código tiene como misión la generación dinámica de parte de la página de respuesta. El ejemplo presentado generaría una página de respuesta con el aspecto que se indica en la figura 1.



**Figura. 1.**

Como vemos, el encabezado “Página de saludo” se crea directamente mediante el elemento HTML `<h1>` incluido en la página, mientras que los 10 mensajes que aparecen a continuación son generados dinámicamente por el bucle `for`. Esta flexibilidad que ofrecen las página JSP, al permitir intercalar bloques de texto entre instrucciones Java, proporciona al programador un gran poder a la hora de generar dinámicamente una página de respuesta.

Si desde el menú del navegador cliente accedemos al código fuente de la página, veríamos únicamente HTML puro:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Página de saludo</h1>

    Mensaje número 1
    <br/>

    Mensaje número 2
    <br/>

    Mensaje número 3
    <br/>

    Mensaje número 4
    <br/>

    Mensaje número 5
    <br/>

    Mensaje número 6
    <br/>

    Mensaje número 7
```

```

<br/>

Mensaje número 8
<br/>

Mensaje número 9
<br/>

Mensaje número 10
<br/>

</body>
</html>

```

Esto significa que el código Java es ejecutado en el servidor en el momento en que la página es solicitada por el cliente, enviándose como respuesta al cliente la página resultante de combinar el bloque HTML fijo que aparece en la página JSP con la parte generada de forma dinámica (figura 2).

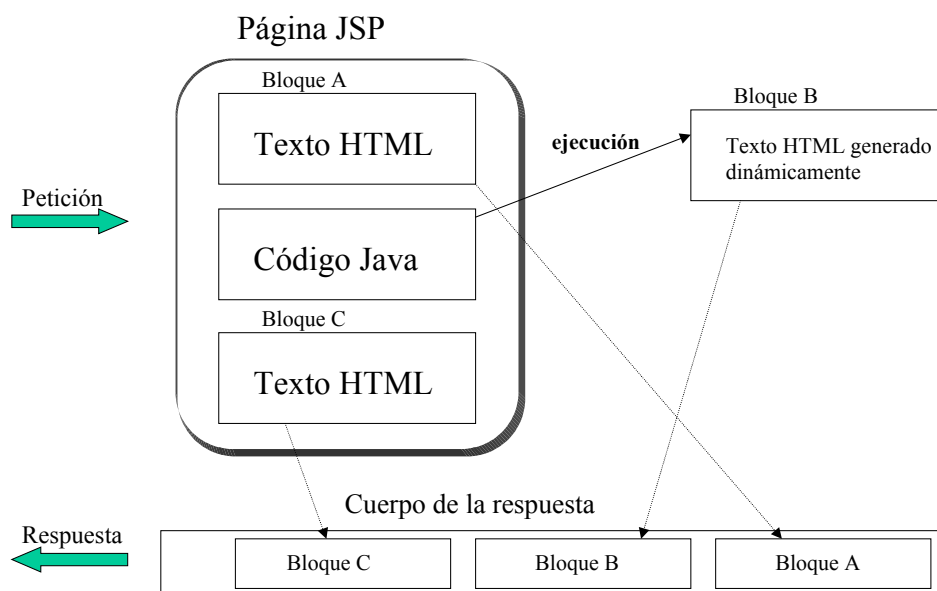


Figura. 2.

## 1.2. VENTAJAS E INCONVENIENTES DE LAS PÁGINAS JSP

Respecto a la tecnología servlet, las páginas JSP presentan una serie de ventajas a la hora de implementar la generación de respuestas cliente:

- **Facilidad de implementación.** Como vimos en el tema anterior, en un servlet las instrucciones HTML para dar el formato a la página se debían incluir dentro del propio código de la clase en el interior de los incómodos `out.println()`, lo que, además de hacer tediosa la generación de la respuesta, resulta propenso a errores. Por el contrario, la página JSP permite introducir los bloques de texto HTML tal cual, resultando más sencillo el desarrollo de la página.
- **Separación del código de servidor de las instrucciones de presentación.** La separación entre estos dos elementos constitutivos de la página JSP permite la división de roles a la hora de acometer el desarrollo de la misma;

por un lado, el código de servidor puede ser implementado por un programador, mientras los bloques de texto HTML pueden ser generados por un diseñador Web utilizando alguna de las numerosas herramientas existentes para ello. Lógicamente, será necesario realizar un proceso de integración final.

- **Posibilidad de realizar modificaciones de una forma rápida y sencilla.** Dado que se trata de un archivo de texto, las modificaciones en las páginas JSP se pueden realizar mediante cualquier editor de texto, sin necesidad de realizar ningún proceso de compilación posterior.

En cuanto a los inconvenientes, todos están relacionados con la inserción de código Java dentro de la página. Al no tratarse de una clase, sino de un archivo de texto, la creación de instrucciones Java en su interior supone:

- Mayor incomodidad que en un servlet a la hora de programar.
- Código más difícil de estructurar.
- Mayor propensión a errores en el código.

Todo esto viene a demostrar lo que comentábamos anteriormente sobre el hecho de que las páginas JSP no sustituyen a los servlets en la construcción de una aplicación Web, sino que cada uno se puede especializar en una determinada tarea; mientras las páginas JSP resultan adecuadas para la generación de las respuestas, los servlets pueden dedicarse más a tareas de control y gestión de peticiones que requieren una mayor labor de programación.

### 1.3. CICLO DE VIDA DE UNA PÁGINA JSP

Aunque al la hora de implementarlas y desplegarlas en una aplicación Web se trate de archivos de texto, toda página JSP es convertida en un servlet antes de ser ejecutada por el contenedor Web. De hecho, un contenedor Web que ejecute servlets servirá también para ejecutar páginas JSP, pues al fin y al cabo se trata en ambos casos de servlets.

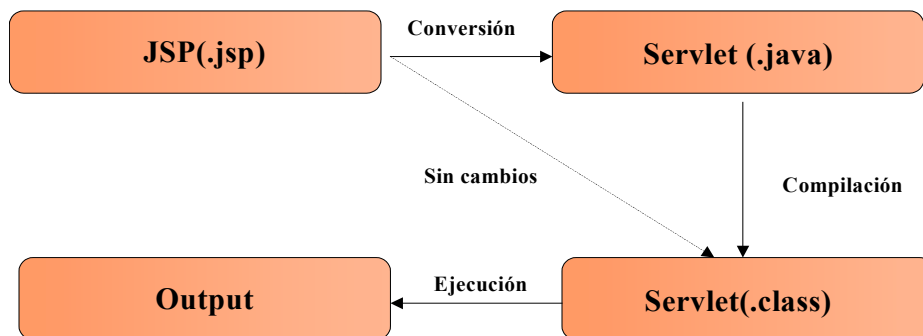
El proceso de conversión de archivo de texto en servlet se puede producir en cualquier momento comprendido entre el despliegue de la aplicación y la primera vez que vaya a ser ejecutada la página. Normalmente, los contenedores Web suelen realizar esta operación cuando la página JSP es solicitada por primera vez.

Durante el proceso de conversión, se genera una subclase de `HttpServlet` en cuyo método de servicio se incluye todo el contenido de la página, tanto el código Java que será incluido directamente en el método, como los bloques de texto HTML que serán insertados como cadenas de texto en el interior de métodos `out.println()`.

Esta subclase de `HttpServlet` es específica de cada contenedor y debe implementar la interfaz `HttpJspPage`. Esta interfaz proporciona los métodos `jspInit()`, `_jspService()` y `jspDestroy()` que son invocados directamente desde los métodos `init()`, `service()` y `destroy()` de la clase. Es, por tanto, en el método de servicio `_jspService()` donde el contenedor incluye el código resultante de la transformación de la página en servlet.

Tras la generación del código fuente del servlet, el contenedor realiza la compilación del mismo y su registro en el archivo de configuración `web.xml`, liberando al programador de esta tarea. A partir de ahí el comportamiento será idéntico al que estudiamos en el tema anterior: con la primera petición cliente se creará una instancia del servlet que será utilizada para atender esta y las posteriores solicitudes que lleguen (figura 3).





**Figura. 3.**

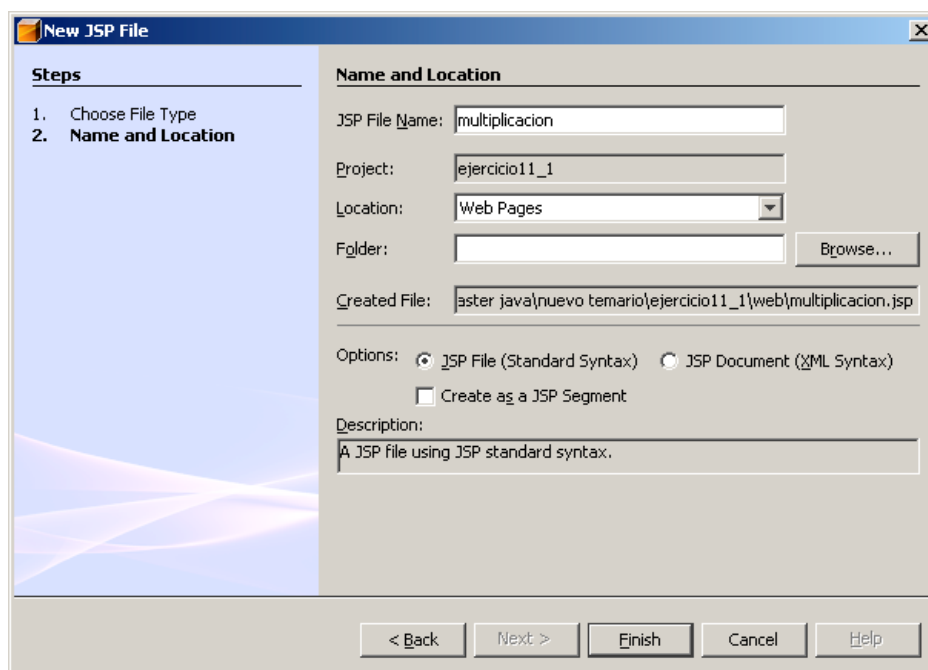
Si después de haber sido creado el servlet a partir de la página JSP el programador lleva a cabo alguna modificación en la misma, ya sea en el código Java o en el texto HTML, el contenedor detectará dicho cambio, volviendo a generar de nuevo el servlet en el momento en que se realice una nueva solicitud de la página.

## EJERCICIO 1

Como primer ejercicio vamos a implementar una página JSP que realice la misma operación del servlet creado en el ejercicio 1 del tema anterior, esto es, la generación de la tabla de multiplicar de los números del 1 al 10.

Lo primero será crear una aplicación Web Java. Si disponemos del entorno de desarrollo Netbeans seguiremos los pasos descritos en el tema anterior.

Con la creación de una aplicación Web con Netbeans se genera automáticamente una página JSP que podemos reutilizar en nuestro proyecto, aunque en este caso la eliminaremos y añadiremos una nueva. Para ello, nos situaremos con el ratón sobre el icono de proyecto y pulsando el botón derecho accederemos al menú contextual, donde elegiremos la opción New->JSP, o New->File/Folder si no se dispone de la opción anterior. Una vez elegido el tipo de archivo JSP, nos aparecerá un cuadro de diálogo como el indicado en la figura 4 donde se nos pedirá el nombre de la página.



**Figura. 4.**

Hemos de tener precaución de introducir el nombre sin la extensión .jsp, ya que esta es añadida automáticamente por el asistente. Para el resto de opciones se dejarán los valores por defecto. En el caso de la opción "Folder", nos pide que indiquemos en subdirectorio, dentro del raíz de la aplicación, donde se incluirá la página, por lo que al dejarlo en blanco se situará en el propio directorio de la aplicación.

Como vemos, el aspecto de la página creada es muy similar al de la página por defecto incluida al crear la aplicación Web. A fin de que resulte más comprensible, eliminaremos todas las líneas que parecen comentadas en la página JSP.

El aspecto final de la misma, una vez incluidas las instrucciones para la generación de la tabla, será el indicado en el siguiente listado:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <title>Ejercicio 1</title>
  </head>
  <body>
    <center>
      <h1>Tabla de multiplicar</h1>
      <table border="1">
        <%for(int i=1;i<=10;i++){%>
          <tr>
            <%for(int k=1;k<=10;k++){%>
              <td><%=i*k%></td>
            <%}%>
          </tr>
        <%}%>
      </table>
    </center>
  </body>
</html>
```

## 2. COMPONENTES DE UNA PÁGINA JSP

---

A pesar de que una página JSP se compone fundamentalmente de texto de marcado y código Java, hay otros elementos de gran importancia que pueden ser utilizados en la misma.

Podemos decir que una página JSP puede incluir los siguientes componentes:

- Bloques de texto estático
- Elementos de script
- Objetos implícitos
- Directivas
- Acciones JSP

Seguidamente, analizaremos en detalle todos estos componentes.

## 2.1. BLOQUES DE TEXTO ESTÁTICO.

Se trata de todo el texto contenido dentro de la página JSP que es enviado “tal cual” al cliente. Cuando el cliente es un navegador, estos bloques de texto están formados por los datos a visualizar y las etiquetas de formato HTML, pudiendo incluir también hojas de estilo y scripts de cliente.

Sin embargo, la especificación JSP no se limita a HTML, en general se puede incluir en una página JSP cualquier formato de texto, según el tipo de cliente al que se enviará la respuesta. De hecho, suele ser habitual en algunas aplicaciones que las páginas JSP generen texto XML, utilizándose después algún mecanismo, tipo XSLT, para transformar el XML en algún formato de presentación según el tipo de cliente.

## 2.2. ELEMENTOS DE SCRIPT

Los elementos de script representan el código Java de servidor incluido en la página.

Estos scripts se pueden dividir a su vez en tres tipos:

- scriptlets
- expresiones
- declaraciones

### 2.2.1. *scriptlets.*

Un scriptlet es cualquier fragmento de código Java encerrado entre los símbolos `<%` y `%>` y que será incluido directamente en el método `service()` del servlet generado. Los scriptlets pueden realizar operaciones que afecten al contenido generado, como es el caso de los scriptlets `<%for(int i=1;i<=10;i++){%>` y `<%}%>` utilizados en el ejemplo anterior y cuya misión consistía en generar 10 veces un mensaje de saludo.

### 2.2.2. *expresiones.*

Una expresión incluye en la página de respuesta el resultado de ejecutar una pieza de código Java. Las expresiones se encierran entre los símbolos `<%=` y `%>` y pueden contener cualquier expresión de código Java válido que devuelva un resultado. Este resultado será convertido a String e incluido en la respuesta generada.

Por ejemplo, la expresión `<%=i%>`, donde `i` es una variable declarada anteriormente en un scriptlet y que contiene algún valor, es convertida en la instrucción `out.println(i);`, dentro del servlet generado.

Es importante destacar el hecho de que **la instrucción Java que forma la expresión no debe finalizar con `;`**, tal y como sucede con el código Java habitual.

### 2.2.3. *declaraciones.*

Como hemos dicho, todo el código contenido en un scriptlet y en una declaración es incluido en el interior del método `_jspService()` del servlet durante la fase de transformación. Si queremos que una determinada variable sea declarada a nivel de clase, o queremos que un

determinado bloque de sentencias esté definida en un método aparte distinto de `_jspService()`, habrá que utilizar una declaración.

Una declaración está delimitada por los símbolos `<%! y %>`, de modo que todo el código que se encuentre dentro de un bloque de este tipo estará situado fuera del método `_jspService()` en el servlet generado. Por ejemplo, lo siguiente definiría una variable de clase "cont" de tipo entero:

```
<%! int cont;%>
```

Mientras que el siguiente bloque de sentencias corresponden a la definición de un método:

```
<%! int calculaSuma(int a, int b){  
    return a+b;  
}%>
```

Podemos también utilizar las declaraciones para sobrescribir los métodos `jspInit()` y `jspDestroy()` que aparecerán en la subclase de `HttpServlet` generada a partir de la página. El siguiente código de ejemplo declara una variable de clase que es inicializada en el método `jspInit()` al valor 25 para, posteriormente, desde un scriptlet mostrar su valor e incrementar a continuación la variable:

```
<%!int dato;%>  
<%!public void jspInit(){  
    dato=25;  
}%>  
El dato vale: <%=dato%>  
<%dato++;%>
```

La primera vez que se solicite la página mostrará el mensaje El dato vale: 25 pero como la inicialización solo se produce una vez, posteriores peticiones harán que se muestre el dato incrementado en una unidad respecto a la petición anterior.

## 2.3. OBJETOS IMPLÍCITOS JSP

JSP define una serie de objetos implícitos (no hay que instanciarlos) que pueden ser utilizados directamente en cualquier script de la página JSP para invocar a sus métodos.

Dado que una página JSP es convertida en un servlet, algunos de estos objetos son los mismos con los que nos encontramos en la programación con servlets y que nos permiten resolver todos los problemas analizados en el tema anterior, como el acceso a los parámetros enviados en la petición cliente, atributos de petición, sesión y aplicación, redireccionamiento, etc.

En la tabla de la figura 5 se indica, para cada uno de estos objetos implícitos, la clase o interfaz del API servlet equivalente o la funcionalidad que implementa. También se muestra un ejemplo de utilización de uno de estos objetos.

Objeto	Descripción
request	Equivale HttpServletRequest
response	Equivale a HttpServletResponse
session	Equivale a HttpSession
application	Equivale a SevletContext
config	Equivale a SevletConfig
pagecontext	Proporciona acceso a los atributos de página
page	Representa la instancia del servlet generada a partir de la página JSP
out	Permite enviar datos a la salida
exception	Proporciona acceso a la excepción generada

jsp1.jsp

```
<% session.setAttribute("nombre","James");%>
```

jsp2.jsp

```
El nombre es:<%=session.getAttribute("nombre")%>
```

Figura. 5.

### 2.3.1. Objetos request, response, session y application

Tal y como se indica en la tabla anterior, estos objetos implementan interfaces del API servlet que ya hemos estudiado en el tema anterior, con lo cual, el acceso a parámetros de petición y a los atributos de distintos ámbitos de la aplicación puede realizarse invocando directamente a los del objeto correspondiente, tal y como queda reflejado en el ejemplo de la figura anterior.

Así pues, todo lo que hemos estudiado en el tema anterior sobre estas interfaces es igualmente aplicable en las páginas JSP, solo que de una forma más sencilla.

## EJERCICIO 2

En este ejercicio vamos a poner en práctica el uso de uno de los objetos implícitos JSP, concretamente el objeto request. Se trata de crear una aplicación que a través de un formulario HTML solicite la introducción de un número al usuario. Tras pulsar el botón enviar, se enviará al usuario una página con la tabla de multiplicar del número (figura 6).

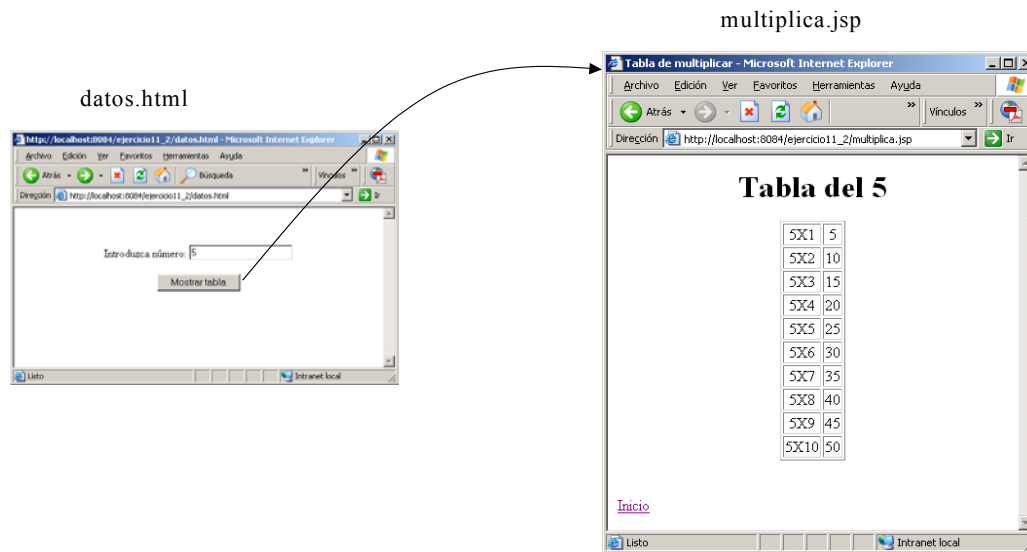


Figura. 6.

La aplicación estará formada por una página HTML para la toma de datos y una JSP encargada de generar la respuesta. He aquí el código fuente de ambas páginas:

### datos.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
    <center>
      <form action="multiplica.jsp" method="POST">
        <br/><br/>
        Introduzca número: <input type="text"
          name="txtnumero"/><br/><br/>
        <input type="submit" value="Mostrar tabla"/>
      </form>
    </center>
  </body>
</html>
```

### multiplica.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Tabla de multiplicar</title>
    </head>
    <body>
        <center>
            <%int numero=Integer.parseInt(
                request.getParameter("txtnumero"));%>
            <h1>Tabla del <%=numero%></h1>
            <table border="1">
                <%for(int i=1;i<=10;i++){%>
                    <tr>
                        <td><%=numero%>X<%=i%></td>
                        <td><%=numero*i%></td>
                    </tr>
                <%}%>
            </table>
        </center>
        <br/><br/>
        <a href="datos.html">Inicio</a>
    </body>
</html>
```

#### 2.3.2. Objeto config

Se trata de un objeto poco utilizado en JSP, su uso está limitado prácticamente a la obtención de parámetros de inicialización del servlet, definidos en el elemento `<init-param>` asociado al elemento `<servlet>` dentro del archivo `web.xml`. A través del método `getInitParameter()` definido en la interfaz `ServletConfig` podemos acceder al valor de estos parámetros a partir de sus nombres.

#### 2.3.3. Objeto PageContext

Se trata de una instancia de la clase `PageContext` definida en el paquete `javax.servlet.jsp.PageContext`. Además de los métodos `getRequest()`, `getResponse()`, `getSession()` y `getServletContext()` que nos proporcionan una referencia a los objetos implícitos anteriores, esta clase dispone de una implementación de los métodos `setAttribute()` y `getAttribute()` con los que podemos acceder a los atributos de cualquier ámbito:

- `getAttribute(String nombre, int ambito)`
- `setAttribute(String nombre, Object valor, int ambito)`

El parámetro entero "ambito") representa el ámbito del atributo al que queremos acceder. Cada uno de los cuatro ámbitos posibles (pagina, petición, sesión y aplicación) tiene definida una constante en la propia clase `PageContext` que lo identifica:

- `PageContext.PAGE_SCOPE`

- `PageContext.REQUEST_SCOPE`
- `PageContext.SESSION_SCOPE`
- `PageContext.APPLICATION_SCOPE`

La siguiente instrucción, situada en una determinada página JSP, almacenaría una cadena de texto en el atributo de sesión “usuario”:

```
pageContext.setAttribute("usuario", "admin",
                        pageContext.SESSION_SCOPE);
```

Por su parte, para obtener este dato desde cualquier otra página se podría utilizar el siguiente código:

```
String user;
user=(String)pageContext.getAttribute("usuario",
                                     PageContext.SESSION_SCOPE);
```

#### **2.3.4. Objeto page**

Este objeto representa la instancia del servlet generado, algo así como el equivalente a `this`. Lo único es que `page` es de tipo `Object`, por lo que si se quiere utilizar este objeto para invocar a algún método de `HttpServlet` habrá que realizar una conversión explícita. El siguiente ejemplo mostraría el nombre del servlet en la página de respuesta:

```
<% HttpServlet servlet = (HttpServlet)page;%>
Servlet: <%=servlet.getServletName() %>
out
```

Este objeto representa un objeto `PrintWriter` asociado a la salida. Mediante su método `println()` se puede enviar texto a la página de respuesta.

Debido a la existencia de las expresiones JSP, su uso es bastante escaso. Por ejemplo, la instrucción:

```
<%out.println("la variable vale: "+i);%>
es equivalente a
la variable vale: <%=i>
```

#### **2.3.5. Objeto exception**

El objeto `exception` representa un objeto de alguna subclase de `Exception` asociado a la última excepción producida.

Este objeto no puede utilizarse en cualquier tipo de página JSP, para ello será necesario indicar que se trata de una página de error a través de la directiva `page`, tal y como estudiaremos en el siguiente capítulo.



## 2.4. DIRECTIVAS

Las directivas son **interpretadas por el compilador durante la fase de transformación de la página JSP en un servlet**. A través de ellas podemos indicar determinadas operaciones que deban ser realizadas antes de la compilación del código, como por ejemplo la importación de alguna clase o la inclusión del contenido de algún archivo externo.

La sintaxis utilizada para la inserción de una directiva en la página es:

```
<%@ directiva atributo1 = "valor" atributo2 = "valor" ..%>
```

donde *directiva* representa el nombre de la directiva utilizada y *atributo1*, *atributo2*, etc. establecen los valores para los distintos atributos soportados por la directiva.

Existen tres posibles directivas que podemos utilizar en una página JSP:

- page
- include
- taglib

Analizaremos a continuación cada una de ellas, así como los principales atributos que soportan.

### 2.4.1. Directiva page

A través de sus atributos permite indicar distintas operaciones de tipo general a realizar en la página. De ser utilizada, la directiva page debe aparecer al principio de la página JSP como primer elemento de la misma.

Entre sus principales atributos están:

- **language**. Indica al compilador el tipo de lenguaje de programación utilizado por los elementos de script, siendo Java el valor predeterminado y único que hasta la fecha soportan los contenedores Web:

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

- **import**. Indica la clase o paquete de clases que deberán ser importados para la compilación del servlet generado a partir de la página. Así pues, la directiva:

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

hará que el contenedor incluya la instrucción:

```
import java.util.*;
```

en el archivo de código fuente del servlet resultante de la transformación.

Si queremos importar más de una clase/paquete de clases, puede utilizarse una directiva page por cada clase/paquete a importar o podemos indicar en una misma directiva cada uno de los elementos importados, separados por una coma ",". Por ejemplo, si queremos importar los paquetes java.util y java.io completos podemos utilizar:

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

○

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

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

El paquete `java.lang` es importado siempre por el contenedor Web sin necesidad de indicarlo a través de la directiva `page`.

- **contentType**. Permite definir el tipo MIME del contenido generado por la página JSP:

```
<%@ page contentType = "text/html" %>
```

La anterior directiva es equivalente a incluir la siguiente instrucción en un scriptlet de la página:

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

Hay que decir que tipo de contenido predeterminado es precisamente HTML, por ello, en caso de que no se especifique ningún valor para el atributo `contentType` de la directiva `page` ni se haga la llamada a `setContentType()` del objeto `response`, el contenedor Web incluirá el valor "text/html" en el encabezado de respuesta correspondiente.

- **session**. Indica si el objeto implícito `session` estará o no disponible en la página JSP. Si la página JSP no va a participar en la sesión, es conveniente indicar el valor `false` en este atributo a fin de mejorar el rendimiento de la aplicación. El valor predeterminado de este atributo es `true`.
- **isELIgnored**. Como ya veremos más adelante, el lenguaje EL permitirá realizar determinadas operaciones, que habitualmente se llevan a cabo con un elemento de script, sin utilizar código Java.

Este lenguaje se basa en la utilización de unas expresiones englobadas entre los símbolos "\${" y "}". Si no vamos a utilizar EL en la página JSP, será necesario indicarle al contenedor Web que no interprete estos símbolos como delimitadores de una expresión, para lo cual tendremos que asignar el valor `true` al atributo `isELIgnored`.

- **errorPage**. Contiene la URL relativa de la página de error (JSP o HTML) a la que será redireccionado el usuario si se produce una excepción no controlada en la página.
- **isErrorPage**. Cuando la página de error es JSP y queremos hacer uso del objeto `exception` en cualquiera de los elementos de script de la misma, debemos incluir en dicha página de error la directiva `page` con el atributo `isErrorPage` al valor "true".

### EJERCICIO 3

Vamos a realizar una nueva versión de la aplicación del ejercicio 2, en donde se incluya una página de error a la que será redireccionado el usuario si se produce alguna excepción al calcular la tabla de multiplicar del número (por ejemplo, en vez de suministrar un número se introduce un texto). Esta página simplemente mostrará el mensaje de error asociado a la excepción.

Para realizar esta tarea, la página `multiplica.jsp` tendrá que incluir el atributo `errorPage` en la directiva `page`, indicando en él la URL relativa de la página de error:

```
<%@page errorPage="error.jsp" %>
```

Por su parte, la página de error estará constituida por el siguiente código:

```
<%@ page contentType="text/html" pageEncoding="UTF-8"
      isErrorPage="true"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <title>Página de error</title>
  </head>
  <body>
    <h1>Error en el cálculo!</h1>
    Mensaje de error: <b><%=exception.toString()%></b>
  </body>
</html>
```

### EJERCICIO 4

Para ilustrar como pueden trabajar conjuntamente las páginas JSP y los servlets dentro de una misma aplicación, vamos a crear una nueva versión del ejercicio de registro de nombres (ejercicio 6) desarrollado en el tema anterior.

La funcionalidad será la misma que se presentó entonces, solo que ciertos componentes serán implementados como servlets y otros como páginas JSP. Concretamente, tenemos dos componentes de lado de servidor en esta aplicación: el encargado de registrar los nombres y el que genera una página HTML con todos los nombres registrados. En la versión anterior, ambos eran implementados como servlet pero en esta el registro de los nombres seguirá realizándose mediante un servlet, mientras que la generación de la página con la lista de éstos se llevará a cabo con una página JSP a la que llamaremos “`muestra.jsp`”.

El registro de los nombres se realizará mediante un servlet puesto que esta tarea no requiere ninguna operación de generación de código HTML, tan solo se debe implementar código Java de servidor. Por el contrario, la generación de la lista de nombres requiere un gran contenido de instrucciones para la creación dinámica de HTML por lo que resulta más conveniente implementarla mediante una página JSP.

Así pues, lo único que cambiará en esta nueva versión será la desaparición del servlet “`muestra`” y su sustitución por la página “`muestra.jsp`”:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="java.util.*,javabeans.Persona"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Muestra Nombres</title>
  </head>
  <body>
    <center>
      <h1>Listado de nombres registrados</h1>
      <%if(application.getAttribute("lista")!=null){
        ArrayList<Persona> v=(ArrayList<Persona>)
          application.getAttribute("lista");%>
      <table border="1">
        <tr>
          <th>Nombre</th>
          <th>Teléfono</th>
        </tr>
        <%for(Persona p:v){%>
          <tr>
            <td><%=p.getNombre()%></td>
            <td><%=p.getTelefono()%></td>
          </tr>
        <%}%>
      </table>
      <%}else{%>
        <h2>No hay usuarios registrados</h2>
      <%}%>
    </center>
  </body>
</html>

```

Así mismo, se deberá realizar una modificación en la página HTML “opciones.html” para que el enlace “Ver registrados” apunte ahora a la página anterior:

```

<a href="muestra.jsp">Ver Registrados</a><br/>

```

#### 2.4.2. Directiva include.

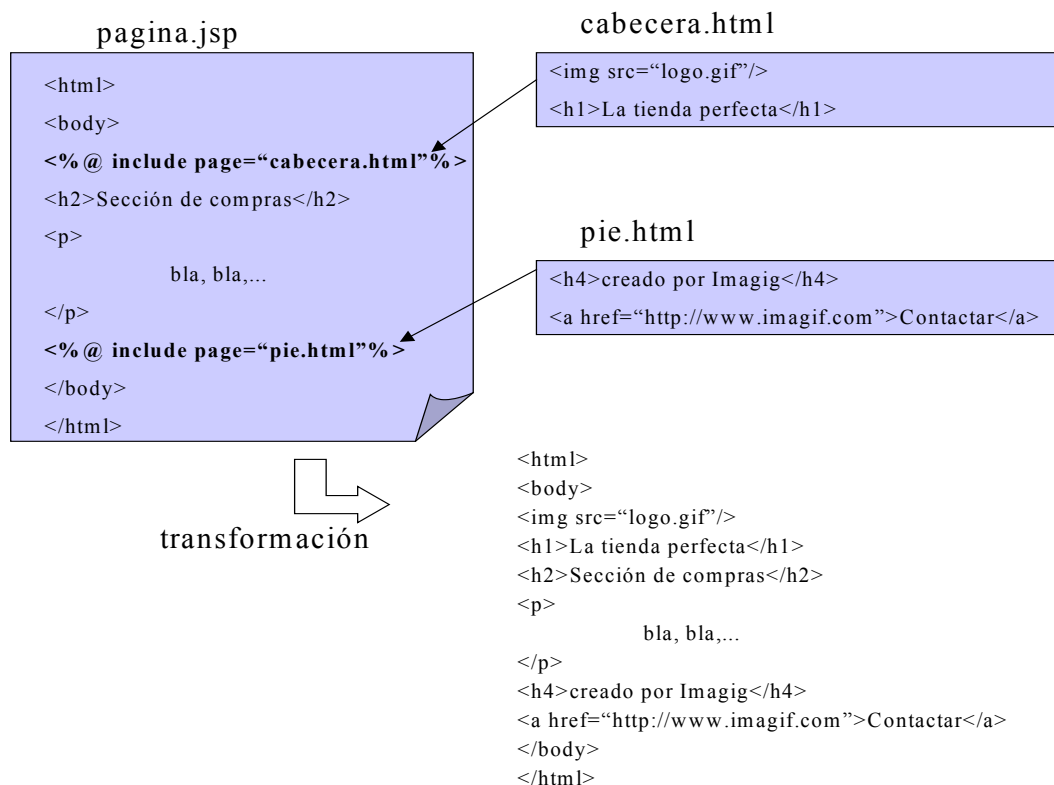
Se utiliza para incluir en una página el contenido de un archivo de texto externo, habitualmente HTML o JSP.

A través de su único atributo file se especifica la URL relativa del archivo a incluir.

A diferencia de page que debe aparecer siempre al principio de la página, la directiva include puede colocarse en cualquier parte, más exactamente, donde queramos incluir el bloque de texto externo (figura 7).

Dado que la inserción del contenido externo en la página se realiza durante la fase de transformación a servlet, **cualquier modificación posterior que se produzca en el archivo incluido no se verá reflejada durante la ejecución de la página JSP.**

La utilización de la directiva include permite reutilizar determinados bloques de código HTML/JSP que deben aparecer en diferentes páginas de la aplicación, como encabezados y pies de página (figura 7).



**Figura. 7.**

### 2.4.3. Directiva taglib

El empleo de esta directiva en una página JSP está relacionado con la utilización en la misma de acciones JSP externas, por lo que será durante el estudio de esta característica cuando estudiemos esta directiva.

De momento, comentar únicamente que taglib se utiliza para indicar al contenedor la ubicación del archivo de librería donde se definen las acciones que se utilizarán en la página.

## 2.5. ACCIONES JSP

Las operaciones definidas por las **acciones JSP son llevadas a cabo en tiempo de ejecución**. Toda acción JSP lleva asociado un código de script de servidor que, al igual que el que se inserta directamente en la página, es integrado por el contenedor Web en el método *service()* del servlet creado en la transformación.

Podemos decir, por tanto, que los objetivos de las acciones JSP son fundamentalmente tres:

- Reutilizar de determinados bloques de código de uso común en las aplicaciones en diferentes páginas JSP.
- Mayor legibilidad de las páginas. Al utilizar una sintaxis basada en XML, la utilización de acciones JSP evita la inserción de scriptlets dentro de la página, haciéndolas más legibles y menos propensas a errores.
- Facilidad en el desarrollo y mantenimiento de la página. El hecho de no necesitar un conocimiento del lenguaje Java para poder utilizar una acción, permite a perfiles no programadores manejar fácilmente estos componentes.

### 2.5.1. Sintaxis de una acción JSP

Como hemos dicho, las acciones JSP utilizan una sintaxis XML. He aquí el formato de una acción:

```
<nombre_accion atributo1 = "valor" atributo2 = "valor" ../>
o
<nombre_accion atributo1 = "valor" ...>
    <!--otras acciones, scriptlet o texto HTML-->
</nombre_accion>
```

nombre\_accion es el nombre de la acción que, en el caso de las acciones estándar de JSP incluye el prefijo jsp.

Las acciones pueden incluir atributos, los cuales representan información adicional suministrada al código asociado a la acción para que ésta pueda llevar a cabo su tarea.

### 2.5.2. Acciones estándar JSP

Se les llama así porque estas acciones están incluidas en la propia especificación JSP.

Las principales acciones estándar JSP son:

- forward
- include
- param
- useBean
- getProperty
- setProperty

Veamos a continuación el significado de cada una de ellas así como de sus atributos más significativos.

#### 2.5.2.1. Acción forward

La acción forward permite derivar la petición a otra página JSP o servlet de la misma aplicación. Su funcionamiento es el mismo que el del método *forward()* del objeto *RequestDispatcher*.

Mediante su único atributo *page* se indica la dirección del componente invocado:

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

Como la acción es evaluada en tiempo de ejecución, la URL indicada en el atributo *page* puede corresponder a un valor fijo, como en el caso anterior, o puede ser obtenido

dinámicamente a través de una expresión JSP. En el siguiente ejemplo la URL del componente destino se obtiene del parámetro “dir” enviado en la petición:

```
<jsp:forward page = "<%=request.getParameter("dir")%"/>" />
```

Cuando se ejecuta la acción *forward* se pasa totalmente el control de la petición al componente destino, por lo que cualquier scriptlet, acción o bloque de texto que aparezca **después de *forward* no será procesado**.

#### 2.5.2.2. Acción *include*

Al igual que *forward*, la acción *include* transfiere la petición a otro componente, aunque una vez que este es ejecutado se vuelve a pasar de nuevo el control a la página origen. Su funcionamiento, como vemos, es el mismo que el del método *include()* de RequestDispatcher.

Al igual que sucede con *forward*, el atributo *page* indica la URL del componente destino:

```
<jsp:include page = "unservlet"/>
```

En este ejemplo, vemos como el componente al que se trasfiere la petición es un servlet.

Es importante resaltar en este punto la diferencia entre la directiva *include* y la acción *include* cuando el componente destino es una página JSP; al ser evaluada en tiempo de ejecución y tanto la página origen como la destino ya se han transformado en servlet, la acción *include* “incluye” en la página origen la respuesta generada por la página destino al producirse la ejecución de esta (o mejor dicho, del servlet equivalente). Por contra, tras la ejecución de la directiva (que tiene lugar una única vez) se genera una única página resultante de incluir el contenido de la página destino en la página origen, transformándose después en un único servlet.

#### 2.5.2.3. Acción *param*

Tanto en la acción *forward* como en *include* puede decidirse agregar algún parámetro adicional a la petición antes de transferirla a la página destino. Esto se lleva a cabo con la acción *param*, indicando en sus atributos *name* y *value* el nombre y valor del parámetro, respectivamente.

La acción *param* debe figurar en el cuerpo de las acciones *include/forward*. En el siguiente ejemplo se añade el parámetro “tipo” a la petición transferida a la página “destino.jsp”:

```
<jsp:forward page = "destino.jsp">
    <jsp:param name = "tipo" value = "x-25"/>
</jsp:forward>
```

Si la petición ya incluyera un parámetro con ese mismo nombre su valor no será sustituido por el nuevo, sino que se añadirá a la lista de valores del parámetro. En ese caso, la página JSP destino deberá utilizar el método *getParameterValues()* para recuperar todos los valores del parámetro.

#### 2.5.2.4. Acción useBean

La acción useBean permite instanciar una clase de tipo JavaBean y almacenar dicha instancia en un atributo de ámbito page, request, session o application para su uso posterior.

Como sabemos, una clase de tipo JavaBean es una clase que encapsula una serie de datos asociados a una determinada entidad, incluyendo métodos de tipo *set/get* para permitir el acceso controlado a los mismos. En el siguiente ejemplo tenemos una clase JavaBean que encapsula los datos asociados a un empleado:

```
public class Empleado{
    private String nombre;
    private int cdep;
    private double salario;
    public Empleado(){
    public Empleado(String nombre, int cdep,
                        double salario){

        this.nombre=nombre;
        this.cdep=cdep;
        this.salario=salario;
    }
    public String getNombre(){
        return nombre;
    }
    public void setNombre(String n){
        this.nombre=n;
    }
    public int getCodigoDep(){
        return cdep;
    }
    public void setCodigoDep(int c){
        this.cdep=c;
    }
    public double getSalario(){
        return salario;
    }
    public void setSalario(double s){
        this.salario=s;
    }
}
```

Como vemos, la clase cuenta con una serie de atributos o datos miembro privados que almacenan la información y unos métodos de tipo *set/get* para establecer y recuperar individualmente cada dato.



A estos métodos se les llama también **métodos de propiedad**, debido a que actúan como propiedades del objeto (set para asignar valor a la propiedad y get para recuperar el valor de la propiedad), siendo su nomenclatura: *setNombrepropiedad* y *getNombrepropiedad* (observar como la primera letra de la propiedad se escribe en mayúsculas).

Aunque suele ser habitual que los nombres de las propiedades coincidan con los nombres de los datos miembro de la clase **no es algo que resulte imprescindible**. Obsérvese por ejemplo en la clase anterior como la propiedad “codigodep” es almacenada en un atributo llamado “cdep”.

Una vez aclarados estos conceptos preliminares sobre JavaBeans, vamos a estudiar el uso de la acción *useBean* dentro de una página JSP. Como ya hemos comentado, esta acción permite crear una instancia de una clase JavaBean, siendo su único requerimiento la existencia de un constructor sin parámetros. Los atributos soportados por useBean para realizar son función son:

- **id**. Identificador asociado a la instancia y con el que se podrá acceder al bean dentro del código. Si ya existe una instancia del JavaBean con este identificador dentro del ámbito definido, la acción no crea una nueva, si no que asigna al identificador una referencia a la existente.
- **class**. Nombre cualificado de la clase. Se deberá especificar obligatoriamente toda la ruta de paquetes y subpaquetes donde está contenida la clase, dado que esta acción no tiene en cuenta las importaciones definidas en el atributo import de la directiva *page*.
- **scope**. Ámbito donde estará ubicada la instancia. Sus posibles valores son: *page*, *request*, *session* y *application*.

Las instancias del JavaBean definidas en una página JSP quedan almacenadas como atributos del objeto de ámbito correspondiente, esto permite que puedan ser utilizadas desde otro servlet o página JSP a través del id del componente. En el siguiente ejemplo crea una instancia del JavaBean Empleado (suponiendo que se encuentra definida en un paquete llamado javabeans) y asigna un valor a una de sus propiedades:

```
<jsp:useBean id="emp" class="javabeans.Empleado"
            scope="session"/>
<%id.setNombre("Juan López");%>
```

Si quisiéramos recuperar el nombre del empleado desde un servlet de la misma aplicación, utilizaríamos:

```
HttpSession ses = request.getSession();
javabeans.Empleado e = (Empleado)ses.getAttribute("emp");
out.println("Nombre del empleado: "+e.getNombre());
```

#### 2.5.2.5. Acción *setProperty*

Además de utilizar directamente el identificador del bean para establecer sus propiedades mediante los métodos set, la acción *setProperty* nos permite realizar esta función sin emplear ningún *scriptlet*.

Los principales atributos de *setProperty* son:

- **name**. Identificador de la instancia cuya propiedad se quiere establecer. Su valor deberá coincidir con el id de alguna instancia creada o recuperada mediante useBean.

- **property.** Nombre de la propiedad que se quiere establecer.
- **value.** Valor que se va a asignar a la propiedad.

Si en el ejemplo anterior quisiéramos utilizar la acción *setProperty* en vez de llamar al método *setNombre()* sería:

```
<jsp:setProperty name="emp" property="nombre"
                value="Juan López"/>
```

Se pueden establecer valores iniciales a las propiedades de un bean en el momento de su creación, para lo cual se deberá utilizar la acción *setProperty* dentro del cuerpo de *useBean*:

```
<jsp:useBean id="emp" class="javabeans.Empleado"
            scope="session">
    <jsp:setProperty name="emp" property="nombre"
                    value="Juan López"/>
    <jsp:setProperty name="emp" property="codigoDep"
                    value="35"/>
    <jsp:setProperty name="emp" property="salario"
                    value="37.400"/>
</jsp:useBean>
```

Obsérvese como el valor de *property* debe corresponder con el nombre de la propiedad, no con el del dato miembro que la almacena.

El uso de la acción *setProperty* no solamente permite **establecer valores a las propiedades de un bean sin utilizar código Java**, también simplifica enormemente el volcado de los datos procedentes de una petición HTTP en un bean. Supongamos por ejemplo que tenemos una página HTML como la que se indica en la figura 8 para capturar los datos de un empleado.

The image shows a screenshot of a Microsoft Internet Explorer browser window. The title bar indicates the URL is 'http://localhost:8084/ejemplo11\_uso\_beans/datos.html'. The browser's address bar also shows this URL. The main content area displays a form titled 'Formulario de datos' in a large, bold, serif font. Below the title, there are three text input fields. The first is preceded by the label 'Introduzca nombre:', the second by 'Introduzca código departamento:', and the third by 'Introduzca salario:'. All three labels are in a standard serif font. Below the third input field is a button labeled 'Enviar'. The browser's status bar at the bottom shows 'Listo' on the left and 'Intranet local' on the right.

**Figura. 8.**

El código HTML de la página anterior será el que se indica en el siguiente listado:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
    <center>
      <h1>Formulario de datos</h1>
      <form action="captura.jsp" method="POST">
        <table>
          <tr>
            <td>Introduzca nombre: </td>
            <td><input type="text" name="nombre"/></td>
          </tr>
          <tr>
            <td>Introduzca código departamento: </td>
            <td><input type="text" name="codigoDep"/></td>
          </tr>
          <tr>
            <td>Introduzca salario: </td>
            <td><input type="text" name="salario"/></td>
          </tr>
          <tr>
            <td align="center" colspan="2">
              <input type="submit" value="Enviar"/>
            </td>
          </tr>
        </table>
      </form>
    </center>
  </body>
</html>

```

Obsérvese como **los nombres de los campos de texto deben coincidir con los nombres de las propiedades de la clase Empleado**. Tal y como se indica además en el atributo *action* del formulario, los datos son enviados a una página llamada *captura.jsp* para su procesamiento. Pues bien, si utilizamos en esta página destino la acción *useBean* para crear una instancia de *Empleado*, podemos mediante una única acción *setProperty*, volcar el contenido de cada uno de los tres campos enviados en la petición en sus respectivas propiedades del bean, tan sólo será necesario indicar el símbolo "\*" como valor del atributo *property* de la acción:

```

<jsp:useBean id="empleado" class="javabeans.Empleado"
              scope="request">

```

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

```
</jsp:useBean>
```

Como se puede ver, la combinación de ambas acciones para recuperar y encapsular los datos cliente permite ahorrar una gran cantidad de código Java, además de simplificar enormemente el proceso.

#### 2.5.2.6. Acción *getProperty*

Esta acción recupera el contenido de alguna de las propiedades de un bean e inserta su valor en la página de respuesta. Los dos atributos que proporciona esta acción tienen el mismo significado que los equivalentes de la acción *setProperty*:

- **name.** Nombre de la instancia.
- **property.** Nombre de la propiedad cuyo valor se puede recuperar.

Supongamos que desde la página *captura.jsp* del ejemplo anterior se quiere reenviar la petición a otra página, llamada *muestra.jsp*, que se encargará de mostrar los datos contenidos en el bean. Para ello, será necesario añadir la acción *forward* a la página *captura.jsp*:

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

Por su parte, la nueva página *muestra.jsp* quedará como se indica a continuación:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Página de datos</title>
    </head>
    <body>
        <h1>Datos de usuario:</h1>
        <jsp:useBean id="empleado" class="javabeans.Empleado"
            scope="request" />
        Nombre: <b><jsp:getProperty name="empleado"
            property="nombre"/></b><br/>
        Departamento: <b><jsp:getProperty name="empleado"
            property="codigoDep"/></b><br/>
        Salario: <b><jsp:getProperty name="empleado"
            property="salario"/></b><br/>
    </body></html>
```

## EJERCICIO 5

En este ejercicio vamos a desarrollar una aplicación que permita intercambiar mensajes entre los usuarios de la misma. Los mensajes serán almacenados en una base de datos cuya única tabla “mensajes” tendrá la estructura que se indica en la siguiente figura:

### mensajes

Campo	Tipo de dato
destinatario	varchar(50)
remitente	varchar(50)
texto	varchar(500)

Figura. 9.

El esquema con los componentes de la aplicación se muestra en la figura 10:

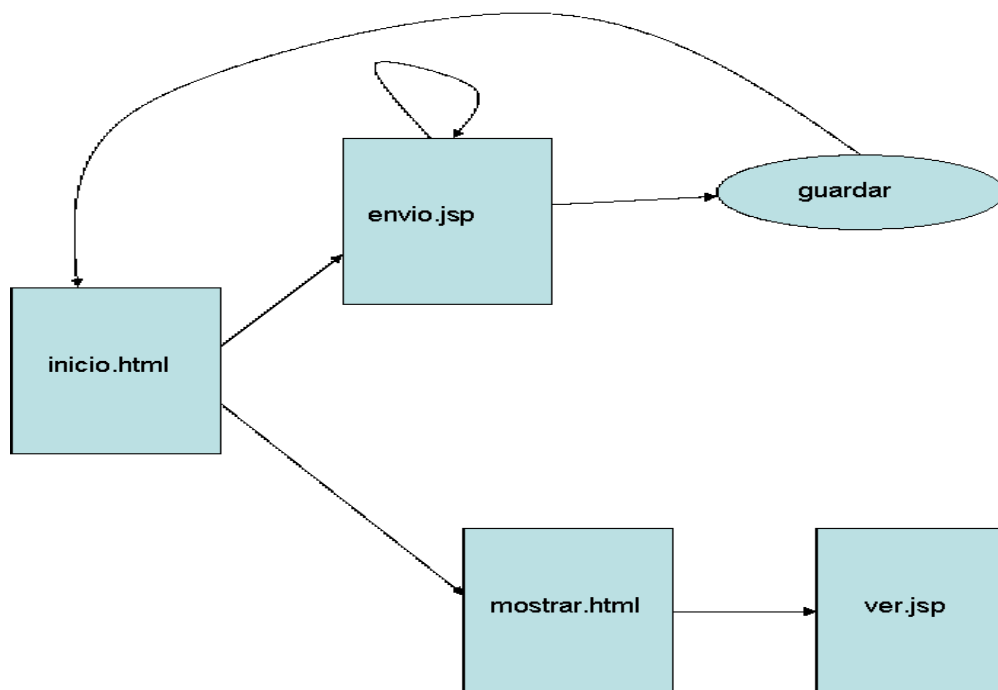


Figura. 10.

Seguidamente, iremos explicando la funcionalidad de cada uno de ellos y mostraremos el código de los mismos.

#### Inicio.html

Es la página de entrada a la aplicación. Dispone de dos enlaces para acceder a las opciones de envío y visualización de mensajes (figura 11).

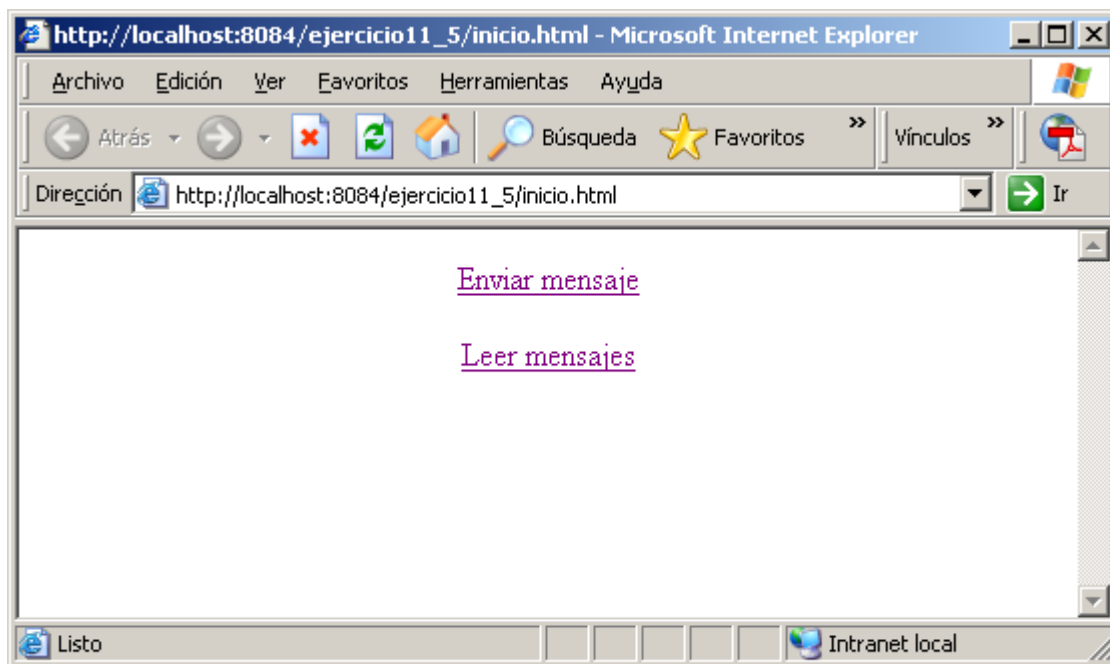


Figura. 11.

El código de esta página HTML se indica en el siguiente listado:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
    <center>
      <a href="envio.jsp">Enviar mensaje</a><br/><br/>
      <a href="mostrar.html">Leer mensajes</a><br/><br/>
    </center>
  </body>
</html>
```

### Envio.jsp

Solicita los datos del mensaje y los encapsula en un javabean de tipo "Mensaje" que encapsula los datos del mismo. Una vez generado el javabean se almacena en un atributo de petición y reenvía esta al servlet "guardar" para que proceda al almacenamiento del mensaje en la BD.

El aspecto de esta página JSP se indica en la figura 12.

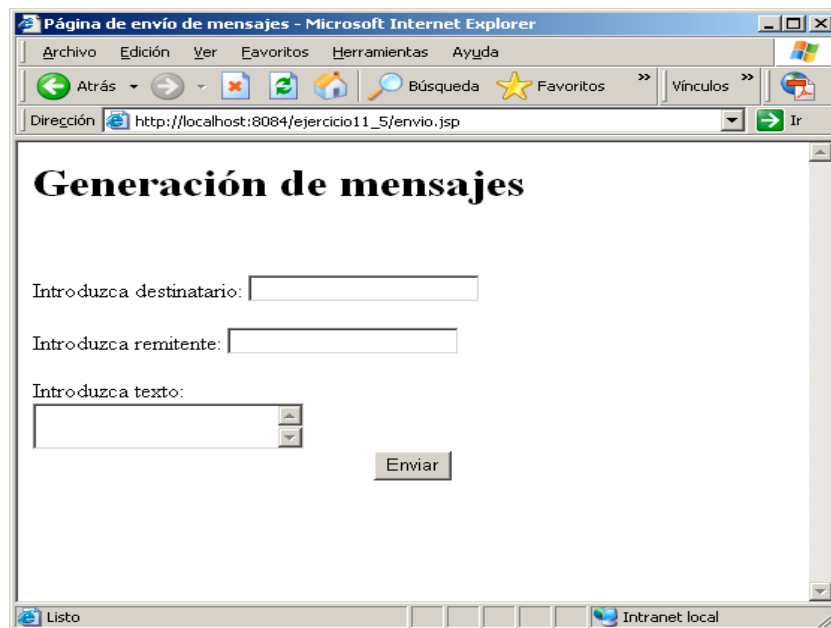


Figura. 12.

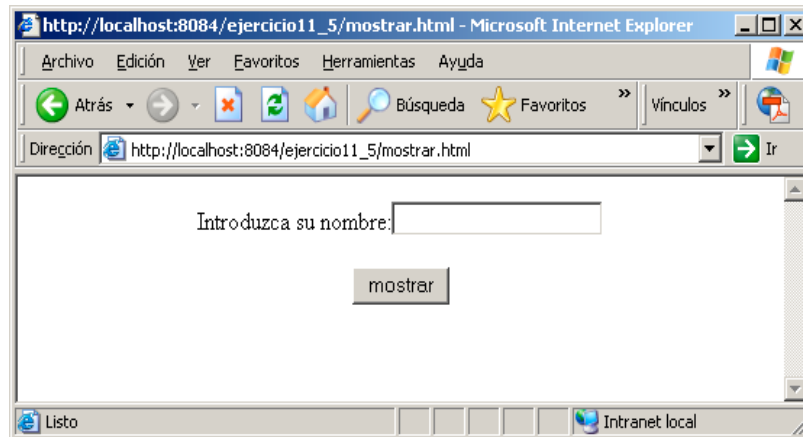
El código de la misma es el que se muestra en el siguiente listado:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Página de envío de mensajes</title>
  </head>
  <body>
    <%if(request.getParameter("texto")!=null){%>
      <jsp:useBean id="mensa" scope="request"
                  class="javabeans.Mensaje" />
      <jsp:setProperty name="mensa" property="*" />
      <!--pasa la petición al servlet guardar
      para que grabe el mensaje en la BD--%>
      <jsp:forward page="guardar"/>
    <%}%>
    <h1>Generación de mensajes</h1>
    <form method="post">
      <br/><br/>
      Introduzca destinatario: <input type="text"
                                   name="destino"/><br/><br/>
      Introduzca remitente: <input type="text"
                                   name="remite"/><br/><br/>
      Introduzca texto: <br/>
      <textarea name="texto">
      </textarea>
      <br/>
      <center>
        <input type="submit" name="Submit"
                value="Enviar"/>
      </center>
    </form>
  </body>
</html>
```

### Mostrar.html

Se encarga de solicitar el nombre del destinatario cuyos mensajes quieren ser leídos (figura 13). El nombre del destinatario es pasado a la página JSP “ver.jsp” que será la encargada de mostrar los mensajes.



**Figura. 13.**

El código HTML de esta página se indica a continuación:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title></title>
  </head>
  <body>
    <center>
      <form action="ver.jsp" method="post">
        Introduzca su nombre:<input type="text"
                                name="nombre"/><br/><br/>
        <input type="submit" value="mostrar"/>
      </form>
    </center>
  </body>
</html>
```

### Guardar

El servlet “guardar” recibe un javabeen “Mensaje” con los datos del mensaje y lo almacena en la base de datos, cuyos datos de conexión (driver y cadena conexión) se encuentran almacenados como parámetros de contexto en el archivo web.xml. El acceso a los datos se realiza a través de una clase de negocio llamada “Operaciones”

Después de registrar el mensaje redireccionará al usuario a la página de inicio. El siguiente listado corresponde con el código de este servlet:

```
package servlets;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
```



```

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import negocio.*;

public class Guardar extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException {
        //recupera el javabean Mensaje
        Mensaje men=(Mensaje)request.getAttribute("mensa");
        //obtiene los parámetros de contexto relativos
        //a la base de datos
        String driver=getServletContext().
            getInitParameter("driver");
        String cadenacon=getServletContext().
            getInitParameter("cadenacon");
        Operaciones oper=new Operaciones(driver,cadenacon);
        oper.graba(men);
        response.sendRedirect("inicio.html");
    }
}

```

### ver.jsp

Esta es la página que muestra los mensajes correspondientes al destinatario que recibe como parámetro. Si no tiene mensajes, se reenviará la petición a una página de error llamada “nomensajes.jsp”.

Al igual que el servlet “guardar”, el acceso a los datos se realiza a través de la clase de apoyo “Operaciones”.

El aspecto de la página se muestra en la figura 14.



**Figura. 14.**

El código de la misma corresponde al siguiente listado:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javax.servlet.*,"
               "javax.servlet.http.*,"
               "negocio.Operaciones"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

```

```

<html>
  <head>
    <title>Página de mensajes</title>
  </head>
  <body>
    <%String nombre=request.getParameter("nombre");
    String driver=application.getInitParameter("driver");
    String cadenacon=application.getInitParameter("cadenacon");
    Operaciones op=new Operaciones(driver,cadenacon);%>
    <h1> Mensajes para <%=nombre%></h1>
    <table border="1">
      <tr><th>Remitente</th><th>Mensaje</th></tr>
      <%ArrayList<Mensaje> lista=op.getMensajes(nombre);
      if(lista!=null && lista.size()>0){
        for(Mensaje m:lista){%>
          <tr><td>
            <%=m.getRemite() %>
          </td><td>
            <%=m.getTexto() %>
          </td></tr>
        <%>
      }
      else{%>
        <jsp:forward page="nomensajes.jsp"/>
      <%}%>
    </table>
    <br/>
    <a href="inicio.html">Inicio</a>
  </body>
</html>

```

### nomensajes.jsp

Su misión es mostrar un mensaje de aviso al usuario advirtiéndole que el destinatario indicado no tiene mensajes. El código de esta página es el que se muestra a continuación:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Página de aviso</title>
  </head>
  <body>
    <center>
      <h2>Lo siento, <%=request.getParameter("nombre")%>
        no tiene mensajes</h2>
    </center>
  </body>
</html>

```

### Mensaje

Este elemento, que no aparece en el esquema de la figura 10, corresponde a la clase javabeans que encapsula los datos del mensaje:

```
package javabeans;
```

```

public class Mensaje {
    private String remite;
    private String destino;
    private String texto;
    public Mensaje(){} //constructor sin parámetros
    public Mensaje(String remite, String destino, String texto){
        this.remite=remite;
        this.destino=destino;
        this.texto=texto;
    }
    public void setRemite(String remite){
        this.remite=remite;
    }
    public String getRemite(){
        return remite;
    }
    public void setDestino(String destino){
        this.destino=destino;
    }
    public String getDestino(){
        return destino;
    }
    public void setTexto(String texto){
        this.texto=texto;
    }
    public String getTexto(){
        return texto;
    }
}

```

## Operaciones

Como ya hiciéramos en el tema anterior con los servlets, el acceso a datos no se incluye directamente en la página JSP, sino que se encapsula en clases de negocio independientes a fin de facilitar la reutilización, estructurar y facilitar el desarrollo:

```

package negocio;

import java.sql.*;
import javax.swing.*;
import java.util.*;

public class Operaciones {
    private String driver, cadenacon;
    public Operaciones(String driver, String cadenacon) {
        this.driver=driver;
        this.cadenacon=cadenacon;
    }
    public Connection obtenerConexion(){
        Connection cn=null;
        try{
            Class.forName(driver);
            cn=DriverManager.getConnection(cadenacon);
        }
        catch(Exception e){e.printStackTrace();}
        return cn;
    }
    public ArrayList<Mensaje> getMensajes(String destino){
        Connection cn=null;
        ArrayList<Mensaje> lista=null;
        Statement st;

```

```

        ResultSet rs;
        try{
            cn=obtenerConexion();
            st=cn.createStatement();
            String tsq;
            tsq="select * from mensajes where destinatario='"+
                                                    destino+"'";

            rs=st.executeQuery(tsq);
            lista=new ArrayList<Mensaje>();
            while(rs.next()){
                Mensaje m=new Mensaje(rs.getString("remitente"),
                                      rs.getString("destinatario"),
                                      rs.getString("texto"));

                lista.add(m);
            }
            cn.close();
        }
        catch(Exception e){e.printStackTrace();}
        return(lista);
    }

    public void graba(Mensaje m){
        Connection cn;
        Statement st;
        ResultSet rs;
        try{
            cn=obtenerConexion();
            st=cn.createStatement();
            String tsq;
            tsq="Insert into mensajes values('";
            tsq+=m.getDestino()+"', '"+
                m.getRemite()+"', '"+
                m.getTexto()+"' )";
            st.execute(tsq);
            cn.close();
        }
        catch(Exception e){e.printStackTrace();}
    }
}

```

### 3. EL LENGUAJE EL

---

El lenguaje de expresiones de JSP, más conocido como EL, fue incorporado a la especificación JSP a partir de la versión 2.0. Consiste en la utilización de una serie de expresiones de texto para la generación de resultados dentro de la página JSP.

Las expresiones EL tienen como **objetivo es reemplazar a las clásicas expresiones JSP basadas en la utilización de código Java**, de hecho, empleadas conjuntamente con las acciones JSTL que veremos en el próximo apartado, permiten crear páginas JSP libres de código Java. Esta tendencia a posibilitar la creación de páginas JSP sin utilizar código Java es quizá el objetivo más importante de las últimas versiones de la especificación JSP, ya que, en teoría, esto permitiría a perfiles con menos nivel de conocimientos de programación (como diseñadores Web) crear este tipo de componentes, además de simplificar el desarrollo y facilitar su mantenimiento.

### 3.1. EXPRESIONES EL

Las expresiones EL aparecen encerradas entre los símbolos “\${” y “}”, devolviendo un valor al lugar de la página donde aparecen situadas.

Por ejemplo, la siguiente línea en el interior de una página JSP generaría una frase en la que se incluiría el valor de la variable “dato”:

```
La información registrada vale: ${dato}
```

Como hemos indicado, estas expresiones llevan a cabo la misma función que la realizada por las expresiones Java encerradas entre “<%=” y “%>”. Así, la línea anterior es equivalente a:

```
La información registrada vale: <%=dato%>
```

Las expresiones EL no se limitan a variables JSP, también pueden incluir objetos implícitos EL, datos almacenados en cualquiera de los ámbitos de aplicación o incluso alguna operación entre datos utilizando alguno de los operadores soportados por EL. Por ejemplo, la siguiente instrucción mostraría en la página el valor del parámetro “nombre” recibido en la petición HTTP:

```
Bienvenido sr./a ${param['nombre']}
```

Es importante destacar que una expresión EL no puede acceder a variables u objetos creados en scriptlets Java. Por ejemplo, el siguiente bloque de sentencias no generaría ningún resultado en la página puesto que la variable “contador” no existiría para EL:

```
<%int contador=1;%>  
El contador vale: ${contador}
```

### 3.2. ACCESO A OBJETOS MEDIANTE EXPRESIONES EL

Para acceder a cualquiera de los objetos utilizados en la aplicación dentro de una expresión EL tan sólo tenemos que indicar el identificador del objeto entre los símbolos “\${” y “}”. Este objeto será localizado por el contenedor en cualquiera de los ámbitos de la aplicación (page, request, session y application).

Si este objeto es de tipo JavaBean el acceso a las propiedades de mismo se llevará a cabo utilizando la sintaxis:

```
objeto.propiedad
```

Por ejemplo, si tenemos un objeto de identificador “cliente” con dos propiedades: “usuario” y “password” y dicho objeto se encuentra almacenado en un atributo de sesión, las siguientes instrucciones mostrarían los valores de las propiedades en la página:

```
Usuario: ${cliente.usuario}<br/>  
Password: ${cliente.password}
```

Cuando el objeto es una colección basada en índices se puede acceder a cada uno de los elementos utilizando la expresión objeto[indice]. Por ejemplo, para mostrar el primero de los nombres de una colección ArrayList sería:

```
El primer nombre es: ${nombres[0]}
```

Si la colección está basada en claves podemos utilizar objeto[“clave”] para acceder al valor del elemento o objeto.clave. Por ejemplo, si tenemos una colección llamada “personas”

donde a cada nombre de persona se le asocia como clave el DNI, para mostrar el nombre de la persona con DNI “18934598J” sería:

```
Nombre: ${personas["18934598J"]}
```

o

```
Nombre: ${personas.18934598J}
```

Si cada persona es un JavaBean con las propiedades “nombre” y “teléfono” el acceso al nombre de la persona anterior sería:

```
Nombre: ${personas["18934598J"].nombre}
```

o

```
Nombre: ${personas.18934598J.nombre}
```

### 3.3. OBJETOS IMPLÍCITOS EL

El lenguaje EL integra una serie de objetos con los que se puede acceder de una forma sencilla a los distintos tipos de información que habitualmente se utilizan en una aplicación Web, como son los atributos de página, petición, sesión o aplicación, los parámetros y encabezados de la petición, las cookies o los parámetros de contexto.

Estos objetos son expuestos como colecciones de tipo Map, accediéndose a sus propiedades y atributos a través de la expresión:

```
${objeto_implicito[clave]}
```

siendo “clave” el nombre de la propiedad o atributo cuyo valor se quiere obtener.

Los objetos implícitos que forman parte del lenguaje EL son:

- **pageScope.** Proporciona acceso a los atributos de ámbito de página, utilizándose como clave el nombre o identificador del atributo. Por ejemplo, tenemos el siguiente javabeen “cliente” instanciado en una página JSP:

```
<jsp:useBean id="cliente" class="javabeans.Cliente">
<jsp:setProperty name="cliente"
                property="usuario" value="profe"/>
<jsp:setProperty name="cliente"
                property="password" value="admin"/>
</jsp:useBean>
```

Si queremos mostrar el valor del campo usuario en cualquier parte de la página (al no haberse definido el atributo “scope” el ámbito por defecto será “page”) utilizaríamos:

```
Nombre: ${pageScope["cliente"].usuario}
```

o

```
Nombre: ${pageScope.cliente.usuario}
```

- **requestScope.** Proporciona acceso a las variables de ámbito de petición. Por ejemplo, dado el siguiente bloque de sentencias incluidas en una página JSP:

```
<%int edad=40
request.setAttribute("edad", edad);%>
<jsp:forward page="prueba.jsp"/>
```

Para mostrar en la página prueba.jsp el valor del atributo de petición “edad”, utilizaríamos:

```
Su edad es de: ${requestScope["edad"]}
```

- **sessionScope.** Proporciona acceso a los atributos de ámbito de sesión. Por ejemplo, supongamos que en una página JSP tenemos el siguiente bloque de instrucciones:

```
<jsp:useBean id="cliente" class="javabeans.Cliente"
scope="session"/>
<jsp:setProperty name="cliente"
property="password" value="admin"/>
<%response.sendRedirect("datos.jsp");%>
```

Para mostrar en datos.jsp el valor del password utilizaríamos:

```
El password registrado es: &{cliente.password}
```

- **applicationScope.** Proporciona acceso a los atributos de ámbito de aplicación, es decir, aquellos registrados en el objeto application (ServletContext).
- **param.** Este objeto nos da acceso a los parámetros enviados en la petición. Por ejemplo, el siguiente bloque de sentencias inicializaría la propiedad “nombre” del JavaBean “usuario” con el valor del parámetro “user” recibido en la petición:

```
<jsp:useBean id="cliente" class="javabeans.Cliente"/>
<jsp:setProperty name="cliente" property="usuario"
value="${param['user']}" />
```

Obsérvese como, en el ejemplo anterior, el resultado de la expresión EL se emplea como valor de atributo para una acción JSP.

- **paramValues.** Al igual que el anterior proporciona acceso a los parámetros de petición, solo que en este caso el valor de cada parámetro se recupera como un array de cadenas. Se utiliza en los casos en que el parámetro incluye múltiples valores. El siguiente ejemplo
- **header.** Proporciona acceso a los encabezados de la petición. El siguiente ejemplo mostraría en la página el valor del encabezado user-agent enviado en la petición actual:

```
Tipo navegador: ${header['user-agent']}
```

- **headerValues.** Al igual que header proporciona acceso a los encabezados de la petición, devolviendo en este caso un array de cadenas con todos los valores que le han sido asignados al encabezado. El siguiente ejemplo mostraría todos los valores indicados en el atributo “accept”:

```
<c:foreach var="tipo" items="${headerValues.accept}">
```

```

        <c:out value="${tipo}"/>
    </c:foreach>

```

En este ejemplo hemos utilizado las acciones JSTL `foreach` y `out` para recorrer el array; en siguiente apartado estudiaremos la sintaxis y funcionamiento de estas acciones.

- **initParam.** Proporciona acceso a los parámetros de contexto definidos en el archivo `web.xml`. Para acceder al valor de cualquiera de estos parámetros utilizaremos la expresión:

```

${initParam["nombre_parametro"]}

```

- **cookie.** Proporciona acceso a las cookies enviadas en la petición. Cada elemento de la colección `Map` asociada representa un objeto `Cookie`. La siguiente expresión de ejemplo mostraría en la página el contenido de la cookie `"user"`:

```

Usuario: ${cookie["user"]}

```

Además de los objetos anteriores, el lenguaje EL proporciona otro objeto implícito llamado **pageContext** que proporciona acceso al contexto de la aplicación. Entre otras dispone de una serie de propiedades que nos dan acceso a los objetos implícitos JSP, por ejemplo, la siguiente expresión EL permitiría recuperar el método de envío utilizado en la petición actual:

```

${pageContext.request.method}

```

### 3.4. OPERADORES EL

El lenguaje EL también incluye una serie de operadores que permiten manipular datos dentro de una expresión.

En la figura 15 se muestran los operadores más utilizados, indicando entre paréntesis la abreviatura alternativa utilizada.

Categoría	Operadores
Aritméticos	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> ( <code>div</code> ), <code>%</code> ( <code>mod</code> )
Relacionales	<code>==</code> ( <code>eq</code> ), <code>!=</code> ( <code>ne</code> ), <code>&lt;</code> ( <code>lt</code> ), <code>&gt;</code> ( <code>gt</code> ), <code>&lt;=</code> ( <code>le</code> ) y <code>&gt;=</code> ( <code>ge</code> )
Lógicos	<code>&amp;&amp;</code> ( <code>and</code> ), <code>  </code> ( <code>or</code> ) y <code>!</code> ( <code>not</code> )
Condicional	<code>?</code>

Figura. 15. Tabla de operadores EL

## 4. LA LIBRERÍA DE ACCIONES JSTL

El lenguaje EL despliega toda su potencia al emplearse conjuntamente con la librería de acciones JSTL.

JSTL consiste en un conjunto de acciones personalizadas JSP, desarrolladas por Sun, que permiten realizar tareas de procesamiento habituales en una página JSP sin necesidad de recurrir a `scriptlets` Java.



La última versión de JSTL (JSTL 1.1) consta de cuatro subtipos de librerías, cada una de las cuales proporciona tags o etiquetas especializadas en un determinado tipo de funcionalidad. Estas sublibrerías son:

- **core.** Es la más importante de todas y en la que centraremos nuestro estudio. Incluye acciones de propósito general para realizar tareas habituales en cualquier tipo de aplicación, como el acceso a atributos en distintos ámbitos, recorrido de colecciones, ejecución condicional, etc.
- **xml.** Proporciona tags para la manipulación de documentos XML dentro de la página.
- **sql.** Contiene acciones especializadas en el acceso a información almacenada en bases de datos.
- **fmt.** Sus acciones están especializadas en el formateo de datos y la internacionalización de aplicaciones.

#### 4.1. INSTALACIÓN DE JSTL

Debido a su popularidad, JSTL ha sido incorporada a la especificación JSP a partir de la versión Java EE 1.5, aunque se puede descargar de manera independiente de la página Web de Sun.

Al hacerlo, obtendremos un .zip con dos archivos `jstl.jar` y `standard.jar` que contienen la implementación de las distintas acciones que componen las librerías. Ambos archivos tendrán que ser incluidos en el directorio `WEB-INF/lib` de la aplicación.

El entorno de desarrollo NetBeans proporciona una librería llamada JSTL 1.1 que contiene ambos archivos, por lo que si estamos utilizando este entorno lo único que tendremos que hacer para poder utilizar JSTL es incluir una referencia a esta librería. Para ello nos situamos sobre la carpeta “libraries” del proyecto y en el menú que aparece al pulsar el botón derecho del mouse elegimos “Add Library”, abriéndose el cuadro de diálogo que se indica en la figura 16.

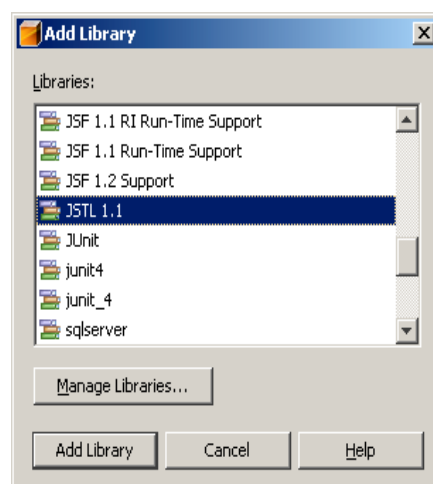


Figura. 16.

En este cuadro de diálogo elegimos la librería indicada y pulsamos el botón “Add Library” para añadirla al proyecto.

## 4.2. UTILIZACIÓN DE JSTL EN UNA PÁGINA JSP

A partir de ahí, ya podemos utilizar las acciones JSTL en cualquiera de las páginas JSP del proyecto. Será necesario, no obstante, **incluir en cada página JSP una referencia al archivo descriptor de la sublibrería que vayamos a utilizar**, en nuestro caso será la sublibrería *core*, cuya uri asociada es:

<http://java.sun.com/jsp/jstl/core>

La referencia a un archivo de librería externo se realiza mediante la directiva *taglib* que comentamos en apartados anteriores pero que no llegamos a estudiar con detalle. Esta directiva requiere dos atributos:

- **uri.** Dirección de la librería cuyas acciones se quieren utilizar. No se trata de una dirección real, sino de una dirección lógica asociada al archivo descriptor de librería *.tld* que describe las diferentes acciones proporcionadas. La asociación entre la dirección lógica y la dirección real del *.tld* puede establecerse en el documento de configuración de la aplicación *web.xml*, aunque en el caso de que el *.tld* esté incluido dentro de los archivos *.jar* de la librería (como es el caso de *core* y el resto de sublibrerías de JSTL) no será necesario incluir esta asociación, ya que el contenedor Web intentará localizar de forma predeterminada los archivos descriptores en los *.jar*.
- **prefix.** Prefijo utilizado para referirse a las acciones de la librería desde el interior de la página JSP. Como en una página JSP es posible utilizar acciones de distintas librerías, la forma que tiene el contenedor de saber a qué librería en concreto pertenece una acción es utilizando el prefijo delante del nombre de ésta:

`<prefijo:nombre_accion>`

El programador es libre de asociar el prefijo que estime apropiado a cada librería.

Así pues, en el caso concreto de la librería *core*, la directiva *taglib* que habría que incluir en cada página JSP tendría el siguiente aspecto:

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

y para referirse desde la página a cualquiera de las acciones de la librería sería `<c:nombre_accion>`. Se ha utilizado el prefijo "c" para la librería *core* por convenio, pero se puede emplear cualquier otro.

## 4.3. ESTUDIO DE LAS PRINCIPALES ACCIONES DEL CORE DE JSTL

Seguidamente vamos a analizar las principales acciones JSTL que componen la librería *core*. Para facilitar el estudio vamos a dividir este conjunto de acciones en dos grupos, según la funcionalidad proporcionada por las mismas:

- Acciones de propósito general
- Acciones de control de flujo

#### 4.3.1. Acciones de propósito general

Como se desprende de su nombre, se trata de acciones que realizan tareas habituales en una página JSP, como la inserción de datos en la página de respuesta, manipulación de variables, etc.

Veamos a continuación las acciones más importantes de este grupo.

##### 4.3.1.1. *out*

Realiza la misma función que una expresión JSP, es decir inserta el valor indicado en su atributo *value* dentro de la página de respuesta. Este valor puede ser una constante, expresión EL o una expresión JSP:

```
<c:out value="${dato}" />
```

En caso de que la variable indicada no exista, se puede suministrar un valor por defecto a través del atributo *value*:

```
<c:out value="${dato}" default="0" />
```

##### 4.3.1.2. *set*

Se emplea para establecer un valor en una variable JSP, en una propiedad de un JavaBean o en una colección de tipo Map. Para establecer un valor en una variable utilizaríamos los siguientes atributos:

- **var.** Identificador de la variable JSP a la que se asignará el valor.
- **value.** Valor que se asignará a la variable

La siguiente instrucción asigna el valor 40 a la variable “num”:

```
<c:set var="num" value="${5*8}" />
```

Lo anterior se podría haber hecho también prescindiendo del atributo *value*, indicando el valor a asignar en el cuerpo de la acción:

```
<c:set var="num">
    ${5*8}
</c:set>
```

Si lo que queremos es asignar un valor a alguna de las propiedades de un JavaBean existente en alguno de los ámbitos de la aplicación, deberíamos utilizar los atributos:

- **target.** Identificador del objeto JavaBean.
- **property.** Propiedad del objeto JavaBean a la que se le asignará el valor
- **scope.** Ámbito del JavaBean. Si no se especifica este atributo se intentará localizar el objeto en todos los ámbitos de la aplicación, siguiendo el orden: *page*, *request*, *session* y *application*

En este otro ejemplo se asigna el valor del parámetro “user” a la propiedad “usuario” del JavaBean “cliente”:

```
<c:set target="cliente"
    property="usuario" value="${param['user']}" />
```

Lo anterior es equivalente a:

```
<c:set target="cliente" property="usuario">
    ${param['user']}
</c:set>
```

Para trabajar con una colección de tipo Map sería similar un JavaBean, excepto que el atributo *property* reflejaría la clave de una de las entradas del mapa. El siguiente ejemplo asigna como valor de la entrada de clave "555059T" el objeto "persona" almacenado en un atributo de aplicación:

```
<c:set target="personas" property="555059T "
    value="${sessionScope['persona']}" />
```

#### 4.3.1.3. *remove*

Elimina una variable existente en uno de los ámbitos de la aplicación. Sus atributos son:

- **var.** Identificador de la variable a eliminar
- **scope.** Ámbito donde se encuentra la variable. Si no se especifica se buscará en todos los ámbitos de la aplicación.

#### 4.3.1.4. *redirect*

### 4.3.2. **Acciones de control de flujo**

Se utilizan para controlar el flujo de ejecución dentro de las páginas JSP, al igual que hacemos con las instrucciones if, switch y for de JSP

Son cuatro las acciones que componen este grupo, pasemos a analizarlas.

#### 4.3.2.1. *if*

Evalúa el cuerpo de la acción si el resultado de la condición indicada en su atributo *test* es *true*. El código del siguiente ejemplo mostraría en la página de respuesta el valor de la variable "dato" en caso de que éste sea un número par:

```
<c:if test="${dato%2 == 0}">
    El valor es <c:out value="${dato}" />
</c:if>
```

**JSTL no dispone de un equivalente a la instrucción else**, por lo que en el caso de querer realizar una acción alternativa habrá que **utilizar choose**.

#### 4.3.2.2. *choose*

Su objetivo es el mismo que el de la instrucción switch de Java, solo que en este caso se comprueban varias condiciones, evaluándose el cuerpo de la primera que resulte verdadera. Cada condición es evaluada con un elemento *when* de estructura similar a if. El formato de <c:choose> es:

```

<c:choose>
  <c:when test="condicion1">
    bloque1
  </c:when>
  <c:when test="condicion2">
    bloque2
  </c:when>
  :

  <c:otherwise>
    otro_bloque
  </c:otherwise>
</c:choose>

```

Si no se cumple ninguna de las condiciones especificadas en *when* se ejecutará el bloque indicado en *otherwise* en caso de que se haya definido.

El siguiente ejemplo muestra distintos mensajes en la página de respuesta en función del valor de la variable hora:

```

<c:choose>
  <c:when test="\${hora}>8 && hora<13}">
    Buenos días
  </c:when>
  <c:when test="\${hora}>13 && hora<20}">
    Buenas tardes
  </c:when>
  <c:otherwise>
    Buenas noches
  </c:otherwise>
</c:choose>

```

#### 4.3.2.3. *foreach*

Procesa de forma repetitiva el cuerpo de la acción. Se puede utilizar de dos maneras:

- **Para recorrer un rango de valores numérico.** En este caso la variable especificada en su atributo *var* es inicializada al valor numérico indicado en el atributo *begin*, evaluándose el cuerpo de la acción hasta que la variable alcance el valor indicado en el atributo *end*. El valor de incremento de la variable al final de cada iteración deberá ser especificado mediante el atributo *step*.

El siguiente bloque de acciones mostraría en la página de respuesta la tabla de multiplicar del número 5:

```

<table border="1">
  <c:foreach var="i" begin="1" end="10" step="1">
    <tr><td><c:out value="\${i*5}" /></td></tr>
  </c:foreach>
</table>

```

- **Para iterar una colección.** En este caso la variable indicada en `var` recorrerá la colección especificada en el atributo `items`, apuntado con la variable indicada en `var` a cada elemento de la colección. El cuerpo de la acción será evaluado con cada iteración. Suponiendo que tenemos una colección llamada "personas" que almacena una lista de cadenas, el siguiente código insertaría en la página de respuesta cada una de esas cadenas:

```
<c:foreach var="cadena" items="{personas}">
    <b><c:out value="{cadena}" /></b>
</c:foreach>
```

#### 4.3.2.4. *fortokens*

Se utiliza para recorrer el conjunto de objetos `String` o *tokens* que componen una determinada cadena de caracteres. El separador de objetos es definido por la propia acción. Los siguientes son los atributos soportados por *fortokens*:

- **items.** Cadena de caracteres cuyo contenido se va a recorrer.
- **var.** Variable que apuntará en cada iteración a uno de los *tokens* de la cadena.
- **delims.** Indica el carácter utilizado como separador de tokens.

En la siguiente página JSP de ejemplo se recorre una cadena de caracteres formada por una lista con los nombres de las estaciones del año, incluyendo cada uno de ellos en una celda de una tabla HTML dentro de la página de respuesta.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head></head>
<body>
    <c:set var="estaciones" value="primavera, verano,
        otoño, invierno"/>
    <table border="1">
        <c:forTokens var="estacion" items="{estaciones}"
                                delims=", ">
            <tr>
                <td>
                    <c:out value="{estacion}" />
                </td>
            </tr>
        </c:forTokens >
    </table>
</body>
</html>
```

Vamos a crear una nueva versión de la aplicación de envío y recepción de mensajes desarrollada en el ejercicio 5, en la que sustituiremos todos los script de código Java de las páginas existentes por acciones JSTL y expresiones EL.

Antes de nada, lo primero que haremos será incluir un constructor sin parámetros en la clase Operaciones que nos permita crear instancias de ella a través de la acción useBean, así como dos nuevos métodos llamados *setDriver()* y *setCadenacon()* que nos permitan asignar valores a estos atributos. También incluiremos una versión sobrecargada del método *getMensajes()* que recupere la lista completa de mensajes existentes en la BD. El siguiente listado corresponde a las modificaciones añadidas en la clase Operaciones, omitiéndose el resto de código existente en la clase que permanecerá intacto:

```
public class Operaciones {
    :
    public Operaciones() {}
    :
    public void setDriver(String driver) {
        this.driver=driver;
    }
    public void setCadenacon(String cadenacon) {
        this.cadenacon=cadenacon;
    }
    :
    public ArrayList<Mensaje> getMensajes() {
        Connection cn=null;
        ArrayList<Mensaje> lista=null;
        Statement st;
        ResultSet rs;
        try{
            cn=obtenerConexion();
            st=cn.createStatement();
            String tsq;
            tsq="select * from mensajes";
            rs=st.executeQuery(tsq);
            lista=new ArrayList<Mensaje>();
            while(rs.next()){
                Mensaje m=new Mensaje(rs.getString("remitente"),
                                     rs.getString("destinatario"),
                                     rs.getString("texto"));
                lista.add(m);
            }
            cn.close();
        }
        catch(Exception e){e.printStackTrace();}
        return(lista);
    }
}
```

En cuanto las páginas JSP, a continuación se indica el nuevo contenido de las mismas:

#### envio.jsp

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Página de envío de mensajes</title>
    </head>
    <body>
```

```

<c:if test="${not empty param.texto}">
    <jsp:useBean id="mensa" scope="request"
        class="javabeans.Mensaje" />
    <jsp:setProperty name="mensa" property="*" />
    <!--pasa la petición al servlet guardar
    para que grabe el mensaje en la BD-->
    <jsp:forward page="guardar"/>
</c:if>
<h1>Generación de mensajes</h1>
<form method="post">
    <br/><br/>
    Introduzca destinatario: <input type="text"
        name="destino"/><br/><br/>
    Introduzca remitente: <input type="text"
        name="remite"/><br/><br/>
    Introduzca texto: <br/>
    <textarea name="texto">
    </textarea>
    <br/>
    <center>
        <input type="submit" name="Submit"
            value="Enviar"/>
    </center>
</form>
</body>
</html>

```

### ver.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javabeans.Mensaje,java.util.*,
                    negocio Operaciones"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Página de mensajes</title>
    </head>
    <body>
        <c:set var="nombre" value="${param.nombre}"/>
        <!--obtiene los parámetros de contexto relativos
        a la conexión con la BD-->
        <c:set var="driver" value="${initParam['driver']}"/>
        <c:set var="cadenacon"
            value="${initParam['cadenacon']}"/>
        <!--instanciación de la clase Operaciones-->
        <jsp:useBean id="oper" class="negocio.Operaciones">
            <jsp:setProperty name="oper" property="driver"
                value="${driver}"/>
            <jsp:setProperty name="oper" property="cadenacon"
                value="${cadenacon}"/>
        </jsp:useBean>
        <h1> Mensajes para <c:out value="${nombre}"/></h1>
        <table border="1">
            <tr><th>Remitente</th><th>Mensaje</th></tr>
            <!--recorre los mensajes existentes y muestra los
            correspondientes al destinatario recibido-->
            <c:forEach var="mensaje" items="${oper.mensajes}">

```



```

        <c:if test="${mensaje.destino==nombre}">
            <tr><td><c:out value="${mensaje.remite}"/></td>
            <td><c:out value="${mensaje.texto}"/></td></tr>
        </c:if>
    </c:forEach>
</table>
<br/>
<a href="inicio.html">Inicio</a>
</body>
</html>

```

### nomensajes.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Página de aviso</title>
    </head>
    <body>
        <center>
            <h2>Lo siento, ${param.nombre} no tiene mensajes
            </h2>
        </center>
    </body>
</html>

```

## 5. CREACIÓN DE ACCIONES JSP PERSONALIZADAS

En la mayoría de las aplicaciones Web que vayamos a desarrollar, la sublibrería *core* de JSTL nos ofrecerá el soporte suficiente para crear páginas JSP libres de código Java.

No obstante, puede haber algunas situaciones en donde ciertos tipos de operaciones sólo puedan resolverse mediante el uso de scriptlets. Si este tipo de operaciones se realiza de forma habitual en las páginas, podemos optar por crear nuestro propio conjunto de acciones personalizadas para realizar estas operaciones, evitando así el uso de scriptlets en todas las situaciones.

La creación de una librería de acciones personalizadas requiere la realización de dos operaciones fundamentales:

- **Implementación de las clases manejadoras.** Toda acción JSP **tiene asociado un código Java** que se implementa en las llamadas clases manejadoras. Por cada tag o acción que queramos generar, habrá que crear una clase manejadora que implemente las operaciones a realizar por la acción. Seguidamente, analizaremos con detalle este proceso y las distintas situaciones que se pueden dar.
- **Creación del archivo de librería.** El archivo de librería **contiene la información** que el contenedor Web necesita para **poder interpretar las acciones** personalizadas que se van a utilizar, como la localización de las clases manejadoras, nombre de las acciones y atributos, etc. A través de la información suministrada en la directiva taglib el contenedor Web deberá ser capaz de localizar este archivo.

## 5.1. IMPLEMENTACIÓN DE LA CLASE MANEJADORA

Como acabamos de indicar, la clase manejadora es la que contiene el código Java asociado a la acción. La interfaz **javax.servlet.jsp.tagext.Tag** proporciona el soporte básico para la creación de una clase manejadora, definiendo los métodos mínimos que se han de implementar.

En vez de implementar esta interfaz se puede heredar la clase `javax.servlet.jsp.tagext.TagSupport` que proporciona una implementación por defecto de `Tag`.

### 5.1.1. Ciclo de vida básico de una acción

Los métodos de la interfaz `Tag` son ejecutados por el contenedor Web durante la vida de la acción, es decir, desde que el contenedor encuentra la acción durante la ejecución de la página hasta que se sale de ella. Estos métodos son:

- **void setPageContext(PageContext pc).** Es el primer método invocado por el contenedor y permite establecer el contexto de la página actual. Normalmente se utiliza la implementación por defecto que realiza `TagSupport` de este método, en la cual se asigna el objeto `PageContext` en un atributo protegido llamado `pageContext`. Como ya vimos en apartados anteriores, a través de los métodos de `PageContext` es posible obtener referencias a los distintos objetos integrados de JSP y poder tener acceso así a los datos manejados en la página.
- **void setParent (Tag parent).** Después de `setPageContext()` el contenedor llama a este método, pasándole como parámetro el objeto `Tag` asociado a la acción padre de ésta (*null* si no se encuentra anidada en otra acción).
- **int doStartTag().** Es invocado al encontrar la etiqueta de comienzo de la acción, justo inmediatamente después de `setPageContext()`. Este método devuelve un número entero que indicará al contenedor como debe comportarse con el cuerpo de la acción. Sus posibles valores son:
  - `Tag.EVAL_BODY_INCLUDE`. Esta constante indica que el cuerpo de la etiqueta debe ser evaluado por el contenedor.
  - `Tag.SKIP_BODY`. Indica que el cuerpo no debe ser evaluado.
- **int doEndTag().** Es invocado cuando se encuentra la etiqueta de cierre de la acción. El valor devuelto indica al contenedor como debe comportarse después de finalizar la acción:
  - **Tag.EVAL\_PAGE.** Indica que el contenedor debería evaluar el resto de la página.
  - **Tag.SKIP\_PAGE.** El resto de la página no debería ser evaluada, por lo que la petición se considera completada.

Los métodos anteriores constituyen el ciclo de vida básico de una acción. Para acciones que requieran realizar iteraciones sobre el cuerpo de la acción o manipular el contenido del mismo, deberíamos implementar `IterationTag` o `BodyTag`, respectivamente. Ambas son subinterfaces de `Tag` que incluyen métodos adicionales para realizar las operaciones indicadas; más adelante analizaremos estas interfaces.

### 5.1.2. Escritura en la página de respuesta

Una de las principales operaciones que se pueden llevar a cabo en una acción JSP es la inserción de datos en la página respuesta. Dicha respuesta está representada por un objeto de la clase `JspWriter`.

A través de `pageContext` se puede tener acceso al objeto `JspWriter` asociado a la página donde se encontrará la acción, utilizando su método `getOut()`.

El siguiente ejemplo representa una clase manejadora asociada a una acción simple (sin cuerpo), cuya única función consiste en mostrar un mensaje de saludo en la página de respuesta en el momento en que la acción sea evaluada. El código para realizar esta tarea será incluido en el método `doStartTag()`:

```
package ejemplo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class Saludo extends TagSupport{
    public int doStartTag() throws JspException{
        try{
            JspWriter out=pageContext.getOut();
            out.println("Bienvenido a mi acción");
        }
        catch(Exception e){e.printStackTrace();}

        //al ser una acción sin cuerpo, tanto SKIP_BODY
        //como EVAL_BODY_INCLUDE generarían el mismo resultado
        return SKIP_BODY;
    }
    public int doEndTag(){
        return EVAL_PAGE;
    }
}
```

## 5.2. CREACIÓN DE UN ARCHIVO DE LIBRERÍA

Un archivo de librería de acciones, conocido también como descriptor de librería, es un documento XML con extensión `.tld` donde se describe el conjunto de acciones personalizadas que queremos pertenezcan a la misma librería.

### 5.2.1. Etiquetas para la definición de una librería de acciones

Como sucede con otros documentos XML, un descriptor de librería comienza con una declaración de tipo que indica el vocabulario o estándar utilizado en la construcción del documento. En nuestro caso la declaración de tipo tendrá el siguiente aspecto:

```
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

A continuación de la declaración de tipo se indicará el elemento raíz `taglib`, dentro del cual se deberá incluir todas las etiquetas del documento; he aquí las más importantes:

- **tlib-version.** Número de versión de la librería

- **jsp-version.** Versión de la especificación JSP utilizada por la librería. En nuestro caso, será la versión 2.0.
- **short-name.** Representa un posible valor elegido para ser utilizado como prefijo en las etiquetas. Representa una sugerencia para ser utilizado como valor del atributo prefix en la directiva taglib.
- **uri.** Contiene la uri asociada a la librería. Si el archivo web.xml no contiene una asociación entre el valor especificado en el atributo uri de la directiva taglib y la dirección real del archivo .tld, el contenedor Web buscará una coincidencia entre el uri especificado en taglib y el valor indicado en este atributo de cada TLD.
- **description.** Texto descriptivo sobre la librería
- **tag.** Definición de una acción. Por cada acción de la librería se incluye una etiqueta tag con los siguientes subelementos.
  - **name.** Nombre de la acción.
  - **tag-class.** Nombre cualificado de la clase que define la acción
  - **tei-class.** Nombre cualificado de la clase TagExtraInfo (solo cuando se utilizan variables de scripting)
  - **body-content.** Información utilizada para determinar como se va a evaluar el cuerpo de la acción:
    - **JSP.** El cuerpo será evaluado en la página.
    - **tagdependent.** El cuerpo no será evaluado por el contenedor, sino que será interpretado por la clase manejadora.
    - **empty.** La acción no tiene cuerpo.
  - **description.** Breve descripción de la acción.
  - **attribute.** Define los atributos de la acción. Aunque su uso se verá más adelante, comentar que sus posibles subelementos son:
    - **name.** Nombre del atributo.
    - **required.** Indica si el atributo es o no obligatorio.
    - **rtexprvalue.** Indica si el atributo puede o no ser calculado dinámicamente utilizando una expresión.
    - **type.** Tipo de dato del atributo (nombre cualificado de la clase). Para literales es siempre String.

Si quisiéramos registrar una acción llamada saludo, asociada a la clase Saludo definida anteriormente, el archivo .tld quedaría:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
```

```

<jsp-version>1.2</jsp-version>
<short-name>ejemplo</short-name>
<uri>http://www.milibreria.tld</uri>
<description>Ejemplo de librería</description>
<tag>
  <name>saludo</name>
  <tag-class>ejemplo.Saludo</tag-class>
  <body-content>empty</body-content>
  <description>tag de ejemplo</description>
</tag>
</taglib>

```

### 5.3. UTILIZACIÓN DE ACCIONES PERSONALIZADAS EN UNA PÁGINA JSP

Antes de seguir profundizando en el desarrollo de las acciones personalizadas, vamos a ver como, una vez que la clase manejadora está implementada y se ha generado el archivo de librería, se puede hacer uso de la acción desde una página JSP.

A la hora de distribuir las clases manejadoras podemos optar por tratar con los .class o, preferiblemente, generar un archivo .jar en el que se incluyan todas estas clases. Tanto en un caso como en otro, las clases tendrán que ser accesibles desde la aplicación Web para lo cual debemos proceder de la siguiente manera según la forma en que se distribuyan las clases:

- Si las clases se distribuyen como .class, todos estos archivos deberán ser incluidos en algún paquete dentro del directorio WEB-INF\classes de la aplicación.
- Si se distribuyen como un archivo .jar, este deberá ser incluido en el directorio WEB-INF\lib de la aplicación.

En cuanto al archivo de librería .tld, **lo más práctico es incluirlo en el .jar junto con las clases manejadoras**, aunque podemos situarlo en cualquier subdirectorio de la aplicación (normalmente, dentro de WEB-INF).

Una vez realizadas las operaciones anteriores, para poder usar las acciones de la librería en cualquier página JSP de la aplicación simplemente tendremos que incluir una referencia a la librería a través de la directiva *taglib*.

*taglib* cuenta como sabemos con los siguientes atributos:

- **uri.** Dirección lógica del archivo .tld. Si dicho archivo se encuentra en el .jar de distribución en este atributo habrá que indicar el mismo valor que se utilizó para el elemento <uri> del archivo de librería. Si el .tld está en cualquier otro directorio de la aplicación entonces habrá que asociar la dirección física del .tld a la uri indicada en taglib, operación esta que se lleva a cabo en el archivo web.xml
- **prefix.** Prefijo utilizado para las etiquetas de la librería

Por ejemplo, para utilizar la acción saludo definida en la librería de ejemplo anterior sería:

```
<%@ taglib uri="http://www.milibreria.tld" prefix= "ejemplo"%>
```

Si el .tld estuviera en el archivo .jar no habría que hacer ninguna otra operación, pero si estuviera situado, por ejemplo, en la ubicación WEB-INF\librerias\ejemplolibreria.tld, sería necesario incluir la siguiente entrada en el archivo web.xml:

```

<web-app>
<!--otros elementos-->
<taglib>
    <taglib-uri>
        http://www.milibreria.tld
    </taglib-uri>
    <taglib-location>
        /WEB-INF/librerias/ejemplolibreria.tld
    </taglib-location>
</taglib>

```

Obsérvese la utilización de la barra "/" al principio de la dirección indicada en taglib-location. Esto indica que se trata de una dirección relativa a la aplicación Web. Si no se indicara la barra, la dirección se tomaría relativa al propio documento web.xml.

La siguiente página JSP representa un ejemplo de uso de la acción saludo:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.milibreria.tld"
    prefix= "ejemplo"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Página de saludo</title>
    </head>
    <body>
        <center>
            Mensaje de saludo: <br/>
            <h2><ejemplo:saludo/>
            </h2>
        </center>
    </body>
</html>

```

#### 5.4. ATRIBUTOS EN ACCIONES JSP

Dotar de atributos a una acción JSP resulta ser una tarea bastante sencilla. A nivel de clase manejadora, **debemos disponer de un método *setNombre\_atributo*** que recoja el valor asignado al atributo. Este método será llamado por el contenedor Web antes de invocar a *doStartTag()*.

Por ejemplo, si queremos incluir en nuestra etiqueta saludo un atributo llamado mensaje que permita al programador especificar el mensaje de bienvenida, el código de la clase manejadora quedaría como se indica a continuación:

```

package ejemplo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

```

```

public class Saludo extends TagSupport{
    private String mensaje;
    public void setMensaje(String mensaje){
        this.mensaje=mensaje;
    }
    public int doStartTag() throws JspException{
        try{
            JspWriter out=pageContext.getOut();
            out.println(mensaje);
        }
        catch(Exception e){e.printStackTrace();}
        return SKIP_BODY;
    }
    public int doEndTag(){
        return EVAL_PAGE;
    }
}

```

Como vemos, el texto es recogido por el método *setMensaje()* y es almacenado en un dato miembro privado de la clase. Posteriormente, *doStartTag()* hace uso de este valor para mostrarlo en la página de respuesta.

Por otro lado, habría que incluir en el archivo .tld la información relativa a este atributo:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>ejemplo</short-name>
    <uri>http://www.milibreria.tld</uri>
    <description>Ejemplo de librería</description>
    <tag>
        <name>saludo</name>
        <tag-class>ejemplo.Saludo</tag-class>
        <body-content>empty</body-content>
        <description>tag de ejemplo</description>
        <attribute>
            <name>mensaje</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
</taglib>

```

Como vemos, se capacita al atributo para que pueda admitir valores procedentes de una expresión.

La siguiente JSP hace uso de la acción saludo, suministrado como valor del atributo *mensaje* el contenido del parámetro "texto" recibido en la petición:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://www.milibreria.tld"
      prefix= "ejemplo"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
      "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Página de saludo</title>
  </head>
  <body>
    <center>
      Mensaje de saludo: <br/>
      <h2><ejemplo:saludo mensaje="\${param['texto']}" />
      </h2>
    </center>
  </body>
</html>

```

## 5.5. ITERACIÓN SOBRE EL CUERPO DE UNA ACCIÓN

El soporte para iterar sobre una acción lo proporciona la **interfaz IterationTag** que es una subinterfaz de Tag. La clase TagSupport implementa también esta interfaz, por lo que podemos seguir heredándola en la creación de clases manejadoras que incluyan iteración sobre el cuerpo de la acción.

El único método que añade esta nueva interfaz es *doAfterBody()*. Este método es invocado por el contenedor después de evaluar el cuerpo de la acción (hecho que ocurre si *doStartTag()* devuelve la constante EVAL\_BODY\_INCLUDE) y antes de llamar a *doEndTag()*.

Este método puede devolver dos posibles resultados al contenedor:

- **La constante Tag.SKIP\_BODY.** Indica que el cuerpo no debe ser evaluado de nuevo, por lo que se procederá a la llamada a *doEndTag()*.
- **La constante IterationTag.EVAL\_BODY\_AGAIN.** El cuerpo de la acción volverá a ser evaluado de nuevo y, tras ello, se volverá a ejecutar de nuevo *doAfterBody()*.

El siguiente listado corresponde a una acción que muestra en la página de respuesta el texto incluido en su cuerpo, tantas veces como se indique en su atributo repeticiones:

```

package manejadores;
import javax.servlet.jsp.tagext.*;
public class MensajeRepetitivo extends TagSupport{
    private int repeticiones;
    private int contador;
    public void setRepeticiones(int rep){
        this.repeticiones=rep;
    }
    public int doStartTag(){
        contador=1;
        return Tag.EVAL_BODY_INCLUDE;
    }
}

```



```

    }
    public int doAfterBody(){
        if(contador++==repeticiones){
            return Tag.SKIP_BODY;
        }
        else{
            return IterationTag.EVAL_BODY_AGAIN;
        }
    }
}

```

El archivo descriptor de librería para una acción de estas características no tiene nada de especial respecto a cualquier otra acción que cuente con atributos, salvo que el contenido del cuerpo será de tipo JSP en vez de empty:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>ejemplo</short-name>
  <uri>http://www.milibreria.tld</uri>
  <description>Ejemplo de librería</description>
  <tag>
    <name>mensajes</name>
    <tag-class>manejadores.MensajeRepetitivo</tag-class>
    <body-content>JSP</body-content>
    <description>tag de mensajes repetitivos</description>
    <attribute>
      <name>repeticiones</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>

```

Un ejemplo de uso de esta acción podría ser la siguiente página JSP donde se utiliza la acción mensajes para mostrar 10 veces en la página de respuesta el contenido del cuerpo de dicha acción:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib uri="/WEB-INF/descripcion.tld" prefix="m"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <title>Ejemplos de acciones</title>
  </head>

```

```

<body>
<h1>Repeticiones de mensaje</h1>
<m:mensajes repeticiones="10">
    Este mensaje se repite varias veces <br/>
</m:mensajes>
</body>
</html>

```

## 5.6. MANIPULACIÓN DEL CUERPO DE LA ACCIÓN

La interfaz `IterationTag` proporciona la capacidad de evaluar repetidamente el cuerpo de una acción, pero no ofrece posibilidad alguna de manipular el contenido de este en cada iteración.

Si queremos contar con esa posibilidad debemos recurrir a **la interfaz `BodyTag`**, subinterfaz de `IterationTag` que añade los siguientes métodos nuevos:

- **`void setBodyContent(BodyContent body)`**. Es invocado por el contenedor después de la ejecución de `doStartTag()`, siempre y cuando este método devuelva la nueva constante incluida en la interfaz `IterationTag`: `EVAL_BODY_BUFFERED`. Si lo que se devuelve es `EVAL_BODY_INCLUDE` entonces el contenedor evaluará directamente el cuerpo de la etiqueta sin ejecutar ninguno de los métodos definidos en `BodyTag`. El diagrama de flujo de la figura 17 nos ayudará a comprender el ciclo de vida de este tipo de acciones.

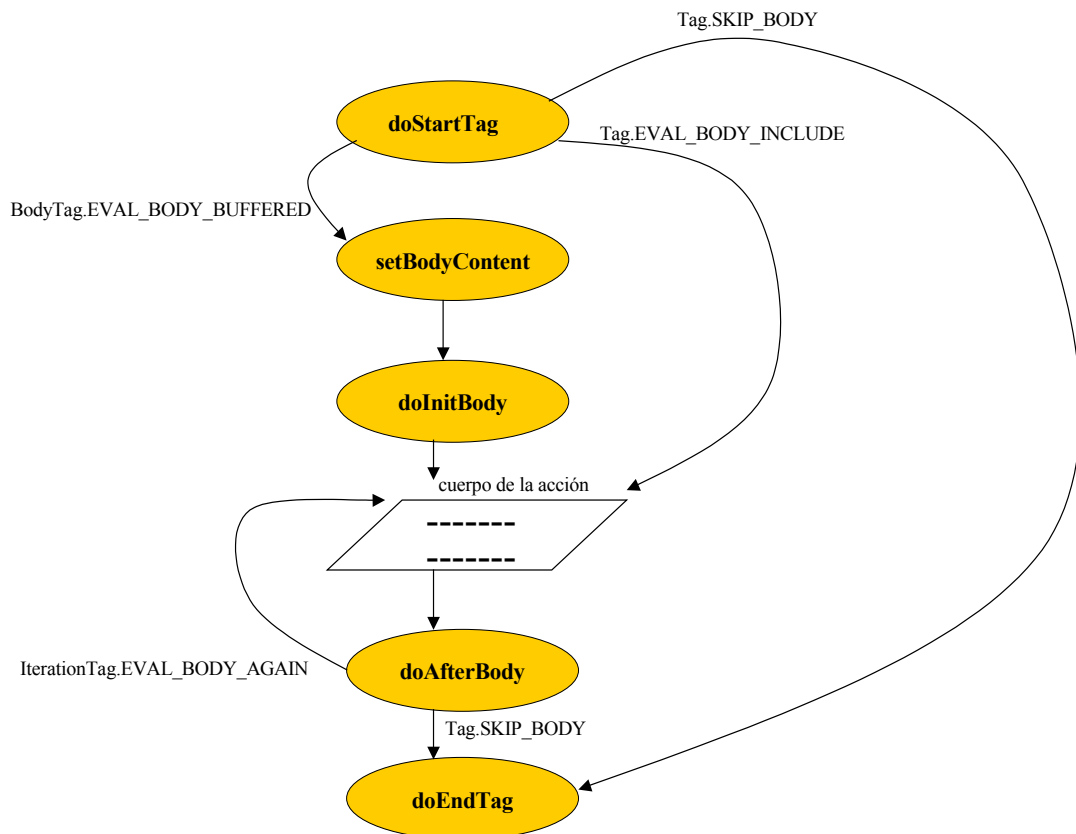


Figura. 17.

Como hemos indicado en el formato del método, `setBodyContent` recibe como parámetro un objeto `BodyContent`, subclase de `JspWriter` que representa un buffer donde se irá acumulando la salida generada por el cuerpo de la acción en cada

interacción. Disponiendo de este objeto podemos incluir contenido adicional en este buffer para, finalmente, enviar su contenido completo a la salida. Normalmente, el método `setBodyContent` únicamente realiza la operación de almacenamiento de este objeto en algún atributo interno de la clase.

En cuanto a los métodos que nos proporciona `BodyContent` para manejar el buffer, he aquí los más importantes:

- **`println(tipo dato)`**. Representa la familia de métodos `println()` heredados de `JspWriter` con los que se puede incluir cualquier contenido en el buffer.
- **`writeOut(Writer out)`**. Permite enviar el contenido del buffer al objeto `Writer` indicado.
- **`getEnclosingWriter()`**. Devuelve un objeto `JspWriter` que representa el entorno donde se encuentra la acción. Si la acción se encuentra dentro de otra, `getEnclosingWriter()` representa el objeto `JspWriter` asociado a la acción padre, si la acción está directamente sobre la página `getEnclosingWriter()` devolverá el `JspWriter` asociado a la salida.

Para enviar a la página o al interior de la acción padre el contenido acumulado hasta el momento se procedería de la siguiente manera:

```
body.writeOut (body.getEnclosingWriter() ) ;
```

Esta operación que se lleva a cabo normalmente en el método `doAfterBody()`, una vez que se han completado todas las iteraciones sobre el cuerpo de la acción.

- **`void doInitBody()`**. Es invocado después de `setBodyContent()`, antes de que el contenedor evalúe el cuerpo de la acción por primera vez. Su función es la de inicializar el objeto `BodyContent` con algún bloque de texto inicial y preparar el cuerpo para la primera evaluación.

Para aclarar el complejo funcionamiento de este tipo de acciones, vamos a realizar una nueva versión de la acción anterior mensajes en la que se va a realizar una manipulación del cuerpo de la acción en cada iteración que permita mostrar en negrita los mensajes impares (figura 18).

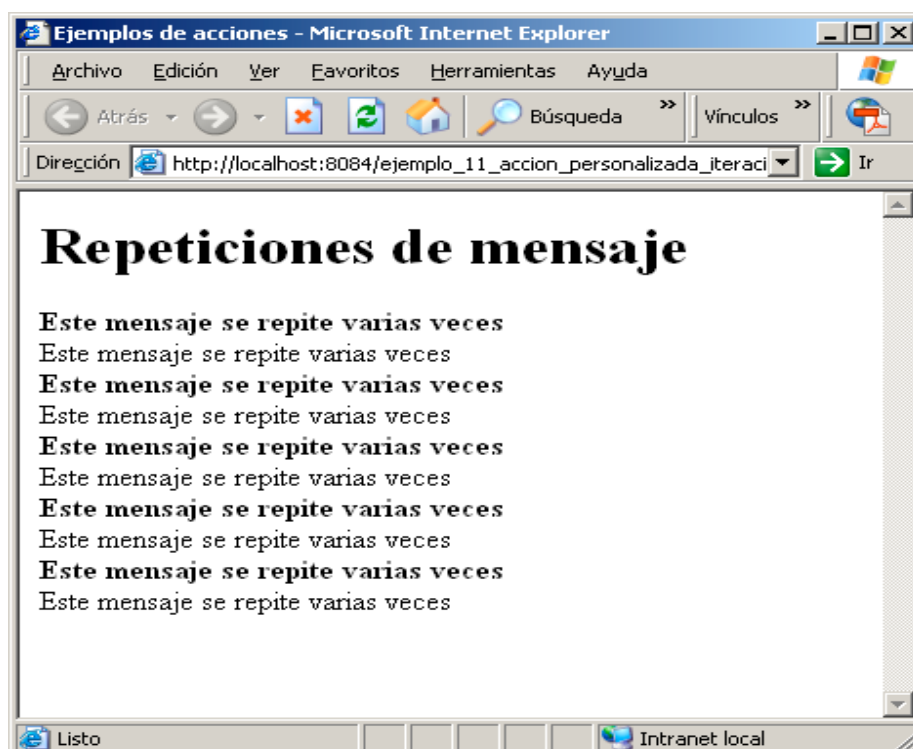


Figura. 18.

Para ello comprobaremos en cada ejecución de *doAfterBody()* el número de iteración actual a fin de incluir adecuadamente las etiquetas "<b>" y "</b>" en el objeto BodyContent. Mientras la iteración actual sea inferior al número de repeticiones se volverá a evaluar el cuerpo de la acción, pero cuando ambos valores se igualen se deberá enviar el contenido de BodyContent a la salida mediante el método *writeOut()*:

```
package manejadores;
import javax.servlet.jsp.tagext.*;
public class MensajeRepetitivo extends BodyTagSupport{
    private int repeticiones;
    private int contador;
    private BodyContent body;
    public void setRepeticiones(int rep){
        this.repeticiones=rep;
    }
    public int doStartTag(){
        contador=1;
        return BodyTag.EVAL_BODY_BUFFERED;
    }
    public void setBodyContent(BodyContent body){
        this.body=body;
    }
    public void doInitBody(){
        try{
            //inicializamos el buffer para que el primer
            //mensaje aparezca en negrita
            body.println("<b>");
        }
        catch(Exception e){e.printStackTrace();}
    }
    public int doAfterBody(){
        if(contador++==repeticiones){
            try{
                //se envía el buffer a la salida
                body.writeOut(body.getEnclosingWriter());
            }
            catch(Exception e){e.printStackTrace();}
            return Tag.SKIP_BODY;
        }
        else{
            try{
                if(contador%2==0){
                    body.println("</b>");
                }
                else{
                    body.println("<b>");
                }
            }
        }
    }
}
```

```
        }
        catch(Exception e){e.printStackTrace();}
        return IterationTag.EVAL_BODY_AGAIN;
    }
}
```

## EJERCICIO 7

Vamos a crear en este ejercicio una acción personalizada que genere una tabla HTML con los datos de los objetos de tipo JavaBean almacenados en una colección de tipo List, existente en alguno de los ámbitos de la aplicación.

Utilizaremos técnicas de reflexión para, además de invocar los métodos set de cada objeto, mostrar inicialmente la fila con los nombres de las propiedades.

Los atributos que tendrá la acción serán los siguientes:

- **claseobjeto.** Nombre cualificado de la clase JavaBean cuyos objetos son almacenados en la colección.
- **objcoleccion.** Nombre del atributo de ámbito donde se almacena la referencia a la colección List
- **ambito.** Ambito del atributo de colección. Su uso es opcional, utilizándose page como ámbito predeterminado.

He aquí el código de la clase manejadora:

```
package manejadores;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.*;
import java.util.*;
import java.lang.reflect.*;
import org.omg.IOP.TAG_ALTERNATE_IIOP_ADDRESS;
public class GeneradorTabla extends BodyTagSupport{
    private String objcoleccion;
    private String claseobjeto;
    private String ambito="page"; //ambito por defecto
    private BodyContent body;
    private Class tipoobjeto;
    private List coleccion;
    private int totalfilas;
    private int filasmostradas=0;
    public void setObjcoleccion(String objcoleccion){
        this.objcoleccion=objcoleccion;
    }
    public void setClaseobjeto(String claseobjeto){
        this.claseobjeto=claseobjeto;
    }
    private void setAmbito(String ambito){
        this.ambito=ambito;
    }
    public int doStartTag(){
        try{
            tipoobjeto=Class.forName(claseobjeto);
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
            return Tag.SKIP_PAGE;
        }
        //comprueba el ámbito de la colección
        if(ambito.equalsIgnoreCase("request")){
            coleccion=(List)pageContext.
```

```

        getAttribute(objcoleccion, PageContext.REQUEST_SCOPE);
    }
    else{
        if(ambito.equalsIgnoreCase("session")){
            coleccion=(List)pageContext.
getAttribute(objcoleccion, PageContext.SESSION_SCOPE);
        }
        else{
            if(ambito.equalsIgnoreCase("application")){
                coleccion=(List)pageContext.
getAttribute(objcoleccion, PageContext.APPLICATION_SCOPE);
            }
            else{
                coleccion=(List)pageContext.
getAttribute(objcoleccion, PageContext.PAGE_SCOPE);
            }
        }
    }
    //si no existe la colección no continua con la acción
    if(coleccion==null){
        return Tag.SKIP_BODY;
    }
    else{
        //procede a generar el contenido en un buffer
        return BodyTag.EVAL_BODY_BUFFERED;
    }
}

public void setBodyContent(BodyContent body){
    this.body=body;
}

public void doInitBody(){
    totalfilas=coleccion.size();
    try{
        //inicializamos el buffer para que
        //aparezca la fila de nombres
        body.println("<table border='1'>");
        body.println("<tr>");
        Method [] metodos =tipoobjeto.getDeclaredMethods();
        for(Method m:metodos){
            if(m.getName().indexOf("get")==0){
                body.println("<th>");
                body.println(m.getName().substring(3));
                body.println("</th>");
            }
        }
        body.println("</tr>");
    }
    catch(Exception e){e.printStackTrace();}
}

public int doAfterBody(){
    try{
        //si hay más filas, genera los contenidos
        if(filasmostradas<totalfilas){
            Object obj=coleccion.get(filasmostradas);
            Method [] metodos =
                tipoobjeto.getDeclaredMethods();
            body.println("<tr>");
            for(Method m:metodos){
                if(m.getName().indexOf("get")==0){
                    body.println("<td>");
                    body.println(m.invoke(obj));
                }
            }
        }
    }
    catch(Exception e){e.printStackTrace();}
}

```

```

        body.println("</td>");
    }
}
body.println("</tr>");
filasmostradas++;
return IterationTag.EVAL_BODY_AGAIN;
}
else{
    body.writeOut(body.getEnclosingWriter());
    return Tag.SKIP_BODY;
}
}
catch(Exception e){e.printStackTrace();}
//entrará aquí si hay algún error
return Tag.SKIP_BODY;
}

public int doEndTag(){
    //cierra la tabla y sigue evaluando la página
    try{
        body.println("</table>");
    }
    catch(Exception e){e.printStackTrace();}
    return Tag.EVAL_PAGE;
}
}

```

En cuanto al archivo descriptor de librería:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN" "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>ejemplo</short-name>
  <uri>http://www.milibreria.tld</uri>
  <description>Ejemplo de librería</description>
  <tag>
    <name>tablados</name>
    <tag-class>manejadores.GeneradorTabla</tag-class>
    <body-content>JSP</body-content>
    <description>tag para tabla de datos en colección
    </description>
    <attribute>
      <name>ambito</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>claseobjeto</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
      <name>objcoleccion</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>

```



```
        </attribute>
    </tag>
</taglib>
```

Lo siguiente representa una página JSP de ejemplo de uso de la acción:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib uri="/WEB-INF/descripcion.tld" prefix="m"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <title>Tabla de datos</title>
    </head>
    <body>
        <h1>Tabla de datos</h1>
        <m:tablados claseobjeto="javabeans.Persona"
            objcoleccion="lista" ambito="request">

        </m:tablados>

    </body>
</html>
```

## AUTOEVALUACIÓN

---

1. Una declaración JSP se utiliza para:
  - A. Declarar cualquier tipo de variable existente en un scriptlet
  - B. Declarar una variable fuera del método `_jspService()`
  - C. Declarar directivas JSP
  - D. Declarar acciones JSP
  
2. Para importar la clase `ArrayList` del paquete `java.util` en una página JSP deberíamos utilizar:
  - A. `<%@ page import = "java.util.ArrayList"%>`
  - B. `<%@ import class = "java.util.ArrayList"%>`
  - C. `<%@ import package = "java.util.*"%>`
  - D. `<%@ ! import = "java.util.ArrayList"%>`
  
3. Tenemos una variable de script llamada `dato` cuyo valor queremos incluir en la página de respuesta. Indica cual de las siguientes operaciones habría que utilizar para llevar a cabo esa tarea:
  - A. `<%=dato;%>`
  - B. `<%dato%>`
  - C. `<%out.println("dato");%>`
  - D. `<%=out.println("dato")%>`
  
4. ¿Cual de los siguientes métodos no puede ser sobrescrito mediante el código de script de servidor incluido en una página JSP?:
  - A. El método `jspInt()`
  - B. El método `jspDestroy()`
  - C. El método `_jspService()`
  - D. Ninguno de los tres métodos anteriores pueden ser sobrescrito.

5. Si queremos recuperar mediante una expresión EL el valor de la propiedad "titulo" de un javabean perteneciente a una clase Libro, que está almacenado como primer elemento de una colección ArrayList de identificador "libros" con un ámbito de aplicación, tendríamos que utilizar:

A. `&{libros[0].titulo}`  
B. `&{libros.Libro.titulo}`  
C. `&{libros["Libro"].titulo}`  
D. `&{libros[0].Libro.titulo}`

6. ¿Cual de los siguientes criterios es obligatorio que cumpla una clase para poder ser instanciada mediante la acción useBean?:

A. La clase debe implementar la interfaz Serializable  
B. Los atributos de la clase deben ser privados  
C. La clase debe disponer de una pareja de métodos set/get por cada uno de los atributos definidos en la clase.  
D. La clase debe disponer de un constructor sin parámetros.

7. La siguiente expresión EL incluida en una página JSP:

`${param.user}`

Producirá el mismo resultado que:

A. `${request.parameters("user")}`  
B. `<c:out property="user" scope="request"/>`  
C. `<%out.println(parameters["user"]);%>`  
D. `<%=request.getParameter("user")%>`

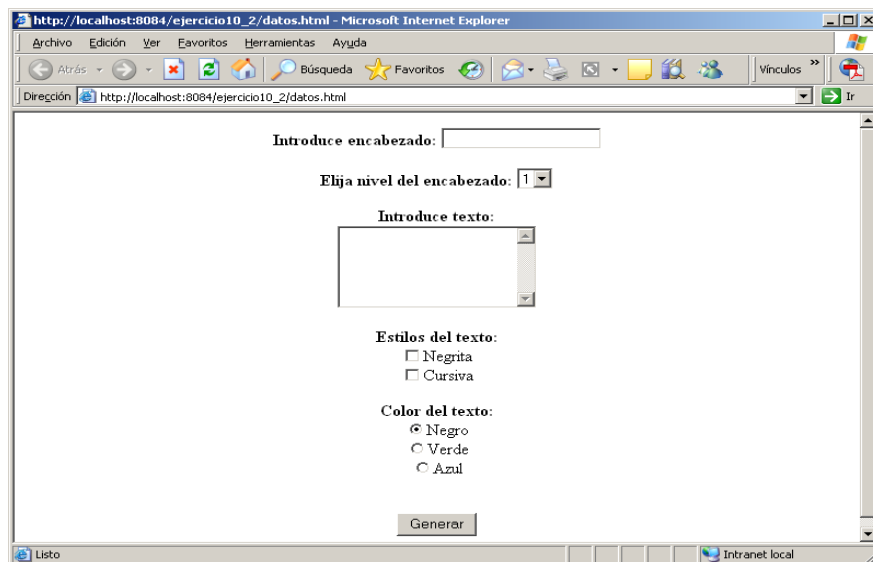
8. Estamos desarrollando una clase manejadora de una acción personalizada. Dado que queremos iterar sobre el cuerpo de la acción, la clase en cuestión implementa la interfaz IterationTag. Si, tras la ejecución del método doEndTag() de nuestra clase manejadora, quisiéramos que se volviera a evaluar el cuerpo de la acción deberíamos incluir como última instrucción del método:

A. `return EVAL_BODY_BUFFERED;`  
B. `return EVAL_BODY_AGAIN;`  
C. `return EVAL_BODY_INCLUDE;`  
D. No es posible volver a evaluar el cuerpo de la acción después de ejecutarse doEndTag().

## Ejercicios Propuestos

---

1. Crear una página JSP que muestre seis veces un mismo mensaje con diferentes niveles de encabezado, desde <h1> hasta <h6>.
2. Crear una página que cada vez que se entre en ella nos muestre el número de veces que ha sido visitada por todos los usuarios de la aplicación, desde que se produjo el inicio de esta. Además, en caso de que el usuario haya visitado con anterioridad esta página (en la misma sesión u otra), se le mostrará la fecha y hora en la que realizó la última visita. Si es la primera vez que visita la página no se indicará nada.
3. Crear una nueva versión del ejercicio 2 resuelto en el tema anterior, consistente en generar dinámicamente una página formateada según las opciones indicadas en una página HTML:



The screenshot shows a Microsoft Internet Explorer window with the address bar displaying 'http://localhost:8084/ejercicio10\_2/datos.html'. The page contains a form with the following elements:

- A text input field labeled 'Introduce encabezado:'.
- A dropdown menu labeled 'Elija nivel del encabezado:' with '1' selected.
- A text input field labeled 'Introduce texto:'.
- A section titled 'Estilos del texto:' containing two checkboxes: 'Negrita' and 'Cursiva'.
- A section titled 'Color del texto:' containing three radio buttons: 'Negro' (selected), 'Verde', and 'Azul'.
- A 'Generar' button at the bottom.

The status bar at the bottom indicates 'Listo' and 'Intranet local'.

**Figura. 19.**

Al pulsar el botón generar los datos serán enviados a una página JSP que será la encargada de generar dinámicamente la nueva página según las opciones elegidas.

4. Crear una nueva versión de la aplicación para la realización de votaciones a través de Internet que se indicó en la lista de ejercicios propuestos del tema anterior. En esta nueva versión el resultado de las votaciones será generado con una página JSP.
5. Tenemos una base de datos para el almacenamiento de libros con la siguiente estructura:

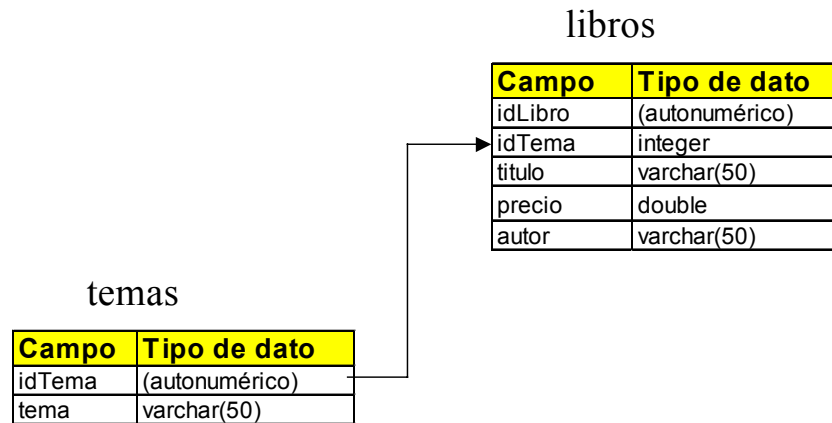


Figura. 20.

Se pide realizar una aplicación Web que cuente con una página inicial que presente dos opciones al usuario mediante sendos link:

- “Añadir libro”. Aparecerá una página que solicitará los datos del nuevo libro. Para el tema se mostrará una lista desplegable con todos los nombres de todos los temas existentes (recordar que en la tabla de libros se ha de grabar el identificador de tema)
- “Ver libros”. Mostrará una página con los datos de todos los libros existentes en la base de datos

Para desarrollar esta aplicación se utilizarán tanto servlets como páginas JSP, según la funcionalidad a implementar.

Para las páginas JSP se utilizará el lenguaje EL y la librería de acciones JSTL, a fin de evitar el uso de código Java en las mismas.

Se deberá además aislar todo el código de acceso a datos en clases y se utilizarán los JavaBean que se consideren necesarios.

6. Diseñar una acción JSP personalizada que genere una tabla HTML con los títulos, precios y autores de la base de datos utilizada en el ejercicio anterior.