

Texto que muestra la utilización de las tecnologías de servidor más destacables de Java, como son los servlets y las páginas JSP (JavaServer Pages).

El objetivo principal del texto es adiestrar a los lectores en el desarrollo de aplicaciones Web basadas en servlets y páginas JSP. También se trata el modelo de componentes JavaBeans. Como contenedor de servlets/páginas JSP se utiliza Jakarta Tomcat.

Los requisitos necesarios para seguir el texto de manera satisfactoria es conocer el lenguaje Java y los fundamentos de Internet/Intranet.



# TECNOLOGÍAS DE SERVIDOR CON JAVA: SERVLETS, JAVABEANS, JSP

ÁNGEL ESTEBAN





## ADVERTENCIA LEGAL

Todos los derechos de esta obra están reservados a Grupo EIDOS Consultoría y Documentación Informática, S.L.

El editor prohíbe cualquier tipo de fijación, reproducción, transformación, distribución, ya sea mediante venta y/o alquiler y/o préstamo y/o cualquier otra forma de cesión de uso, y/o comunicación pública de la misma, total o parcialmente, por cualquier sistema o en cualquier soporte, ya sea por fotocopia, medio mecánico o electrónico, incluido el tratamiento informático de la misma, en cualquier lugar del universo.

El almacenamiento o archivo de esta obra en un ordenador diferente al inicial está expresamente prohibido, así como cualquier otra forma de descarga (downloading), transmisión o puesta a disposición (aún en sistema streaming).

La vulneración de cualesquiera de estos derechos podrá ser considerada como una actividad penal tipificada en los artículos 270 y siguientes del Código Penal.

La protección de esta obra se extiende al universo, de acuerdo con las leyes y convenios internacionales.

Esta obra está destinada exclusivamente para el uso particular del usuario, quedando expresamente prohibido su uso profesional en empresas, centros docentes o cualquier otro, incluyendo a sus empleados de cualquier tipo, colaboradores y/o alumnos.

Si Vd. desea autorización para el uso profesional, puede obtenerla enviando un e-mail [fmarin@eidos.es](mailto:fmarin@eidos.es) o al fax (34)-91-5017824.

Si piensa o tiene alguna duda sobre la legalidad de la autorización de la obra, o que la misma ha llegado hasta Vd. vulnerando lo anterior, le agradeceremos que nos lo comunique al e-mail [fmarin@eidos.es](mailto:fmarin@eidos.es) o al fax (34)-91-5017824). Esta comunicación será absolutamente confidencial.

Colabore contra el fraude. Si usted piensa que esta obra le ha sido de utilidad, pero no se han abonado los derechos correspondientes, no podremos hacer más obras como ésta.

© Ángel Esteban, 2000

© Grupo EIDOS Consultoría y Documentación Informática, S.L., 2000

ISBN 84-88457-21-9

## Tecnologías de servidor con Java

Ángel Esteban

### Responsable editorial

Paco Marín ([fmarin@eidos.es](mailto:fmarin@eidos.es))

### Coordinación de la edición

Antonio Quirós ([aquiros@eidos.es](mailto:aquiros@eidos.es))

### Autoedición

Magdalena Marín ([mmarin@eidos.es](mailto:mmarin@eidos.es))

Ángel Esteban ([aesteban@eidos.es](mailto:aesteban@eidos.es))

### Grupo EIDOS

C/ Téllez 30 Oficina 2

28007-Madrid (España)

Tel: 91 5013234 Fax: 91 (34) 5017824

[www.grupoeidos.com/www.eidos.es](http://www.grupoeidos.com/www.eidos.es)

[www.LaLibreriaDigital.com](http://www.LaLibreriaDigital.com)



# Índice

<b>ÍNDICE.....</b>	<b>5</b>
<b>INTRODUCCIÓN: SERVLETS, JSP Y JAVABEANS.....</b>	<b>11</b>
INTRODUCCIÓN .....	11
VERSIONES DE LAS ESPECIFICACIONES DE SERVLETS Y JAVASERVER PAGES.....	13
LA PLATAFORMA JAVA 2 .....	13
SERVLETS.....	16
JAVASERVER PAGES (JSP).....	17
JAVABEANS .....	18
J2EE (JAVA 2 ENTERPRISE EDITION) .....	20
ENTERPRISE JAVABEANS.....	21
<b>INTRODUCCIÓN A LOS SERVLETS .....</b>	<b>23</b>
INTRODUCCIÓN .....	23
DEFINICIÓN DE SERVLET.....	23
COMPARACIÓN ENTRE LOS SERVLETS Y LOS SCRIPTS CGI.....	25
PREPARANDO EL ENTORNO.....	26
ESTRUCTURA BÁSICA DE UN SERVLET.....	30
LOS PAQUETES DE LA ESPECIFICACIÓN JAVA SERVLET 2.2 .....	31
HTML, HTTP Y LOS SERVLETS.....	34
EL SERVLET HOLA MUNDO .....	36
<b>SERVLETS: CONCEPTOS BÁSICOS.....</b>	<b>41</b>
INTRODUCCIÓN .....	41
LA CLASE HTTPServlet .....	42

LA CLASE <code>GenericServlet</code> .....	43
EL CICLO DE VIDA DE UN <code>Servlet</code> .....	45
APLICACIONES WEB .....	48
PARÁMETROS DE INICIALIZACIÓN .....	51
ORGANIZACIÓN DE <code>Servlets</code> .....	57
<b>SERVLETS: EL INTERFAZ <code>HttpServletRequest</code>.....</b>	<b>61</b>
INTRODUCCIÓN .....	61
EL INTERFAZ <code>HttpServletRequest</code> .....	61
CABECERAS DE PETICIÓN DEL PROTOCOLO HTTP .....	64
LEYENDO LAS CABECERAS DE PETICIÓN DESDE LOS <code>Servlets</code> .....	67
EL INTERFAZ <code>ServletRequest</code> .....	71
VARIABLES CGI.....	72
FORMULARIOS Y <code>Servlets</code> .....	76
<b>SERVLETS: EL INTERFAZ <code>HttpServletResponse</code>.....</b>	<b>85</b>
INTRODUCCIÓN .....	85
EL INTERFAZ <code>HttpServletResponse</code> .....	85
EL INTERFAZ <code>ServletResponse</code> .....	87
CABECERAS DE RESPUESTA HTTP .....	88
ENVIANDO INFORMACIÓN AL CLIENTE .....	89
CÓDIGOS DE ESTADO .....	96
<b>SERVLETS: LA CLASE <code>Cookie</code> Y EL INTERFAZ <code>RequestDispatcher</code>.....</b>	<b>103</b>
INTRODUCCIÓN .....	103
DEFINICIÓN DE <code>COOKIES</code> .....	104
LA CLASE <code>Cookie</code> .....	107
CREACIÓN DE <code>COOKIES</code> .....	108
UTILIZANDO LAS <code>COOKIES</code> .....	112
EL INTERFAZ <code>RequestDispatcher</code> .....	118
<b>SERVLETS: EL INTERFAZ <code>HttpSession</code> Y <code>ServletContext</code> .....</b>	<b>127</b>
INTRODUCCIÓN .....	127
SOLUCIONES TRADICIONALES.....	128
EL INTERFAZ <code>HttpSession</code> .....	129
CREACIÓN DE SESIONES.....	130
ALMACENANDO Y RECUPERANDO OBJETOS DE LA SESIÓN.....	132
EL INTERFAZ <code>ServletContext</code> .....	138
ALMACENANDO Y RECUPERANDO OBJETOS DE LA APLICACIÓN .....	140
CARACTERÍSTICAS DEL CONTENEDOR DE <code>Servlets</code> .....	144
<b>INTRODUCCIÓN A JSP (JAVASERVER PAGES).....</b>	<b>147</b>
INTRODUCCIÓN .....	147
DEFINICIÓN DE PÁGINA JSP .....	148
BENEFICIOS DE LAS PÁGINAS JSP .....	149
ELEMENTOS DE LAS PÁGINAS JSP .....	151
HOLA MUNDO CON JSP .....	152
Ejecución de las páginas JSP .....	156
EL API <code>JavaServer Pages 1.1</code> .....	162
<b>PÁGINAS JSP: DIRECTIVAS .....</b>	<b>165</b>
INTRODUCCIÓN .....	165
DIRECTIVAS .....	166
DIRECTIVA <code>page</code> .....	167
<i>Atributo info</i> .....	169

<i>Atributo language</i> .....	171
<i>Atributo contentType</i> .....	172
<i>Atributo extends</i> .....	172
<i>Atributo import</i> .....	173
<i>Atributo session</i> .....	174
<i>Atributo buffer</i> .....	176
<i>Atributo autoFlush</i> .....	177
<i>Atributo isThreadSafe</i> .....	178
<i>Atributo errorPage</i> .....	180
<i>Atributo isErrorPage</i> .....	180
DIRECTIVA INCLUDE .....	182
LA DIRECTIVA TAGLIB .....	188
<b>PÁGINAS JSP: ELEMENTOS DE SCRIPTING .....</b>	<b>191</b>
INTRODUCCIÓN .....	191
DECLARACIONES .....	192
SCRIPTLETS .....	200
EXPRESIONES .....	205
COMENTARIOS .....	209
<b>PÁGINAS JSP: OBJETOS INTEGRADOS I.....</b>	<b>211</b>
INTRODUCCIÓN .....	211
OBJETOS INTEGRADOS .....	212
<i>request</i> .....	212
<i>response</i> .....	212
<i>pageContext</i> .....	213
<i>session</i> .....	213
<i>application</i> .....	214
<i>out</i> .....	214
<i>config</i> .....	214
<i>page</i> .....	214
<i>exception</i> .....	215
EL OBJETO REQUEST .....	219
EL OBJETO RESPONSE .....	225
EL OBJETO OUT .....	228
EL OBJETO EXCEPTION .....	230
<b>PÁGINAS JSP: OBJETOS INTEGRADOS II.....</b>	<b>233</b>
INTRODUCCIÓN .....	233
EL OBJETO SESSION .....	233
EL OBJETO APPLICATION .....	238
EL OBJETO PAGECONTEXT .....	246
OBJETO PAGE .....	251
EL OBJETO CONFIG .....	251
<b>PÁGINAS JSP: ACCIONES .....</b>	<b>253</b>
INTRODUCCIÓN .....	253
LA ACCIÓN <JSP:FORWARD> .....	254
LA ACCIÓN <JSP:PARAM> .....	257
LA ACCIÓN <JSP:INCLUDE> .....	258
LA ACCIÓN <JSP:PLUGIN> .....	261
LA ACCIÓN <JSP:USEBEAN> .....	266
LA ACCIÓN <JSP:GETPROPERTY> .....	270
LA ACCIÓN <JSP:SETPROPERTY> .....	272
<b>JSP Y COMPONENTES JAVABEANS.....</b>	<b>277</b>

INTRODUCCIÓN .....	277
COMPONENTES.....	278
¿QUÉ ES UN JAVABEAN/BEAN/BEAN DE JAVA? .....	278
FUNDAMENTOS DE JAVABEANS .....	280
<i>Contenedores de Beans</i> .....	280
<i>Las propiedades de los Beans</i> .....	281
UTILIZACIÓN DE LOS COMPONENTES JAVABEANS .....	283
<b>DESARROLLO DE COMPONENTES JAVABEANS.....</b>	<b>293</b>
INTRODUCCIÓN .....	293
LAS CLASES DE LOS BEANS .....	293
LOS CONSTRUCTORES DE LOS BEANS .....	294
DEFINIENDO LAS PROPIEDADES .....	295
<i>Propiedades indexadas</i> .....	299
<i>Propiedades booleanas</i> .....	305
<i>Conversión de tipos</i> .....	306
EL INTERFAZ HTTPSESSIONBINDINGLISTENER .....	308
<b>TRATAMIENTO DE ERRORES EN JSP.....</b>	<b>311</b>
INTRODUCCIÓN .....	311
TIPOS DE ERRORES Y EXCEPCIONES .....	311
PÁGINAS JSP DE ERROR.....	314
EL OBJETO EXCEPTION .....	317
EJEMPLO DE TRATAMIENTO DE ERRORES .....	320
TRATAMIENTO DE ERRORES EN LOS SERVLETS .....	324
<b>ACCESO A DATOS DESDE JSP I .....</b>	<b>327</b>
INTRODUCCIÓN .....	327
INTRODUCCIÓN A JDBC .....	328
EL PAQUETE JAVA.SQL.....	328
DRIVERS JDBC .....	330
URLS DE JDBC.....	331
REALIZANDO UNA CONEXIÓN CON LA BASE DE DATOS .....	332
TRATANDO LAS EXCEPCIONES .....	335
CREACIÓN Y EJECUCIÓN DE SENTENCIAS SENCILLAS .....	339
TRATANDO LOS DATOS, EL INTERFAZ RESULTSET .....	344
<b>ACCESO A DATOS DESDE JSP II.....</b>	<b>351</b>
INTRODUCCIÓN .....	351
EL INTERFAZ PREPAREDSTATEMENT.....	351
EL INTERFAZ CALLABLESTATEMENT .....	355
EL INTERFAZ DATABASEMETADATA .....	359
EL INTERFAZ RESULTSETMETADATA .....	362
LOS COMPONENTES JAVABEANS Y JDBC .....	365
<b>ETIQUETAS PERSONALIZADAS .....</b>	<b>371</b>
INTRODUCCIÓN .....	371
LA DIRECTIVA TAGLIB .....	372
DESCRIPTORES DE LIBRERÍA DE ETIQUETAS .....	374
MANEJADORES DE ETIQUETAS.....	378
CREACIÓN DE ETIQUETAS PERSONALIZADAS .....	381
<b>JAKARTA TOMCAT .....</b>	<b>387</b>
INTRODUCCIÓN .....	387
INSTALACIÓN DE JAKARTA TOMCAT .....	387



ESTRUCTURA DE DIRECTORIOS DE JAKARTA TOMCAT .....	390
FICHEROS DE CONFIGURACIÓN .....	391
<i>Fichero SERVER.XML</i> .....	391
<i>Fichero WEB.XML</i> .....	395
ESTRUCTURA DE DIRECTORIOS DE UNA APLICACIÓN WEB .....	400
<b>JSP/ASP, UNA COMPARATIVA .....</b>	<b>403</b>
INTRODUCCIÓN .....	403
OBJETO APPLICATION .....	404
OBJETO CONFIG.....	405
OBJETO EXCEPTION.....	405
OBJETO OUT .....	406
OBJETO PAGE .....	407
OBJETO PAGECONTEXT .....	407
OBJETO REQUEST .....	407
OBJETO RESPONSE .....	408
OBJETO SERVER DE ASP.....	409
EL OBJETO SESSION.....	409
<b>BIBLIOGRAFÍA .....</b>	<b>411</b>



# 1

## Introducción: Servlets, JSP y JavaBeans

---

### Introducción

En este capítulo se va a realizar una breve introducción a las principales tecnologías de servidor ofrecidas por Sun Microsystems, y que trataremos a lo largo del presente texto. También se comentarán algunos aspectos relacionados con estas tecnologías así como la distribución, requisitos, premisas y objetivos del texto.

Podemos dividir este texto en dos grandes bloques, aunque como veremos más adelante se encuentran muy relacionados entre sí. Existe una primera parte dedicada a los servlets y una segunda parte dedicada a las páginas Java se servidor, o como normalmente se denominan JavaServer Pages (JSP).

Es fundamental entender la primera parte del texto para poder aprovechar las ventajas que nos ofrece la segunda, es decir, antes de aprender a desarrollar aplicaciones basadas en JavaServer Pages (JSP), se muy útil y necesario conocer la tecnología sobre la que se sustentan, que no es otra que los Servlets de Java.

¿Y los JavaBeans?, no nos hemos olvidado de ellos, a lo largo de los distintos capítulos veremos como tanto las páginas JSP como los servlets utilizan estos componentes denominados JavaBeans, Beans o Beans de Java. Incluso dedicaremos un capítulo monográfico a este modelo de componentes que ofrece Java.

El objetivo del texto es el de mostrar a los lectores como desarrollar aplicaciones Web para el entorno de Internet e Intranet basadas en servlets y páginas JSP.

Para aprovechar de forma correcta toda la información de este texto, el lector deberá conocer el lenguaje de programación Java, así como el acceso a datos que propone Java a través de su API de

acceso a datos denominado JDBC, aunque se realizará un pequeño repaso a los distintos interfaces y clases que nos ofrece JDBC. También es necesario que el lector se encuentre familiarizado con el entorno de Internet.

Para los lectores ya conocedores o familiarizados con el desarrollo de aplicaciones Web, podemos adelantar que los servlets es la respuesta (o versión) de Sun a los tradicionales scripts CGI (Common Gateway Interface), y las páginas JSP es la versión que ha realizado Sun de las páginas ASP (Active Server Pages) de Microsoft.

Existe una gran similitud entre JSP (JavaServer Pages) y ASP (Active Server Pages), aunque esta similitud es sobretudo a primera vista, ya que si vemos lo que hay detrás de cada tecnología comprobaremos que esta similitud es aparente.

JSP se ha planteado como un fuerte competidor o alternativa a ASP, tanto es así, que se ha dedicado un capítulo en el que se comentan las similitudes y diferencias existentes entre ambas tecnologías de servidor, e incluso se ofrece una pequeña guía dirigida a los desarrolladores de aplicaciones ASP en la que se realizan una serie de equivalencias en lo que a objetos integrados (implícitos) se refiere.

También veremos que tanto los servlets como las páginas JSP se encuentran incluidos en forma de APIs (a lo largo del texto el acrónimo API se va a referir a un conjunto de clases e interfaces relacionados y que representan una tecnología determinada) dentro de lo que se denomina la plataforma Java 2 Enterprise Edition (edición empresarial de Java 2). Este es el marco oficial que ofrece Sun para el desarrollo empresarial de aplicaciones, y se suele denominar mediante las iniciales J2EE (Java 2 Enterprise Edition).

La plataforma J2EE se encuentra en parte formada por diversos elementos clasificados como componentes de aplicación, contenedores, drivers gestores de recursos y bases de datos. Tanto servlets como páginas JSP entran dentro de la categoría de componentes de aplicación.

Ahora ya sabemos que los servlets y las páginas JSP no son dos tecnologías aisladas pertenecientes al lenguaje Java. Adelantamos que los servlets se definen dentro del paquete `javax.servlet` y las páginas JSP en el paquete `javax.servlet.jsp`. A la vista de los paquetes que contienen el API de los servlets y páginas JSP podemos comprobar que la relación entre ambos no es meramente casual.

Para seguir de forma satisfactoria este texto se requiere el siguiente software mínimo:

- La plataforma Java 2 SDK Standard Edition (JDK 1.3), como implementación del lenguaje Java y herramienta de desarrollo del mismo.
- Apache Jakarta Tomcat 3.1, como servidor Web y motor (engine) o contenedor de servlets y páginas JSP.

Los ejemplos se han probado y realizado utilizando como sistema operativo Windows 98 y el software señalado anteriormente.

Además para el desarrollo de páginas JSP se puede utilizar una herramienta bastante útil llamada JRun Studio de Allaire, sin embargo este software no es indispensable para el seguimiento adecuado del texto, y se puede conseguir una versión de evaluación en la dirección <http://www.allaire.com>.

De todas formas en los capítulos correspondientes volveremos a recordar el software necesario y veremos en detalle su utilización y configuración, así como conseguir dicho software.

## Versiones de las especificaciones de servlets y JavaServer Pages

En el presente texto vamos a tratar las últimas versiones de ambas tecnologías que se corresponden con el API servlet 2.2 y con el API JSP 1.1, en cada caso.

Ambas especificaciones, Java servlet 2.2 y JavaServer Pages 1.1, se encuentran implementadas en el servidor Web Jakarta Tomcat 3.1 de Apache, y forman parte de la plataforma Java 2 Enterprise Edition (J2EE).

Existen otros servidores Web y motores que implementan las especificaciones servlet 2.2 y JSP 1.1, aunque el más extendido, que es reconocido como la implementación oficial, es el servidor Web y a la vez motor de servlets y páginas JSP de Apache llamado Jakarta Tomcat 3.1.

## La plataforma Java 2

La última versión del lenguaje Java es lo que se denomina Plataforma Java 2 (Java 2 Platform) que se corresponde con las dos últimas versiones de la herramienta de desarrollo de Sun Microsystems, es decir, con el JDK (Java Development Kit) 1.2 y el JDK 1.3, este último de reciente aparición. Anteriormente había una correspondencia entre la versión del lenguaje Java y la versión de la herramienta JDK, así la versión 1.1 de Java se correspondía con la versión 1.1 del JDK.

El lenguaje Java todavía no ha tenido una versión definitiva ni estable, cosa que supone un grave inconveniente para los sufridos desarrolladores, que tenemos que estar actualizándonos continuamente. Según se ha comunicado desde Sun Microsystems, la versión Java 2 Platform es la versión definitiva del lenguaje, esperemos que esto sea cierto, ya que desde mediados / finales del año 1996 se han sucedido un gran número de versiones, y el seguimiento y aprendizaje del lenguaje resulta verdaderamente agotador y desconcertante.

La herramienta de desarrollo oficial de Java es el JDK (Java Development Kit). La herramienta JDK la podemos obtener de forma gratuita desde el sitio Web que posee Sun Microsystems dedicado por completo al lenguaje Java, <http://java.sun.com>.

Las versiones actuales de este producto son la 1.1.8, la 1.2 y la 1.3, correspondiéndose la primera con la versión 1.1 de Java y las dos últimas con la última versión denominada Java 2 Platform. Debido a que muchos fabricantes todavía no soportan la versión nueva de Java y también debido a que la nueva versión es bastante reciente, coexisten las dos versiones del lenguaje.

Sun Microsystems no nos lo pone fácil con las nomenclaturas de versiones y de productos, y si acudimos a su sitio Web para descargar el JDK, vemos que no aparece la versión 1.2 ni la 1.3. Esto es debido a que el JDK se encuentra incluido en lo que se denomina Java 2 SDK (Software Development Kit). De esta forma si deseamos la versión 1.2 del JDK veremos descargar el producto Java 2 SDK Standard Edition v 1.2, y si lo que deseamos es tener el JDK 1.3 debemos conseguir el Java 2 SDK Standard Edition v 1.3.

Podemos decir que el JDK de Sun ha pasado a llamarse SDK (Software Development Kit), existen varios SDK, cada uno de ellos se corresponde con una de las ediciones de la plataforma Java 2.

La plataforma Java 2 consta de tres ediciones distintas:

- Java 2 Standard Edition (J2SE): esta versión de la plataforma Java 2 contiene todas las clases tradicionales pertenecientes al lenguaje Java, se corresponde con la última versión de la

herramienta de desarrollo de Sun JDK (Java Development Kit). Esta plataforma es muy adecuada para la realización de aplicaciones tradicionales y para la construcción de applets. Esta versión de la plataforma Java 2 la implementa y la ofrece la herramienta Java 2 SDK Standard Edition (SDKSE).

- Java 2 Micro Edition (J2ME): esta edición ofrece un subconjunto de las clases principales del lenguaje Java, se trata de un entorno Java reducido para utilizar en pequeños procesadores o dispositivos.
- Java 2 Enterprise Edition (J2EE): frente al subconjunto que ofrecía la edición J2ME, nos encontramos esta versión extendida de la plataforma Java 2. J2EE se apoya en la plataforma Java 2 Standard Edition (J2SE), por lo tanto contiene además de las clase principales (core) del lenguaje Java una serie de extensiones para el desarrollo de aplicaciones empresariales. Dentro de estas extensiones, como intuirá el lector, encontramos la especificación 2.2 de los servlets y la especificación 1.1 de JavaServer Pages (JSP), aunque como veremos en el apartado correspondiente, estas extensiones no son las únicas que ofrece la plataforma J2EE. Esta edición ofrece también una serie de guías para el desarrollo de software basado en J2EE, estas guías toman la forma de una recomendación de arquitectura de software denominado modelo de aplicaciones J2EE (J2EE Application Model). Las aplicaciones Web son un buen ejemplo de aplicaciones que pueden seguir este modelo. El software que implementa esta edición de la plataforma Java 2 es el 2 SDK Enterprise Edition (SDKEE), aunque previamente será necesario disponer del Java 2 SDK Standard Edition(SDKSE). El SDKSE ofrece la implementación básica de plataforma Java 2 y el SDKEE ofrece las extensiones empresariales a esa plataforma.

A continuación vamos a comentar de forma general las características que incluye el lenguaje Java en su última versión y que han ido evolucionando desde la versión 1.0. El contenido de este apartado es a modo informativo. Este apartado puede ser útil a lectores que ya conozcan algunas de las versiones anteriores del lenguaje y también nos sirve para observar las nuevas características que ha ido implementando el lenguaje.

- Internacionalización: permite el desarrollo de applets y aplicaciones localizables, es decir, un mecanismo de localización sensible a la hora y fecha locales. Se incluye también la utilización de caracteres Unicode. Unicode tiene la capacidad de representar unos 65.000 caracteres, un número bastante amplio para incluir los caracteres de la mayoría de los lenguajes hablados hoy en día.
- Seguridad y firma de applets: el API (aquí entendemos como API un conjunto de clases e interfaces más o menos complejo que cumplen una funcionalidad común) de seguridad de Java está diseñado para permitir a los desarrolladores incorporar funcionalidades de seguridad a sus aplicaciones. Contiene APIs para firma digital y tratamiento de mensajes. Existen interfaces para la gestión de claves, tratamiento de certificados y control de accesos. También posee APIs específicos para el mantenimiento de certificados X.509 v3 y para otros formatos de certificado. Además se ofrecen herramientas para firmar ficheros JAR (Java ARchive).
- Ampliaciones del AWT: el AWT (Abstract Window Toolkit) es un API encargado de construir el GUI (Graphical User Interface, interfaz de usuario gráfico). El AWT de la versión 1.0 fue diseñado para construir sencillos interfaces de usuario por lo tanto se hizo necesario el ampliar el AWT en la versión 1.1. Estas ampliaciones tratan de resolver las principales deficiencias del AWT , además, representan un comienzo en la creación de una infraestructura más rica para el desarrollo de complicados interfaces de usuario, esto incluye: APIs para la impresión, componentes scroll, menús popup, portapapeles (copiar/pegar), cursores para cada componente, un modelo de eventos nuevo basado en la delegación de eventos (Delegation

Event Model), ampliaciones en el tratamiento de imágenes y gráficos, y un tratamiento de fuentes más flexible de cara a la característica de internacionalización.

- JavaBeans: inicialmente se puede definir un JavaBean como un componente software reutilizable, que puede ser manipulado visualmente en una herramienta de desarrollo. Un JavaBean sirve como un objeto independiente y reusable. El API de los JavaBeans define un modelo de componentes software para Java. Este modelo, desarrollado de forma coordinada entre las compañías Sun, Borland Inprise y otras, es una especificación de como codificar estos componentes para que puedan ser utilizados en diferentes entornos de programación (nosotros los utilizaremos desde las páginas JSP). Si creamos un componente con las especificaciones de los JavaBeans, ocultando los detalles de implementación y solamente mostrando las propiedades, métodos y eventos públicos conseguiremos las siguientes ventajas:
  1. Pueden ser utilizados por otros desarrolladores usando un interfaz estándar.
  2. Se pueden situar dentro de las tablas de componentes de diferentes herramientas de desarrollo.
  3. Se pueden comercializar por separado como si se tratara de un producto.
  4. Puede ser actualizado con un impacto mínimo sobre los sistemas en los que se encuentre.
  5. Es un componente escrito en Java puro, por lo tanto es independiente de la plataforma.
- Ficheros JAR (Java ARchive): este formato de fichero presenta muchos ficheros dentro de uno, y además permite la compresión de los mismos. Varios applets y sus componentes requeridos (ficheros .class, imágenes y sonidos) pueden encontrarse dentro de un fichero JAR y por lo tanto cargados por un navegador en una sola petición HTTP. Este formato de ficheros es parecido a los ficheros en formato Cabinet de Microsoft.
- Mejoras en Swing: Swing es el otro API existente para crear interfaces de usuario gráficos, ofrece muchos más componentes que el API AWT. Además los componentes Swing permiten modificar su aspecto y comportamiento.
- Ampliaciones de Entrada/Salida: el paquete java.io ha sido ampliado con flujos de caracteres, que son parecidos a los flujos de bytes excepto en que contienen caracteres Unicode de 16 bits en lugar de 8 bits. Los flujos de caracteres simplifican la escritura de programas que no dependen de una codificación de caracteres determinada, y son más fácil de internacionalizar.
- El paquete java.math: este paquete ofrece dos nuevas clases BigInteger y BigDecimal, para el tratamiento de operaciones numéricas.
- JDBC 2.0: JDBC es un API para ejecutar sentencias SQL (Structured Query Language), que ofrece un interfaz estándar para el acceso a bases de datos. Ofrece un acceso uniforme a un amplio número de bases de datos. El código de este API está completamente escrito en Java, de hecho se encuentra en el paquete java.sql, por lo tanto ofrece también independencia de la plataforma. Está basado y es muy similar a ODBC (Open Database Connectivity). Usando JDBC es muy sencillo enviar sentencias SQL a cualquier base de datos relacional, gracias al API de JDBC no es necesario escribir un programa que acceda a una base de datos Sybase, otro programa que acceda a una base de datos Oracle u otro que acceda a una base de datos Informix, el mismo programa servirá para ejecutar las sentencias SQL sobre todas estas bases de datos. Básicamente JDBC permite: establecer una conexión con una base de datos, enviar sentencias SQL y procesar los resultados. En la versión 2.0 de JDBC se ha mejorado el acceso

a datos haciéndolo más potente a través de la creación de diferentes tipos de cursores para acceder a los datos, entre otras muchas nuevas características.

- JFC (Java Foundation Classes): con este nombre se agrupan un gran número de clases especializadas en la construcción de interfaces de usuario. Dentro de JFC se encuentran las APIs ya mencionadas para la construcción de interfaces de usuario, es decir, Swing y AWT. Además incluye características de accesibilidad que permiten utilizar tecnologías tales como lectores de pantalla o dispositivos Braille, utilización de gráficos en dos dimensiones, soporte para arrastrar y soltar (Drag and Drop), posibilidad de modificar el aspecto y comportamiento de los componentes gráficos (pluggable Look and Feel), etc.
- Servlets: los servlets son aplicaciones Java que se ejecutan en servidores Web y que permiten construir páginas Web de forma dinámica. También permiten obtener información de los formularios enviados por los clientes. Son una tecnología que realiza funciones similares a los scripts CGI (Common Gateway Interface).
- JavaServer Pages: son páginas dinámicas que pueden utilizar código Java y hacer uso de los componentes JavaBeans, es una tecnología que realiza funciones similares a las páginas ASP (Active Server Pages) de Microsoft.

Ya hemos adelantado mediante pinceladas tres términos que vamos a utilizar de forma intensiva durante todo el texto, se trata de los conceptos de servlet, JavaServer Pages y JavaBeans, a continuación se ofrece una explicación algo más detallada (pero sin profundizar demasiado) de todos estos conceptos que constituyen los objetivos básicos de texto.

## Servlets

Ya hemos comentado que un servlet es un programa comparable a un CGI, es decir, se trata de un programa que se ejecuta en un servidor Web y que es utilizado para generar contenidos dinámicos en los sitios Web.

Básicamente un servlet recibe una petición de un usuario y devuelve un resultado de esa petición, que puede ser por ejemplo, el resultado de una búsqueda en una base de datos.

Pueden existir distintos tipos de servlets, no sólo limitados a servidores Web y al protocolo HTTP (HyperText Transfer Protocol), es decir, podemos tener servlets que se utilicen en servidores de correo o FTP, sin embargo en el mundo real, únicamente se utilizan servlets HTTP.

Si nos centramos en los servlets HTTP, el cliente del servlet será un navegador Web que realiza una petición utilizando para ello una cabecera de petición HTTP. El servlet procesará la petición y realizará las operaciones pertinentes, estas operaciones pueden ser todo lo variadas que precisemos. Normalmente el servlet devolverá una respuesta en formato HTML al navegador que le realizó la petición, esta respuesta se generará de forma dinámica.

En los servlets no existe una clara distinción entre el código HTML (contenido estático) y el código Java (lógica del programa), el contenido HTML que se envía al cliente se suele encontrar como cadenas de texto (objetos de la clase `java.lang.String`) que se incluyen en sentencias del tipo `out.println("<h1>Código HTML</h1>")`. Esto ocurría también con los predecesores de los servlets, es decir, los scripts CGI (Common Gateway Interface).

El API servlet 2.2 posee una serie de clases e interfaces que nos permiten interactuar con las peticiones y respuestas del protocolo HTTP. De esto se deduce que para entender los servlets y poder realizarlos de manera correcta deberemos comprender también el protocolo HTTP.



Ya comentamos que el API correspondiente a la especificación servlet 2.2, es decir, los servlets genéricos, se encontraba en el paquete `javax.servlet`, y los servlets HTTP se encuentran en un subpaquete del paquete anterior, este paquete se llama `javax.servlet.http`.

La dependencia de los servlets con el protocolo HTTP se hace patente en las distintas clases e interfaces que encontramos en el paquete `javax.servlet.http`. Así por ejemplo el interfaz `HttpServletRequest` representa la petición que se ha realizado a un servlet, es decir, una cabecera de petición del protocolo HTTP, y nos permite obtener información variada de la misma, como se puede comprobar el nombre del interfaz es bastante intuitivo. De esta misma forma existe un interfaz llamado `javax.servlet.http.HttpServletResponse`, que representa una cabecera de respuesta del protocolo HTTP y que nos ofrece un conjunto de métodos que nos permiten manipular y configurar de forma precisa la respuesta que se le va enviar al usuario.

En siguientes capítulos realizaremos una aproximación al protocolo HTTP, no de manera exhaustiva, sino de forma breve y sencilla para poder comprender satisfactoriamente las correspondencias entre el protocolo HTTP y los servlets.

También veremos a lo largo del texto, que tanto en los servlets como en las páginas JSP podemos utilizar todas las clases de la plataforma Java 2, es decir, podemos utilizar las mismas clases e interfaces que se utilizarían en una aplicación tradicional, así por ejemplo, si deseamos realizar un acceso a datos, utilizaremos desde nuestro servlet o página JSP el API JDBC de Java, simplemente deberemos importar el paquete `java.sql`, al igual que lo hacemos en un applet o aplicación Java.

Desde los servlets y páginas JSP podemos utilizar también componentes JavaBeans.

Por lo tanto, desde los servlets y las páginas JSP tenemos acceso a toda la potencia de la plataforma Java 2 sin realizar ningún tipo de esfuerzo. Pero no sólo tenemos la misma potencia que la ofrecida por el lenguaje Java, sino que páginas JSP y servlets tienen las mismas características del lenguaje, siendo una de las más destacables la independencia de la plataforma, así como el sólido modelo de orientación a objetos.

Para poder ejecutar servlets en un servidor Web, el servidor Web debe disponer motor o contenedor de servlets, en algunos casos el contenedor o motor es un añadido que se instala como un complemento al servidor Web, y en otros casos el propio servidor Web es también un motor de servlets.

## JavaServer Pages (JSP)

JavaServer Pages es una tecnología basada en la plataforma Java 2 que simplifica el proceso de desarrollo de sitios Web dinámicos. Con JSP, tanto los desarrolladores como los diseñadores Web pueden incorporar de forma rápida elementos dinámicos en páginas Web, utilizando código Java y una serie de etiquetas especiales determinadas.

Como una primera aproximación, y comparándolo con ASP (Active Server Pages), podemos decir que JSP es un tipo de lenguaje de script de servidor, aunque como veremos en siguientes capítulos, internamente funciona de manera completamente distinta, diferenciándose claramente de las páginas ASP, ya que ASP es interpretado cada vez que se carga una página y una página JSP es compilada a una clase de Java.

Las páginas JSP las podemos encontrar como ficheros de texto que poseen las extensión `.JSP`, y que tienen un lugar similar al de las páginas HTML tradicionales. Los ficheros JSP contienen código HTML tradicional junto con código en Java incrustado en la página (llamado `scriptlet`) y una serie de etiquetas especiales, que se ejecutarán en el servidor y darán lugar al resultado de la ejecución de la página en forma de código HTML.

Como se puede ver, si vemos de forma superficial las páginas JSP, podemos concluir que son muy parecidas a las páginas ASP, más adelante veremos que esto no es así.

En los servlets el código HTML que se genera y el código Java que contienen se encuentran completamente mezclados, sin embargo las páginas JSP ofrecen una clara distinción entre la parte dinámica de la página (código Java y etiquetas especiales) y la parte estática de la página (código HTML), es decir, existe una separación entre la lógica de la programación y la presentación de los contenidos.

Al separar de esta forma tan clara la presentación del contenido, podemos repartir de forma más sencilla las funciones del equipo de desarrollo para que cada miembro del equipo realice la tarea más adecuada según su perfil. De esta forma los diseñadores Web podrán construir el código HTML y los desarrolladores Java podrán construir toda la lógica de la página JSP, incluyendo los componentes JavaBeans que pueda utilizar la página.

Una página JSP posee diversos elementos, a los bloques o fragmentos de código Java se les denomina scriptlets, a un conjunto de etiquetas especiales se les denomina directivas y a otro conjunto de estas etiquetas se les denomina acciones.

Una página JSP tiene dos posibilidades a la hora de generar contenido dinámico, por un lado este contenido se puede generar a través de código Java incrustado en la página JSP (scriptlets), de forma similar a como lo generan las páginas ASP a través del lenguaje de script VBScript, y por otro lado el contenido dinámico se puede generar a través de las etiquetas especiales (acciones y directivas), de forma similar a como lo hacen algunos sistemas de plantillas como pueden ser ColdFusion o WebMacro, algunas de estas etiquetas especiales están basadas en XML.

Podemos decir que JSP es un híbrido entre los scripts de servidor y los sistemas o herramientas basados en plantillas (template systems).

JSP 1.1 extiende la especificación 2.2 de los servlets permitiendo una clara separación entre la lógica del programa y la presentación, por lo tanto los servlets y las páginas JSP se encuentran íntimamente relacionados, de hecho todo lo que podemos realizar con una página JSP lo podemos realizar también con un servlet.

Las páginas JSP, antes de ser ejecutadas, se traducen a un servlet equivalente de manera automática. Esto lo veremos detenidamente cuando nos ocupemos del funcionamiento interno del motor JSP (también denominado contenedor) que permite ejecutar las páginas JSP.

Con las páginas JSP ocurre lo mismo que con los servlets, es necesario la presencia de un motor de JSP para poder ejecutar las páginas en un servidor Web. Y también en algunos casos el motor JSP o contenedor JSP es un añadido que se instala en el servidor Web y en otros casos se trata de un servidor Web que a su vez es un motor JSP.

Normalmente los motores o contenedores que permiten la ejecución de servlets y de páginas JSP suelen ser el mismo producto, así por ejemplo Jakarta Tomcat 3.1 permite la ejecución tanto de servlets como de páginas JSP, ofreciendo la implementación oficial de los estándares servlet 2.2 y JSP 1.1, y además se puede utilizar como un servidor Web.

## JavaBeans

Para definir de forma sencilla lo que es JavaBeans, podemos decir que se trata de una especificación que permite escribir componentes software en Java. Los componentes que siguen la arquitectura descrita por JavaBeans se denominan Beans o incluso JavaBeans o Beans de Java.

Para que una clase determinada pueda ser considerada un Bean debe seguir una serie de especificaciones descritas en el API JavaBeans.

Se puede distinguir tres categorías de Beans:

- Componentes visuales (visual component Beans): estos Beans se utilizan como elementos dentro de interfaces de usuario gráficos (GUI, Graphical User Interface).
- Beans de datos (data Beans): estos componentes ofrecen acceso a una serie de información que puede estar almacenada en tablas de una base de datos.
- Beans de servicios (service Beans): también conocidos como Beans trabajadores (worker Beans), en este caso se encuentran especializados en la realización de una serie de tareas específicas o cálculos.

Esta separación o distinción de categorías de los Beans no es rígida, podemos tener Beans que pertenezcan a más de una de las categorías mencionadas.

El API JavaBeans ofrece un formato estándar para las clases Java. Las clases que quieran ser consideradas y tratadas como Beans deben seguir las reglas definidas por el API JavaBeans.

Al igual que ocurre con muchos componentes software los Beans encapsulan tanto su estado como comportamiento. El aspecto que presenta la implementación un Bean es como el de una clase típica de Java, lo único que esta clase debe seguir las especificaciones que indica JavaBeans, en el capítulo correspondiente se mostrarán estas características y veremos como podemos crear nuestros propios componentes JavaBeans.

Los Beans los podemos utilizar tanto desde los servlets como desde las páginas JSP, aunque la forma de utilizarlos es distinta. Desde un servlet utilizaremos un Bean de la misma forma que utilizamos una instancia de una clase cualquiera de Java, sin embargo desde las páginas JSP, los Beans los utilizamos a través de varias etiquetas especiales de la especificación JSP 1.1, que se encuentran incluidas junto al código HTML de la página y que forman parte de lo que se denomina acciones.

Mediante estas etiquetas, que veremos oportunamente más adelante, podremos instanciar Beans, modificar las propiedades de los Beans creados y obtener también los valores de las mismas. La forma de hacer unos de componentes JavaBeans dentro de una página JSP es distinta a la de un servlet, debido a que una página JSP es un contenedor Bean.

Un contenedor Bean (Bean container) es una aplicación, entorno o lenguaje de programación que permite a los desarrolladores realizar llamadas a Beans, configurarlos y acceder a su información y comportamiento.

Los contenedores Beans permiten a los desarrolladores trabajar con los Beans a un nivel conceptual mayor que el que se obtendría accediendo a los Beans desde una aplicación Java. Esto es posible debido a que los Beans exponen sus características (atributos o propiedades) y comportamientos (métodos) al contenedor de Beans, permitiendo al desarrollador trabajar con los Beans de una manera más intuitiva.

Existen herramientas de desarrollo que contienen contenedores Bean que permiten trabajar con componentes Beans de una manera visual, algunas de estas herramientas son Bean Box de Sun, Visual Age for Java de IBM, Visual Café de Symantec o JBuilder de Borland. Con estas aplicaciones se pueden manipular los Beans arrastrándolos a una posición determinada y definiendo sus propiedades y comportamiento. La herramienta generará de manera más o menos automática todo el código Java necesario.

De forma similar a las herramientas de programación visual las páginas JSP permiten a los desarrolladores crear aplicaciones Web basadas en Java sin la necesidad de escribir código Java, debido a que desde JSP podemos interactuar con los Beans a través de las ya mencionadas etiquetas especiales que se alojan junto con el código HTML de la página.

Por lo tanto las herramientas de programación Visual, como puede ser JBuilder, y otros programas, como pueden ser las páginas JSP, pueden descubrir de manera automática la información sobre las clases que siguen la especificación JavaBeans y pueden crear y manipular los componentes JavaBeans sin que el desarrollador tenga que escribir código de manera explícita.

## J2EE (Java 2 Enterprise Edition)

Ya hemos comentado que el API servlet 2.2 y el API JSP 1.1 forman parte de una tecnología más amplia denominada Java 2 Enterprise Edition (J2EE). La plataforma J2EE es una especificación que se encuentra compuesta de varias partes.

J2EE está formada por un gran número de tecnologías diferentes y una especificación que indica como deben trabajar estas tecnologías conjuntamente. Las partes principales del entorno J2EE son:

- Componentes de aplicación (application components): existen tres grandes grupos de componentes de aplicación, se trata de los siguientes.
  - Aplicaciones clientes: estos componentes son clientes gruesos implementados mediante aplicaciones Java tradicionales. Acceden al servidor de aplicaciones utilizando el mecanismo conocido como Remote Method Invocation (RMI).
  - Applets: son clientes que suelen formar parte del interfaz de usuario gráfico y que se ejecutan dentro de un navegador Web.
  - Servlets y JavaServer Pages: componentes que nos permiten generar aplicaciones Web dinámicas y a los que se les ha dedicado el presente texto.
  - Enterprise JavaBeans (EJB): son componentes que se ejecutan en un contenedor (contenedor EJB) en el servidor de aplicaciones que gestiona y trata gran parte de su funcionamiento interno.
- Contenedores: cada tipo de componente se ejecuta dentro de un contenedor, este contenedor ofrece al componente una serie de servicios en tiempo de ejecución. Hay un tipo de contenedor para cada tipo de componente. Los contenedores más interesantes son los contenedores Web que permite ejecutar servlets y páginas JSP, y el contenedor EJB que gestiona los componentes Enterprise JavaBeans (EJB).
- Drivers gestores de recursos: un driver gestor de recursos es una driver que ofrece alguna clase de conectividad con un componente externo, algunos de estos drivers son por ejemplo los APIs utilizados por JDBC, Java Messaging Service (JMS) o JavaMail.
- Bases de datos: las bases de datos dentro de la plataforma J2EE son accesibles a través del API que ofrece el acceso a datos en Java, que no es otro que JDBC.

## Enterprise JavaBeans

Enterprise JavaBeans (EJB) es una especificación de Sun que estandariza como construir componentes de servidor para aplicaciones distribuidas.

La especificación Enterprise JavaBeans (EJB) 1.1 define una arquitectura para el desarrollo y distribución de aplicaciones transaccionales basadas en componentes software de servidor. Las organizaciones pueden construir sus propios componentes o bien comprarlos a terceras partes. Estos componentes de servidor denominados Enterprise Beans, son objetos distribuidos que se alojan en un contenedor denominado contenedor Enterprise JavaBeans (EJB container) y ofrece acceso remoto a servicios para clientes distribuidos por toda la red.

El contenedor EJB aloja y gestiona un Enterprise Bean de la misma forma que un servidor Web aloja un servlet o un navegador Web contiene un applet de Java. Un Enterprise JavaBean no puede funcionar fuera del contenedor EJB.

El contenedor EJB gestiona cada uno de los aspectos de un Enterprise Bean en tiempo de ejecución, como puede ser al acceso remoto al Bean (RMI, Remote Method Invocation), la seguridad, la persistencia, las transacciones, gestión del ciclo de vida, concurrencia y el acceso a los recursos. Los contenedores pueden gestionar diversos Beans de manera simultánea, de forma similar a como puede suceder con un servidor Web con soporte para servlets que puede manejar numerosos servlets simultáneamente.

Existen dos tipos de Enterprise Beans, los Beans de entidad (Entity Beans) y los Beans de sesión (Session Beans). Básicamente un Entity Bean representa datos originarios de una base de datos, y un Session Bean representa un proceso y actúa como un agente que realiza una tarea.

A la hora de realizar y desarrollar Enterprise JavaBeans se deben evitar realizar una serie de acciones para que el contenedor EJB pueda gestionar el funcionamiento del componente de forma satisfactoria. Debido a esto los EJB presentan una serie de restricciones que no permiten realizar las siguientes tareas:

- Gestionar hilos de ejecución (threads).
- Acceder a ficheros de manera directa.
- Utilizar gráficos.
- Escuchar sockets.
- Cargar una librería nativa.



# Introducción a los servlets

---

## Introducción

En este capítulo vamos a realizar una presentación de los servlets, definiéndolos de forma exacta y comparándolos también con otra tecnología alternativa en la que se basa y mejora, estamos hablando de la especificación CGI (Common Gateway Interface).

También comentaremos los requisitos mínimos que debe poseer nuestra máquina para poder seguir el texto con éxito, y además se mostrará como realizar la configuración mínima del servidor Web Jakarta Tomcat 3.1 de Apache, que implementa la especificación Java servlet 2.2.

Otro punto que trataremos será la estructura básica de un servlet, incluso crearemos y ejecutaremos nuestro primer servlet.

Veremos que los servlets y el protocolo http se encuentran muy relacionados, por lo que dedicaremos un apartado al protocolo HTTP.

## Definición de servlet

Los servlets son clases Java que se ejecutan en un servidor de aplicación, para contestar a las peticiones de los clientes. Los servlets no se encuentran limitados a un protocolo de comunicaciones específico entre clientes y servidores, pero en la práctica podemos decir que se utilizan únicamente con el protocolo HTTP, por lo que el servidor de aplicación pasa a denominarse entonces servidor Web.

Por lo tanto pueden existir distintos tipos de servlets, no sólo limitados a servidores Web y al protocolo HTTP (HyperText Transfer Protocol), es decir, podemos tener servlets que se utilicen en servidores de correo o FTP, sin embargo en el mundo real, únicamente se utilizan servlets HTTP.

En realidad este texto trata los servlets HTTP, aunque siempre los vamos a llamar simplemente servlets, que son los que realmente se utilizan.

Los servlets HTTP implementan la especificación servlet 2.2 pero adaptándola al entorno de los servidores Web e Internet.

Un servlet es muy similar a un script CGI, es un programa que se ejecuta en un servidor Web actuando como una capa intermediaria entre una petición procedente de un navegador Web y aplicaciones, bases de datos o recursos del servidor Web.

La especificación que definen los servlets en su versión 2.2 la encontramos implementada mediante dos paquetes que podemos encontrar formando parte de la plataforma Java 2 Enterprise Edition. El paquete `javax.servlet`, define el marco básico que se corresponde con los servlets genéricos y el paquete `javax.servlet.http` contiene las extensiones que se realizan a los servlets genéricos necesarias para los servlets HTTP, que como ya hemos dicho son los servlets que vamos a tratar en el texto.

Los servlets pueden utilizar todos los APIs que ofrece el lenguaje Java, además tienen las mismas características que define Java, y por lo tanto son una solución idónea para el desarrollo de aplicaciones Web de forma independiente del servidor Web y del sistema operativo en el que se encuentren, esto es así gracias a la característica que ofrece Java de independencia de la plataforma.

Ya hemos comentado que los servlets se ejecutan de manera independiente del servidor Web en el que se hallen, pero este servidor Web debe cumplir un requisito, implementar la especificación Java servlet 2.2. Existen varias opciones referentes a este aspecto, el propio servidor Web puede implementar por sí mismo la especificación Java servlet 2.2, es decir, además de servidor Web es contenedor de servlets, o bien podemos instalar junto al servidor Web un añadido que funciona como un motor o contenedor de servlets. Que el lector no se vea abrumado, en este mismo capítulo indicaremos la configuración mínima que debemos realizar para disponer de un servidor Web que implemente la especificación Java servlet 2.2.

Volvamos al objetivo principal del presente apartado, es decir, la definición de un servlet. Un servlet también se define por los trabajos o tareas típicas que realiza, estas tareas se comentan a continuación en el orden lógico en el que se realizan:

1. Leer los datos enviados por el usuario: normalmente estos datos se indican a través de formularios HTML que se encuentran en páginas Web. Aunque esta información también puede provenir de otras herramientas HTTP o bien desde applets.
2. Buscar otra información sobre la petición que se encuentra incluida en la petición HTTP: esta información incluye detalles tales como las capacidades y características del navegador, cookies, el nombre de la máquina del cliente, etc.
3. Generar los resultados: este proceso puede requerir acceder a una base de datos utilizando JDBC, utilizar un componente JavaBean, o generar la respuesta de manera directa.
4. Formatear los resultados en un documento: en la mayoría de los casos implica incluir los resultados en una página HTML.
5. Asignar los parámetros apropiados de la respuesta HTTP: esto implica indicar al navegador el tipo de documento que se le envía (por ejemplo HTML), asignar valores a las cookies, y otras tareas.



6. Enviar el documento al cliente: el documento se puede enviar en formato de texto (HTML), formato binario (imágenes GIF), o incluso en formato comprimido como un fichero ZIP.

A continuación exponemos una serie de razones por las que pueden ser interesante generar una página Web de forma dinámica:

- La página está basada en la información enviada por el usuario. Por ejemplo, la página de resultados de un buscador o motor de búsqueda.
- La página Web se deriva de datos que cambian frecuentemente. Un ejemplo puede ser una página de información meteorológica o una página de titulares de las últimas noticias.
- La página utiliza información de bases de datos corporativas o de otros recursos del lado del servidor. Por ejemplo, un sitio Web de comercio electrónico.

## Comparación entre los servlets y los scripts CGI

Tradicionalmente la forma de añadir funcionalidad a un servidor Web era mediante scripts CGI (Common Gateway Interface).

Java ofrece una alternativa más sencilla, rápida, eficiente y potente a través de la especificación Java servlet 2.2. A continuación vamos a comentar las ventajas y diferencias que muestran los servlets frente a los scripts CGI.

- **Eficiencia:** con los scripts CGI, cada vez que se realiza una petición HTTP sobre un script CGI se inicia un nuevo proceso en el servidor Web. Con los servlets, la máquina virtual de Java (JVM, Java Virtual Machine) trata cada petición realizada utilizando un hilo de ejecución (thread) ligero de Java, no un proceso pesado de sistema. De forma similar, si se realizan varias peticiones simultáneas sobre el mismo script CGI, el script es cargado varias veces en memoria, por el contrario los servlets existirán varios hilos de ejecución pero únicamente una sola copia de la clase del servlet. Además un servlet se mantiene en memoria entre distintas peticiones, sin embargo un script CGI necesita ser cargado e iniciado cada vez que se produce una petición sobre el script CGI.
- **Conveniencia:** los servlets ofrecen una gran infraestructura para manejar y decodificar datos de formularios HTML, leer y establecer encabezados HTTP, manejar cookies, mantener sesiones y otras utilidades de servidor. Si ya conocemos el lenguaje Java, para qué podría interesarnos aprender Perl o C++, si se supone que Java ofrece una orientación a objetos más sencilla y un modelo de programación más robusto.
- **Potencia:** los servlets soportan una serie de capacidades que con los scripts CGI es muy difícil o incluso imposible de conseguir. Los servlets se pueden comunicar de manera directa con el servidor Web, mientras que los scripts CGI puede hacerlo pero utilizando un API específico del servidor Web. Los servlets pueden mantener la información entre distintas peticiones, simplificando las técnicas del mantenimiento de la sesión.
- **Portabilidad e independencia de la plataforma:** los servlets se construyen e implementan atendiendo a la especificación Java servlet 2.2 definida en la plataforma J2EE (Java 2 Enterprise Edition), por lo tanto si escribimos un servlet para un servidor Web Apache, ese mismo servlet se podrá ejecutar sin ningún cambio sobre un servidor Web Internet Information Server (IIS) de Microsoft, ya que el servlet no se ajusta a un servidor Web concreto, sino que se basa en la utilización de la especificación Java servlet 2.2. El único

requisito es que el servidor Web debe soportar servlets. Un servidor Web soporta servlets si implementa la especificación Java servlet 2.2, es decir, debe poseer un contenedor de servlets.

- Seguridad: un servlet se ejecuta dentro de un entorno bien delimitado, este entorno es el contenedor de servlets o motor de servlets. Los scripts CGI realizan numerosas llamadas al sistema y adolecen de algunos problemas de seguridad como puede ser la utilización de punteros que acceden directamente a posiciones de memoria.
- Baratos: existen algunos servidores Web que soportan servlets y que son gratuitos. Muchos de estos servidores se puede utilizar para uso personal o para sitios Web de bajo volumen, la excepción es el servidor Web Apache, que es gratuito y se utiliza en sitios Web de gran volumen. De todas formas, una vez que tenemos nuestro servidor Web, añadirle soporte para servlets no supone un gran coste adicional, dependiendo del añadido o extensión que seleccionemos, para instalar en nuestro servidor Web para que éste soporte servlets, incluso puede resultar gratuito.

En este apartado hemos comentado el antecedente de los servlets, los scripts CGI, y además hemos realizado una comparación entre ambas tecnologías. El siguiente apartado pretende ser más práctico, ya que en él vamos a comentar los requisitos mínimos que se necesitan para realizar los distintos ejemplos de servlets que se presentan en el texto.

## Preparando el entorno

A continuación vamos a comentar los requisitos mínimos con los que debemos dotar a nuestra máquina para poder seguir el texto de forma correcta. No vamos a mostrar la configuración completa del software necesario, sólo nos vamos a fijar en los aspectos más relevantes, que nos van a servir para poder probar nuestros primeros servlets HTTP de Java (a partir de ahora siempre servlets).

Lo primero es obtener el compilador y la máquina virtual de Java, junto con toda su referencia, es decir, necesitamos un SDK (Software Development Kit) de la plataforma Java 2. Podemos elegir entre dos SDKs distintos, el SDK Standard Edition (SDKSE) que se corresponde con la plataforma J2SE (Java 2 Standard Edition) o bien el SDK Enterprise Edition (SDKEE) que se corresponde con la plataforma J2EE (Java 2 Enterprise Edition). Si elegimos esta última opción deberemos instalar además del SDKEE y el SDKSE, ya que el primero ofrece las extensiones empresariales a la plataforma Java 2, y el segundo ofrece el estándar de la plataforma Java 2, es decir, toda la estructura de las clases del lenguaje Java. El texto se ha realizado utilizando la plataforma Java2 Standard Edition.

Cualquiera de las versiones de la implementación de la plataforma Java 2 las podemos obtener del sitio Web de Java: <http://java.sun.com>. En este sitio Web podemos consultar por los productos existentes que ofrece Sun Microsystems.

La instalación del SDK es muy sencilla y se supone que los lectores están familiarizados con su utilización. El único apunte que realizamos es que se debe situar en la variable de entorno PATH la ruta del directorio BIN contenido en el directorio en el que se encuentra instalado el SDK de la plataforma Java 2, así por ejemplo en el caso de instalar la implementación de la plataforma Java 2 ofrecida por el SDKSE en la variable de entorno PATH debe añadir la siguiente ruta `c:\jdk1.3\bin`.

Ya tenemos la primera parte del entorno, una implementación de la plataforma Java 2, en siguiente paso es conseguir que un servidor Web pueda ejecutar servlets, es decir, que implemente la especificación Java servlet 2.2. En este caso existen dos posibilidades, podemos disponer de un servidor Web que implemente por sí sólo la especificación Java servlet 2.2. o bien podemos tener un

servidor Web al que se le instala un añadido que implementa la especificación Java servlet 2.2 y que realiza la función de motor o contenedor de servlets.

En nuestro caso vamos a optar por la solución más sencilla que es la de obtener un servidor Web que implemente por si mismo la especificación Java servlet 2.2. La implementación oficial de la especificación servlet 2.2 la ofrece el servidor Web Jakarta Tomcat 3.1 de Apache.

El servidor Web Jakarta Tomcat se puede utilizar como un servidor Web de prueba para ejecutar servlets y de páginas JSP (JavaServer Pages), es decir, además de implementar la especificación Java servlet 2.2 implementa la especificación JavaServer Pages 1.1. Jakarta Tomcat también puede utilizarse como un añadido para el servidor Apache, es decir, permite que el servidor Web Apache pueda ejecutar servlets y páginas JSP. Esta última utilización es la que se le debería dar a Jakarta Tomcat en un entorno de producción real.

Sin embargo, como ya hemos comentado, para poder seguir los ejemplos de este texto nosotros vamos a utilizar Jakarta Tomcat como un servidor Web completo, es decir, como un servidor Web que soporta servlets y no como un módulo añadido a un servidor Web existente. Además Jakarta Tomcat está preparado para poder ejecutarse sin problemas en una máquina en la que ya se encuentre instalado un servidor Web, ya que por defecto el servidor Web Jakarta Tomcat utiliza el puerto 8080 para el protocolo HTTP, en lugar del estándar, es decir, el puerto 80.

Jakarta Tomcat es un servidor Web gratuito ofrecido por Apache, y que podemos obtener en la siguiente dirección <http://jakarta.apache.org>.

Jakarta Tomcat se ofrece en forma binaria, es decir, compilado y listo para utilizar, y también se ofrece el código fuente para que sea el desarrollador quien lo compile. Por sencillez vamos a utilizar la primera opción. Lo que obtenemos del sitio Web de Jakarta Tomcat es un fichero comprimido en formato ZIP (`jakarta-tomcat.zip`), que debemos descomprimir en nuestra máquina.

Al descomprimir el fichero, se nos generará una estructura de directorios que contiene los distintos elementos del servidor Web Jakarta Tomcat, en el capítulo correspondiente detallaremos esta estructura de directorios y que funcionalidad tiene cada uno de ellos. El directorio en el que se encuentra toda la estructura de Jakarta Tomcat se llama `jakarta-tomcat`.

La configuración del servidor Web Jakarta Tomcat no resulta nada intuitiva ni fácil de realizar, al igual que ocurre con el servidor Web Apache. En este apartado vamos a comentar únicamente la configuración mínima necesaria que se necesita para poder ejecutar Tomcat.

Dos ficheros que se adjuntan junto Jakarta Tomcat, y a los que hay que prestar una atención especial son: `STARTUP.BAT` y `SHUTDOWN.BAT`. Al ejecutar el primero de ellos iniciaremos la ejecución del servidor Web Tomcat, y al ejecutar el segundo se finalizará la ejecución del servidor Jakarta Tomcat. Estos dos ficheros se encuentran en el directorio `jakarta-tomcat\bin`.

Para poder ejecutar el servidor Web Jakarta Tomcat debemos modificar el fichero de inicio del mismo, es decir, el fichero `STARTUP.BAT`. Esta modificación consiste en añadir una línea que especifique un parámetro que indicará la localización de la implementación de la plataforma Java 2, es decir, la localización del Java 2 SDK (Software Development Kit), que se supone ya tenemos instalado del paso anterior.

En el fichero `STARTUP.BAT` se debe especificar la variable `JAVA_HOME` después de los comentarios que parecen en el fichero (varias líneas comenzando por `rem`). Tanto si hemos instalado la plataforma J2EE o la plataforma J2SE, deberemos indicar el directorio de instalación del Java 2 SDK Estándar Edition, que por defecto es `c:\jdk1.3\`, de todas formas el lector puede comprobar fácilmente

el directorio raíz en el que ha instalado el Java 2 SDKSE. Por lo tanto la línea que debemos añadir en STARTUP.BAT es:

```
SET JAVA_HOME=c:\jdk1.3\
```

Un ejemplo de fichero STARTUP.BAT completo se ofrece a continuación:

```
@echo off
rem $Id: startup.bat,v 1.7 2000/03/31 19:40:02 craigmcc Exp $
rem Startup batch file for tomcat servner.

rem This batch file written and tested under Windows NT
rem Improvements to this file are welcome

SET JAVA_HOME=C:\jdk1.3\

if not "%TOMCAT_HOME%" == "" goto start

SET TOMCAT_HOME=.
if exist %TOMCAT_HOME%\bin\tomcat.bat goto start

SET TOMCAT_HOME=..
if exist %TOMCAT_HOME%\bin\tomcat.bat goto start

SET TOMCAT_HOME=
echo Unable to determine the value of TOMCAT_HOME.
goto eof

:start
call %TOMCAT_HOME%\bin\tomcat start %1 %2 %3 %4 %5 %6 %7 %8 %9

:eof
```

Una vez realizado este cambio ya estamos preparados para iniciar la ejecución de Jakarta Tomcat. Para iniciar la ejecución de Jakarta Tomcat simplemente debemos realizar doble clic sobre el fichero STARTUP.BAT, y aparecerá una ventana muy similar a la de la Figura 1.

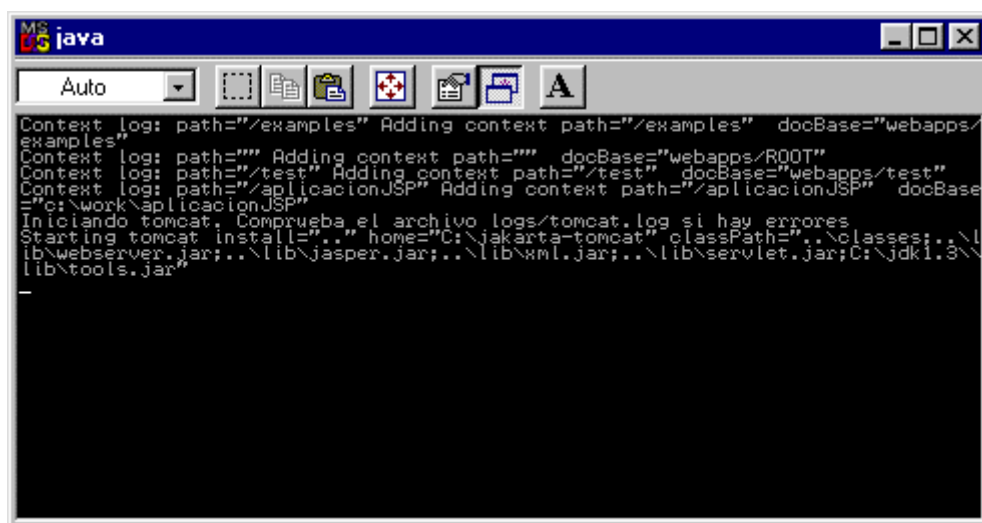


Figura 1. Inicio de Jakarta Tomcat

Para verificar que está todo correcto y que Jakarta Tomcat puede ejecutar servlets sin ningún problema vamos a realizar una sencilla comprobación. Debemos ejecutar un navegador Web, ya sea Internet Explorer o Netscape Communicator, y debemos especificar la siguiente URL `http://localhost:8080/examples`. Esta URL se corresponde con los ejemplos de servlets y páginas JSP, que se ofrecen en la instalación de Jakarta Tomcat, el aspecto de esta página se puede comprobar en la Figura 2.

Para comprobar el correcto funcionamiento de un servlet en el servidor Tomcat pulsaremos sobre el enlace servlets. Este enlace nos lleva a la página de ejemplos de servlets, en la que al pulsar cada uno de los enlaces de cada ejemplo se invocará el servlet correspondiente.

Sugiero al lector que pruebe todos los ejemplos disponibles e incluso que pulse la opción de ver el código fuente de cada ejemplo, para que vaya teniendo un primer contacto con la forma en la que se va a implementar un servlet. Así por ejemplo si pulsamos sobre el primer ejemplo, llamado Hello World, ejecutaremos el servlet HelloWorldExample que nos mostrará la página de la Figura 3.

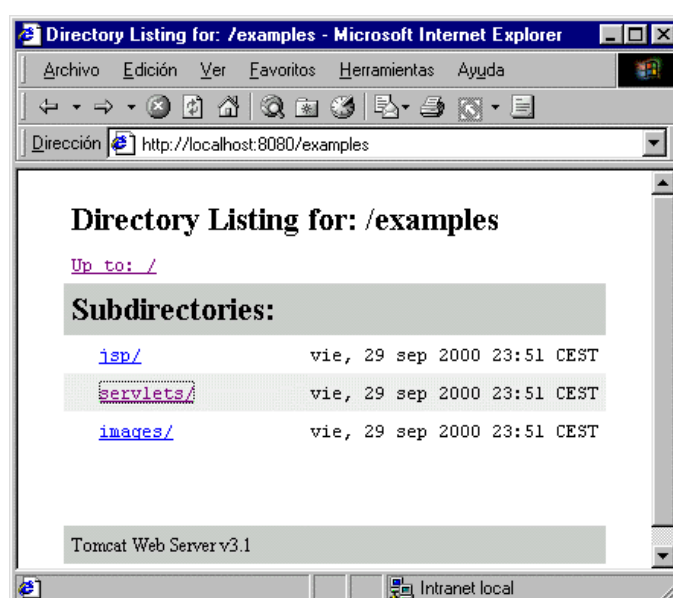


Figura 2. Ejemplos disponibles en Tomcat



Figura 3. Ejecución de un servlet de ejemplo

El código fuente de este servlet es el que se ofrece en el Código Fuente 1. El lector no se debe preocupar si no entiende el código que se ofrece, en el siguiente apartado construiremos paso a paso un servlet sencillo muy similar al del ejemplo ofrecido por Tomcat.

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

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Código Fuente 1

Si deseamos detener Tomcat no tenemos nada más que ejecutar el fichero SHUTDOWN.BAT. Al ejecutar este fichero se detendrá el servidor Web Jakarta Tomcat y se cerrará la ventana del intérprete de comandos que se había abierto al iniciarse su ejecución.

## Estructura básica de un servlet

Ya tenemos todo nuestro entorno preparado para la ejecución de servlets desde un servidor Web, pero antes de crear nuestro primer servlet vamos a realizar unos comentarios para adelantar algunos conceptos relativos a la estructura de un servlet que explicaremos con más detalle más adelante.

Un servlet va a ser invocado a través de una petición del usuario desde el navegador Web, esta petición se puede realizar de varias formas, a través de un enlace o a través del envío de un formulario HTML. En el primer caso estamos utilizando un método de petición del protocolo HTTP denominado GET, y en el segundo caso indicaremos mediante el atributo METHOD de la etiqueta <FORM> del formulario HTML, el método de petición HTTP utilizado, que pueden ser los métodos GET o POST. En el apartado correspondiente trataremos en más detalle el protocolo HTTP, comentando los aspectos que nos interesan desde el punto de vista de los servlets, no debemos olvidar que los servlets se basan en gran medida y utilizan el protocolo HTTP.

Para que una clase pueda ser considerada un servlet y tratada como tal, debe heredar de la clase javax.servlet.http.HttpServlet. La clase HttpServlet es una clase abstracta que representa a los servlets HTTP y de la que deben heredar todas las clases que deseen ser servlets. Esta clase ofrece una serie de métodos como pueden ser doGet(), doDelete() o doPost() que podrá sobrescribir el servlet, es decir, la clase que hereda de la clase abstracta HttpServlet.

Lo más normal es que el servlet sobrescriba el método `doGet()` o `doPost()` o ambos. Cada uno de estos métodos se ejecutarán cada vez que el cliente desde su navegador Web realice una petición HTTP mediante el método GET o POST respectivamente. Si no queremos distinguir entre el método utilizado para invocar al servlets, es decir, queremos que el servlet reaccione igualmente a peticiones del método GET y POST, se puede llamar al método `doPost()` desde el método `doGet()` o viceversa, de esta forma el servlet se ejecutará de la misma forma con independencia del método de petición HTTP empleado.

Tanto el método `doPost()` como el método `doGet()` poseen dos parámetros: un objeto `HttpServletRequest` y un objeto `HttpServletResponse`. `HttpServletRequest` y `HttpServletResponse` son dos interfaces pertenecientes al paquete `javax.servlet.http`, que representan respectivamente una petición del protocolo HTTP y una respuesta del protocolo HTTP. Como podemos ver la relación entre los servlets y el protocolo HTTP no es meramente casual.

El interfaz `HttpServletRequest` ofrece una serie de métodos mediante los cuales podemos obtener información relativa a la petición HTTP realizada, como puede ser información incluida en formularios HTML, cabeceras de petición HTTP (más adelante veremos el significado de estas cabeceras), el nombre de máquina del cliente, etc. Se puede considerar que este interfaz representa la entrada que realiza el usuario.

El interfaz `HttpServletResponse` nos permite especificar información de respuesta a través del protocolo HTTP, esta información pueden ser cabeceras de respuesta del protocolo HTTP, códigos de estado, y lo más importante, el interfaz `HttpServletResponse` nos permite obtener un objeto `PrintWriter`, utilizado para enviar el contenido del documento al usuario. Muchos servlets sencillos, como el que se ha mostrado en el apartado anterior, se limitan a ejecutar sentencias `println()` sobre el objeto `PrintWriter` para generar la página correspondiente. Se puede considerar que este interfaz representa la salida que se envía al usuario.

Los métodos `doGet()` y `doPost()` de la clase `HttpServlet` lanzan dos excepciones, `ServletException` e `IOException`, por lo tanto debemos incluir estas dos excepciones en la declaración de los métodos.

Como mínimo un servlet debe importar el paquete `javax.servlet`, para poder tratar las excepciones `ServletException` y el paquete `javax.servlet.http`, para poder heredar de la clase `HttpServlet` y para poder utilizar los objetos que representan la petición HTTP del usuario y la respuesta que se le envía, es decir, para poder utilizar los objetos `HttpServletRequest` y `HttpServletResponse`. También es bastante común que se deba importar el paquete de entrada/salida de Java, es decir, el paquete `java.io`, que nos permite utilizar el objeto `PrintWriter` obtenido a través del objeto `HttpServletResponse`.

De los paquetes característicos de los servlets vamos a tratar en el siguiente apartado.

## Los paquetes de la especificación Java servlet 2.2

La especificación Java servlet 2.2 podemos considerar que se encuentra formada por dos paquetes pertenecientes a la plataforma Java 2 Enterprise Edition (J2EE). Estos dos paquetes son el paquete `javax.servlet` y el paquete `javax.servlet.http`. El primer paquete representa a los servlets genéricos y el segundo a los servlets HTTP, es decir, aquellos que se van a ejecutar en el entorno de un servidor Web.

A través de las clases e interfaces del paquete `javax.servlet.http` podemos construir servlets que se ejecuten en un servidor Web y que a partir de una información ofrecida por el usuario, generen una respuesta dinámica al usuario basada en la información facilitada por éste.

El paquete `javax.servlet.http` lo tendremos que importar, junto con el paquete `javax.servlet`, si queremos que una clase sea un servlet de Java. El paquete `javax.servlet` proporciona los servlets genéricos y además nos ofrece la clase de excepciones para los servlets, `ServletException`, además de otras clases e interfaces de gran utilidad.

A continuación vamos a comentar de forma breve el contenido de estos dos paquetes, es decir, las clases e interfaces que contienen.

Dentro del paquete `javax.servlet` podemos encontrar los siguientes elementos (Tabla 1, Tabla 2 y Tabla 3):

<b>Interfaces</b>	
RequestDispatcher	Define un objeto que recibe peticiones desde un cliente y las envía a otro recurso, como puede ser un servlet, una página HTML o una página JSP.
Servlet	Define una serie de métodos que todos los servlets deben implementar.
ServletConfig	Es un objeto de configuración de servlets utilizado por el contenedor o motor de servlets para pasar información a un servlet durante su inicialización.
ServletContext	Define una serie de métodos que un servlet va a utilizar para comunicarse con el contenedor de servlets correspondiente, por ejemplo, para obtener el tipo MIME de un fichero, reenviar peticiones, o escribir a un fichero de registro.
ServletRequest	Define un objeto que ofrece información de la petición del cliente al servlet correspondiente.
ServletResponse	Este interfaz define un objeto que permite al servlet correspondiente enviar información al cliente, como por ejemplo, código HTML generado de forma dinámica a partir de los contenidos de una tabla en una base de datos.
SingleThreadModel	Este interfaz nos asegura que los servlets tratan una única petición al mismo tiempo

Tabla 1. Interfaces del paquete `javax.servlet`

<b>Clases</b>	
GenericServlet	Esta clase define y representa un servlet genérico independiente del protocolo utilizado.



ServletInputStream	Ofrece un flujo o canal de entrada (input stream) para leer datos binarios procedentes de la información de la petición de un cliente.
ServletOutputStream	Ofrece un flujo de salida (output stream) para enviar datos binarios al cliente.

Tabla 2. Clases del paquete javax.servlet

Excepciones	
ServletException	Define una excepción general que podrán lanzar los servlets
UnavailableException	Esta excepción será lanzada por un servlet para indicar que está permanentemente o temporalmente no disponible.

Tabla 3. Excepciones del paquete javax.servlet

Y dentro del paquete javax.servlet.http, dedicado a los servlets más concretos, es decir a los servlets HTTP, encontramos estos otros elementos de interés (Tabla 4 y Tabla 5), como se puede comprobar en este caso no se ofrece ninguna nueva excepción.

Interfaces	
HttpServletRequest	Hereda del interfaz ServletRequest y lo amplía para ofrecer información específica de las peticiones realizadas a servlets HTTP, como puede ser, recuperar el contenido de un formulario HTML.
HttpServletResponse	Hereda del interfaz ServletResponse y lo amplía para ofrecer funcionalidad específica del protocolo http a la hora de enviar respuestas al cliente, como por ejemplo, para establecer el tipo de documento que se le devuelve al cliente.
HttpSession	Este interfaz ofrece un mecanismo mediante el cual podemos identificar a un usuario a través de distintas peticiones de páginas (o servlets) dentro un sitio Web. Además permite almacenar información sobre cada usuario concreto.
HttpSessionBindingListener	Permite que se le notifique a un objeto cuando se le incluye en la sesión o se le destruye eliminándolo de la sesión.
HttpSessionContext	Este interfaz ha sido depreciado (deprecated). Es específico de la versión anterior de los servlets, es decir, de la especificación Java servlet 2.1 y ya no se utiliza.

Tabla 4. Interfaces del paquete javax.servlet.http

Clases	
Cookie	Esta clase permite crear y modificar cookies. Una cookie es una pequeña información enviada por un servlet a un navegador Web, almacenada por el navegador y enviada más tarde por el navegador al servidor Web.
HttpServlet	Esta clase abstracta representa un servlet HTTP y de ella deberán heredar todas las clases que quieran ser reconocidas y ejecutadas como servlets HTTP en un servidor Web.
HttpSessionBindingEvent	Este evento se envía a un objeto que implementa el interfaz HttpSessionBindingListener, cuando el objeto se ha añadido o eliminado de la sesión.
HttpUtils	Clase que contiene una colección de métodos que se utilizan como utilidades para los servlets.

Tabla 5. Clases del paquete javax.servlet.http

A lo largo del presente texto iremos bien en detenimiento las distintas clases e interfaces, sobre todo los del paquete javax.servlet.http.

## HTML, HTTP y los servlets

En este apartado vamos a comentar la relaciones entre HTML, el protocolo HTTP y los servlets.

El protocolo HTTP (HyperText Transfer Protocol) es el protocolo que ofrece el mecanismo de transporte para enviar datos entre el los servlets y el usuario final. El servlet recoge las peticiones HTTP del usuario y le envía respuestas HTTP.

El código HTML (HyperText Markup Lenguaje) define la forma en al que el usuario va observar el interfaz de nuestro servlet y también es responsable de recoger los datos de entrada. Además los servlets normalmente generarán código HTML para mostrar los resultados al cliente.

Inicialmente HTML se desarrolló para formatear texto, básicamente se trata de un lenguaje de formateo de texto, que especifica la forma del texto, color, distribución, etc. Desde algunas etiquetas de HTML podremos llamar a un servlet. Desde el código HTML podremos invocar un servlet de dos formas diferentes:

- Mediante la etiqueta <FORM>: esta etiqueta define la creación de un formulario dentro de HTML. Los formularios son el mecanismo más extendido dentro de la WWW para obtener información del usuario. Si necesitamos que nuestro servlet utilice la información ofrecida por el usuario en un formulario lo debemos indicar en la propiedad ACTION de la etiqueta <FORM>. Al pulsar el botón de envío del formulario, es decir, el botón SUBMIT, se realizará la llamada al servlet y se le enviará la información contenida en el formulario HTML. Dentro de la propiedad ACTION indicaremos una URL válida para nuestro servlet, una información más detallada sobre los formularios HTML y su relación con los servlets la veremos en el

capítulo dedicado al interfaz `javax.servlet.http.HttpServletRequest`, que es precisamente éste el interfaz que nos permite obtener los datos de las peticiones HTTP realizadas a los servlets. Si queremos enviar los datos de un formulario que recoge los nombres y apellidos de un usuario a un servlet llamado `ServletTrataDatos` escribiremos el siguiente código HTML (Código Fuente 2):

```
<FORM METHOD="POST" ACTION="servlet/ServletTrataDatos">
  <INPUT TYPE="TEXT" NAME="nombre" VALUE="">
  <INPUT TYPE="TEXT" NAME="apellidos" VALUE="">
  <INPUT TYPE="SUBMIT" NAME="enviar" VALUE="Enviar Datos">
</FORM>
```

Código Fuente 2. Formulario que invoca a un servlet

- Mediante la etiqueta `<A>`: esta etiqueta crea un enlace a una URL determinada, puede ser otra página Web, una imagen, etc., y también puede ser un enlace a un servlet. Al pulsar sobre el enlace se producirá la llamada al servlet. También le podemos pasar información al servlet a través de la etiqueta `<A>`, todo lo que debemos hacer es añadir más información después del nombre de nuestro servlet. Este es uno de los mecanismos que tenemos disponibles para pasarle parámetros a un servlet. A continuación podemos ver como llamamos a través de la etiqueta `<A>` a un servlet pasándole dos parámetros (Código Fuente 3).

```
<A HREF="servlet/ServletTrataDatos?nombre=Pepe&apellidos=Garcia">Pulse aquí</A>
```

Código Fuente 3. Enlace que hace referencia un servlet

Uno de los aspectos más importantes del lenguaje HTML en relación con los servlets, es que nos ofrece el interfaz de usuario que permite obtener información de nuestros clientes. Este interfaz de usuario se consigue gracias a la utilización de formularios HTML. Como acabamos de ver, el lenguaje HTML ofrece la etiqueta `<FORM>` para crear estos formularios. A través de estos formularios recogemos la información y se la enviaremos al servlet correspondiente.

En el capítulo dedicado al interfaz `HttpServletRequest` se detallará como interaccionan los formularios HTML y los servlets.

Si una de las funciones del lenguaje HTML es la de recoger la información del usuario, a través de formularios HTML, y enviársela a un servlet, nos podemos preguntar cómo puede llegar esta información. Esta información es transferida desde el navegador al servlet a través de cabeceras de petición del protocolo HTTP. La información enviada se formateará a una codificación denominada codificación URL.

El navegador creará una cabecera de petición HTTP a la que le acompañarán toda la información facilitada por el usuario en forma de parámetros, y también identificará la localización del propio servlet. Esta cabecera de petición HTTP es enviada por el navegador al servidor Web. El servidor recibirá la cabecera de petición HTTP y ejecutará el servlet correspondiente, el cual tendrá disponibles la información ofrecida en el formulario por el usuario.

El servlet se ejecuta, utiliza la información contenida en las cabeceras de petición HTTP, realiza las tareas que tenga descritas en su código, y devolverá la salida correspondiente a su ejecución. El servlet

generará los encabezados de respuesta HTTP que serán enviados al navegador Web, así como el contenido de la respuesta que se envía al cliente.

La forma más común de devolver encabezados de respuesta HTTP por parte servlet, es generar los encabezados de respuesta HTTP mínimos que serán requeridos para devolver una respuesta HTTP correcta al navegador.

Una vez que el servidor Web ha enviado al navegador las cabeceras de respuesta HTTP correspondientes, el navegador desplegará la información contenida en estas respuestas HTTP utilizando las cabeceras HTTP para averiguar la forma en la que deber desplegar el contenido de la respuesta HTTP.

## El servlet Hola Mundo

En este apartado vamos a comentar paso a paso el conocido ejemplo “Hola Mundo”, pero aplicado a un servlet. En un apartado anterior ya vimos un código similar, pero en este nuevo apartado no nos vamos a limitar a mostrar el código fuente, sino que vamos a comentar cada una de las líneas y además veremos como compilar un servlet, como realizar su llamada y dónde lo debemos situar para que el servidor Web Jakarta Tomcat pueda reconocerlo y ejecutarlo.

Un servlet lo implementamos de forma muy similar a cualquier otra clase Java, lo que les diferencia son los distintos paquetes que se deben importar y la clase de la que se hereda.

En primer lugar deberemos importar los paquetes relacionados con la especificación Java servlet 2.2. En este caso concreto del paquete `javax.servlet` vamos a utilizar la excepción `ServletException` y del paquete `javax.servlet.http` la clase abstracta `HttpServlet`, de la que deben heredar todos los servlets, y los interfaces `HttpServletRequest` y `HttpServletResponse`, que representan una petición HTTP y una respuesta HTTP respectivamente.

También se debe importar el paquete `java.io` para capturar excepciones de entrada/salida, es decir, excepciones `IOException`.

Una vez que hemos importado los paquetes correspondientes, creamos la declaración de la clase del servlet. La única condición que debe tener es que herede de la clase abstracta `HttpServlet`.

Lo normal es que el servlet implemente el método `doGet()` o `doPost()` o ambos, pertenecientes a la clase `HttpServlet`. Estos métodos se ejecutan cada vez que se realiza una petición HTTP mediante los métodos GET o POST del protocolo HTTP, es decir, cada vez que se invoca un servlet a través de un enlace o un formulario. Es bastante común sobrescribir los métodos `doGet()` y `doPost()` para que desde uno se llame al otro, y de esta forma siempre nos aseguremos que el servlet se ejecutará con independencia del método del protocolo HTTP que se haya utilizado para invocar al servlet.

Hasta ahora el código que llevaríamos en nuestro servlet sería el siguiente (Código Fuente 4):

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

public class ServletHolaMundo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
    }
}
```

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException{
    doGet(request,response);
}
}
```

Código Fuente 4

Este servlet de momento no hace mucho, ya que los cuerpos de los métodos `doGet()` y `doPost()` se encuentran vacíos, lo único que se hace es llamar al método `doGet()` desde el método `doPost()`.

Los métodos `doGet()` y `doPost()` reciben como parámetros un objeto `HttpServletRequest` y un objeto `HttpServletResponse`. En nuestro caso vamos a utilizar el segundo de ellos para devolver el saludo “Hola Mundo” al cliente que ha invocado al servlet desde el navegador. Para poder enviar información al usuario para que se muestre en el navegador debemos obtener el objeto `PrintWriter` que va a representar el flujo de salida del objeto `HttpServletResponse`. El objeto `PrintWriter` lo obtenemos a través del método `getWriter()` del interfaz `HttpServletResponse`.

Antes de escribir el mensaje de saludo utilizando el objeto `PrintWriter`, le vamos a indicar al navegador del cliente el tipo de información que le va a devolver el servlet, para ello vamos a utilizar otro método del interfaz `HttpServletResponse`, en este caso se trata del método `setContentType()`. Este método recibe como parámetro un objeto `String` que indica el tipo MIME que define el tipo de contenido que se envía al navegador Web, en este caso, se trata del tipo MIME `text/html`. En el tema dedicado al interfaz `HttpServletResponse` realizaremos un comentario detallado acerca de los tipos MIME.

Sobre el objeto `PrintWriter` lanzamos el método `println()` pasándole como argumento un objeto `String` con el código HTML que queremos enviar al navegador Web del cliente.

Si realizamos todas las sentencias comentadas hasta ahora en el método `doGet()` de nuestro servlet, el código completo de nuestro sencillo servlet es el que se puede observar en el Código Fuente 5.

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

public class ServletHolaMundo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.println("<html><body><h1>Hola Mundo</h1></body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 5

Ya hemos visto como construir este sencillo servlet, comentando paso a paso todas las líneas de código utilizadas, ahora vamos a comentar como compilar la clase del servlet y como podemos hacerlo disponible para el servidor Web Jakarta Tomcat 3.1.

Para compilar el servlet anterior debemos indicar en la variable de entorno CLASSPATH dónde se encuentran las clases e interfaces que implementan la especificación Java servlet 2.2, es decir, los paquetes javax.servlet y javax.servlet.http.

Si estamos utilizando como herramienta de desarrollo de Java el Java 2 SDK Enterprise Edition, no es necesario realizar ninguna modificación en la variable de entorno CLASSPATH, ya que el propio Java 2 SDK EE contiene estos dos paquetes. Sin embargo si estamos utilizando el Java 2 SDK Standard Edition, si que debemos modificar la variable de entorno CLASSPATH, ya que los paquetes de los servlets no se encuentran en el Java 2 SDK SE, sino que los ofrece el contenedor de servlets, en nuestro caso, Jakarta Tomcat.

Tomcat 3.1 ofrece un fichero JAR (Java Archive), llamado SERVLET.JAR que contiene los paquetes de la especificación Java servlet 2.2 y de la especificación JavaServer Pages 1.1. Por lo tanto debemos realizar la referencia correspondiente a este fichero de clases en el CLASSPATH. El fichero SERVLET.JAR se encuentra en el siguiente directorio de Tomcat 3.1, c:\jakarta-tomcat\lib.

Al añadir la variable de entorno CLASSPATH también es necesario indicar dónde se encuentran las clases que forman parte del propio contenedor de servlets y páginas JSP. Estas clases se encuentran en el fichero JASPER.JAR y en el fichero WEBSERVER.JAR, ambos dentro del directorio c:\jakarta-tomcat\lib. Tampoco debemos incluir el directorio actual como parte del CLASSPATH, esto se consigue añadiendo un punto (.) en el establecimiento de la variable de entorno.

Por lo tanto la variable de entorno CLASSPATH quedaría establecida de la siguiente forma:

```
set CLASSPATH=.;c:\jakarta-tomcat\lib\servlet.jar;c:\jakarta-tomcat\lib\jasper.jar;c:\jakarta-tomcat\lib\webserver.jar
```

Ahora debemos situar la clase del servlet (fichero .CLASS) en el lugar correcto de la estructura de directorios de Tomcat para que podamos invocar al servlet desde un navegador Web utilizando el protocolo HTTP.

No vamos a crear una aplicación nueva de Tomcat para situar nuestro servlet (esto lo veremos más adelante en siguientes capítulos), sino que vamos a utilizar la aplicación de ejemplos que nos ofrece Tomcat en su instalación, esta aplicación se corresponde con el directorio c:\jakarta-tomcat\webapps\examples. Dentro del directorio de la aplicación existe un directorio llamado web-inf que a su vez contiene un directorio llamado classes. Va a ser el directorio classes el que va a contener todas las clases correspondientes a los servlets de esta aplicación.

Por lo tanto deberemos copiar el fichero compilado de nuestro servlet al directorio c:\jakarta-tomcat\webapps\examples\web-inf\classes.

Ahora debemos arrancar el servidor Tomcat ejecutando el fichero STARTUP.BAT, una vez que se está ejecutando Tomcat escribiremos la siguiente URL en el navegador Web y obtendremos el resultado de la Figura 4, <http://localhost:8080/examples/servlet/ServletHolaMundo>.

Aquí podemos observar la forma que tenemos de invocar un servlet, como se puede comprobar hemos utilizado en la URL el nombre de la aplicación y el directorio servlet, que no existe en ningún lugar, ya que si recordamos anteriormente hemos copiado al clase de nuestro servlet en un directorio llamado web-inf\classes, no en un directorio servlet. Sin embargo el utilizar en la URL el directorio servlet es una convención.



Figura 4. Resultado del servlet Hola Mundo

La forma genérica de invocar a un servlet es:

```
http://servidor/aplicación/servlet/NombreClaseServlet
```

Se debe tener especial cuidado a la hora de hacer referencia a la clase del servlet, ya que se distingue entre mayúsculas y minúsculas, al igual que ocurre en todo el lenguaje Java, ya que es case-sensitive. De todas formas, como veremos más adelante, podemos registrar nombres para nuestros servlets para realizar llamadas distintas a las genéricas.

Con la compilación y ejecución de este sencillo servlet damos por concluido este capítulo. En el siguiente seguiremos tratando algunos conceptos básicos en torno a los servlets.





# Servlets: conceptos básicos

---

## Introducción

En este capítulo, como su nombre indica, vamos a ver una serie de conceptos básicos que presentan los servlets. Además se profundizará en algunos de los aspectos comentados en el capítulo anterior.

Antes de iniciar este capítulo vamos a recapitular sobre algunos aspectos vistos anteriormente, destacando los siguientes puntos, que son relativos a las distintas características que debe ofrecer una clase típica de un servlet:

- Debe importar los paquetes `javax.servlet`, `javax.servlet.http` y `java.io`.
- Debe heredar de la clase abstracta `HttpServlet`.
- Suele implementar los métodos `doGet()` y `doPost()` de la clase `HttpServlet`.
- La clase se debe situar en un directorio específico del servidor Web Jakarta Tomcat.
- El servlet es invocado de la siguiente forma genérica:  
`http://servidor/aplicacion/servlet/ClaseServlet`.

En los dos siguientes apartados comentaremos en detalle dos clases abstractas muy relacionadas con los servlets, por un lado veremos la clase `javax.servlet.http.HttpServlet` que representa a los servlets HTTP y de la que deberán heredar todos los servlets que se ejecuten en un servidor Web, y por otro lado veremos la clase `javax.servlet.GenericServlet`, que es la superclase de la clase `HttpServlet` y que

representa a los servlets de forma genérica e independiente del protocolo utilizado por el cliente para realizar las peticiones al servlet.

Se ha creído conveniente explicar estas clases debido a que podemos tener una visión más general y profunda de lo que es exactamente un servlet y en que lugar de la estructura de clase de la plataforma Java 2 Enterprise Edition se encuentran localizados, y de dónde provienen todos los métodos que utilizamos y sobrescribimos en nuestros servlets HTTP.

## La clase `HttpServlet`

Esta clase abstracta del paquete `javax.servlet.http` ofrece la implementación de los servlets HTTP, y por lo tanto todo servlet debe heredar de ella y sobrescribir al menos uno de sus métodos. La clase `HttpServlet` ofrece una serie de métodos que se corresponden con distintos tipos de peticiones del protocolo HTTP, y además ofrece un método que tiene que ver con el ciclo de vida de un servlet. El ciclo de vida de un servlet lo veremos con detalle en el apartado correspondiente.

Los métodos que podemos encontrar en la clase `HttpServlet` y que podemos sobrescribir en nuestro servlet son los siguientes:

- `void doDelete(HttpServletRequest req, HttpServletResponse res):` este método será invocado por el servidor (contenedor o motor de servlets) a través del método `service()` de la clase `HttpServlet`, para permitir que el servlet trate una petición DELETE del protocolo HTTP. La operación DELETE permite a un usuario eliminar un documento o página Web del servidor.
- `void doGet(HttpServletRequest req, HttpServletResponse res):` este método será invocado por el servidor a través del método `service()`, para permitir que el servlet trate una petición GET del protocolo HTTP. La operación GET es de las más comunes del protocolo HTTP y permite a un usuario realizar una petición de obtención de un documento o página Web alojados en el servidor. Este método es de los típicos que podemos encontrar en un servlet.
- `void doOptions(HttpServletRequest req, HttpServletResponse res):` al igual que los métodos anteriores, este método será invocado por el servidor a través del método `service()`, para permitir que el servlet trate una petición OPTIONS del protocolo HTTP. La petición OPTIONS determina que métodos del protocolo HTTP soporta el servidor. Este método no se suele utilizar.
- `void doPost(HttpServletRequest req, HttpServletResponse res):` método invocado por el servidor a través del método `service()`, para permitir que el servlet trate una petición POST. Este método es de los típicos que podemos encontrar en un servlet.
- `void doPut(HttpServletRequest req, HttpServletResponse res):` método invocado por el servidor a través del método `service()`, para permitir que el servlet trate una petición PUT del protocolo HTTP. La petición PUT permite al usuario enviar un fichero al servidor de forma similar a como se envía un fichero a través del protocolo FTP (File Transfer Protocol).
- `void doTrace(HttpServletRequest req, HttpServletResponse res):` este método también es invocado a través del método `service()` y permite al servlet tratar peticiones TRACE. Una petición TRACE devuelve las cabeceras enviadas junto con esta

petición, de vuelta al cliente, por lo que se utiliza para depuración. Este método no se suele sobrescribir en los servlets.

- `long getLastModified(HttpServletRequest req)`: devuelve en milisegundos la hora y la fecha en la que el objeto `HttpServletRequest` se modificó por última vez.
- `void service(HttpServletRequest req, HttpServletResponse res)`: recibe las peticiones estándar del protocolo HTTP y las redirecciona a los métodos `doXXX` adecuados definidos en la clase del servlet, se puede decir que es el punto común de entrada de todos los tipos de peticiones realizadas a un servlet. Desde el método `service()` se invocarán los distintos métodos `doXXX` de la clase `HttpServlet`. No es necesario ni recomendable sobrescribir este método. Este es el método que tiene que ver con el ciclo de vida definido por los servlets.
- `void service(ServletRequest req, ServletResponse res)`: este método dirige todas las peticiones que realizan los clientes al método `service()` anterior. No hay ninguna necesidad para sobrescribir este método.

Se debe señalar que todos estos métodos, menos el método `getLastModified()`, lanzan dos excepciones: `javax.servlet.ServletException` y `java.io.IOException`.

Como ya hemos comentado los métodos de la clase `HttpServlet` que se sobrescriben de manera más usual son los métodos  `doGet()` y  `doPost()`.

## La clase `GenericServlet`

Esta clase abstracta perteneciente al paquete `javax.servlet`, representa a los servlets de forma genérica de forma independiente al protocolo utilizado para recibir y servir las peticiones de los clientes. La clase `GenericServlet` implementa los interfaces `Servlet` y `ServletConfig`, también pertenecientes al paquete `javax.servlet`.

De esta clase hereda la clase `HttpServlet`, vista anteriormente, ya que debe acomodar las especificaciones de los servlets genéricos a los servlets HTTP, que por el momento son los únicos tipos de servlets que se utilizan realmente.

La clase `GenericServlet` ofrece una serie de métodos que tienen que ver con el ciclo de vida de los servlets, y también otros métodos que tienen que ver con la configuración del servlet, además permite llevar un registro de la actividad del servlet. En los apartados correspondientes trataremos en detalle el ciclo de vida de los servlets y la configuración de los mismos.

Veamos todos los métodos de la clase `GenericServlet` a continuación.

- `void destroy()`: este método pertenece al ciclo de vida definido para los servlets y es llamado por el contenedor de servlets (`servlet container`) para indicar que el servlet va a finalizar su servicio, es decir, el servlet va a ser destruido.
- `String getInitParameter(String nombre)`: este método devuelve un objeto `String` que contiene el valor del parámetro de inicialización cuyo nombre se le pasa por parámetro. Este método devolverá `null` si el parámetro no existe.
- `Enumeration getInitParameterNames()`: devuelve el nombre de todos los parámetros de inicialización del servlet como un objeto `java.util.Enumeration` que contiene

objetos String, uno por cada nombre del parámetro. Devolverá un objeto Enumeration vacío si el servlet no tiene parámetros de inicialización.

- `ServletConfig getServletConfig()`: este método devuelve el objeto `ServletConfig` que tiene asociado el servlet. El objeto `ServletConfig` es un objeto de configuración utilizado por el contenedor de servlets para pasar información a un servlet durante su proceso de inicialización.
- `ServletContext getServletContext()`: este método devuelve una referencia al objeto `ServletContext` en el que se está ejecutando el servlet. El interfaz `ServletContext` define un conjunto de métodos que el servlet puede utilizar para comunicarse con su contenedor de servlets, por ejemplo, para obtener el tipo MIME de un fichero, redirigir peticiones o escribir en un fichero de registro.
- `String getServletInfo()`: devuelve información relativa al servlet, como puede ser el autor, versión y copyright. Por defecto este método devuelve un objeto String vacío, si queremos devolver información relevante debemos sobrescribir este método.
- `String getServletName()`: devuelve el nombre de la instancia actual del servlet.
- `void init(ServletConfig config)`: este es otro método perteneciente al ciclo de vida de los servlets, es el opuesto al método `destroy()`. Este método es invocado por el contenedor de servlets para indicar que el servlet se ha puesto en servicio. Este método recibe como parámetro un objeto `ServletConfig` que representa la configuración de inicialización que se va a aplicar a este servlet, es decir, los parámetros de inicialización utilizados.
- `void init()`: este método es la segunda versión del método `init()`, y se utiliza cuando el servlet no posee parámetros de inicialización.
- `void log(String mensaje)`: este método escribe el mensaje indicado por parámetro en el fichero de registro (log) de los servlets, el mensaje va precedido del nombre del servlet que lo ha generado. En el caso del servidor Web Jakarta Tomcat, este fichero de registro se llama `SERVLET.LOG` y se encuentra en el directorio `c:\jakarta-tomcat\logs`.
- `void log(String mensaje, Throwable t)`: escribe un mensaje explicativo y un volcado de pila para una excepción `Throwable` especificada, en el fichero de registro correspondiente, precedido también por el nombre del servlet correspondiente.
- `void service(ServletRequest req, ServletResponse res)`: método perteneciente al ciclo de vida de los servlets, y que ya conocemos de la clase `HttpServlet`, ya que es en esta clase dónde es sobrescrito. Este método es invocado por el contenedor de servlets para permitir que el servlet responda a una petición.

Esta clase no la vamos a utilizar nunca de manera directa, sino que la utilizaremos a través de la clase `HttpServlet`, que es clase hija de la clase `GenericServlet`.

En los dos siguientes apartados vamos a tratar dos aspectos de los servlets que ya hemos ido adelantando en este capítulo, el ciclo de vida de los servlets y la configuración e inicialización de los servlets.

## El ciclo de vida de un servlet

En este apartado vamos a abordar la forma en la que los servlets son creados y destruidos en el contenedor de servlets. Cuando se crea un servlet, va a existir una única instancia de este servlet, que por cada petición que le realicen los usuarios del mismo creará un hilo de ejecución (thread) en el que se tratará el método `doGet()` o `doPost()` correspondiente. A continuación vamos a tratar todo este proceso en detalle.

Primero vamos a realizar un pequeño resumen acerca del ciclo de vida de un servlet para tener una visión más general cuando comentemos de forma detallada cada una de las partes del ciclo de vida.

Cuando el servlet se crea por primera vez se invoca el método `init()`, por lo tanto este método contendrá el código de inicialización del servlet.

Después de esto, cada petición realizada por un usuario sobre el servlet se traduce en un nuevo hilo de ejecución (thread) que realiza una llamada al método `service()`. Múltiples peticiones concurrentes normalmente generan múltiples hilos de ejecución que llaman de forma simultánea al método `service()`, aunque el servlet puede implementar un interfaz especial que permite que únicamente se pueda ejecutar un único hilo de ejecución al mismo tiempo.

El método `service()` invocará los métodos `doGet()` o `doPost()` o cualquier otro método `doXXX`, dependiendo del tipo de petición HTTP recibida.

Finalmente, cuando el servidor decide descargar el servlet, se ejecuta anteriormente el método `destroy()`.

Como se puede ver el ciclo de vida de un servlet sigue el siguiente esquema:

- Ejecución del método `init()` para inicialización del servlet.
- Sucesivas ejecuciones del método `service()` en distintos hilos de ejecución, que resultan en una llamada a un método `doXXX`.
- Ejecución del método `destroy()` para realizar labores de liberación de recursos.

Este es el ciclo de vida de un servlet comentado a grandes rasgos, a continuación vamos a mostrar con detenimiento cada una de las fases del ciclo de vida de un servlet.

El método `init()` es invocado cuando el servlet es creado y no es llamado de nuevo por cada petición que los usuarios realicen sobre el servlet. De esta forma es utilizado por inicializaciones que tiene lugar una única vez, en este aspecto es muy similar al método `init()` del ciclo de vida de los applets.

El servlet puede ser creado cuando un usuario utiliza por primera vez el servlet a través de la URL correspondiente o bien cuando se inicia la ejecución de servidor que contiene los servlets. La forma en la que se crea el servlet depende si el servlet se encuentra registrado en el servidor Web o no, si no es encontrado registrado en el servidor Web, el servlet se creará cuando el primer usuario lo invoque, pero si se encuentra registrado en el servidor se creará cuando se inicie la ejecución del servidor. El proceso de registro de un servlet en el servidor lo veremos más adelante en este texto.

Como ya hemos comentado anteriormente, el método `init()` ofrece dos versiones, una de ellas no recibe ningún parámetro y la otra versión recibe como parámetro un objeto `ServletConfig`. La primera versión se utiliza cuando el servlet no necesita leer ninguna configuración que puede variar entre distintos servidores. La definición de este método tiene el aspecto del Código Fuente 6.

```
public void init() throws ServletException{
    //Código de inicialización
}
```

Código Fuente 6

La segunda versión del método `init()` se utiliza cuando el servlet necesita obtener una configuración específica antes de inicializarse. Por ejemplo el servlet puede necesitar información sobre la configuración de la base de datos, ficheros de contraseñas, parámetros específicos del servidor, etc. La definición segunda versión de `init()` se puede comprobar en el Código Fuente 7.

```
public void init(ServletConfig config) throws ServletException{
    super.init(config);
    //Código de inicialización
}
```

Código Fuente 7

Como se puede observar en el Código Fuente 7, la primera línea del método `init()` consiste en invocar el método `init()` de la clase padre pasándole por parámetro el mismo objeto `ServletConfig`. En esta versión del método `init()` esta sentencia es siempre necesaria incluirla en el código fuente, ya que de esta forma la superclase del servlet también se inicializa de manera adecuada.

El interfaz `ServletConfig` ofrece un método llamado `getInitParameter()` que permite obtener el parámetro de inicialización del servlet cuyo nombre pasamos por parámetro como un `String`, se podría comparar con el método `getParameter()` que podemos utilizar dentro del método `init()` de un applet. En Tomcat podemos especificar los parámetros de inicialización de un servlet a través de un fichero XML (Extended Markup Language) llamado `WEB.XML`, que lo poseerán cada una de las aplicaciones Web del servidor, en los apartados siguientes veremos el concepto de aplicación Web y como utilizar el fichero `WEB.XML` para indicar los parámetros de inicialización de un servlet.

En este punto damos por finalizada la fase de inicialización del servlet, veamos que sucede después de esta fase.

Cada vez que el servidor recibe una petición de un servlet, ya creado, el servidor crea una nueva hebra o hilo de ejecución (thread) y llama al método `service()` del servlet. El método `service()` comprueba el tipo de petición HTTP (GET, POST, PUT, DELETE, etc.) y llama al método `doGet()`, `doPost()`, `doPut()`, `doDelete()`, etc., según corresponda en cada caso.

Si necesitamos un servlet que necesita tratar de forma idéntica las peticiones GET y POST, podemos caer en la tentación de sobrescribir el método `service()`, en lugar de implementar los métodos `doGet()` y `doPost()`. Pero esta solución, que se puede ver en el Código Fuente 8, no es la correcta, sino que es más adecuado hacer que el método `doPost()` llame al método `doGet()` o viceversa, como se puede observar en el Código Fuente 9.

```
public void service(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException{
    //Código del servlet
}
```

Código Fuente 8

```
public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException{
    //Código del servlet
}
public void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException{
    doGet(req, res);
}
```

Código Fuente 9

No sobrescribir el método `service()` permite que se pueda añadir soporte para nuevos servicios más tarde añadiendo la implementación de los métodos `doPut()`, `doTrace()`, etc.

Los métodos `doXXX` son los que contienen por lo tanto, todo el código que va a ejecutar el servlet. En la mayoría de los casos sólo nos interesarán peticiones GET/POST, por lo que únicamente deberemos sobrescribir los métodos `doGet()` y `doPost()` de la clase abstracta `HttpServlet`.

Al contrario que ocurría con el método `init()`, que se ejecutaba una única vez en el ciclo de vida de un servlet, los métodos `service()` y `doXXX` se ejecutarán múltiples veces durante el ciclo de vida de un servlet, según las peticiones que se realicen de los servlets.

En este punto es necesario realizar una aclaración. Normalmente, el servidor crea una única instancia de un servlet y crea un nuevo hilo de ejecución para cada petición de usuario, de esta forma, si se tienen múltiples peticiones concurrentes de un servlet, se tendrán también varios hilos de ejecución concurrentes.

Debido a esto los métodos `doGet()` y `doPost()` deben sincronizar el acceso a datos compartidos como pueden ser los atributos de un servlet, ya que múltiples hilos de ejecución pueden acceder a los mismos datos de manera simultánea, para ello se debe hacer uso de sentencia `synchronized` a la hora de acceder a objetos o información compartidos.

Una solución alternativa es la de no permitir los múltiples hilos de ejecución (`multithread`), de esta forma nos aseguramos que en un mismo instante únicamente hay un solo hilo de ejecución accediendo a la instancia del servlet. Para ello la clase de nuestro servlet debe implementar el interfaz `javax.servlet.SingleThreadModel`, como se puede observar en el Código Fuente 10.

```
public class MiServlet extends HttpServlet implements SingleThreadModel{
    .....
}
```

Código Fuente 10

Si nuestro servlet implementa el interfaz `SingleThreadModel`, se crearán varias instancias del servlet, pero cada una de ellas puede contener en un mismo espacio de tiempo un único hilo de ejecución, se crea un pool de instancias de la clase del servlet y las peticiones se irán sirviendo según se vayan quedando libres estas instancias. Las peticiones se irán encolando para esperar el turno en el que pueden ejecutar una instancia libre del servlet.

Esta aproximación no es recomendable, sobretodo para servlets que tienen peticiones muy frecuentes, ya que pueden afectar seriamente al rendimiento del servidor, repercutiendo entonces en el tiempo de

espera de las peticiones que se realicen al servlet, ya que estas peticiones se irán encolando hasta que vaya llegando su turno, es decir, no existe otra petición ejecutándose en el servlet en ese momento.

Ya hemos comentado dos de las tres fases de la vida de un servlet, hemos visto la inicialización, la ejecución, y ahora vamos a finalizar este apartado viendo la última fase del ciclo de vida de un servlet, es decir, la destrucción de un servlet.

El servidor que contiene los servlets puede decidir eliminar la instancia de un servlet por diversos motivos, puede ser porque se lo haya indicado directamente el administrador del servidor o porque el servlet lleve mucho tiempo inactivo. Antes de proceder a la descarga de la instancia del servlet, el contenedor de servlets invoca el método `destroy()` del servlet correspondiente.

El método `destroy()`, cuya definición se puede observar en el Código Fuente 11, permite al servlet realizar labores de liberación de recursos antes de ser destruido, es muy similar al método `destroy()` del ciclo de vida de los applets. En el método `destroy()` se suelen cerrar conexiones con bases de datos, finalizar hilos de ejecución, escribir cookies a disco o contadores, cerrar ficheros, y otras tareas de limpieza similares.

```
public void destroy() throws ServletException{  
    //Código de finalización  
}
```

Código Fuente 11

## Aplicaciones Web

Llegados a este punto del texto, se hace necesario definir lo que se entiende por aplicación Web, y como podemos crearlas en el servidor Jakarta Tomcat 3.1, el siguiente apartado estará muy relacionado con este tema.

De forma sencilla podemos definir una aplicación Web como una estructura de directorios determinada que contiene una serie de recursos (páginas HTML, páginas JSP, servlets, applets, imágenes, etc.) relacionados entre sí, y que desde el punto de vista de Tomcat, se puede establecer una serie de parámetros y configuraciones comunes.

El usuario final verá la aplicación Web como un subdirectorío de un sitio Web que le ofrece una funcionalidad determinada.

A continuación vamos a mostrar como crear una aplicación en el servidor Jakarta Tomcat. Las aplicaciones Web dentro de Tomcat se definen dentro de un fichero de configuración general del servidor llamado `SERVER.XML`, este fichero se encuentra en el directorio `c:\jakarta-tomcat\conf`. Como sospechará el lector, este fichero de configuración está definido en formato XML (Extensive Markup Language), no vamos a entrar a comentar el lenguaje XML, sino que simplemente vamos a comentar y describir las etiquetas que nos permiten definir una nueva aplicación en el servidor Tomcat.

Antes de seguir comentando como definir una aplicación en Tomcat en el fichero `SERVER.XML`, vamos a mostrar en el Código Fuente 12 el contenido del fichero `SERVER.XML` que viene por defecto junto con la instalación del servidor Jakarta Tomcat.



```

<?xml version="1.0" encoding="ISO-8859-1"?>

<Server>
  <!-- Debug low-level events in XmlMapper startup -->
  <xmlmapper:debug level="0" />

  <!-- This is quite flexible; we can either have a log file per
        module in Tomcat (example: ContextManager) or we can have
        one for Servlets and one for Jasper, or we can just have
        one tomcat.log for both Servlet and Jasper.

        If you omit "path" there, then stderr should be used.

        verbosityLevel values can be:
            FATAL
            ERROR
            WARNING
            INFORMATION
            DEBUG
        -->

  <Logger name="tc_log"
    path="logs/tomcat.log"
    customOutput="yes" />

  <Logger name="servlet_log"
    path="logs/servlet.log"
    customOutput="yes" />

  <Logger name="JASPER_LOG"
    path="logs/jasper.log"
    verbosityLevel = "INFORMATION" />

  <!-- Add "home" attribute if you want tomcat to be based on a different
directory
        "home" is used to create work and to read webapps, but not for libs or
CLASSPATH.
        Note that TOMCAT_HOME is where tomcat is installed, while ContextManager
home is the
        base directory for contexts, webapps/ and work/
    -->
  <ContextManager debug="0" workDir="work" >
    <!-- ContextInterceptor className="org.apache.tomcat.context.LogEvents" / -
->
    <ContextInterceptor className="org.apache.tomcat.context.AutoSetup" />
    <ContextInterceptor className="org.apache.tomcat.context.DefaultCMSetter"
/>
    <ContextInterceptor
className="org.apache.tomcat.context.WorkDirInterceptor" />
    <ContextInterceptor className="org.apache.tomcat.context.WebXmlReader" />
    <ContextInterceptor
className="org.apache.tomcat.context.LoadOnStartupInterceptor" />
    <!-- Request processing -->
    <RequestInterceptor className="org.apache.tomcat.request.SimpleMapper"
debug="0" />
    <RequestInterceptor
className="org.apache.tomcat.request.SessionInterceptor" />
    <RequestInterceptor className="org.apache.tomcat.request.SecurityCheck" />
    <RequestInterceptor className="org.apache.tomcat.request.FixHeaders" />

    <Connector className="org.apache.tomcat.service.SimpleTcpConnector">
      <Parameter name="handler"
value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
      <Parameter name="port" value="8080"/>
    </Connector>

```

```

        <Connector className="org.apache.tomcat.service.SimpleTcpConnector">
            <Parameter name="handler"
value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter name="port" value="8007"/>
        </Connector>

        <!-- example - how to override AutoSetup actions -->
        <Context path="/examples" docBase="webapps/examples" debug="0"
reloadable="true" >
        </Context>
        <!-- example - how to override AutoSetup actions -->
        <Context path="/" docBase="webapps/ROOT" debug="0" reloadable="true" >
        </Context>

        <Context path="/test" docBase="webapps/test" debug="0" reloadable="true" >
        </Context>

    </ContextManager>
</Server>

```

Código Fuente 12. Contenido del fichero SERVER.XML

No debemos asustarnos con tanto código XML, por ahora sólo nos vamos a fijar en las etiquetas `<Context>` que aparecen al final del fichero SERVER.XML. Para definir una aplicación dentro del fichero SERVER.XML debemos utilizar la etiqueta `<Context>`, existirán tantas etiquetas `<Context>` como aplicaciones definidas en el servidor Tomcat. Debido a esto, las aplicaciones Web en el entorno de Tomcat también se pueden denominar contextos.

El aspecto de la etiqueta `<Context>` lo podemos ver en el Código Fuente 13 para la aplicación que se ofrece de ejemplo (examples) con el servidor Jakarta Tomcat.

```

<Context path="/examples" docBase="webapps/examples" debug="0" reloadable="true" >

```

Código Fuente 13

Vamos a comentar las propiedades principales que posee la etiqueta `<Context>`:

- **path:** la propiedad `path` indica el nombre de directorio que va a recibir la aplicación en su URL correspondiente, así por ejemplo, para acceder a la aplicación `examples` debemos escribir la siguiente URL en el navegador Web: `http://localhost:8080/examples`.
- **docBase:** la propiedad `docBase` de la etiqueta `<Context>` indica la localización física de los ficheros que forman parte de la aplicación. En este caso los ficheros que forman la aplicación se identifican con un camino relativo, que es un subdirectorío del directorío de instalación de Tomcat y que es dónde por defecto se encuentran todas las aplicaciones Web de Tomcat, este directorío se llama `c:\jakarta-tomcat\webapps`, por lo tanto la ruta física completa del directorío físico de la aplicación `examples` es `c:\jakarta-tomcat\webapps\examples`.
- **debug:** mediante `debug` indicamos el nivel de detalle, de cero a nueve, de los diferentes mensajes de depuración que se registren, cero es el menor nivel de detalle y nueve es el que ofrece mayor información.
- **reloadable:** la propiedad `reloadable` de la etiqueta `<Context>` puede tomar un valor booleano, que indicará si el servidor Tomcat detecta automáticamente las modificaciones que se realicen

en los servlets de la aplicación. Es muy recomendable dar el valor de verdadero (true) a esta propiedad en servidores de desarrollo, ya que de esta forma podremos realizar modificaciones sobre los servlets sin tener que reiniciar el servidor de nuevo. De todas formas esta operación es costosa y como tal requiere un tiempo extra en la ejecución del servidor.

De esta forma, si queremos crear una aplicación nueva llamada ejemplos, cuyos ficheros se encuentran en la ruta física c:\work\ejemplos, y además queremos utilizar el menor detalle en la depuración y que se recarguen los servlets de forma automática al modificarse, añadiremos la siguiente etiqueta <Context> (Código Fuente 14) en el fichero SERVER.XML

```
<Context path="/ejemplos" docBase="c:\work\ejemplos" debug="0" reloadable="true" >
</Context>
```

Código Fuente 14

Para probar si la aplicación funciona correctamente y se encuentra disponible para los usuarios, crearemos los subdirectorios web-inf\classes a partir de la ruta física de la aplicación ejemplos. Cuando retomemos el concepto de aplicación Web y la estructura de directorios que contiene describiremos en más detalle la función de cada directorio y fichero de la aplicación, de momento vamos a decir sencillamente que el directorio classes de una aplicación va a contener las clases de los servlets de esa aplicación.

En el directorio classes vamos a copiar la clase del servlet ServletHolaMundo que construimos en el capítulo anterior, de esta forma podremos ejecutar el servlet desde la aplicación ejemplos. Ahora no queda más que invocar a este servlet desde la nueva aplicación, para ello escribiremos la siguiente URL en el navegador. <http://localhost:8080/ejemplos/servlet/ServletHolaMundo>, el resultado se puede ver en la Figura 5.

En el siguiente apartado iremos viendo algunos detalles más sobre las aplicaciones Web, sobretodo lo que tiene que ver con la inicialización de los servlets que contiene la aplicación Web.

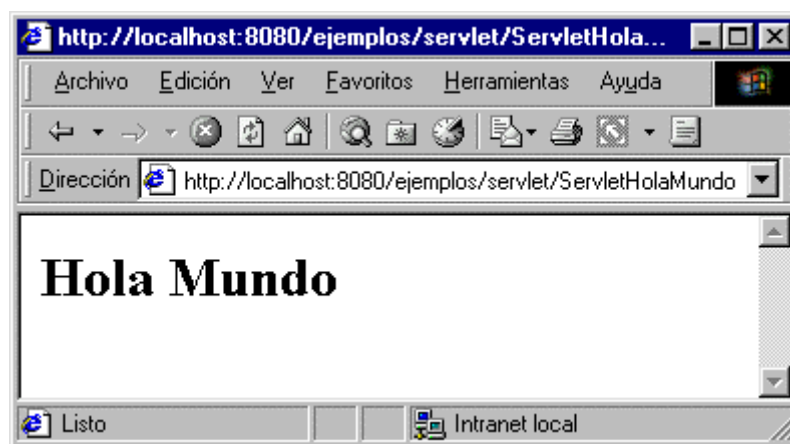


Figura 5. Ejecución de un servlet en la nueva aplicación

## Parámetros de inicialización

En un apartado anterior ya vimos el cometido del objeto ServletConfig que le pasábamos por parámetro al método init() de un servlet. Este objeto nos permitía establecer la configuración inicial de

un servlet cuando se instanciaba en el contenedor de servlets, esta configuración se realiza a través de parámetros de inicialización.

El interfaz `ServletConfig` ofrece una serie de métodos que nos permiten manejar los parámetros de inicialización de un servlet. Los métodos del interfaz `ServletConfig` son:

- `String getInitParameter(String nombreParametro)`: este método devuelve un objeto `String` que representa el valor del parámetro de inicialización cuyo nombre le pasamos por parámetro. Si el parámetro no existe se devolverá `null`.
- `Enumeration getInitParameterNames()`: devuelve el nombre de todos los parámetros de inicialización del servlet como un objeto `java.util.Enumeration` que contiene objetos `String`, uno por cada nombre del parámetro. Devolverá un objeto `Enumeration` vacío si el servlet no tiene parámetros de inicialización.
- `ServletContext getServletContext()`: este método devuelve una referencia al objeto `ServletContext` en el que se está ejecutando el servlet. El interfaz `ServletContext` define un conjunto de métodos que el servlet puede utilizar para comunicarse con su contenedor de servlets, por ejemplo, para obtener el tipo MIME de un fichero, redirigir peticiones o escribir en un fichero de registro.
- `String getServletName()`: devuelve el nombre de la instancia actual del servlet.

Todos estos métodos ya los hemos comentado a la hora de tratar la clase `GenericServlet`, no se debe olvidar que la clase abstracta `GenericServlet` implementaba el interfaz `ServletConfig`.

Vamos a realizar un sencillo ejemplo que va a consistir en un servlet que va a mostrar un número de veces un mensaje determinado. Tanto el número de veces que se debe mostrar el mensaje, como el texto del mensaje a mostrar van a ser dos parámetros de inicialización de nuestro servlet, que los obtendremos a partir del objeto `ServletConfig` del método de inicio del servlet `init()`.

Veamos primero el código fuente completo de este servlet (Código Fuente 15), y a continuación lo comentaremos.

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

public class MuestraMensaje extends HttpServlet {
    private String mensaje;
    private String mensajePordefecto = "Hola";
    private int repeticiones = 1;

    public void init(ServletConfig config) throws ServletException {
        //siempre debemos añadir esta línea para esta versión del método init()
        super.init(config);
        mensaje = config.getInitParameter("mensaje");
        if (mensaje == null) {
            mensaje = mensajePordefecto;
        }
        try {
            String cadenaRepeticiones = config.getInitParameter("repeticiones");
            //debemos convertirlo a un entero, ya que getInitParameters() siempre
            //devuelve objetos String
            repeticiones = Integer.parseInt(cadenaRepeticiones);
        } catch (NumberFormatException ex) {}
    }
}
```

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String titulo = "El Servlet muestra mensajes";
    out.println("<html><head><title>" + titulo + "</title></head>" +
        "<BODY BGCOLOR=\"#FDF5E6\">\n" +
        "<H1 ALIGN=CENTER>" + titulo + "</H1>");
    for(int i=0; i<repeticiones; i++) {
        out.println(mensaje + "<BR>");
    }
    out.println("</BODY></HTML>");
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}
```

Código Fuente 15

Como se puede comprobar este servlet posee tres métodos, el método `init()` en el que se produce la inicialización del servlet a través del objeto `ServletConfig`, el método `doGet()` que es el encargado de mostrar el mensaje especificado el número de veces indicado, y por último el método `doPost()` que simplemente llama al método `doGet()`.

El servlet posee varios atributos, por un lado tiene dos objetos de la clase `String` llamados `mensaje`, y `mensajePordefecto`, el primero de ellos va a representar el valor del parámetro de inicialización, y el segundo el mensaje que se mostrará en el caso de no existir el parámetro de inicialización llamado `mensaje`. Y por otro lado el servlet tiene una variable de tipo `int` llamada `repeticiones` que va a contener el valor del parámetro de inicialización del mismo nombre. La variable `repeticiones` se inicializa a uno, por lo tanto este será su valor por defecto.

Como ya hemos comentado es el método `init()` el encargado de establecer los parámetros de inicialización del servlet. Después de la primera sentencia, obligada en esta versión del método `init()`, se recupera el parámetro de inicialización llamado `mensaje` a través del método `getInitParameter()` del interfaz `ServletConfig`. En el caso de que el parámetro sea nulo (`null`), es decir, en el caso de que no exista, se establecerá el valor por defecto. A continuación se obtiene el parámetro `repeticiones`, al que se le debe aplicar una conversión de tipos mediante el método `Integer.parseInt()`, ya que el método `getInitParameter()` del interfaz `ServletConfig` siempre devuelve el valor de los parámetros de inicialización como objetos `String`. Si se produce algún error de conversión se lanzará una excepción `NumberFormatException`, en el tratamiento de esta excepción no se va a hacer nada, porque si falla la ejecución de la sentencia `repeticiones = Integer.parseInt(cadenaRepeticiones)`, el valor de la variable `repeticiones` será el asignado por defecto, es decir, uno.

Esto es todo lo que hace el método `init()`, el método `doGet()` se encargará de mostrar el mensaje correspondiente el número de veces indicado. Para ello utilizará el objeto `PrintWriter` que representa la salida que se le enviará al usuario. Antes de escribir el mensaje mediante un bucle `for`, indica el tipo de documento que va a devolver al usuario mediante el método `setContentType()` del interfaz `HttpServletResponse`, en este caso se indica una página HTML (`text/html`). Junto con la repetición del mensaje también se envía un código HTML mínimo como la cabecera de título y un encabezado de tipo `H1`.

Una vez compilado este servlet vamos a probarlo en la aplicación que creamos previamente, es decir, la aplicación ejemplos. Para ello debemos copiar la clase del servlet al directorio `c:\work\ejemplos`

\web-inf\classes, y para ejecutarlo desde el navegador escribiremos la URL siguiente, y obtendremos un resultado como el de la Figura 6, <http://localhost:8080/ejemplos/servlet/MuestraMensaje>.



Figura 6. Ejecución del servlet con parámetros de inicialización

Como se puede comprobar se ha utilizado los valores por defecto de los parámetros de inicialización, ya que ¿dónde hemos especificado el valor de los parámetros?, pues de momento en ningún lugar. A continuación vamos a mostrar como definir los parámetros de inicialización de un servlet determinado dentro del servidor Jakarta Tomcat 3.1.

Para iniciar los parámetros de inicialización de un servlet y para otras tareas que tienen que ver con la configuración específica de cada una de las aplicaciones Web definidas en Tomcat, el servidor Tomcat utiliza un fichero XML llamado WEB.XML, que se debe situar en el directorio web-inf de cada aplicación cuando sea necesario, como es el caso que nos ocupa.

Si no especificamos un fichero de configuración de la aplicación Web, se utilizará el fichero WEB.XML, que se encuentra en el directorio c:\jakarta-tomcat\conf, que define los valores por defecto de todas las aplicaciones Web. Pero si construimos un fichero WEB.XML en el directorio web-inf de la aplicación Web, se combinarán los valores definidos en el fichero WEB.XML general del servidor con el de la aplicación Web.

El fichero WEB.XML lo vamos a utilizar en este ejemplo para definir los parámetros de inicialización de un servlet, pero este fichero también posee las siguientes funciones:

- Ofrecer una descripción de la aplicación Web.
- Realizar una correspondencia de nombre de servlets con URLs.
- Establecer la configuración de la sesión.
- Mapeo de librerías de etiquetas.
- Informar de los tipos MIME soportados.
- Establecer la página por defecto de la aplicación Web.

Muchos de los conceptos comentados en estos puntos no los conocerá el lector, pero que no se preocupe el lector, cada una de las funciones del fichero WEB.XML las iremos descubriendo a lo

largo del texto. De momento vamos a comentar como definir los parámetros de inicialización de un servlet, pero antes de esto tenemos que ver como asignar un nombre a un servlet, ya que los parámetros de inicialización se le asignan a un servlet a través de su nombre.

Primero debemos asociar un nombre con el fichero de la clase del servlet, una vez hecho esto asociamos el nombre del servlet con el parámetro de inicialización, para ello utilizaremos etiquetas al estilo del fichero SERVER.XML que nos permitía definir aplicaciones Web del servidor. Todas las etiquetas XML que vamos a utilizar se encuentran englobadas por una etiqueta general llamada `<web-app>` y que contiene toda la configuración para la aplicación Web en la que se encuentra el fichero WEB.XML. La estructura inicial de este fichero es la que se muestra en el Código Fuente 16.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>

</web-app>
```

Código Fuente 16

Dentro de una etiqueta `<servlet>` indicaremos la correspondencia entre el nombre que se le va a dar al servlet y el fichero que contiene la clase del mismo y los parámetros de inicialización que se van a utilizar en el servlet, además de otros aspectos de configuración del servlet. Es precisamente dentro de esta etiqueta dónde podremos indicar si el servlet se debe instanciar cuando se inicie la ejecución del servidor Tomcat.

Mediante la etiqueta `<servlet-name>` indicamos el nombre que se le va a asignar al servlet. El valor que se indique en esta etiqueta puede ser utilizado en la URL que invoca al servlet, así por ejemplo si al servlet `MuestraMensaje` le asignamos el nombre `mensaje`, podremos invocarlo desde el navegador con la URL `http://localhost:8080/ejemplos/servlet/mensaje`. Como se puede comprobar debemos seguir utilizando el directorio `servlet`, en el siguiente apartado veremos como asignar una URL alternativa a un servlet determinado.

Para indicar el servlet al que vamos a asignar el nombre especificado en la etiqueta `<servlet-name>`, utilizaremos la etiqueta `<servlet-class>`. En esta etiqueta se indica el nombre de la clase del servlet, así si tomamos el ejemplo anterior, deberíamos especificar la clase `MuestraMensaje`.

Tanto la etiqueta `<servlet-name>` como `<servlet-class>` son etiquetas obligatorias que forman parte de la etiqueta `<servlet>`.

De esta forma ya tendríamos un servlet determinado identificado mediante un nombre. En este momento el fichero WEB.XML tiene el aspecto del mostrado en el Código Fuente 17.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
```

```

<servlet>
  <servlet-name>
    mensaje
  </servlet-name>
  <servlet-class>
    MuestraMensaje
  </servlet-class>
</servlet>

</web-app>

```

Código Fuente 17

Para asignar ahora los parámetros de inicialización al servlet hacemos uso de otras etiquetas llamadas `<init-param>`, `<param-name>` y `<param-value>`. La etiqueta `<init-param>` indica que se va a definir un parámetro para el servlet actual, es decir, en servlet que se encuentra definido entre las etiquetas `<servlet>` y `</servlet>`.

La etiqueta `<init-param>` posee varios subelementos que comentamos a continuación:

- La etiqueta `<param-name>` en la que indicamos el nombre del parámetro de inicialización del servlet. El nombre del parámetro es case-sensitive, es decir, se distingue entre minúsculas y mayúsculas, esta norma la podemos aplicar como general a todos los elementos identificativos de los ficheros XML de configuración del servidor Tomcat y sus aplicaciones.
- La etiqueta `<param-value>` en el que se indica el valor del parámetro. Estos valores siempre se van a recuperar como objeto String, el servlet será encargado de convertir el valor del parámetro al tipo correspondiente.
- La etiqueta `<description>` permite especificar una descripción del parámetro, esta etiqueta se utiliza a nivel de documentación.

De esta forma si queremos inicializar el servlet asignándole al parámetro repeticiones el valor de 4 y al parámetro mensaje el valor de la cadena "Hola, que tal", el fichero de configuración de la aplicación Web, tendrá el aspecto del Código fuente 18.

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>
      mensaje
    </servlet-name>
    <servlet-class>
      MuestraMensaje
    </servlet-class>
    <init-param>
      <param-name>repeticiones</param-name>
      <param-value>4</param-value>
      <description>Número de veces que se repite el mensaje</description>
    </init-param>
    <init-param>
      <param-name>mensaje</param-name>
      <param-value>Hola, que tal</param-value>
    </init-param>
  </servlet>

```



```
<description>Texto del mensaje</description>
</init-param>
</servlet>

</web-app>
```

Código fuente 18

Si ejecutamos el servlet con los parámetros de inicialización asignados en el fichero WEB.XML, obtendremos el resultado de la Figura 7.

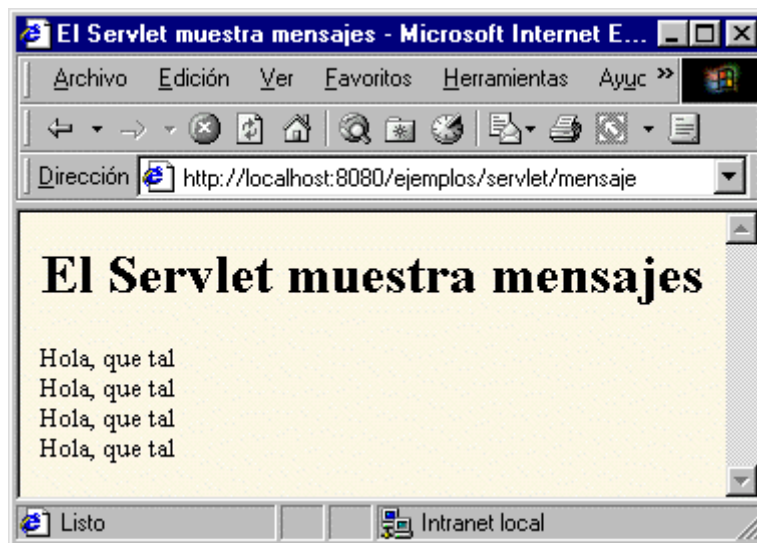


Figura 7. Servlet que recupera los parámetros de inicialización

Dentro de la etiqueta `<servlet>` también podemos indicar que el servlet se cree, y ejecute el método `init()`, al iniciar la ejecución del servidor Tomcat. Para ello utilizaremos la etiqueta `<load-on-startup>`, si a este etiqueta no se le facilita ningún valor, el servidor instanciará y cargará el servlet cuando le sea posible al iniciarse. Pero si deseamos que se instancie el servlet atendiendo a una prioridad con respecto al resto de los servlets de la aplicación Web, especificaremos un entero como valor de la etiqueta. Cuanto mayor sea este número menor será la prioridad de creación del servlet, de esta forma primero se crearán los servlets con prioridad 1, luego los de prioridad 2, y así con el resto.

En el siguiente apartado vamos a ver otras etiquetas que se sitúan dentro de la etiqueta `<servlet>` y que nos van a servir para definir otros aspectos de la configuración de un servlets, relativos a la forma en la que podemos invocarlos y a la correspondencia de los servlet con patrones de URLs.

## Organización de servlets

Como ya hemos comentado las clases de los servlets (ficheros `.CLASS`) se deben situar en el directorio `web-inf\classes` de la aplicación Web desde la que los vamos a utilizar.

Cuando tenemos unos pocos servlets no existe un problema de organización excesivo, pero cuando la aplicación Web se vuelve más compleja, podemos tener un gran número de servlets dentro del directorio `web-inf\classes`, por lo que la organización de los mismos puede convertir en un problema. Por lo tanto lo recomendable en estos casos es organizar los servlets mediante el mecanismo de paquetes que ofrece el lenguaje Java.

Para agrupar los servlets en paquetes simplemente debemos crear a partir del subdirectorio `classes` la estructura de directorios que coincide con el nombre del paquete. Así por ejemplo, si el servlet `MuestraMensaje` queremos que pertenezca al paquete `ejemplos.servlet`, debemos crear el subdirectorio `classes\ejemplos\servlets` en la aplicación Web correspondiente, y copiar la clase del servlet. El código del servlet sería idéntico, lo único que se debe añadir es al principio la línea que se muestra en el Código Fuente 19, para indicar el paquete al que pertenece el servlet.

```
package ejemplos.servlets;
```

Código Fuente 19

Ahora para hacer referencia a este nuevo servlet incluido en un paquete determinado, debemos utilizar el nombre de la clase completa del servlet, es decir, el nombre de la clase precedido del nombre de los paquetes y subpaquetes. En este caso sería `ejemplos.servlets.MuestraMensaje`, y por lo tanto si queremos invocar al servlet desde el navegador Web debemos utilizar la URL <http://localhost:8080/ejemplos/servlet/ejemplos.servlets.MuestraMensaje>. El nombre completo de la clase del servlet también se deberá utilizar en el fichero de configuración de la aplicación Web, es decir el fichero `WEB.XML`, para hacer referencia al servlet.

En el apartado anterior vimos como podíamos asignar un nombre a un servlet, y cómo este mismo nombre se puede utilizar en la URL que invoca al servlet. Pero el fichero de configuración `WEB.XML` nos permite ir más allá, pudiendo definir nosotros mismos la URL específica del servlet, sin que tenga que ser de la forma genérica `http://servidor/aplicación/servlet/nombreClase(o nombreServlet)`.

Para ello disponemos de etiqueta `<servlet-mapping>`. Esta etiqueta nos permite asignar a un servlet, identificado por su nombre asignado en la subetiqueta `<servlet-name>` de la etiqueta `<servlet>`, una URL determinada. La etiqueta `<servlet-mapping>` contiene dos subetiquetas `<servlet-name>` y `<url-pattern>`, que pasamos a comentar a continuación:

- En la etiqueta `<servlet-name>` indicamos el nombre del servlet al que queremos asociar con una URL determinada, o bien con un patrón de URL, como veremos más adelante.
- En la etiqueta `<url-pattern>` especificamos el patrón de la URL que queremos asignar al servlet. El contenido del cuerpo de esta etiqueta puede ser una cadena sencilla que indique una ruta más o menos compleja de la URL que se va a asociar con el servlet correspondiente, o bien una cadena con caracteres especiales que sirvan como un patrón.

Así por ejemplo si queremos invocar al servlet `MuestraMensaje` utilizando la URL <http://localhost:8080/ejemplos/miServlet>, deberemos añadir el Código Fuente 20 en el fichero `WEB.XML` de la aplicación `ejemplos`, justamente después del cierre de la etiqueta `<servlet>` que habíamos utilizado para definir el nombre del servlet y sus parámetros de inicialización.

```
<servlet-mapping>
  <servlet-name>mensaje</servlet-name>
  <url-pattern>/miServlet</url-pattern>
</servlet-mapping>
```

Código Fuente 20

Como se puede comprobar ya no se utiliza el directorio servlet en ningún lugar de la URL que utilizamos para invocar el servlet.

Se debe señalar que para que las modificaciones en el fichero WEB.XML tengan efecto, se debe parar el servidor Tomcat y volver a iniciarlo, para que el contenedor de servlets tenga en cuenta la nueva configuración de la aplicación Web.

Ahora nuestro fichero WEB.XML de la aplicación ejemplos presenta el aspecto del Código Fuente 21.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>
      mensaje
    </servlet-name>
    <servlet-class>
      MuestraMensaje
    </servlet-class>
    <init-param>
      <param-name>repeticiones</param-name>
      <param-value>4</param-value>
      <description>Número de veces que se repite el mensaje</description>
    </init-param>
    <init-param>
      <param-name>mensaje</param-name>
      <param-value>Hola, que tal</param-value>
      <description>Texto del mensaje</description>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>mensaje</servlet-name>
    <url-pattern>/miServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Código Fuente 21

Ahora podremos invocar al servlet MuestraMensaje de tres formas distintas, a continuación indicamos las distintas URLs:

- <http://localhost:8080/ejemplos/servlet/MuestraMensaje>. Esta es la URL por defecto del servlet. Aunque en este caso no se tomarán los parámetros de inicialización del servlet, para que se tengan en cuenta estos parámetros se debe utilizar la URL con el nombre del servlet, que es el siguiente punto.
- <http://localhost:8080/ejemplos/servlet/mensaje>. Esta es la URL que utiliza el nombre del servlet que se ha definido, y que por lo tanto utilizará los parámetros de inicialización definidos en el fichero WEB.XML.

- `http://localhost:8080/ejemplos/miServlet`. Esta otra forma de invocar al servlet utiliza la URL que se le ha asignado al mismo, en este caso también se utilizan los parámetros de inicialización definidos para el servlet.

En el ejemplo anterior la etiqueta `<url-pattern>` contenía en su cuerpo únicamente una cadena que identificaba un directorio a partir de la URL de la aplicación actual, pero esto puede ser más complejo teniendo la posibilidad de haber especificado una estructura de directorios más compleja, como podría haber sido `miServlet/primeros/uno`. De esta forma el servlet `MuestraMensaje` podría haber sido invocado mediante la URL `http://localhost:8080/ejemplos/miServlet/primeros/uno`.

También es posible el uso de asteriscos (\*) como caracteres especiales. El asterisco se puede utilizar como un comodín en la parte final de la URL que se va a asignar al servlet, así por ejemplo si en la etiqueta `<url-pattern>`, utilizamos la cadena `miServlet/mensajes/*`, todas las URLs que comiencen de la forma `http://localhost:8080/ejemplos/miServlet/mensajes`, invocarán el servlet `MuestraMensaje`. De esta forma si utilizamos la URL `http://localhost:8080/ejemplos/miServlet/mensajes/la/898989/8898`, invocaremos al ya conocido servlet `MuestraMensaje`.

Otro uso avanzado que podemos realizar de la etiqueta `<url-pattern>` es la de asignar los ficheros con una extensión determinada a un servlet determinado. Para ello en el cuerpo de la etiqueta `<url-pattern>` utilizaremos una cadena compuesta de un asterisco, un punto y el nombre de la extensión de los ficheros que queremos que se correspondan con la invocación del servlet. Así por ejemplo, si queremos que al utilizar una URL con un fichero que posea la extensión `.ser` se invoque al servlet `MuestraMensaje` deberemos añadir el Código Fuente 22 al fichero `WEB.XML`.

```
<servlet-mapping>
  <servlet-name>mensaje</servlet-name>
  <url-pattern>*.ser</url-pattern>
</servlet-mapping>
```

Código Fuente 22

Adelantamos que este es el mecanismo utilizado para invocar las JavaServer Pages (JSP), todos los ficheros con extensión `.jsp` se mapean o se corresponden con el servlet que realiza las funciones de compilador de páginas JSP.

De momento damos por terminado los comentarios acerca del fichero `WEB.XML`, de todas formas en el capítulo dedicado a Jakarta Tomcat volveremos a retomar tanto el fichero `SERVER.XML` como el fichero `WEB.XML`, para profundizar en la configuración del servidor y en las aplicaciones Web.

En el siguiente capítulo trataremos en profundidad uno de los principales interfaces que podemos utilizar en un servlet, se trata del interfaz `javax.servlet.http.HttpServletRequest`, que nos permite acceder a la información que contiene la petición HTTP de un usuario, es decir, nos permite acceder a la entrada del usuario.

# Servlets: el interfaz `HttpServletRequest`

---

## Introducción

Como vimos en el capítulo anterior, los distintos métodos `doXXX` de la clase `HttpServlet` recibían por parámetro objetos `HttpServletRequest`. El interfaz `HttpServletRequest` nos permite obtener la información que envía el usuario al realizar una petición de un servlet, esta información puede ser muy variada, puede ser desde encabezados de petición del protocolo HTTP, cookies enviadas por el usuario o los datos de un formulario.

El contenedor de servlets creará un objeto `HttpServletRequest` y se lo pasará por parámetro a los distintos métodos de servicio de un servlet, como pueden ser `doGet()`, `doPost()` o `doPut()`.

En este capítulo vamos a tratar de forma completa el interfaz `HttpServletRequest`, veremos a través de ejemplos las distintas formas que tenemos de obtener la información enviada por el cliente. Aunque no sólo trataremos este interfaz, sino que veremos también el interfaz del que hereda llamado `javax.servlet.ServletRequest`. También se realizará una comparativa con las variables de entorno de los scripts CGI.

## El interfaz `HttpServletRequest`

En este apartado vamos a comentar de forma breve todos los métodos que pone a nuestra disposición este interfaz, a lo largo de todo este capítulo iremos detallando los más interesantes.

Cuando tratemos más adelante en este mismo texto las páginas JSP (JavaServer Pages), veremos que este interfaz se corresponde con un objeto integrado de JavaServer Pages, este objeto es el objeto

request. Si el lector es conocedor de las páginas ASP (Active Server Pages), le será familiar el objeto integrado request, ya que ASP también lo ofrece, además en JSP tiene el mismo sentido que en ASP.

Veremos que esta correspondencia entre objetos que se utilizan en servlets y objeto integrados de JSP no es una mera coincidencia, y que además se repite en diversos casos. Aunque por el momento no vamos a adelantar más.

El interfaz `javax.servlet.http.HttpServletRequest` hereda del interfaz general `javax.servlet.ServletRequest` ofreciendo soporte para el protocolo HTTP, es decir, para que pueda ser utilizado en servlets HTTP.

Los métodos que ofrece este interfaz los comentamos a continuación:

- `String getAuthType()`: este método devuelve el tipo de autenticación utilizado para proteger el servlet, por ejemplo puede devolver la cadena BASIC o SSL o bien el valor nulo (null) si el servlet no se encuentra protegido mediante ningún mecanismo.
- `String getContextPath()`: devuelve la porción de la URL de petición que indica el contexto de la misma. El contexto comienza con una barra “/”.
- `Cookie[] getCookies()`: devuelve en un array todos los objetos Cookie que el cliente envió junto con su petición.
- `long getDateHeader(String nombre)`: devuelve el valor de una cabecera de petición como un long que representa un objeto Date. Más adelante se comentarán las distintas cabeceras de petición del protocolo HTTP.
- `String getHeader(String name)`: devuelve el valor de la cabecera de petición especificada como un objeto String.
- `Enumeration getHeaderNames()`: devuelve todos los nombres de cabecera que contiene la petición, en forma de un objeto `java.util.Enumeration`.
- `Enumeration getHeaders(String nombre)`: devuelve todos los valores de la cabecera de petición especificada, este método se utiliza para cabeceras que pueden contener varios valores.
- `int getIntHeader(String nombre)`: devuelve el valor en forma de int de la cabecera que se indica por parámetro.
- `String getMethod()`: devuelve el nombre del método del protocolo http que se ha utilizado para realizar la petición, por ejemplo, GET, POST o PUT.
- `String getPathInfo()`: devuelve cualquier información de camino extra asociada con la URL que utilizó el cliente para realizar la petición.
- `String getPathTranslated()`: devuelve cualquier información de rutas extra después del nombre del servlet pero antes de la cadena de consulta (QueryString), y lo traduce a una ruta real.
- `String getQueryString()`: devuelve la información de consulta contenida en la URL de petición, después del camino.

- `String getRemoteUser()`: devuelve el login del usuario que ha realizado la petición, si el usuario se ha autenticado o null si el usuario no se ha autenticado.
- `String getRequestedSessionId()`: devuelve el identificador de sesión indicado por el cliente.
- `String getRequestURI()`: devuelve una parte de la URL de la petición realizada. Esta parte se corresponde con el contenido de la URL a partir del nombre del servidor y el número de puerto.
- `String getServletPath()`: devuelve la ruta del servlet contenido en la URL de petición.
- `HttpSession getSession()`: devuelve la sesión actual asociada con la petición, o si la petición no tiene una sesión se crea una.
- `HttpSession getSession(boolean crear)`: devuelve la sesión actual relacionada con la petición, si no existe una sesión y el parámetro crear tiene el valor de true, devolverá una nueva sesión.
- `Principal getUserPrincipal()`: devuelve un objeto `java.security.Principal` que contiene el nombre del usuario autenticado actualmente.
- `boolean isRequestedSessionIdFromCookie()`: indica si el identificador de sesión pedido proviene de una cookie.
- `boolean isRequestedSessionIdFromURL()`: indica si el identificador de sesión pedido proviene de la URL de la petición.
- `boolean isRequestedSessionIdValid()`: indica si el identificador de sesión pedido es todavía válido.
- `boolean isUserInRole(String perfil)`: indica si el usuario autenticado está incluido en el perfil lógico especificado.

En esta lista de métodos echamos en falta todos los métodos relacionados con los formularios HTML, es decir, todos aquellos métodos que nos permiten obtener la información que envía el usuario dentro de un formulario. Esto es así porque los métodos de acceso a la información de los formularios los hereda de un interfaz genérico de los servlets, del interfaz `javax.servlet.ServletRequest`.

El interfaz `javax.servlet.ServletRequest` además de aportar los métodos que permiten acceder a la información contenida en los formularios existentes en las peticiones, ofrece una serie de métodos con variadas funciones, desde los que permiten obtener información acerca del servidor o del protocolo utilizado, hasta los que permiten almacenar y obtener atributos (entendiendo por atributo cualquier tipo de objeto Java) de la petición.

En el apartado correspondiente ofreceremos brevemente los distintos métodos que ofrece este interfaz, al igual que lo hemos hecho su interfaz hijo `HttpServletRequest`.

## Cabeceras de petición del protocolo HTTP

Llevamos comentado ya en varios lugares el término cabecera de petición o petición HTTP, en este apartado vamos a definir este termino y vamos a comentar las cabeceras de petición que existen en el protocolo HTTP, de esta forma podremos entender mejor los servlets que puede hacer uso de estas cabeceras de petición. En el siguiente apartado veremos como los servlets pueden acceder a estas cabeceras.

Las cabeceras de petición del protocolo HTTP se utilizan para que el cliente (navegador Web) realice una petición determinada de un recurso al servidor Web. Estas cabeceras llevan una serie de información que el servidor puede utilizar para devolver el resultado de la petición.

Podemos considerar que las cabeceras del protocolo HTTP es el lenguaje que utilizan el navegador y el servidor Web para comunicarse entre sí. Las peticiones del navegador serán cabeceras de petición HTTP, y las respuestas a estas peticiones serán las cabeceras de respuesta HTTP.

Cuando se activa un enlace a un recurso (página HTML, servlet, página JSP, imagen, etc.), el navegador o cliente genera una serie de cabeceras de petición HTTP. El servidor Web recibe estas cabeceras de petición, que incluyen la dirección del recurso dentro del servidor. En el caso de ser un servlet el recurso, el servidor ejecuta el servlet que deberá generar las cabeceras de respuesta HTTP mínimas requeridas y el código HTML, si procede, para que sea enviado por el servidor Web al navegador Web, como respuesta a su petición.

El cliente o navegador Web envía al servidor Web una serie de cabeceras de petición HTTP para indicarle los recursos que está demandando, cómo puede el cliente aceptar los datos devueltos y dónde se encuentran los datos adicionales que acompañan a la petición.

La cabecera que contiene el método completo de petición HTTP, es la primera cabecera de petición que se envía desde el navegador hacia el servidor Web, también se le denomina línea de petición. El método completo de la cabecera de petición es una línea formada por tres elementos separados por espacios. La sintaxis general de método completo es la siguiente:

```
Metodo_de_la_Peticion URL HTTP_version \n
```

Vamos a comentar cada uno de los elementos:

- Metodo\_de\_la\_Peticion: este método puede ser uno de los siguientes: GET, POST, HEAD, PUT, DELETE, LINK y UNLINK.
- URL: es la dirección o localización del fichero, programa o directorio al que se está intentando acceder.
- HTTP\_version: indica la versión del protocolo HTTP que puede manejar el navegador.
- Al final se indica un salto de línea.

Un ejemplo de método completo válido sería el siguiente:

```
GET http://www.eidos.es/index.html HTTP/1.1
```

El método GET es el método por defecto para seguir enlaces y enviar datos a través de Internet. Después de pulsar sobre un enlace, el navegador enviará el método GET.



El proceso de comunicación entre el navegador y el servidor Web comienza cuando el navegador realiza una conexión con el servidor. El navegador utiliza el nombre de dominio del servidor para identificar con que servidor se quiere conectar. A continuación el navegador envía las siguientes cabeceras de petición HTTP al dominio identificado, es decir, dirección IP en la que se encuentra el servidor correspondiente:

- Una cabecera de petición identificando el fichero (línea de petición vista anteriormente), recurso o servicio que se está demandando.
- Una serie de campos de la cabecera de petición HTTP que identificarán el navegador Web que está realizando la petición.
- Información especializada adicional acerca de la petición realizada.
- Cualquier tipo de datos que deban acompañar a la petición, como puede ser los parámetros que se envían por la cadena de consulta (QueryString) a un servlet.

Estas cabeceras de petición HTTP indican al servidor Web la información que el navegador Web está demandando y el tipo de respuesta que puede ser aceptada por el navegador. El servidor tomará todas estas cabeceras de petición HTTP enviadas por el navegador, y las hará disponibles para el servlet invocado, a través de un objeto *HttpServletRequest* que se pasa como parámetro al método *doXXX* correspondiente.

El servidor examina la primera cabecera de petición HTTP, que se denomina cabecera del método de petición, y trata de localizar la URL. Este proceso de búsqueda se inicia en el directorio raíz del servidor Web, este directorio raíz se denomina directorio de publicación en Internet, y a partir de él es dónde se encuentra toda la información que el servidor Web hace disponible para la World Wide Web.

Por ejemplo si el servidor identificado con el nombre de dominio *www.eidos.es*, recibe la siguiente petición: *http://www.eidos.es/cursos/temas/tema1.html*, el servidor primero comprobará el elemento *cursos*, identificando que es un directorio pasará a la siguiente parte. El siguiente elemento, *temas*, también es un directorio, por lo que se moverá a este directorio y continuará la búsqueda. El siguiente fragmento ya es el nombre de un fichero.

El servidor comprueba si la extensión del fichero indica que es un formato de texto válido para enviar al navegador, en este caso la extensión HTML sería correcta. En este punto el servidor comenzaría la tarea de enviar la información demandada al navegador.

El servidor Web debe responder a la petición de un navegador no sólo con la información pedida, sino también con una serie de cabeceras de respuesta HTTP. La primera de estas cabeceras de respuesta es la línea de estado. La línea de estado indica al navegador el resultado de la búsqueda de la URL demandada. La información que ofrece esta línea de estado puede variar, desde indicar un éxito hasta autorización requerida o incluso que la localización se ha modificado. Si el estado indica éxito, los contenidos de la URL se devolverán al navegador y serán desplegados por el mismo en la pantalla de la máquina del usuario. Para construir la línea de estado se utiliza una serie de códigos de estado que veremos con más profundidad en el siguiente capítulo, ya que se deberá utilizar el interfaz *HttpServletResponse*.

Todas las cabeceras del protocolo HTTP, las líneas de estado, y otros datos se intercambian entre el navegador y el servidor Web a través de Internet. En Internet las conexiones y comunicaciones se realizan el protocolo de comunicaciones llamado TCP/IP (Transmission Control Protocol/Internet Protocol). Este protocolo de comunicaciones permite tener interconectados entre sí distintos tipos de máquinas, y permite una comunicación entre ellas, sin tener en cuenta el tipo de máquina.

El protocolo TCP/IP ofrece comunicaciones fiables mediante servicios orientados a la conexión (protocolo TCP) y no fiables a través de servicios no orientados a la conexión (protocolo UDP, User Datagram Protocol).

Un servicio orientado a la conexión significa que permite intercambiar un gran volumen de datos de una manera correcta, es decir, se asegura que los datos llegan en el orden en el que se mandaron y no existen duplicados, además tiene mecanismos que le permiten recuperarse ante errores.

Realizada esta breve introducción al protocolo HTTP y a las cabeceras de petición del mismo, vamos a mostrar las distintas cabeceras de petición del protocolo HTTP acompañadas de una breve descripción (Tabla 6).

Cabecera de petición	Descripción
Accept	Indica al servidor el tipo de datos que el navegador puede aceptar. Estos tipos se indican mediante el uso de tipos MIME.
Accept-Charset	Indica al servidor que conjunto de caracteres que puede utilizar el navegador.
Accept-Encoding	Indica al servidor el tipo de codificación de datos que el navegador puede aceptar.
Accept-Language	Indica al servidor el lenguaje natural que prefiere el navegador.
Authorization	Utilizado por el navegador para autenticarse con el servidor.
Cache-Control	Se utiliza por el cliente para indicar como se almacenarán las páginas en la caché de los servidores proxy.
Connection	Indica si el cliente puede mantener conexiones HTTP persistentes.
Content-Length	Identifica el tamaño de los datos transferidos en bytes. Aplicable únicamente a peticiones POST.
Content-Type	Identifica el tipo de datos que se está transfiriendo.
Cookie	Esta cabecera es utilizada para devolver las cookies a los servidores que previamente las han enviado a los navegadores.
Expect	Esta cabecera no casi utilizada, y permite al cliente decir al servidor que tipo de comportamiento espera de él.
From	Esta cabecera indica la dirección de correo electrónico de la persona responsable de la petición HTTP.
Host	Esta cabecera requerida indica el nombre del servidor y el número de puerto.

If-Match	Cabecera casi no utilizada, se aplica a peticiones PUT, y permite especificar al cliente si una operación se va a realizar atendiendo a la coincidencia de unas etiquetas.
If-Modified-Since	Indica que el cliente quiere una página sólo si ha sido modificada después de una fecha especificada.
If-None-Match	Similar a If-Match, excepto que la operación se realizará sólo si las etiquetas no coinciden.
If-Range	Cabecera que tampoco se usa demasiado, permite al cliente pedir partes perdidas de un documento parcial.
If-Unmodified-Since	Esta cabecera realiza la función contraria a If-Modified-Since.
Pragma	Sólo puede tener el valor no-cache, con lo que indica que siempre se debe obtener una página aunque se posea una copia local.
Proxy-Authorization	Permite a los clientes identificarse antes los servidores proxy que lo requieran.
Range	Muy similar a If-Range.
Referer	Indica al servidor la URL del enlace que fue utilizado para enviar la cabecera del método de petición al servidor.
Upgrade	Permite especificar al cliente si prefiere utilizar un protocolo preferido distinto de HTTP 1.1.
User-Agent	Indica el tipo de navegador que está realizando la petición.
Via	Esta cabecera es añadida por pasarelas y proxies para mostrar los sitios intermedios por lo que ha pasado una petición.
Warning	Raramente utilizado, permite indicar a los clientes errores de caché o de transformación.

Tabla 6. cabeceras de petición del protocolo http

En el siguiente apartado veremos los distintos métodos que el interfaz *HttpServletRequest* pone a nuestra disposición para acceder a la información de las cabeceras de petición del protocolo HTTP.

## Leyendo las cabeceras de petición desde los servlets

Para leer una cabecera de petición podemos hacerlo de forma muy sencilla mediante el método `getHeader()` del interfaz *HttpServletRequest*, pasándole por parámetro el nombre de la cabecera de la

que queremos obtener su valor. En el Código Fuente 23 se muestra un servlet que devuelve al cliente el navegador Web que ha utilizado para realizar la petición.

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

public class ServletNavegador extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.println("<html><body><h1>Navegador:" +
                    request.getHeader("User-Agent") + "</h1></body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request, response);
    }
}
```

Código Fuente 23

Como se puede comprobar se ha recuperado la cabecera de petición User-Agent.

El resultado de la ejecución de este servlet con un navegador Internet Explorer 5 es el que se muestra en la Figura 8.

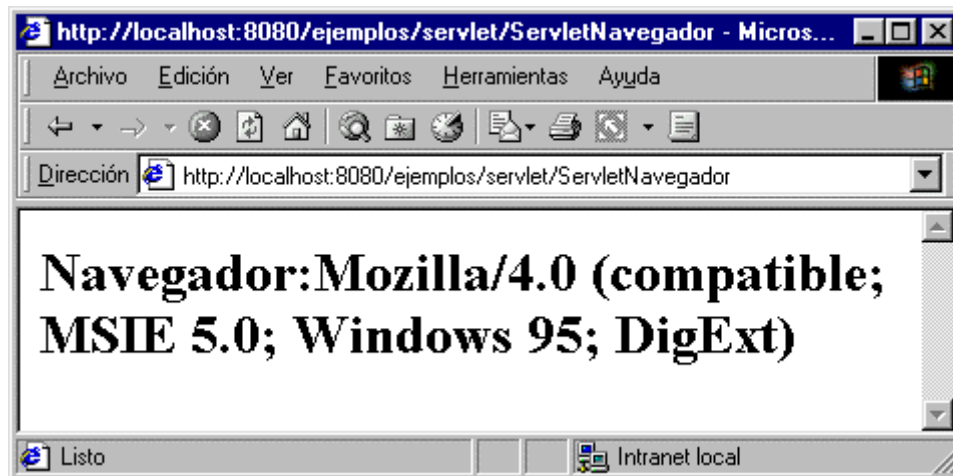


Figura 8

Aunque el método `getHeader()` es el método de propósito general para obtener cabeceras de petición HTTP, el interfaz `HttpServletRequest` ofrece otros métodos especializados en obtener las cabeceras más comunes.

- `getCookies()`: este método devuelve los contenidos de la cabecera de petición HTTP Cookies en forma de un array de objetos `javax.servlet.http.Cookies`. En el capítulo correspondiente trataremos en detalle la clase `Cookie`.
- `getAuthType()` y `getRemoteUser()` parten la cabecera `Authorization` en dos fragmentos de información, el tipo de autenticación y el nombre del usuario que se autentifica.
- `getContentLength()`: devuelve un entero que se corresponde con la cabecera `Content-Length`.
- `getContentType()`: devuelve una cadena que devuelve el valor de la cabecera `Content-Type`.
- `getHeaderNames()`: devuelve en un objeto `java.util.Enumeration` los nombres de todas las cabeceras HTTP existentes en una petición.
- `getHeaders()` : en ocasiones una cabecera puede aparecer varias veces en una petición HTTP, ofreciendo distintos valores, un ejemplo puede ser la cabecera `Accept-Language`. Este método devuelve un objeto `Enumeration` con cada uno de los valores de la cabecera.

El interfaz `HttpServletRequest` también nos permite obtener información de la línea principal de petición, mediante los siguientes métodos:

- `getMethod()`: devuelve el método utilizado para realizar la petición, normalmente será GET o POST.
- `getRequestURI()`: este método devuelve el fragmento de URL que se encuentra entre el número de puerto y los datos que se envían.
- `getProtocol()`: devuelve la versión utilizada del protocolo HTTP. Este método lo hereda el interfaz `HttpServletRequest` del interfaz más general `javax.servlet.ServletRequest`.

A continuación vamos a mostrar en un ejemplo estos dos grupos de métodos del interfaz `HttpServletRequest` en acción.

En un primer lugar el servlet lanza sobre el objeto `HttpServletRequest` el método `getHeaderNames()` para obtener los nombres de todas las cabeceras de petición HTTP, a continuación recorre el objeto `Enumeration` resultante y para cada elemento le pasa por parámetro al método `getHeader()` el nombre de la cabecera correspondiente. Se debe tener en cuenta que para poder utilizar el objeto `Enumeration` debemos importar el paquete `java.util`.

A continuación lanza los métodos correspondientes sobre el objeto `HttpServletRequest` para obtener el resto de información de la petición http.

En el Código Fuente 24 se puede ver el aspecto completo de este servlet, además al observar el código del servlet podemos adelantar una de las desventajas de los servlets, existe una gran mezcla entre lo que es el código fuente propiamente dicho y los elementos HTML utilizados para realizar la presentación de la información, ya que en las distintas sentencias `out.println()` parecen juntos estos dos elementos. Tanto es así que para poder enviar a la salida del usuario las comillas dobles (") se debe utilizar un carácter de escape (\).

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

```

public class ServletMuestraPetición extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><BODY BGCOLOR=\"#FDF5E6\">" +
            "<TABLE BORDER=1 ALIGN=CENTER>" +
            "<TR BGCOLOR=\"#FFAD00\">" +
            "<TH>Nombre cabecera</TH><TH>Valor cabecera</TH>");
        Enumeration nombresCabeceras = request.getHeaderNames();
        while(nombresCabeceras.hasMoreElements()) {
            String nombre = (String)nombresCabeceras.nextElement();
            out.println("<TR><TD>" + nombre+"</TD>");
            out.println("<TD>" + request.getHeader(nombre)+"</TD>");
        }
        out.println("</TABLE>"+ "<B>Método de petición: </B>" +
            request.getMethod() + "<BR>" +
            "<B>URI pedida: </B>" +
            request.getRequestURI() + "<BR>" +
            "<B>Protocolo: </B>" +
            request.getProtocol() + "</BODY></HTML>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

Código Fuente 24

El método `nextElement()` del objeto `Enumeration` devuelve un objeto `Object`, por lo tanto es necesario realizar la transformación del tipo de objeto (casting), en este caso se hace a un objeto de la clase `String`.

Si compilamos este servlet y lo copiamos al directorio correspondiente de nuestra aplicación Web, al ejecutarlo mediante el servidor Jakarta Tomcat obtenemos el resultado de la Figura 9.

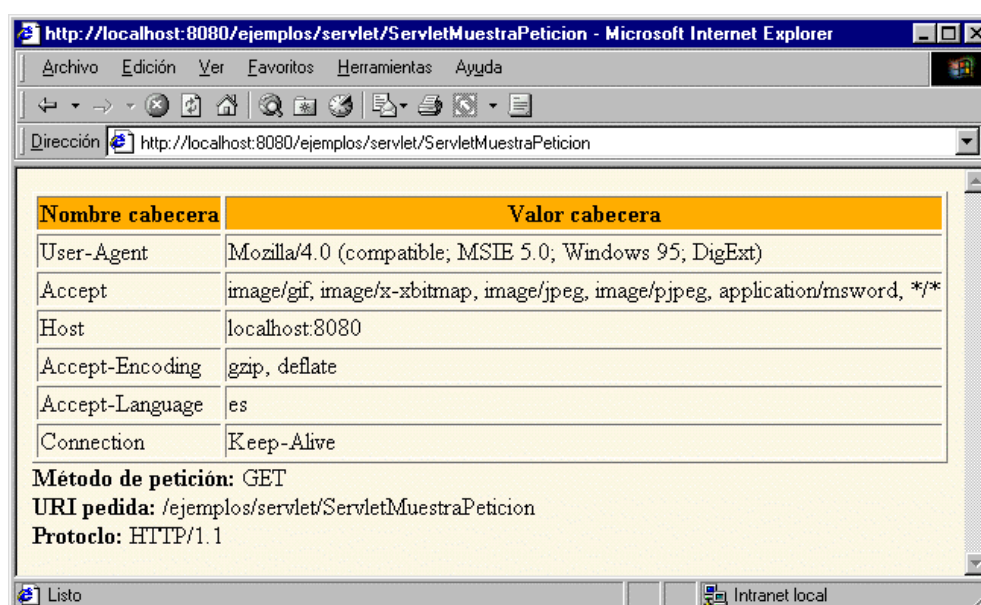


Figura 9

En un apartado próximo vamos a comentar la correspondencia entre las variables de entorno de los scripts CGI y una serie de métodos que nos ofrecen los servlets, sobretodo a través del interfaz `HttpServletRequest`.

## El interfaz `ServletRequest`

Este interfaz que se encuentra en el paquete `javax.servlet`, es el interfaz del que hereda el interfaz `javax.servlet.HttpServletRequest`. El interfaz `ServletRequest` ofrece una serie de métodos que nos permiten tratar las peticiones realizadas a un servlet. Mediante un objeto `ServletRequest` podemos obtener información de parámetros, formularios, atributos, etc., aunque nunca vamos a utilizar un objeto `ServletRequest`, sino que utilizaremos un objeto `HttpServletRequest`, que hereda de este interfaz.

A continuación ofrecemos una vista previa de todos los métodos del interfaz `ServletRequest`, en el apartado dedicado al tratamiento de los formularios dentro de un servlet veremos en acción y en más detalle algunos de estos métodos.

- `Object getAttribute(String nombre)`: devuelve el valor de un atributo determinado almacenado en la petición, si el atributo no existe devuelve el valor null. Más adelante veremos que este método, y todos los relacionados con los atributos de la petición, es muy interesante a la hora de pasar información en forma de objetos entre distintos servlets, o entre un servlet y una página JSP (JavaServer Pages).
- `Enumeration getAttributeNames()`: devuelve en un objeto `Enumeration` los nombres de todos los parámetros disponibles en una petición.
- `String getCharacterEncoding()`: devuelve el nombre del tipo de codificación empleado en el cuerpo de la petición.
- `int getContentLength()`: devuelve la longitud, en bytes, del cuerpo de la petición. Si no es conocida la longitud se devuelve `-1`.
- `String getContentType()`: devuelve el tipo MIME que se corresponde con el cuerpo de la petición, o el valor null si el tipo es desconocido.
- `ServletInputStream getInputStream()`: devuelve el cuerpo de la petición como datos binarios utilizando un objeto `ServletInputStream`.
- `Locale getLocale()`: devuelve el objeto `java.util.Locale` preferido para el que el cliente aceptará el contenido, basado en la cabecera `Accept-Language`.
- `Enumeration getLocales()`: devuelve un objeto `Enumeration` que contiene objetos `Locale`, según el orden de preferencia del cliente, y basándose siempre en la cabecera `Accept-Language`.
- `String getParameter(String nombre)`: devuelve el valor de un parámetro, que se corresponde con un campo de un formulario, como un objeto `String`. Si el parámetro no existe se devolverá el valor null.
- `Enumeration getParameterNames()`: devuelve en un objeto `Enumeration` los nombres de todos los parámetros contenidos en una petición determinada.

- `String[] getParameterValues(String nombre)`: devuelve en un array de objetos `String` todos los valores del parámetro cuyo nombre se indica, si no existen estos valores se devolverá `null`.
- `String getProtocol()`: devuelve el nombre y versión del protocolo que utiliza la petición, de la forma `protocolo/versión_mayor.versión_menor.`, por ejemplo, `HTTP/1.1`.
- `BufferedReader getReader()`: devuelve el cuerpo de la petición como caracteres utilizando un objeto `java.io.BufferedReader`.
- `String getRemoteAddr()`: devuelve la dirección IP del cliente que envió la petición.
- `String getRemoteHost()`: devuelve el nombre de la máquina del cliente que realizó la petición, si este nombre no está disponible, devuelve la dirección IP.
- `RequestDispatcher getRequestDispatcher(String ruta)`: devuelve un objeto `javax.servlet.RequestDispatcher`, que actúa como un envoltorio para un recurso localizado en la ruta dada.
- `String getScheme()`: devuelve el nombre del esquema utilizado para realizar la petición, por ejemplo, `http`, `https` o `ftp`.
- `String getServerName()`: devuelve el nombre del servidor que ha recibido la petición.
- `int getServerPort()`: devuelve el número de puerto en el que se ha recibido la petición.
- `boolean isSecure()`: devuelve un valor booleano indicando si la petición se ha realizado a través de un canal seguro, como puede ser el protocolo `HTTPS`.
- `void removeAttribute(String nombre)`: elimina un atributo de la petición.
- `void setAttribute(String nombre, Object objeto)`: almacena un atributo en la petición.

## Variables CGI

Los lectores que provengan de los scripts CGI tendrán claro lo que son las variables CGI o variables de entorno. La información que ofrecen estas variables de entorno también está disponible a través de los servlets. En este capítulo además de mostrar la correspondencia entre las variables CGI y los métodos ofrecidos por los servlets, vamos a comentar las variables CGI para aquellos lectores que no provengan del entorno de los scripts CGI.

Las variables de entorno de los scripts CGI es un conjunto de variables que contiene una colección variada de información. Algunas de estas variables se basan en la línea de petición `HTTP` o en sus cabeceras de petición, otras se derivan del propio socket utilizado para realizar la conexión o del servidor Web.

Los servlets ofrecen mecanismos mucho más sencillos para obtener esta información a través de los interfaces tales como `ServletRequest` y `HttpServletRequest`.



A continuación vamos a ir comentando cada una de las variables de entorno de los scripts CGI y su correspondencia con los servlets.

- **AUTH\_TYPE**: se corresponde con la información contenida en la cabecera *Authorization*, su equivalente en un servlet es `request.getAuthType()`, siendo `request` un objeto *HttpServletRequest*.
- **CONTENT\_LENGTH**: sólo disponible para peticiones de tipo *POST*, esta variable indica el número de bytes de los datos enviados, se corresponde con el valor de la cabecera *Content-Length*. El equivalente en un servlet es `request.getContentLength()`.
- **CONTENT\_TYPE**: devuelve el tipo *MIME* de los datos enviados. Su correspondencia en un servlet es la sentencia `request.getContentType()`.
- **DOCUMENT\_ROOT**: esta variable especifica el directorio real que se corresponde con la URL `http://servidor/aplicacion`. En este caso el método correspondiente del servlet no lo ofrece el interfaz *HttpServletRequest* ni el interfaz *ServletRequest*, sino que lo ofrece el interfaz *ServletContext*. Un objeto *ServletContext* lo obtenemos mediante el método `getServletContext()` de la clase *GenericServlet*, de la que hereda la clase *HttpServlet*.

El interfaz *ServletContext* ofrece una serie de métodos que permiten que el servlet se comunique con el contenedor en el que se ejecuta. Así por ejemplo nos ofrece el método `getRealPath()`, este método recibe como parámetro una ruta virtual y nos devuelve la ruta física correspondiente.

Así el equivalente en un servlet, a la variable de entorno CGI que nos ocupa, sería la sentencia `getServletContext().getRealPath("/")`. Al indicar la barra ("/") este método nos devuelve la ruta física de la URL que se corresponde con la aplicación actual.

- **HTTP\_XXX\_YYY**: existen un grupo de variables de entorno del tipo *HTTP\_NOMBRE\_CABECERA*, los equivalentes en los servlets ya los vimos un apartado anterior.
- **PATH\_INFO**: esta variable contiene cualquier información relativa a rutas que se añade al final de la URL que se corresponde con un servlet, pero antes de los datos de consulta (*QueryString*). Así en la URL `http://servidor/servlet/UnServlet/info/ruta?id=2`, el contenido de la variable *PATH\_INFO* sería `/info/ruta`. El equivalente a esta variable dentro de un servlet es `request.getPathInfo()`.
- **PATH\_TRANSLATED**: ofrece la ruta de una URL mapeada a su ruta física en el servidor. El equivalente en un servlet sería la sentencia `request.getPathTranslated()`.
- **QUERY\_STRING**: esta otra variable, llamada cadena de consulta, contiene la información que se adjunta a una petición *GET* y que se encuentra codificada según la codificación URL. En el siguiente apartado veremos como tratar esta información que se añade en forma de parámetros a la URL de la petición. La equivalencia en un servlet es `request.getQueryString()`.
- **REMOTE\_ADDR**: representa la dirección IP del cliente que realizó la petición. Se corresponde con `request.getRemoteAddr()`.
- **REMOTE\_HOST**: contiene el nombre completo del dominio del cliente que realizó la petición. Su correspondencia en un servlet sería `request.getRemoteHost()`.

- **REMOTE\_USER**: contiene el nombre del usuario que se ha autenticado con el servidor. Tenemos acceso a este valor dentro de un servlet mediante la sentencia `request.getRemoteUser()`.
- **REQUEST\_METHOD**: ofrece el tipo de método de petición HTTP que se ha utilizado para realizar la petición correspondiente. Su equivalente en un servlet es la sentencia `request.getMethod()`.
- **SCRIPT\_NAME**: especifica el camino a un servlet, relativo al directorio raíz del servidor. Esta información la obtenemos en un servlet mediante la sentencia `request.getServletPath()`.
- **SERVER\_NAME**: esta variable de entorno contiene el nombre de la máquina del servidor. Se corresponde con la sentencia `request.getServerName()`.
- **SERVER\_PORT**: esta variable almacena el puerto en el que se encuentra escuchando el servidor. La equivalencia en un servlet es `request.getServerPort()`.
- **SERVER\_PROTOCOL**: indica el protocolo que se encuentra en la petición realizada. Está información se obtiene en un servlet mediante la sentencia `request.getProtocol()`.
- **SERVER\_SOFTWARE**: esta variable ofrece información que permite identificar el software que implementa el servidor Web. El acceso a esta información en un servlet se realiza a través del método `getServerInfo()` del interfaz `ServletContext`. Por lo tanto esta variable de entorno se corresponde con la sentencia `getServletContext().getServerInfo()`.

Para mostrar todas estas equivalencias entre variables CGI y la forma de obtenerlas en los servlets, vamos a realizar un servlet que se encarga de mostrar la información que se corresponde con estas variables. El servlet completo se puede observar en el Código Fuente 25.

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

public class MuestraVariablesCGI extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String[][] variables = {
            { "AUTH_TYPE", request.getAuthType() },
            { "CONTENT_LENGTH", String.valueOf(request.getContentLength()) },
            { "CONTENT_TYPE", request.getContentType() },
            { "DOCUMENT_ROOT", getServletContext().getRealPath("/") },
            { "PATH_INFO", request.getPathInfo() },
            { "PATH_TRANSLATED", request.getPathTranslated() },
            { "QUERY_STRING", request.getQueryString() },
            { "REMOTE_ADDR", request.getRemoteAddr() },
            { "REMOTE_HOST", request.getRemoteHost() },
            { "REMOTE_USER", request.getRemoteUser() },
            { "REQUEST_METHOD", request.getMethod() },
            { "SCRIPT_NAME", request.getServletPath() },
            { "SERVER_NAME", request.getServerName() },
            { "SERVER_PORT", String.valueOf(request.getServerPort()) },
            { "SERVER_PROTOCOL", request.getProtocol() },
            { "SERVER_SOFTWARE", getServletContext().getServerInfo() }
        };
    }
}
```

```

        out.println("<html><BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<TABLE BORDER=1 ALIGN=\"CENTER\"> " +
            "<TR BGCOLOR=\"#FFAD00\">" +
            "<TH>Variable CGI</TH><TH>Valor</TH>");
        for(int i=0; i<variables.length; i++) {
            String varNombre = variables[i][0];
            String varValor = variables[i][1];
            if (varValor == null) varValor = "<I>No especificado</I>";
            out.println("<TR><TD>" + varNombre+ "</TD><TD>" + varValor+"</TD>");
        }
        out.println("</TABLE></BODY></HTML>");
    }

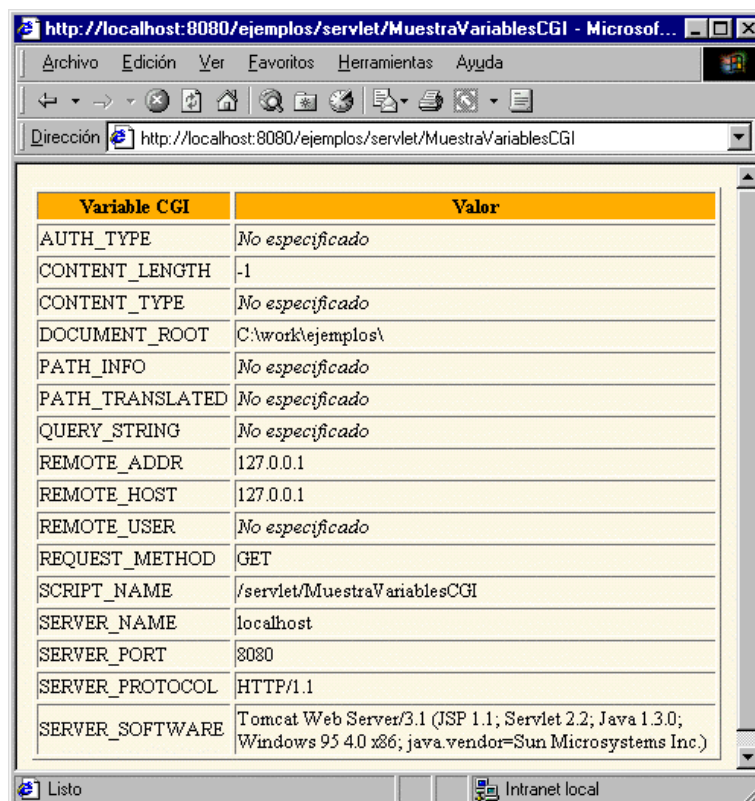
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        doGet(request, response);
    }
}

```

Código Fuente 25

Como se puede comprobar se ha creado un array de dos dimensiones, en una de ellas se almacena una cadena con el nombre de la variable CGI, y en la otra dimensión el valor de esa variable. Los valores de las variables se recuperan de las distintas formas que hemos visto anteriormente. A continuación el servlet se limita a mostrar en un bucle el contenido de el array mediante una tabla. Un ejemplo de ejecución de este servlet se puede ver en la Figura 10.



Variable CGI	Valor
AUTH_TYPE	No especificado
CONTENT_LENGTH	-1
CONTENT_TYPE	No especificado
DOCUMENT_ROOT	C:\work\ejemplos\
PATH_INFO	No especificado
PATH_TRANSLATED	No especificado
QUERY_STRING	No especificado
REMOTE_ADDR	127.0.0.1
REMOTE_HOST	127.0.0.1
REMOTE_USER	No especificado
REQUEST_METHOD	GET
SCRIPT_NAME	/servlet/MuestraVariablesCGI
SERVER_NAME	localhost
SERVER_PORT	8080
SERVER_PROTOCOL	HTTP/1.1
SERVER_SOFTWARE	Tomcat Web Server/3.1 (JSP 1.1; Servlet 2.2; Java 1.3.0; Windows 95 4.0 x86; java.vendor=Sun Microsystems Inc.)

Figura 10

En el siguiente apartado vamos a mostrar como se puede tratar y acceder a la información contenida en los formularios que forman parte de la petición que se realiza a un servlet.

## Formularios y servlets

Un formulario HTML es uno de los medios más utilizados para obtener información de un usuario a través de la Web. Un formulario se puede componer de cajas de texto, botones de opción, casillas de verificación, áreas de texto, etc, el formulario se desplegará en el navegador cliente dentro del código HTML, no se debe olvidar que un formulario forma parte de la especificación HTML y viene delimitado por las etiquetas `<FORM></FORM>`.

Los formularios suelen tener un botón del tipo Submit, es decir, enviar. Si el usuario pulsa sobre este botón, toda la información incluida dentro del formulario asociado se enviará al servidor, es decir, se enviarán los valores de todos los controles que posean un nombre, esto es, la propiedad NAME de cada elemento del formulario debe tener un valor. El cliente puede enviar información al servidor de dos maneras, con el método POST o el método GET. El método utilizado se indica en el atributo METHOD de la etiqueta `<FORM>`.

Antes de que el navegador envíe la información al servidor, se codifica en un formato llamado codificación URL. En esta codificación los pares nombre/valor (nombre del elemento del formulario y su valor) se unen con símbolos de igual y los diferentes pares se separan con el símbolo del ampersand (&). Los espacios son sustituidos por el carácter +, y otros caracteres no alfanuméricos son reemplazados con valores hexadecimales de acuerdo con la especificación RFC 1738. A continuación se ofrece un ejemplo de una cadena de parámetros.

```
nombre1=valor1&nombre2=valor2&nombre3=valor3
```

Cuando se utiliza el método GET se envía la información codificada del usuario añadida a la petición del recurso, ya sea un servlet una página JSP. La URL que identifica el servlet y la información codificada se separa mediante un signo de interrogación (?), así por ejemplo si se quiere acceder a la dirección `www.eidos.es/default.htm` y se envía también el nombre y la edad proporcionados por el usuario veríamos la siguiente cadena:

```
http://www.eidos.es/default.htm?nombre=pepe&edad=23
```

El método GET muestra esta cadena en la barra de direcciones del navegador. No sólo es poco estético sino que no es seguro, ya que la información contenida en esta cadena es visible para todo el mundo. Por lo tanto nunca se debe utilizar el método GET para enviar contraseñas o información confidencial. El método GET posee una limitación de tamaño, el máximo de caracteres que puede haber en una cadena es de 1024, es decir, como máximo se pueden enviar 1024K, sin embargo el método POST no posee ninguna limitación de tamaño.

El método GET muchas veces es útil para a través de un enlace pasar parámetros sin la necesidad de utilizar formularios, por ejemplo, si es necesario enviar a un servlet llamado `ProcesaDatos` algún dato, se podría hacer mediante el siguiente enlace:

```
http://localhost:8080/servlet/ProcesaDatos?nombreDato=valorDato
```

El método GET envía la información a través de cabeceras del protocolo HTTP. La información es codificada como se mostró anteriormente y enviada dentro de una cabecera llamada `QUERY_STRING`. Por el contrario, si utilizamos el método POST, la información no irá en una cabecera sino que irá en el cuerpo de la petición HTTP.

El método POST sólo se puede utilizar cuando la información es enviada a un fichero ejecutable, como puede ser un CGI (Common Gateway Interface), una página ASP (Active Server Pages), un servlet o una página JSP (JavaServer Pages), para que puedan extraer la información del cuerpo de la petición HTTP, en caso contrario se producirá un error. El destino al que se envía la información de un formulario se indica en el atributo ACTION de la etiqueta <FORM>.

Si consideramos que tenemos un formulario cuya función es recoger el nombre y edad del usuario y que se crea con el siguiente código HTML (Código Fuente 26):

```
<form action="servlet/ProcesaInfo" method="POST">
  Nombre: <input type="Text" name="nombre" value="">
  Edad: <input type="Text" name="edad" value="">
  <input type="Submit" name="boton" value="Enviar">
</form>
```

Código Fuente 26

Si se cambia la primera línea por <form action="pagina.html" method="POST"> se producirá un error ya que la información no se envía a un recurso ejecutable, sino que se envía a una página estática. Pero si la cambiamos por <form action="pagina.html" method="GET"> no dará error ya que no se utiliza el método POST.

Una vez enviada la información al servidor debe ser extraída y decodificada a un formato útil para poder ser tratada, pero esto no supone ningún problema, los servlets realizan esta tarea a través del parámetro *HttpServletRequest* de sus distintos métodos *doXXX* (*doGet()* y *doPost()* sobretodo).

El interfaz *HttpServletRequest* ofrece los métodos *getParameter()*, *getParameterValues()* y *getParameterNames()* para recuperar el valor de los campos de los formularios HTML que realizan la petición del servlet correspondiente.

Para obtener el valor de un campo (parámetro) determinado podemos utilizar el método *getParameter()* del interfaz *HttpServletRequest*, pasándole como argumento a este método un objeto *String* que representa el nombre del parámetro a recuperar, teniendo en cuenta que los nombres de los parámetros son case-sensitive, es decir, se distingue entre mayúsculas y minúsculas, no será lo mismo *request.getParameter("param1")* que *request.getParameter("Param1")*, estaremos recuperando valores de parámetros distintos.

El método *getParameter()* nos devolverá un objeto *String* que contiene el valor del campo del formulario, después podremos aplicar las transformaciones de datos que sean necesarias dentro del servlet.

Si el parámetro que queremos recuperar no existe, el método *getParameter()* devolverá el valor null. Este método, al igual que todos los que permiten recuperar información de un formulario, se pueden utilizar indistintamente sin tener en cuenta el método de petición HTTP empleado, es decir, *getParameter()* es válido tanto para peticiones GET como POST.

A continuación vamos a mostrar un servlet muy sencillo que se limita a utilizar el método *getParameter()* para mostrar los valores de los campos del formulario que se le ha pasado en la petición.

El código HTML de la página Web que contiene el formulario se puede ver en el Código Fuente 27, y el del servlet en el Código Fuente 28.

```

<html>
<head>
<title>Página que envía un formulario a un servlet</title>
</head>
<body>
<form action="servlet/ServletProcesaInfo" method="GET">
  Nombre: <input type="Text" name="nombre" size="20"><br>
  Apellidos: <input type="Text" name="apellidos" size="20"><br>
  Edad: <input type="Text" name="edad" size="20"><br>
  Email:<input type="Text" name="email" size="30"><br>
  <input type="Submit" name="boton" value="Enviar">
</form>
</body>
</html>

```

Código Fuente 27

```

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

public class ServletProcesaInfo extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body><ul>"+
            "<li>nombre: <i>"+request.getParameter("nombre")+"</i>"+
            "<li>apellidos: <i>"+request.getParameter("apellidos")+"</i>"+
            "<li>edad: <i>"+request.getParameter("edad")+"</i>"+
            "<li>email: <i>"+request.getParameter("email")+"</i>"+
        );
        out.println("</ul></BODY></HTML>");
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

Código Fuente 28

La página Web la podemos situar en cualquier directorio de nuestra aplicación, en nuestro caso la vamos a situar en el directorio raíz de nuestra aplicación Web de ejemplo, es decir, en c:\work\ejemplos.

El aspecto de la página HTML que contiene el formulario se puede observar en la Figura 11, y si pulsamos el botón de envío del formulario se invocará el servlet que recibirá todos los datos facilitados por el usuario en el formulario y los mostrará en pantalla, el resultado de la ejecución del servlet está en la Figura 12.

Hay casos en los que un parámetro puede tener varios valores, y se deberá utilizar el método `getParameterValues()`. Este método devuelve un array de objetos `String` con cada uno de los valores de un parámetro del formulario. Si el parámetro no existe devolverá `null`, y si existe pero sólo tiene un valor, este método devolverá un array de un único elemento.

Figura 11. Formulario que envía sus campos a un servlet

Figura 12. Servlet que muestra el contenido del formulario

Vamos a ampliar el formulario anterior añadiendo varios parámetros que puede tener varios valores, se trata de varias casillas de selección y una lista de selección múltiple. El código HTML de la página que muestra el formulario se puede observar a continuación(Código Fuente 29).

```
<html>
<head>
<title>Página que envía un formulario a un servlet</title>
</head>
<body>
<form action="servlet/ServletProcesaInfo" method="GET">
  <b>Datos personales</b><br>
  Nombre: <input type="Text" name="nombre" size="20"><br>
  Apellidos: <input type="Text" name="apellidos" size="20"><br>
  Edad: <input type="Text" name="edad" size="20"><br>
  Email:<input type="Text" name="email" size="30"><br>
  <br>
  <b>Departamentos</b><br>
  <select name="departamentos" multiple size="5">
    <option value="Sistemas">Sistemas</option>
```

```

<option value="Desarrollo">Desarrollo</option>
<option value="Comercial">Comercial</option>
<option value="Administración">Administración</option>
<option value="Formación">Formación</option>
</select><br><br>
<b>Lenguajes de programación y entornos de desarrollo</b><br>
<input type="checkbox" name="lenguajes" value="C++">C++
<input type="checkbox" name="lenguajes" value="Visual Basic">Visual Basic
<input type="checkbox" name="lenguajes" value="Java">Java
<input type="checkbox" name="lenguajes" value="ASP">ASP
<input type="checkbox" name="lenguajes" value="Delphi">Delphi
<input type="checkbox" name="lenguajes" value="JSP">JSP
<br>
<input type="Submit" name="boton" value="Enviar">
</form>
</body>
</html>

```

Código Fuente 29

Y el nuevo aspecto del formulario es el de la Figura 13.

Ahora nuestro servlet, además de recoger los valores anteriores, debe recoger los nuevos campos del formulario que son los departamentos y los lenguajes de programación. Como se puede ver pueden tener varios valores estos nuevos campos, por lo tanto se utilizará el método `getParameterValues()` y se recorrerá el array con los valores y se irán mostrando uno a uno. El Código Fuente 30 muestra el servlet completo.

```

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

public class ServletProcesaInfo extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body><ul>" +
            "<li>nombre: <i>" + request.getParameter("nombre") + "</i>" +
            "<li>apellidos: <i>" + request.getParameter("apellidos") + "</i>" +
            "<li>edad: <i>" + request.getParameter("edad") + "</i>" +
            "<li>email: <i>" + request.getParameter("email") + "</i>"
        );
        String[] valores=request.getParameterValues("departamentos");
        out.print("<li>departamentos: <i>");
        int i;
        if (valores!=null){
            for (i=0;i<=valores.length-2;i++){
                out.print(valores[i]+",");
            }
            out.print(valores[i]+ "</i>");
        }
        valores=request.getParameterValues("lenguajes");
        out.print("<li>lenguajes: <i>");
        if (valores!=null){
            for (i=0;i<=valores.length-2;i++){
                out.print(valores[i]+",");
            }
            out.print(valores[i]+ "</i>");
        }
        out.println("</ul></BODY></HTML>\n");
    }
}

```



```

    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

Código Fuente 30

Figura 13. Nuevo aspecto del formulario

El nuevo resultado de la ejecución de este servlet es el de la Figura 14.

El otro método que nos ofrecía el interfaz *HttpServletRequest* para manejar los parámetros enviados junto con la petición era el método *getParameterNames()*. Este método devuelve en un objeto *Enumeration* los nombres de todos los parámetros que contenía la petición. Los nombres de los parámetros se encuentran en el objeto *Enumeration* sin ningún orden específico.

Vamos a realizar un nuevo ejemplo sobre el formulario anterior, pero esta vez utilizaremos los métodos *getParameterNames()* y *getParameterValues()* en combinación. El formulario va a ser el mismo que el descrito en el , pero vamos a modificar el código del servlet para que haga uso del método *getParameterNames()*.

En un primer lugar el servlet obtiene en un objeto *Enumeration* los nombres de todos los parámetros existentes en la petición, a continuación entra en un bucle del que se saldrá cuando se haya terminado de recorrer todos los parámetros.

Utilizaremos el método *hasMoreElements()* del interfaz *Enumeration* para averiguar si quedan más nombres de parámetros. Cada nombre de parámetro lo recuperamos mediante el método *nextElement()* del interfaz *Enumeration*, aunque debemos realizar una transformación de datos (casting), ya que el método *nextElement()* devuelve objetos de la clase *Object*, y nosotros necesitamos objetos *String*, ya

que los utilizaremos para pasárselos como argumento al método `getParameterValues()`, y así obtener el valor o valores del parámetro correspondiente.

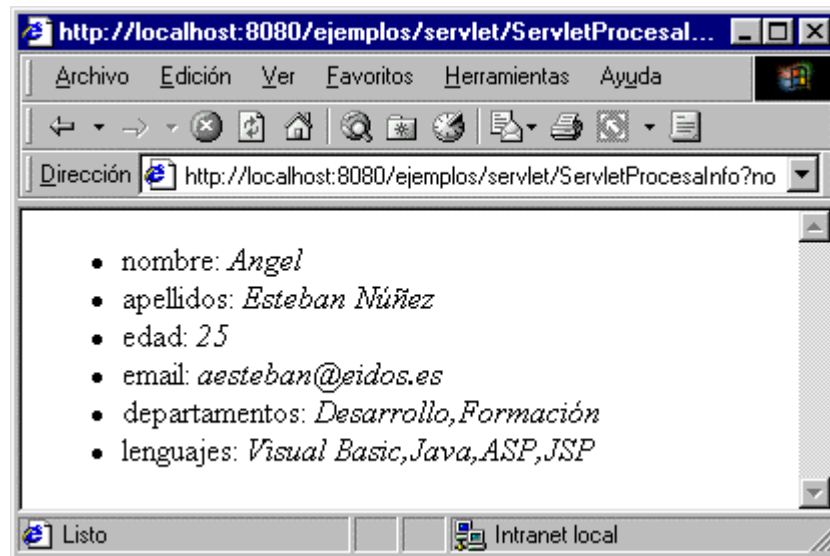


Figura 14. El nuevo resultado del servlet

Antes de mostrar el valor o valores de un parámetro, se comprueba si realmente el parámetro posee varios valores o no, si el parámetro tiene varios valores recorreremos el array de valores devuelto por el método `getParameterValues()`, y si tiene un único el valor el parámetro, recuperaremos el primer elemento del array. Por cada parámetro se comprueba si tiene valor o no.

Todo esto que hemos comentado se puede seguir de forma sencilla en el Código Fuente 31, que es el código del nuevo servlet. En este caso se ha optado por cambiar también la presentación de los datos de los parámetros, incluyéndolos dentro de una tabla.

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

public class ServletProcesaInfo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><BODY BGCOLOR=\"#FDF5E6\">" +
            "<TABLE BORDER=1 ALIGN=CENTER>" +
            "<TR BGCOLOR=\"#FFAD00\">" +
            "<TH>Nombre parámetro</TH><TH>Valor</TH>");
        Enumeration nombresParametros = request.getParameterNames();
        while(nombresParametros.hasMoreElements()) {
            String nombreParametro = (String)nombresParametros.nextElement();
            out.print("<TR><TD>" + nombreParametro + "</TD><TD>");
            String[] valores = request.getParameterValues(nombreParametro);
            if (valores.length == 1) {
                String valor = valores[0];
                if (valor.length() == 0)
                    out.println("<I>Sin valor</I>");
                else
                    out.println(valor);
            }
        }
    }
}
```

```

        out.println(valor);
    } else {
        out.println("<UL>");
        for(int i=0; i<valores.length; i++) {
            out.println("<LI>" + valores[i]);
        }
        out.println("</UL>");
    }
}
out.println("</TABLE></BODY></HTML>");
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    doGet(request, response);
}
}

```

Código Fuente 31

El resultado que muestra este nuevo servlet es el de la Figura 15. Como se puede comprobar el orden de los parámetros es significativo.

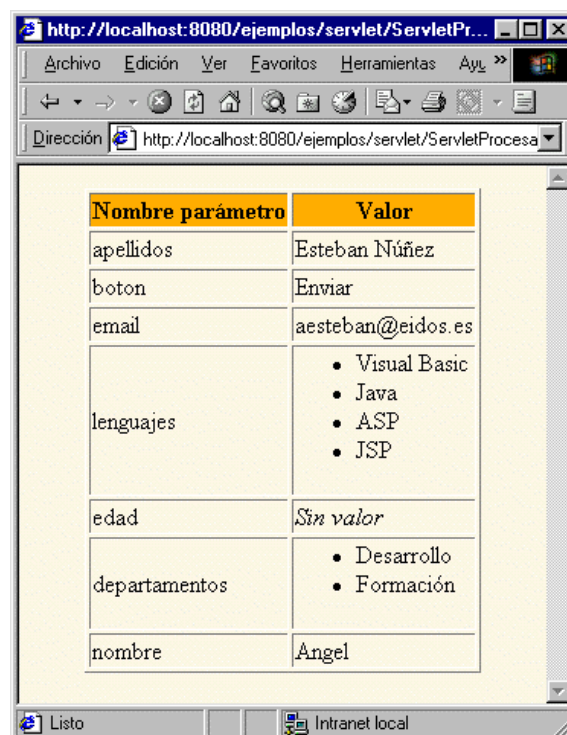


Figura 15. Nuevo aspecto del servlet

Con este ejemplo damos por terminado el capítulo dedicado al interfaz `javax.servlet.http.HttpServletRequest`. En el siguiente capítulo veremos el interfaz complementario, es decir, el que nos permite enviar la salida al usuario, se trata del interfaz `HttpServletResponse`, que ya hemos utilizado en muchos de los ejemplos vistos hasta ahora, obteniendo su flujo de salida para enviar información al usuario y también indicando al navegador el tipo de contenido que se le va a enviar.



# Servlets: el interfaz `HttpServletResponse`

---

## Introducción

En el capítulo anterior veíamos el interfaz que nos permitía obtener información de los usuarios, es decir, el interfaz `javax.servlet.http.HttpServletRequest`, y en este nuevo capítulo vamos a tratar el interfaz que nos permite enviar información al usuario, es decir, vamos a tratar el interfaz `javax.servlet.http.HttpServletResponse`.

Aunque el objeto `HttpServletResponse` ya lo hemos estado utilizando en muchos de los ejemplos del texto, ya sea a la hora de establecer el tipo de contenido que se corresponde con la salida del usuario o bien a la hora de obtener el flujo de salida que se va a utilizar para enviar información al usuario.

## El interfaz `HttpServletResponse`

En este apartado vamos a describir de forma breve los distintos métodos que pone a nuestra disposición el interfaz `HttpServletResponse`. Como ocurría con el interfaz `HttpServletRequest`, este nuevo interfaz hereda de una clase más general, que define para los servlets genéricos la forma de enviar información al usuario. El interfaz `HttpServletResponse` ofrece funcionalidad específica para que los servlets HTTP puedan generar una respuesta del protocolo HTTP para el cliente que realizó la petición del servlet.

En el siguiente apartado trataremos el interfaz `javax.servlet.ServletResponse`, ahora vamos a mostrar los distintos métodos del interfaz `javax.servlet.http.HttpServletResponse`.

- `void addCookie(Cookie cookie)`: añade la cookie indicada a la respuesta. En el tema correspondiente trataremos el mecanismo de las cookies y como lo podemos utilizar desde los servlets.
- `void addDateHeader(String nombre, long fecha)`: añade una cabecera de respuesta con el nombre y fecha indicados. Como se puede ver este interfaz ofrece métodos complementarios a los del interfaz `HttpServletRequest`. Los métodos que ofrecía el interfaz `HttpServletRequest` permitían obtener las cabeceras de petición HTTP, y los métodos que ofrece el interfaz `HttpServletResponse` permiten añadir cabeceras de respuesta HTTP.
- `void addHeader(String nombre, String valor)`: este método añade una cabecera de petición con el valor indicado.
- `void addIntHeader(String nombre, int valor)`: añade una cabecera de petición con el nombre dado y el valor entero indicado.
- `boolean containsHeader(String nombre)`: devuelve un valor booleano indicando si la cabecera de respuesta facilitada ha sido ya establecida.
- `String encodeRedirectURL(String url)`: codifica la URL especificada, devolviendo el resultado como un objeto `String`.
- `String encodeURL(String url)`: codifica la URL especificada incluyendo el identificador de sesión si es necesario. EL resultado se devuelve como un objeto `String`. En el capítulo en el que se tratará las sesiones veremos el significado del identificador de sesión y la forma de utilizar este método.
- `void sendError(int codigoDeEstado)`: envía una respuesta de error al cliente utilizando el código de estado indicado.
- `void sendError(int codigoDeEstado, String mensaje)`: envía una respuesta de error al cliente utilizando el código de estado especificado y el mensaje descriptivo.
- `void sendRedirect(String localización)`: envía una cabecera de respuesta de redirección al cliente utilizando la localización indicada por parámetro.
- `void setDateHeader(String nombre, long fecha)`: establece una cabecera de respuesta con el nombre indicado y la fecha indicada. La diferencia con el método `addDateHeader()`, es que `addDateHeader()` permite añadir nuevos valores a cabeceras existentes, y el método `setDateHeader()` establece un valor de la cabecera que sobrescribe el existente.
- `void setHeader(String nombre, String valor)`: establece una cabecera de respuesta con el nombre y valor indicados.
- `void setIntHeader(String nombre, int valor)`: establece una cabecera de respuesta con el nombre y valor entero indicados.
- `void setStatus(int codigoDeEstado)`: establece el código de estado para la respuesta correspondiente.

El uso de todos estos métodos lo veremos a lo largo del presente capítulo y también parte en el siguiente capítulo.

## El interfaz **ServletResponse**

En nuestros servlets también deberemos utilizar los métodos del interfaz padre del interfaz `HttpServletResponse`. Por lo tanto en este apartado vamos a exponer de forma breve los distintos métodos que nos ofrece este otro interfaz, que representa la respuesta de los servlets genéricos.

- `void flushBuffer()`: fuerza que cualquier contenido del búfer de salida sea enviado al cliente. Más adelante veremos como tratan los servlets el búfer de salida, ahora diremos simplemente que es un espacio de almacenamiento intermedio que contiene el contenido que se va enviando al cliente, es decir, al utilizar el objeto `PrintWriter` para enviar la respuesta al cliente, no se envía de manera inmediata, sino que pasa antes por el búfer.
- `int getBufferSize()`: devuelve el tamaño actual del búfer de la respuesta.
- `String getCharacterEncoding()`: devuelve el nombre del conjunto de caracteres utilizado por la respuesta que se envía al cliente.
- `Locale getLocale()`: devuelve el identificador local asignado a la respuesta, como un objeto de clase `java.util.Locale`.
- `ServletOutputStream getOutputStream()`: devuelve un flujo de salida de la clase `javax.servlet.ServletOutputStream`, utilizado para escribir datos binarios en la respuesta.
- `PrintWriter getWriter()`: devuelve un flujo de salida de la clase `java.io.PrintWriter`, que permite enviar datos en formato de texto al cliente. Este método será muy utilizado por los servlets, ya que es el que nos permite construir el documento que se le devuelve al cliente correspondiente.
- `boolean isCommitted()`: devuelve verdadero si la respuesta se ha enviado.
- `void reset()`: elimina cualquier información que se encuentre almacenada en el búfer así como cualquier código de estado o cabecera de respuesta.
- `void setBufferSize(int tamaño)`: establece el tamaño del búfer que se utilizará para enviar el cuerpo de la respuesta. El tamaño del búfer debe ser establecido antes de escribir cualquier contenido en el cuerpo de la respuesta, si se hace después se producirá una excepción.
- `void setContentLength(int longitud)`: este método permite especificar la longitud del contenido del cuerpo de la respuesta. En servlets HTTP, este método establece la cabecera del protocolo HTTP llamada `Content-Length`.
- `void setContentType(String type)`: establece el tipo de contenido de la respuesta enviada al cliente, para ello se utilizan los tipos MIME.
- `void setLocale(Locale local)`: establece la configuración local de la respuesta enviada al cliente.

## Cabeceras de respuesta HTTP

Cuando el servidor Web ha recibido las cabeceras de petición HTTP, se dispone a generar la respuesta correcta. El servidor comienza localizando la URL del método GET y entonces genera las cabeceras de respuesta. El método GET indica la URL deseada al servidor. Las otras cabeceras de petición HTTP indican al servidor cómo enviar los datos al navegador

Básicamente, las cabeceras de respuesta HTTP son la respuesta del servidor a la URL demandada por el cliente. Una vez que el servidor ha recibido la petición, debe elegir una respuesta válida.

La respuesta ofrecida por el servidor comienza con una línea de estado de respuesta. Esta línea está formada por la versión del protocolo, seguida por un código de estado. El formato de una línea de estado de respuesta es el siguiente:

```
Protocolo/version codigo_estado descripcion_estado
```

El protocolo será HTTP y su versión 1.1. Un ejemplo de línea de estado de respuesta válido podría ser el siguiente:

```
HTTP/1.1 200 OK
```

El protocolo es HTTP en su versión 1.1, el código de estado es 200 y la descripción del estado es OK. Esta línea nos indicaría que el servidor ha localizado la URL pedida por el navegador y va a enviársela al navegador.

Otra cabecera de respuesta HTTP es la línea de respuesta Date (fecha), un ejemplo válido de esta cabecera de respuesta sería:

```
Date: Sat, 31 Oct 1998 18:00:10 GMT
```

El ejemplo anterior nos indicaría la fecha y hora en la que el servidor generó la respuesta HTTP. Únicamente una cabecera de respuesta Date será permitida por mensaje, ya que esta cabecera es utilizada para evaluar las respuestas que se encuentren en la caché del navegador, el servidor debería incluir siempre una cabecera de respuesta Date.

La cabecera de respuesta Server contiene información acerca del software de servidor que se encuentra en la máquina servidor. Un ejemplo de esta cabecera de respuesta, sería la siguiente, que pertenece al servidor Web de Microsoft Internet Information Server 5.0:

```
Microsoft-IIS/5.0
```

La cabecera de respuesta Content-Type (tipo de contenido), indicará al navegador el tipo de información que se ha añadido después de la última cabecera de respuesta. Para indicar que se el servidor va a enviar al navegador una página HTML debería enviar la siguiente cabecera de respuesta Content-Type:

```
Content-Type: text/html
```

La cabecera de respuesta Content-Length (tamaño del contenido) contendrá el número de bytes de información que se envían como respuesta al navegador.

```
Content-length: n° bytes
```



La cabecera de respuesta `Last-Modified` indica la fecha/hora de modificación del recurso de la URL que se está devolviendo. Si se ha enviado previamente una cabecera de petición `If-Modified-Since`, es utilizada para determinar si los datos deberían ser transferidos o no.

La última línea de las cabeceras de respuesta debe ser una línea en blanco. Después de esta línea en blanco, el recurso indicado por la URL demandada se envía al cliente. Lo que hemos denominado anteriormente cuerpo de la respuesta.

Todas estas cabeceras de respuesta se deben tener en cuenta a la hora de escribir servlet, ya que si queremos devolver una salida que pueda entender el navegador, nuestro servlet deberá generar las cabeceras de respuesta mínimas para que el navegador pueda interpretar la respuesta de forma satisfactoria. Muchas veces se sólo necesario generar una cabecera de respuesta `Content-Type` para indicar el tipo de datos que va a devolver el servlet a continuación.

Todas las cabeceras de respuesta HTTP deben terminar con una línea en blanco. Todo lo que se encuentre a partir de esta línea en blanco se supone que está en el formato indicado por la cabecera `Content-Type`.

Otra cabecera de respuesta interesante que podremos devolver en algún momento desde nuestro servlet va a ser la cabecera `Location`. Esta cabecera redireccionará al navegador hacia otra URL. Su sintaxis es la siguiente:

`Location: URL`

Esta cabecera la podremos utilizar para enviar al navegador hacia otra URL. Esto puede ser útil para enviar a un usuario a diferentes versiones de nuestro sitio Web atendiendo al tipo de navegador que ha realizado la petición.

## Enviando información al cliente

La función principal del interfaz `HttpServletResponse` es la de permitir enviar información al cliente en forma de un documento HTML. Esto lo hemos podido comprobar en los ejemplos realizados hasta ahora a lo largo del texto. Antes de empezar a generar el documento, nuestro servlet debe especificar el tipo de contenido que se va a enviar al navegador, para que éste lo pueda interpretar de forma correcta.

Para ello utilizaremos el método `setContentType()`, pasándole por parámetro un objeto de la clase `String` que se corresponde con el tipo MIME (Multipurpose Internet Mail Extensión) del documento, en casi todos los casos el servlet indica el tipo de contenido `text/html`. Los tipos MIME más comunes son los que se ofrecen en la Tabla 7.

Tipo MIME	Descripción
Application	Indica al servidor que aplicación ejecutar basándose en la extensión del fichero.
Audio	Especifica el tipo de audio que puede tratar el navegador. Suele ser basic, x-aiff y x-wav.

Image	Especifica el tipo de imagen que puede tratar el navegador. Suele ser gif y jpeg.
Text	Especifica el tipo de texto que puede ser manejado por el navegador. Comúnmente los tipos de texto son: html, plain, rich text y x-setext.
Video	Especifica el tipo de video que puede ser tratado por el navegador. Suele ser: mpeg y quicktime.

Tabla 7

Los valores más comunes de estos tipos y subtipos MIME son los mostrados en la tabla que aparece a continuación (Tabla 8).

Tipo/subtipo	Descripción
text/html	Código HTML. Especifica que el navegador del cliente debería interpretar el código HTML a mostrar
text/plain	Indica un texto ASCII. Especifica que el navegador debería únicamente mostrar texto sin interpretar código.
image/gif	Un fichero gráfico GIF. El navegador debería mostrar la imagen o iniciar una aplicación que pueda visualizar la imagen.
image/jpeg	Un fichero gráfico JPEG.

Tabla 8

El tipo de contenido también lo podemos establecer utilizando el método `setHeader()` especificando que la cabecera que se va a establecer es `Content-Type`, aunque lo más usual es utilizar el método `setContentType()`. En el Código Fuente 32 se muestran las dos formas equivalentes de establecer el tipo de contenido desde un servlet.

```
response.setHeader("Content-Type", "text/html");
response.setContentType("text/html");
```

Código Fuente 32

Lo primero que se debe hacer es establecer el tipo de contenido, las cabeceras HTTP de respuesta y el código de estado. Esto es debido a que una respuesta HTTP se compone de la línea de estado, una a o varias cabeceras de respuesta, una línea en blanco y el cuerpo de la respuesta, todo en ese orden. Aunque realmente en el inicio de un servlet lo único obligatorio es establecer el tipo de contenido.

Una vez establecido el tipo de contenido que va a generar nuestro servlet, podremos obtener el objeto `PrintWriter`, que pertenece al objeto `HttpServletResponse`, para enviar la información que forma parte del documento que se va a ir generando. Para ello utilizaremos el método `getWriter()` del interfaz `HttpServletResponse`.

La clase `PrintWriter` pertenece al paquete `java.io` y representa un flujo o canal de salida que envía los datos en modo de texto. Los métodos que vamos a utilizar de la clase `PrintWriter` dentro de nuestros servlets van a ser sobre todo los métodos `print()` y `println()`. Ambos métodos reciben como parámetro la cadena (objeto `String`) que se quiere devolver a través del flujo de salida, y la única diferencia entre los dos métodos es que el método `println()` añade un salto de línea al final de la cadena.

En los servlets, a la hora de generar el código HTML a través del objeto `PrintWriter`, es indiferente utilizar el método `print()` que el método `println()`, ya que el navegador Web no va a interpretar los saltos de línea.

El flujo de salida `PrintWriter` utiliza un búfer de almacenamiento intermedio al que se va enviando la información que se va devolviendo con los distintas llamadas al método `print()` o `println()`. Este buffer tiene un tamaño que podemos obtener mediante el método `getBufferSize()`, y también podemos establecer su tamaño utilizando el método `setBufferSize()`, ambos métodos pertenecen al interfaz `javax.servlet.ServletResponse`. El tamaño del búfer se expresa en bytes, y por defecto el tamaño del búfer es de 8192 bytes, o lo que es lo mismo 8 KB.

El contenido que se va generando se enviará al cliente según se vaya llenado el búfer, una vez que se ha enviado algún tipo de contenido al cliente (aquí entendemos por contenido información que forma parte del cuerpo de la respuesta HTTP, como por ejemplo código HTML) no es posible cambiar o añadir cabeceras de respuesta, tampoco se puede redirigir al cliente a otra página o recurso.

El contenido se enviará al cliente bien porque se haya llenado el búfer o bien porque el servlet lo ha forzado llamando al método `flushBuffer()` del interfaz `ServletResponse`, este método enviará el contenido del búfer al cliente. El contenido del búfer también se envía cuando el servlet finaliza con su ejecución.

Un método relacionado directamente relacionado con el búfer de la respuesta es el método `isCommitted()` del interfaz `ServletResponse`, este método devolverá un valor booleano indicando si se ha enviado algún contenido al usuario o no, es decir, si el búfer ya ha sido enviado en algún momento.

Para entender mejor todos estos conceptos y métodos relacionados con el envío de una respuesta al cliente, vamos a mostrar y comentar el Código Fuente 33.

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

public class ServletSalida extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        out.println("<br>Búfer enviado:"+response.isCommitted()+"</body></html>");
    }
}
```

```

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}

```

Código Fuente 33

El resultado de la ejecución de este servlet es el de la Figura 16.

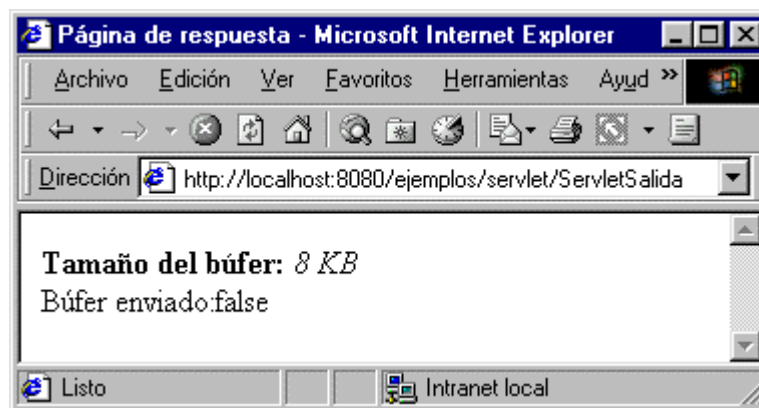


Figura 16

Si ejecutamos este código nos mostrará el tamaño del búfer y nos indicará si el búfer ha sido enviado o no. Como se puede comprobar el servlet nos indica que el búfer no ha sido enviado en el momento en el que hemos lanzado el método `isCommitted()` sobre el objeto `response`. Esto es así porque no se ha realizado ninguna llamada al método `flushBuffer()`, ni tampoco se ha sobrepasado el tamaño establecido para el búfer.

Pero si ejecutamos el servlet del Código Fuente 34, obtenemos el resultado de la Figura 17, ya que en este caso sí que se ha llamado al método `flushBuffer()`, y por lo tanto el contenido del búfer se ha enviado al cliente.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletSalida extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        response.flushBuffer();
        out.println("<br>Búfer enviado:"+response.isCommitted()+"</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

```

```

        doGet(request, response);
    }
}

```

Código Fuente 34

El método `reset()` del interfaz `javax.servlet.ServletResponse` vacía el contenido del búfer, pero sin enviarlo al cliente, es decir, se descarta el contenido generado hasta el momento, si ya se ha enviado algún contenido al cliente se producirá un error. Así si modificamos alguno de los ejemplos anteriores, realizando una llamada al método `reset()`, el servlet nos devolverá una página vacía o bien un error si hemos realizado una llamada al método `flushBuffer()` previamente.

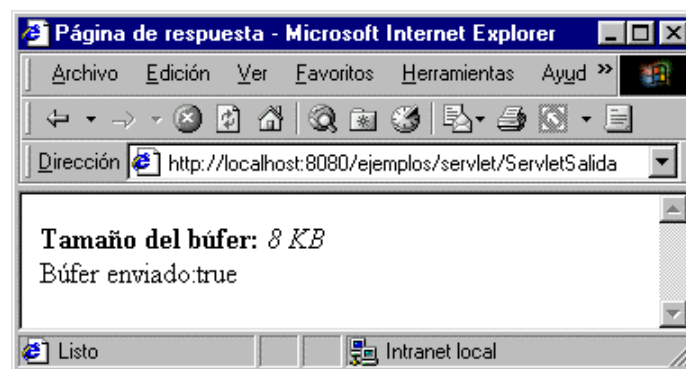


Figura 17

Así el servlet que se corresponde con el Código Fuente 35, producirá el error que se muestra en la Figura 18.

```

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

public class ServletSalida extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"+
        );
        response.flushBuffer();
        out.println("<br>Búfer enviado:"+response.isCommitted()+"</body></html>");
        response.reset();
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request, response);
    }
}

```

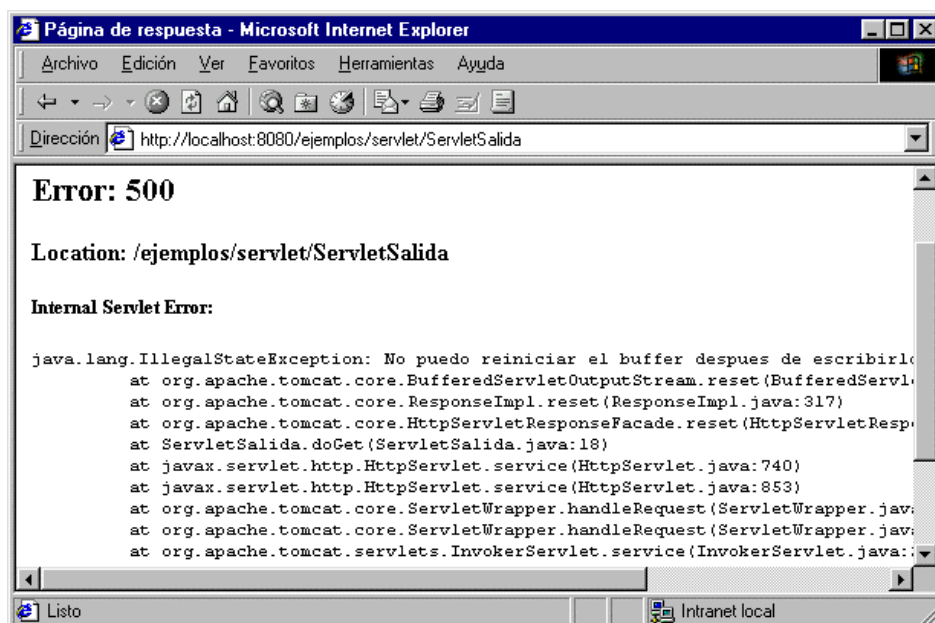
Código Fuente 35

Un método que también producirá un error si ya hemos enviado alguna parte del contenido del cuerpo de la respuesta, es el método `sendRedirect()` del interfaz `HttpServletResponse`. Este método produce una redirección en la URL que se va a cargar en el cliente, así el resultado del Código Fuente 36 es el de la Figura 19, como se puede observar la URL a la que hemos redirigido al cliente no aparece en la dirección del navegador.

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

public class ServletSalida extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        out.println("<br>Búfer enviado:"+response.isCommitted()+"</body></html>");
        response.sendRedirect("/ejemplos/servlet/ServletHolaMundo");
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 36

Figura 18. Error al utilizar el método `reset()`

Como se puede ver se ha redirigido al cliente a otro servlet sin ningún problema, ya que el búfer no ha sido enviado, pero si antes de la llamada al método `sendRedirect()` añadimos una llamada al método `flushBuffer()`, al ejecutar el servlet aparecerá un error similar al de la Figura 20.



Figura 19. Redirección en un servlet

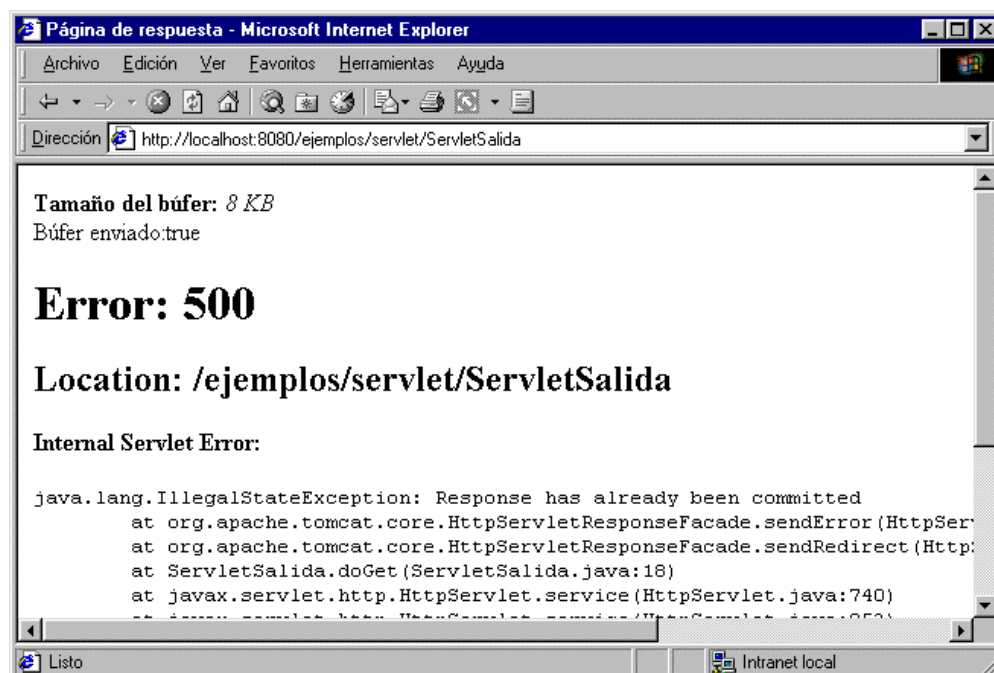


Figura 20. Error en la redirección en un servlet

Haciendo uso del interfaz `HttpServletResponse` y del interfaz `HttpServletRequest` podemos realizar un sencillo servlet que redirija el navegador del usuario a la URL que le indiquemos en un campo de texto de un formulario. El código de este servlet es tan sencillo como el que se muestra a continuación (Código Fuente 37).

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

public class ServletRedireccion extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.sendRedirect(request.getParameter("URL"));
    }
}
```

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{
    doGet(request,response);
}
}
```

Código Fuente 37

Y en el Código Fuente 38 se puede observar el código HTML que realiza la llamada al servlet anterior para que realice la redirección.

```
<html>
<head>
<title>Redirecciona</title>
</head>
<body>
<form method="GET" action="servlet/ServletRedireccion">
URL:<input type="text" name="URL" size="30">
<input type="submit" name="enviar" value="Cargar URL">
</form>
</body>
</html>
```

Código Fuente 38

## Códigos de estado

Mediante el interfaz `HttpServletResponse` también podemos especificar códigos de estado al cliente, para ello el interfaz `HttpServletResponse` ofrece el método `setStatus()`.

Ya comentamos en apartados anteriores que el código de estado y la descripción del mismo forman parte de la línea de estado de una respuesta http. Los códigos de estado informan al cliente del resultado de la petición realizada, así si la petición se ha servido con éxito, así por ejemplo si el recurso que se ha pedido se ha devuelto de forma correcta se devolverá un código de estado 200 (OK), para indicar que todo ha sido correcto.

Un código de estado se compone de un entero y una breve descripción del estado. Al método `setStatus()` se le pasa por parámetro un entero, que se corresponderá con el valor de una de las constantes del interfaz `HttpServletResponse`, que representan los distintos códigos de estado. Así por ejemplo el código de recurso no encontrado, que es el código 404 (Not Found), se corresponde con la constante `HttpServletResponse.SC_NOT_FOUND`.

Los códigos de estado los podemos agrupar en cinco categorías, atendiendo al estado al que se refieren cada uno de ellos, estas categorías son:

- Códigos de estado informativos: indican que el cliente debería responder con alguna otra acción, se numeran en los 100 (100-199). Por ejemplo el código 100 (Continue), representado por la constante `SC_CONTINUE`, indica al cliente que puede continuar con la operación demandada.
- Códigos de estado de éxito: se numeran en los 200 (200-299). Estos códigos de estado indican el éxito de la petición realizada. Por ejemplo el código de estado 201 (Created), representado



por la constante `SC_CREATED`, indica que el servidor a creado un nuevo documento en respuesta a la petición realizada.

- **Códigos de estado de redirección:** se encuentran en los 300 (300-399). Indicarán que una URL se ha movido a una localización diferente. Así por ejemplo el código de estado 301 (Moved Permanently), representado por el constante `SC_MOVED_PERMANENTLY`, indica que el documento pedido ha sido movido a una nueva dirección, causando por lo tanto la redirección del navegador a esta nueva dirección.
- **Códigos de error del cliente:** se encuentran en los 400 (400-499). Son errores que se generan en el navegador, es decir, en el cliente, son de los más útiles. De esta forma el código de estado 404 (Not Found), representado por la constante `SC_NOT_FOUND`, indica que la dirección indicada por el cliente no se ha podido localizar en el servidor.
- **Códigos de error del servidor:** se encuentran en los 500 (500-599). Representan los errores que se producen en el servidor. Por ejemplo el código de estado 505 (HTTP Versión Not Supported), representado por la constante `SC_HTTP_VERSION_NOT_SUPPORTED` del interfaz `HttpServletResponse`, significa que el servidor no soporta la versión del protocolo HTTP especificado en la petición.

En la Tabla 9 se muestran los códigos de estado más comunes, junto con su descripción y una breve explicación de los mismos.

Código	Descripción	Significado y constante
200	OK	La petición se ha servido sin problemas. ( <code>SC_OK</code> )
202	Accepted	La petición ha sido aceptada pero todavía se está procesando. ( <code>SC_ACCEPTED</code> )
301	Moved Permanently	El documento se ha trasladado a una nueva localización. ( <code>SC_MOVED_PERMANENTLY</code> )
302	Found	El documento está en el servidor pero en una localización diferente. ( <code>SC_MOVED_TEMPORARILY</code> )
400	Bad Request	La sintaxis de la petición es incorrecta. ( <code>SC_BAD_REQUEST</code> )
401	Unauthorized	El servidor tiene restricciones sobre este documento. ( <code>SC_UNAUTHORIZED</code> )
403	Forbidden	La petición se ha prohibido, debido a derechos de accesos o por otras razones. ( <code>SC_FORBIDDEN</code> )
404	Not Found	La petición no se ha podido encontrar. ( <code>SC_NOT_FOUND</code> )
405	Method Not Allowed	El método con el que se ha realizado la petición (GET, POST, PUT, etc.) no se permite para el recurso. ( <code>SC_METHOD_NOT_ALLOWED</code> )

500	Internal Server Error	El servidor ha fallado inesperadamente al intentar servir la petición. (SC_INTERNAL_SERVER_ERROR)
503	Service Unavailable	El servidor no puede responder debido a una sobrecarga en el servicio o porque se encuentra en mantenimiento. (SC_SERVICE_UNAVAILABLE)

Tabla 9. Códigos de estado

Los códigos de estado se deben establecer antes de enviar cualquier contenido del cuerpo de la respuesta HTTP al cliente, es decir, antes de que se haya enviado el contenido del búfer.

En el Código Fuente 39 se puede ver un servlet que genera un código de estado para indicar que la petición se ha realizado de forma correcta.

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

public class ServletCodigosEstado extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setStatus(response.SC_OK);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Códigos de estado</title></head>"+
            "<body><h1>Servlet que genera códigos de estado"+
            "</h1></body></html>"+
        );
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 39

Si cambiamos el código anterior, indicando el código de estado de error interno del servidor (500, SC\_INTERNAL\_SERVER\_ERROR, el navegador mostrará la página de error típica que se muestra cuando se produce este tipo de errores, la ejecución de este servlet se puede ver en la Figura 21.

El siguiente servlet recoge el código de estado que se le pasa como parámetro en la petición, y lo establece en la respuesta al cliente (Código Fuente 40). Este ejemplo puede ser útil para probar la utilización de los distintos códigos de estado, y el resultado que recibe el cliente.

La página HTML que realiza la llamada a este servlet es tan sencilla como el Código Fuente 41.

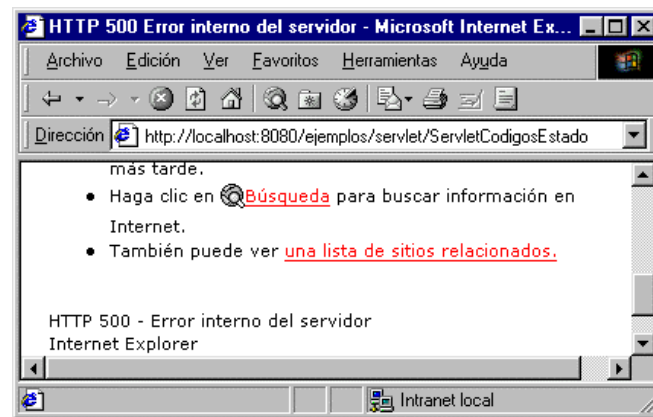


Figura 21. Código de estado 500

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

public class ServletCodigosEstado extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        int codigo=Integer.parseInt(request.getParameter("codigo"));
        response.setStatus(codigo);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Códigos de estado</title></head>"+
            "<body><h1>Servlet que genera códigos de estado"+
            "</h1></body></html>"+
        );
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 40

```
<html>
<head>
<title>Códigos de estado</title>
</head>
<body>
<form method="GET" action="servlet/ServletCodigosEstado">
Código de estado:<input type="text" name="codigo" maxlength="3" size="3">
<input type="submit" name="enviar" value="Enviar código">
</form>
</body>
</html>
```

Código Fuente 41

En el ejemplo anterior se pueden dar varias situaciones de error muy claras, una de las puede ser que el cliente indique un código no numérico, y otra situación es que el cliente puede dejar el campo del formulario vacío, por el contrario, si especificamos un código de estado no existente, el navegador simplemente lo ignorará, sin embargo si que se producirá un error al nivel del contenedor de servlets, es decir, el servidor Jakarta Tomcat mostrará en su pantalla de error, un error como el de la Figura 22, pero el cliente no podrá percibir este error.

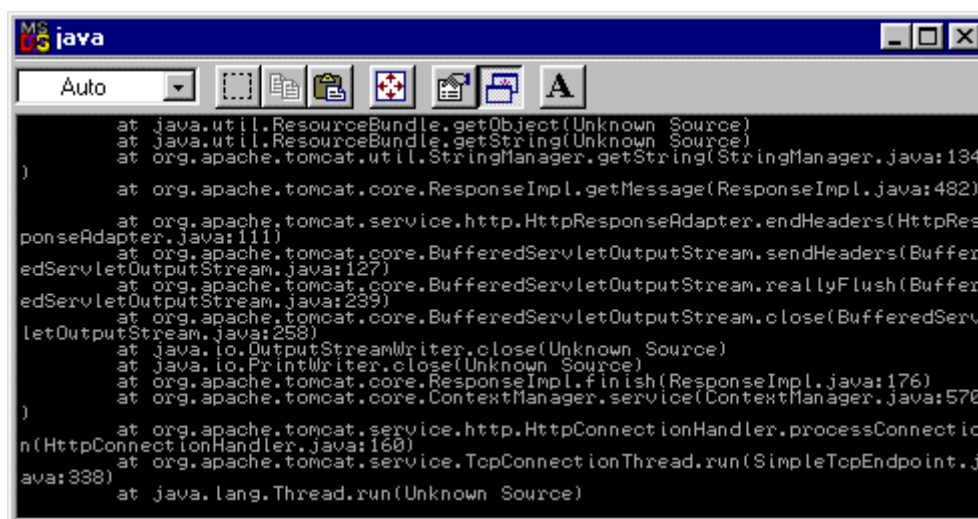


Figura 22. Error de código de estado no existente

Para manejar el error que se producirá al lanzar el método `parseInt()` de la clase `Integer`, debemos atrapar la excepción `NumberFormatException`. Si se produce una excepción de este tipo llamaremos al método `sendError()` del interfaz `HttpServletResponse`, a este método le pasaremos un código de estado dentro del rango de los códigos de estado que indican un error por parte del cliente, y una cadena que describe el error que se ha producido y que realiza la función de mensaje informativo para el cliente.

El método `sendError()` enviará el código de estado especificado al cliente y la cadena que contiene el mensaje se insertará dentro del código HTML.

En nuestro caso hemos utilizado un código de estado con el número 420 y el mensaje indica que el formato del número es incorrecto. Además utilizamos una variable booleana para indicar si se ha producido un error o no. El código fuente completo de esta nueva versión del servlet se muestra a continuación (Código Fuente 42).

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

public class ServletCodigosEstado extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        boolean error=false;
        int codigo=200;
        try{
            codigo=Integer.parseInt(request.getParameter("codigo"));
        }catch(Exception NumberFormatException){
            response.sendError(420,"<h1>Formato numérico incorrecto</h1>");
            error=true;
        }
    }
}
```

```

    }
    if(!error){
        response.setStatus(codigo);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Códigos de estado</title></head>"+
            "<body><h1>Servlet que genera códigos de estado"+
            "</h1></body></html>");
    }
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{
    doGet(request,response);
}
}

```

Código Fuente 42

Así si en el formulario indicamos un código de estado no numérico, el servlet nos mostrará la siguiente página de error (Figura 23).

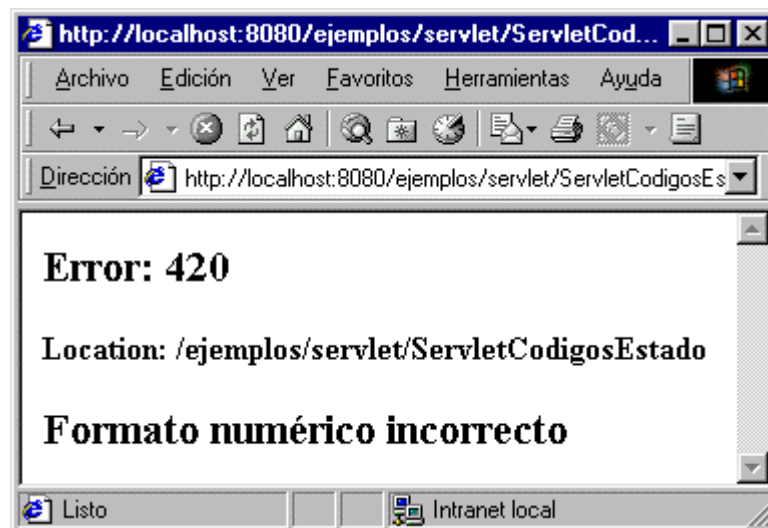


Figura 23. Error generado por el servlet

Podemos pensar que para solucionar el error que se muestra en la Figura 22 es posible utilizar un mecanismo similar, pero aunque situemos la sentencia `response.setStatus()` en un bloque `try{}catch` para atraparla una excepción genérica de la clase `Exception`, el error se sigue produciendo.

Con este apartado finalizamos el capítulo dedicado al interfaz `HttpServletResponse`, pero en un capítulo siguiente cuando tratemos el interfaz `HttpSession` comentaremos también algunos métodos del interfaz `HttpServletResponse` relacionados con la utilización de sesiones. De momento en el siguiente capítulo vamos a ocuparnos de otros elementos del API Java servlet 2.2, se trata de la clase `Cookie` y del interfaz `RequestDispatcher`.



# Servlets: La clase `Cookie` y el interfaz `RequestDispatcher`

---

## Introducción

En este nuevo capítulo vamos a tratar dos temas bien diferenciados, por un lado trataremos la clase `Cookie` y por otro trataremos el interfaz `javax.servlet.Dispatcher`.

Veremos como utilizar cookies para almacenar y recuperar información del cliente entre diferentes conexiones haciendo uso de la clase `javax.servlet.http.Cookie`, las cookies están relacionadas con el contenido que trataremos en el siguiente capítulo, en el que se realizará el mantenimiento de una sesión dentro de distintos servlets haciendo uso del interfaz `javax.servlet.http.HttpSession`.

Una cookie es un pequeño fragmento de información enviado por un servlet a un navegador Web, almacenado entonces por el navegador, y enviado más tarde al servidor a través de la petición HTTP realizada por el cliente.

Las sesiones nos permiten almacenar información particular de cada uno de los usuarios conectados, esta información se mantiene entre las distintas peticiones realizadas por el usuario dentro de la aplicación Web. Las sesiones utilizan internamente el mecanismo de las cookies.

El segundo punto importante de este capítulo es el interfaz `RequestDispatcher` que va a representar a un recurso dentro del servidor, que podrá ser una página JSP, un servlet, una página HTML, etc., y que nos va a permitir redirigir la petición de un cliente a ese recurso, o bien incluir el resultado de la ejecución del recurso o su contenido, dentro del servlet actual.

Vamos a comenzar este capítulo centrándonos en la definición de las cookies detallando las ventajas que ofrecen y las desventajas, así como en que escenarios las podemos utilizar.

## Definición de cookies

Una cookie, físicamente, es un fichero que se escribe en la máquina local del cliente que se conecta a un sitio Web y que contiene información relativa a la conexión.

Una cookie es utilizada para mantener información entre diferentes conexiones HTTP. Se debe recordar que el protocolo HTTP es un protocolo sin estado, es decir, no se retiene información entre las diferentes conexiones que se realicen. Por esta razón, ni el cliente ni el servidor pueden mantener información entre diferentes peticiones o a través de diferentes páginas Web.

Este mecanismo para mantener información entre diferentes conexiones HTTP fue propuesto e implementado en primera instancia por la compañía Netscape, más tarde pasó a formar parte del protocolo estándar HTTP.

Existen varios usos prácticos de las cookies, a continuación se van a comentar los más destacados:

- Para almacenar información acerca de las preferencias del cliente que se conecta a nuestro sitio Web, por ejemplo el color seleccionado de la página, el tipo de letra, etc.
- Para conservar información personal del usuario, como puede ser el nombre, el país de origen, código postal, el número de veces que ha accedido a nuestro sitio Web, etc.
- Para identificar a un usuario durante una sesión de comercio electrónico.

Por lo tanto el uso de cookies nos puede permite personalizar las páginas generadas por los servlets según el cliente que se haya conectado y sus preferencias y datos personales, por ejemplo podemos saludarle con su nombre y asignar al color de fondo la página con su color favorito o también podremos indicarle el número de veces que ha accedido a nuestro sitio Web. De esta forma podemos evitar preguntar al usuario sus preferencias o datos personales cada vez que entra en nuestro sitio Web.

Siempre debe haber una primera ocasión en la que el cliente conectado especifique el valor de la cookie, una vez especificado este valor ya puede ser utilizada la cookie en las diferentes conexiones que realice ese cliente, ya que la información ha quedado almacenada en la cookie.

Esta información se almacena físicamente en un fichero. En el caso del navegador Internet Explorer de Microsoft las cookies se almacenan en ficheros de texto que se encuentran en el directorio de Windows y en el subdirectorio Cookies, y en el caso del navegador Communicator de Netscape las cookies se almacenan todas en un fichero llamado cookies.txt en el subdirectorio Netscape\User\usuario.

Hay una serie de consideraciones que se deben tener en cuenta a la hora de utilizar cookies en nuestra aplicación Web:

- Las cookies se pueden perder, por lo tanto nunca se debe depender de las cookies para almacenar una información que no se pueda volver a generar. Este tipo de información se debería almacenar en una base de datos del servidor. No se debe olvidar que las cookies se almacenan en el disco local del cliente en ficheros, y por lo tanto estos ficheros se pueden dañar, ser borrados o sobrescritos.



- Las cookies pueden ser modificadas por el cliente, por lo tanto nunca se debe considerar que la información ofrecida por una cookie es auténtica. Si estamos usando una cookie para determinar la fecha de la última vez que un usuario visitó nuestro sitio Web podemos considerar como auténtica esta información sin ningún problema, pero sin embargo no es nada recomendable considerar auténtica una cookie que posee el número de cuenta de un usuario. Como regla general nunca se debe utilizar una cookie para almacenar información confidencial, este tipo de información se debería almacenar en una base de datos en el servidor Web.
- Las cookies pueden ser copiadas sin el consentimiento del propietario, nunca se debería utilizar una cookie para identificar a un usuario, una solución mejor es utilizar una contraseña y validarla con una base de datos del servidor.
- Algunos navegadores no reconocen las cookies. Incluso los más populares como es el caso de Internet Explorer y Netscape Communicator, se pueden configurar para no utilizar cookies. En la Figura 24, se puede observar la pantalla de configuración de seguridad del navegador Internet Explorer, que permite deshabilitar las cookies o indicar que se avise cuando se van a utilizar. Netscape Communicator ofrece también una opción muy similar.
- Las cookies tienen muy mala fama en lo que a amenazas de seguridad se refiere, por lo que muchos usuarios las desactivan en sus navegadores.

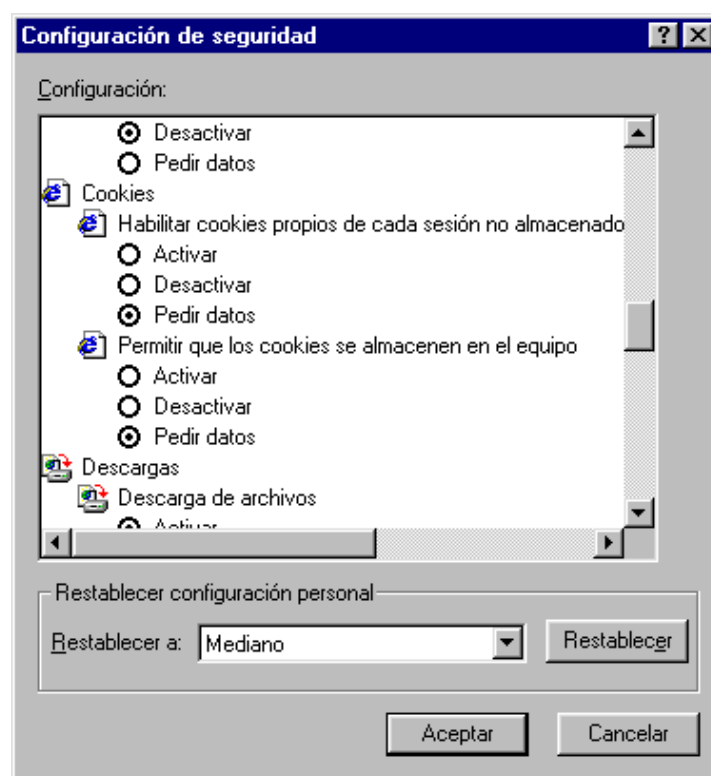


Figura 24. Configuración del navegador para la aceptación de cookies

En realidad las cookies no suponen una grave amenaza a la seguridad de las máquinas de los clientes, las cookies no se pueden interpretar ni ejecutar, por lo tanto es bastante improbable que puedan atacar el sistema local del cliente o introducir un virus. Normalmente los navegadores únicamente aceptan veinte cookies por sitio Web y como máximo un total de 300, cada cookie se limita a 4 KB, por lo tanto no se pueden utilizar las cookies para llenar el disco de la máquina del cliente.

Sin embargo las cookies si que plantean un problema de privacidad, ya que algún cliente puede estar interesado en no dejar ninguna huella en el sitio Web al que ha accedido. Otra amenaza a la privacidad la ofrecen sitios Web que recogen información sensible dentro de las cookies.

Se debe señalar que los servlets para mantener estados, es decir, valores de variables, objetos..., entre diferentes servlets de una aplicación, la especificación Java servlet 2.2 utiliza cookies. El uso de estas cookies por parte de los servlets, se oculta completamente al programador para que se abstraiga de la utilización lógica de las cookies, todo ello gracias al interfaz `javax.servlet.http.HttpSession` que las gestionan internamente, más tarde en este mismo capítulo trataremos en detenimiento la utilización de sesiones desde los servlets. Como vemos el uso de cookies y de sesiones se encuentra muy relacionados.

Así por ejemplo, cuando un servlet se inicia una sesión se crea la cookie `JSESSIONID` (cookie de inicio de sesión), aunque esta cookie no se grabará en el disco duro del cliente, sino que permanecerá en memoria, por lo tanto podemos asegurar al usuario que no vamos a acceder a su disco, ya que la cookie permanece en memoria del navegador, y sólo existirá mientras el navegador esté abierto.

De esta forma si un navegador no acepta cookies, la aplicación Web no podrá mantener el estado entre los diferentes servlets utilizando cookies, pero entonces recurrirá a una alternativa denominada sobrescritura de URLs (URL-Rewriting), que permite mantener la sesión del usuario simulando la utilización de cookies, cuanto tratemos el interfaz `HttpSession` veremos que el interfaz `HttpServletResponse` ofrece el método `encodeURL()` para posibilitar el mecanismo de sobrescritura de URLs.

Si tenemos configurado nuestro navegador para que nos avise cuando recibe una cookie, al iniciar una sesión con un servlet (que utilice la sesión) de una aplicación aparecerá una ventana similar a la que se muestra en la Figura 25.

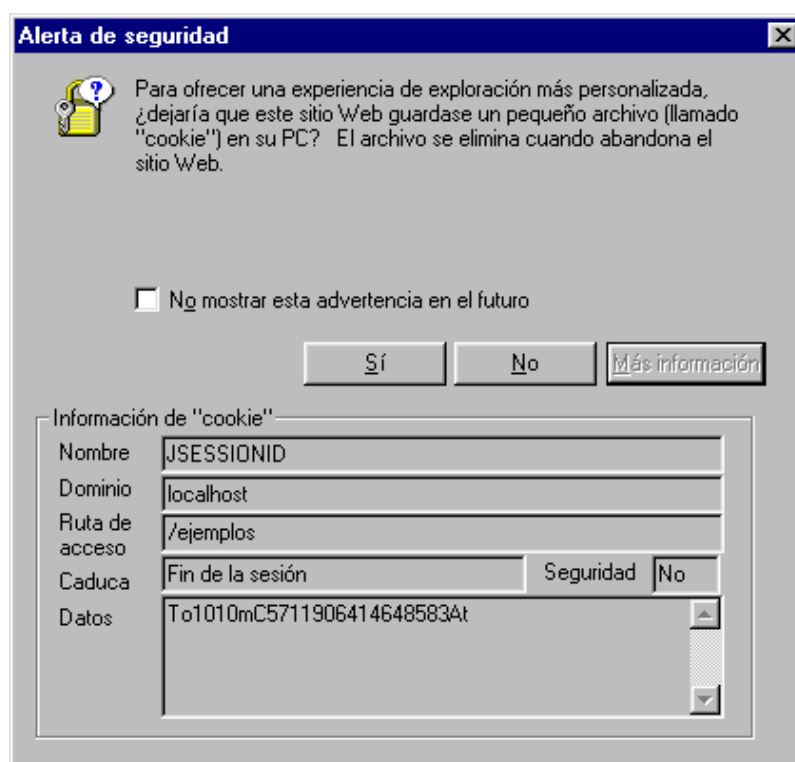


Figura 25. Cookie de inicio de sesión

Esta cookie es utilizada para identificar de forma única cada una de las sesiones que se corresponden con cada uno de los usuarios conectados actualmente a la aplicación Web. Como se puede comprobar en la Figura 25, también se nos muestra información acerca del valor de las propiedades de la cookie de inicio de sesión. Más adelante veremos que crear cookies y especificar sus valores y propiedades.

Ahora vamos a pasar a describir la clase que nos permite manipular las cookies, ya sea para leer sus valores, modificarlos y o crearlas.

## La clase Cookie

Esta clase perteneciente al paquete `javax.servlet.http` representa una cookie la cual podremos manipular dentro de nuestros servlets.

El constructor que presenta esta clase recibe por parámetro una cadena que va a representar el nombre de la cookie, y otra cadena que va a ser el valor que posee la cookie que se crea. A continuación se ofrece el aspecto de este constructor:

- `Cookie(String nombre, String valor)`

A continuación vamos a describir los métodos que ofrece esta clase para manipular las cookies:

- `Object clone()`: este método sobrescribe el método `clone()` estándar de la clase `Object`, para devolver una copia de una cookie.
- `String getComment()`: devuelve el comentario que describe el propósito de la cookie. Si la cookie no posee comentario se devolverá un valor nulo (`null`).
- `String getDomain()`: devuelve el dominio al que pertenece la cookie.
- `int getMaxAge()`: devuelve la duración máxima de una cookie (caducidad), especificado en segundos, por defecto, se establece a `-1`, indicando que la cookie permanecerá hasta que el navegador se cierre.
- `String getName()`: devuelve el nombre de la cookie.
- `String getPath()`: devuelve la ruta de acceso del servidor para la cual se devuelve esta cookie.
- `boolean getSecure()`: indica si el navegador ha enviado la cookie a través de un protocolo seguro.
- `String getValue()`: devuelve el valor de la cookie.
- `int getVersion()`: devuelve la versión de la especificación a la que pertenece la cookie.
- `void setComment(String propósito)`: establece el comentario que describe el propósito de la cookie.
- `void setDomain(String patrón)`: especifica el dominio al que pertenece la cookie.
- `void setMaxAge(int caducidad)`: establece, en segundos, el tiempo máximo (caducidad) de validez de la cookie. Por defecto el valor que posee la caducidad de una cookie

es de  $-1$ , por lo que la cookie sólo existirá en la sesión actual del navegador. Si establecemos el valor 0 se eliminará la cookie.

- `void setPath(String uri)`: establece la ruta para la cual el cliente debería devolver la cookie.
- `void setSecure(boolean segura)`: indica al navegador si la cookie debería ser enviada únicamente utilizando un protocolo seguro, como puede ser HTTPS.
- `void setValue(String valor)`: establece el valor de una cookie.
- `void setVersion(int versión)`: indica la versión de la especificación a la que pertenece la cookie. Existen dos versiones actualmente, la 0 y la 1.

Como se puede extraer de los distintos métodos que ofrece la clase `Cookie`, una cookie posee un nombre, un valor único, y otros atributos opcionales como pueden ser un comentario, una ruta de acceso, el dominio al que pertenece, la duración de la misma y el número de versión a la que pertenece.

Un servlet envía cookies al navegador Web utilizando el método `addCookie()` del interfaz `HttpServletResponse`, este método añadirá las cabeceras de respuesta HTTP necesarias para enviar las cookies al navegador. El navegador puede soportar 20 cookies por cada sitio Web, 300 en total, y cada cookie puede tener como máximo un tamaño de 4KB.

El navegador devuelve las cookies a un servlet añadiendo campos a las cabeceras de petición HTTP. Las cookies se pueden obtener de una petición utilizando el método `getCookies()` del interfaz `HttpServletRequest`.

La clase `Cookies` soporta cookies pertenecientes a las versiones 0 y 1 de la especificación de las cookies. Por defecto una cookie se creará utilizando la versión 0, más adelante veremos la diferencia existente entre ambas versiones.

En los siguientes apartados trataremos con detenimiento tanto la creación de cookies como la recuperación de cookies existentes.

## Creación de cookies

El primer paso para crear una cookie es crear una instancia de un objeto de la clase utilizando el constructor correspondiente. Como vimos en el apartado anterior, a este constructor se le deben pasar dos parámetros que son el nombre y el valor de la cookie. Se debe señalar que las cookies son case-sensitive en lo que a su nombre se refiere, y por lo tanto no será lo mismo crear una cookie llamada `nombreCookie`, que otra llamada `nombrecookie`, serán dos cookies distintas.

Una vez creada la cookie deberemos establecer sus propiedades de manera adecuada utilizando los distintos métodos `setXXX` que la clase `Cookie` pone a nuestra disposición.

Para indicar un comentario relativo al propósito que tiene la cookie utilizaremos el método `setComment()`, en la versión 0 de las cookies este comentario no se envía al cliente, siendo únicamente de carácter informativo para el servidor.

Normalmente el navegador sólo devuelve las cookies que coinciden con el nombre del dominio del servidor que las envió, pero si queremos que se apliquen a otros dominios deberemos utilizar la siguiente llamada: `cookie.setDomain(".nombre.com")`, así las cookies se enviarán a todos los dominios

que finalicen con nombre.com, por ejemplo dominio1.nombre.com y dominio2.nombre.com. Por defecto el dominio de una cookie es el dominio en el que se creó la misma.

También puede ser necesario especificar el tiempo de caducidad de la cookie, para ello utilizaremos el método `setMaxAge()` indicando los segundos de duración de la misma. Pasado este tiempo la cookie se eliminará. Por defecto el valor que posee la caducidad de una cookie es de `-1`, por lo que la cookie sólo existirá en la sesión actual del navegador. Si establecemos el valor `0` se eliminará la cookie.

Además de establecer la caducidad y el dominio de la cookie creada, también nos puede interesar indicar la ruta para la que se aplica la cookie. El navegador sólo devolverá las cookies que se correspondan con la ruta a la que pertenece la cookie, por defecto el navegador devolverá la cookie sólo a la URL o directorios inferiores que contenía la página que envió la cookie. Por ejemplo si el servidor envió la cookie desde la URL `http://mi.sitio.es/dir1/pagina.html`, el navegador devolverá la cookie cuando se conecte a `http://mi.sitio.es/dir1/subdir/otrapagina.html`, pero no cuando se conecte a `http://mi.sitio.es/dir2/otrapagina.html`.

Si queremos que el navegador envíe las cookies a todas las páginas del servidor debemos pasar como parámetro la barra ("`/`") al método `setPath()` de la clase `Cookie`.

Lo último que nos quedaría para terminar de configurar nuestra cookie es establecer la versión a la que pertenece mediante el método `setVersion()`.

Una vez que ya tenemos nuestro objeto de la clase `Cookie` completamente configurado con los valores deseados, debemos enviárselo al navegador para que la próxima vez que realice una petición a nuestra aplicación Web, en las cabeceras de petición HTTP incluya la cookie que hemos creado.

Para añadir una cookie a la respuesta que se le envía al cliente se utiliza el método `addCookie()` del interfaz `HttpServletResponse`, pasándole como parámetro el objeto `Cookie` que hemos creado. Este método enviará una cabecera de respuesta HTTP `Set-Cookie` al navegador. En el Código Fuente 43 se muestra un servlet que crea una cookie y luego la añade a la respuesta.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletCreaCookie extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        Cookie cookie=new Cookie("nombre","Angel");
        cookie.setMaxAge(3600);
        cookie.setComment("Cookie de prueba");
        cookie.setSecure(false);
        response.addCookie(cookie);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Creación de una cookie</title></head>"+
            "<body><h1>Servlet que crea una cookie </h1>"+
            "</body></html>");
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 43

En este caso hemos creado una cookie llamada nombre, con el valor Angel y con una duración de una hora (3600 segundos), esta cookie pertenecerá a la versión 0 de la especificación de las cookies. También hemos indicado un comentario acerca de la utilidad o propósito de la cookie, el dominio de la cookie será el tomado por defecto, en este caso localhost, así como la ruta de la cookie, que en este caso será /ejemplos/servlet. Se ha utilizado el método setSecure() para indicar que la cookie no se envía por canal seguro, en realidad esta sentencia se podría omitir, ya que es el comportamiento por defecto.

Si nuestro navegador está configurado para avisarnos antes de recibir una cookie, aparecerá una pantalla informativa como la que ofrece la Figura 26.

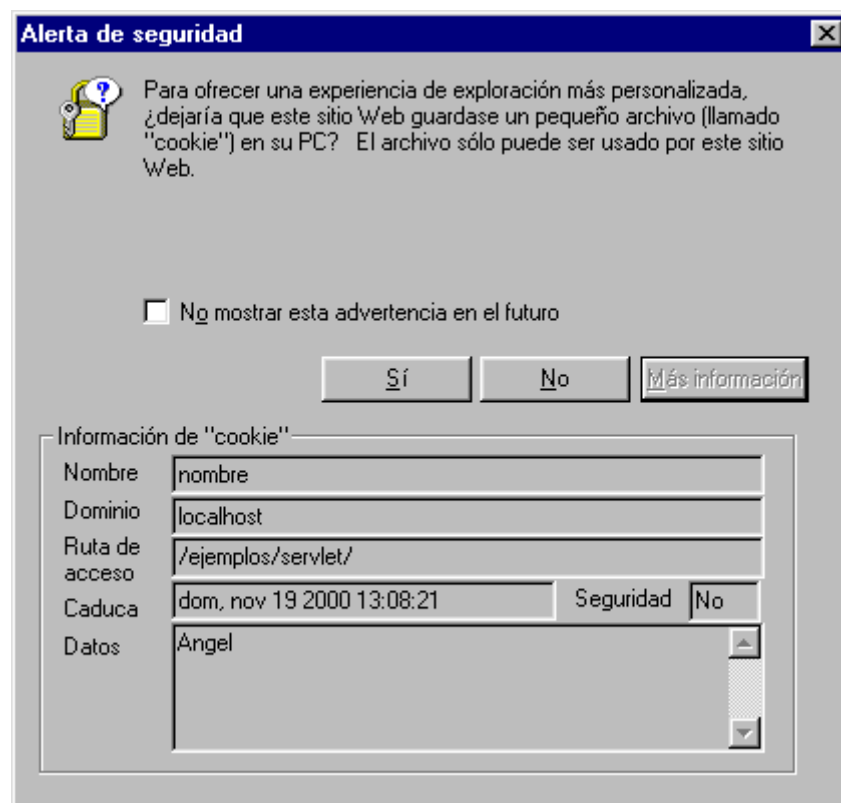


Figura 26. Información que ofrece el navegador acerca de la cookie

Si hemos ejecutado este servlet con el navegador Internet Explorer, al acudir al directorio windows\cookies veremos un fichero de texto con el nombre ANGEL@SERVLET(1).TXT (o similar), este es el fichero que representa físicamente a la cookie, si abrimos este fichero su contenido será similar al mostrado a continuación.

```

nombre
Angel
localhost/ejemplos/servlet/
0
3396489344
29381161
1927127712
29381153
*
```

Si queremos añadir varias cookies, no tenemos nada más que repetir los mismos pasos que en el caso de una. En el Código Fuente 44 se muestra una nueva versión del servlet, que en este caso crea tres cookies distintas. Además de crear la cookie que contiene el nombre, se crean otras dos más, una que contiene la edad y otra el domicilio.

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

public class ServletCreaCookie extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        Cookie cookie=new Cookie("nombre","Angel");
        cookie.setMaxAge(3600);
        cookie.setComment("Cookie que contiene el nombre");
        cookie.setSecure(false);
        response.addCookie(cookie);
        Cookie cookieDos=new Cookie("edad","25");
        cookieDos.setMaxAge(3600);
        cookieDos.setComment("Cookie que contiene la edad");
        response.addCookie(cookieDos);
        Cookie cookieTres=new Cookie("domicilio","Periana 8");
        cookieTres.setComment("Cookie que contiene el domicilio");
        response.addCookie(cookieTres);

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Creación de cookies</title></head>"+
            "<body><h1>Servlet que crea tres cookies </h1>"+
            "</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 44

En este caso si acudimos al directorio windows\cookies y abrimos el fichero de texto representando a las cookies, no aparecen datos relativos a cookie domicilio, ya que la cookie que contiene el domicilio no le hemos indicado duración, y por lo tanto se restringe únicamente a la memoria del navegador, eliminándose al cerrar la sesión del navegador. A continuación se muestra el nuevo contenido del fichero que contiene las cookies.

```
nombre
Angel
localhost/ejemplos/servlet/
0
1071652864
29381166
3734958528
29381157
*
```

```

edad
25
localhost/ejemplos/servlet/
0
1071652864
29381166
3782758528
29381157
*
```

Al cargar el servlet de nuevo sólo han aparecido dos ventanas de advertencia de cookies, ya que la cookie nombre ya la hemos creado previamente y al realizar la llamada al servlet, el navegador la ha enviado. Es decir, si la cookie ya existe, no se vuelve a crear únicamente se modificará su valor, en el caso de que sea distinto.

En el siguiente apartado veremos como recuperar desde nuestros servlets las cookies que nos envía el navegador.

## Utilizando las cookies

En el apartado anterior hemos visto como crear una cookie y enviarla al navegador para que la almacene y la devuelva en sucesivas llamadas a nuestra aplicación Web, por lo tanto además de tener un mecanismo para añadir cookies a la respuesta de los servlets, debemos tener un mecanismo que nos permita recuperar las cookies que nos devuelven los navegadores a través de las peticiones HTTP.

Para recuperar las cookies enviadas por el navegador en al petición HTTP correspondiente el interfaz `HttpServletRequest` ofrece el método `getCookies()`. Este método cuando se lanza sobre el objeto `HttpServletRequest` del servlet devuelve un array de objetos de la clase `javax.servlet.http.Cookie`, que contendrá cada una de las cookies enviadas por el navegador.

En el Código Fuente 45 se muestra un servlet que recupera todas las cookies enviadas por un navegador.

```

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

public class ServletMuestraCookies extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Muestra cookies</title></head>" +
            "<BODY BGCOLOR=\"#FDF5E6\">" +
            "<H1 ALIGN=\"CENTER\">Muestra cookies</H1>" +
            "<TABLE BORDER=1 ALIGN=\"CENTER\">" +
            "<TR BGCOLOR=\"#FFAD00\">" +
            "  <TH>Nombre de la cookie</TH>" +
            "  <TH>valor de la cookie</TH></TR>"
        );
        Cookie[] cookies = request.getCookies();
        Cookie cookie;
        for(int i=0; i<cookies.length; i++) {
            cookie = cookies[i];
            out.println("<TR>" +
```



```

        " <TD>" + cookie.getName() + "</TD>" +
        " <TD>" + cookie.getValue() + "</TD></TR>"
    );
}
out.println("</TABLE></BODY></HTML>");
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

    doGet(request, response);
}
}

```

Código Fuente 45

Este servlet recorre el array de objetos Cookie devuelto por el método `getCookies()` y muestra el nombre y valor de cada uno de los objetos.

Si ejecutamos previamente el servlet del apartado anterior, en el que se creaban tres cookies, el resultado de la ejecución del servlet que muestra las cookies se puede ver en la Figura 27.



Figura 27. Cookies que envía el navegador

Como se puede comprobar, para recuperar una cookie concreta deberemos recorrer el array de objetos de la clase Cookie hasta encontrar la cookie deseada. En el Código Fuente 46 se ofrece una clase, llamada Utilidades, que ofrece un par de métodos que pueden ser muy útiles a la hora de recuperar un objeto Cookie concreto o bien su valor.

```

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

public class Utilidades{

    public static String devuelveValorCookie(Cookie[] cookies, String nombreCookie){
        String valor="";
        if (cookies != null) {

```

```

        for(int i=0; i<cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (nombreCookie.equals(cookie.getName()))
                return(cookie.getValue());
        }
    }
    return(valor);
}

public static Cookie devuelveCookie(Cookie[] cookies, String nombreCookie){
    if (cookies != null) {
        for(int i=0; i<cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (nombreCookie.equals(cookie.getName()))
                return(cookie);
        }
    }
    return(null);
}
}

```

Código Fuente 46

El método `devuelveValorCookie()` recorre el array de objetos `Cookie` que le pasamos por parámetro y localiza la cookie que se corresponde con el nombre que le hemos pasado por parámetro, devolviendo en un objeto `String` el valor que contiene la misma.

El método `devuelveCookie()` es muy similar al anterior, pero en lugar de devolver únicamente el valor de la cookie que coincide con el nombre que le pasamos por parámetro, devuelve el objeto `Cookie` correspondiente, este método es muy interesante cuando necesitamos algo más que el valor de la cookie.

Ambos métodos son estáticos, por lo tanto para utilizarlos no hay que instanciar un objeto de la clase `Utilidades`, sino que los lanzamos directamente sobre la clase.

A continuación se ofrece un servlet que utiliza estos métodos. Este servlet, que se puede observar en el Código Fuente 47, utiliza el método `devuelveValorCookie()` para obtener el valor de una cookie concreta indicada a través de un formulario, y el método `devuelveCookie()` lo utiliza para borrar la cookie cuyo nombre se indica por parámetro en el mismo formulario, el código HTML de la página que contiene este formulario se ofrece en el Código Fuente 48.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ServletAccedeCookies extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        Cookie[] cookies=request.getCookies();
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Servlet que obtiene una cookie"+
            " y borra otra</title></head>"+
            "<body><b>Valor de la cookie "
            +request.getParameter("nombreDevuelve")+": </b>");
        String valor=Utilidades.devuelveValorCookie(
            cookies,request.getParameter("nombreDevuelve"));
        if (valor.equals(""))
            out.println("La cookie indicada no existe");
    }
}

```

```

else
    out.println(valor);
out.println("<br><br>");
Cookie cookie=Utilidades.devuelveCookie(
                                cookies,request.getParameter("nombreBorra"));
if(cookie!=null){
    cookie.setMaxAge(0);
    response.addCookie(cookie);
    out.println("<b>La cookie "+request.getParameter("nombreBorra")+
                " ha sido borrada</b>");
}else
    out.println("<b>La cookie indicada no existe</b>");
out.println("<br><br><a href=\"\"/ejemplos/cookies.html\"\"/>Volver</a>");
out.println("</body></html>");
}
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{
    doGet(request,response);
}
}

```

Código Fuente 47

```

<html>
<body>
<form method="GET" action="servlet/ServletAccedeCookies">
Cookie de la que se quiere obtener su valor:
<input type="text" name="nombreDevuelve" size="20"><br>
Cookie que se quiere borrar:
<input type="text" name="nombreBorra" size="20"><br>
<input type="submit" name="enviar" value="Enviar datos">
</form>
</body>
</html>

```

Código Fuente 48

Para ejecutar esta ejemplo se recomienda ejecutar antes el servlet que vimos en el apartado anterior y que creaba varias cookies. Si indicamos que queremos eliminar la cookie llamada nombre y obtener el valor de la cookie llamada domicilio, obtenemos el resultado de la Figura 28.

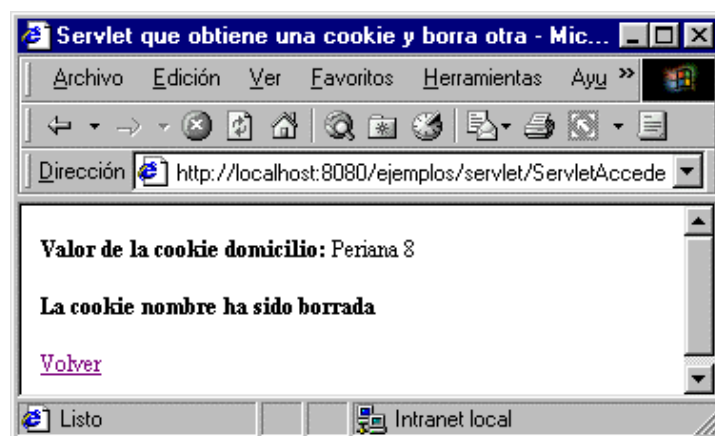


Figura 28. Utilizando los métodos de la clase Utilidades

Vamos a pasar a comentar algunos aspectos del Código Fuente 47. Para eliminar una cookie utilizamos el método `setMaxAge()` pasándole por parámetro el valor cero. Una vez realizada esta modificación sobre la cookie debemos enviarla de nuevo al navegador para que se actualicen los valores de la misma, para ello utilizamos el método `addCookie()` del interfaz `HttpServletResponse`.

Para finalizar este apartado vamos a realizar un servlet que va añadiendo las cookies que se indican a través del formulario que muestra el propio servlet. Además de añadir las cookies a la respuesta, muestra también las cookies actuales de la petición. El código fuente completo de este servlet se puede ver a continuación (Código Fuente 49).

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

public class ServletCookies extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Muestra y crea cookies</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h1>Muestra y crea cookies</h1>");
        //se muestran las cookies existentes en la petición
        out.println("<TABLE BORDER=1> " +
            "<caption>cookies exitentes</caption>" +
            "<TR BGCOLOR=\"#FFAD00\">" +
            " <TH>Nombre de la cookie</TH>" +
            " <TH>valor de la cookie</TH></TR>"
        );
        Cookie[] cookies = request.getCookies();
        Cookie cookie;
        for(int i=0; i<cookies.length; i++) {
            cookie = cookies[i];
            out.println("<TR>" +
                " <TD>" + cookie.getName() + "</TD>" +
                " <TD>" + cookie.getValue() + "</TD></TR>"
            );
        }
        out.println("</TABLE>");
        out.println("<P><b>Creación de una nueva cookie</b><br>");
        //se crea el formulario
        out.print("<form action=\"ServletCookies\" method=GET>");
        out.print("<b>Nombre:</b>");
        out.println("<input type=text length=20 name=nombreCookie><br>");
        out.print("<b>Valor:</b>");
        out.println("<input type=text length=20 name=valorCookie><br>");
        out.println("<input type=submit value=Enviar></form>");
        //se recuperan los valores del formulario
        String nombreCookie = request.getParameter("nombreCookie");
        String valorCookie = request.getParameter("valorCookie");
        //se crea la nueva cookie
        if (nombreCookie != null && valorCookie != null) {
            cookie = new Cookie(nombreCookie, valorCookie);
            response.addCookie(cookie);
            out.println("<P>Se ha añadido la cookie "+nombreCookie);
            out.print(" con el valor "+valorCookie);
        }
        out.println("</body>");
    }
}
```

```
        out.println("</html>");  
    }  
  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws IOException, ServletException{  
  
        doGet(request, response);  
    }  
}
```

Código Fuente 49

Como se puede observar es el propio servlet el encargado de generar el formulario HTML, en los ejemplos anteriores disponíamos de una página HTML que contenía el formulario.

Las cookies creadas por este servlet se perderán al cerrar la sesión del navegador, ya que no hemos utilizado el método `setMaxAge()`. Las cookies que se van creando se podrán ver a la siguiente petición del servlet.

En la Figura 29 se muestra un ejemplos de ejecución de este servlet, que se ha utilizado para la creación de tres cookies.

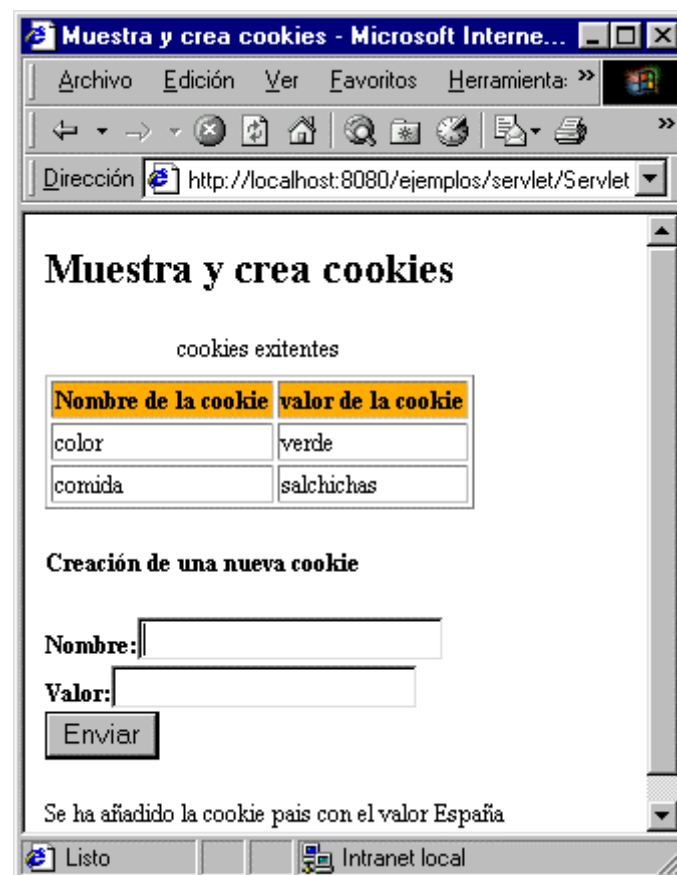


Figura 29. Cookie que muestra y crea cookies

En el siguiente capítulo trataremos el mantenimiento, desde un servlet, de la sesión perteneciente al usuario conectado a nuestra aplicación Web. Esto lo conseguiremos a través del interfaz `javax.servlet.http.HttpSession`.

En el Código Fuente 49 se puede observar que se repite el código visto en un ejemplo anterior, este código muestra las cookies existentes en una petición (objeto `HttpServletResponse`), por lo tanto podría interesarnos incluir el servlet que muestra las cookies dentro del servlet que crea las cookies, esto lo podemos hacer a través de un interfaz llamado `javax.servlet.RequestDispatcher`.

En le siguiente apartado vamos a comentar el interfaz `RequestDispatcher`, que nos permitirá redirigir la petición de un cliente hacia otro servlet o recurso del servidor y también nos permitirá incluir la salida de otro documento (página JSP, servlet, página HTML, etc.).

## El interfaz `RequestDispatcher`

Este interfaz que podemos encontrar en el paquete `javax.servlet` representa un recurso localizado en una ruta determinada, y a través de un objeto `RequestDispatcher` podremos redirigir una petición de un cliente a ese recurso o bien incluir la salida de la ejecución de ese recurso dentro del servlet correspondiente.

Para obtener un objeto `RequestDispatcher` lanzaremos el método `getRequestDispatcher()` sobre el objeto `HttpServletRequest` correspondiente. A este método, perteneciente al interfaz `javax.servlet.ServletRequest`, se le pasa como parámetro un objeto `String` que contiene la ruta del recurso correspondiente, representado mediante su URI.

Como ya hemos dicho, la finalidad del interfaz `RequestDispatcher` es la de ofrecer una especie de envoltorio sobre un recurso del servidor, para permitirnos realizar dos acciones básicas con este recurso desde nuestro servlet. Estas acciones son, por un lado redirigir la petición del cliente al recurso que representa el objeto `RequestDispatcher` correspondiente y por otro lado incluir en el servlet el resultado de la ejecución del recurso del servidor.

El interfaz `RequestDispatcher` ofrece únicamente dos métodos, y cada uno de ellos cubre la funcionalidad comentada en el párrafo anterior. Los métodos del interfaz `RequestDispatcher` son:

- `void forward(ServletRequest request, ServletResponse response):` redirige una petición desde un servlet a otro recurso (servlet, página JSP o página HTML) dentro del servidor.
- `void include(ServletRequest request, ServletResponse response):` incluye el contenido de un recurso (servlet, página JSP o página HTML) en la respuesta.

En un principio se puede pensar que la funcionalidad ofrecida por el método `forward()` del interfaz `RequestDispatcher` es muy similar a la ofrecida por el método `sendRedirect()` del interfaz `HttpServletResponse`, sin embargo son completamente diferentes.

El método `forward()` redirige al recurso que representa el objeto `RequestDispatcher` correspondiente, pero también le pasa la petición inicial al recurso y además una vez ejecutado el método `forward()` se vuelve a pasar el control de la ejecución al servlet original que realizó la redirección. Se puede decir que el método `forward()` es más parecido al método `include()`, ya que ambos incluyen el contenido del recurso que representa el objeto `RequestDispatcher` sobre el que se pueden lanzar ambos métodos.

El método `forward()` generará un error si ya hemos enviado el contenido del búfer al cliente, al igual que sucedía con el método `sendRedirect()` del interfaz `HttpServletResponse`.

El Código Fuente 50 es un servlet que muestra el método `forward()` en acción. Este servlet realiza unas cuantas sentencias `out.println()` y luego comprueba, con el método `isCommitted()` del interfaz `HttpServletResponse`, si ya ha sido enviado algún contenido del cuerpo de la respuesta al cliente. En caso negativo se obtiene el objeto `RequestDispatcher` mediante el método `getRequestDispatcher()` del interfaz `javax.servlet.ServletRequest`. Este objeto va a representar a otro servlet, cuyo código se ofrece en el Código Fuente 51, y que simplemente se ocupa de mostrar el parámetro que le pasan a través de la petición.

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

public class ServletForward extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        out.println("<br>Búfer enviado:"+response.isCommitted()+"<br>");
        if (!response.isCommitted()){
            out.println("<I>No se ha enviado el buffer, por lo tanto
redirecciono</i><br>");
            RequestDispatcher rd=request.getRequestDispatcher("OtroServlet");
            rd.forward(request,response);
            out.println("<br><I>Soy el servlet ServletForward de nuevo</i>");
            out.println("</body></html>");
        }
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 50

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

public class OtroServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Muestra parámetro";
        out.println("<html><BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n");
        out.println("<br><b>Soy OtroServlet y muestro el parámetro de la
petición</b><br>");
        out.println("parametro:" +request.getParameter("parametro"));
    }
}
```

```
        out.println("</BODY></HTML>");  
    }  
  
    public void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        doGet(request, response);  
    }  
}
```

Código Fuente 51

En la Figura 30 se puede ver un ejemplo de ejecución del servlet que realiza la redirección, como se puede comprobar, la salida que contempla el usuario se encuentra compuesta por el servlet ServletForward y el servlet OtroServlet.

Para ejecutar el servlet podemos pasarle una cadena consulta indicando un parámetro llamado parametro, con un valor cualquiera. Este parámetro lo recuperará el servlet al que se redirige la petición, ya que este servlet tendrá acceso completo a la petición original.



Figura 30. Utilizando el método forward()

Si se hubiera ejecutado el método sendRedirect() sobre el objeto HttpServletResponse, el resultado hubiera sido muy diferente, como se puede corroborar en la Figura 31.

Si en el código anterior hubiéramos utilizado el método include(), el resultado hubiera sido idéntico. La única diferencia entre los métodos forward() e include() es que siempre podemos lanzar el método include(), sin embargo si ya hemos enviado algún contenido del búfer al cliente el método forward() producirá un error.

El objeto RequestDispatcher se puede obtener también a través del método getRequestDispatcher() del interfaz javax.servlet.ServletContext. La única diferencia con el método getRequestDispatcher() del interfaz ServletRequest es que el método del interfaz ServletContext sólo admite rutas absolutas como parámetro, mientras que en el interfaz ServletRequest admite también rutas relativas.





Figura 31. Utilizando el método sendRedirect()

Así si deseamos rescribir el servlet del ejemplo anterior, llamado ServletForward, para utilizar un objeto ServletContext que nos ofrezca un objeto RequestDispatcher, obtendríamos el Código Fuente 52. El resultado de la ejecución de este servlet es idéntico a la versión anterior.

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

public class ServletForward extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        out.println("<br>Búfer enviado:"+response.isCommitted()+"<br>");
        if (!response.isCommitted()){
            out.println("<i>No se ha enviado el buffer, por lo tanto
redirecciono</i><br>");
            ServletContext context=getServletContext();
            RequestDispatcher rd=context.getRequestDispatcher
                ("/ejemplos/servlet/OtroServlet");
            rd.forward(request,response);
            out.println("<br><i>Soy el servlet ServletForward de nuevo</i>");
            out.println("</body></html>");
        }
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 52

Como se puede comprobar para obtener un objeto `ServletContext` debemos utilizar el método `getServletContext()`, que heredamos de la clase `GenericServlet`, a través de la clase `HttpServlet`.

En este momento aprovecho para realizar una advertencia a todos los lectores. En diversas ocasiones hemos ido realizando distintas modificaciones sobre un mismo servlet, ya sea para realizar algunas comprobaciones o para obtener nuevos resultados. Para actualizar el servlet simplemente debemos copiarlo al directorio que contiene los servlets de nuestra aplicación Web, es decir, el directorio `CLASSES`, y el contenedor Jakarta Tomcat recargará de forma automática la nueva versión del servlet.

Ahora bien, este mecanismo de recarga y actualización automática de los servlets ofrecido por Tomcat, hay en algunos casos en los que falla, sobretodo cuando hemos recargado y modificado un servlet varias veces. En este caso es mejor parar el servidor Tomcat y volver a iniciarlo.

Realizado este paréntesis, volvemos a retomar la explicación del interfaz `RequestDispatcher`, centrándonos ahora en el método `include()`.

Como ya hemos comentado el método `include()` incluye en la salida del servlet en el que se invoca, la ejecución del recurso que representa. Puede ser un contenido estático, como una página HTML, o un contenido dinámico como un servlet o una página JSP. Este método se puede lanzar en cualquier momento sin tener en cuenta el estado del búfer.

En el Código Fuente 53 se ofrece un servlet que utiliza el método `include()` para incluir en la respuesta ofrecida al usuario la salida de un servlet que muestra la fecha y hora actuales.

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

public class ServletInclude extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Página de respuesta</title></head>"+
            "<body><b>Tamaño del búfer: </b>"+
            "<i>"+response.getBufferSize()/1024+" KB</i>"
        );
        response.flushBuffer();
        out.println("<br>Búfer enviado:"+response.isCommitted()+"<br>");
        out.println("<br><i>Se incluye el servlet que muestra la fecha
actual</i><br>");
        RequestDispatcher rd=request.getRequestDispatcher("ServletFecha");
        rd.include(request,response);
        out.println("<br><i>Soy el servlet ServletInclude de nuevo</i>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 53

Como se puede apreciar en el código fuente anterior, se fuerza el envío del búfer al usuario utilizando el método `flushBuffer()` sobre el objeto `HttpServletResponse`, para demostrar la utilidad que ofrece el método `include()`.

Un ejemplo de la ejecución de este servlet se ofrece en la Figura 32.

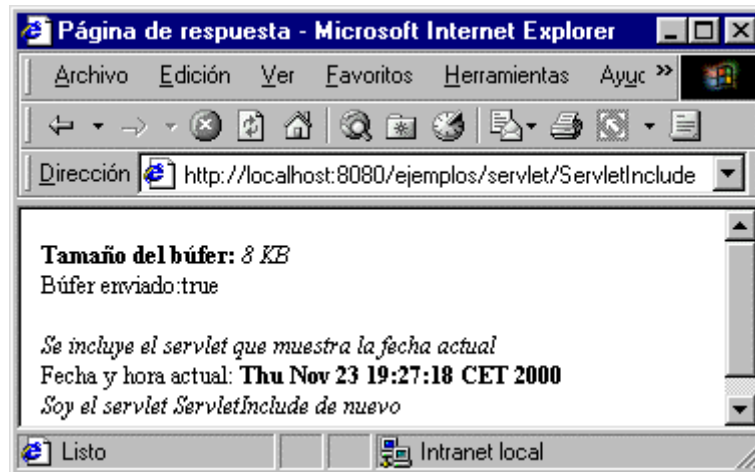


Figura 32. utilizando el método `include()`

El código fuente del servlet que muestra la fecha, es muy sencillo y es el que se ofrece a continuación(Código Fuente 54).

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

public class ServletFecha extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Date fecha=new Date();
        out.println("<html><body>");
        out.println("Fecha y hora actual: <b>"+fecha.toString()+"</b>");
        out.println("<body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 54

El servlet `ServletFecha` necesita importar el paquete `java.util` para poder hacer uso de la clase `Date`.

A continuación se ofrece un servlet (Código Fuente 55) que muestra un formulario para permitirnos indicar el recurso que deseamos incluir en la respuesta del mismo. Al pulsar el botón de envío el servlet recupera el campo del formulario y obtiene el objeto `RequestDispatcher` que se va a corresponder con el recurso indicado, a continuación realiza la inclusión del recurso.

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

public class ServletIncluyeRecursos extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Incluye un recurso en el servlet</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.print("<form action=\"ServletIncluyeRecursos\" method=GET>");
        out.print("<b>Recurso para incluir:</b>");
        out.println("<input type=text length=20 name=recurso><br>");
        out.println("<input type=submit value=Enviar></form>");
        String recurso = response.encodeURL(request.getParameter("recurso"));
        if (recurso!=null){
            out.println("<TABLE BORDER='1' ALIGN='CENTER'><TR><TD>");
            try{
                RequestDispatcher rd=request.getRequestDispatcher(recurso);
                out.println("Se incluye el recurso "+recurso+"<br><br><br>");
                rd.include(request,response);
            }catch(Exception ex){
                out.println("<h1>Se ha producido un error:</h1>" + ex);
            }
            out.println("</TD></TR></TABLE>");
        }
        out.println("</body>");
        out.println("</html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request, response);
    }
}
```

Código Fuente 55

Como se puede comprobar se ha utilizado el método `encodeURL()` del interfaz `HttpServletRequest` para realizar la codificación URL de forma correcta del parámetro del formulario enviado. Podemos indicar tanto rutas relativas como absolutas. En la Figura 33 se puede ver el resultado de incluir el servlet `ServletFecha`, utilizando una ruta absoluta.

Para finalizar este apartado y este capítulo, vamos a aplicar la utilización del método `include()` a un ejemplo visto en el apartado anterior. Este ejemplo es un servlet que muestra las cookies existentes en la petición y también permite crearlas. Ya dimos solución a este servlet a través del Código Fuente 49,

pero podemos utilizar el método `include()` del interfaz `RequestDispatcher`, para utilizar el servlet que mostraba las cookies existentes en la petición, este servlet se puede ver en el Código Fuente 45.

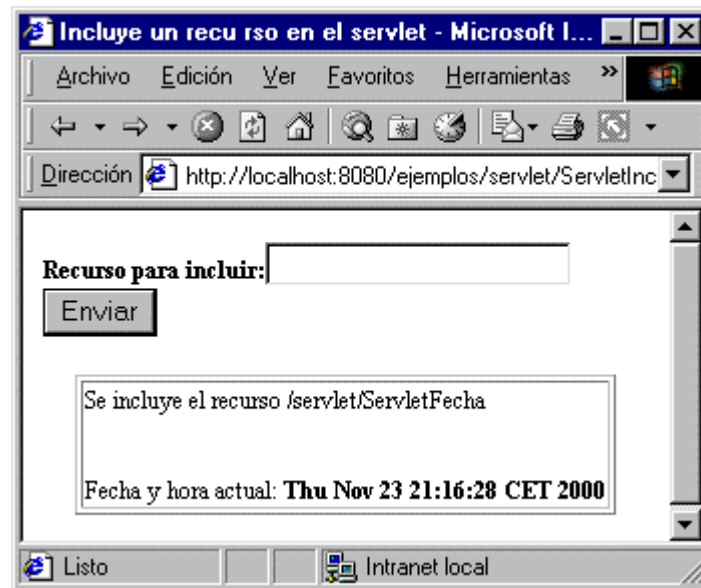


Figura 33. Insertando el recurso indicado en el formulario

La nueva versión del servlet sería la mostrada en el Código Fuente 56, como se puede ver se elimina todo el código encargado de mostrar las cookies de la petición, y en su lugar se hace una llamada al método `include()` sobre el objeto `RequestDispatcher` que representa al servlet `ServletMuestraCookies`.

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

public class ServletCookiesDispatcher extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Muestra y crea cookies</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h1>Muestra y crea cookies</h1>");

        RequestDispatcher rd=request.getRequestDispatcher("ServletMuestraCookies");
        rd.include(request,response);

        out.println("<P><b>Creación de una nueva cookie</b><br>");
        //se crea el formulario
        out.print("<form action=\"ServletCookiesDispatcher\" method=GET>");
        out.print("<b>Nombre:</b>");
        out.println("<input type=text length=20 name=nombreCookie><br>");
        out.print("<b>Valor:</b>");
        out.println("<input type=text length=20 name=valorCookie><br>");
        out.println("<input type=submit value=Enviar></form>");
        //se recuperan los valores del formulario
    }
}
```

```
String nombreCookie = request.getParameter("nombreCookie");
String valorCookie = request.getParameter("valorCookie");
//se crea la nueva cookie
if (nombreCookie != null && valorCookie != null) {
    Cookie cookie = new Cookie(nombreCookie, valorCookie);
    response.addCookie(cookie);
    out.println("<P>Se ha añadido la cookie "+nombreCookie);
    out.print(" con el valor "+valorCookie);
}
out.println("</body>");
out.println("</html>");
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{

    doGet(request, response);
}
}
```

Código Fuente 56

En el siguiente capítulo vamos a tratar las sesiones dentro de los servlets, a través del interfaz `javax.servlet.http.HttpSession`, y también trataremos otro tipo de mantenimiento de estado, que se realiza a través del interfaz `javax.servlet.ServletContext`, aunque ya hemos adelantado un posible uso de este interfaz en este mismo apartado.

# Servlets: el interfaz HttpSession y ServletContext

---

## Introducción

Este capítulo se compone de dos partes, por un lado trataremos el interfaz `javax.servlet.http.HttpSession`, que nos permite mantener a través de nuestros servlets información a lo largo de una sesión del cliente con nuestra aplicación Web. Cada cliente tendrá su propia sesión, y dentro de la sesión podremos almacenar cualquier tipo de objeto.

También veremos los distintos mecanismos que existen para simular el mantenimiento de sesiones, y veremos los mecanismos sobre los que se apoya el interfaz `HttpSession`.

Se debe recordar que el protocolo HTTP es un protocolo sin estado, es decir, no se puede almacenar información entre diferentes conexiones HTTP. No se puede mantener el estado entre diferentes páginas Web (o servlets) a través del protocolo HTTP, sino que se deben utilizar otros mecanismos como las cookies. Pero el interfaz `HttpSession` junto con el interfaz `ServletContext` nos permite de forma sencilla y directa almacenar información abstrayéndonos del uso de cookies y de encabezados HTTP.

La información almacenada en objetos `HttpSession` y `ServletContext` difieren en el ámbito de la misma, una información tendrá el ámbito de la sesión particular de un usuario y la otra el ámbito más general de la aplicación Web, respectivamente.

Los objetos o atributos almacenados dentro del objeto `ServletContext` son visibles para todos los usuarios que están utilizando la misma aplicación Web, es decir son compartidas por varios usuarios.

En contraposición al objeto `HttpSession`, cuyos atributos son para cada uno de los usuarios conectados, es decir, no se comparten y son propios de cada sesión.

El mantenimiento de una información a nivel de sesión es muy interesante a la hora de realizar aplicaciones Web, ya que tenemos un mayor control sobre el estado en el que se encuentra el cliente en cada momento, no debemos olvidar que las aplicaciones Web son muy particulares, y no ofrecen el control que otro tipo de aplicaciones tradicionales poseen.

En la segunda parte de este capítulo trataremos el interfaz `javax.servlet.ServletContext` que nos permite comunicarlos con el contenedor de servlets, y que permite mantener un estado más general para que se comparta entre todos los usuarios de la aplicación Web. Podemos decir un objeto de este interfaz representa a la aplicación Web.

## Soluciones tradicionales

Existen tres soluciones o mecanismos tradicionales para mantener el estado entre diferentes peticiones HTTP, estas soluciones son las cookies (ya vistas y comentadas de forma extensa en el capítulo anterior), los campos ocultos de los formularios y el mecanismo de reescritura de URLs (URL-Rewriting).

Las cookies permiten almacenar información del usuario e identificar al usuario de una forma única, y en cada conexión se puede obtener la cookie correspondiente, que contendrá el identificador de la sesión actual que pertenece al usuario conectado. Pero el manejo de estas cookies a lo largo de distintos servlets puede ser algo realmente engorroso y complicado, por lo que es preferible utilizar un API de más alto nivel que nos abstraiga de estos procesos. Este API lo ofrece el interfaz `javax.servlet.http.HttpSession`.

Otra solución alternativa es el uso de los campos ocultos de un formulario HTML. Estos campos son los que se definen de tipo `HIDDEN` dentro de una etiqueta `<INPUT>` de un formulario. Estos campos serán enviados cuando se envíe el formulario, ya sea con el método `POST` o con el método `GET`. Los campos ocultos pueden contener información que identifique de forma única al usuario conectado actualmente, pero el problema lo encontramos a la hora de ir generando las distintas páginas que contienen estos campos ocultos, ya que se deben generar siempre de forma dinámica.

A pesar de ser campos ocultos y no aparecer en la página HTML, el usuario puede acceder a ver su contenido si indica al navegador que desea ver el código fuente de la página. Este mecanismo resulta bastante artesanal y poco seguro.

El último mecanismo para permitir mantener información entre distintas peticiones es el mecanismo de reescritura de URLs (URL-Rewriting). En esta aproximación el cliente añade información extra al final de cada URL, esta información identificará la sesión propia del usuario conectado a la aplicación Web. Por ejemplo, en la URL `http://localhost:8080/ ejemplos/pagina.html;jsessionid=123`, la información relativa a la sesión del usuario es `jsessionid=123`.

El mecanismo URL-Rewriting se puede utilizar como una solución alternativa a las cookies cuando los navegadores no soportan cookies o es encuentran desactivadas. Sin embargo sigue siendo bastante costoso la generación de estas URLs y la obtención de sus valores, al igual que sucedía con las cookies se hace patente la necesidad de un API de más alto nivel.

La solución que vamos a utilizar en los servlets para poder mantener la información entre las distintas conexiones que realiza un usuario en una sesión con nuestra aplicación Web, es el API de alto nivel representado por el interfaz `javax.servlet.HttpSession`. Este API de alto nivel se basa en los mecanismos de cookies y URL-Rewriting, según la situación. Se utilizarán las cookies si el navegador



las soporta, sino de forma automática se pasará a utilizar el mecanismo de URL-Rewriting, pero afortunadamente, el desarrollador de servlets no tiene que preocuparse por ninguno de los detalles relacionados con los mecanismos utilizados, como puede ser la recuperación de las cookies y la información que se debe añadir a las URLs, sino que es completamente transparente para el desarrollador, que tratará con un objeto *HttpSession* para utilizar y acceder a la sesión de cada usuario.

En el siguiente apartado se define y se muestra brevemente los métodos que ofrece el interfaz *HttpSession* para realizar el seguimiento y mantenimiento de sesiones.

## El interfaz *HttpSession*

Este interfaz define una forma de identificar a un usuario a través de las distintas peticiones a distintos recursos dentro de una misma aplicación Web. Cada aplicación Web, incluso en el mismo servidor, poseerá una sesión distinta.

El contenedor de servlets, Jakarta Tomcat en este caso, utiliza este interfaz para crear una sesión entre el cliente HTTP (navegador) y el servidor HTTP (servidor Web). La sesión se mantiene durante un periodo de tiempo especificado entre varias peticiones de un usuario. A cada usuario dentro de una aplicación Web le corresponderá una sesión distinta, siendo imposible intercambiar información entre sesiones, ya que con tienen información específica de cada usuario.

Es importante aclarar que las sesiones de un mismo usuario en distintas aplicaciones Web (contextos definidos en el fichero *SERVER.XML* del servidor Tomcat) no son visibles entre sí. Para el servidor se trata de dos usuarios distintos, cada uno pertenece a una aplicación distinta.

El servidor puede mantener el estado de la sesión utilizando el mecanismo de cookies o de reescritura de URLs, dependiendo de si el navegador cliente permite la utilización de cookies o no.

En una sesión de un usuario podemos almacenar cualquier tipo de objeto, a los distintos objetos que se almacenan en la sesión de un usuario se les denomina atributos de la sesión.

A continuación se muestran los distintos métodos que ofrece el interfaz *HttpSession*.

- `Object getAttribute(String nombreAtributo):` devuelve el objeto almacenado en la sesión actual, y cuya referencia se corresponde con el nombre de atributo indicado por parámetro como un objeto de la clase `String`. Sea cual sea la clase del objeto almacenado en la sesión, este método siempre devolverá un objeto de la clase genérica `Object`, a la hora de recuperarlo deberemos realizar la transformación de clases correspondiente. Este método devuelve `null` si el objeto indicado no existe.
- `Enumeration getAttributeNames():` este método devuelve en un objeto `Enumeration` del paquete `java.util`, que contiene los nombres de todos los objetos almacenados en la sesión actual.
- `long getCreationTime():` devuelve la fecha y hora en la que fue creada la sesión, medido en milisegundos desde el 1 de enero de 1970.
- `String getId():` devuelve una cadena que se corresponde con el identificador único asignado a la sesión. Luego veremos que este valor se corresponde con el valor de la cookie `JSESSIONID` utilizada para poder realizar el mantenimiento de la sesión de un usuario determinado, y en el caso de no utilizar cookies se corresponde con la información que se añade al final de cada enlace cuando se utiliza el mecanismo de reescritura de URLs.

- `long getLastAccessedTime()`: devuelve en milisegundos la fecha y hora de la última vez que el cliente realizó una petición asociada con la sesión actual.
- `int getMaxInactiveInterval()`: devuelve el máximo intervalo de tiempo, en segundos, en el que una sesión permanece activa entre dos peticiones distintas de un mismo cliente, es decir, es el tiempo de espera máximo en el que pertenece activa una sesión sin que el cliente realice ninguna petición relacionada con la sesión actual. El valor por defecto que puede permanecer inactiva una sesión es de 30 segundos. Una vez transcurrido este tiempo el contenedor de servlets (servlet container) destruirá la sesión, liberando de la memoria todos los objetos que contiene la sesión que ha caducado.
- `void invalidate()`: este método destruye la sesión de forma explícita, y libera de memoria todos los objetos (atributos) que contiene la sesión.
- `boolean isNew()`: devuelve verdadero si la sesión se acaba de crear en la petición actual o el cliente todavía no ha aceptado la sesión (puede rechazar la cookie de inicio de sesión). Este método devolverá falso si la petición que ha realizado el cliente ya pertenece a la sesión actual, es decir, la sesión ya ha sido creada previamente,
- `void removeAttribute(String nombreAtributo)`: elimina el objeto almacenado en la sesión cuyo nombre se pasa por parámetro. Si el nombre del objeto indicado no se corresponde con ninguno de los almacenados en la sesión, este método no realizará ninguna acción.
- `void setAttribute(String nombre, Object valor)`: almacena un objeto en la sesión utilizando como referencia el nombre indicado como parámetro a través de un objeto `String`.
- `void setMaxInactiveInterval(int intervalo)`: establece, en segundos, el máximo tiempo que una sesión puede permanecer inactiva antes de ser destruida por el contenedor de servlets.

Por defecto los servlets no utilizan sesiones ni pertenecen a una sesión concreta. Si queremos que un servlet cree una sesión que pueda ser utilizada por todos los servlets que se encuentran en la misma aplicación Web debemos hacerlo a través del código fuente de nuestro servlet. En el siguiente apartado veremos como crear una sesión desde un servlet.

## Creación de sesiones

Utilizar sesiones en nuestros servlets es bastante sencillo, ya que el interfaz `HttpServletRequest` nos ofrece un método específico para la creación de una sesión, creándola cuando sea necesaria, es decir, cuando todavía no ha sido creada.

Para comprobar si existe un objeto `HttpSession` que puede utilizar el servlet actual para almacenar información en la sesión del usuario actual, el servlet lanzará sobre el objeto `HttpServletRequest` el método `getSession()`. Una llamada al método `getSession()` provocará que el contenedor de servlets compruebe si existe una cookie, llamada `JSESSIONID`, que represente el identificador de la sesión actual, también comprobará si existe una reescritura de URLs en el caso de que el cliente no acepte cookies.

Si el contenedor de servlets no obtiene una cookie ni una información de la URL que le indique el identificador de la sesión actual, significará que la sesión todavía no ha sido creada. Si al método

getSession() le indicamos el parámetro con el valor true, creará la sesión si no existe, y si ya existe la sesión, este método devolverá el objeto HttpSession que representará a la sesión actual del cliente.

Si al método getSession() le indicamos el valor false, y la sesión no existe, nos devolverá un valor nulo (null), pero si la sesión si existe devolverá el objeto HttpSession correspondiente.

Por lo tanto la forma más sencilla que nos asegurará que desde un servlet se creará una sesión si no existe, o que en caso contrario se utilizará la sesión existente, es lanzar dentro de nuestro servlet el método getSession() pasándole por parámetro el valor true sobre el objeto HttpServletRequest correspondiente.

En el Código Fuente 57 se muestra un servlet que crea una sesión si no existe previamente, y acto seguido comprueba si la sesión es nueva o bien se ha recuperado una sesión ya existente.

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

public class ServletSession extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        HttpSession session=request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Sesiones</title></head>"+
            "<body><h1>Servlet que crea una sesión</h1>");
        if (session.isNew())
            out.println("<b>La sesión es nueva.</b>");
        else
            out.println("<b>La sesión había sido creada previamente.</b>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 57

Si ejecutamos este servlet utilizando un navegador configurado para avisarnos de la creación de cookies, se mostrará la pantalla de información de la Figura 34, que indica que se va a crear la cookie especial de inicio de sesión llamada JSESSIONID.

Y la ejecución de este servlet, mostrará el siguiente aspecto en el navegador (Figura 35)

Si decidimos rechazar la cookie, no se creará la sesión. Pero tenemos la alternativa de utilizar el método de reescritura de URLs (URL-rewriting), para ello utilizaremos el método encodeURL() del interfaz HttpServletResponse.

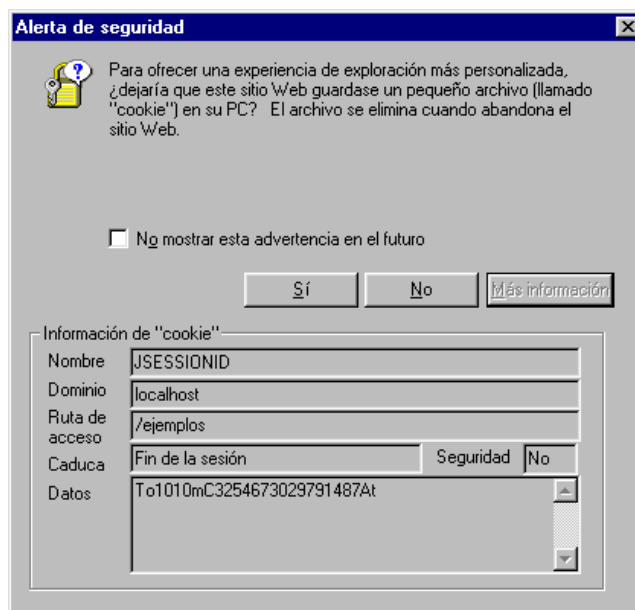


Figura 34. Cookie de inicio de sesión



Figura 35. El servlet ha creado la sesión

Este método añadirá al final de una URL un parámetro que contendrá el identificador de la sesión, pero si se están utilizando las cookies, se escribirá la URL tal cual. La utilización de este mecanismo es bastante tedioso, ya que debemos llevar a través del código fuente todo el control del mantenimiento de la sesión y el almacenamiento de los objetos en la misma, sin poder utilizar un objeto HttpSession.

Ya hemos visto lo sencillo que resulta desde un servlet crear una sesión en el caso de que no exista y recuperar una sesión existente. En el siguiente apartado veremos como almacenar y recuperar objetos de la sesión, así como mostrar toda la información referente a la sesión actual.

## Almacenando y recuperando objetos de la sesión

Si queremos recuperar una sesión creada en un servlet distinto del actual, procederemos de la misma manera que cuando deseamos crear la sesión, es decir, lanzando sobre el objeto HttpServletRequest el

método getSession(). Nos puede interesar recuperar una sesión en otro servlet para recuperar algún objeto almacenado previamente en el objeto HttpSession.

Para almacenar objetos en la sesión actual utilizaremos el método setAttribute() del interfaz HttpSession, a este método le debemos pasar por parámetro el nombre con el que queremos referirnos al objeto para recuperarlo más tarde, y también instancia del objeto correspondiente.

En una sesión podremos almacenar cualquier tipo de objeto, ya que el parámetro que especificamos en el método setAttribute(), que se corresponde con la instancia del objeto que vamos a almacenar, pertenece a la clase Object, que es la clase raíz de la jerarquía de clases que presenta el lenguaje Java.

En el siguiente servlet (Código Fuente 58) se almacenan tres objetos en la sesión, en este caso se trata de almacenar un objeto de la clase String, otro de la clase Integer y otro de la clase Date. Como se puede ver el mecanismo para obtener la sesión es el mismo que el visto en el apartado anterior, si la sesión no existe se crea y se obtiene en un objeto HttpSession, y si la sesión ya existía, el objeto HttpSession devuelto por el método getSession() contendrá la sesión actual.

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

public class ServletSession extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        HttpSession session=request.getSession(true);
        session.setAttribute("nombre",new String("Angel"));
        session.setAttribute("edad",new Integer(25));
        session.setAttribute("fecha", new Date());
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
                    "<title>Sesiones</title></head>"+
                    "<body><h1>Servlet que crea una sesión "+
                    "y almacena en ella tres objetos</h1></body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 58

Para recuperar los valores objetos almacenados en una sesión, utilizaremos el método getAttribute(), al que le pasamos por parámetro el nombre que hace referencia al objeto almacenado que queremos obtener. El siguiente servlet, Código Fuente 59 , obtiene una referencia a la sesión actual y muestra los valores de los distintos atributos (objetos) almacenados en la sesión.

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

```

public class ServletSesionDos extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        HttpSession session=request.getSession(true);
        if(session.isNew())
            response.sendRedirect("/ejemplos/servlet/ServletSesion");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Sesiones</title></head>"+
            "<body><h1>Servlet que utiliza una sesión</h1>");
        out.println("Nombre: <b>"+session.getAttribute("nombre")+"</b><br>");
        out.println("Edad: <b>"+session.getAttribute("edad")+"</b><br>");
        out.println("Fecha: <b>"+session.getAttribute("fecha")+"</b><br>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}

```

Código Fuente 59

Al recuperar un objeto de una sesión mediante el método `getAttribute()`, siempre se obtiene un objeto de la clase `Object`, al que le deberemos aplicar la transformación adecuada (casting) según la operación que deseemos realizar, en el ejemplo anterior se han convertido todos los objetos a objetos de la clase `String` utilizando el operador de concatenación (+).

En este servlet se comprueba que la sesión exista previamente, utilizando el método `isNew()` del interfaz `HttpSession`, en caso contrario se redirecciona al cliente al servlet que crea la sesión y almacena los distintos objetos en la misma.

Si ejecutamos previamente el servlet `ServletSesion`, y a continuación el servlet `ServletSesionDos`, obtenemos el resultado de la Figura 36.

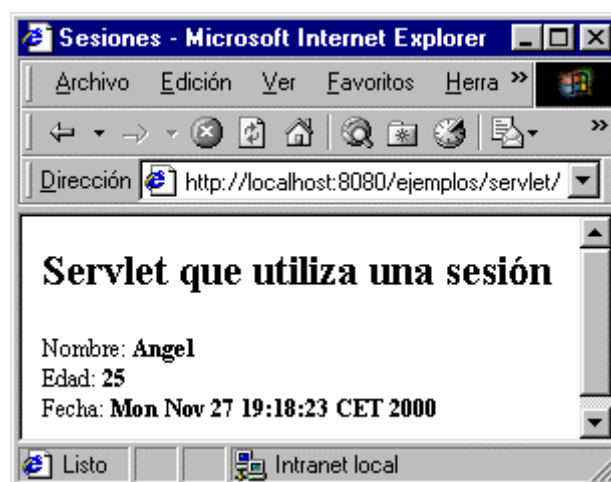


Figura 36. Servlet que muestra objetos de la sesión

Para recuperar todos los objetos de una sesión podemos hacer uso también del método `getAttributeNames()` del interfaz `HttpSession`. Si reescribimos el servlet `ServletSesionDos` según el Código Fuente 60, también obtendremos todos los objetos almacenados en la sesión, para ello recorreremos el objeto `Enumeration` que contiene el nombre de todos los objetos que contiene la sesión y que ha sido devuelto por el método `getAttributeNames()`. Cada nombre de atributo de la sesión se utiliza en la llamada a cada método `getAttribute()`, que nos devolverá el objeto correspondiente.

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

public class ServletSesionDos extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        HttpSession session=request.getSession(true);
        if(session.isNew())
            response.sendRedirect("/ejemplos/servlet/ServletSesion");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Sesiones</title></head>"+
            "<body><h1>Servlet que utiliza una sesión</h1>");
        out.println("<table border='1' align='center'>");
        out.println("<tr><th>Atributo</th><th>Valor</th></tr>");
        Enumeration atributos=session.getAttributeNames();
        while (atributos.hasMoreElements()){
            String nombreAtributo=(String)atributos.nextElement();
            out.println("<tr><td>"+nombreAtributo+"</td>");
            out.println("<td>"+session.getAttribute(nombreAtributo)+"</td></tr>");
        }
        out.println("</table></body></html>");
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 60

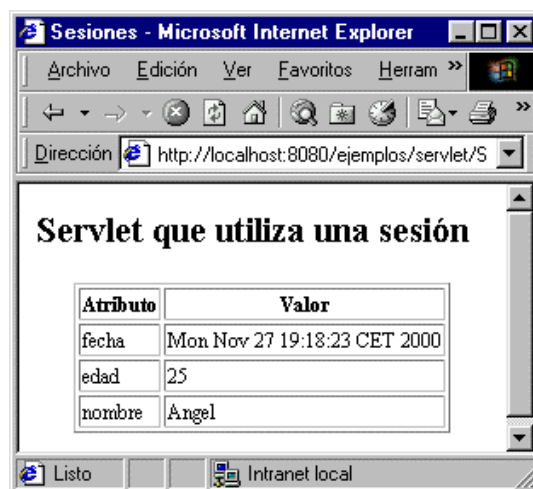


Figura 37. Servlet que muestra todos los objetos contenidos en el sesión

A continuación vamos a mostrar un servlet (Código Fuente 61) que muestra las características de la sesión actual, como puede ser si la sesión es nueva, su identificador, la última vez que el cliente accedió a la misma, la caducidad que tiene asignada la sesión o su fecha de creación.

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

public class ServletPropiedadesSesion extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        HttpSession session=request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Objeto HttpSession</title></head><body>");
        if (session.isNew())
            out.println("La sesión es nueva<br>");
        else
            out.println("La sesión ya se habia creado");
        out.println("La sesión fue creada: "+
            new Date(session.getCreationTime())+"<br>");
        out.println("La sesión tiene el identificador: "+session.getId()+"<br>");
        out.println("Se accedió la sesión por última vez: "+
            new Date(session.getLastAccessedTime())+"<br>");
        out.println("Una sesión puede permanecer inactiva: "+
            session.getMaxInactiveInterval()+" segundos<br>");
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 61

Un ejemplo de la ejecución de este servlet se puede ver en la Figura 38.

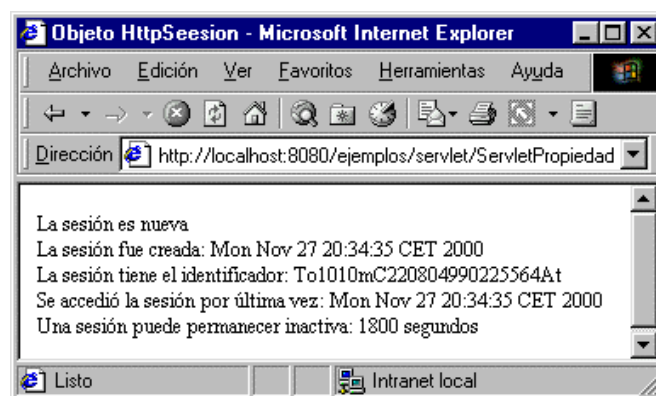


Figura 38. Características de una sesión



El servlet mostrado en el Código Fuente 62, es otro ejemplo de utilización del objeto HttpSession, en este caso la sesión se utiliza para realizar la cuenta del número de accesos que ha realizado el usuario de la sesión correspondiente.

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

public class ServletAccesos extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        HttpSession session=request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Objeto HttpSession</title></head><body>");
        if (session.isNew())
            session.setAttribute("accesos",new Integer(1));
        else{
            Integer valorActual= (Integer)session.getAttribute("accesos");
            session.setAttribute("accesos",new Integer(valorActual.intValue()+1));
        }
        out.println("A este servlet se han accedido: <b>"+
            session.getAttribute("accesos")+"</b> veces");
        out.println("</body></html>");

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 62

Como se puede observar en el código fuente anterior, se ha realizado una transformación entre clases, en este caso se ha realizado un casting entre la clase Object que devuelve el método getAttribute() y la clase Integer que necesitamos para manipular el objeto almacenado en la sesión. En la Figura 39 se muestra un ejemplo de ejecución de este servlet, en este caso se ha cargado el servlet cuatro veces.

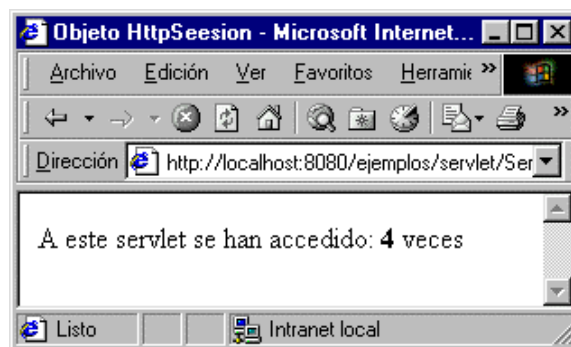


Figura 39. Servlet que cuenta los accesos por sesión

Cada vez que se almacena un nuevo objeto en una sesión utilizando el método `setAttribute()`, si el objeto que se está almacenado implementa el interfaz `javax.servlet.http.HttpSessionBindingListener`, se ejecutará el método `valueBound()` de este interfaz. El método `setAttribute()` lanzará un evento de la clase `javax.servlet.http.HttpSessionBindingEvent`, para indicar al objeto oyente de estos eventos que ha sido almacenado en una sesión.

Cuando se elimina un objeto de la sesión también se lanza el evento `HttpSessionBindingEvent`, pero en este caso se ejecuta el método `valueUnbound()` del interfaz `HttpSessionBindingListener`. Un objeto se eliminará de la sesión de forma directa cuando lo indiquemos mediante el método `removeAttribute()`, y también cuando la sesión se destruya una vez que haya pasado el intervalo de inactividad máximo de la sesión. La destrucción de una sesión se puede realizar lanzando sobre el objeto `HttpSession` correspondiente el método `invalidate()`.

Los eventos que lanzan los objetos `HttpSession` y la forma de recuperarlos en objetos oyentes lo veremos con detenimiento en el capítulo dedicado al modelo de componentes JavaBeans.

En el siguiente apartado pasamos a la siguiente parte del presente capítulo, se trata de comentar el interfaz `ServletContext`, que entre otras funcionalidades, que ya hemos visto en capítulos anteriores, nos ofrece la posibilidad de mantener un estado más general que el de la sesión, y que se corresponde con el estado de la aplicación Web, permitiendo almacenar objetos que pueden ser utilizados por todos los usuarios de la sesión.

## El interfaz `ServletContext`

En capítulos anteriores ya hemos tratado este interfaz que nos permitía comunicarnos con el contenedor de servlets. Hemos visto varias facetas de este interfaz, por un lado hemos utilizado algún método para obtener información acerca del servidor y de contenedor de servlets y por otro lo hemos utilizado para obtener una referencia a un objeto `RequestDispatcher` que representa un recurso en el servidor Web que podremos utilizar para incluir en un servlet.

Otra función del interfaz `javax.servlet.ServletContext`, es la de representar a un contexto (context) existente en el contenedor de servlets. En capítulos anteriores vimos que el fichero de configuración del servidor Yakarta Tomcat, llamado `SERVER.XML`, permitía definir aplicaciones Web o contextos mediante la etiqueta `<Context>`, estos contextos o aplicaciones Web van a estar representados mediante un objeto `ServletContext`.

Una referencia al objeto `ServletContext` que va a representar a la aplicación actual la obtenemos utilizando el método `getServletContext()` del interfaz `javax.servlet.Servlet`, por lo tanto este método lo lanzaremos sobre el propio servlet.

El interfaz `ServletContext` ofrece un estado más general que el interfaz `HttpSession`, un objeto `HttpSession` va a representar la sesión de cada usuario, y un objeto `ServletContext` representará la aplicación en la que se encuentran todos los usuarios, cada usuario tiene su sesión, pero todos los usuarios comparten una misma aplicación.

En un objeto `ServletContext` podremos almacenar objetos al igual que lo hacíamos a nivel de sesión, pero en este caso se trata de objetos a nivel de aplicación comunes a todos los usuarios. Al ser comunes estos objetos para todos los usuarios deberemos tratar con problemas de concurrencia, que veremos más adelante.

A continuación vamos a comentar los métodos que posee este interfaz:

- `Object getAttribute(String nombre)`: devuelve el objeto (atributo) almacenado en la aplicación (contexto), que se corresponde con el nombre que le indicamos por parámetro. Si el atributo no existe se devuelve un valor nulo (null).
- `Enumeration getAttributeNames()`: devuelve un objeto `java.util.Enumeration` que contiene todos los nombres de los atributos existentes en la aplicación Web.
- `ServletContext getContext(String ruta)`: devuelve un objeto `ServletContext` que se corresponde con un recurso en el servidor.
- `String getInitParameter(String nombre)`: este método devuelve una cadena que contiene el valor de un parámetro de inicialización del contexto. Si el parámetro no existe se devolverá el valor null. Estos parámetros serían muy similares a los parámetros de inicialización de un servlet, pero en este caso se aplican a una aplicación Web completa.
- `Enumeration getInitParameters()`: devuelve en un objeto `java.util.Enumeration` los nombres de todos los parámetros de inicialización de la aplicación Web.
- `int getMajorVersion()`: devuelve el número de versión superior que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
- `String getMimeType(String fichero)`: devuelve en una cadena el tipo MIME del fichero especificado, si el tipo MIME es desconocido se devolverá el valor null.
- `int getMinorVersion()`: devuelve el número de versión inferior que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
- `RequestDispatcher getNamedDispatcher(String nombre)`: devuelve un objeto `RequestDispatcher` que actuará como un envoltorio que representa al servlet cuyo nombre se pasa por parámetro.
- `String getRealPath(String ruta)`: devuelve un objeto `String` que contiene la ruta real (física) de una ruta virtual especificada como parámetro.
- `RequestDispatcher getRequestDispatcher(String ruta)`: este método ya lo hemos utilizado en capítulos anteriores para obtener una referencia a un recurso en el servidor y poder incluir su ejecución en el servlet actual o redirigir la ejecución a este recurso.
- `URL getResource(String ruta)`: devuelve un objeto `URL` que se corresponde con el recurso localizado en la ruta indicada como parámetro.
- `InputStream getResourceAsStream(String ruta)`: devuelve el recurso localizado en la ruta indicada como un objeto `InputStream`.
- `String getServerInfo()`: devuelve el nombre y la versión del contenedor de servlets en el que el servlet se está ejecutando.
- `void log(String mensaje)`: escribe el mensaje especificado en el fichero de registro de un servlet.
- `void log(String mensaje, Throwable throwable)`: escribe un mensaje descriptivo y un volcado de pila para una excepción determinada en el fichero de registro de un servlet.

- `void removeAttribute(String nombre)`: elimina el atributo indicado de la aplicación Web, liberando todos los recursos asociados.
- `void setAttribute(String nombre, Object objeto)`: almacena un objeto determinado en la aplicación, este método y el anterior tienen el mismo significado que los que aparecían en el interfaz `HttpSession`, pero en este caso se trata de almacenar, recuperar y eliminar objetos a nivel de aplicación. Los objetos almacenados en la aplicación existirán durante toda la vida de la aplicación Web, a no ser que se eliminen mediante el método `removeAttribute()`. Por lo tanto no existe un tiempo máximo de inactividad ni un método para destruir la aplicación, como ocurría con el interfaz `HttpSession`. Una aplicación se destruirá cuando se detenga el contenedor de servlets que la contiene.

En el siguiente apartado nos vamos a centrar en como manejar los atributos a nivel de aplicación.

## Almacenando y recuperando objetos de la aplicación

La forma de recuperar y almacenar objetos en la aplicación (objeto `ServletContext`), es prácticamente igual a como lo hacíamos con la sesión (objeto `HttpSession`), igualmente disponemos de los métodos `getAttribute()` para obtener un objeto de la aplicación, `setAttribute()` para almacenar un objeto en la aplicación, y `removeAttribute()` para eliminar un objeto de la aplicación.

Se debe indicar que para mantener el estado de aplicación, es decir, almacenar y recuperar objetos de la aplicación, no es necesario utilizar un mecanismo de más bajo nivel, como ocurría con la sesión que se basaba en la utilización de cookies.

Pero debemos tener en cuenta dos apreciaciones importantes acerca del uso de la aplicación. La primera de estas consideraciones es relativa a la necesidad de una sincronización a la hora de utilizar los atributos de la aplicación. Un objeto almacenado en una aplicación es accesible por todos los usuarios conectados a la aplicación, a diferencia de la sesión, que cada usuario posee su propia sesión y por lo tanto sus propios objetos. Debido a esto a la hora de acceder a un objeto de la aplicación se puede producir accesos simultáneos, dando lugar a problemas de concurrencia, por ejemplo a la hora de modificar un valor de un objeto (atributo) de la aplicación.

Para evitar estos problemas de concurrencia tenemos dos soluciones. Una de ellas es la implementación por parte del servlet, que va a acceder a los objetos de la aplicación, del interfaz `javax.servlet.SingleThreadModel`. Si el servlet implementa este interfaz, nos asegura que sólo va a servir una petición cada vez, por lo tanto no existirán múltiples hilos de ejecución (uno por cada petición) que pueda acceder a un mismo objeto de la aplicación al mismo tiempo.

El interfaz `SingleThreadModel` no posee ningún método, si queremos que un servlet lo implemente simplemente debemos indicarlo en la cláusula `implements` de la declaración de la clase del servlet.

Esta solución es la más sencilla, pero también la más drástica, ya que eliminamos la interesante característica que nos brindaban los servlets a través de los múltiples hilos de ejecución simultáneos. Además si se accede al servlet que implementa el interfaz `SingleThreadModel` de forma frecuente, se verá afectado el rendimiento de la aplicación Web, ya que cada petición deberá esperar en una cola hasta que la instancia del servlet se encuentre libre.

Otro motivo para desechar esta solución de forma definitiva, es el hecho de que algunos contenedores de servlets, cuando se indica que el servlet implementa el interfaz `SingleThreadModel`, crearán un conjunto de instancias de servlets, y por lo tanto no se podrá asegurar el acceso exclusivo a un objeto de la aplicación, ya que dos instancias distintas de un servlet podrían estar accediendo al mismo objeto de la aplicación al mismo tiempo.

Se debe señalar que los problemas de concurrencia no afectan exclusivamente a la hora de utilizar los objetos a nivel de aplicación, sino que también podemos tener el problema de los accesos simultáneos a la hora de acceder a los atributos del propio servlet (variables miembro) y a la hora de acceder a métodos estáticos de un servlet.

La segunda solución que se comentaba, y que resulta la ideal, consiste en permitir la ejecución multihilo de los servlets y solucionar los problemas de sincronización utilizando la palabra clave *synchronized* cuando exista un problema potencial de accesos concurrentes. En los siguientes ejemplos veremos como utilizar este mecanismo.

Otra consideración a la hora de utilizar los objetos a nivel de aplicación y que difiere con los objetos a nivel de sesión, se refiere a que la aplicación existe mientras el contenedor de servlets se esté ejecutando, por lo tanto no será necesario crear la aplicación en ningún momento, y la destrucción o eliminación de un objeto almacenado en la aplicación no se notifica al objeto afectado mediante ningún evento.

Una vez realizadas estas consideraciones y apreciaciones vamos a pasar a mostrar un ejemplo de un servlet que comprueba si existe un objeto en la aplicación, y si no existe lo crea. Si el objeto existe simplemente mostrará el valor del mismo. El Código Fuente 63 es el que se corresponde con este servlet.

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

public class ServletAplicacion extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        ServletContext application=getServletContext();
        out.println("<html><head>"+
            "<title>Objeto ServletContext</title></head><body>");
        synchronized(application){
            String nombreAplicacion=(String)application.getAttribute
                ("nombreAplicacion");
            if(nombreAplicacion!=null)
                out.println("El objeto nombreAplicacion ya existe y su valor es: <b>"+
                    nombreAplicacion+"<b>");
            else{
                application.setAttribute("nombreAplicacion",
                    new String("Aplicación con servlets"));
                out.println("Se ha creado el objeto nombreAplicacion");
            }
        }
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 63

Se puede observar en el código fuente anterior que se ha utilizado la palabra clave `synchronized` aplicándola al objeto `ServletContext`. Sólo se debe utilizar `synchronized` cuando sea absolutamente necesario, y se debe aplicar al objeto adecuado, como ocurre en este caso.

Si ejecutamos el servlet por primera vez, aparecerá un resultado como el de la Figura 40, ya que en un primer momento se creará el objeto y se almacenará en la aplicación.

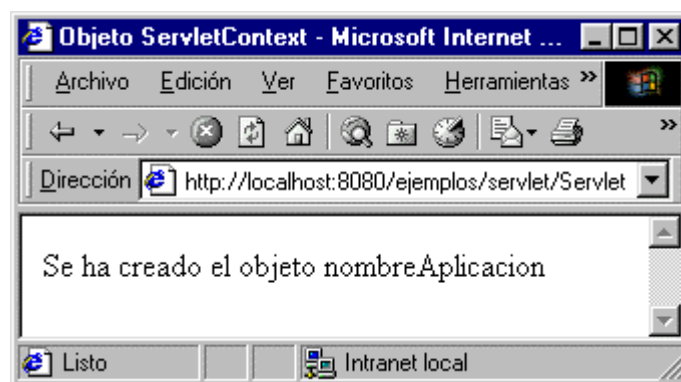


Figura 40. Creación del objeto a nivel de aplicación

Si cerramos el navegador y volvemos a ejecutar aparecerá un resultado como el de la Figura 41, ya que el objeto ya se encuentra almacenado en la aplicación y tendrá vigencia hasta que se detenga la ejecución del contenedor de servlets.

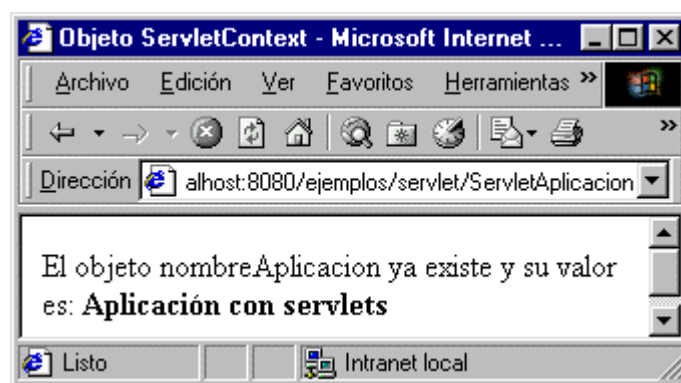


Figura 41. Se recupera el atributo de la aplicación ya creado

Otro ejemplo interesante de utilización de los atributos de la aplicación es la de describir el ejemplo de un apartado anterior en el que se mostraba un contador de accesos para una sesión determinada de un cliente. En este caso se a realizar un contador global, que va a almacenar los accesos de todos los clientes de la aplicación a un servlet determinado.

El código (Código Fuente 64) de este servlet es el que se muestra a continuación.

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

```
public class ServletAccesosAplicacion extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        ServletContext application=getServletContext();
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Accesos a la aplicación</title></head><body>");
        synchronized(application){
            Integer valorActual= (Integer)application.getAttribute("accesos");
            if (valorActual!=null)
                application.setAttribute("accesos",
                    new Integer(valorActual.intValue()+1));
            else
                application.setAttribute("accesos",new Integer(1));
            out.println("A este servlet se ha accedido: <b>"+
                application.getAttribute("accesos")+"</b> veces");
        }
        out.println("</body></html>");
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 64

Como se puede comprobar se utiliza también la sincronización, ya que se puede dar la situación en la que dos clientes accedan al mismo tiempo al servlet, con lo que se podría perder uno de los accesos en la cuenta de accesos, o realizar una cuenta incorrecta, por lo tanto es necesario utilizar la palabra clave `synchronized` sobre el objeto `ServletContext`.

Para realizar una prueba con este nuevo servlet podemos cargar varias veces el servlet en una misma sesión del navegador, luego cerraremos el navegador y al volver a cargar el servlet comprobaremos que la cuenta de accesos se mantiene. Esta cuenta de accesos se pondrá a cero si detenemos el contenedor de servlets.

En la Figura 42 se puede ver un ejemplo de ejecución de este servlet.



Figura 42. Accesos realizados a nivel de aplicación

En el siguiente apartado, que será el que cerrará este capítulo vamos a mostrar los métodos que ofrece el interfaz ServletContext para obtener información relativa al contenedor de servlets.

## Características del contenedor de servlets

Como ya hemos comentado en diversas ocasiones, el interfaz ServletContext nos permite tener acceso al contenedor de servlets en el que se está ejecutando nuestro servlet. Esto nos permite obtener información acerca del producto utilizado y las versiones que soporta del API Java servlet.

También podremos almacenar información del tipo de mensajes de alerta o de depuración en los ficheros de registro del contenedor de servlets, gracias también al interfaz ServletContext.

Vamos a ver estas dos funcionalidades ofrecidas por el interfaz ServletContext a través de un par de ejemplos muy sencillos.

El primer ejemplo se trata de un servlet que va a mostrar el producto utilizado como contenedor de servlets así como la versión que soporta del API servlet. El Código Fuente 65 muestra este servlet.

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

public class ServletContenedor extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{

        ServletContext application=getServletContext();
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "    <title>Características del contenedor de servlets"+
            "    </title></head><body>");
        out.println("Nombre del contenedor: <b>"+
            application.getServerInfo()+"</b><br>");
        out.println("Versión del API servlet: <b>"+application.getMajorVersion()+"."+
            application.getMinorVersion()+"</b>");
        out.println("</body></html>");

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }

}
```

Código Fuente 65

Para obtener el nombre del software utilizado como contenedor de servlets se ha utilizado el método `getServerInfo()`, y para obtener las versiones del API servlet que se implementan se han utilizado los métodos `getMajorVersion()` y `getMinorVersion()`.



En la Figura 43 se puede ver un ejemplo de ejecución del mismo.



Figura 43. Información sobre el contenedor de servlets

En nuestro segundo ejemplo vamos a mostrar un servlet que escribe en el fichero de registro del contenedor los distintos pasos por los que va pasando en su ciclo de vida. Para ello utiliza el método `log()` del interfaz `ServletContext`, en cada método que se corresponde con el ciclo de vida del servlet, se indica en el fichero de registro que se está ejecutando dicho método. El código completo de este servlet se puede ver a continuación (Código Fuente 66).

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

public class ServletLog extends HttpServlet {

    public void init() throws ServletException{
        ServletContext application=getServletContext();
        synchronized(application){
            application.log("Ejecuto init()");
        }
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException{

        ServletContext application=getServletContext();
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>"+
            "<title>Servlet que escribe en el log</title></head><body>");
        synchronized(application){
            application.log("Ejecuto doGet()");
        }
        out.println("<h1>Servlet que escribe en el log</h1>");
        out.println("</body></html>");

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException{
        doGet(request,response);
    }

    public void destroy(){
```

```
ServletContext application=getServletContext();
synchronized(application){
    application.log("Ejecuto destroy()");
}
}
```

Código Fuente 66

El fichero de registro en el caso del contenedor Jakarta Tomcat, se encuentra en el directorio `c:\jakarta-tomcat\logs` y se llama `SERVLET.LOG`. Por lo tanto si ejecutamos el servlet anterior y recargamos varias veces el navegador y detenemos la ejecución del contenedor de servlets, al abrir el fichero `SERVLET.LOG`, entre otras muchas líneas, podremos ver las que se muestran a continuación como resultado de la ejecución de este servlet.

```
Context log path="/ejemplos" :invoker: init
Context log path="/ejemplos" :ServletLog: init
Context log path="/ejemplos" :Ejecuto init()
Context log path="/ejemplos" :Ejecuto doGet()
Context log path="/ejemplos" :Ejecuto doGet()
Context log path="/ejemplos" :Ejecuto doGet()
Context log path="/ejemplos" :Ejecuto doGet()
Context log path="/ejemplos" :invoker: destroy
Context log path="/ejemplos" :Ejecuto destroy()
```

Como se puede ver el método `init()` y `destroy()` se han ejecutado una única vez cada uno de ellos, sin embargo el método `doGet()` se ha ejecutado varias veces.

Con este capítulo finalizamos la primera parte de este texto en la que se pretendía ofrecer una visión más o menos detallada del desarrollo de los servlets de Java. En la segunda parte del texto, que comienza con el siguiente capítulo, vamos a mostrar una extensión de los servlets que se denomina `JavaServer Pages (JSP)`.

Aunque no vamos a abandonar los servlets de forma definitiva y radical, ya que como veremos más adelante las páginas `JSP (JavaServer Pages)` y los servlets se encuentran íntimamente relacionados, de hecho veremos que una página `JSP` será convertida a un servlet por el contenedor `JSP` correspondiente, y en nuestro caso el contenedor de páginas `JSP` va a ser el mismo que el contenedor de servlets, es decir, el servidor Jakarta Tomcat, ya que Tomcat implementa también el `API JSP 1.1`.

En el siguiente capítulo se realizará una introducción a las páginas activas de servidor Java, es decir, las páginas `JSP (JavaServer Pages)`, además veremos porque es necesaria la especificación `JavaServer Pages`, y se comentarán las diferencias, ventajas y relación que poseen con los servlets.

El texto se ha estructurado en estas dos partes debido a que para entender bien las páginas `JSP` junto con su modelo de objetos es necesario previamente conocer y entender los servlets.

# Introducción a JSP (JavaServer Pages)

---

## Introducción

Con este capítulo se inicia la segunda parte del presente texto, los siete capítulos anteriores los hemos dedicado por completo al API Java servlet 2.2, ahora nos vamos a centrar en otro API perteneciente a la plataforma Java2 Enterprise Edition (J2EE), se trata del API JSP 1.1, que implementa la tecnología JavaServer Pages.

Este capítulo es, como su título indica, introductorio y se intenta definir la tecnología JSP comparándola con los servlets, y además se intenta mostrar que utilizar páginas JSP no implica abandonar los servlets, ya que pueden trabajar ambas tecnologías de forma conjunta.

También se comentarán las características de las páginas JSP y que beneficios aportan sobre los ya conocidos servlets. Los beneficios más importantes que ofrecen las páginas JSP adelantamos que son: la posibilidad de separar la presentación de la información de la lógica propia de la aplicación, utilización sencilla de componentes basados en el modelo de componentes de Java denominado JavaBeans, permitir la separación del trabajo de los desarrolladores y los diseñadores de contenidos para la Web.

Se puede decir que la tecnología JSP es una extensión de la tecnología ofrecida por los servlets creada para ofrecer soporte a la creación de páginas HTML y XML. Las páginas JSP hacen muchos más fácil combinar plantillas de presentación de la información con contenido dinámico. Las páginas JSP realmente son compiladas dinámicamente y automáticamente en forma de servlets, de esta forma presentan todos los beneficios de los servlets, así como el completo acceso a todos los APIs que ofrece el lenguaje Java.

Como ya hemos apuntado, y queremos que quede claro, JSP es una extensión de los servlets, y las páginas JSP y los servlet se pueden combinar para trabajar conjuntamente, veremos que las páginas se compilan de forma automática en servlets y veremos que desde una página JSP podremos utilizar un servlet y viceversa (incluyendo su contenido, ejecutándolo o redirigiendo el navegador).

La especificación JavaServer Pages 1.1 se encuentra implementada en los paquetes `java.servlet.jsp` y `java.servlet.jsp.tagext`, estos paquetes los podemos encontrar dentro de la plataforma Java 2 Enterprise Edition (J2EE). Aunque veremos que las páginas JSP hacen uso continuo de los paquetes que realizaban la implementación de la especificación Java servlet 2.2, sobretodo del paquete `javax.servlet.http`, ya que los objetos integrados (o implícitos) que ofrecen las páginas JSP son objetos que son instancias de los interfaces que podemos encontrar en un servlet.

En este capítulo se mostrará una visión general de las distintas clases e interfaces que realizan la implementación de la especificación JavaServer Pages 1.1.

Para ejecutar las páginas JSP en un servidor Web es necesario disponer de un motor (engine) o contenedor (container) de páginas JSP. El servidor Yakarta Tomcat ofrece también soporte para páginas JSP, implementando a través del contenedor de páginas JSP la especificación JavaServer Pages 1.1, por lo tanto no es necesario disponer de ningún software adicional a la hora de seguir esta segunda parte del texto, aunque si recomendamos una sencilla y útil herramienta que nos permitirá desarrollar nuestras páginas JSP, se trata de JRun Studio de Allaire, aunque no es necesaria para el correcto seguimiento del texto, ofrece una serie de facilidades, de este software se puede conseguir una versión de evaluación en la dirección <http://www.allaire.com>.

La configuración de la que ya disponemos en nuestra máquina de trabajo, es decir, la herramienta JDK 1.3 (o plataforma Java 2 Standard Edition, J2SE), variable de entorno CLASSPATH y servidor Jakarta Tomcat 3.1 nos sirve para la segunda parte de nuestro texto.

Más adelante, en este mismo capítulo, veremos como crear y ejecutar nuestra primera página JSP, comentando y comprendiendo todo lo que ocurre detrás de la ejecución de la misma.

## Definición de página JSP

Una página JSP va a ser un fichero de texto con la extensión JSP, que combina etiquetas HTML con nuevas etiquetas de script pertenecientes a la especificación JavaServer Pages 1.1. Una página JSP va a tener un aspecto muy similar al de una página HTML, pero se transformarán en clases de Java que son servlets para compilarse y generar los ficheros de clase correspondientes, esta operación se dará cuando se ejecute una página JSP por primera vez, o cuando se modifique una página JSP existente.

El servlet que resulta de la transformación de la página JSP es una combinación del código HTML contenido en la página JSP y del contenido dinámico indicado por las etiquetas especiales pertenecientes a la especificación JSP.

Dentro de una página JSP se puede diferenciar claramente entre dos elementos (o categorías), por un lado encontramos elementos que son procesados por el servidor que contiene el contenedor de páginas JSP, y por otro lado encontramos código que el contenedor de páginas JSP ignora, normalmente este código suele ser código HTML que se devuelve al cliente como tal. A lo largo de los distintos capítulos de este texto iremos viendo y utilizando los distintos elementos que la especificación JavaServer Pages 1.1 ofrece para utilizar dentro de las páginas JSP.

Cuando un cliente realiza una petición de una página JSP, se ejecutará dicha página JSP devolviendo como resultado código HTML que se mostrará en el navegador, este código HTML es el resultado de la ejecución de la página JSP y comprende el código HTML contenido en la página y el resultado del

contenido dinámico que ha sido ejecutado por el contenedor de páginas JSP. El cliente (navegador Web), nunca va a poder observar el código fuente de la página JSP, lo que recibe es el resultado de la ejecución de la misma.

Para el lector que conozca la tecnología de Microsoft llamada páginas ASP (Active Server Pages), debe pensar que las páginas JSP son muy similares a las páginas ASP, pero sólo conceptualmente, luego veremos que su funcionamiento interno y ejecución son bastante distintos. Dedicaremos un capítulo para realizar una comparativa entre las páginas JSP y las páginas ASP, comentando sus similitudes y diferencias.

Un página JSP podemos decir que presenta un mayor nivel de abstracción de los servlets, ya que en muchos aspectos se puede considerar que el API JavaServer Pages es de más alto nivel que el API Java servlet.

Las páginas JSP las situaremos en el directorio que deseemos dentro de nuestra aplicación Web, no tiene un directorio de publicación determinado como ocurría con los servlets, que se debían situar siempre a partir del directorio WEB-INF\CLASSES de la aplicación Web. De todas formas cuando más adelante escribamos nuestra propia página JSP veremos dónde podemos situarla.

Si al lector le han quedado claros los distintos conceptos, clases e interfaces vistos en la primera parte del curso dedicada a los servlets, no le debería resultar muy complicado la segunda parte del texto dedicada, como todos ya sabemos, a la especificación JSP. Las páginas JSP utilizan muchos de los interfaces vistos en capítulos anteriores a través de los objetos integrados que poseen, como pueden ser el interfaz HttpSession, HttpServletRequest o HttpServletResponse.

Las páginas JSP se encuentran orientadas tanto para desarrolladores como para diseñadores de sitios Web, ya que ofrece una serie de etiquetas especiales, al estilo de XML, que permiten obtener una funcionalidad muy interesante, y también es posible incluir código Java tradicional dentro de las páginas JSP, pero no adelantemos acontecimientos, que veremos con más detalle cuando tratemos de forma exhaustiva la sintaxis de las páginas JSP.

## Beneficios de las páginas JSP

Las páginas JSP ofrecen una serie de ventajas sobre los servlets, que hacen su uso más idóneo para la generación de contenidos dinámicos dentro de un sitio Web. En este apartado vamos a comentar lo que pueden ofrecernos las páginas JSP y porque puede ser interesante utilizarlas, ya que para generar contenido dinámico ya disponemos de los servlets, pero cada tecnología la utilizaremos para un contexto y una función determinada.

Unos de los mayores beneficios que ofrecen las páginas JSP es la separación del contenido estático o presentación del resultado del contenido dinámico o la implementación de la lógica que produce el resultado. Cuando en capítulos anteriores utilizábamos un servlet para generar una salida para el usuario, el aspecto de esta salida, es decir, las etiquetas HTML que utilizamos para ofrecer una presentación del resultado devuelto por el servlet, eran parte del propio servlet, ya que debíamos incluir la generación de las mismas en instrucciones del tipo `out.println("<etiqueta>")`.

En los servlets si deseamos modificar algo relacionado con el interfaz de usuario debemos modificar el código fuente del servlet y volver a compilarlo, lo que supone un problema, ya que por muy pequeño que sea el cambio a realizar siempre afecta al código fuente del servlet.

En muchos casos la lectura y comprensión del código fuente de los servlets se presenta bastante complicada, ya que la presentación de la información se confunde fácilmente con la implementación propiamente dicha de las funciones que realiza el servlet.

Sin embargo, como iremos viendo a lo largo de los siguientes capítulos del texto, las páginas JSP ofrecen una separación más clara entre presentación e implementación, ya que las páginas JSP ofrecen unos delimitadores especiales, que se denominan delimitadores de scriptlets, que permiten distinguir el código que se encuentra escrito completamente en Java y que constituye la implementación de la página JSP, del código HTML, que se escribirá como si estuviéramos diseñando una página HTML tradicional, y que constituye la presentación (interfaz de usuario) de la información.

Además de los delimitadores de scriptlets (fragmentos de código fuente escritos en lenguaje Java), las páginas JSP ofrecen una serie de etiquetas especiales al estilo de las etiquetas XML, que permiten realizar una serie de acciones determinadas. Todas estas etiquetas las veremos con más detenimiento en capítulos siguientes.

A través de las etiquetas propias de JSP podemos reutilizar componentes Java que siguen la especificación JavaBeans, con lo que nos permite abstraernos mucho más de la implementación de la aplicación Web.

Una modificación en una página JSP la recogerá de forma automática el contendor de páginas JSP, precompilando de forma automática la página para generar la clase del servlet correspondiente, ya que, como ya hemos dicho anteriormente,

De la característica de la separación entre la presentación y la implementación que nos ofrecen las páginas JSP, se extrae otro beneficio de las páginas JSP, que consiste en ofrecer una división más clara del trabajo a la hora de acometer desarrollos de aplicaciones Web basadas en páginas JSP.

En grandes o medianos desarrollos podemos separar el trabajo de los diseñadores Web de los desarrolladores Java, de esta forma un diseñador Web se puede encargar de diseñar la presentación ofrecida por la página JSP, mientras que el desarrollador puede encargarse de la implementación del código fuente y de los componentes JavaBeans.

Para un diseñador Web será mucho más sencillo acceder a una página JSP para editarla y modificarla, ya que como veremos más adelante, podemos tener páginas JSP que no presenten ninguna línea de código fuente en Java, sino que la implementación la consiguen a través de la utilización de etiquetas XML especiales y del uso de componentes JavaBeans. De todas formas, aunque una página JSP posea código fuente en Java está claramente delimitado, por lo que al diseñador Web le seguirá resultando más fácil que modificar un servlet o encargarse de la presentación del mismo.

Las páginas JSP ofrecen un mecanismo que permite definir etiquetas de usuario para utilizarlas dentro de las propias páginas JSP, en este caso se produce una mezcla entre presentación e implementación, pero puede ser un mecanismo muy potente en manos de los desarrolladores Java, que pueden definir una serie de etiquetas que luego los diseñadores pueden utilizar de forma sencilla dentro de las páginas JSP.

Otro beneficio de las páginas JSP es que si ya sabemos desarrollar servlets, desarrollar páginas JSP no nos va a resultar complicado, ya que se basan en los mismos principios que los servlets, utilizando objetos integrados que corresponden con interfaces de la especificación Java servlet. Incluso podemos utilizar de forma conjunta los servlets y las páginas JSP, como se propone en diferentes arquitecturas de aplicaciones Web dentro de la plataforma Java 2 Enterprise Edition, que veremos más adelante en este texto.

Como resumen de las facilidades ofrecidas por las páginas JSP sobre los servlets podemos destacar los siguientes puntos:

- Con las páginas JSP es más sencillo combinar plantillas estáticas de etiquetas HTML o XML con el código que genera el contenido dinámico.

- Las páginas JSP se compilan de forma dinámica en un servlet cuando se realiza una petición.
- La estructura de las páginas JSP hace más fácil el diseño de páginas a manos de los diseñadores Web. Además están apareciendo herramientas que nos asisten en la creación y diseño de páginas JSP, como puede ser JRun Studio de Allaire.
- Las etiquetas que ofrecen las páginas JSP para invocar a componentes JavaBeans, permiten utilizar a los autores de estas páginas la funcionalidad de estos componentes sin tener que conocer la complejidad de su implementación.

A continuación vamos a comentar de forma breve los distintos elementos que nos puede ofrecer una página JSP, estos elementos se tratarán con mayor profundidad a mediada que vayamos avanzando en esta segunda parte del texto.

## Elementos de las páginas JSP

Para clasificar los elementos que podemos encontrar dentro de una página JSP podemos realizar una primera división muy general en la que dividimos los elementos de las páginas JSP en dos grandes categorías: elementos estáticos que no son interpretados por el contenedor de páginas JSP y que son devueltos sin ningún tipo de tratamiento para que los interprete el navegador, y elementos dinámicos que son interpretados por el contenedor de páginas JSP devolviendo al navegador el resultado de la ejecución de los mismos.

Dentro del conjunto de elementos dinámicos que puede presentar una página JSP, y que forma parte de la especificación JavaServer Pages 1.1, distinguimos los siguientes:

- Directivas: las directivas funcionan como mensajes que se envían desde la página JSP al contenedor de páginas JSP. Se utilizan para establecer valores globales que afectarán a la página JSP actual, estas directivas no presentan ninguna salida al cliente. Su sintaxis general es la que se indica a continuación.

```
<%@nombreDirectiva atributo1="valor1"...atributon="valorn"%>
```

- Elementos de scripting: estos elementos permiten incluir código Java en la página JSP. Permiten declarar objetos, instanciarlos, ejecutar métodos, definirlos, etc. Los elementos de scripting se pueden separar en cuatro subelementos que son:
  - Declaraciones: son bloques de código Java incluido en la página JSP utilizados para declarar variables y métodos propios dentro de la página JSP. Un bloque de declaración se encuentra entre los delimitadores `<%! %>`.
  - Scriptlets: un scriptlet es un fragmento de código Java incluido en una página JSP que se ejecutará cuando se realice una petición de la misma. Un scriptlet se encontrará entre los delimitadores `<% %>`, como curiosidad cabe comentar que estos delimitadores son los mismos que utilizan las páginas ASP (Active Server Pages) para delimitar el script de servidor. En un scriptlet podemos encontrar cualquier código Java válida, no debemos olvidar que las páginas JSP al igual que los servlets pueden utilizar todos los APIs ofrecidos por el lenguaje Java.
  - Expresiones: una expresión es una notación especial para un scriptlet que devuelve un resultado a la respuesta dada al usuario, la expresión se evalúa y se devuelve como una cadena que se envía al cliente. Una expresión se encuentra entre los delimitadores

`<%= %>`, que son también los mismos delimitadores que se utilizan para devolver el valor de expresiones dentro de una página ASP.

- Comentarios: estos elementos permiten documentar nuestro código fuente, se encuentran entre los delimitadores `<%-- --%>`. Estos comentarios no serán visibles en el navegador, ya que son comentarios de JSP que serán tratados por el contenedor de páginas JSP, no se deben confundir con los comentarios HTML (`<!-- -->`), que si serán visibles desde el navegador y enviados como tales al usuario.
- Acciones: las acciones son etiquetas específicas de JSP que afectan al comportamiento de la página JSP y a la respuesta enviada al usuario. La especificación JSP define una serie de acciones estándar que todo contenedor de páginas JSP debe ofrecer. Estas acciones ofrecen al diseñador de páginas JSP una funcionalidad bastante interesante, que veremos en detalle cuando tratemos las acciones en siguientes capítulos. También es posible definir acciones determinadas mediante el mecanismo de librerías de etiquetas. La sintaxis general de una acción JSP es:

```
<jsp: accion>
```

- Objetos implícitos o integrados: estos objetos se suelen utilizar dentro de expresiones o scriptlets, y no necesitan ser declarados ni instanciados, se encuentran disponibles dentro de cada página JSP. Casi todos estos objetos implementan interfaces que podíamos encontrar en los servlets, como pueden ser el interfaz `HttpServletRequest`, `HttpServletResponse` o `HttpSession`. Estos objetos son similares a los objetos integrados que ofrece ASP:

Aunque el lector no haya entendido algunos de los conceptos comentados en este apartado, o no los tenga demasiado claros, no debe preocuparse, ya que se ha pretendido ofrecer una visión muy general de los elementos que podemos encontrar en una página JSP. En siguientes capítulos trataremos estos elementos en profundidad, pero en ese momento puede ser interesante revisar este apartado debido a la visión general que ofrece.

En el siguiente apartado vas a crear nuestra primera página JSP para hacer un pequeño paréntesis en la a veces aburrida, pero siempre necesaria teoría.

## Hola Mundo con JSP

Hasta ahora no hemos visto ni una sola línea de una página JSP, por lo tanto este apartado pretende ser más práctico y mostrar una sencilla página JSP. También veremos como podemos ejecutar la página JSP.

Esta primera página JSP va a ser muy sencilla y va a consistir simplemente en mostrar el típico mensaje “Hola Mundo”. El Código Fuente 67 muestra el contenido de esta página JSP.

```
<%@ page language="Java"%>
<html>
<head>
  <title>Hola mundo con JSP</title>
</head>
<body>
  <!--Esto es un comentario--%>
  <div align="center">
    <b><%out.println("Hola Mundo");%></b>
  </div>
```



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

Código Fuente 67

A la vista de este código podemos identificar algunos de los elementos que podemos encontrar dentro de una página JSP. En primer lugar encontramos una directiva que indica el lenguaje que se va a utilizar en la página JSP, a continuación aparecen elementos estáticos que son etiquetas HTML que se enviarán tal cual al cliente (navegador Web).

A continuación aparece un comentario de JSP entre los delimitadores `<%-- --%>`. Y entre estos elementos estáticos tenemos un scriptlet identificado por los delimitadores `<% %>`, que se encarga de mostrar el mensaje **Hola Mundo** en pantalla.

Dentro de este scriptlet se hace uso de un objeto integrado de JSP, se trata del objeto `out`, este objeto es un objeto de la clase `javax.servlet.jsp.JspWriter`, siendo muy similar a la clase `java.io.PrintWriter` que utilizábamos en los servlet para enviar contenidos al usuario.

Para ejecutar esta página JSP se supone que ya tenemos disponible todo el entorno de los servlets, es decir, variable `CLASSPATH` configurada y servidor Jakarta Tomcat 3.1 instalado y configurado. Una vez que hemos creado el fichero `HOLAMUNDO.JSP`, utilizando para ello el bloc de notas o cualquier editor de texto sencillo, que contendrá el código de la página JSP correspondiente, debemos copiar el fichero de la página JSP al directorio de publicación que se corresponde con la aplicación Web dentro de Jakarta Tomcat que va a contener la página JSP.

En este caso si la aplicación se llama `ejemplos` y su directorio físico es `c:\work\ejemplos`, la podemos copiar a partir de este directorio en cualquier localización, no es necesario copiar la página JSP a un directorio especial como ocurría con los servlets.

Así si copiamos la página `HOLAMUNDO.JSP` al directorio `c:\work\ejemplos`, la podemos ejecutar desde el navegador indicando la siguiente dirección: `http://localhost:8080/ejemplos/holamundo.jsp`. Al contrario que con los servlets, con las páginas JSP es indiferente la utilización de mayúsculas o minúsculas a la hora de invocar su ejecución desde el navegador.

El resultado de la ejecución de esta página JSP se puede ver en la Figura 44.



Figura 44. Hola Mundo con JSP

Y si seleccionamos la opción del navegador para ver el código fuente de la página observaremos el código HTML del Código Fuente 68, y como se puede comprobar no hay rastro del código JSP que

pertenecía a la directiva, el comentario y el scriptlet de la página, ya que todos ellos pertenecen a la categoría de elementos dinámicos.

```
<html>
<head>
    <title>Hola mundo con JSP</title>
</head>

<body>
<div align="center">
<b>Hola Mundo
</b>
</div>
</body>
</html>
```

Código Fuente 68

Si comparamos el código fuente de la página JSP, con el código fuente del servlet equivalente que devolverá el mismo resultado (Código Fuente 69), vemos que es mucho más sencillo el código fuente de la página JSP, además de ser más claro.

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

public class ServletHolaMundo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>"+
            "<head><title>Hola Mundo con un servlet</title></head>"+
            "<body><div align='center'><b>Hola Mundo </b></div>"+
            "</body></html>"
        );
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        doGet(request,response);
    }
}
```

Código Fuente 69

Podemos modificar esta página para que salude al nombre que se le pasa como parámetro a través de una cadena de consulta (querystring) el nuevo código se puede ver en el Código Fuente 70.

```
<%@ page language="Java"%>
<html>
<head>
    <title>Hola mundo con JSP</title>
```

```

</head>
<!--Esto es un comentario--%>
<body>
<div align="center">
<b>
<%!String nombre;%>
<%nombre=request.getParameter("nombre");
if (nombre!=null)
    out.println("Hola "+nombre);
else
    out.println("Hola Mundo");%></b>
</div>
</body>
</html>

```

Código Fuente 70

En este caso añadimos un elemento más, que es un subelemento de los elementos de scripting, y se trata de la declaración de un objeto, que contendrá el valor del parámetro que se pasa a la página, además se utiliza otro objeto integrado en JSP, se trata del objeto request, que es un objeto que implementa el interfaz `javax.servlet.http.HttpServletRequest`. Como se puede comprobar al funcionalidad ofrecida por el objeto integrado request, así como su utilización son idénticas a la utilización del objeto instancia del interfaz `HttpServletRequest` que utilizábamos en nuestros servlets.

Para modificar la página JSP podemos hacerlo directamente sobre la página que se encuentra en el directorio de la aplicación Web, de esta forma al ejecutarse la página JSP se volverá a compilar en el servlet correspondiente, recogiendo los nuevos cambios.

En la Figura 45 se puede ver un ejemplo de ejecución de esta nueva versión de la página JSP.



Figura 45. Saludo con parámetro

Para finalizar este apartado vamos a realizar un cambio más a esta página JSP, para que además de saludar al nombre que se le pasa como parámetro indique la fecha y hora actuales. El Código Fuente 71 es el nuevo código de esta página JSP.

```

<%@ page language="Java" import="java.util.*"%>
<html>
<head>
    <title>Hola mundo con JSP</title>
</head>
<!--Esto es un comentario--%>
<body>
<div align="center">

```

```
<b>
<%!String nombre;%>
<%nombre=request.getParameter("nombre");
if (nombre!=null)
    out.println("Hola "+nombre);
else
    out.println("Hola Mundo");%>
<br>La fecha y hora actuales son:</b><i><%=new Date()%></i>
</div>
</body>
</html>
```

Código Fuente 71

En este nuevo ejemplo en la directiva se ha utilizado un atributo más para indicar que es necesario importar el paquete `java.util`, y para devolver la fecha y hora actual se ha utilizado una expresión de JSP con los delimitadores `<%= %>`.

Un ejemplo de ejecución de este ejemplo se encuentra en la Figura 46.

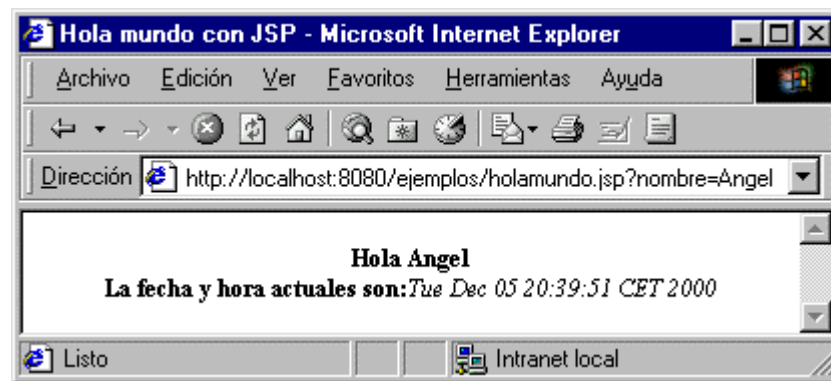


Figura 46. Saluda y muestra fecha

Con este ejemplo damos por terminado este apartado en el que se ha mostrado el aspecto de una página JSP muy sencilla y los pasos que necesitamos para ejecutarla. En el siguiente apartado veremos que ocurre internamente cuando se realiza una petición de una página JSP para proceder a su ejecución.

## Ejecución de las páginas JSP

Como ya hemos repetido en diversas ocasiones, a cada página JSP le corresponde un servlet que se genera de forma automática, y al invocar la página JSP es realmente este servlet el que se ejecuta. A continuación vamos a ver paso a paso el proceso que tiene lugar cada vez que realizamos una petición de una página JSP desde un navegador.

Todo se inicia cuando indicamos una URL que identifica a la página JSP deseada, tal como hacíamos para ejecutar los ejemplos anteriores. El servidor Web, en este caso Jakarta Tomcat reconoce la extensión `.JSP` del recurso que hemos especificado en la URL, y por lo tanto sabe que la petición se refiere a una página JSP que será tratada por el contenedor (container) o motor (engine) de páginas JSP. El motor de páginas JSP en el caso de Jakarta Tomcat se encuentra incluido en el propio servidor Web, por lo tanto podemos decir que Jakarta Tomcat es un servidor Web, un contenedor de servlets y un contenedor de páginas JSP.

Una vez identificado el recurso demandado como una página JSP, se procede a traducir la página JSP a un servlet equivalente, que va a realizar la misma función que la página JSP, es decir a partir de la página JSP se genera un fichero fuente de una clase de Java, que representa a un servlet. Una vez generado el fichero fuente se procede a realizar su compilación, generándose entonces el fichero de la clase del servlet que se corresponde con la página JSP.

Una vez generada la clase que se corresponde con el servlet equivalente a la página JSP, el contenedor de páginas JSP invoca a este servlet para que se ejecute y devuelva al cliente la respuesta correspondiente, que se corresponderá con el contenido dinámico especificado en la página JSP.

Este proceso de traducción de la página JSP a un fichero fuente de Java y su posterior compilación en la clase correspondiente, sólo tiene lugar la primera vez que se invoca a la página JSP o cuando la página sufre algún cambio, entonces será necesario generar de nuevo el servlet equivalente y compilarlo para que contemple los cambios realizados.

Si realizamos una nueva llamada a una página JSP que ya se ha ejecutado, y su código fuente no se ha modificado, directamente se ejecutará la clase del servlet equivalente, obteniendo al respuesta más rápidamente que en la primera invocación de la página JSP.

Incluso para mejorar el rendimiento de la ejecución de las páginas JSP, una vez que se ha ejecutado una página JSP su servlet equivalente permanece cargado en memoria, por lo que las siguientes llamadas a la misma página JSP serán más rápidas, ya que el acceso a memoria es siempre más rápido que el acceso a disco. El servlet se mantendrá en memoria mientras se sigan realizando peticiones a una página JSP con una cierta frecuencia.

Todo este proceso de traducción y compilación es completamente transparente al usuario, en ningún momento el usuario ni el navegador Web son conscientes de lo que realmente está sucediendo internamente en el servidor Jakarta Tomcat.

El encargado de realizar el proceso de traducción de la página JSP en el fichero fuente del servlet es curiosamente un servlet especial que se encuentra en el contenedor de páginas JSP, este servlet se le denomina compilador de páginas (page compiler). El compilador de páginas recorre el contenido de la página JSP para ir traduciéndolos en código fuente Java equivalente, que será un servlet. El código HTML estático se traduce a cadenas (objetos String) de Java, que se escribirán en el flujo de salida del servlet sin realizar ninguna modificación, los distintos elementos de la página JSP, etiquetas especiales, scriptlets, etc., se traducirán a código fuente Java equivalente.

Una vez que se ha construido el código fuente del servlet el servlet compilador de páginas llama al compilador de Java para que compile el código fuente del servlet que se ha generado. El compilador de Java generará el fichero .CLASS correspondiente que se situará junto con el fichero fuente en el servidor Jakarta Tomcat. Una vez que se ha compilado el servlet, el servlet compilador de páginas invoca al servlet resultante de la traducción de la página JSP para que se ejecute y devuelva la respuesta correspondiente al usuario. El usuario no tiene conocimiento de la existencia de este servlet, y para el usuario el resultado le ha sido devuelto directamente por la página JSP.

Cuando el compilador de páginas detecta una petición de una página JSP comprueba la fecha y hora de modificación de la página y la compara con el servlet equivalente para verificar si se ha producido una modificación en la página. Si no se encuentra un servlet equivalente esto es debido a que es la primera vez que se ha ejecutado la página y por lo tanto es necesario todo el proceso de traducción y compilación, y si la fecha y hora de modificación de la página JSP no coincide con la del servlet quiere decir que la página JSP se ha modificado y se deberá volver a generar el servlet.

La forma de ejecutar las páginas JSP ofrece una gran eficiencia respecto a otras tecnologías de servidor como pueda ser ASP (Active Server Pages). En otras tecnologías de generación dinámica de

contenidos, cada vez que se realiza una petición de una página se debe interpretar línea a línea, sin embargo con JSP se accede a una versión ya compilada de la página. Las páginas JSP serán más lentas que otras tecnologías la primera vez que se realice la petición de la página, pero en las siguientes peticiones serán más rápidas. En la Figura 47 se ofrece un esquema que intenta resumir todo el proceso que tiene lugar cada vez que se realiza una petición de una página JSP.

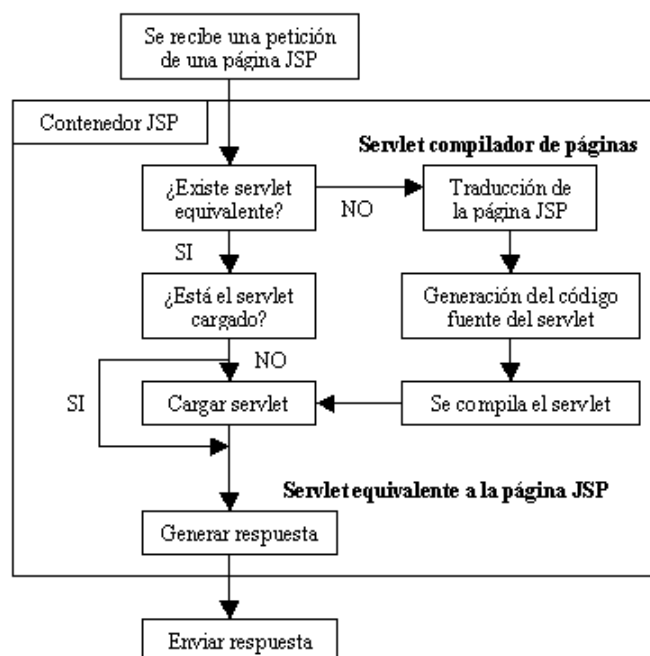


Figura 47. Proceso de ejecución de una página JSP

Y en la Figura 48 se muestra un esquema de la estructura interna del servidor Jakarta Tomcat, relacionándolo con los distintos contenedores y las páginas JSP y los servlets.

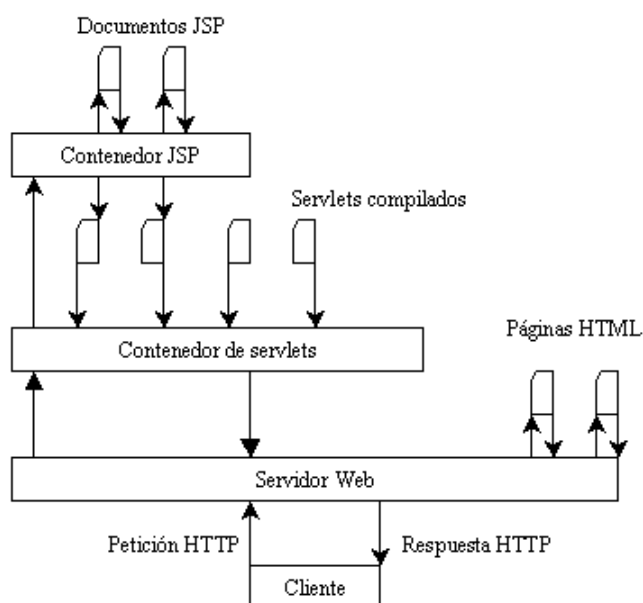


Figura 48. Estructura de los contenedores del servidor Tomcat

Para tener más clara la relación entre las páginas JSP y los servlets que se generan vamos a realizar una página JSP y luego vamos a mostrar el código fuente que genera el contenedor de páginas JSP para generar el servlet equivalente.

La página JSP del ejemplo va a ser muy sencilla, va a consistir en un bucle for que va a ir mostrando el mensaje Hola Mundo varias veces, y cada vez con un mayor tamaño de letra. En Código Fuente 72 el se muestra el contenido de esta página JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Hola Mundo 7 veces</title>
</head>

<body>
<%for (int i=1;i<=7;i++){%>
  <font size="<%=i%>">Hola Mundo</font><br>
<%}%>
</body>
</html>
```

Código Fuente 72

Si copiamos la página JSP al directorio de la aplicación Web correspondiente y la ejecutamos obtendremos un resultado como el de la Figura 49.



Figura 49. Resultado de la ejecución del bucle for.

La ejecución de esta página JSP ha generado dos ficheros, un fichero fuente de Java (.JAVA) y un fichero de clase (.CLASS), que se corresponden con el servlet equivalente a la página JSP. El servidor Jakarta Tomcat sitúa estos dos ficheros en un directorio de trabajo, existirá un directorio de trabajo

para cada aplicación Web, así si nuestra aplicación Web se llama ejemplos, su directorio de trabajo será c:\jakarta-tomcat\work\localhost\_8080%2Fejemplos.

El directorio de trabajo de la aplicación encontraremos un fichero .JAVA y otro fichero .CLASS que contienen en su nombre la cadena paginaJSP, es decir, el nombre de la página JSP a la que corresponden y a partir de la que se han generado. En mi caso se han generado los ficheros \_0002fpaginaJSP\_0002ejspaginaJSP\_jsp\_0.java y \_0002fpaginaJSP\_0002ejspaginaJSP.class. Si abrimos el fichero fuente del servlet, podemos ver el código Java que ha generado el contenedor de página JSP para crear el servlet equivalente a la página. El Código Fuente 73 muestra el contenido del fichero \_0002fpaginaJSP\_0002ejspaginaJSP\_jsp\_0.java.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fpaginaJSP_0002ejspaginaJSP_jsp_0 extends HttpJspBase {

    static {
    }
    public _0002fpaginaJSP_0002ejspaginaJSP_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext= null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
```



```

        session = pageContext.getSession();
        out = pageContext.getOut();

        // HTML // begin [file="C:\\paginaJSP.jsp";from=(0,0);to=(8,0)]
        out.write("<!DOCTYPE HTML PUBLIC \"/>
Transitional//EN\">\r\n\r\n<html>\r\n<head>\r\n\t<title>Hola Mundo 7
veces</title>\r\n</head>\r\n\r\n<body>\r\n");
        // end
        // begin [file="C:\\paginaJSP.jsp";from=(8,2);to=(8,25)]
        for (int i=1;i<=7;i++){
        // end
        // HTML // begin [file="C:\\paginaJSP.jsp";from=(8,27);to=(9,13)]
        out.write("\r\n\t<font size=\"");
        // end
        // begin [file="C:\\paginaJSP.jsp";from=(9,16);to=(9,17)]
        out.print(i);
        // end
        // HTML // begin [file="C:\\paginaJSP.jsp";from=(9,19);to=(10,0)]
        out.write("\r\n\t>Hola Mundo</font><br>\r\n");
        // end
        // begin [file="C:\\paginaJSP.jsp";from=(10,2);to=(10,3)]
        }
        // end
        // HTML // begin [file="C:\\paginaJSP.jsp";from=(10,5);to=(13,0)]
        out.write("\r\n</body>\r\n</html>\r\n");
        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

Código Fuente 73

Se ha considerado interesante mostrar este código fuente completo para que el lector vea claramente el proceso de traducción que se sigue para obtener el servlet equivalente a la página JSP. Las líneas de código que aparecen resaltadas son las que realizan la generación del contenido dinámico que pretendíamos obtener mediante la página JSP.

Como se puede ver en el Código Fuente 73 casi todo el contenido del servlet se encuentra el método `_jspService()`, cada vez que se produce una petición de la página JSP se ejecuta el método `_jspService()`, es el método equivalente al método `service()` que veíamos en el ciclo de vida de los servlets. Cuando más adelante tratemos el ciclo de vida de las páginas JSP veremos que además del método `_jspService()` disponemos de los métodos `jspInit()` y `jspDestroy()` para realizar labores de inicialización y de limpieza, respectivamente.

En el siguiente apartado vamos a comentar los paquetes que constituyen el API JavaServer Pages 1.1, comentaremos brevemente cada una de las clases e interfaces de estos paquetes.

## El API JavaServer Pages 1.1

El API JavaServer Pages se encuentra formado por las distintas clases e interfaces que ofrecen los paquetes `javax.servlet.jsp` y `javax.servlet.jsp.tagext`. A continuación vamos a comentar las clases, interfaces y excepciones que ofrece el paquete `javax.servlet.jsp` (Tabla 10, Tabla 11 y Tabla 12).

Interfaces	
HttpJspPage	Este es el interfaz que una clase generada por el procesador de páginas JSP debe implementar si se trata de utilizar el protocolo http, como normalmente suele suceder. Este interfaz posee solamente el método <code>_jspService()</code> , que será generado de forma automática por el contenedor de servlets a la hora de generar el servlet correspondiente a una página JSP.
JspPage	Este interfaz es el interfaz padre de <code>HttpJspPage</code> , y también debe ser implementado por la clase generada a partir de una página JSP. Este interfaz posee los métodos <code>jspInit()</code> , para realizar procesos de inicialización, y <code>jspDestroy()</code> para realizar labores de limpieza y liberación de recursos. Estos dos métodos puede ser implementados por el desarrollador de páginas JSP, sin embargo el método <code>jspService()</code> del interfaz hijo nunca debe ser implementado por el desarrollador (o diseñador) de páginas JSP, sólo puede encontrarse en el servlet generado de forma automática.

Tabla 10. Interfaces del paquete `javax.servlet.jsp`

Clases	
JspEngineInfo	Se trata de una clase abstracta que ofrece información relativa al contenedor de páginas JSP que se está utilizando. Posee un único método llamado <code>getSpecificationVersion()</code> , que devuelve el número de especificación del API JavaServer Pages implementado por el contenedor de páginas JSP.
JspFactory	Clase abstracta que define una serie de métodos disponibles para una página JSP en tiempo de ejecución, que permiten crear instancias de varios interfaces y clases utilizadas para soportar la especificación JSP.
JspWriter	Clase abstracta que emula parte de la funcionalidad ofrecida por las clases <code>java.io.BufferedWriter</code> y <code>java.io.PrintWriter</code> . Se utilizará para enviar contenido a la respuesta del cliente. Esta clase hereda de la clase <code>java.io.Writer</code> .
PageContext	Una instancia de esta clase ofrece acceso a todos los contenidos asociados con una página JSP, ofrece acceso a ciertos atributos de la página, así como una capa de abstracción sobre los detalles de implementación.

Tabla 11. Clases del paquete `javax.servlet.jsp`

Excepciones	
JspException	Una excepción genérica que será lanzada por el contenedor de páginas JSP.
JspTagException	Excepción que será lanzada por un manejador de etiquetas para indicar un error. El manejador o gestor de etiquetas es una clase especial que se utiliza dentro del mecanismo de etiquetas personalizadas de JSP, que se verá con más detalle en el momento oportuno.

Tabla 12. Excepciones del paquete javax.servlet.jsp

El segundo paquete que forma parte del API JavaServer Pages es javax.servlet.jsp.tagext, pero no vamos a mostrar ni comentar las distintas clases y interfaces, ya que este paquete tiene como función ofrecer el mecanismo de las librerías de etiquetas personalizadas, que todavía no hemos visto es este texto, pero que veremos más adelante, y será en ese momento cuando volveremos a retomar el paquete javax.servlet.jsp.tagext.

En este capítulo se ha pretendido realizar una introducción de la especificación JavaServer Pages, además se ha mostrado la relación existente entre las páginas JSP y los servlets, mostrando los beneficios que ofrecen las páginas JSP sobre los servlets.

También hemos construido unas cuantas páginas JSP sencillas y hemos visto como podemos ejecutarlas situándolas en los directorios adecuados dentro del servidor Web e incluso se ha mostrado todo el proceso interno que tiene lugar cada vez que se realiza la ejecución de una página JSP.

En este capítulo se han comentado de forma muy breve los distintos elementos que posee una página JSP, en el siguiente capítulo veremos con más detenimiento y profundidad todos estos elementos y también mostraremos su sintaxis correspondiente.

En el siguiente capítulo nos centraremos principalmente en las directivas de las páginas JSP, los distintos elementos de scripting y las acciones estándar que nos ofrece la especificación JSP.



# Páginas JSP: Directivas

---

## Introducción

En el capítulo anterior adelantábamos que una página JSP podía contener dos tipos de elementos bien diferenciados, por un lado teníamos los elementos estáticos como etiquetas HTML, texto, script de cliente, etiquetas XML, etc., que no necesitaban de ningún procesamiento por parte del contenedor de páginas JSP, sino que eran traducidos en el servlet correspondiente mediante sentencias `out.print()`, es decir, se devolvían directamente a la respuesta que se enviaba al cliente.

Y por otro lado teníamos otro grupo de elementos que eran las directivas, elementos de scripting (elementos para construir script de servidor con sentencias Java), acciones y objetos integrados. Este grupo de elementos necesitan de un procesamiento en el servidor (contenedor de páginas JSP) ya que son los encargados de generar el contenido dinámico, es decir, son los elementos que en forma de etiquetas similares a XML nos permiten programar nuestras páginas JSP.

En este capítulo y en los siguientes iremos detallando el uso y sintaxis de estas etiquetas especiales que ofrece la especificación JSP, cada etiqueta tendrá una traducción en el servlet que genere el contenedor de páginas JSP.

El primer elemento de las páginas JSP que vamos a tratar son las directivas, que se corresponden con una etiqueta especial de JSP que puede utilizarse utilizando la sintaxis XML, y se comenta en el siguiente apartado.

## Directivas

Las directivas son un conjunto de etiquetas JSP que ofrecen al contenedor de páginas JSP instrucciones específicas de cómo se debe procesar una página determinada. Las directivas definen propiedades generales que afectan a la página, incluso algunas de estas directivas se pueden utilizar para generar de forma indirecta contenido dinámico que será devuelto como parte de la respuesta enviada al cliente, y por lo tanto tienen una traducción correspondiente dentro del servlet equivalente a la página JSP. Las directivas sirven como mensajes que se envían desde la página JSP al contenedor JSP que la ejecuta.

Un ejemplo de uso de las directivas puede ser para indicar el tipo de lenguaje (lenguaje de script) que se va a utilizar dentro de la página JSP, para incluir contenidos de otra página o recurso del servidor, para indicar que la página usa una librería de etiquetas personalizada determinada, para importar paquetes del lenguaje Java, etc.

Las directivas no generan directamente una salida que se envíe al cliente de forma directa, sino que generan efectos laterales en la forma en la que el contenedor JSP procesa la página en la que se encuentra definida la directiva.

Las directivas se pueden situar en cualquier lugar de la página, pero para una mayor legibilidad del código de la página se recomienda que se sitúen al principio de la misma.

Antes de comenzar a comentar la sintaxis de las directivas que nos ofrece la especificación JSP, es necesario comentar una serie de reglas generales que podemos aplicar a todas las etiquetas JSP. Estas reglas son las siguientes:

- Las etiquetas tienen una etiqueta de inicio con atributos opcionales, un cuerpo opcional, y una etiqueta de cierre, como se puede ver en la siguiente sintaxis:

```
<etiquetaJSP nombreAtributo="valor atributo">
cuerpo
</etiquetaJSP>
```

También puede aparecer estas etiquetas sólo con una etiqueta de inicio sin cuerpo y con atributos opcionales, como se puede ver en el siguiente esquema:

```
<etiquetaJSP nombreAtributo="valor atributo"/>
```

- Los valores de los atributos de las etiquetas siempre aparecen entre comillas dobles o sencillas. Las cadenas especiales `&apos;` y `&quot;` se pueden utilizar (como se hace en HTML) si las comillas son parte del valor del atributo.
- Un espacio en el cuerpo del contenido (texto) de un documento no es significativo, pero es preservado, es decir, cualquier espacio en blanco en la página JSP se traduce y mantiene durante el proceso de generación del servlet correspondiente.
- El carácter `\` se puede utilizar como carácter de escape en una etiqueta, por ejemplo para utilizar el carácter `%` (que veremos más adelante es un carácter especial que forma parte de la propia etiqueta de las directivas y de varios elementos de scripting) debemos usar la notación `\%`.
- Las URLs utilizadas en las páginas JSP siguen las mismas convenciones que en los servlets. Si se inicia una URL con `/` se supone que es una referencia al contexto actual, es decir, a la

aplicación Web actual. Si la URL no comienza con / se supone que es relativa a la página JSP actual.

Realizadas estas aclaraciones generales volvemos a retomar el tema principal de este apartado.

Las directivas de JSP tiene la siguiente sintaxis general.

```
<%@ nombreDirectiva atributo1="valor" ... atributon="valorn">
```

Pero también es posible utilizar una sintaxis equivalente utilizando la siguiente notación basada en XML:

```
<jsp:directive.nombreDirectiva atributo1="valor1" ...  
atributon="valorn"/>
```

La especificación JavaServer Pages define tres directivas distintas que son page, include y taglib, a continuación en los siguientes apartados vamos a comentar cada una de estas directivas.

## Directiva page

Esta es la directiva más compleja de JSP debido a que permite especificar un gran número de propiedades que afectan a la página actual. Su sintaxis general es la siguiente:

```
<%@ page atributo1="valor1" ... Atributon="valorn"%>
```

Y la sintaxis equivalente en XML es:

```
<jsp:directive.page atributo1="valor1" ... atributon="valorn"/>
```

La directiva page ofrece hasta once atributos que nos permiten indicar al contenedor de páginas JSP una serie de propiedades que presenta la página JSP actual. Estos atributos se pueden especificar todos juntos en una sólo línea de la directiva page, o bien en diferentes líneas utilizando distintas directivas page, pero teniendo en cuenta que un mismo atributo no puede aparecer en distintas directivas page, el único atributo que puede aparecer varias veces es el atributo import, que permite importar paquetes de clases del lenguaje Java.

Teniendo en cuenta lo anterior el siguiente conjunto de directivas page definidos dentro de una página JSP (Código Fuente 74) es correcto, ya que el único atributo que aparece repetido es el atributo import.

```
<%@ page info="Conjunto válido de directivas"%>  
<%@ page language="java" import="java.io.*"%>  
<%@ page import="java.net.*, java.util.*" %>  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
<html>  
<head>  
    <title>Directivas</title>  
</head>  
<body>  
<%out.println("Hola Mundo");%>  
</body>  
</html>
```

Código Fuente 74

Y la siguiente página JSP (Código Fuente 75) es completamente equivalente a la anterior, ya que lo único que se ha hecho es agrupar las tres directivas page en una única directiva.

```
<%@ page info="Conjunto válido de directivas" language="java"
import="java.net.*, java.util.*, java.io.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Directivas</title>
</head>
<body>
<%out.println("Hola Mundo");%>
</body>
</html>
```

Código Fuente 75

Sin embargo la siguiente página JSP (Código Fuente 76) generará un error ya que el atributo session aparece definido dos veces.

```
<%@ page info="Conjunto no válido de directivas" language="java"
session="false" buffer="16k" autoFlush="false" session="false"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Directivas</title>
</head>
<body>
<%out.println("Hola Mundo");%>
</body>
</html>
```

Código Fuente 76

La siguiente página JSP (Código Fuente 77) también es incorrecta, ya que el atributo info se encuentra repetido.

```
<%@ page info="Conjunto no válido de directivas" language="java"
session="true" buffer="16k" autoFlush="false"%>
<%@ page info="Esto produce un error" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Directivas</title>
</head>
<body>
<%out.println("Hola Mundo");%>
</body>
</html>
```

Código Fuente 77



Sin embargo se debe hacer una aclaración, si definimos en una misma directiva con un atributo repetido, el servidor Jakarta Tomcat no generará ningún error, sino que mantendrá el atributo con el último valor asignado. Aunque en este caso no se genere un error se recomienda no utilizar esta técnica, ya que puede ser confusa y además otros contenedores de páginas JSP se pueden comportar de forma distinta generando un error.

Los atributos no reconocidos también generarán un error, que se dará a la hora de traducir la página JSP al servlet correspondiente. En la Figura 50 se muestra el error que se produciría al utilizar un atributo incorrecto en la directiva page.

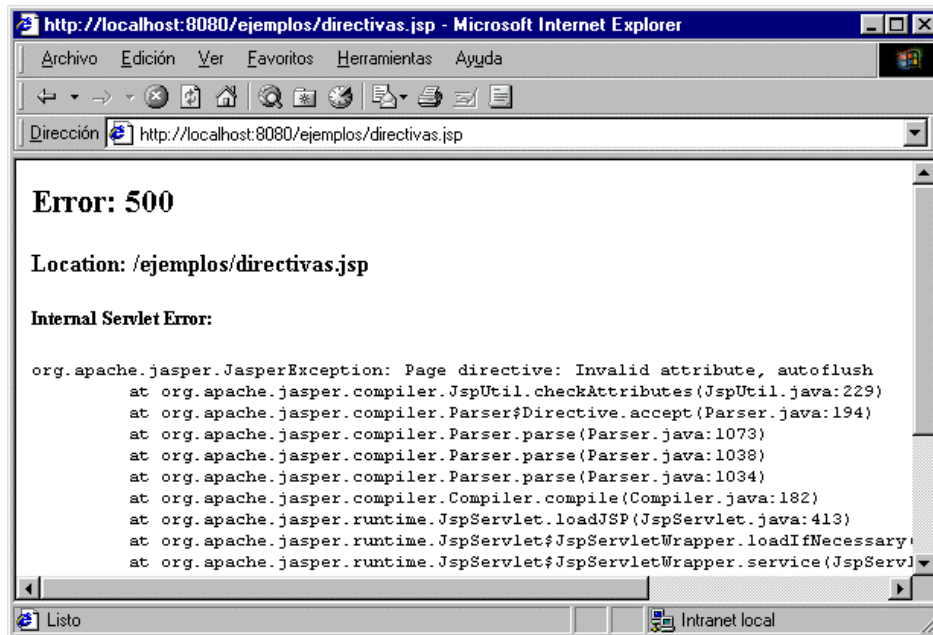


Figura 50. Error generado por un atributo incorrecto

Del error anterior se deduce que en los atributos de la directiva page se distingue entre mayúsculas y minúsculas, esta característica es aplicable a todas las etiquetas de la especificación JavaServer Pages.

A continuación vamos a comentar cada uno de los atributos que posee la directiva buffer, indicando su finalidad y que valores pueden tomar. Ninguno de los atributos de la directiva page es obligatorio, incluso si lo deseamos no es necesario utilizar la directiva page, y si no utilizamos la directiva page el contenedor de páginas JSP tratará a la página JSP con los valores por defecto de los atributos de la directiva page.

## Atributo info

El primero de los atributos que vamos a tratar es el atributo info. El atributo info permite al autor de la página JSP añadir una cadena de documentación a la página para indicar que funcionalidad posee la misma. Por lo tanto el valor del atributo info será una cadena de texto. No existen restricciones en la longitud de la cadena de texto a signada al atributo info. Esta cadena es útil a nivel de documentación, normalmente se utiliza también para indicar información acerca del autor y del copyright. Por defecto el valor del atributo info es una cadena vacía.

En el Código Fuente 78 se muestra un ejemplo de utilización del atributo info de la directiva page.

```

<%@ page info="Ejemplo de directivas, Autor: Angel Esteban"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Ejemplo de atributo info
</body>
</html>

```

Código Fuente 78

Si acudimos al fichero de código fuente que se ha generado del resultado de la traducción de la página JSP en servlet (recordamos al lector que este fichero lo puede encontrar en el directorio `c:\jakarta-yomcat\work\localhost_8080%2FnombreAplicacion`), podemos comprobar que el valor del atributo `info` se devuelve mediante el método `getServletInfo()`. En el Código Fuente 79 se muestra de forma completa el servlet que se ha generado, y la parte que aparece destacada es la que se corresponde con la traducción de la directiva `page` de la página JSP del ejemplo.

```

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fdirectivas_0002ejspdirectivas_jsp_8 extends HttpJspBase {

    // begin [file="C:\\directivas.jsp";from=(0,0);to=(0,61)]
    public String getServletInfo() {
        return "Ejemplo de directivas, Autor: Angel Esteban";
    }
    // end

    static {
    }
    public _0002fdirectivas_0002ejspdirectivas_jsp_8( ) {
    }

    private static boolean _jspx_initd = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;

```

```

JspWriter out = null;
Object page = this;
String _value = null;
try {

    if (_jspx_inited == false) {
        _jspx_init();
        _jspx_inited = true;
    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        "", true, 8192, true);

    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();

    // HTML // begin [file="C:\\directivas.jsp";from=(0,61);to=(10,0)]
    out.write("\r\n<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\">\r\n<html>\r\n<head>\r\n<title>Directivas</title>\r\n</head>\r
\n<body>\r\nEjemplo de atributo info\r\n</body>\r\n</html>\r\n");
    // end

} catch (Exception ex) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(ex);
} finally {
    out.flush();
    _jspxFactory.releasePageContext(pageContext);
}
}

```

Código Fuente 79

## Atributo language

El atributo language indica el lenguaje de script que se va a utilizar en todos los elementos de la página. Todos los contenedores de páginas JSP deben soportar como lenguaje de script el lenguaje Java, y es el lenguaje por defecto si no se indica otro. En el momento actual no existen más lenguajes de script disponibles para las páginas JSP, por lo tanto actualmente es igual indicar el atributo language u omitirlo.

En el Código Fuente 80 se muestra un ejemplo de utilización del atributo language.

```

<%@ page info="Ejemplo de directivas, Autor: Angel Esteban" language="java"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Ejemplo de atributo info y language
</body>
</html>

```

Código Fuente 80

## Atributo contentType

El atributo contentType se utiliza para indicar el tipo MIME que le corresponde a la respuesta que genera la página JSP. Esta directiva tiene una finalidad similar al método `setContentType()` del interfaz `javax.servlet.ServletResponse` utilizado en los servlets. Los tipos MIME más comunes para las páginas JSP son `text/html`, `text/xml` y `text/plain`. El tipo MIME por defecto es `text/html`, y es el valor por defecto del atributo contentType.

En el atributo contentType podemos indicar también el conjunto de caracteres que deseamos utilizar en la respuesta generada por la página JSP. El Código Fuente 81 muestra un ejemplo de utilización de este atributo.

```
<%@ page info="Ejemplo de directivas, Autor: Angel Esteban" language="java"
contentType="text/html; charset=ISO-8859-1"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Ejemplos de atributos de la directiva page
</body>
</html>
```

Código Fuente 81

Si acudimos al servlet equivalente a la página JSP anterior encontraremos una sentencia que realiza la función especificada en el atributo contentType. Esta sentencia se muestra resaltada en el siguiente fragmento de código del servlet.

```
__jspxFactory = JspFactory.getDefaultFactory();
response.setContentType("text/html; charset=ISO-8859-1");
pageContext = __jspxFactory.getPageContext(this, request, response,
    "", true, 8192, true);

application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();

// HTML // begin [file="C:\\directivas.jsp";from=(1,44);to=(10,0)]
out.write("\r\n<html>\r\n<head>\r\n\t<title>Directivas</title>\r\n</head>\r\n<body>\r\nEjemplos de atributos de la directiva page\r\n</body>\r\n</html>\r\n");
// end
```

Código Fuente 82

## Atributo extends

Otro atributo de la directiva page es el atributo extends. La función de este atributo es similar a la cláusula extends de la definición de una clase, es decir, nos permite indicar la superclase de la que hereda la página JSP actual. No existe un valor por defecto para este atributo. Este atributo no se suele utilizar en las páginas JSP.

Un ejemplo de utilización de este atributo lo podemos observar en el Código Fuente 83.

```
<%@ page info="Ejemplo de directivas, Autor: Angel Esteban" language="java"
contentType="text/html; charset=ISO-8859-1" extends="com.paquete.MiClaseJSP"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Ejemplos de atributos de la directiva page
</body>
</html>
```

Código Fuente 83

Este atributo se traducirá de la siguiente forma dentro del servlet equivalente (Código Fuente 84).

```
public class _0002fdirectivas_0002ejspdirectivas_jsp_4 extends
com.paquete.MiClaseJSP {
```

Código Fuente 84

## Atributo import

Sin embargo el atributo import de la directiva page si que se suele utilizar de forma muy común, tiene la misma función que la sentencia import de Java, es decir, nos permite importar clases y paquetes enteros para poder utilizarlos dentro de nuestra página JSP. En el ejemplo del Código Fuente 85 se muestra una página JSP que importa el paquete java.util para poder utilizar la clase Date y así obtener la fecha y hora actual.

```
<%@ page import="java.util.*"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Fecha y hora actuales: <%=new Date()%>
</body>
</html>
```

Código Fuente 85

El atributo import se traduce en el servlet correspondiente por sentencias import, así las sentencias import que aparecerán en el servlet equivalente al ejemplo anterior son las del siguiente fragmento de código (Código Fuente 86).

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

```
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;
```

Código Fuente 86

Este es el único atributo que se puede repetir en una página JSP. Si queremos importar varios paquetes o clases, es decir, utilizar varios atributos import, podemos hacerlo de dos formas distintas, a través de varias directivas page o bien separando con comas cada valor del atributo import, de esta forma la página JSP mostrada en el Código Fuente 87 es equivalente a la página JSP mostrada en el Código Fuente 88.

```
<%@ page import="java.util.*"%>
<%@ page import="java.net.*"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Fecha y hora actuales: <%=new Date()%><br>
<%URL url=new URL("http://mi.servidor.com");%>
Procolo utilizado:<%=url.getProtocol()%>
</body>
</html>
```

Código Fuente 87

```
<%@ page import="java.util.*, java.net.*"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Fecha y hora actuales: <%=new Date()%><br>
<%URL url=new URL("http://mi.servidor.com");%>
Procolo utilizado:<%=url.getProtocol()%>
</body>
</html>
```

Código Fuente 88

## Atributo session

El siguiente atributo a tratar es el atributo session, utilizaremos este atributo para indicar si queremos que la página JSP participe o no en la sesión actual del usuario, es decir, decidimos si la página debe utilizar o no el estado de sesión que veíamos en los servlets. Gracias al atributo session y al objeto integrado session, el mantenimiento y utilización de sesiones dentro de las páginas JSP es mucho más sencillo que en los servlets.

Este atributo puede tener los valores verdadero o falso (true/false) para indicar si la página va a pertenecer a una sesión o no, por defecto el atributo session tiene el valor true, por lo tanto de forma predeterminada una página JSP va a pertenecer a una sesión.

Cuando tratemos el objeto integrado de JSP session, en el capítulo correspondiente, veremos que una página JSP puede utilizar este objeto sólo si pertenece a una sesión. El mecanismo que siguen las páginas JSP para mantener sesiones es el mismo que el visto para los servlets, utilizan también la cookie JSPSESSIONID.

En el Código Fuente 89 se muestra una página JSP que utiliza el atributo session para indicar que pertenece a la sesión actual, en el caso de no existir una sesión actual se creará de forma automática. En el ejemplo también se utiliza el objeto integrado session, que es una instancia del interfaz javax.servlet.http.HttpSession, para mostrar el identificador de la sesión actual. El resultado de la ejecución de la página anterior se puede apreciar en la Figura 51.

```
<%@ page session="true"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
El identificador de la sesion es: <%=session.getId()%>
</body>
</html>
```

Código Fuente 89

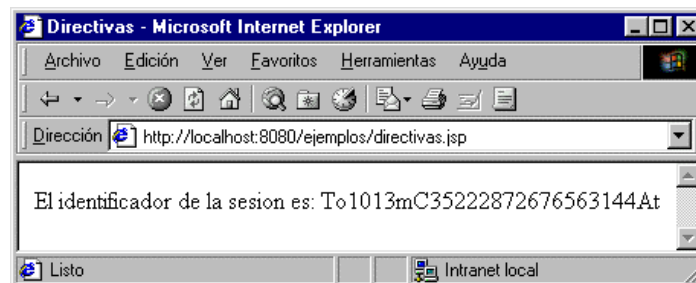


Figura 51. Accediendo al objeto session

El siguiente ejemplo (Código Fuente 90) generaría un error como el de la Figura 52, ya que el atributo session tiene el valor false y se intenta acceder al objeto session.

```
<%@ page session="false"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
El identificador de la sesion es: <%=session.getId()%>
</body>
</html>
```

Código Fuente 90

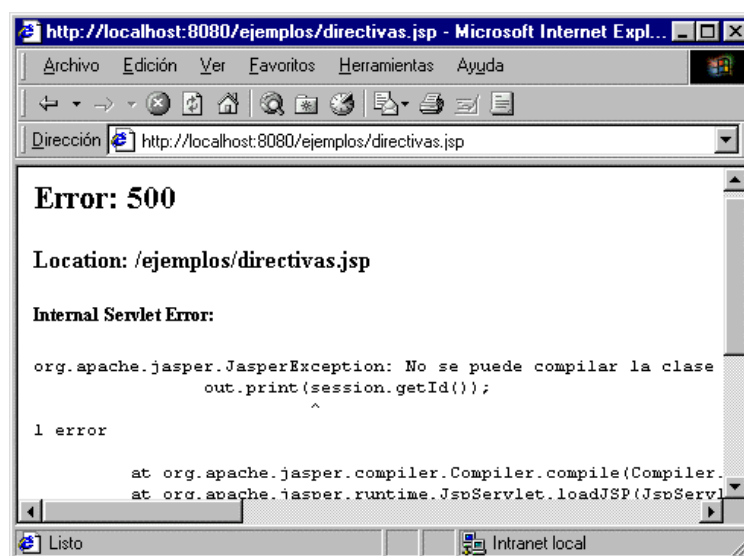


Figura 52. Error generado al intentar utilizar el objeto session

## Atributo buffer

Otro atributo a tener en cuenta de la directiva page es el atributo buffer. Este atributo controla si en la respuesta que se envía al cliente se va utilizar un búfer intermedio. Si no queremos utilizar un búfer intermedio y enviar al contenido de forma directa al navegador le asignaremos el valor none a este atributo. En caso contrario si deseamos que antes de enviar la salida al navegador pase por el búfer intermedio, deberemos indicar el tamaño del búfer en kilobytes.

En el Código Fuente 91 se muestra una página JSP que utiliza un búfer de 12 kilobytes. Para verificar el tamaño del búfer utilizado utilizamos el método `getBufferSize()` del objeto integrado `response`.

```
<%@ page buffer="12kb"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Tamaño del búfer: <%=response.getBufferSize()/1024%>KB
</body>
</html>
```

Código Fuente 91

Y en el Código Fuente 92 se muestra una página JSP que no utiliza ningún búfer y que envía la respuesta generada de forma directa al usuario.

```
<%@ page buffer="none"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
```



```
Tamaño del búfer: <%=response.getBufferSize()/1024%>KB
</body>
</html>
```

Código Fuente 92

Por defecto el valor del atributo buffer es 8kb, es decir, por defecto una página JSP tiene un búfer intermedio de 8 KB. Cuando el búfer se llena se debe enviar su contenido al navegador para dejarlo libre, este mecanismo de liberación del espacio del búfer lo trataremos en el siguiente apartado, ya que se encuentra regulado por el atributo autoFlush de la directiva page.

Es una buena práctica utilizar el búfer en la generación de la respuesta de la página JSP, ya que en un momento dado podemos anular toda la respuesta generada y generar otra distinta o bien redirigir la ejecución de la página a otra página distinta. Veremos que también es conveniente utilizar el búfer para poder hacer un uso satisfactorio del atributo errorPage de la directiva page, que trataremos en el apartado correspondiente.

## Atributo autoFlush

Este atributo se encuentra relacionado con el atributo búfer, ya que indica el comportamiento que se debe seguir cuando el búfer intermedio se ha llenado. Si al atributo autoFlush se le asigna el valor true (que es su valor por defecto), cuando se llene el búfer de forma automática se liberará su espacio dejándolo vacío y enviando sus contenidos al navegador que realizó la petición de la página JSP, y de esta forma la página se seguirá procesando volviendo a llenar el búfer y volviéndose a vaciar si es necesario.

Una vez que se ha vaciado el búfer y ha sido enviado su contenido al navegador que realizó la petición de la página JSP, no es posible redirigir la ejecución de la página JSP a otra página, servlet o recurso.

Si el atributo tiene el valor false, el contenedor de páginas JSP no vaciará el búfer de forma automática, en este caso se lanzará una excepción que afectará al resultado de la página JSP ya que mostrará este error en el navegador.

Si el atributo buffer tiene el valor none, es decir, no se está utilizando el búfer, no podremos establecer el valor del atributo autoFlush al valor false. La más normal es establecer el valor del atributo autoFlush a verdadero, para que cada vez que se llene el búfer se vacíe y se envíe su contenido a la respuesta que se devolverá al usuario.

La siguiente página JSP (Código Fuente 93) utiliza el atributo autoFlush para indicar que se vaya vaciando el búfer de manera automática, aunque en realidad podríamos no haber utilizado la directiva page, ya que el valor por defecto de este atributo es true.

```
<%@ page autoFlush="true"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
    El búfer se vacía de forma automática
</body>
</html>
```

Código Fuente 93

La siguiente página JSP (Código Fuente 94) producirá un error, que se puede ver en la Figura 53, ya que se produce un conflicto entre los valores de los atributos buffer y autoFlush.

```
<%@ page autoFlush="false" buffer="none"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Genera un error
</body>
</html>
```

Código Fuente 94

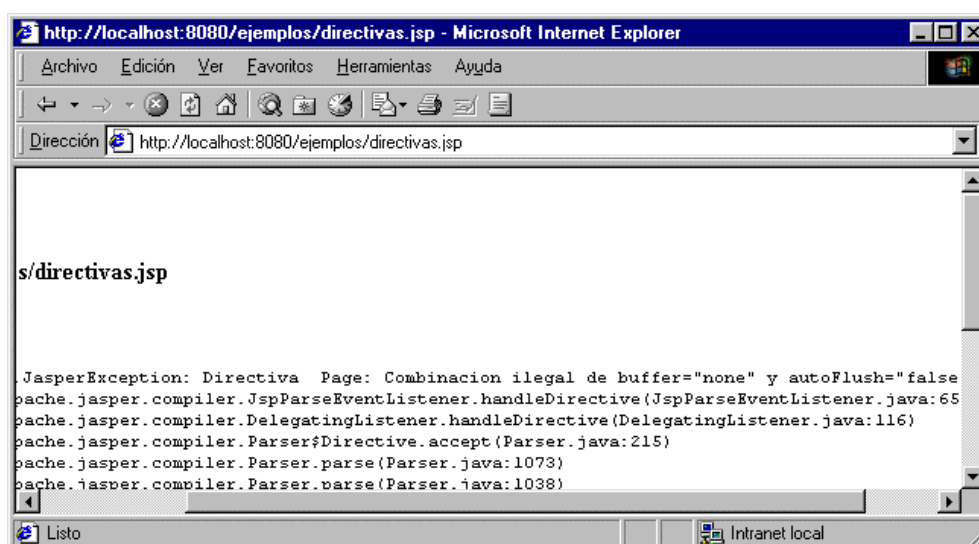


Figura 53. Conflicto entre buffer y autoFlush

## Atributo isThreadSafe

El atributo `isThreadSafe` se utiliza para indicar si la página JSP (una vez convertida en servlet), va a ser capaz de atender peticiones múltiples de manera simultánea, es decir, si es posible que se ejecute en varios hilos de ejecución.

Cuando el atributo `isThreadSafe` tiene el valor `false`, el contenedor de páginas JSP procesará de forma secuencial las distintas peticiones que se realicen a la página JSP correspondiente, teniendo en cuenta el orden en el que se recibieron las peticiones cada una de las peticiones debe esperar a que termine la que se encuentra delante de ella, esto produce un rendimiento menor de las páginas JSP.

Sin embargo cuando el atributo `isThreadSafe` posee el valor `true`, se creará una nuevo hilo de ejecución por cada petición que reciba la página JSP, de esta forma se podrán tratar distintas peticiones al mismo tiempo de manera simultánea, cada una en su hilo de ejecución correspondiente.

Siempre que sea posible el atributo `isThreadSafe` debe tener el valor `true` para que no se afecte al rendimiento de la ejecución de las páginas JSP. Al establecer el valor a verdadero de este atributo deberemos tratar a través del código de la página JSP los posibles problemas derivados del acceso

concurrente al permitir la ejecución simultánea de varios hilos de ejecución, para ellos utilizaremos las herramientas de sincronización que nos ofrece el lenguaje Java. Estos problemas de sincronización los volveremos a tratar más adelante cuando veamos el objeto integrado de las páginas JSP application.

El lector puede suponer que el comportamiento de este atributo es muy similar al interfaz `javax.servlet.SingleThreadModel`, cuando indicábamos que un servlet implementara este interfaz para que no se produjeran ejecuciones simultáneas del mismo, es equivalente a asignar al atributo `isThreadSafe` el valor `false`.

La siguiente página JSP (Código Fuente 95) indica que su ejecución en múltiples hilos de ejecución es segura.

```
<%@ page isThreadSafe="true"%>
<html>
<head>
  <title>Directivas</title>
</head>
<body>
Página que se ejecutará en múltiples hilos de ejecución
</body>
</html>
```

Código Fuente 95

Y la siguiente página JSP (Código Fuente 96) indica lo contrario.

```
<%@ page isThreadSafe="false"%>
<html>
<head>
  <title>Directivas</title>
</head>
<body>
Página que se ejecutará en un único hilo de ejecución
</body>
</html>
```

Código Fuente 96

Si accedemos al servlet que se genera a partir de la página anterior veremos que implementa el interfaz `javax.servlet.SingleThreadModel`, tal como se puede observar en el siguiente fragmento de código (Código Fuente 97).

```
public class _0002fdirectivas_0002ejspdirectivas_jsp_6 extends HttpJspBase
    implements
    SingleThreadModel
{
```

Código Fuente 97

## Atributo errorPage

El atributo `errorPage` es utilizado para especificar la página de error en el caso de que se produzca una excepción no tratada en la página JSP actual. A este atributo se le asigna la URL de la página que va a tratar los errores no atrapados que se produzcan en la página JSP. La URL indicada puede ser absoluta o relativa a la ruta correspondiente a la página actual.

Para que el contenedor de páginas JSP pueda realizar la redirección a la página de error cuando se produce una excepción, previamente no se ha tenido que enviar ningún contenido al cliente, es decir, en ningún momento se ha vaciado el búfer de almacenamiento, en caso contrario se producirá un error de redirección, ya que lo que hace el contenedor de páginas JSP es redirigir la ejecución del página JSP a la URL que se corresponde con la página de error indicada.

El valor por defecto de este atributo es una cadena vacía, es decir, de forma predeterminada no se utiliza ninguna página de error, sino que se muestran los mensajes de error que genera de forma automática el contenedor de páginas JSP.

En el Código Fuente 98 se puede ver la utilización de este atributo, en este caso se ha utilizado una URL relativa que hace referencia a la página de error llamada `PAGINAERROR.JSP`.

```
<%@ page errorPage="paginaError.jsp"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Página que llama a una página de error si se produce
una excepción no tratada.
</body>
</html>
```

Código Fuente 98

## Atributo isErrorPage

Este es el último de los atributos pertenecientes a la directiva `page`, y se encuentra muy relacionado con el anterior. Este atributo permite indicar si la página JSP actual es una página de tratamiento de errores o no. A este atributo le asignaremos los valores verdadero o falso. Si este atributo tiene el valor `true`, la página JSP actual será una página de error, por lo que podrá ser referenciada desde el atributo `errorPage` de otra página JSP distinta. Si este atributo tiene el valor `false` (su valor por defecto), la página se tratará como una página cualquiera.

Una página de error puede acceder al objeto integrado `exception`, instancia de la clase `java.lang.Throwable`, para obtener información acerca del error que se ha producido en la página que le ha invocado.

En el Código Fuente 99 se puede observar una página JSP para el tratamiento de errores, como se puede comprobar hace uso del objeto integrado `exception`, este objeto junto con el tratamiento de errores en JSP lo veremos en detalle en el capítulo correspondiente.

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

```

<head>
    <title>Página de error</title>
</head>

<body>
<h1>Se ha producido una excepción</h1>
<b>ERROR:</b> <%=exception.toString()%><br>
<b>MENSAJE:</b> <%=exception.getMessage()%><br>
<b>VOLCADO DE PILA:</b>
<%=StringWriter sSalida=new StringWriter();
PrintWriter salida=new PrintWriter(sSalida);
exception.printStackTrace(salida);%>
<%=sSalida%>
</body>
</html>

```

Código Fuente 99

La página JSP del Código Fuente 100 generará una excepción ya que realiza una división por cero, y por lo tanto al ejecutarse el tratamiento del error lo realizará la página de error indicada en el atributo `errorPage`, que no es otra que la página del ejemplo anterior que mostrará el resultado de la Figura 54.

```

<%@ page errorPage="paginaError.jsp"%>
<html>
<head>
    <title>Directivas</title>
</head>
<body>
Página que genera un error.
<%=Integer valor=new Integer(3/0);%>
</body>
</html>

```

Código Fuente 100

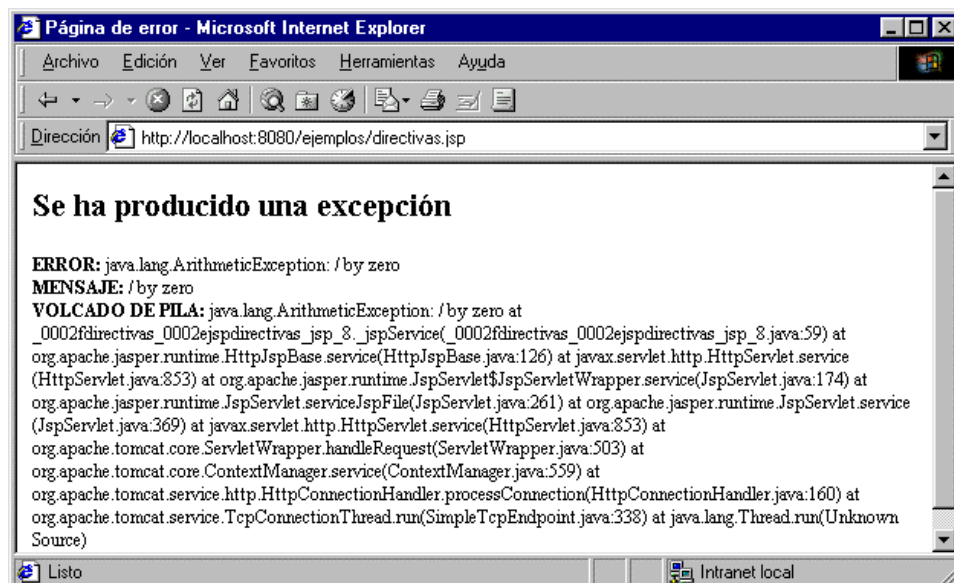


Figura 54. Página de error personalizada

Una vez finalizada la directiva page se ofrece en la Tabla 13 un resumen de todos sus atributos, en el siguiente apartado trataremos una directiva mucho más sencilla, la directiva include.

Atributo	Descripción	Valor por defecto	Ejemplo
info	Descripción de la página.	Cadena vacía.	Info="página de ejemplo"
language	Lenguaje de script utilizado.	java	language="java"
contentType	Tipo de contenido devuelto por la página.	text/html	contentType="text/html"
extends	Superclase de la que se hereda.	Ninguno	extends="paquete.MiClase"
import	Importan clases y paquetes para utilizar en la página.	Ninguno	import="java.io"
session	Indica si la página pertenece a una sesión.	true	session="true"
buffer	Indica si se utiliza un espacio de almacenamiento intermedio antes de enviar la respuesta al cliente.	8kb	buffer="12kb"
autoFlush	Indica el comportamiento que se sigue cuando se ha llenado el búfer.	true	autoFlush="false"
isThreadSafe	Indica si la página se puede ejecutar el múltiples hilos de ejecución.	true	isThreadSafe="true"
errorPage	Indica la página de error que se debe utilizar si se produce una excepción.	Ninguno	errorPage="dir/paginaError.jsp"
isErrorPage	Indica si la página actual es una página de error.	false	isErrorPage="true"

Tabla 13. Resumen de los atributos de la directiva page

## Directiva include

Esta directiva permite a los autores de páginas incluir el contenido de un recurso dentro del contenido generado por una página JSP. El recurso a incluir dentro de la página JSP se indica mediante una URL relativa o absoluta, pero que debe hacer referencia al servidor en el que se ejecutan las páginas JSP. La sintaxis general de esta directiva será la siguiente.

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

Y su sintaxis equivalente en XML es:

```
<jsp:directiva.include file="URLlocal"/>
```

Como se puede comprobar esta directiva posee únicamente el atributo file, en el que se indica la URL del recurso que se desea incluir en la respuesta de la página JSP.

No existen restricciones en el número de directivas include que pueden utilizarse en una misma página JSP. En el Código Fuente 101 se muestra una página JSP que utiliza dos directivas include para incluir en su respuesta dos páginas HTML, una de ellas realiza la función de cabecera y otra de pie de la página.

```
<html>
<head>
    <title>Directivas</title>
</head>
<body>
<%@ include file="cabecera.html"%>
<br>
Esto es la página JSP
<br>
<%@ include file="pie.html"%>
</body>
</html>
```

Código Fuente 101

Si el código de la página CABECERA.HTML es el Código Fuente 102 y el de la página PIE.HTML es el Código Fuente 103, el resultado de la ejecución de la página JSP será como el resultado de la .

```
<h1>Esto es la cabecera</h1><hr>
```

Código Fuente 102

```
<hr><small><i>Esto es el pie</i></small>
```

Código Fuente 103

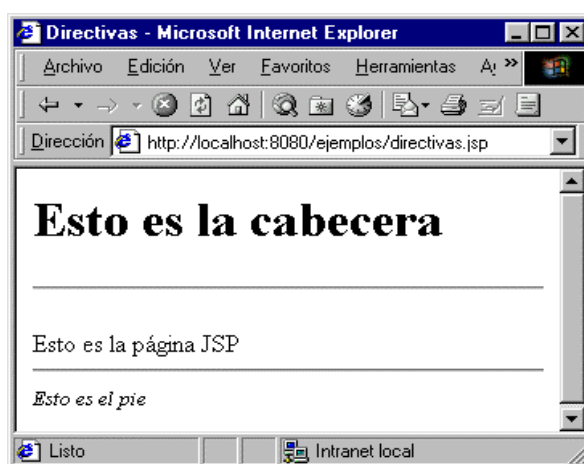


Figura 55. Utilizando la directiva include

Como se puede ver la inclusión del contenido del recurso se realiza en el lugar en el que se encuentra la directiva include correspondiente.

Internamente, dentro del proceso de traducción de la página JSP en el servlet correspondiente, la directiva include tiene el efecto de sustituir los contenidos de la página incluida antes de compilar el servlet. Los contenidos del recurso incluido pueden ser elementos estáticos como HTML o bien otra página JSP con su código dinámico correspondiente.

En el Código Fuente 104 se puede observar el servlet que se ha generado a la hora de traducir la página JSP que utiliza las dos directivas include para incluir una cabecera y un pie. El código que aparece resaltado es el que se corresponde con las directivas include.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fdirectivas_0002ejspdirectivas_jsp_10 extends HttpJspBase {

    static {
    }
    public _0002fdirectivas_0002ejspdirectivas_jsp_10( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);
```



```

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        // HTML // begin [file="C:\\directivas.jsp";from=(0,0);to=(5,0)]
out.write("<html>\r\n<head>\r\n\t<title>Directivas</title>\r\n</head>\r\n<body>\r\n"
");
        // end
        // HTML // begin [file="C:\\cabecera.html";from=(0,0);to=(1,0)]
        out.write("<h1>Esto es la cabecera</h1><hr>\r\n");
        // end
        // HTML // begin [file="C:\\directivas.jsp";from=(5,34);to=(9,0)]
        out.write("\r\n<br>\r\nEsto es la p gina JSP\r\n<br>\r\n");
        // end
        // HTML // begin [file="C:\\pie.html";from=(0,0);to=(1,0)]
        out.write("<hr><small><i>Esto es el pie</i></small>\r\n");
        // end
        // HTML // begin [file="C:\\directivas.jsp";from=(9,29);to=(13,0)]
        out.write("\r\n</body>\r\n</html>\r\n\r\n");
        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

C digo Fuente 104

Como se puede comprobar el servlet generado trata los ficheros incluidos como si formaran parte de la propia p gina JSP.

Es posible hacer referencia en la p gina principal a variables que se encuentren definidas en la p gina incluida, y viceversa.

Como ya comentamos en el cap tulo anterior, cuando se modifica una p gina JSP, el contenedor de p ginas JSP de forma autom tica detecta esta situaci n y reconstruye el servlet equivalente. Pero este mecanismo s lo se aplica a la propia p gina JSP, no a las p ginas o recursos que incluya utilizando la directiva `include`, el contenedor de p ginas JSP no es capaz de mantener la pista a las distintas dependencias de la p ginas JSP respecto a otros ficheros que se han incluido a trav s de la directiva `include`. De esta forma las modificaciones que se realicen en los ficheros que se han incluido no se reflejar n en la p gina JSP, ya que no se generar  un nuevo servlet, porque las modificaciones han sido realizadas en otros ficheros que no son la propia p gina JSP.

Por lo tanto es necesario forzar la reconstrucci n del servlet, lo m s sencillo es modificar manualmente al fecha de modificaci n de la p gina JSP que realiza las distintas directivas `include`, de esta forma se volver  a incluir los contenidos de los ficheros incluidos en el servlet resultante.

Cuando tratemos las acciones (otro tipo de elemento de la especificaci n JavaServer Pages) veremos que existe tambi n una acci n `include` pero que funciona con una serie de matices diferentes que trataremos en su momento.

Para finalizar con este apartado dedicado a la directiva include, vamos a mostrar una nueva página JSP de ejemplo (Código Fuente 105) que además de utilizar dos directivas include para incluir un contenido estático en la página actual, utiliza una directiva include para incluir el contenido dinámico de una página JSP que muestra información referente a la sesión actual.

```
<html>
<head>
  <title>Directivas</title>
</head>
<body>
<%@ include file="cabecera.html"%>
<br>
Esto es la página JSP
<br>
INFORMACIÓN ACERCA DE LA SESIÓN:<br>
<%@ include file="sesion.jsp"%>
<%@ include file="pie.html"%>
</body>
</html>
```

Código Fuente 105

Si el código de la página JSP SESION.JSP es el mostrado en el Código Fuente 106, el resultado de la ejecución de la página JSP que utiliza las directivas include será como el mostrado en la Figura 56.

```
Creación de la sesión:<%=new java.util.Date(session.getCreationTime())%><br>
Último acceso a la sesión:<%=session.getLastAccessedTime()%><br>
ID de la sesión:<%=session.getId()%><br>
Mantenimiento de la sesión inactiva:<%=session.getMaxInactiveInterval()/60%>
minutos<br>
```

Código Fuente 106



Figura 56. Incluyendo contenidos dinámicos

Y en este caso el servlet resultante contiene las sentencias de la página JSP que se ha incluido, en el Código Fuente 107 se pueden ver resaltadas estas sentencias.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fdirectivas_0002ejspdirectivas_jsp_11 extends HttpJspBase {

    static {
    }
    public _0002fdirectivas_0002ejspdirectivas_jsp_11( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\directivas.jsp";from=(0,0);to=(5,0)]

out.write("<html>\r\n<head>\r\n\t<title>Directivas</title>\r\n</head>\r\n<body>\r\n
");
            // end
```

```

// HTML // begin [file="C:\\cabecera.html";from=(0,0);to=(1,0)]
    out.write("<h1>Esto es la cabecera</h1><hr>\r\n");
// end
// HTML // begin [file="C:\\directivas.jsp";from=(5,34);to=(9,0)]
    out.write("\r\n<br>\r\nEsto es la página JSP\r\n<br>\r\n");
// end
// HTML // begin [file="C:\\sesion.jsp";from=(0,0);to=(2,22)]
    out.write("<html>\r\n<body>\r\nCreaci3n de la sesi3n:");
// end
// begin [file="C:\\sesion.jsp";from=(2,25);to=(2,70)]
    out.print(new java.util.Date(session.getCreationTime()));
// end
// HTML // begin [file="C:\\sesion.jsp";from=(2,72);to=(3,26)]
    out.write("<br>\r\n3ltimo acceso a la sesi3n:");
// end
// begin [file="C:\\sesion.jsp";from=(3,29);to=(3,58)]
    out.print(session.getLastAccessedTime());
// end
// HTML // begin [file="C:\\sesion.jsp";from=(3,60);to=(4,16)]
    out.write("<br>\r\nID de la sesi3n:");
// end
// begin [file="C:\\sesion.jsp";from=(4,19);to=(4,34)]
    out.print(session.getId());
// end
// HTML // begin [file="C:\\sesion.jsp";from=(4,36);to=(5,36)]
    out.write("<br>\r\nMantenimiento de la sesi3n inactiva:");
// end
// begin [file="C:\\sesion.jsp";from=(5,39);to=(5,74)]
    out.print(session.getMaxInactiveInterval()/60);
// end
// HTML // begin [file="C:\\sesion.jsp";from=(5,76);to=(7,7)]
    out.write(" minutos<br>\r\n</body>\r\n</html>");
// end
// HTML // begin [file="C:\\directivas.jsp";from=(9,31);to=(10,0)]
    out.write("\r\n");
// end
// HTML // begin [file="C:\\pie.html";from=(0,0);to=(1,0)]
    out.write("<hr><small><i>Esto es el pie</i></small>\r\n");
// end
// HTML // begin [file="C:\\directivas.jsp";from=(10,29);to=(14,0)]
    out.write("\r\n</body>\r\n</html>\r\n\r\n");
// end

} catch (Exception ex) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(ex);
} finally {
    out.flush();
    _jspxFactory.releasePageContext(pageContext);
}
}

```

Código Fuente 107

En el siguiente apartado trataremos la tercera directiva ofrecida JSP, se trata de la directiva taglib.

## La directiva taglib

Esta directiva es utilizada para indicar al contenedor de páginas JSP que la página JSP actual utiliza una librería de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas

personalizadas que extienden la funcionalidad de una página JSP. Una vez que se ha utilizado esta directiva para indicar la librería de etiquetas que se van a utilizar, todas las etiquetas personalizadas definidas en la librería están a nuestra disposición para hacer uso de ellas en nuestra página JSP actual.

La sintaxis de esta directiva es la siguiente:

```
<%@ taglib uri="URLLibreria" prefix="prefijoEtiquetas"%>
```

Y como el resto de las directivas tiene su equivalente en XML:

```
<jsp:directive.taglib uri="URLLibreria" prefix="prefijo"/>
```

Esta directiva posee dos atributos, el primero de ellos, llamado uri, se utiliza para indicar la localización del descriptor de la librería de etiquetas (TLD, Tag Library Descriptor), y el segundo atributo, llamado prefix indica el prefijo que se va a utilizar para hacer referencia a las etiquetas personalizadas.

El descriptor de la librería de etiquetas es un fichero especial que indica las clases que implementan y forman una librería, también indica el nombre de las etiquetas y los atributos que poseen.

La siguiente página JSP (Código Fuente 108) utiliza dos librerías de etiquetas, que se encuentran definidas en los descriptors de etiquetas LIBRERIAETIQUETAS.TLD Y UTILIDADES.TLD, a la primera librería se le asigna el prefijo libreria y a la segunda el prefijo utils. Dentro de estas librerías se utilizan dos etiquetas, la etiqueta tam para obtener el tamaño de los ficheros que se indican en su atributo uri, y la etiqueta buffer que muestra información relativa al búfer que se ha utilizado y el tanto por ciento que queda libre.

```
<html>
<title>Directivas</title>
<body>
<%@taglib uri="tlds/libreriaEtiquetas.tld" prefix="libreria"%>
<%@taglib uri="tlds/utilidades.tld" prefix="utils"%>
Tamaños de ficheros:<br>
El fichero fich1.jpg tiene el tamaño de <libreria:tam uri="fich1.jpg"/><br>
El fichero fich2.java tiene el tamaño de <libreria:tam uri="fich2.java"/><br>
<br>
Información del búfer:
<br>
<utils:buffer/>
</body>
</html>
```

Código Fuente 108

El resultado de la ejecución de esta página JSP se puede ver en la Figura 57.

Puede que el lector no entienda muy bien el mecanismo de las librerías de etiquetas, pero no debe preocuparse, aquí hemos pretendido mostrar únicamente la directiva que nos permite utilizarlas, más adelante, en el capítulo correspondiente las veremos más en detalle y veremos también como construirlas.

Con este apartado finalizamos este capítulo dedicado a las directivas de JSP, en el siguiente capítulo trataremos los elementos de scripting, es decir, los elementos que podemos encontrar dentro del script de una página JSP, entendiendo por script código escrito en la página JSP.

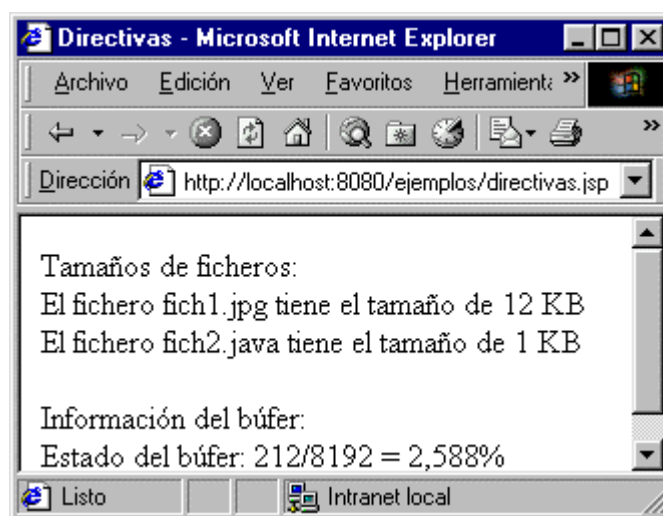


Figura 57. utilizando la directiva taglib

# Páginas JSP: Elementos de scripting

---

## Introducción

En este capítulo vamos a tratar el segundo tipo de elementos dinámicos que nos podemos encontrar en una página JSP. Los elementos de scripting, a su vez se dividen en tres subelementos que son: declaraciones, scriptlets, expresiones y comentarios.

Mientras que las directivas influían en la forma de procesar la página JSP por parte del contenedor de páginas JSP, los elementos de scripting son los que nos permiten incluir código Java en las páginas JSP, los elementos de scripting nos van a permitir declarar objetos, instanciarlos, ejecutar métodos, definirlos, etc., es decir son los que realmente nos permiten realizar la programación de nuestra página JSP.

En este capítulo vamos a tratar los cuatro elementos de scripting, cuya funcionalidad resumimos a continuación:

- **Declaraciones:** son bloques de código Java incluido en la página JSP utilizados para declarar variables y métodos propios dentro de la página JSP. Un bloque de declaración se encuentra entre los delimitadores `<%! %>`.
- **Scriptlets:** un scriptlet es un fragmento de código Java incluido en una página JSP que se ejecutará cuando se realice una petición de la misma. Un scriptlet se encontrará entre los delimitadores `<% %>`, como curiosidad cabe comentar que estos delimitadores son los mismos que utilizan las páginas ASP (Active Server Pages) para delimitar el script de servidor. En un scriptlet podemos encontrar cualquier código Java válida, no debemos olvidar

que las páginas JSP al igual que los servlets pueden utilizar todos los APIs ofrecidos por el lenguaje Java.

- **Expresiones:** una expresión es una notación especial para un scriptlet que devuelve un resultado a la respuesta dada al usuario, la expresión se evalúa y se devuelve como una cadena que se envía al cliente. Una expresión se encuentra entre los delimitadores `<%= %>`, que son también los mismos delimitadores que se utilizan para devolver el valor de expresiones dentro de una página ASP.
- **Comentarios:** estos elementos permiten documentar nuestro código fuente, se encuentran entre los delimitadores `<%-- --%>`. Estos comentarios no serán visibles en el navegador, ya que son comentarios de JSP que serán tratados por el contenedor de páginas JSP, no se deben confundir con los comentarios HTML (`<!-- -->`), que si serán visibles desde el navegador y enviados como tales al usuario.

En los siguientes apartados veremos en detalle cada uno de estos elementos, utilizando para ello distintos ejemplos de páginas JSP.

## Declaraciones

Las declaraciones se utilizan para definir variables (objetos) y métodos específicos de una página JSP, tanto las variables como los métodos declarados se pueden referenciar por otros elementos de script de la misma página JSP. La sintaxis general de una declaración es:

```
<%! Declaración (es) %>
```

Y existe también una sintaxis XML equivalente:

```
<jsp:declaration> declaración (es) </jsp:declaration>
```

En una única etiqueta se pueden incluir varias declaraciones, pero cada declaración debe ser una sentencia declarativa completa en el lenguaje de script correspondiente, que como sabemos el único disponible para las páginas JSP por el momento es el lenguaje Java.

Cuando declaramos una variable dentro de una página JSP, esta variable se transformará en un atributo (variable de instancia) del servlet que se corresponde con la página JSP, de esta forma la página JSP mostrada en el Código Fuente 109, se traducirá en el servlet equivalente mostrado en el Código Fuente 110.

```
<%! private int edad; private String nombre; private String domicilio;%>
<html>
<head>
    <title>Declaraciones</title>
</head>
<body>
<%out.println("Página que declara tres variables");%>
</body>
</html>
```

Código Fuente 109

```
import javax.servlet.*;
```



```

import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fdeclaraciones_0002ejspdeclaraciones_jsp_4 extends HttpJspBase {

    // begin [file="C:\\declaraciones.jsp";from=(0,3);to=(0,70)]
    private int edad; private String nombre; private String domicilio;
    // end

    static {
    }
    public _0002fdeclaraciones_0002ejspdeclaraciones_jsp_4( ) {

    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {

    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\declaraciones.jsp";from=(0,72);to=(6,0)]
out.write("\r\n<html>\r\n<head>\r\n<title>Declaraciones</title>\r\n</head>\r\n<bo
dy>\r\n");
            // end
            // begin [file="C:\\declaraciones.jsp";from=(6,2);to=(6,51)]
            out.println("P gina que declara tres variables");
            // end
            // HTML // begin [file="C:\\declaraciones.jsp";from=(6,53);to=(9,0)]
            out.write("\r\n</body>\r\n</html>\r\n");

```

```

        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

Código Fuente 110

Estas variables se puede referenciar en cualquier elemento de script dentro de la página JSP incluso en aquellos que se encuentren antes de la declaración. En el Código Fuente 111 se puede observar una variación sobre la página JSP actual, que modifica y obtiene el valor de una de las variables declaradas, y en el Código Fuente 112 se muestra un fragmento que se corresponde con el servlet generado a partir de la página JSP.

```

<%! private int edad; private String nombre; private String domicilio;%>
<html>
<head>
    <title>Declaraciones</title>
</head>
<body>
<%edad=25;
out.println("EL valor de edad es:"+edad);%>
</body>
</html>

```

Código Fuente 111

```

// HTML // begin [file="C:\\declaraciones.jsp";from=(0,72);to=(6,0)]
out.write("\r\n<html>\r\n<head>\r\n\t<title>Declaraciones</title>\r\n</head>\r\n<bo
dy>\r\n");
// end
// begin [file="C:\\declaraciones.jsp";from=(6,2);to=(7,41)]
edad=25;
out.println("EL valor de edad es:"+edad);
// end
// HTML // begin [file="C:\\declaraciones.jsp";from=(7,43);to=(10,0)]
out.write("\r\n</body>\r\n</html>\r\n");
// end

```

Código Fuente 112

A la hora declarar y utilizar variables de instancia dentro de las páginas JSP se debe tener en cuenta que varios hilos de ejecución pueden estar accediendo a la misma página JSP de forma simultánea, representando diferentes peticiones de la página JSP. Si un elemento de script modifica el valor de una variable, todas las referencias a la variable contendrán el nuevo valor, incluso las referencias que se encuentren en otros hilos de ejecución.

Si deseamos utilizar una variable cuyo valor debe ser local a una única petición se debe utilizar un scriptlet, ya que las variables declaradas están asociadas con la página misma, no con peticiones individuales, y por lo tanto las variables declaradas se podrán compartir entre distintas peticiones.

Debido a que el contenedor de páginas JSP crea una única instancia del servlet que representa a la página JSP correspondiente, se pueden dar problemas de concurrencia a la hora de producirse los accesos simultáneos sobre las variables declaradas en la página, ya que hemos dicho antes pueden ser accedidas por diferentes hilos de ejecución (uno por cada petición) de forma simultánea.

Para evitar los problemas de accesos simultáneos la solución es utilizar la palabra clave `synchronized`, un problema similar lo vimos en capítulos anteriores con los servlets, aquí el problema es el mismo ya que se trata de sincronizar el acceso a las variables de instancia (atributos) de un servlet que representa a la página JSP. Podríamos rescribir el ejemplo de página JSP anterior para que utilice la palabra reservada `synchronized`, para asegurar que en un momento determinado únicamente hay un hilo de ejecución que está accediendo a la variable definida en la página JSP, cuando este hilo de ejecución haya terminado de acceder a la variable dará paso al siguiente hilo de ejecución, es decir, a la siguiente petición de la página JSP. El nuevo código lo podemos ver en el Código Fuente 113.

```
<%! private Integer edad=new Integer("0");
    private String nombre;
    private String domicilio;%>
<html>
<head>
    <title>Declaraciones</title>
</head>
<body>
<%synchronized (edad){
    edad=new Integer(25);
    out.println("EL valor de edad es:"+edad);
}%>
</body>
</html>
```

Código Fuente 113

Como se puede comprobar en el código anterior, hemos tenido de convertir la variable `edad` de tipo `int` a un objeto de la clase `Integer`, ya que la sentencia `synchronized` sólo se puede utilizar con objetos, no con tipos primitivos.

Además de declarar variables, como es lógico, también podemos realizar la declaración de métodos dentro de nuestras páginas JSP. La sintaxis para definir un método es la misma que para las variables, únicamente debemos indicar la cabecera de la declaración del método y el cuerpo del mismo, es decir, igual que lo hacemos en el lenguaje Java de forma tradicional. Así por ejemplo, el Código Fuente 114 muestra una página que define un método que devuelve el cuadrado del entero que le pasemos por parámetro.

```
<%! public int cuadrado(int num){
    return num*num;
}%>
<html>
<head>
    <title>Declaraciones</title>
</head>
<body>
<%out.println(cuadrado(4));%>
```

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

## Código Fuente 114

Este método podrá ser utilizado en cualquier lugar de los elementos de scripting de la página, al igual que sucedía cuando declarábamos una variable, y la traducción de la declaración de un método en una página JSP, será un método del servlet resultante. En el Código Fuente 115 se muestra el servlet que se obtiene de la página JSP anterior.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fdeclaraciones_0002ejspdeclaraciones_jsp_11 extends HttpJspBase {

    // begin [file="C:\\declaraciones.jsp";from=(0,3);to=(2,1)]
    public int cuadrado(int num){
        return num*num;
    }
    // end

    static {
    }
    public _0002fdeclaraciones_0002ejspdeclaraciones_jsp_11( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
```

```

        pageContext = _jspxFactory.getPageContext(this, request, response,
            "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        // HTML // begin [file="C:\\declaraciones.jsp";from=(2,3);to=(8,0)]
out.write("\r\n<html>\r\n<head>\r\n\t<title>Declaraciones</title>\r\n</head>\r\n<bo
dy>\r\n");
        // end
        // begin [file="C:\\declaraciones.jsp";from=(8,2);to=(8,27)]
            out.println(cuadrado(4));
        // end
        // HTML // begin [file="C:\\declaraciones.jsp";from=(8,29);to=(11,0)]
            out.write("\r\n</body>\r\n</html>\r\n");
        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

Código Fuente 115

Las páginas JSP poseen un ciclo de vida muy similar al que presentaban los servlets, ya que al fin y al cabo una página JSP a la hora de ejecutarse se convertirá en un servlet. Una página JSP lanza dos eventos que tiene que ver con dos fases de su ciclo de vida, un evento de inicialización y otro de finalización.

El evento de inicialización se produce cuando el contenedor de páginas JSP recibe la primera petición de una página JSP y por lo tanto debe cargar el servlet correspondiente. El evento de destrucción de una página JSP se produce cuando el contenedor de páginas JSP descarga de memoria el servlet correspondiente a la página JSP, esta destrucción de la página JSP se produce cuando se detiene el contenedor de páginas JSP o cuando ha pasado un tiempo desde que se realizó la última petición de la página JSP y el contenedor de páginas JSP necesita liberar recursos.

Estos eventos son tratados por dos métodos que se ejecutarán cuando se produzca cada uno de los eventos. Cuando se lance el evento de inicialización se ejecutará el método `jspInit()`, y cuando se lance el de destrucción se ejecutará el método `jspDestroy()`. Si deseamos realizar algún tratamiento especial en alguno de los casos, deberemos implementar en nuestra página JSP el método correspondiente.

Los métodos `jspInit()` y `jspDestroy()` pertenecen al ciclo de vida de una página JSP y serán ejecutados de forma automática cuando se produzcan los eventos correspondientes. En el Código Fuente 116 se muestra una página JSP que implementa estos dos métodos, cuando se produce la inicialización de la página JSP se escribe en pantalla un mensaje de inicialización y cuando se realiza la destrucción de la página JSP se muestra en pantalla el mensaje de destrucción.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

```

<html>
<head>
  <title>Eventos</title>
</head>
<body>
<%!public void jspInit(){
    System.out.println("Inicialización");
}
public void jspDestroy(){
    System.out.println("Destrucción");
}%>
<h1>Eventos y ciclo de vida</h1>
</body>
</html>

```

Código Fuente 116

Como se puede comprobar se ha utilizado la sentencia `System.out.println()`, por lo que los mensajes aparecerán en el servidor en el que se esté ejecutando el contenedor de páginas JSP, y la pantalla será la sesión con el intérprete de comandos que tenga iniciada el contenedor de páginas JSP. En mi caso particular, si ejecuto la página JSP anterior y a continuación termino la ejecución de Tomcat, la consola de MS-DOS tiene el aspecto de la Figura 58, en el momento antes de finalizar la ejecución de Tomcat.

No es obligatoria la declaración e implementación de estos métodos por parte de las páginas JSP, sólo será necesario si nos interesan realizar labores de inicialización o de limpieza y liberación de recursos. El método `jspInit()` sería equivalente al método `init()` del ciclo de vida de los servlets, y el método `jspDestroy()` sería el equivalente al método `destroy()` de los servlets.

El método `service()` del ciclo de vida de los servlets también posee una correspondencia en las páginas JSP, se trata del método `_jspService()`. El método `_jspService()` se ejecutará cada vez que se realice la petición de una página JSP, pero este método no lo podemos implementar en nuestra página JSP, sino que lo hace por nosotros el servlet que se genera de forma automática y que se corresponde con la página JSP. Así si accedemos al servlet generado de forma automática a partir de la página JSP del ejemplo anterior observaremos un código similar al del Código Fuente 117.

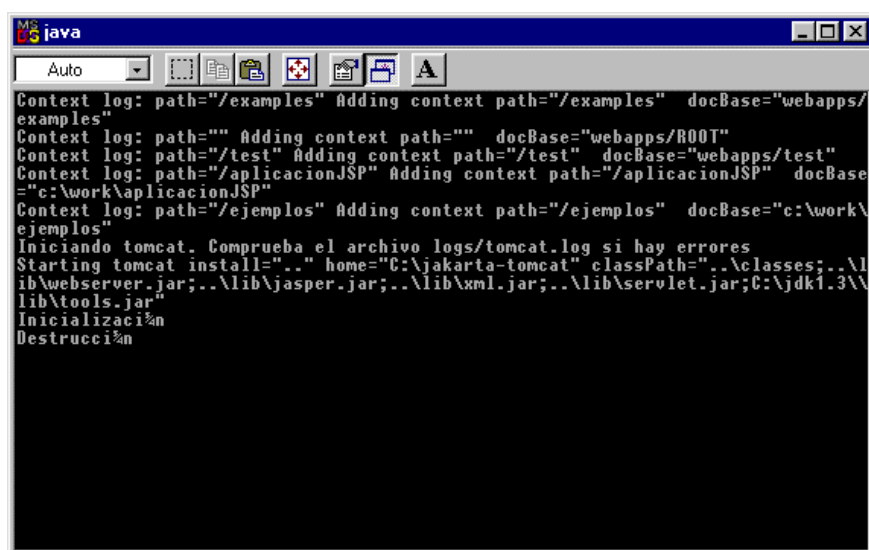


Figura 58. Utilizando los eventos de una página JSP

```

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002feventos_0002ejspeventos_jsp_2 extends HttpJspBase {

    // begin [file="C:\\eventos.jsp";from=(7,3);to=(12,1)]
    public void jspInit(){
        System.out.println("Inicializaci3n");
    }
    public void jspDestroy(){
        System.out.println("Destrucci3n");
    }
    // end

    static {
    }
    public _0002feventos_0002ejspeventos_jsp_2( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\eventos.jsp";from=(0,0);to=(7,0)]

```

```

        out.write("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\">\r\n\r\n<html>\r\n<head>\r\n\t<title>Eventos</title>\r\n</head>\r\n<body>\r\n");
        // end
        // HTML // begin [file="C:\\eventos.jsp";from=(12,3);to=(16,0)]
        out.write("\r\n<h1>Eventos y ciclo de
vida</h1>\r\n</body>\r\n</html>\r\n");
        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}
}

```

Código Fuente 117

Si se modifica una página JSP que ya se encuentra cargada en memoria, cuando se vuelva a realizar una petición de esta página, primero lanzará el evento de destrucción, y a continuación el de inicialización de nuevo, ya que se debe descargar de memoria el servlet antiguo y cargar el nuevo servlet que le corresponde a la página JSP.

Una vez finalizado el apartado dedicado a las declaraciones, vamos a tratar el siguiente elemento de scripting que podemos encontrar dentro de una página JSP, se trata de los scriptlets

## Scriptlets

Un scriptlet es realmente el código fuente de la página JSP, contiene las sentencias correspondientes al lenguaje de script en el que se ha desarrollado la página (actualmente sólo el lenguaje Java). Dentro de un scriptlet nos podemos encontrar cualquier sentencia válida del lenguaje Java, podemos agrupar sentencias, utilizar estructuras de iteración y condicionales, nos permite acceder y manipular los objetos integrados dentro de la página JSP, etc.

La sintaxis general de un scriptlet es:

```
<%scriptlet%>
```

Y el equivalente en XML es:

```
<jsp:scriptlet>scriptlet</jsp:scriptlet>
```

La sintaxis utilizada en las distintas sentencias que forman parte de un scriptlet, es la misma que la utilizada en el lenguaje Java: delimitadores de sentencias, bloques de código, creación de objetos, llamada de métodos, estructuras de control, palabras reservadas (aunque veremos que JSP ofrece una palabras reservadas más que se corresponden con los objetos integrados), etc.

Los lectores que conozcan ASP (Active Server Pages) pueden considerar que un scriptlet es similar a un fragmento de código de script de servidor de una página ASP.

Los scriptlets se ejecutarán cada vez que se realice una petición de una página JSP, ya que se incluirán dentro del método `_jspService()` del servlet generado.



En el Código Fuente 118 se ofrece una página que muestra la fecha y hora actual, para ello utiliza un scriptlet que crea un objeto de la clase Date, y utilizando el objeto integrado out (más tarde lo veremos) devuelve al usuario la fecha y hora actual, también se puede apreciar que se utiliza la directiva page para importar el paquete java.util, ya que contiene a la clase Date.

```
<%@ page import="java.util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Scriptlets</title>
</head>

<body>
<%Date fecha=new Date();
out.println("La fecha actual y hora es: "+fecha);%>
</body>
</html>
```

Código Fuente 118

El código que se genera en el servlet correspondiente es el Código Fuente 119.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;

public class _0002fscriptlet_0002ejspscriptlet_jsp_0 extends HttpJspBase {

    static {
    }
    public _0002fscriptlet_0002ejspscriptlet_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
```

```

JspWriter out = null;
Object page = this;
String _value = null;
try {

    if (_jspx_inited == false) {
        _jspx_init();
        _jspx_inited = true;
    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        "", true, 8192, true);

    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();

    // HTML // begin [file="C:\\scriptlet.jsp";from=(0,31);to=(9,0)]
    out.write("\r\n<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\">\r\n\r\n<html>\r\n<head>\r\n\t<title>Scriptlets</title>\r\n</hea
d>\r\n\r\n<body>\r\n");
    // end
    // begin [file="C:\\scriptlet.jsp";from=(9,2);to=(10,49)]
    Date fecha=new Date();
    out.println("La fecha actual y hora es: "+fecha);
    // end
    // HTML // begin [file="C:\\scriptlet.jsp";from=(10,51);to=(13,0)]
    out.write("\r\n</body>\r\n</html>\r\n");
    // end

} catch (Exception ex) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(ex);
} finally {
    out.flush();
    _jspxFactory.releasePageContext(pageContext);
}
}

```

Código Fuente 119

Como se puede ver el scriptlet se encuentra completamente dentro del método `_jspService()` y el objeto de la clase `Date` no aparece como una variable de clase (atributo) del servlet, sino que es un objeto local al método `_jspService()`, de esta forma cada petición realizada a la página JSP tendrá su propia instancia de este objeto, esto significa que cada vez que se realice una petición a esta página mostrará una fecha y hora distintas, la correspondiente al momento actual de la petición correspondiente.

Sin embargo, si en lugar de instanciar en el scriptlet el objeto de la clase `Date`, lo hacemos en una declaración de las vistas en el apartado anterior, la fecha y hora que mostrará la página JSP será igual para todas las peticiones realizadas a la página, y se corresponderá con la de la primera petición. El Código Fuente 120 muestra esta situación.

```

<%@ page import="java.util.*"%>
<%! Date fecha=new Date();%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

```

```

<html>
<head>
    <title>Scriptlets</title>
</head>

<body>
<%out.println("La fecha actual y hora es: "+fecha);%>
</body>
</html>

```

Código Fuente 120

En el Código Fuente 121 se muestra el servlet que se ha generado en este caso, es importante advertir en que lugar se instancia el objeto fecha en cada uno de los casos.

```

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;

public class _0002fscriptlet_0002ejspscriptlet_jsp_1 extends HttpJspBase {

    // begin [file="C:\\scriptlet.jsp";from=(1,3);to=(1,26)]
        Date fecha=new Date();
    // end

    static {
    }
    public _0002fscriptlet_0002ejspscriptlet_jsp_1( ) {

    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {

    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
            }

```

```

        _jspx_initied = true;
    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        "", true, 8192, true);

    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();

    // HTML // begin [file="C:\\scriptlet.jsp";from=(0,31);to=(1,0)]
    out.write("\r\n");
    // end
    // HTML // begin [file="C:\\scriptlet.jsp";from=(1,28);to=(10,0)]
    out.write("\r\n<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\">\r\n\r\n<html>\r\n<head>\r\n\t<title>Scriptlets</title>\r\n</hea
d>\r\n\r\n<body>\r\n");
    // end
    // begin [file="C:\\scriptlet.jsp";from=(10,2);to=(10,51)]
    out.println("La fecha actual y hora es: "+fecha);
    // end
    // HTML // begin [file="C:\\scriptlet.jsp";from=(10,53);to=(13,0)]
    out.write("\r\n</body>\r\n</html>\r\n");
    // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

Código Fuente 121

Podemos alternar scriptlets con código HTML o XML (elementos estáticos) o con otros elementos de scripting, únicamente se debe cerrar y abrir los delimitadores de los scriptlets cuando sea necesario. En el siguiente ejemplo (Código Fuente 122) se muestra una página JSP que realiza una saludo según la hora actual, según se cumpla una condición se generará un código HTML distinto.

```

<%@ page import="java.util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Scriptlets</title>
</head>

<body>
<%Calendar fecha = Calendar.getInstance();
int hora=fecha.get(Calendar.HOUR_OF_DAY);
if (hora==0) hora=24;
if (hora<=14){%>
    <h1>Buenos días</h1>
<%}else if(hora>14&&hora<21){%>
    <h1>Buenas tardes</h1>
<%}else{%>

```

```
<h1>Buenas noches</h1>
<%}%>
</body>
</html>
```

Código Fuente 122

Los scriptlets no generan de manera directa un contenido en la respuesta enviada al cliente, sino que lo hacen a través de otros medios como puede ser el objeto integrado out. Los elementos de scripting que si realizan esta función son las expresiones, que pasamos a comentar en el siguiente apartado.

## Expresiones

Una expresión es un tipo especial de scriptlet que genera de forma directa una respuesta que se enviará al cliente como parte del contenido generado por la página JSP, es decir, contribuye directamente a la salida de la página JSP. La sintaxis de las expresiones es:

```
<%=expresión%>
```

Y su equivalente en XML es:

```
<jsp:expression>expresión</jsp:expression>
```

Los lectores que conozcan ASP, pueden ver que las sintaxis de las expresiones para las páginas JSP es idéntica a la utilizada en las páginas ASP.

La expresión que se encuentre entre los delimitadores indicados debe ser una expresión válida en el lenguaje de script (Java) correspondiente. El efecto de este elemento de scripting es evaluar la expresión especificada y devolver su resultado formando parte del contenido de la página. Así por ejemplo la página JSP del Código Fuente 123 devolverá la fecha y hora actuales. El resultado de la ejecución de la página se puede observar en la Figura 59.

```
<%@ page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Expresiones</title>
</head>

<body>
La fecha y hora actual es: <i><b><%=new Date()%></b></i>
</body>
</html>
```

Código Fuente 123

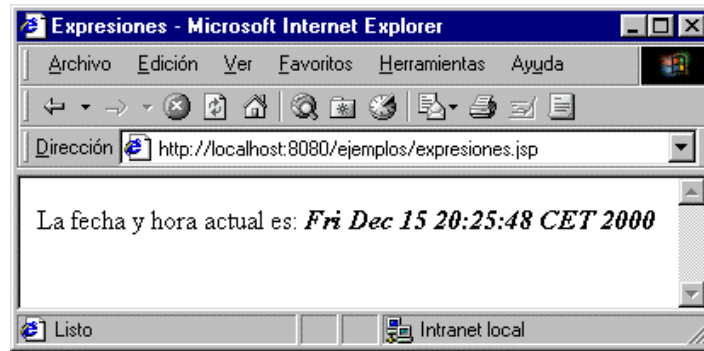


Figura 59. Utilizando expresiones

Se puede decir que una expresión es equivalente a realizar una sentencia `out.print()` a la que le pasamos por parámetro la expresión correspondiente, realmente esto es lo que hace el servlet con el que se corresponde una página JSP que contiene una expresión, en el Código Fuente 124 se puede observar el servlet resultante de la página JSP del ejemplo anterior.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.util.*;

public class _0002fexpresiones_0002ejspexpresiones_jsp_0 extends HttpJspBase {

    static {
    }
    public _0002fexpresiones_0002ejspexpresiones_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {
```

```

        if (_jspx_inited == false) {
            _jspx_init();
            _jspx_inited = true;
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=8859_1");
        pageContext = _jspxFactory.getPageContext(this, request, response,
            "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        // HTML // begin [file="C:\\expresiones.jsp";from=(0,32);to=(9,33)]
        out.write("\r\n<!DOCTYPE HTML PUBLIC \"/>

```

Código Fuente 124

Si rescribimos el último ejemplo del apartado anterior (ejemplo que mostraba un saludo según la hora actual) para que utilice expresiones, podríamos tener la página que se muestra en el Código Fuente 125, la ejecución de esta página es equivalente a la original.

```

<%@ page import="java.util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Scriptlets-Expresiones</title>
</head>

<body>
<%Calendar fecha = Calendar.getInstance();
String mensaje="";
int hora=fecha.get(Calendar.HOUR_OF_DAY);
if (hora==0) hora=24;
if (hora<=14){
    mensaje="Buenos días";
}else if(hora>14&&hora<21){
    mensaje="Buenas tardes";
}else{
    mensaje="Buenas noches";
}
%>

```

```
}%>
<h1><%=mensaje%></h1>
</body>
</html>
```

Código Fuente 125

En el Código Fuente 126 se muestra un fragmento del servlet generado, como se puede ver se sustituye la expresión de la página JSP con una sentencia `out.print()` a la que se le pasa por parámetro la expresión correspondiente, también se puede ver que las sentencias del scriptlet se traducen directamente a sentencias dentro del servlet.

```
// HTML // begin [file="C:\\scriptlet.jsp";from=(0,31);to=(9,0)]
    out.write("\r\n<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0
Transitional//EN\">\r\n\r\n<html>\r\n<head>\r\n\t<title>Scriptlets-
Expresiones</title>\r\n</head>\r\n\r\n<body>\r\n");
// end
// begin [file="C:\\scriptlet.jsp";from=(9,2);to=(19,1)]
    Calendar fecha = Calendar.getInstance();
    String mensaje="";
    int hora=fecha.get(Calendar.HOUR_OF_DAY);
    if (hora==0) hora=24;
    if (hora<=14){
        mensaje="Buenos días";
    }else if(hora>14&&hora<21){
        mensaje="Buenas tardes";
    }else{
        mensaje="Buenas noches";
    }
// end
// HTML // begin [file="C:\\scriptlet.jsp";from=(19,3);to=(20,4)]
    out.write("\r\n<h1>");
// end
// begin [file="C:\\scriptlet.jsp";from=(20,7);to=(20,14)]
    out.print(mensaje);
// end
// HTML // begin [file="C:\\scriptlet.jsp";from=(20,16);to=(22,7)]
    out.write("</h1>\r\n</body>\r\n</html>");
// end
```

Código Fuente 126

Del ejemplo anterior se puede extraer que podemos mezclar expresiones con scriptlets y con elementos estáticos.

Las expresiones se pueden utilizar para mostrar valores individuales de una variable o para mostrar el resultado de un cálculo, como ya hemos dicho como expresión se puede utilizar cualquier expresión válida del lenguaje Java.

En el siguiente apartado se finaliza este capítulo y tratamos el último de los elementos de scripting, los comentarios.



## Comentarios

Dentro de una página JSP podemos distinguir tres tipos de comentarios, los que son propios de la especificación JSP y constituyen un elemento de scripting, los comentarios del lenguaje de scripting, en este caso los comentarios que podemos utilizar en el lenguaje Java, y los comentarios del lenguaje HTML y XML.

La sintaxis de los comentarios JSP es la siguiente:

```
<%--Comentario--%>
```

En este caso no existe una sintaxis equivalente en XML.

En el Código Fuente 127 se muestra una página JSP que contiene estos tres tipos de comentarios, el único comentario que aparecerá en la salida devuelta por la página será el comentario HTML/XML, los comentarios Java aparecerán en el servlet generado, y los comentarios JSP serán pasados por alto, y por lo tanto aparecerán sólo en el código fuente de la página JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Comentarios</title>
</head>
<!--Comentario HTML/XML-->
<body>
    <%//comentario del lenguaje de scripting
out.print("Hola Mundo");%>
    <%--Comentario JSP--%>
</body>
</html>
```

Código Fuente 127

Este ha sido el último elemento de scripting y el último apartado de este capítulo, en los dos siguientes capítulos veremos otro elemento más de la especificación JSP, los objetos integrados y más adelante trataremos las acciones.



# Páginas JSP: objetos integrados I

---

## Introducción

Este capítulo sigue con el ciclo de capítulos que tratan los distintos elementos que podemos encontrar dentro de una página JSP (directivas, elementos de scripting, acciones y objetos integrados), en este caso vamos a tratar los objetos integrados o implícitos que nos ofrece la especificación JSP, el siguiente capítulo tratará la segunda parte de estos objetos implícitos y luego dedicaremos un nuevo capítulo a las acciones estándar.

En este nuevo capítulo (y en el siguiente) trataremos los objetos integrados que nos ofrece la especificación JSP, estos objetos podríamos decir que son el alma de las páginas JSP y permiten desarrollar las páginas JSP de una forma más sencilla que los servlets. Veremos que estos objetos se corresponden con instancias de los interfaces que nos ofrecía la especificación Java servlets. Existen nueve objetos integrados cada uno de ellos con una funcionalidad específica. Ya hemos utilizado un objeto integrado en algunos ejemplos de este texto, el objeto integrado out. En algunos textos los objetos integrados se denominan también variables predefinidas.

Y en un próximo capítulo nos ocuparemos de otro elemento de las páginas JSP, las acciones estándar, que son una serie de etiquetas ofrecidas por la especificación JSP para realizar una tarea determinada, como puede ser el transferir la ejecución de una página JSP a otra página JSP o recurso. Más adelante veremos que nosotros mismos podemos definir nuestras propias acciones a través del mecanismo de librerías de etiquetas personalizadas. JSP ofrece siete acciones distintas, algunas de ellas relacionadas entre sí.

## Objetos integrados

La especificación JSP trata de simplificar la creación y desarrollo de páginas JSP a través de ciertos objetos integrados que se encuentran disponibles en las páginas JSP. Para utilizar estos objetos no hay que declararlos ni instanciarlos sino que son ofrecidos por el contenedor de páginas JSP para que se utilicen en las mismas.

Los objetos integrados se encuentran disponibles en los scriptlets y en las expresiones, sin embargo no se pueden utilizar en las declaraciones, ya sean de métodos o de variables. Esto es debido a que las declaraciones, como veíamos en capítulos anteriores, se traducen en atributos o métodos del servlet resultante, mientras que los objetos integrados sólo están disponibles dentro del método `_jspService()` del servlet generado, debemos recordar que todo el código fuente de los scriptlets de la página JSP se traduce en código fuente del método `_jspService()` del servlet generado.

Los objetos integrados actúan como envoltorios de los interfaces definidos en el API Java servlet, así por ejemplo el objeto integrado `request` representa una instancia del interfaz `javax.servlet.http.HttpServletRequest`, más adelante veremos que utilizar los objetos integrados de JSP nos va a resultar muy sencillo, ya que presentan los mismos métodos que algunos de los interfaces y clases vistos en los servlets, y su cometido es el mismo.

Muchos de los objetos integrados de JSP se obtienen de forma interna, en el servlet generado a partir de la página JSP, a través de la clase abstracta `javax.servlet.jsp.PageContext` a través de varios métodos de esta clase, es decir gracias a una clase que forma parte del API JavaServer Pages, esta clase la veremos con más detenimiento un poco más adelante en este mismo capítulo, pero ahora vamos a presentar brevemente los distintos objetos integrados que ofrece JSP.

### request

Este objeto representa una petición realizada a una página JSP y es una instancia del interfaz `javax.servlet.http.HttpServletRequest`. El objeto `request` es generado por el contenedor de páginas JSP y pasado como un parámetro del método `_jspService()` contenido en el servlet correspondiente a la página JSP.

Este objeto cumple con el cometido del interfaz `HttpServletRequest`, que si recordamos era el de permitir obtener la información que envía el usuario al realizar una petición de un servlet (en este caso a una página JSP), esta información puede ser muy variada, puede ser desde encabezados de petición del protocolo HTTP, cookies enviadas por el usuario o los datos de un formulario.

Este objeto entra en la categoría de los objetos que realizan funciones de entrada/salida, en este caso funciones de entrada.

### response

El objeto `response` es una instancia del interfaz `javax.servlet.http.HttpServletResponse` y encapsula la respuesta que le es enviada al usuario como el resultado de la petición de una página JSP. Al igual que sucedía con el objeto `request`, este objeto es generado por el contenedor de página JSP y pasado como parámetro de método `_jspService()` del servlet correspondiente. En el Código Fuente 128 se ofrece el aspecto que ofrece el método `_jspService()` en un servlet generado a partir de una página JSP.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
```

```
throws IOException, ServletException {
```

Código Fuente 128

El objeto `response` ofrece la funcionalidad de la interfaz `HttpServletResponse`, que era la siguiente: la interfaz `HttpServletResponse` ofrece funcionalidad específica para que los servlets HTTP puedan generar una respuesta del protocolo HTTP para el cliente que realizó la petición del servlet. En nuestro caso, se debe sustituir servlet por página JSP, aunque realmente es finalmente un servlet quien realiza la tarea indicada en la página JSP.

Este objeto, al igual que el anterior, entra en la categoría de objetos de entrada/salida, y en este caso es de salida.

## pageContext

Este objeto no tiene correspondencia con ninguna clase ni interfaz del API de los servlets, sino que es una instancia de una clase perteneciente al API JSP, se trata de la clase `javax.servlet.jsp.PageContext`.

El objeto `pageContext` encapsula el contexto para una página JSP particular, a través de este objeto tenemos acceso a el resto de los objetos implícitos de JSP. Este objeto es creado de manera automática dentro del método `_jspService()` del servlet generado a partir de la página JSP correspondiente, y partir del objeto `pageContext` se obtienen otros objetos integrados dentro del servlet. Esto se puede ver en el Código Fuente 129, que representa un fragmento de un servlet generado a partir de una página JSP cualquiera.

```
application = pageContext.getServletContext();  
config = pageContext.getServletConfig();  
session = pageContext.getSession();  
out = pageContext.getOut();
```

Código Fuente 129

Este objeto pertenece a la categoría de objetos que representan un contexto, en este caso el contexto de una página JSP.

## session

Este objeto representa una sesión de un usuario con nuestra aplicación Web, y es una instancia de la interfaz `javax.servlet.http.HttpSession`. Las sesiones son creadas de forma automática, a no ser que se indique lo contrario dentro de la directiva `page` de la página JSP.

El objeto `session` ofrece toda la funcionalidad que ofrecía la interfaz `HttpSession`, que recordamos era: mantener a través de nuestros servlets información a lo largo de una sesión del cliente con nuestra aplicación Web. Cada cliente tendrá su propia sesión, y dentro de la sesión podremos almacenar cualquier tipo de objeto.

Este objeto se obtiene en el método `_jspService()` del servlet correspondiente a la página JSP que estamos ejecutando.

Este objeto pertenece a la categoría de objetos integrados que representan un contexto, en este caso se trata del contexto de la sesión de un usuario determinado.

## application

El objeto application representa a la aplicación Web (contexto) en la que se encuentra una aplicación JSP, es una instancia del interfaz `javax.servlet.ServletContext`, y por lo tanto posee la misma funcionalidad que es la de permitir comunicarlos con el contenedor de páginas JSP, y permitir mantener un estado más general que el de la sesión para que se comparta entre todos los usuarios de la aplicación Web.

Este objeto se creará en el método `_jspService()` del servlet generado a partir de la página JSP de la que hemos realizado su petición.

El objeto application también pertenece a la categoría de objetos integrados que representan un contexto, en este caso se representa el contexto de la aplicación Web.

## out

Este objeto representa el flujo de salida que se envía al cliente y que forma parte del cuerpo de la respuesta HTTP, esta salida utiliza un búfer intermedio que podemos activar o desactivar e indicar su tamaño utilizando la directiva `page`. El objeto out es una instancia de una clase del API JavaServer Pages llamada `javax.servlet.jsp.JspWriter`, esta clase ofrece una funcionalidad muy similar a la que ofrecía la clase `java.io.PrintWriter`, debemos recordar que en los servlets obteníamos una referencia al flujo de salida de la respuesta a través del método `getWriter()` del interfaz `javax.servlet.ServletResponse`.

Una instancia de este objeto se obtiene en el método `_jspService()` del servlet generado.

Este objeto pertenece a la categoría de los objetos integrados de entrada/salida, y en este caso se trata de un objeto que representa la salida enviada en el cuerpo de la respuesta enviada al cliente.

## config

Este objeto integrado es una instancia del interfaz `javax.servlet.ServletConfig`, y su función es la de almacenar información de configuración del servlet generado a partir de la página JSP correspondiente. Normalmente las páginas JSP no suelen interactuar con parámetros de inicialización del servlet generado, por lo tanto el objeto config en la práctica no se suele utilizar.

El objeto config pertenece a la categoría de objetos integrados relacionados con los servlets. Este objeto se crea, al igual que casi todos los objetos integrados, en el método `_jspService()` del servlet que representa a una página JSP.

## page

El objeto page es una instancia de la clase `java.lang.Object`, y representa la página JSP actual, o para ser más exactos, una instancia de la clase del servlet generado a partir de la página JSP actual. Se puede utilizar para realizar una llamada a cualquiera de los métodos definidos por la clase del servlet. Utilizar el objeto page, es similar a utilizar la referencia a la clase actual, es decir, la referencia `this`.

Al igual que el objeto anterior, este objeto pertenece a la categoría de objetos integrados relacionados con los servlets, en esta caso representa una referencia al propio servlet.

En la práctica el objeto page no se suele utilizar, ya que no suele ser necesario.

Una instancia de este objeto se crea también en el método `_jspService()` del servlet correspondiente.

## exception

Instancia de la clase `java.lang.Throwable`, representa una excepción lanzada en tiempo de ejecución. Este objeto únicamente se encuentra disponible en las páginas de error (indicado mediante la directiva `page`), por lo tanto no se encuentra disponible de forma automática en cualquier página JSP, sólo en aquellas que sean utilizadas para el tratamiento de errores.

Este es un objeto que no pertenece a ninguna categoría de objetos integrados, sino que el mismo es un tipo de objeto para el tratamiento de errores.

En este apartado se ha realizado una primera aproximación a los distintos objetos integrados que podemos utilizar dentro de una página JSP, en los siguientes apartados veremos distintos ejemplos de utilización y comentaremos sus métodos más importantes.

En el Código Fuente 130 se muestra un servlet generado a partir de una página JSP, en este código aparecen destacadas las líneas en las que se crean los distintos objetos integrados.

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002fholamundo_0002ejspholamundo_jsp_0 extends HttpJspBase {

    static {
    }
    public _0002fholamundo_0002ejspholamundo_jsp_0( ) {

    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {

    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
```

```

ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
String _value = null;
try {

    if (_jspx_inited == false) {
        _jspx_init();
        _jspx_inited = true;
    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        "", true, 8192, true);

    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();

    // HTML // begin [file="C:\\holamundo.jsp";from=(0,0);to=(6,4)]
        out.write("<html>\r\n<head>\r\n\t<title>Hola mundo con
JSP</title>\r\n</head>\r\n<body>\r\n<div align=\"center\">\r\n<h1>");
    // end
    // begin [file="C:\\holamundo.jsp";from=(6,6);to=(6,30)]
        out.print("Hola Mundo");
    // end
    // HTML // begin [file="C:\\holamundo.jsp";from=(6,32);to=(11,0)]
        out.write("</h1>\r\n</div>\r\n</body>\r\n</html>\r\n\r\n");
    // end

} catch (Exception ex) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(ex);
} finally {
    out.flush();
    _jspxFactory.releasePageContext(pageContext);
}
}

```

Código Fuente 130

Si nos fijamos bien en el código anterior, sólo aparecen ocho objetos, falta el objeto exception, esto es debido a que este objeto se crea únicamente en las páginas de error. En el Código Fuente 131 se muestra el código correspondiente a un servlet generado a partir de una página JSP de tratamiento de errores, debemos recordar que esto lo indicamos en el atributo isErrorPage de la directiva page de JSP.

```

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

```



```

import java.io.*;

public class _0002fpaginaError_0002ejspfpaginaError_jsp_0 extends HttpJspBase {

    static {
    }
    public _0002fpaginaError_0002ejspfpaginaError_jsp_0( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        Throwable exception = (Throwable)
request.getAttribute("javax.servlet.jsp.jspException");
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=8859_1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="C:\\paginaError.jsp";from=(0,48);to=(7,14)]
            out.write("\r\n<html>\r\n<head>\r\n<title>P gina de
error</title>\r\n</head>\r\n<body>\r\n<h1>Se ha producido una
excepci n</h1>\r\n<b>ERROR:</b> ");
            // end
            // begin [file="C:\\paginaError.jsp";from=(7,17);to=(7,37)]
            out.print(exception.toString());
            // end
            // HTML // begin [file="C:\\paginaError.jsp";from=(7,39);to=(8,16)]
            out.write("<br>\r\n<b>MENSAJE:</b> ");
            // end
            // begin [file="C:\\paginaError.jsp";from=(8,19);to=(8,41)]
            out.print(exception.getMessage());
            // end
            // HTML // begin [file="C:\\paginaError.jsp";from=(8,43);to=(10,0)]
            out.write("<br>\r\n<b>VOLCADO DE PILA:</b> \r\n");
            // end
            // begin [file="C:\\paginaError.jsp";from=(10,2);to=(12,34)]
            StringWriter sSalida=new StringWriter();
            PrintWriter salida=new PrintWriter(sSalida);
            exception.printStackTrace(salida);

```

```

        // end
        // HTML // begin [file="C:\\paginaError.jsp";from=(12,36);to=(13,0)]
        out.write("\r\n");
        // end
        // begin [file="C:\\paginaError.jsp";from=(13,3);to=(13,10)]
        out.print(sSalida);
        // end
        // HTML // begin [file="C:\\paginaError.jsp";from=(13,12);to=(16,0)]
        out.write("\r\n</body>\r\n</html>\r\n");
        // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}
}

```

Código Fuente 131

Para finalizar este apartado se ofrece, a modo de resumen, la Tabla 14. En esta tabla se indica cada uno de los objetos integrados que ofrece la especificación JavaServer Pages, indicando su funcionalidad, interfaz o clase de la que son instancia y categoría a la que pertenecen.

Objetos	Descripción	Clase o interfaz	Categoría
page	Instancia de la página JSP (servlet) actual.	java.lang.Object	Relacionados con los servlets
config	Contiene información relativa a la configuración del servlet generado.	javax.servlet.ServletConfig	Relacionados con los servlets
request	Representa la petición realizada a la página JSP.	javax.servlet.http.HttpServletRequest	Entrada/salida
response	Representa la respuesta dada por la página JSP.	javax.servlet.http.HttpServletResponse	Entrada/salida
out	Representa el flujo de salida del cuerpo de la respuesta HTTP.	javax.servlet.jsp.JspWriter	Entrada/salida
session	Permite mantener una sesión para cada uno de los usuarios conectados a nuestra aplicación Web.	javax.servlet.http.HttpSession	Contexto

application	Representa a la propia aplicación Web en la que nos encontramos.	javax.servlet.ServletContext	Contexto
pageContext	Contexto en el que se ejecuta la página JSP.	javax.servlet.jsp.PageContext	Contexto
exception	Excepción que se ha producido en una página JSP.	java.lang.Throwable	Tratamiento de errores

Tabla 14. Objetos integrados de JSP

En los siguientes apartados trataremos en profundidad cada uno de estos objetos.

## El objeto request

Este objeto va a representar la petición realizada por el usuario que a demandado la página JSP actual. Este objeto ofrece acceso a toda la información asociada con la petición, podemos acceder a los campos enviados de un formulario, cabeceras de petición HTTP, cookies, etc. El objeto request debe implementar el interfaz `javax.servlet.ServletRequest`, y en este caso, como el protocolo es HTTP debe implementar el interfaz `javax.servlet.http.HttpServletRequest`, que es un interfaz que hereda del anterior.

El objeto request es uno de los cuatro objetos integrados que permiten el almacenamiento y recuperación de atributos, la utilización de los atributos ya la vimos al tratar los servlets. Entendíamos por atributo cualquier tipo de objeto Java, que en este caso se va a poder almacenar en una petición realizada a una página JSP o a un recurso del servidor.

El objeto request va a presentar todos los métodos contenidos en los interfaces `ServletRequest` y `HttpServletRequest`, los cuales ya conocemos de los servlets, aunque vamos a comentar de forma breve los métodos más relevantes del objeto request para refrescar la memoria del lector.

Podemos agrupar los métodos del objeto integrado request atendiendo a la funcionalidad que ofrecen. En un primer grupo podemos tener aquellos métodos que nos permiten utilizar los atributos en la petición, estos métodos se comentan a continuación:

- `void setAttribute(String nombre, Object valor)`: almacena un atributo en la petición actual. Como se puede comprobar el atributo puede ser cualquier tipo de objeto.
- `Object getAttribute(String nombre)`: devuelve un atributo almacenado en la petición actual.
- `Enumeration getAttributeNames()`: devuelve dentro de un objeto `java.util.Enumeration` los nombres de todos los atributos disponibles en la petición.
- `void removeAttribute(String nombre)`: elimina de la petición el atributo indicado por parámetro.

Estos métodos para la manipulación de atributos son comunes a cuatro objetos integrados, al objeto request, al objeto session, al objeto application y al objeto pageContext, es decir todos estos objetos

permiten almacenar atributos. A través de la asignación y recuperación de atributos es posible transferir información, en forma de los objetos (atributos) almacenados, entre distintas páginas JSP y servlets.

La diferencia que existe a la hora de almacenar un atributo en cualquiera de los cuatro objetos integrados, es el ámbito del objeto que se va a almacenar como atributo, existen cuatro ámbitos distintos, uno por cada objeto integrado que permite almacenar y recuperar atributos.

Los atributos almacenados dentro del objeto request poseen un ámbito de petición, esto quiere decir que sólo se encontrarán disponibles en la página JSP o servlet a los que se ha realizado una petición.

En el Código Fuente 132 se muestra una página JSP que almacena en el objeto request la fecha actual y a continuación redirige la ejecución hacia otra página JSP, utilizando para ello el método `getRequestDispatcher()` del objeto `application`. Este método lo veremos con más detenimiento cuando tratemos el objeto `application`, pero tiene el mismo significado que el que comentábamos al tratar el interfaz `javax.servlet.ServletContext`, es decir, obtener una referencia a un recurso en el servidor y poder incluir su ejecución en el servlet (página JSP) actual o redirigir la ejecución a este recurso.

```
<%@ page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Origen</title>
</head>
<body>
<%request.setAttribute("fecha",new Date());
RequestDispatcher rd=application.getRequestDispatcher("/destino.jsp");
rd.forward(request,response);%>
</body>
</html>
```

Código Fuente 132

El código fuente de la página JSP de destino que recupera el atributo con ámbito de petición es el que se muestra a continuación (Código Fuente 133).

```
<%@ page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Destino</title>
</head>
<body>
<%Date fecha=(Date)request.getAttribute("fecha");%>
Atributo de la petición: <%=fecha%><br>
</body>
</html>
```

Código Fuente 133

El siguiente conjunto de métodos del objeto request son aquellos que se utilizan para acceder a los parámetros de la petición, estos parámetros suelen ser los campos de un formulario o el contenido de una cadena de consulta (QueryString).

- Enumeration `getParameterNames()`: devuelve en un objeto Enumeration los nombres de los distintos parámetros presentes en la petición.
- String `getParameter(String nombre)`: devuelve el valor del parámetro indicado, si el parámetro no existe devolverá un valor null. Los valores de los parámetros siempre se obtienen como un objeto de la clase String.
- String[] `getParameterValues(String nombre)`: devuelve en un array de objetos String los distintos valores que tiene un parámetro determinado, devolverá un valor nulo si el parámetro no existe.

A continuación vamos a mostrar un ejemplo de utilización de estos métodos por parte del objeto request, vamos a disponer de un formulario con diversos elementos, entre estos elementos vamos a tener algunos con varios valores como son un grupo de casillas de verificación y una lista de selección múltiple. Este formulario va a ser tratado por la propia página JSP que lo crea, y va a mostrar como respuesta los valores facilitados en el formulario. El Código Fuente 134 es el código completo de esta página JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Untitled</title>
</head>

<body>
<%if(request.getParameter("enviar")==null){%>
    <!--No se ha pulsado el botón de envío, por lo tanto se muestra
    el formulario en el navegador-->
    <form action="formulario.jsp" method="GET">
    <b>Datos personales</b><br>
    Nombre: <input type="Text" name="nombre" size="20"><br>
    Apellidos: <input type="Text" name="apellidos" size="20"><br>
    Edad: <input type="Text" name="edad" size="20"><br>
    Email:<input type="Text" name="email" size="30"><br>
    <br>
    <b>Departamentos</b><br>
    <select name="departamentos" multiple size="5">
    <option value="Sistemas">Sistemas</option>
    <option value="Desarrollo">Desarrollo</option>
    <option value="Comercial">Comercial</option>
    <option value="Administración">Administración</option>
    <option value="Formación">Formación</option>
    </select><br><br>
    <b>Lenguajes de programación y entornos de desarrollo</b><br>
    <input type="checkbox" name="lenguajes" value="C++">C++
    <input type="checkbox" name="lenguajes" value="Visual Basic">Visual Basic
    <input type="checkbox" name="lenguajes" value="Java">Java
    <input type="checkbox" name="lenguajes" value="ASP">ASP
    <input type="checkbox" name="lenguajes" value="Delphi">Delphi
    <input type="checkbox" name="lenguajes" value="JSP">JSP
    <br>
    <input type="Submit" name="enviar" value="Enviar">
    </form>
<%}else{%>
    <!--Se ha pulsado el botón de envío y se muestran los datos
    contenidos en el formulario-->
    <ul>
    <li>nombre: <i><%=request.getParameter("nombre")%></i>
    <li>apellidos: <i><%=request.getParameter("apellidos")%></i>
```

```

<li>edad: <i><%=request.getParameter("edad")%></i>
<li>email: <i><%=request.getParameter("email")%></i>
<%String[] valores=request.getParameterValues("departamentos");%>
<li>departamentos: <i>
<% int i;
if (valores!=null){
    for (i=0;i<=valores.length-2;i++){%>
        <%=valores[i]%>,
        <%}%>
        <%=valores[i]%></i>
<%}
valores=request.getParameterValues("lenguajes");%>
<li>lenguajes: <i>
<%if (valores!=null){
    for (i=0;i<=valores.length-2;i++){%>
        <%=valores[i]%>,
        <%}%>
        <%=valores[i]%></i>
<%}%>
</ul>
<a href="formulario.jsp">Volver</a>
<%}%>
</body>
</html>

```

Código Fuente 134

En el código fuente anterior se pueden ver varios tipos de elementos de la página JSP: scriptlets, comentarios, expresiones y objetos integrados, además de código HTML. En la Figura 60 se puede ver el aspecto del formulario que se ha creado, y en la Figura 61 se puede ver el resultado que se obtiene al pulsar el botón de envío de este formulario y ser procesado por la página JSP.

Figura 60: Formulario para tomar los datos

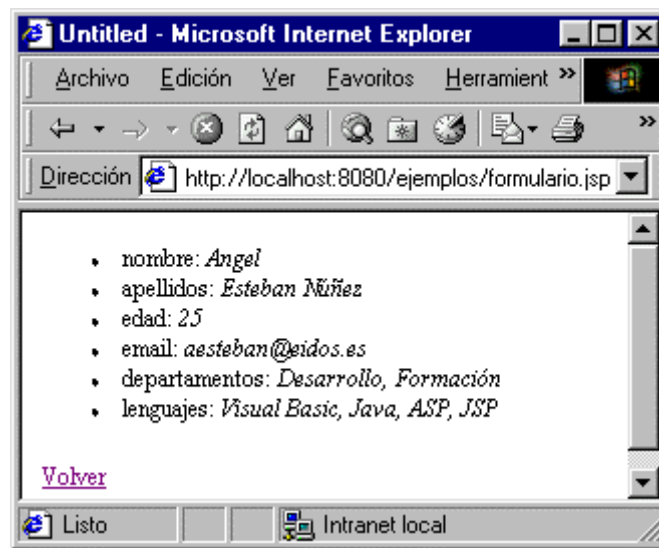


Figura 61. Recuperación de los datos del formulario

El siguiente grupo de métodos del objeto request (ya conocido por todos del interfaz `HttpServletRequest`) es utilizado para obtener información de las cabeceras de petición del protocolo HTTP.

- `Enumeration getHeaderNames()`: devuelve en un objeto `java.util.Enumeration` todas las cabeceras presentes en la petición HTTP.
- `String getHeader(String nombre)`: devuelve el valor de una cabecera de la petición HTTP como un objeto de la clase `String`.
- `Enumeration getHeaders(String nombre)`: devuelve todos los valores de la cabecera especificada como un objeto `java.util.Enumeration`.
- `int getIntHeader(String nombre)`: devuelve el valor de una cabecera como un entero.
- `long getDateHeader(String nombre)`: devuelve el valor de una cabecera como un entero largo que se corresponde con el valor de una fecha en milisegundos.
- `Cookie[] getCookies()`: devuelve todas las cookies asociadas con la petición de una array de objetos `Cookie`.

El último conjunto de métodos del objeto request no tienen una funcionalidad común, sino que poseen funciones variadas, a continuación comentamos cada uno de ellos.

- `String getMethod()`: devuelve el método HTTP que se ha utilizado para realizar la petición.
- `String getRequestURI()`: devuelve la URL sobre la que se ha realizado la petición, pero sin incluir ninguna información del QueryString (cadena de consulta).
- `String getQueryString()`: devuelve la cadena consulta, si existe, que se encuentra en la URL de la petición.

- `HttpSession getSession(boolean crear)`: devuelve un objeto `HttpSession` que representa la sesión del usuario actual, si la sesión no existe y se indica el parámetro `true`, se creará una nueva sesión, que es la que se devuelve.
- `RequestDispatcher getRequestDispatcher(String ruta)`: devuelve una referencia al recurso indicado en la ruta que se pasa por parámetro. Tiene el mismo significado que el visto en los servlets, es decir, nos va a permitir incluir un recurso en la salida de la página JSP actual o redirigir la ejecución de la página JSP a ese recurso.
- `String getRemoteHost()`: devuelve el nombre de la máquina del cliente que realizó la petición.
- `String getRemoteAddr()`: devuelve la dirección IP del cliente que realizó la petición.
- `String getRemoteUser()`: devuelve el identificador de usuario (login) del cliente que realizó la petición, si el usuario se ha autenticado, en caso contrario devolverá el valor nulo (`null`).

A continuación vamos a mostrar un ejemplo de página JSP (Código Fuente 135) que utiliza algunos de los métodos del objeto integrado `request` vistos en estos dos últimos grupos. Se trata de una página que muestra todas la cabeceras de petición existentes, así como las cookies existentes en la petición y algunos datos relativos al cliente y en la forma que se ha realizado la petición.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Objeto request</title>
</head>

<body>
<TABLE BORDER="1" ALIGN="CENTER">
<caption>Cabeceras existentes</caption>
<TR BGCOLOR="#FFAD00">
<TH>Nombre cabecera</TH>
<TH>Valor cabecera</TH>
<%Enumeration nombresCabeceras = request.getHeaderNames();
while(nombresCabeceras.hasMoreElements()) {
    String nombre = (String)nombresCabeceras.nextElement();%>
    <TR><TD><%=nombre%></TD>
    <TD><%=request.getHeader(nombre)%></TD>
<%}%>
</TABLE>
<br>
<div align="center">
<B>Método de petición: </B><%=request.getMethod()%> <BR>
<B>URI pedida: </B><%=request.getRequestURI()%><BR>
<B>Protocolo: </B><%=request.getProtocol()%>
</div>
<br>
<TABLE BORDER="1" ALIGN="CENTER">
<caption>Cookies existentes</caption>
<TR BGCOLOR="#FFAD00">
<TH>Nombre de la cookie</TH>
<TH>valor de la cookie</TH></TR>
<%Cookie[] cookies = request.getCookies();
Cookie cookie;
for(int i=0; i<cookies.length; i++) {
    cookie = cookies[i];%>
```



```

<TR>
<TD><%=cookie.getName()%></TD>
<TD><%=cookie.getValue()%></TD></TR>
<%}%>
</body>
</html>

```

Código Fuente 135

En la Figura 62 se puede ver un ejemplo de ejecución de esta página JSP.

A continuación vamos a tratar el segundo objeto integrado de JSP, el objeto response, que se encuentra en el grupo de objetos de entrada/salida, al igual que el objeto request, ya que el objeto response representa el otro lado de la comunicación entre un cliente y una página JSP, es decir, la respuesta que se envía al cliente.

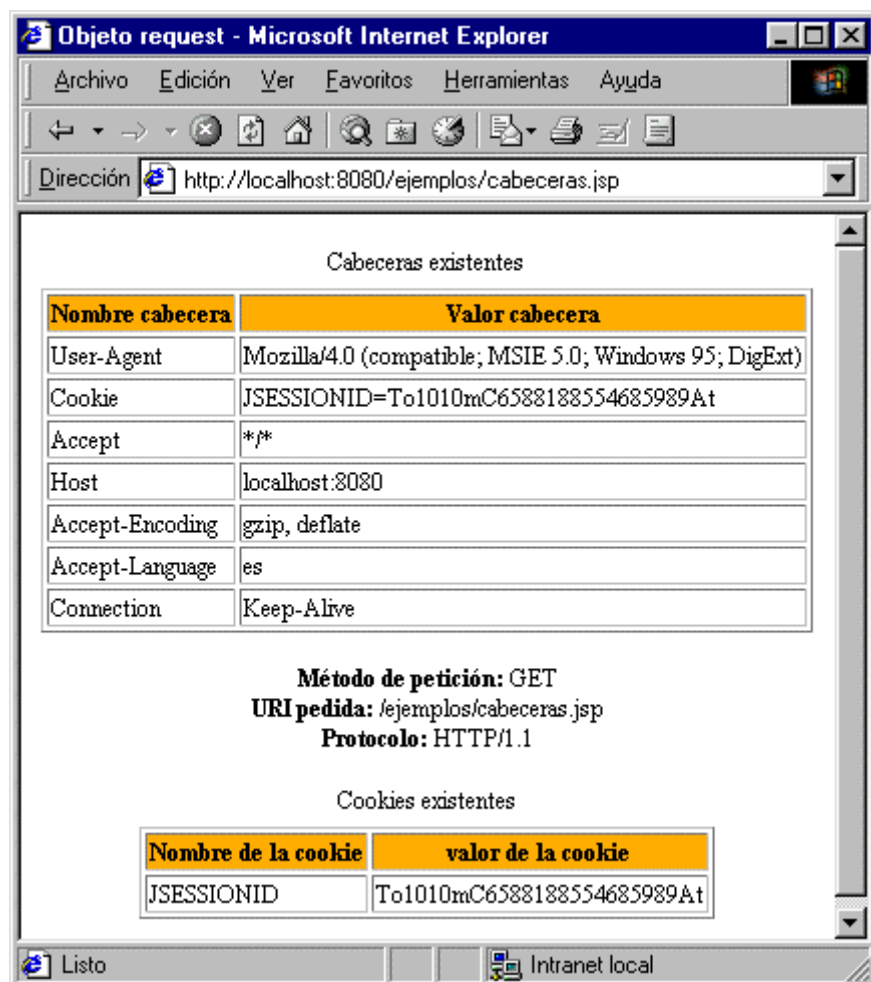


Figura 62. Distintos usos del objeto request

## El objeto response

Este objeto es una instancia del interfaz `javax.servlet.http.HttpServletResponse`, y representa la respuesta del protocolo http que se devuelve al cliente que realizó una petición como resultado de la

ejecución de la página JSP que demandó. Por lo tanto el objeto response presentará los métodos del interfaz `javax.servlet.ServletResponse` y del interfaz `javax.servlet.http.HttpServletResponse`.

No vamos a entrar en demasiado detalle a la hora de tratar este objeto, ya que si sabemos utilizar el interfaz `HttpServletResponse` del API de los servlet (de capítulos anteriores), ya sabremos utilizar el objeto integrado response de la especificación JavaServer Pages.

Existen un par de métodos de este objeto que podemos utilizar para especificar, mediante los tipos MIME, el tipo de respuesta que va a devolver la página JSP. Estos métodos son:

- `void setContentType(String tipo)`: establece el tipo MIME que identifica el tipo de resultado que va a devolver la página JSP al cliente. Opcionalmente se puede especificar la codificación de caracteres del contenido de la respuesta.
- `String getCharacterEncoding()`: devuelve el tipo de codificación que tiene establecida los contenidos de la respuesta.

Existe otro grupo de métodos del objeto response que es utilizado para añadir cabeceras a la respuesta HTTP devuelta por la página JSP.

- `void addCookie(Cookie cookie)`: añade una cookie a la respuesta, representada por el objeto `Cookie` que se le pasa como parámetro.
- `boolean containsHeader(String nombre)`: indica si la cabecera que se pasa como parámetro existe o no en la respuesta.
- `void setHeader(String nombre, String valor)`: establece el encabezado de respuesta HTTP correspondiente con el valor indicado.
- `void setIntHeader(String nombre, int valor)`: establece el encabezado de respuesta HTTP con el valor entero indicado.
- `void setDateHeader(String nombre, long valor)`: establece el encabezado de respuesta HTTP con el valor de fecha indicado.
- `void addHeader(String nombre, String valor)`: añade un encabezado de respuesta con el valor indicado.
- `void addIntHeader(String nombre, int valor)`: añade un encabezado de respuesta con el valor de entero indicado.
- `void addDateHeader(String nombre, long valor)`: añade un encabezado de respuesta con el valor de fecha indicado como una variable long.

El siguiente conjunto de métodos del objeto response es utilizado para establecer códigos de respuesta.

- `void setStatus(int código)`: establece un código de estado para la respuesta.
- `void sendError(int código, String mensaje)`: envía al cliente una respuesta de error indicando el código correspondiente y una descripción del mismo.
- `void sendRedirect(String url)`: se redirecciona la respuesta enviada al cliente a la URL del recurso indicado.

Teniendo en cuenta este último grupo de métodos, si tenemos las página JSP RESPONSE1.JSP (Código Fuente 136) y la página RESPONSE2.JSP (Código Fuente 137), al ejecutar RESPONSE1.JSP obtenemos el resultado de la Figura 63.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Response 1</title>
</head>
<body>
<%response.sendRedirect("response2.jsp");%>
</body>
</html>
```

Código Fuente 136

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Response 2</title>
</head>
<body>
<%response.sendError(545,"Se ha producido un error");%>
</body>
</html>
```

Código Fuente 137

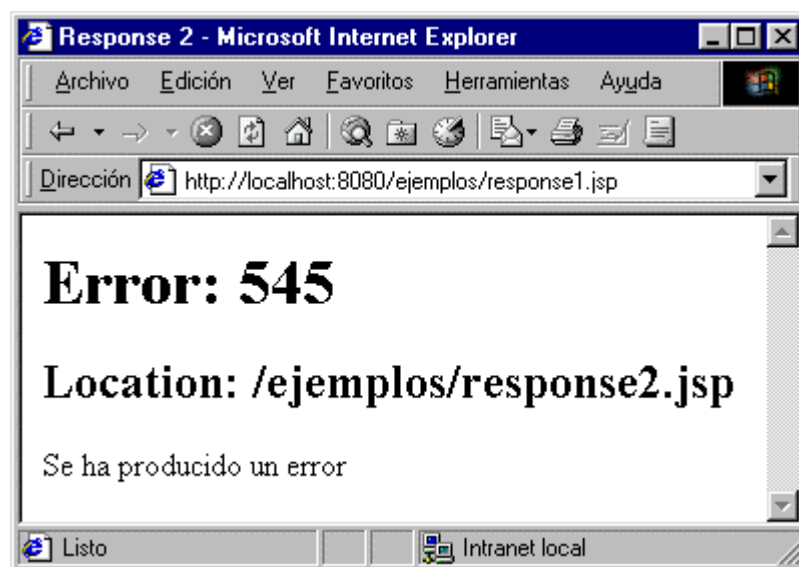


Figura 63. Utilizando el objeto response

Los últimos métodos que vamos a tratar del objeto response son los que permiten utilizar el mecanismo de reescritura de URLs (URL rewriting).

- `String encodeRedirectURL(String url)`: codifica la URL especificada para utilizarla en el método `sendRedirect()`. Si la codificación es necesaria se incluye el identificador de sesión, en caso contrario se devuelve al URL sin cambiar.

- `String encodeURL(String url)`: codifica la URL especificada para incluir en un enlace información correspondiente a la sesión. Si la codificación es necesaria se incluye el identificador de sesión, en caso contrario se devuelve al URL sin cambiar.

El siguiente objeto que vamos a tratar es también un objeto perteneciente al conjunto de objetos de entrada/salida, es el objeto out. Este objeto va a representar el flujo de salida que forma parte de la respuesta enviada al cliente.

## El objeto out

Este objeto es una instancia de la clase `javax.servlet.jsp.JspWriter`, y representa el flujo de salida de la página JSP.

La clase `JspWriter` es una clase abstracta que hereda de la clase `java.io.Writer`, implementando varios de los métodos de la clase `java.io.PrintWriter`. En particular hereda los distintos métodos `write()` de la clase `Writer`, e implementa todos los métodos `print()` y `println()` ofrecidos por la clase `PrintWriter`.

La clase `java.io.PrintWriter` ya la utilizábamos para manipular el flujo de salida de los servlets, para enviar contenidos al cuerpo de la respuesta HTTP, para ello utilizábamos el método `getWriter()` del interfaz `javax.servlet.ServletResponse`.

El objeto out va a realizar la misma labor que el objeto `PrintWriter` que manejábamos en los servlets. Se debe señalar que el objeto integrado out utilizará un búfer intermedio, para enviar el contenido del cuerpo de la respuesta HTTP al cliente, si no indicamos lo contrario en la directiva `page`.

En este caso, al no ser el objeto out instancia de una clase ya conocida por todos de los capítulos de los servlets, vamos a comentar todos los métodos que ofrece la clase `JspWriter`:

- `void clear()`: elimina el contenido del búfer. Si el búfer ya ha sido vaciado para enviar su contenido al cliente y se lanza el método `clear()` se producirá una excepción de la clase `IOException`.
- `void clearBuffer()`: elimina los contenidos actuales del búfer. Si algún contenido del búfer ya ha sido enviado al cliente no se producirá una excepción.
- `void close()`: cierra el flujo de salida enviando al cliente los contenidos del búfer, una vez cerrado el flujo de salida, no podremos utilizar el método `print()` o `write()` sobre el objeto out ya que no generará salida alguna. Una vez que ha sido lanzado el método `close()` sobre el objeto out no se devolverá ningún contenido al cliente.
- `void flush()`: vacía el contenido actual del búfer del objeto out (flujo de salida) enviándolo al cliente.
- `int getBufferSize()`: devuelve el tamaño del búfer de salida en bytes. El tamaño por defecto de este búfer es de 8KB, pero podemos modificarlo a través de la propiedad `buffer` de la directiva `page`.
- `int getRemaining()`: devuelve el número de bytes que no se han utilizado todavía del búfer de salida.
- `boolean isAutoFlush()`: indica si el contenido del búfer es enviado de forma automática al cliente cuando el búfer se llena. Por defecto el contenido del búfer se envía de

forma automática al cliente cuando se llena, este comportamiento se puede modificar a través de la propiedad `autoFlush` de la directiva `page`.

- `void newLine()`: escribe un separador de línea.

La clase `javax.servlet.jsp.JspWriter` posee numerosos métodos `print()` y `println()` para escribir en el flujo de salida que se devuelve al cliente distintos tipos de datos y objetos del lenguaje Java.

La clase `JspWriter` hereda de la clase `java.io.Writer`, y por lo tanto el objeto `out` también presentará los métodos de esta otra clase, como pueden ser los distintos métodos `write()` que permiten escribir en un flujo de salida de caracteres.

El objeto `out` lo vamos a utilizar sobretodo para enviar contenidos al cliente, dónde utilizamos una expresión de JSP con la notación `<%= %>` se puede utilizar el objeto `out` con su correspondiente método `print()` o `println()`, pasándole por parámetro la expresión que se quiere evaluar y devolver.

A continuación se ofrece una página JSP (Código Fuente 138) que utiliza el objeto `out` para mostrar información acerca del búfer de salida. Como se puede observar se ha indicado un valor para el tamaño del búfer a utilizar mediante la directiva `page`.

```
<%@ page buffer="10kb"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Objeto out</title>
</head>

<body>
<%out.println("<h3>Pagina que muestra información sobre el búfer</h3><br>");
//out.flush();
int total=out.getBufferSize();
int disponible=out.getRemaining();
int utilizado=total-disponible;
out.print("Estado del búfer:<br>");%>
<%=utilizado%>bytes/<%=total%>bytes=<%= (100.0*utilizado)/total%>%><br>
AutoFlush=<%=out.isAutoFlush()%>
</body>

</html>
```

Código Fuente 138

El resultado de la ejecución de esta página se puede ver en la Figura 64.

Pero si antes de obtener la información sobre el búfer, lanzamos el método `flush()` sobre el objeto `out`, la cantidad de búfer utilizado será cero, ya que en ese momento se habrá vaciado el búfer y enviado al cliente.

El siguiente apartado que cierra el presente capítulo está dedicado al objeto integrado `exception`, que nos permite obtener información acerca de la excepción que se ha producido en una página JSP.

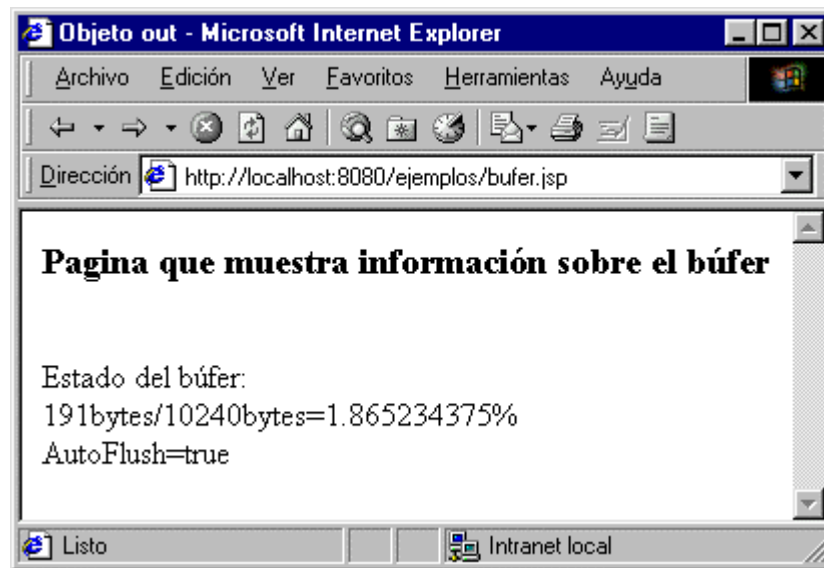


Figura 64. Utilizando el objeto out

## El objeto exception

Este objeto ya no pertenece al grupo de objetos integrados de entrada/salida, este objeto tiene una categoría propia y es la de el tratamiento de errores. El objeto integrado exception es una instancia de la clase `java.lang.Throwable` y nos permite obtener información relativa al error que se ha producido.

Al contrario que los objetos integrados vistos hasta ahora, este objeto no está disponible en cualquier página JSP, sino sólo en aquellas que se utilizan para el tratamiento de errores. Indicaremos que una página es de tratamiento de errores mediante la propiedad `isErrorPage` de la directiva `page`.

A continuación mostramos los distintos métodos que nos ofrece el objeto exception:

- `String getMessage()`: devuelve el mensaje de error descriptivo que se corresponde con la excepción que se ha lanzado.
- `void printStackTrace(PrintWriter salida)`: devuelve en un objeto `PrintWriter` el volcado de pila que se corresponde que la excepción que se ha producido.
- `String toString()`: devuelve una breve descripción del objeto `Throwable`, que consiste en el nombre de la clase de la excepción y una breve descripción.

A continuación se muestra una página JSP que utiliza el objeto exception (Código Fuente 139), esta página debe ser por lo tanto una página de tratamiento de errores. Esta es una página de error bastante típica.

```
<%@ page isErrorPage="true" import="java.io.*"%>
<html>
<head>
    <title>Página de error</title>
</head>
<body>
<h1>Se ha producido una excepción</h1>
```

```

<b>ERROR:</b> <%=exception.toString()%><br>
<b>MENSAJE:</b> <%=exception.getMessage()%><br>
<b>VOLCADO DE PILA:</b>
<%=StringWriter sSalida=new StringWriter();
PrintWriter salida=new PrintWriter(sSalida);
exception.printStackTrace(salida);%>
<%=sSalida%>
</body>
</html>

```

Código Fuente 139

En el Código Fuente 140 se muestra una página que genera una excepción (división por cero) y que indica que la página de error que se debe utilizar es la del ejemplo anterior. Si ejecutamos esta última página obtenemos un resultado como el de la Figura 65.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page errorPage="paginaError.jsp" %>
<html>
<head>
    <title>Genera excepción</title>
</head>
<body>
<%=int i=1/0;%>
</body>
</html>

```

Código Fuente 140

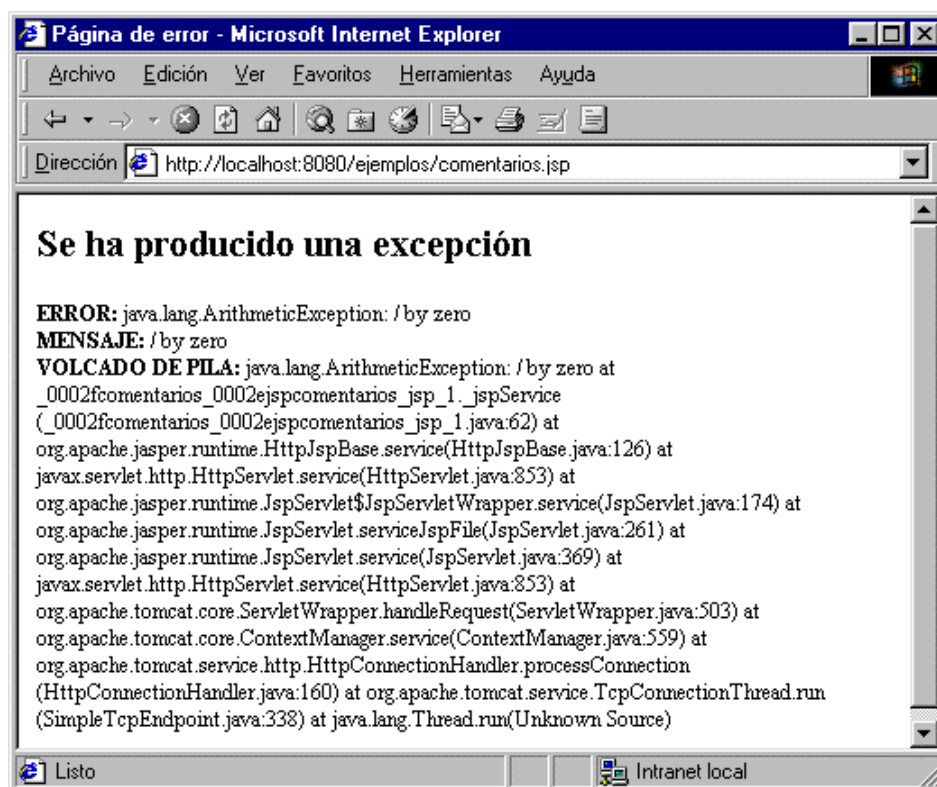


Figura 65. Utilizando el objeto exception

Más adelante en este texto dedicaremos un capítulo completo al tratamiento de errores, será en este capítulo dónde retomemos el objeto integrado exception.

En el siguiente capítulo seguiremos viendo los objetos integrados de JSP que nos quedan, que son los objetos contextuales session, application y pageContext, y los objetos relacionados con los servlets page y config.



# Páginas JSP: objetos integrados II

---

## Introducción

Este capítulo sigue perteneciendo a los elementos de las páginas JSP, en este caso seguimos tratando los objetos integrados de JSP, comentando los cinco objetos integrados que nos faltan por tratar. En el siguiente capítulo trataremos otro elemento de las páginas JSP que son las acciones.

El primer objeto integrado que vamos a explicar es el objeto session que pertenece al grupo de objetos de contexto o contextuales y que es utilizado para representar la sesión actual del usuario, y por lo tanto permite almacenar información relativa ese usuario.

## El objeto session

Este objeto integrado es una instancia del interfaz `javax.servlet.http.HttpSession` y va a representar la sesión del usuario actual conectado a nuestra aplicación Web, este objeto lo vamos a utilizar sobretodo para almacenar y recuperar información relativa al usuario actual.

Las sesiones en las páginas JSP tienen el mismo significado que en los servlets, es decir, una sesión se mantiene entre distintas páginas JSP pertenecientes a una misma aplicación Web y además podemos almacenar información en el objeto session en forma de atributos (objetos) para que se mantenga entre distintas páginas JSP, aquí nos encontramos con un nuevo ámbito de los objetos, el ámbito de sesión. En el capítulo anterior ya comentábamos que un objeto puede tener distintos ámbitos a la hora de almacenarlo como atributo, vimos el ámbito de la petición cuando se almacenaba un objeto en el objeto request. El objeto session también permite almacenar objetos en forma de atributos dentro del ámbito de la sesión, y por lo tanto presenta los mismos métodos que el objeto request para almacenar y

recuperar los atributos con ámbito de sesión. A continuación recordamos brevemente estos métodos que eran los siguientes:

- `void setAttribute(String nombre, Object valor)`: almacena un atributo en la sesión actual. Como se puede comprobar el atributo puede ser cualquier tipo de objeto.
- `Object getAttribute(String nombre)`: devuelve un atributo almacenado en la sesión actual.
- `Enumeration getAttributeNames()`: devuelve dentro de un objeto `java.util.Enumeration` los nombres de todos los atributos disponibles en la sesión.
- `void removeAttribute(String nombre)`: elimina de la sesión el atributo indicado por parámetro.

En este mismo capítulo veremos los dos ámbitos restantes para los objetos que pueden ser almacenados como atributos de objetos integrados, que serán el ámbito de aplicación mantenido por el objeto integrado `application`, y el ámbito de página que lo mantiene el objeto integrado `pageContext`.

En el Código Fuente 141 se muestra una página JSP que almacena en el objeto `session` varios objetos, y en el Código Fuente 142 se puede observar otra página JSP que recupera los atributos almacenados en el objeto `session`. Lógicamente estas dos páginas deben encontrarse en el mismo contexto Web, es decir, en la misma aplicación Web, recordamos que las aplicaciones Web las definíamos en el fichero de configuración del servidor Jakarta Tomcat llamado `SERVER.XML`, a través de las etiquetas XML `<Context>`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Objeto session</title>
</head>

<body>
<h1>Almacena objetos en la sesión</h1>
<%session.setAttribute("nombre",new String("Angel"));
session.setAttribute("edad",new Integer(25));
session.setAttribute("fecha", new Date());%>
</body>
</html>
```

Código Fuente 141

```
<%@ page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Objeto session</title>
</head>

<body>
<h1>Muestra los atributos de la sesión</h1>
<table border="1" align="center">
<tr><th>Atributo</th><th>Valor</th></tr>
<%Enumeration atributos=session.getAttributeNames();
```

```

while (atributos.hasMoreElements()) {
    String nombreAtributo=(String) atributos.nextElement();
    out.println("<tr><td>"+nombreAtributo+"</td>");
    out.println("<td>"+session.getAttribute(nombreAtributo)+"</td></tr>");
}
out.println("</table>");%>
</body>
</html>

```

Código Fuente 142

Si en el navegador indicamos que nos avise cuando se va a crear una cookie, al cargar la primera página JSP de una aplicación Web el navegador mostrará una ventana como la de la Figura 66, para advertir de que se va a crear una cookie de inicio de sesión de JSP, pero si en la página que cargamos tiene el valor false en el atributo session de la directiva page no se creará esta cookie y por lo tanto no estará disponible el objeto session ya que la página no participará en la sesión actual.

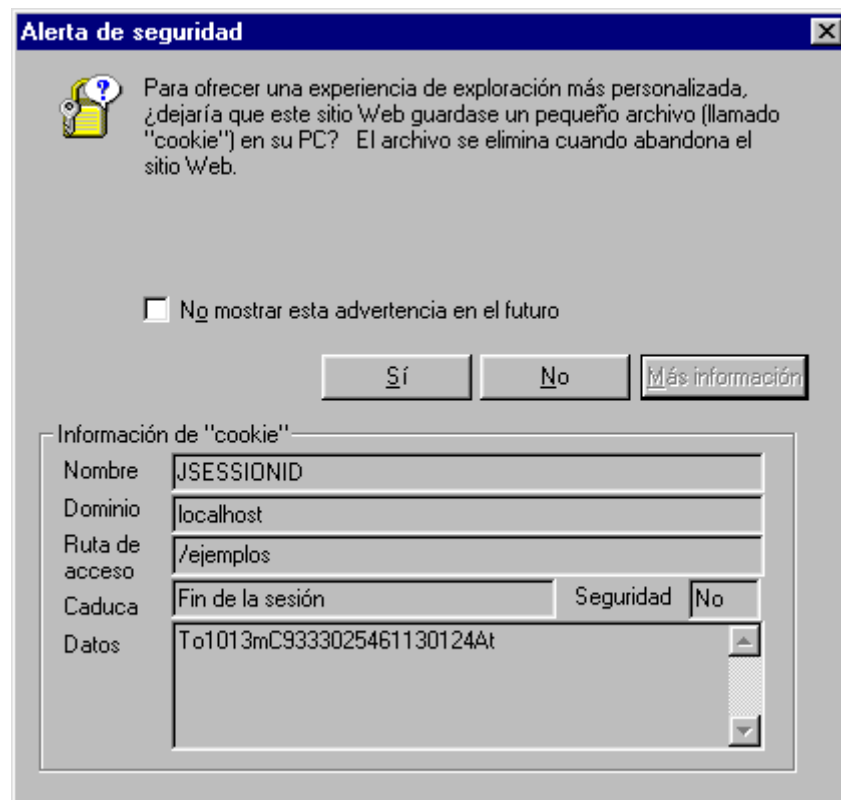


Figura 66. Cookie de inicio de sesión

Si ejecutamos las dos páginas anteriores en el orden lógico, es decir, primero la que almacena los atributos en la sesión y a continuación la que los muestra obtendremos el resultado de la Figura 67.

El objeto session no se encuentra disponible en todas las páginas JSP, sólo estará disponible en aquellas que pertenezcan a la sesión actual. Por defecto todas las páginas JSP incluidas dentro de una misma aplicación Web pertenecen a la sesión actual, pero esto podemos alterarlo a través del atributo session de la directiva page. Si asignamos el valor false a este atributo la página no participará en la sesión actual y por lo tanto no se podrá utilizar el objeto session, y si intentamos utilizarlo obtendremos un error indicando que el objeto session no se encuentra definido.

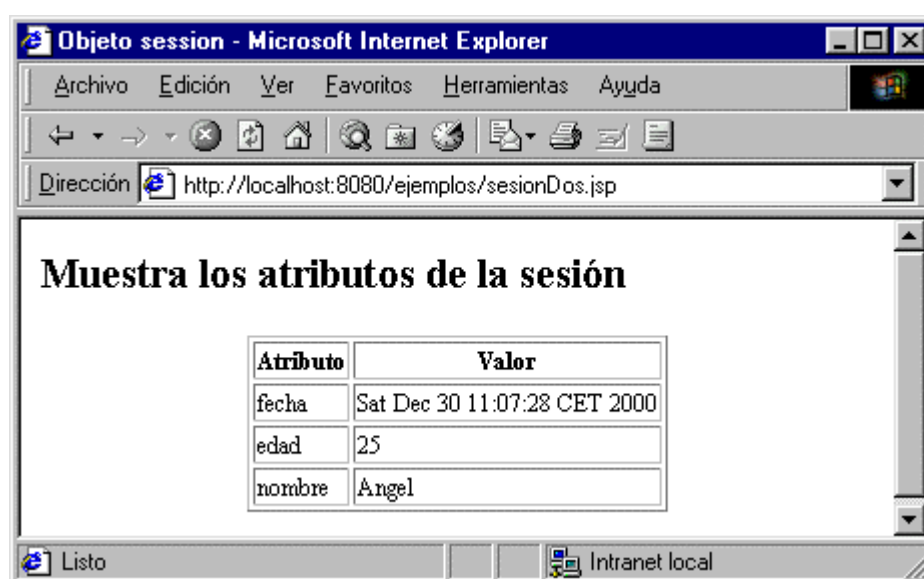


Figura 67. Recuperando los atributos almacenados

El objeto session presenta los mismos métodos que veíamos para los servlets con el interfaz `javax.servlet.http.HttpSession`, de todas formas vamos a comentar los más relevantes.

- `long getCreationTime()`: devuelve la fecha y hora en la que fue creada la sesión, medido en milisegundos desde el 1 de enero de 1970.
- `String getId()`: devuelve una cadena que se corresponde con el identificador único asignado a la sesión, este valor se corresponde con el valor de la cookie `JSESSIONID` utilizada para poder realizar el mantenimiento de la sesión de un usuario determinado.
- `long getLastAccessedTime()`: devuelve en milisegundos la fecha y hora de la última vez que el cliente realizó una petición asociada con la sesión actual.
- `int getMaxInactiveInterval()`: devuelve el máximo intervalo de tiempo, en segundos, en el que una sesión permanece activa entre dos peticiones distintas de un mismo cliente, es decir, es el tiempo de espera máximo en el que pertenece activa una sesión sin que el cliente realice ninguna petición relacionada con la sesión actual. El valor por defecto que puede permanecer inactiva una sesión es de 30 segundos. Una vez transcurrido este tiempo el contenedor de páginas JSP destruirá la sesión, liberando de la memoria todos los objetos que contiene la sesión que ha caducado.
- `void invalidate()`: este método destruye la sesión de forma explícita, y libera de memoria todos los objetos (atributos) que contiene la sesión.
- `boolean isNew()`: devuelve verdadero si la sesión se acaba de crear en la petición actual o el cliente todavía no ha aceptado la sesión (puede rechazar la cookie de inicio de sesión). Este método devolverá falso si la petición que ha realizado el cliente ya pertenece a la sesión actual, es decir, la sesión ya ha sido creada previamente,
- `void setMaxInactiveInterval(int intervalo)`: establece, en segundos, el máximo tiempo que una sesión puede permanecer inactiva antes de ser destruida por el contenedor de páginas JSP.

En el Código Fuente 143 se muestra una página JSP que envía al cliente una serie de datos relativos a la sesión actual.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Objeto session</title>
</head>

<body>
<h1>Información de la sesión</h1>
Sesion nueva: <%=session.isNew()%><br>
Creación de la sesión: <%=new Date(session.getCreationTime())%><br>
Último acceso a la sesión: <%=new Date(session.getLastAccessedTime())%><br>
ID de la sesión: <%=session.getId()%><br>
Mantenimiento de la sesión inactiva: <%=session.getMaxInactiveInterval()/60%>
minutos<br>
</body>
</html>
```

Código Fuente 143

El resultado de la ejecución de esta página JSP se puede observar en la Figura 68.

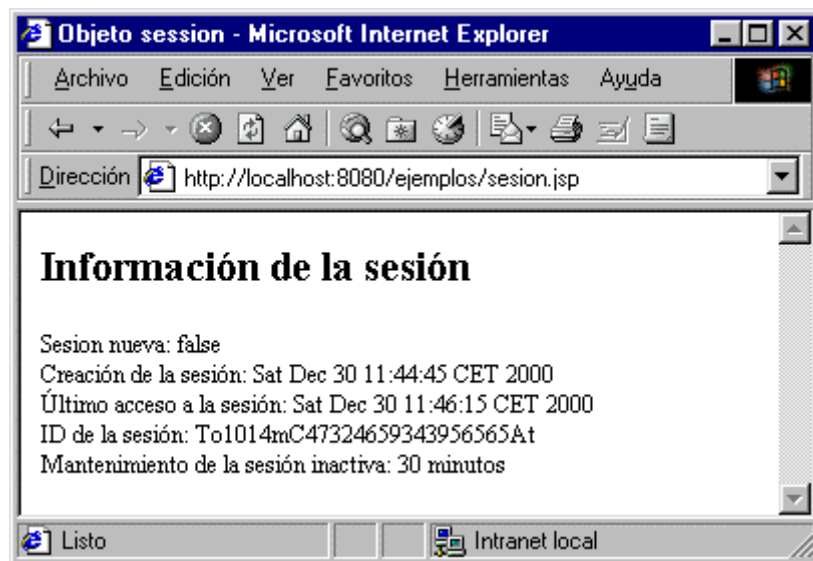


Figura 68. Obteniendo información relativa a la sesión actual

Cada vez que se almacena un nuevo objeto en una sesión utilizando el método `setAttribute()`, si el objeto que se está almacenado implementa el interfaz `javax.servlet.http.HttpSessionBindingListener`, se ejecutará el método `valueBound()` de este interfaz. El método `setAttribute()` lanzará un evento de la clase `javax.servlet.http.HttpSessionBindingEvent`, para indicar al objeto oyente de estos eventos que ha sido almacenado en una sesión.

Cuando se elimina un objeto de la sesión también se lanza el evento `HttpSessionBindingEvent`, pero en este caso se ejecuta el método `valueUnbound()` del interfaz `HttpSessionBindingListener`. Un objeto se eliminará de la sesión de forma directa cuando lo indiquemos mediante el método

`removeAttribute()`, y también cuando la sesión se destruya una vez que haya pasado el intervalo de inactividad máximo de la sesión. La destrucción de una sesión se puede realizar lanzando sobre el objeto `HttpSession` correspondiente el método `invalidate()`.

Los eventos que lanzan los objetos `HttpSession` y la forma de recuperarlos en objetos oyentes lo veremos con detenimiento en el capítulo dedicado al modelo de componentes JavaBeans.

El siguiente objeto integrado que vamos a tratar está muy relacionado con el objeto `session`, se trata del objeto `application`, que representa también un contexto, pero mucho más general, representa a la aplicación Web actual.

## El objeto `application`

Este objeto integrado es una instancia del interfaz `javax.servlet.ServletContext` y representa la aplicación Web a la que pertenece la página JSP actual. Debemos recordar que las aplicaciones Web se encuentran definidas en el fichero de configuración del contenedor de páginas JSP, y una página JSP pertenecerá a una aplicación u otra dependiendo de su URL.

El objeto `application` se encuentra disponible en cualquier página JSP y como instancia del interfaz `ServletContext` ofrece todos los métodos de este interfaz, además de los métodos ya conocidos que se utilizan para almacenar y recuperar los atributos con ámbito de aplicación, no debemos olvidar que el objeto `application` define otro ámbito para los objetos que se almacenan como atributos, y en este caso es el ámbito más general posible.

En el Código Fuente 144 se muestra una página JSP que almacena varios atributos en la aplicación Web actual, y en el Código Fuente 145 se ofrece otra página JSP que muestra todos los atributos presentes en la aplicación. Al ejecutar la primera página JSP los atributos que se han almacenado estarán disponibles para cualquier página JSP que pertenezca a la aplicación Web actual, aunque no participe en la misma sesión, incluso si no utiliza la sesión.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Objeto application</title>
</head>

<body>
<h1>Almacena objetos en la aplicación</h1>
<%application.setAttribute("nombre",new String("Angel"));
application.setAttribute("edad",new Integer(25));
application.setAttribute("fecha", new Date());%>
</body>
</html>
```

Código Fuente 144

```
<%@ page import="java.util.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Objeto application</title>
```

```

</head>

<body>
<h1>Muestra los atributos de la aplicación</h1>
<table border="1" align="center">
<tr><th>Atributo</th><th>Valor</th></tr>
<%Enumeration atributos=application.getAttributeNames();
while (atributos.hasMoreElements()) {
    String nombreAtributo=(String)atributos.nextElement();
    out.println("<tr><td>" + nombreAtributo + "</td>");

    out.println("<td>" + application.getAttribute(nombreAtributo) + "</td></tr>");
}
out.println("</table>"); %>
</body>
</html>

```

Código Fuente 145

El resultado de la ejecución de la segunda página JSP que muestra los atributos de la aplicación se puede ver en la Figura 69.

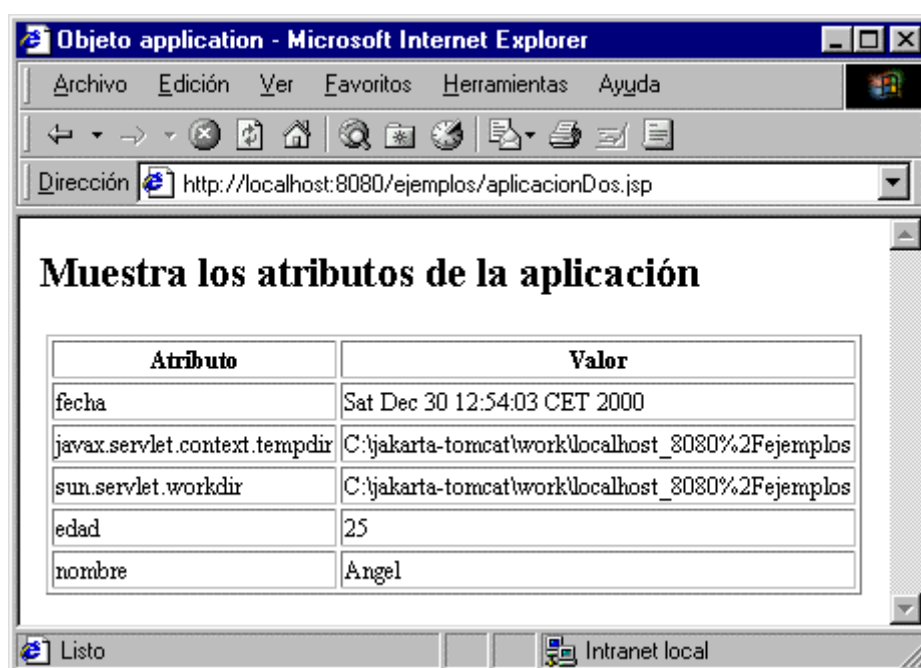


Figura 69. Obteniendo los atributos de la aplicación

A la vista de la figura anterior se puede comprobar que además de los atributos que hemos almacenado nosotros en nuestra aplicación, existen un par de atributos que toda aplicación Web contiene, estos atributos son `javax.servlet.context.tempdir` y `sun.servlet.workdir` e indican el directorio de trabajo asignado a la aplicación Web, en el que se generarán los ficheros fuente Java que se corresponden con los servlets que se generan a partir de las páginas JSP y los ficheros de clase correspondientes.

A la hora de utilizar los atributos almacenados en el objeto `application` es necesaria una sincronización, ya que objeto almacenado en un objeto `application` (un objeto con ámbito de aplicación) es accesible por todos los usuarios conectados a la aplicación, a diferencia de la sesión, que cada usuario posee su propia sesión y por lo tanto sus propios objetos. Debido a esto a la hora de acceder a un atributo del objeto `application` se puede producir accesos simultáneos, dando lugar a

problemas de concurrencia, por ejemplo a la hora de modificar un valor de un objeto (atributo) de la aplicación.

Para evitar estos problemas de concurrencia tenemos dos soluciones. Una de ellas es la utilización del atributo `isThreadSafe` de la directiva `page`, si le asignamos a este atributo el valor `false`, el servlet resultante de la página JSP implementará el interfaz `javax.servlet.SingleThreadModel`. Si el servlet implementa este interfaz, nos asegura que sólo va a servir una petición cada vez, por lo tanto no existirán múltiples hilos de ejecución (uno por cada petición) que pueda acceder a un mismo objeto de la aplicación al mismo tiempo.

El interfaz `SingleThreadModel` no posee ningún método, si queremos que una página JSP utilice este modelo de ejecución simplemente debemos indicarlo en el atributo `isThreadSafe` de la directiva `page` asignándole el valor `false`.

Esta solución es la más sencilla, pero también la más drástica, ya que eliminamos la interesante característica que nos brindaban los servlets y las páginas JSP a través de los múltiples hilos de ejecución simultáneos. Además si se accede al servlet que implementa el interfaz `SingleThreadModel` de forma frecuente, se verá afectado el rendimiento de la aplicación Web, ya que cada petición deberá esperar en una cola hasta que la instancia del servlet se encuentre libre.

Otro motivo para desechar esta solución de forma definitiva, es el hecho de que algunos contenedores de servlets, cuando se indica que el servlet implementa el interfaz `SingleThreadModel`, crearán un conjunto de instancias de servlets, y por lo tanto no se podrá asegurar el acceso exclusivo a un objeto de la aplicación, ya que dos instancias distintas de un servlet podrían estar accediendo al mismo objeto de la aplicación al mismo tiempo.

Se debe señalar que los problemas de concurrencia no afectan exclusivamente a la hora de utilizar los objetos a nivel de aplicación, sino que también podemos tener el problema de los accesos simultáneos a la hora de acceder a los atributos de la propia página JSP (variables miembro definidas en la página JSP).

La segunda solución que se comentaba, y que resulta la ideal, consiste en permitir la ejecución multihilo de las páginas JSP sin modificar el valor del atributo `isThreadSafe`, que por defecto es `true`, y solucionar los problemas de sincronización utilizando la palabra clave `synchronized` cuando exista un problema potencial de accesos concurrentes. En el siguiente ejemplo veremos como utilizar este mecanismo. Como se puede comprobar el problema de la concurrencia es idéntico al que teníamos con los servlets.

El ejemplo que vamos a utilizar es el de crear un contador de accesos a una página JSP determinada, se va a utilizar un atributo del objeto `application` para ir almacenado la cuenta de accesos. El Código Fuente 146 muestra esta página.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Accesos</title>
</head>
<body>
<%synchronized(application){
    Integer valorActual= (Integer)application.getAttribute("accesos");
    if (valorActual!=null)
        application.setAttribute("accesos",
                                new Integer(valorActual.intValue()+1));
    else
        application.setAttribute("accesos",new Integer(1));%>
```



```
A esta página se ha accedido: <b>
<%=application.getAttribute("accesos")%>
</b> veces
<%}%>
</body>
</html>
```

Código Fuente 146

Como se puede comprobar se utiliza la sincronización, ya que se puede dar la situación en la que dos clientes accedan al mismo tiempo a la página JSP, con lo que se podría perder uno de los accesos en la cuenta de accesos, o realizar una cuenta incorrecta, por lo tanto es necesario utilizar la palabra clave `synchronized` sobre el objeto integrado `application`.

Para realizar una prueba con esta página JSP podemos cargar varias veces la página en una misma sesión del navegador, luego cerraremos el navegador y al volver a cargar la página comprobaremos que la cuenta de accesos se mantiene. Esta cuenta de accesos se pondrá a cero si detenemos el contenedor de páginas JSP.

Un resultado de la ejecución de esta página JSP se ofrece en la Figura 70.



Figura 70. Cuenta de accesos a la página

Los métodos principales del objeto `application`, que se encuentran presentes en el interfaz `ServletContext`, los podemos agrupar en cuatro categorías que pasamos a comentar a continuación.

El primer conjunto de métodos es utilizado para obtener información relativa al contenedor de servlets/páginas JSP en el que estamos ejecutando nuestras páginas JSP.

- `int getMajorVersion():` devuelve el número de versión superior que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
- `int getMinorVersion():` devuelve el número de versión inferior que se corresponde con la versión del API servlet implementada por el contenedor de servlets.
- `String getServerInfo():` devuelve el nombre y la versión del contenedor de servlets en el que la página JSP se está ejecutando.

La siguiente página JSP (Código Fuente 147) utiliza estos tres métodos para obtener información relativa al contenedor.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Información del contenedor</title>
</head>
<body>
Información contenedor:<b><%=application.getServerInfo()%></b><br>
Versión del API Java Servlet:<b><%=application.getMajorVersion()%>.
<%=application.getMajorVersion()%></b>
</body>
</html>

```

Código Fuente 147

El resultado de la ejecución de esta sencilla página JSP es el de la Figura 71.



Figura 71. Información del contenedor

Como se puede comprobar toda la información que obtenemos a través de estos métodos es relativa al contenedor de servlets en el que se está ejecutando nuestra página JSP, pero también puede ser interesante obtener información relativa a cerca de la especificación JavaServer Pages que implementa el contenedor. Para ello la especificación JavaServer Pages ofrece una clase abstracta llamada `javax.servlet.jsp.JspEngineInfo` que nos permite obtener la versión de JSP que implementa el contenedor. La forma de obtener esta información se puede ver en el Código Fuente 148.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Versión JSP</title>
</head>

<body>
<%JspFactory fac=JspFactory.getDefaultFactory();%>
<b>JavaServer Pages versión
<%=fac.getEngineInfo().getSpecificationVersion()%></b>
</body>
</html>

```

Código Fuente 148

Como se puede observar debemos utilizar la clase `javax.servlet.jsp.JspFactory` como intermediaria para poder obtener la información deseada. Un ejemplo de ejecución de esta página se ofrece en la.

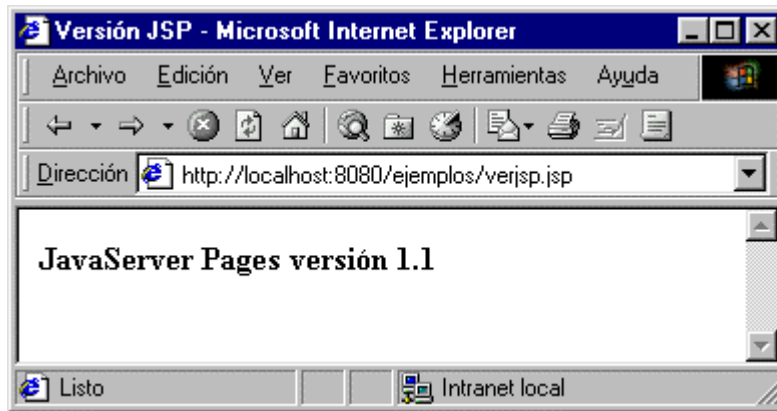


Figura 72. Información del contenedor JSP

El segundo grupo de métodos del objeto `application` tiene la misión de interactuar con rutas del servidor y ficheros (recursos) del servidor.

- `ServletContext getContext(String ruta)`: devuelve el contexto de la aplicación que se corresponde con la ruta que se le pasa por parámetro
- `String getMimeType(String fichero)`: devuelve en una cadena el tipo MIME del fichero especificado, si el tipo MIME es desconocido se devolverá el valor `null`.
- `String getRealPath(String ruta)`: devuelve un objeto `String` que contiene la ruta real (física) de una ruta virtual especificada como parámetro.
- `RequestDispatcher getRequestDispatcher(String ruta)`: este método nos permite obtener una referencia a un recurso en el servidor y poder incluir su ejecución en la página JSP actual o redirigir la ejecución a este recurso. Este método lo utilizaremos a la hora de crear las arquitecturas de las páginas JSP y su integración con los servlets.
- `URL getResource(String ruta)`: devuelve un objeto `URL` que se corresponde con el recurso localizado en la ruta indicada como parámetro.
- `InputStream getResourceAsStream(String ruta)`: devuelve el recurso localizado en la ruta indicada como un objeto `InputStream`.

A continuación se ofrece una página JSP (Código Fuente 149) que utiliza algunos de los métodos mostrados.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Objeto application</title>
</head>
<body>
El tipo MIME del fichero bufer.jsp es:
<%=application.getMimeType("/cabecera.html")%><br>
```

```
La ruta física de la aplicación actual es:  
<%=application.getRealPath("")%><br>  
Se incluye el contenido del fichero accesos.jsp: <br><br>  
<%out.flush();  
RequestDispatcher rq=application.getRequestDispatcher("/accesos.jsp");  
rq.include(request,response);%>  
</body>  
</html>
```

Código Fuente 149

El resultado de la ejecución de esta página es el de la Figura 73.

El siguiente grupo de métodos del objeto application es utilizado para obtener los parámetros de inicialización de la aplicación Web correspondiente, estos parámetros son muy similares a los parámetros de inicialización de un servlet, pero en este caso no se utilizan para un servlet concreto, sino para toda la aplicación Web correspondiente.

- `String getInitParameter(String nombre)`: este método devuelve una cadena que contiene el valor de un parámetro de inicialización del contexto (de la aplicación). Si el parámetro no existe se devolverá el valor null. Estos parámetros serían muy similares a los parámetros de inicialización de un servlet, pero en este caso se aplican a una aplicación Web completa.
- `Enumeration getInitParameters()`: devuelve en un objeto `java.util.Enumeration` los nombres de todos los parámetros de inicialización de la aplicación Web.

Los parámetros de inicialización de la aplicación Web se indican en el fichero de configuración de la aplicación correspondiente, es decir, en el fichero WEB.XML.

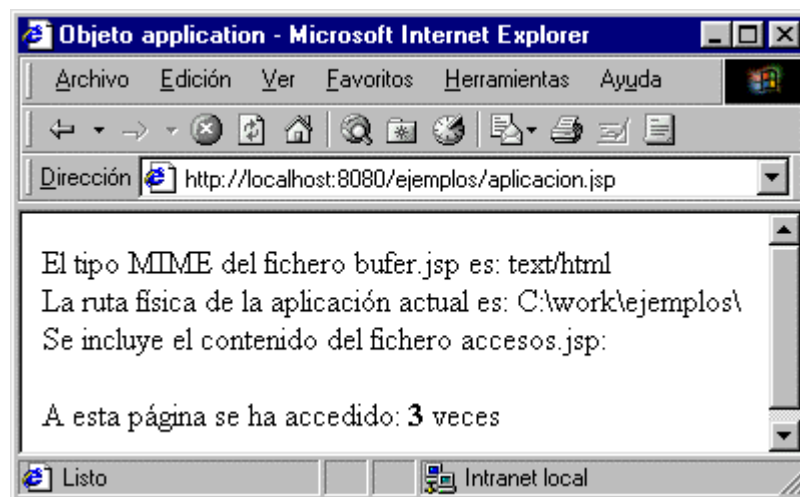


Figura 73. Algunas utilidades del objeto application

Para definir un parámetro de inicialización de la aplicación utilizaremos la etiqueta XML `<context-param>`, que a su vez contiene las siguientes etiquetas que podemos utilizar para definir el parámetro y que comentamos a continuación:

- `<param-name>`: nombre del parámetro de inicialización de la aplicación.

- `<param-value>`: lo utilizaremos para indicar el valor del parámetro.
- `<description>`: descripción de la funcionalidad del parámetro.

En el Código Fuente 150 se muestra una página JSP que recupera un parámetro de la aplicación que contiene la dirección de correo del administrador del sistema.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Objeto application</title>
</head>

<body>
<%String email=application.getInitParameter("email");%>
En caso de error póngase en contacto con el administrador<br>
del sistema (<a href="mailto:<%=email%>"><%=email%></a>)
</body>
</html>
```

Código Fuente 150

Si en el fichero WEB.XML tenemos el Código Fuente 151, el resultado de la ejecución de la página anterior sería el de la Figura 74.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
    <context-param>
        <param-name>email</param-name>
        <param-value>aesteban@eidos.es</param-value>
        <description>
            Dirección de correo del administrador del sistema
        </description>
    </context-param>
</web-app>
```

Código Fuente 151

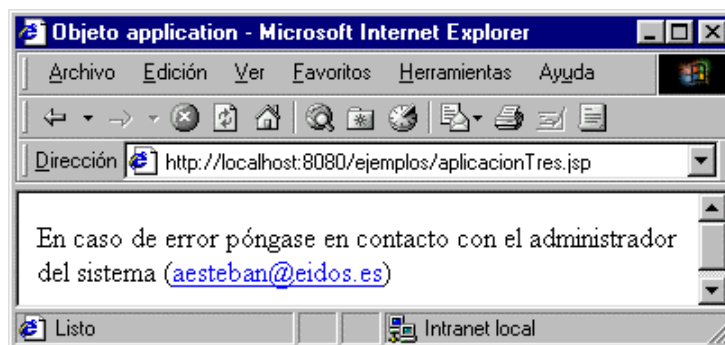


Figura 74. utilizando los parámetros de inicialización

El último grupo de métodos del objeto `application` es utilizado para almacenar información en el fichero de registro de la aplicación Web.

- `void log(String mensaje)`: escribe el mensaje especificado en el fichero de registro de las aplicaciones Web.
- `void log(String mensaje, Throwable throwable)`: escribe un mensaje descriptivo y un volcado de pila para una excepción determinada en el fichero de registro de las aplicaciones Web.

La página JSP del Código Fuente 152 escribe un mensaje en el fichero de registro de la aplicación Web, que es el mismo fichero en el que escribían los servlets, es decir, el fichero `SERVLET.LOG` que se encuentra dentro del directorio `jakarta-tomcat/logs`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Objeto application</title>
</head>

<body>
<h1>Página que escribe en el fichero de registro</h1>
<%application.log("Escribe en el fichero de registro");%>
</body>
</html>
```

Código Fuente 152

La ejecución de la página anterior añadiría la siguiente línea al fichero de registro de la aplicación Web.

```
Context log path="/ejemplos" :Escribe en el fichero de registro
```

El siguiente objeto que vamos a tratar es el objeto `pageContext`, que como su nombre indica también representa un contexto y por lo tanto un ámbito para el almacenamiento de los objetos, en este caso se trata de la propia página JSP.

## El objeto `pageContext`

Este es el último de los objetos pertenecientes al grupo de objetos que representan un contexto, este objeto instancia de la clase abstracta `javax.servlet.jsp.PageContext`, permite acceder al espacio de nombres de la página JSP actual, ofrece también acceso a varios atributos de la página así como una capa sobre los detalles de implementación.

Además el objeto `pageContext` ofrece una serie de métodos que permiten obtener el resto de los objetos integrados de JSP, también nos va a permitir acceder a los atributos pertenecientes a distintos ámbitos.

Podemos distinguir varios grupos de métodos en la clase `PageContext`, el primero de estos grupos es el que nos permite acceder y obtener los distintos objetos integrados de JSP. Estos métodos no los utilizaremos nunca directamente en nuestras páginas JSP veremos que serán muy útiles cuando

tratemos la implementación del mecanismo de etiquetas personalizadas, además estos métodos sí que son utilizados por el servlet equivalente a una página JSP determinada para obtener los distintos objetos integrados y así poder realizar la traducción. En el se muestra un fragmento de un servlet equivalente a una página JSP.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    try {

        if (_jspx_inited == false) {
            _jspx_init();
            _jspx_inited = true;
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=8859_1");
        pageContext = _jspxFactory.getPageContext(this, request, response,
            "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        .....
        .....
        .....
```

Código fuente 153

Los métodos que permiten obtener los objetos integrados son los siguientes.

- `Exception getException()`: devuelve la excepción que se le ha pasado a la página de error, es decir, devuelve una referencia al objeto integrado exception.
- `JspWriter getOut()`: devuelve el flujo de salida actual, es decir, el objeto out.
- `Object getPage()`: devuelve la instancia del servlet para la página actual, es decir, el objeto integrado page.
- `ServletRequest getRequest()`: devuelve la petición que inició el procesamiento de la página JSP actual, es decir, el objeto integrado request.
- `ServletResponse getResponse()`: devuelve la respuesta que la página envía al cliente que realizó la petición, es decir, el objeto integrado response.
- `ServletConfig getServletConfig()`: devuelve un objeto que va a representar la configuración del servlet con el que se corresponde la página JSP actual, es decir, se obtiene una referencia al objeto integrado config.

- `ServletContext getServletContext()`: devuelve el contexto en el que se ejecuta el servlet que se corresponde con la página actual, es decir, se obtiene una referencia al objeto integrado `application`.
- `HttpSession getSession()`: devuelve la sesión asociada con la petición de página actual, si existe se obtendrá una referencia al objeto integrado `session`.

El siguiente conjunto de métodos es utilizado para controlar el procesamiento de páginas, indicando si se desea incluir un contenido en la página actual o redirigir la ejecución de la página hacia otro recurso.

- `void forward(String ruta)`: este método redirige la ejecución de la página actual hacia el recurso indicado mediante su ruta relativa, sin embargo una vez que se haya procesado el recurso al que se ha enviado la ejecución, se vuelve a procesar la página de origen. Si el contenido del búfer del objeto `out` ya ha sido enviado (aunque sea parcialmente) al cliente se producirá un error.
- `void include(String ruta)`: incluye el resultado de la ejecución del recurso indicado a través de su ruta relativa. La diferencia con el método anterior es que en la página de origen es posible enviar previamente el contenido del búfer.

El Código Fuente 154 y el Código Fuente 155 muestra una página JSP que redirige su ejecución hacia otra página JSP y una página que incluye el resultado de la ejecución de otra página JSP, respectivamente. El resultado de la ejecución de ambas páginas es el mismo, pero si antepone la sentencia `out.flush()` a la sentencia `pageContext.forward()` y `pageContext.include()`, en el primer caso se producirá una excepción, ya que se ha enviado algún contenido al cliente.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Objeto pageContext</title>
</head>

<body>
<h1>Redirige la ejecución hacia otro recurso</h1>
<%pageContext.forward("verjsp.jsp");%>
</body>
</html>
```

Código Fuente 154

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Objeto pageContext</title>
</head>

<body>
<h1>Incluye la ejecución de otro recurso</h1>
<%pageContext.include("verjsp.jsp");%>
</body>
</html>
```

Código Fuente 155



Como ya hemos dicho el objeto `pageContext` representa el ámbito de la página actual, por lo tanto poseerá los métodos ya conocidos para acceder y almacenar atributos, pero además de estos métodos el objeto `pageContext` posee otros métodos que nos permiten acceder a atributos de diverso ámbito (aplicación, sesión, petición y la propia página). Estos métodos son los que se muestran a continuación:

- `void setAttribute(String nombre, Object valor, int ámbito):` almacena el atributo correspondiente con el nombre y ámbitos indicados. Los ámbitos se encuentran definidos como constantes de tipo entero de la clase `pageContext`, más adelante comentaremos la correspondencia de la constante con el ámbito. Esta clase también presenta el método `setAttribute()` sin el último parámetro, es decir, sin el parámetro del ámbito, en ese caso se almacenará el atributo con ámbito de página.
- `Enumeration getAttributeNamesInScope(int ámbito):` devuelve en un objeto del interfaz `java.util.Enumeration` todos los nombres de los atributos que pertenecen al ámbito indicado.
- `Object getAttribute(String nombre, int ámbito):` devuelve el valor del atributo indicado que pertenece al ámbito que le pasamos por parámetro, también disponemos del método `getAttribute()` sin utilizar el parámetro del ámbito, en este caso nos devolverá el valor del atributo que pertenece al ámbito de la página.
- `Object findAttribute(String nombre):` devuelve el valor del atributo indicado buscándolo en los distintos ámbitos, empezando del más particular al más general. Se devolverá el valor del primer atributo que se encuentre, el orden de consulta de los ámbitos es página, petición, sesión y aplicación.
- `int getAttributesScope(String nombre):` devuelve el ámbito al que pertenece el atributo cuyo nombre le pasamos por parámetro. El orden de búsqueda en los ámbitos es: es página, petición, sesión y aplicación, se devuelve el primer ámbito en el que se encuentre el atributo.
- `void removeAttribute(String nombre, int ámbito):` elimina el atributo indicado en un ámbito específico. También podemos utilizar el método `removeAttribute()` sin el parámetro del ámbito, de esta forma se eliminará el atributo en el ámbito de página.

Las distintas constantes definidas en la clase `PageContext` que se corresponden con los cuatro ámbitos distintos de un atributo son las siguientes (de más particular a más general):

- `PAGE_SCOPE`: ámbito de los atributos almacenados en el objeto integrado `pageContext`, representa el ámbito de página.
- `REQUEST_SCOPE`: ámbito de los atributos almacenados en el objeto `request`, representa el ámbito de petición.
- `SESSION_SCOPE`: ámbito de los atributos almacenados en el objeto integrado `session`, representa el ámbito de sesión.
- `APPLICATION_SCOPE`: ámbito de los atributos almacenados en el objeto integrado `application`, representa el ámbito de aplicación.

La página JSP del Código Fuente 156 utiliza algunos de los métodos vistos anteriormente, accediendo a los atributos de distintos ámbitos. El resultado de la ejecución de la página anterior lo podemos ver en la Figura 75.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<%!public String ambito(int ambi){
    if(ambi==PageContext.SESSION_SCOPE) return "SESSION_SCOPE";
    if(ambi==PageContext.APPLICATION_SCOPE) return "APPLICATION_SCOPE";
    if(ambi==PageContext.REQUEST_SCOPE) return "REQUEST_SCOPE";
    if(ambi==PageContext.PAGE_SCOPE) return "PAGE_SCOPE";
    return "";
}%>
<html>
<head>
    <title>Objeto pageContext</title>
</head>

<body>
<%session.setAttribute("ciudad",new String("Madrid"));
pageContext.setAttribute("nombre",new String("Angel"),
    PageContext.SESSION_SCOPE);
pageContext.setAttribute("pais",new String("España"),
    PageContext.PAGE_SCOPE);
pageContext.setAttribute("pais",new String("Francia"),
    PageContext.SESSION_SCOPE);
Enumeration atributos=pageContext.getAttributeNamesInScope(
    PageContext.SESSION_SCOPE);
while(atributos.hasMoreElements()) {
    String nombre = (String)atributos.nextElement();%>
    <%=nombre%>=
    <%=pageContext.getAttribute(nombre,PageContext.SESSION_SCOPE)%><br>
<%}%><br>
pais=<%=pageContext.findAttribute("pais")%><br><br>
Ámbito de país=<%=ambito(pageContext.getAttributesScope("pais"))%>
</body>
</html>
```

Código Fuente 156

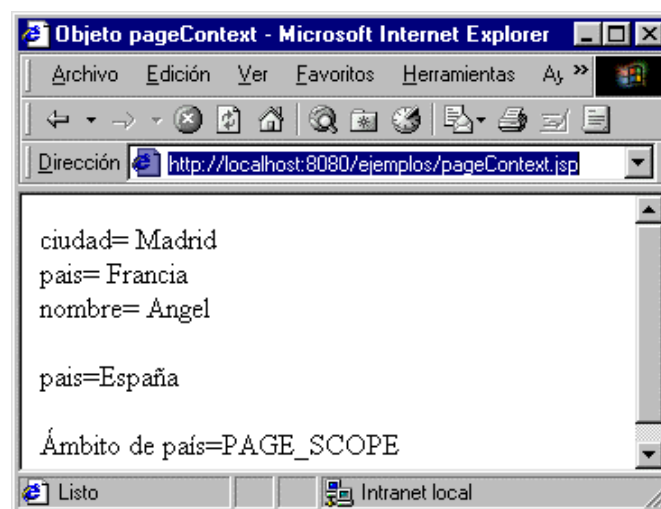


Figura 75. Accediendo a atributos de distintos ámbitos

El siguiente objeto que vamos a tratar es el objeto integrado page, que no debemos confundir con el objeto pageContext, este objeto representa a la propia página JSP, o más correctamente, a la instancia de la clase del servlet resultante de la traducción de la página JSP correspondiente.

## Objeto page

El objeto page es una instancia de la clase `java.lang.Object`, y representa la página JSP actual, o para ser más exactos, una instancia de la clase del servlet generado a partir de la página JSP actual. Se puede utilizar para realizar una llamada a cualquiera de los métodos definidos por la clase del servlet. Utilizar el objeto page, es similar a utilizar la referencia a la clase actual, es decir, la referencia `this`.

Este objeto pertenece a la categoría de objetos integrados relacionados con los servlets, en esta caso representa una referencia al propio servlet.

En nuestras páginas JSP no es muy común utilizar el objeto page.

En el siguiente apartado, que finaliza este capítulo y que contiene el último de los objeto integrados, trataremos el objeto config que representa la configuración del servlet con el que se corresponde la página JSP actual.

## El objeto config

Este objeto integrado es una instancia del interfaz `javax.servlet.ServletConfig`, y su función es la de almacenar información de configuración del servlet generado a partir de la página JSP correspondiente. Normalmente las páginas JSP no suelen interactuar con parámetros de inicialización del servlet generado, por lo tanto el objeto config en la práctica no se suele utilizar.

El objeto config pertenece a la categoría de objetos integrados relacionados con los servlets.

Los métodos que ofrece este objeto integrado son los siguientes.

- `String getInitParameter(String nombreParametro)`: este método devuelve un objeto `String` que representa el valor del parámetro de inicialización cuyo nombre le pasamos por parámetro. Si el parámetro no existe se devolverá `null`.
- `Enumeration getInitParameterNames()`: devuelve el nombre de todos los parámetros de inicialización del servlet como un objeto `java.util.Enumeration` que contiene objetos `String`, uno por cada nombre del parámetro. Devolverá un objeto `Enumeration` vacío si el servlet que se corresponde con la página JSP actual no tiene parámetros de inicialización.
- `ServletContext getServletContext()`: este método devuelve una referencia al objeto `ServletContext` en el que se está ejecutando el servlet que se corresponde con la página JSP actual
- `String getServletName()`: devuelve el nombre de la instancia actual del servlet.

El siguiente capítulo será último capítulo dedicado a los elementos de las páginas JSP. Trataremos las acciones que son una serie de etiquetas ofrecidas por la especificación JSP para realizar una tarea determinada, como puede ser el transferir la ejecución de una página JSP a otra página JSP o recurso. Más adelante veremos que nosotros mismos podemos definir nuestras propias acciones a través del

mecanismo de librerías de etiquetas personalizadas. JSP ofrece siete acciones distintas, algunas de ellas relacionadas entre sí.

# Páginas JSP: acciones

---

## Introducción

Este capítulo está dedicado a un conjunto de etiquetas especiales que podemos encontrar en la especificación JSP y que se denominan acciones. Este capítulo cierra el ciclo de capítulos dedicados a los elementos que podemos encontrar y utilizan en una página JSP.

Existen siete acciones estándar dentro de la especificación JSP. Estas etiquetas afectan le comportamiento de la ejecución de las páginas JSP y afectan también a la respuesta devuelta al cliente. Las acciones estándar ofrecen al desarrollador o autor de páginas JSP una funcionalidad básica.

Las acciones estándar son:

- `<jsp:forward>`
- `<jsp:include>`
- `<jsp:plugin>`
- `<jsp:param>`
- `<jsp:useBean>`
- `<jsp:setProperty>`
- `<jsp:getProperty>`

A continuación comentaremos cada una de ellas.

## La acción `<jsp:forward>`

Esta acción permite redirigir la ejecución de la página JSP actual hacia otro recurso de forma permanente, si antes de utilizar esta etiqueta ya se ha enviado algún contenido del búfer del flujo de salida al cliente se producirá un error.

Cuando se localiza una etiqueta `<jsp:forward>` todo el contenido previo generado por la página JSP se descarta y se inicia la ejecución del recurso indicado en la acción `<jsp:forward>`.

La sintaxis general de esta etiqueta es la siguiente:

```
<jsp:forward page="URLLocal" />
```

El atributo `page` de la acción `<jsp:forward>` indica la localización del recurso al que se va a redirigir la ejecución de la página JSP actual. El valor de este atributo no tiene porque ser una constante, podemos utilizar expresiones y variables dentro de nuestra página JSP para indicar el valor correspondiente.

Cuando mediante la etiqueta `<jsp:forward>` se transfiere el control a la página JSP correspondiente, en el navegador no se muestra la URL de la nueva página, sino que se mantiene el anterior, esta nueva página tendrá un nuevo contexto, pero hay algunos atributos que se seguirán manteniendo en la nueva página dependiendo de su ámbito.

Así por ejemplo se mantendrán los objetos (atributos) con ámbito de petición (almacenados en el objeto `request`) y con ámbito de sesión (almacenados en el objeto `session`), los objetos con ámbito de aplicación (almacenados en el objeto `application`), se mantendrán en la nueva página si esta página pertenece a la misma aplicación Web que la página de origen. Los atributos que no se comparten son los que poseen ámbito de página.

En el Código Fuente 157 se ofrece una página JSP que utiliza la acción `<jsp:forward>` para redirigir la ejecución de la página actual a la página `DESTINO.JSP` (Código Fuente 158), que recupera los parámetros de la página original, que se le pasan a través de la petición, en forma de una cadena de consulta (`QueryString`).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Acción jsp:forward</title>
</head>
<body>
<h1>Página que redirige hacia otra página</h1>
<jsp:forward page="destino.jsp"/>
</body>
</html>
```

Código Fuente 157

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
  <title>Destino</title>
```

```

</head>

<body>
<TABLE BORDER="1" ALIGN="CENTER">
<caption>Parámetros existentes</caption>
<TR BGCOLOR="#FFAD00">
<TH>Nombre parámetro</TH>
<TH>Valor parámetro</TH>
<%Enumeration nombresParametros = request.getParameterNames();
while(nombresParametros.hasMoreElements()) {
    String nombre = (String)nombresParametros.nextElement();%>
    <TR><TD><%=nombre%></TD>
    <TD><%=request.getParameter(nombre)%></TD>
<%}%>
</TABLE>
</body>
</html>

```

Código Fuente 158

El resultado de la ejecución del ejemplo anterior se puede apreciar en la Figura 76.



Figura 76. recuperando los parámetros de la página de origen

Si antes de utilizar la acción `<jsp:forward>` enviamos algún contenido del búfer del flujo de salida, por ejemplo utilizando la sentencia `out.flush()`, se producirá un error cuando se pretenda redirigir la ejecución de la página actual hacia el recurso indicado.

Si deseamos indicar algún parámetro adicional a la página destino desde la página de origen podemos utilizar otra acción estándar que tenemos disponible dentro de JSP, se trata de la acción `<jsp:param>`.

Esta acción, que comentaremos en el siguiente apartado, permite especificar parámetros que se utilizarán en las acciones `<jsp:forward>`, `<jsp:include>` y `<jsp:plugin>` para indicar información adicional en forma de pares nombre de parámetro / valor.

El ejemplo del Código Fuente 159 muestra una página JSP que redirige su ejecución a otra página JSP, a la cual le indica dos parámetros a través de la acción `<jsp:param>`.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:forward</title>
</head>

<body>
<h1>Página que redirige hacia otra página</h1>
<jsp:forward page="destino.jsp">
    <jsp:param name="ciudad" value="Madrid"/>
    <jsp:param name="domicilio" value="Oligisto 2"/>
</jsp:forward>
</body>
</html>

```

Código Fuente 159

El Código Fuente 160 muestra la página JSP que recupera los parámetros, como se puede comprobar para recuperar los parámetros se utiliza el método `getParameter()` del objeto `request`, de la misma forma que obtenemos los parámetros de una petición.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Destino</title>
</head>

<body>
<TABLE BORDER="1" ALIGN="CENTER">
<caption>Parámetros existentes</caption>
<TR BGCOLOR="#FFAD00">
<TH>Nombre parámetro</TH>
<TH>Valor parámetro</TH>
<%Enumeration nombresParametros = request.getParameterNames();
while(nombresParametros.hasMoreElements()) {
    String nombre = (String)nombresParametros.nextElement();%>
    <TR><TD><%=nombre%></TD>
    <TD><%=request.getParameter(nombre)%></TD>
<%}%>
</TABLE>
</body>
</html>

```

Código Fuente 160

El resultado de la ejecución de este ejemplo es el de la Figura 77.

A continuación tratamos la acción estándar `<jsp:param>`.



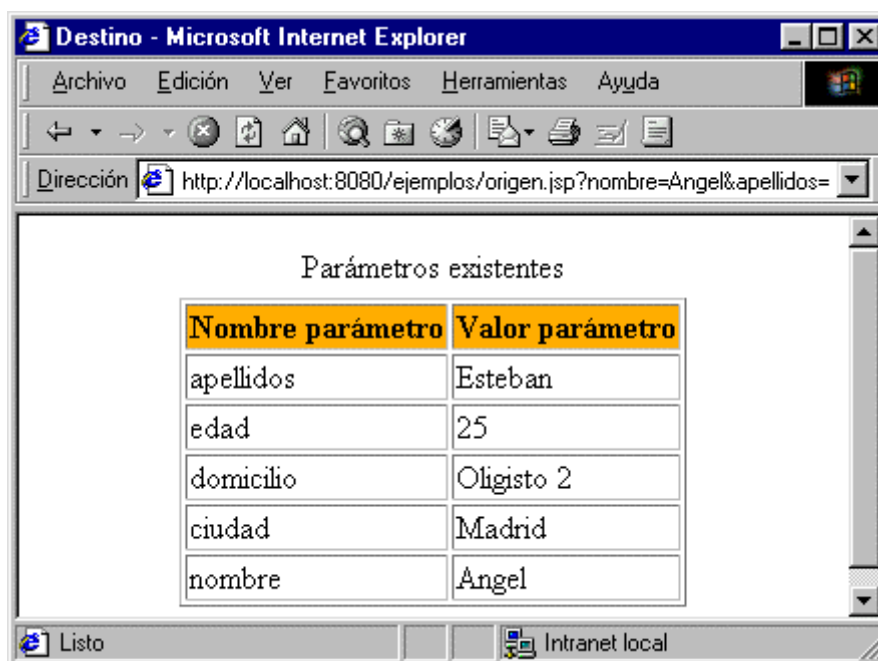


Figura 77. Recuperando los parámetros

## La acción `<jsp:param>`

Como ya hemos adelantado en el apartado anterior, esta acción se utiliza en colaboración con cualquiera de las siguientes acciones: `<jsp:forward>`, `<jsp:include>` o `<jsp:plugin>`. La sintaxis general de esta acción es la siguiente:

```
<jsp:param name="nombreParametro" value="valorParametro"/>
```

La acción `<jsp:param>` permite ofrecer información adicional a otra acción.

El Código Fuente 161 muestra una página JSP que utiliza la acción `<jsp:param>` para indicar dos parámetros a la acción `<jsp:include>`, adelantamos que esta nueva acción incluirá el contenido del recurso indicado en la página actual, formando parte de la respuesta que se devuelve al cliente.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Param e include</title>
</head>

<body>
<h3>Página que incluye a otra</h3>
<jsp:include page="incluida.jsp" flush="true">
  <jsp:param name="ciudad" value="Madrid"/>
  <jsp:param name="domicilio" value="Oligisto 2"/>
</jsp:include>
<h3>Se vuelve a la página de origen</h3>
</body>
</html>
```

Código Fuente 161

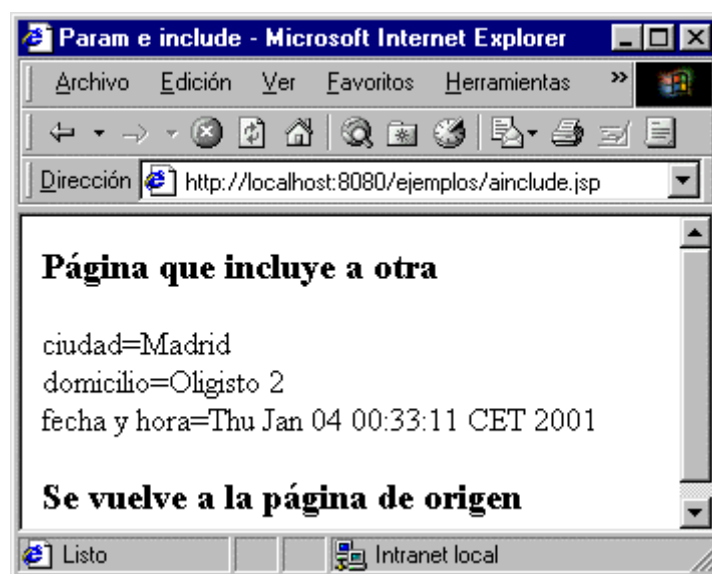
El Código Fuente 162 muestra la página JSP que se ha utilizado en la acción `<jsp:include>`, se puede comprobar que para recuperar el valor del parámetro indicado en la acción `<jsp:param>` se debe utilizar el objeto integrado `request`, y lanzar sobre él el método `getParameter()`, de la misma forma que hacíamos para obtener los parámetros que se pasaban a una petición de una página JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page import="java.util.*" %>
<html>
<head>
    <title>Página incluida</title>
</head>

<body>
    ciudad=<%=request.getParameter("ciudad")%><br>
    domicilio=<%=request.getParameter("domicilio")%><br>
    fecha y hora=<%=new Date()%>
</body>
</html>
```

Código Fuente 162

El resultado de la ejecución de este ejemplo se encuentra en la Figura 78.

Figura 78. Utilizando la acción `<jsp:param>`

Esta acción siempre deberá ir encerrada por las etiquetas de las acciones ya comentadas.

En siguiente apartado trata de la acción aquí hemos introducido, la acción `<jsp:include>`

## La acción `<jsp:include>`

La acción `<jsp:include>` permite a los autores de páginas JSP incorporar en el contenido generado por la página actual, el contenido de otro recurso distinto, resultando la salida final que se envía al usuario

en una combinación de ambos contenidos. Al contrario de lo que ocurría en la acción `<jsp:forward>`, el control de la ejecución vuelve a la página original una vez que se ha terminado la ejecución de la página incluida.

La sintaxis general de la acción `<jsp:include>` es la siguiente:

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

Como se puede comprobar de la sintaxis anterior esta acción utiliza dos atributos, por un lado la URL que identifica la página o recurso que queremos incluir en el contenido que se devuelve al cliente, y por otro lado un el atributo `flush` que indica si el búfer del flujo de salida de la página actual se vacía enviándolo al cliente antes de realizar la inclusión de la página indicada. Si el atributo `flush` tiene el valor de `false`, primero se incluirá el contenido de la página indicada, y luego el de la página original.

En el Código Fuente 163 se muestra una página JSP que incluye otra página JSP. Y en el Código Fuente 164 se muestra el código de la página que se incluye.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Acción include</title>
</head>

<body>
<h3>Se incluye el resultado de la ejecución una página</h3>
<jsp:include page="paginaJSP.jsp" flush="true"/>
<h3>Se vuelve a la página inicial</h3>
</body>
</html>
```

Código Fuente 163

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Hola</title>
</head>

<body>
<%for (int i=1;i<=7;i++){%>
  <font size="<%=i%>">Hola</font><br>
<}%>
</body>
</html>
```

Código Fuente 164

Como se puede comprobar el atributo `flush` tiene el valor `true`, y por lo tanto el resultado de la ejecución de este ejemplo será el de la Figura 79.

Si asignamos al atributo `true` el valor `false`, se producirá un error, ya que todavía no es posible utilizar esta característica en la versión actual de la especificación JavaServer Pages, JSP 1.1.

Figura 79. Utilizando la acción `<jsp:include>`

Recordamos al lector que con la acción `<jsp:include>` podemos utilizar la acción `<jsp:param>` para pasar información adicional a la página que vamos a incluir en forma de parámetros. La visibilidad en cuanto al ámbito de los atributos de la página original respecto a la incluida, es la misma que con la acción `<jsp:forward>`, es decir, son visibles los atributos con ámbito de petición y con ámbito de sesión, y los que tienen ámbito de aplicación su visibilidad deberá de si la página incluida pertenece a la misma aplicación Web que la página de origen o no.

Cuando se realiza una modificación en los recursos que se incluyen mediante la acción `<jsp:include>`, esta modificación se refleja de forma automática en el resultado de la ejecución de la página JSP que utiliza la acción. Esto es debido a que la incorporación del contenido de la página indicada en la acción `<jsp:include>` se realiza cada vez que se ejecuta la página original, es decir, el contenido de incluye en tiempo de ejecución.

Sin embargo la directiva `include` (que ya vimos en el capítulo correspondiente), no reflejará los cambios en la página incluida de forma automática, ya que el contenido de la página que se desea incluir se incorpora no en tiempo de ejecución, sino cuando se traduce la página JSP al servlet correspondiente, es decir, en tiempo de traducción.

Las ventajas que ofrece la acción `<jsp:include>` sobre la directiva `include` son la recompilación automática, las clases de los servlets son más pequeñas, ya que la página incluida se incluye en tiempo de ejecución, además se permite la utilización de parámetros adicionales mediante la acción `<jsp:param>`. Por otro lado la directiva `include` ofrece la posibilidad de utilizar las variables con ámbito de la página y ofrece una mayor eficiencia en la ejecución, ya que a la hora de ejecutarse en servlet que se corresponde con la página JSP ya tiene incluido el contenido de la página que se desea incluir.

## La acción `<jsp:plugin>`

Esta acción permite generar código HTML para asegurar que el navegador utiliza el plug-in de Java, se generarán por lo tanto las etiquetas `<OBJECT>` y `<EMBED>` correspondientes para indicar al navegador que para un applet determinado debe cargar el plug-in de Java para que el applet pueda ser ejecutado satisfactoriamente por el navegador.

Los applets que pertenecen a la versión 2 del lenguaje Java no podrán ejecutarse directamente en los navegadores Web ya que las MV (maquinas virtuales) que poseen los navegadores Netscape o Explorer no poseen la última versión del lenguaje Java, como ejemplo cabe reseñar que Internet Explorer 5 o Netscape Navigator 4.5 no permiten la ejecución de estos applets, mejor dicho, no ocurre nada cuando se ejecuta una página que invoca a uno de estos applets. La máquina virtuales de estos dos navegadores se corresponden con la versión 1.1 del lenguaje Java, es decir, no soportan Java 2.

La solución para poder utilizar applets de Java 2(Swing/JApplet) en un navegador Web consiste en instalar un software (a modo de parche) que podemos encontrar en el sitio Web de Sun y que se denomina Java Plug-in. Este añadido permite ejecutar applets implementados en Swing, es decir en la versión 2 del lenguaje Java.

El Plug-in lo podemos obtener en la dirección <http://java.sun.com/products/plugin/>, a la hora de obtener el Plug-in deberemos indicar el tipo de plataforma en el que vamos a utilizarlo. También existen varias versiones del Plug-in en nuestro caso nos interesa la última que se corresponde con el JDK 1.3, por lo tanto seleccionaremos el software Java Plug-in 1.3.

La instalación del Plug-in en un sistema operativo Windows no tiene ningún tipo de complicación, simplemente deberemos ejecutar el fichero que hemos obtenido de Sun y seguir las instrucciones correspondientes incluso el hecho de que se tenga que utilizar diferentes navegadores Internet Explorer o Netscape Navigator no va a resultar ningún problema, ya que se instala en el sistema operativo independientemente del navegador que se utilice.

Solo a la hora de crear la página Web que vaya a hacer la llamada al applet es la que va a ser diferente dependiendo del tipo de navegador.

Otra cosa importante es que a diferencia de los applets que no utilizan Swing, con este tipo de applets no vamos a utilizar la etiqueta `<APPLET>` en el código HTML en ningún caso, puesto que en realidad lo que estamos haciendo es una llamada a un componente ActiveX (proporcionado por el Java Plug-in) que se encarga de su visualización. Debido a esto la etiqueta utilizada es `<OBJECT>` o `<EMBED>` dependiendo de si el navegador Web es Internet Explorer o Netscape Navigator respectivamente.

Así por ejemplo si tenemos la clase `SwingApplet`, que es la clase que representa a un applet, si queremos incluirla en una página HTML escribiríamos el siguiente código HTML (Código Fuente 165):

```
<APPLET code="SwingApplet.class" align="baseline" width="200" height="200">
</APPLET>
```

Código Fuente 165

Pero si esta clase es un applet de Swing(versión 2 del lenguaje Java), es decir, hereda de la clase `JApplet`, deberemos escribir este otro código HTML (Código Fuente 166), suponiendo que el navegador Web que va a cargar la página es Internet Explorer.

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="200" height="200" align="baseline">
<PARAM NAME="code" VALUE="SwingApplet.class">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.3">
<PARAM NAME="scriptable" VALUE="true">
</OBJECT>
```

Código Fuente 166

Pero si el navegador Web es Netscape Navigator escribiremos este otro código HTML (Código Fuente 167):

```
<EMBED type="application/x-java-applet;version=1.3" width="200"
height="200" align="baseline" code="SwingApplet.class"
</EMBED>
```

Código Fuente 167

Y lo más recomendable es no presuponer nada sobre el navegador Web que va a ejecutar el applet, y utilizar este otro código HTML (Código Fuente 168) que es una mezcla de los dos anteriores, para que funcione el applet correctamente tanto con el navegador Web Internet Explorer como con el navegador Web Netscape Navigator.

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="200" height="200" align="baseline">
<PARAM NAME="code" VALUE="SwingApplet.class">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.3">
<PARAM NAME="scriptable" VALUE="true">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.3" width="200"
height="200" align="baseline" code="SwingApplet.class"
</EMBED>
</COMMENT>
</OBJECT>
```

Código Fuente 168

Precisamente esta alternativa es la que utiliza la acción `<jsp:plugin>` de JSP, y el código HTML que genera servirá para ambos navegadores.

La sintaxis general de esta acción es más compleja que la de las anteriores y es la que se muestra a continuación:

```
<jsp:plugin type="tipo" code="CódigoObjeto"
codebase="CodeBaseObjeto" atributo1=valor1 ...
atributo2=valor2"....>
    <jsp:params>
        <jsp:param name="nombre1" value="valor1"/>
        ...
        <jsp:param name="nombren" value="valorn"/>
    </jsp:params>
```

```
<jsp:fallback>Texto de fallo</jsp:fallback>
</jsp:plugin>
```

A continuación vamos a comentar en detalle los distintos elementos. Como se puede observar existen tres atributos obligatorios y una serie de atributos de la etiqueta `<jsp:plugin>` que son opcionales, otro elemento que podemos observar es el conjunto de parámetros, que se le pasarán al applet, y que se encuentran entre las etiquetas `<jsp:params></jsp:params>`. Por último se indica entre las etiquetas `<jsp:fallback></jsp:fallback>` un mensaje que se mostrará en el navegador en caso de no poder cargar el plug-in de Java.

A continuación vamos a comentar los atributos obligatorios de la acción `<jsp:plugin>`, como ya hemos adelantado hay tres atributos obligatorios, pero para poder mantener la compatibilidad entre distintos navegadores vamos a añadir dos atributos más, ya que algunos navegadores requieren estos atributos.

- `type`: indica el tipo de componente Java que se va a cargar en el navegador, puede tener los valores “applet” o “bean”.
- `code`: indica el fichero que contiene la clase Java que se corresponde con el componente Java.
- `codebase`: indica la URL del directorio dentro del servidor Web que contiene la clase del componente. Este atributo no será necesario si la clase del componente se encuentra en el mismo directorio que la página JSP que quiere utilizar el componente.
- `width`: es un valor entero que indica la anchura del componente, en pixels, dentro de la página.
- `height`: es un valor entero que indica la anchura del componente, en pixels, dentro de la página.

Además de estos valores obligatorios, la acción `<jsp:plugin>` soporta una serie de atributos opcionales, siendo algunos de ellos equivalentes a los atributos de la etiqueta `<applet>` de HTML. Estos atributos son los siguientes:

- `align`: cadena de texto que indica la alineación del componente Java respecto a otros elementos de la página, los valores permitidos son left, right, top, texttop, middle, absmiddle, baseline, bottom y absbottom. El valor por defecto es baseline.
- `name`: cadena de texto que indica el nombre mediante el cual otros componentes en la misma página pueden hacer referencia al componente. Por defecto este atributo no posee ningún valor.
- `archive`: cadena de texto que contiene una lista de ficheros JAR (Java archive) separados por comas, que indican una serie de recursos adicionales que se necesitan cargar. Este atributo no tiene un valor por defecto.
- `vspace`: valor entero que indica el margen, en pixels, que debe aparecer a la izquierda y a la derecha del componente dentro de la ventana del navegador. Por defecto el valor que presenta este atributo es 0.
- `hspace`: valor entero que indica el margen, en pixels, que debe aparecer arriba y abajo del componente dentro de la ventana del navegador. Por defecto el valor que presenta este atributo es 0.

- `jreversion`: indica la versión del JRE (Java Runtime Environment) necesaria para ejecutar el componente Java. Este atributo y los siguientes ya no se corresponden con atributos de la etiqueta `<applet>` de HTML, sino que son propios de la acción `<jsp:plugin>`. El valor por defecto es 1.1
- `nsplugin`: URL que indica la localización de la que se puede descargar el plug-in de Java para el navegador Web Netscape Communicator.
- `ieplugin`: URL que indica la localización de la que se puede descargar el plug-in de Java para el navegador Web Microsoft Internet Explorer.

En el Código Fuente 169 se muestra una página JSP que genera el código HTML necesario, mediante la acción `<jsp:plugin>`, para cargar en el navegador un applet que se encuentra representado por la clase `Reloj`, este sencillo applet muestra la hora actual del sistema.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Acción jsp:plugin</title>
</head>
<body>
<jsp:plugin type="applet" code="RelojApplet2" height="20"
width="100" jreversion="1.1">
<jsp:fallback>
No es posible cargar el plug-in de Java
</jsp:fallback>
</jsp:plugin>
</body>
</html>
```

Código Fuente 169

Si se ejecuta esta ejemplo, antes de aparecer el applet en el navegador aparecerá un mensaje del tipo “cargando applet de Java”, este mensaje es mostrado por el plug-in de Java para indicarnos que va a ejecutar el applet.

El resultado de la ejecución del ejemplo anterior es el de la Figura 80.

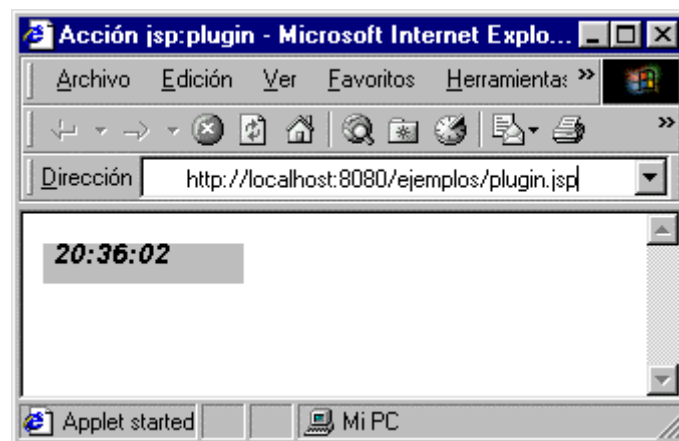


Figura 80. Utilización de la acción `<jsp:plugin>`



Y el código HTML que se genera es el que se muestra en el Código Fuente 170.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Acción jsp:plugin</title>
</head>
<body>
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" width="100"
height="20" codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
<PARAM name="java_code" value="RelojApplet2">
<PARAM name="type" value="application/x-java-applet;version=1.1">
<COMMENT>
<EMBED type="application/x-java-applet;version=1.1" width="100" height="20"
pluginspage="http://java.sun.com/products/plugin/" java_code="RelojApplet2" >
<NOEMBED>
</COMMENT>
No es posible cargar el plug-in de Java

</NOEMBED></EMBED>
</OBJECT>

</body>
</html>
```

Código Fuente 170

La acción `<jsp:plugin>` también permite indicar parámetros al componente Java que se cargará en el navegador, para ello se deben incluir las etiquetas `<jsp:param>` que sean necesarias, una para cada parámetro, dentro de las etiquetas `<jsp:params></jsp:params>`. La sintaxis general para indicar parámetros a un componente Java es la que se muestra a continuación.

```
<jsp:plugin...>
  .....
  <jsp:params>
    <jsp:param name="nombre1" value="valor1"/>
    ...
    <jsp:param name="nombren" value="valorn"/>
  </jsp:params>
  ...
</jsp:plugin>
```

En el caso de los applets estos parámetros se obtendrán mediante el método `getParameter()` de la clase `java.applet.Applet`, todos los valores de los parámetros se interpretarán como cadenas de caracteres.

En el caso de los componentes JavaBeans (que veremos con detenimiento en el capítulo siguiente), los nombres de los parámetros deben coincidir con el nombre de las propiedades del Bean correspondiente.

Si el navegador no puede cargar el plug-in de Java para ejecutar el componente correspondiente indicado en la acción `<jsp:plugin>`, se mostrará el texto que se corresponde con el mensaje dentro de las etiquetas `<jsp:fallback></jsp:fallback>`.

Ya hemos visto cuatro de las siete acciones estándar que nos ofrece la especificación `JavaServer Pages`, las tres restantes son relativas a la utilización de componentes JavaBeans dentro de la página

JSP. Más adelante, en siguientes capítulos, veremos con detalle los componentes JavaBeans y como podemos utilizarlos en nuestras páginas JSP, por lo que vamos a realizar una explicación breve de estas acciones, ya que más adelante las retomaremos para utilizar los componentes JavaBeans dentro de las páginas JSP.

## La acción `<jsp:useBean>`

Esta acción se utiliza para poder utilizar dentro de una página JSP un componente JavaBean en un ámbito determinado. El componente JavaBean podrá ser utilizado dentro de la página JSP haciendo referencia al nombre indicado dentro de la acción `<jsp:useBean>`, teniendo siempre en cuenta el ámbito al que pertenece el Bean, y que se indica también en la acción `<jsp:useBean>`.

Para definir de forma sencilla lo que es JavaBeans, podemos decir que se trata de una especificación que permite escribir componentes software en Java. Los componentes que siguen la arquitectura descrita por JavaBeans se denominan Beans o incluso JavaBeans o Beans de Java. Para que una clase determinada pueda ser considerada un Bean debe seguir una serie de especificaciones descritas en el API JavaBeans.

La arquitectura de componentes JavaBeans y la forma de construir los componentes lo veremos en el siguiente capítulo, en el presente capítulo únicamente vamos a comentar las tres acciones que nos ofrece la especificación JavaServer Pages para poder hacer uso de los componentes JavaBeans dentro de nuestras páginas JSP.

La acción `<jsp:useBean>` indica a la página JSP que deseamos tener un Bean determinado disponible, el contenedor de páginas JSP creará el Bean correspondiente o bien lo recuperará del ámbito correspondiente si éste ya existe

La sintaxis básica de esta acción es la siguiente:

```
<jsp:useBean id="nombre" class="nombreClase" />
```

También tenemos esta otra posibilidad:

```
<jsp:useBean id="nombre" class="nombreClase" />  
código de inicialización  
</jsp:useBean>
```

Es decir podemos utilizar la sintaxis de una única línea, o bien la de varias líneas indicando un código de inicialización que queramos que se ejecute, este código de inicialización sólo se ejecutará si se crea el Bean.

A continuación comentamos los distintos atributos de la acción `<jsp:useBean>`.

- `id`: es el identificador que vamos a utilizar dentro de la página JSP, y durante el resto del ciclo de vida del Bean para hacer referencia al Bean que se crea o se utiliza. Se puede elegir cualquier nombre para hacer referencia a un Bean, aunque se deben seguir una serie de normas: este identificador debe ser único en la página, se distingue entre mayúsculas y minúsculas, el primer carácter debes ser una letra, sólo se permiten letras, números y carácter de subrayado (`_`), no se permiten por lo tanto espacios.
- `class`: en este atributo se indica el nombre de la clase del Bean, a cada Bean le va a corresponder una clase, al igual que sucede con los applets, que poseen una clase principal que representa la clase del componente, para organizar los componentes JavaBeans estas clases

suelen encontrarse en paquetes, y si no los hemos importado en la directiva page, deberemos utilizar el nombre completo de la clase del Bean, indicando paquetes y subpaquetes.

- **type**: este atributo no suele ser utilizado, es opcional e indica el tipo de la clase del Bean, este tipo suele ser la clase padre, un interfaz o la propia clase, se utilizará para casos muy concretos cuando queremos realizar una casting de clases. Por defecto tiene el mismo valor que el indicado atributo **class**.
- **scope**: indica el ámbito que le va a corresponder al Bean, ya sabemos que existen cuatro ámbitos distintos, y por lo tanto este atributo podrá tener los valores **page**, **request**, **session** o **application**, por defecto se utiliza el ámbito de página (**page**).
- **beanName**: es el nombre del Bean que se le pasa al método **instantiate()** de la clase **java.beans.Beans**, es posible indicar el atributo **type** y el atributo **beanName** y omitir el atributo **class**.

En el Código Fuente 171 se muestra un sencillo ejemplo de una página JSP que crear un Bean con ámbito de página. Como se puede observar se ha utilizado la clase **java.util.Date** como clase del Bean, lo normal es utilizar un componente JavaBeans pero para este ejemplo la clase **Date** nos sirve perfectamente para mostrar como más tarde se puede acceder en la página al Bean que se ha creado utilizando su identificador.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Acción jsp:useBean</title>
</head>

<body>
<jsp:useBean id="fecha" scope="page" class="java.util.Date"/>
<%=fecha%>
</body>
</html>
```

Código Fuente 171

El resultado de la ejecución de esta página se puede ver en la Figura 81.

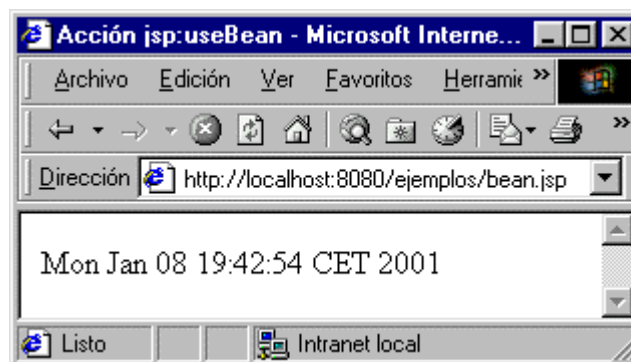


Figura 81. Utilizando la acción `<jsp:useBean>`

En el Código Fuente 172 se muestra un ejemplo más convencional de utilización de la acción `<jsp:useBean>`, en este nuevo ejemplo se crea un componente JavaBeans cuya clase es `HolaBean`, y se crea con ámbito de sesión, por lo tanto este Bean será accesible por el resto de páginas JSP que compartan esta misma sesión.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:useBean</title>
</head>

<body>
<jsp:useBean id="miBean" scope="session" class="HolaBean">
    <%out.println("<h1>Se ha creado el Bean</h1>");%>
</jsp:useBean>
<h1>Hola <%=miBean.getNombre()%></h1>
<%miBean.setNombre("Angel");%>
</body>
</html>
```

Código Fuente 172

Para acceder desde otra página al Bean que hemos creado con ámbito de sesión únicamente se debe utilizar la misma acción `<jsp:useBean>` que hemos utilizado para crear el Bean. En el Código Fuente 173 se muestra una página JSP que accede a este Bean que se supone que ya hemos creado en la página anterior, si el Bean no estuviera creado se instanciaría en esta segunda página.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:useBean II</title>
</head>

<body>
<jsp:useBean id="miBean" scope="session" class="HolaBean">
    <%out.println("<h1>Se ha creado el Bean</h1>");%>
</jsp:useBean>
<h1>Hola <%=miBean.getNombre()%></h1>
</body>
</html>
```

Código Fuente 173

Al ejecutar la primera página obtenemos el resultado de la Figura 82, y al ejecutar a segunda página obtenemos el resultado de la Figura 83.

En el Código Fuente 174 se muestra un fragmento de código que se genera en el servlet que se corresponde con la página JSP a partir de la etiqueta `<jsp:useBean>`.

En este código se puede comprobar como se intenta recuperar el Bean indicado en el ámbito indicado, en este caso es el ámbito de sesión, y el identificador del Bean es `miBean`, siendo su clase `HolaBean`.

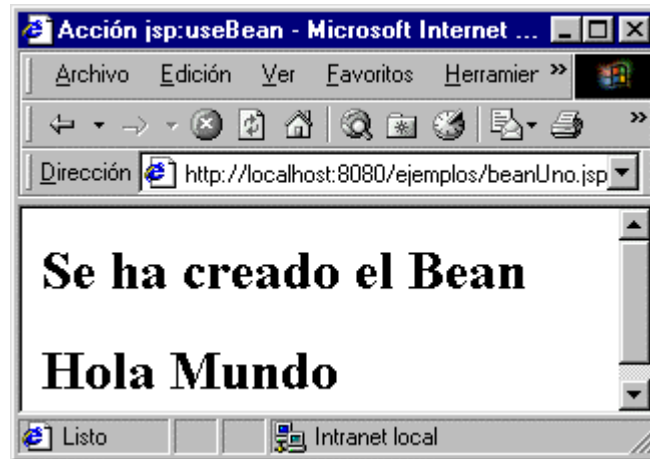


Figura 82. Creando un Bean de sesión.



Figura 83. Utilizando un Bean de sesión

```
// begin [file="C:\\beanUno.jsp";from=(8,0);to=(8,58)]
HolaBean miBean = null;
boolean _jspx_specialmiBean = false;
synchronized (session) {
    miBean= (HolaBean)
    pageContext.getAttribute("miBean",PageContext.SESSION_SCOPE);
    if ( miBean == null ) {
        _jspx_specialmiBean = true;
        try {
            miBean = (HolaBean) Beans.instantiate(getClassLoader(), "HolaBean");
        } catch (Exception exc) {
            throw new ServletException (" Cannot create bean of class "+ "HolaBean");
        }
        pageContext.setAttribute("miBean", miBean, PageContext.SESSION_SCOPE);
    }
}
if(_jspx_specialmiBean == true) {
    // end
    // HTML // begin [file="C:\\beanUno.jsp";from=(8,58);to=(9,1)]
    out.write("\r\n\t");
    // end
    // begin [file="C:\\beanUno.jsp";from=(9,3);to=(9,48)]
    out.println("<h1>Se ha creado el Bean</h1>");
    // end
    // HTML // begin [file="C:\\beanUno.jsp";from=(8,58);to=(9,1)]
    out.write("\r\n");
}
```

```
// end
// begin [file="C:\\beanUno.jsp";from=(8,0);to=(8,58)]
}
// end
```

Código Fuente 174

A continuación vamos a comentar con detalle los pasos que tienen lugar cuando el contenedor de página JSP encuentra una acción `<jsp:useBean>`.

1. El contenedor trata de localizar un objeto que posea el identificador y ámbito indicados en la acción `<jsp:useBean>`.
2. Si se encuentra el objeto, y se ha especificado un atributo `type`, el contenedor trata de utilizar la conversión de tipos del objeto encontrado con el tipo (`type`) especificado, si la conversión falla se lanzará una excepción `java.lang.ClassCastException`. Si únicamente se ha indicado un atributo `class` se creará una nueva referencia al objeto en el ámbito indicado utilizando el identificador correspondiente.
3. Si el objeto no se encuentra en el ámbito especificado y no se ha indicado un atributo `class` o `beanName`, se lanzará una excepción `java.lang.InstantiationException`.
4. Si el objeto no se ha encontrado en el ámbito indicado, y se ha especificado una clase no abstracta con un constructor público sin argumentos, se instanciará un objeto de esa clase. Un nuevo objeto se asociará con el identificador y ámbitos indicados. Si no se da alguna de las condiciones comentadas, se lanzará una excepción `InstantiationException`.
5. Si el objeto no se localiza en el ámbito indicado, y se ha especificado un atributo `beanName`, se invocará al método `instantiate()` de la clase `java.beans.Beans`, pasándole como parámetro el valor de la propiedad `beanName`. Si el método tiene éxito un nuevo objeto se asociará con el identificador y ámbitos indicados.
6. Si la acción `<jsp:useBean>` no posee un cuerpo vacío, se procesará el cuerpo de la etiqueta, únicamente si el objeto se crea como un objeto nuevo, es decir, no se había encontrado en el ámbito indicado.

Como se puede ver en los ejemplos anteriores, para acceder a una propiedad del Bean utilizamos los métodos de acceso correspondientes que ofrecen la clase del Bean, esta no es la forma usual de acceder a las propiedades de un `JavaBean` para obtener o establecer los valores de sus propiedades, sino que desde una página JSP utilizaremos un par de acciones para realizar estas tareas. Se trata de las acciones `<jsp:getProperty>` y `<jsp:setProperty>`, que permiten obtener el valor de una propiedad de un Bean y establecer el valor de una propiedad de un Bean, respectivamente. Estas dos acciones las veremos en los siguientes apartados.

## La acción `<jsp:getProperty>`

Esta nueva acción forma parte de las acciones que nos permiten utilizar componentes `JavaBeans` dentro de nuestras páginas JSP, en este caso la acción `<jsp:getProperty>` nos va a permitir obtener el valor de la propiedad de un `JavaBean` creado en la página con el ámbito correspondiente.

La sintaxis de esta acción es muy sencilla, no posee cuerpo y únicamente presenta dos atributos o propiedades, como se puede observar a continuación.

```
<jsp:getProperty name="nombreBean" property="nombrePropiedad" />
```

La propiedad name indica el identificador del Bean que hemos creado con la acción <jsp:useBean>, y cuyo valor de la propiedad queremos obtener. Se corresponderá con el valor del atributo id de la acción <jsp:useBean> correspondiente.

El atributo property indica el nombre de la propiedad del Bean cuyo valor se desea obtener. El valor de la propiedad se mostrará como código HTML, reemplazando en tiempo de ejecución a la acción <jsp:getProperty> correspondiente.

Esta acción accede al valor de la propiedad especificada del Bean correspondiente, la convierte en un objeto de la clase String y la imprime en el flujo de salida de la página JSP.

Utilizando esta nueva acción podríamos rescribir alguno de los ejemplos anteriores para reemplazar el uso de los métodos de la clase del Bean por la utilización de la acción <jsp:getProperty> para obtener el valor de la propiedad del Bean. La nueva versión de esta página JSP se puede observar en el Código Fuente 175.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
  <title>Acción jsp:getProperty</title>
</head>

<body>
<jsp:useBean id="miBean" scope="session" class="HolaBean">
  <%out.println("<h1>Se ha creado el Bean</h1>");%>
</jsp:useBean>
<h1>Hola <jsp:getProperty name="miBean" property="nombre"/></h1>
<%miBean.setNombre("Angel");%>
</body>
</html>
```

Código Fuente 175

El resultado de la ejecución de esta página es idéntico al de la versión anterior, ya que según veremos en el siguiente capítulo existe una estrecha relación entre los nombre de los métodos de acceso de las clases de los componentes JavaBeans y las propiedades que poseen los mismos. En el Código Fuente 176 se muestra un fragmento del servlet resultante que se generaría a partir de la página JSP anterior, se puede comprobar el código que se sustituye por la acción <jsp:getProperty>.

```
// HTML // begin [file="C:\\beanUno.jsp";from=(10,14);to=(11,9)]
out.write("\r\n<h1>Hola ");
// end
// begin [file="C:\\beanUno.jsp";from=(11,9);to=(11,59)]

out.print(JspRuntimeLibrary.toString(((HolaBean)pageContext.findAttribute
                                     ("miBean")).getNombre()));
// end
// HTML // begin [file="C:\\beanUno.jsp";from=(11,59);to=(12,0)]
out.write("</h1>\r\n");
// end
```

Código Fuente 176

Podemos utilizar la acción `<jsp:getProperty>` antes de utilizar la acción `<jsp:useBean>`, así por ejemplo el Código Fuente 177 sería completamente correcto.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:getProperty </title>
</head>

<body>
<h1>Hola <jsp:getProperty name="miBean" property="nombre"/></h1>
</body>
</html>
<jsp:useBean id="miBean" scope="session" class="HolaBean">
    <%out.println("<h1>Se ha creado el Bean</h1>");%>
</jsp:useBean>
```

Código Fuente 177

Si la propiedad que indicamos en esta acción no se corresponde con ninguna propiedad del Bean indicado se producirá un error.

En el siguiente y último apartado vamos a tratar la acción complementaria, que nos permite establecer el valor de las propiedades de un componente JavaBean, es decir, la acción `<jsp:setProperty>`.

## La acción `<jsp:setProperty>`

Esta acción permite modificar las propiedades de los Beans a los que hacemos referencia en nuestras páginas JSP a través de la acción `<jsp:useBean>`, es la acción complementaria a la acción `<jsp:getProperty>`. Su sintaxis general es la que se muestra a continuación:

```
<jsp:setProperty name="nombreBean" detallesPropiedad/>
```

El atributo `name` tiene el mismo significado que en la acción vista anteriormente, es decir, es el identificador del componente JavaBean al que se hace referencia en la página.

Los detalles de la propiedad son una serie de atributos que combinados entre sí permiten asignar el valor a la propiedad del Bean de distinta forma. Así por ejemplo la forma de establecer el valor de la propiedad de un Bean puede ser cualquiera de las que aparecen a continuación:

- `property=" * "`
- `property="nombrePropiedad"`
- `property="nombrePropiedad" param="nombreParámetro"`
- `property="nombrePropiedad" value="valorPropiedad"`

El valor de una propiedad de un Bean se pueden establecer a partir de varios elementos:

- En el tiempo de la petición de la página a partir de los parámetros existentes en el objeto integrado request.



- En el tiempo de ejecución de la página a partir de la evaluación de una expresión válida de JSP.
- A partir de una cadena de caracteres indicada o como una constante en la propia página.

A continuación se comentan todos los atributos de la acción `<jsp:setProperty>`:

- **name:** nombre o identificador del Bean que se ha instanciado mediante la acción `<jsp:useBean>`.
- **property:** el nombre de la propiedad del Bean cuyo valor se desea establecer. Este atributo puede tener asignado un valor especial que el asterisco ("\*"). Si indicamos el asterisco, de forma automática la etiqueta iterará sobre todos los parámetros del objeto request correspondiente estableciendo los nombres de las propiedades del Bean que se coincidan con el nombre de los parámetros del objeto request, asignándole el valor del parámetro cuando se de dicha coincidencia. Si un parámetro del objeto request posee el valor de vacío ("") no se modificará el valor de la propiedad del Bean. Con el asterisco podemos establecer el valor de varias propiedades del Bean de una sola vez, más adelante lo veremos mediante un ejemplo.
- **param:** este atributo permite indica el nombre del parámetro del objeto request que se va a utilizar para establecer el valor de la propiedad del Bean indicadas en el atributo property. Gracias a este atributo no es necesario que el Bean posea el mismo nombre de propiedad que el parámetro del objeto request cuyo valor deseamos establecer para la propiedad. SI no se especifica el atributo param se asume que el nombre de la propiedad y el nombre del parámetro del objeto request es el mismo.
- **value:** contiene el valor que se va a asignar a la propiedad, puede ser una cadena o una expresión válida. Una acción `<jsp:setProperty>` no puede presentar los atributos value y param al mismo tiempo.

A continuación vamos a ver la distinta utilización de esta acción a través de distintos ejemplos. Suponemos que tenemos un Bean de la clase `PersonaBean` que tiene las propiedades nombre, apellidos y edad.

En el ejemplo del Código Fuente 178 se muestra una página JSP que establece el valor de todas las propiedades del Bean de la clase `PersonaBean` a partir de los valores que se indican en los parámetros del objeto integrado request.

Por lo tanto los parámetros del objeto request y las propiedades del Bean debe tener el mismo nombre, si uno de los nombres no coincidiera, la propiedad correspondiente no se podría establecer con el valor del parámetro.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:setProperty</title>
</head>

<body>
<jsp:useBean id="miBean" scope="page" class="PersonaBean"/>
<jsp:setProperty name="miBean" property="*" />
Nombre:<b><jsp:getProperty name="miBean" property="nombre"/></b><br>
Apellidos:<b><jsp:getProperty name="miBean" property="apellidos"/></b><br>
Edad:<b><jsp:getProperty name="miBean" property="edad"/></b>
```

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

Código Fuente 178

En Código Fuente 179 se muestra un fragmento del código que se genera para el servlet equivalente a la página JSP.

```
JspRuntimeLibrary.introspect(pageContext.findAttribute("miBean"), request);
// end
// HTML // begin [file="C:\\setPropertyUno.jsp";from=(9,45);to=(10,10)]
out.write("\r\nNombre:<b>");
// end
// begin [file="C:\\setPropertyUno.jsp";from=(10,10);to=(10,60)]
out.print(JspRuntimeLibrary.toString(((PersonaBean)pageContext.findAttribute("miBean")).getNombre()));
// end
// HTML // begin [file="C:\\setPropertyUno.jsp";from=(10,60);to=(11,13)]
out.write("</b><br>\r\nApellidos:<b>");
// end
// begin [file="C:\\setPropertyUno.jsp";from=(11,13);to=(11,66)]
out.print(JspRuntimeLibrary.toString(((PersonaBean)pageContext.findAttribute("miBean")).getApellidos()));
// end
// HTML // begin [file="C:\\setPropertyUno.jsp";from=(11,66);to=(12,8)]
out.write("</b><br>\r\nEdad:<b>");
// end
// begin [file="C:\\setPropertyUno.jsp";from=(12,8);to=(12,56)]
out.print(JspRuntimeLibrary.toString(((PersonaBean)pageContext.findAttribute("miBean")).getEdad()));
// end
// HTML // begin [file="C:\\setPropertyUno.jsp";from=(12,56);to=(16,0)]
out.write("</b>\r\n\r\n</body>\r\n</html>\r\n");
```

Código Fuente 179

En la Figura 84 se muestra un ejemplo de ejecución de esta página JSP.

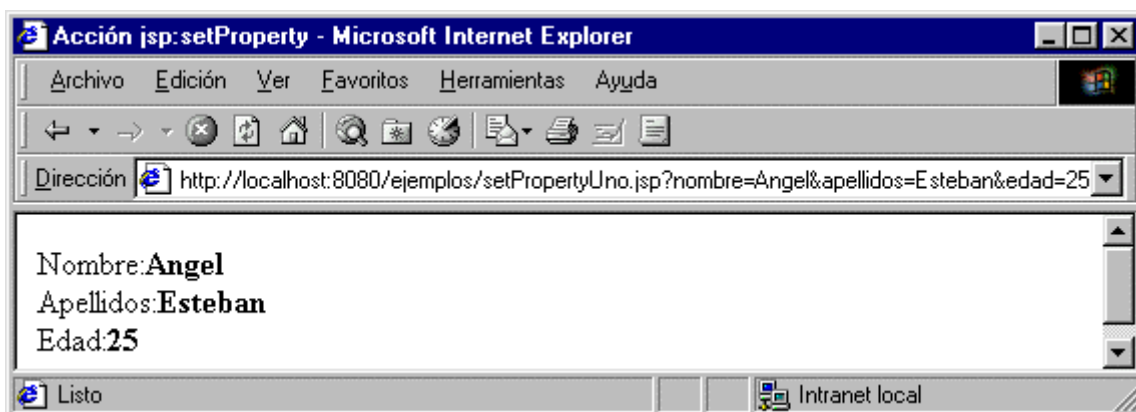


Figura 84. Estableciendo los valores de las propiedades a partir del objeto request

Pero si uno de los nombres de los parámetros del objeto request no coincide, o no existe, el valor de esa propiedad se quedará sin modificar. Como se puede comprobar en la Figura 85.



Figura 85. Parámetros y propiedades distintas.

La forma de solucionar el problema anterior es indicar mediante el atributo `param` las correspondencias que deseemos entre la propiedad del Bean y el parámetro del objeto request correspondiente. El Código Fuente 180 muestra la utilización del atributo `param`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Acción jsp:setProperty II</title>
</head>
<body>
<jsp:useBean id="miBean" scope="page" class="PersonaBean"/>
<jsp:setProperty name="miBean" property="*/>
<jsp:setProperty name="miBean" property="nombre" param="nom"/>
Nombre:<b><jsp:getProperty name="miBean" property="nombre"/></b><br>
Apellidos:<b><jsp:getProperty name="miBean" property="apellidos"/></b><br>
Edad:<b><jsp:getProperty name="miBean" property="edad"/></b>
</body>
</html>
```

Código Fuente 180

El código de la siguiente página JSP (Código Fuente 181) es completamente equivalente al código inicial, se establece el valor de la propiedad mediante una expresión que se asigna al atributo `value` de la acción `<jsp:setProperty>`, recuperándole valor del parámetro mediante el método `getParameter()` del objeto integrado request, esta forma de asignar los valores a las propiedades de un Bean no se suele utilizar, ya que para ello disponemos de opciones más sencillas como hemos visto antes.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Acción jsp:setProperty II</title>
</head>
<body>
<%String valor=request.getParameter("nombre");%>
<jsp:useBean id="miBean" scope="page" class="PersonaBean"/>
<jsp:setProperty name="miBean" property="*/>
<jsp:setProperty name="miBean" property="nombre" value="<%=valor%>"/>
Nombre:<b><jsp:getProperty name="miBean" property="nombre"/></b><br>
```

```
Apellidos:<b><jsp:getProperty name="miBean" property="apellidos"/></b><br>
Edad:<b><jsp:getProperty name="miBean" property="edad"/></b>

</body>
</html>
```

Código Fuente 181

Con esta acción finalizamos este capítulo, y el ciclo de capítulos dedicados a los distintos elementos que podemos utilizar y encontrar dentro de una página JSP. En este capítulo hemos introducido un concepto muy importante dentro del diseño de páginas JSP, la utilización de los componentes JavaBeans, que nos permiten crear aplicaciones Web en las que podremos separar la presentación de la implementación.

En el siguiente capítulo profundizaremos en la utilización de los componentes JavaBeans desde nuestras páginas JSP, y también veremos como crear nuestros propios componentes.

# JSP y componentes JavaBeans

---

## Introducción

En los capítulos anteriores hemos estado viendo todos los elementos que podemos encontrar dentro de una página JSP, y como podemos utilizar estos elementos, y en el capítulo anterior comentamos los mecanismos que nos ofrecía JSP para utilizar componentes JavaBeans mediante sus acciones estándar `<jsp:useBean>`, `<jsp:getProperty>` y `<jsp:setProperty>`.

En este capítulo, además de profundizar un poco más en las acciones JSP anteriores para utilizar los componentes JavaBeans, veremos en que consisten los componentes JavaBeans, y en el siguiente capítulo veremos como construir nuestros propios componentes JavaBeans para poder utilizarlos desde nuestras páginas JSP, debemos tener en cuenta que JavaBeans es el modelo de componentes que nos va a permitir separar la lógica de la implementación de nuestras páginas de la lógica de la presentación.

Los detalles de la implementación los eliminaremos de las páginas JSP y los pasaremos a los componentes JavaBeans, y las páginas JSP se encargarán de los detalles de presentación y de la interacción del usuario con la aplicación Web.

Además desde las páginas JSP accederemos a los Beans de forma sencilla y estandarizada, a través de las acciones estándar de JSP ya conocidas por todos.

Los componentes JavaBeans utilizados dentro de las páginas JSP nos permite realizar la división clara del trabajo entre diseñadores Web y desarrolladores Java. Los primeros se encargarán de diseñar y desarrollar las páginas JSP que utilizarán los Beans a través de las acciones, y los segundos se encargarán de desarrollar los componentes JavaBeans necesarios, utilizando los distintos APIs que ofrece el lenguaje Java, de esta forma se abstrae al diseñador Web de la implementación. Aunque

muchas veces en proyectos pequeños o medianos la figura del diseñador Web y el desarrollador Java la encontramos en una misma persona.

La utilización de la arquitectura de componentes JavaBeans es uno de los puntos más fuertes de la especificación JSP. La especificación JavaBeans permite desarrollar componentes en el lenguaje Java, los cuales encapsulan la lógica de la aplicación Web y trasladan todo el código fuente de la implementación de los scriptlets de las páginas JSP a los componentes. El resultado son unas páginas JSP más sencillas y simples, más fáciles de mantener y más accesibles para los diseñadores Web que no son desarrolladores.

JavaBeans es la respuesta de Sun a la creciente demanda de la industria del software de un conjunto de estándares que definen los componentes software. Los componentes se utilizan para simplificar la distribución, ampliaciones y mantenimiento. Un cambio mínimo en la funcionalidad de un sistema no supone un gran esfuerzo gracias a la arquitectura basada en componentes, cada componente es una caja negra que nos ofrece una funcionalidad específica y que además es reutilizable.

## Componentes

Los componentes son elementos software con entidad propia que son reutilizables y encapsulan el comportamiento de una aplicación o el acceso a sus datos en paquetes discretos. Podemos considerar que los componentes son como cajas negras que realizan operaciones específicas sin revelar los detalles de lo que está ocurriendo realmente. Los componentes abstraen su comportamiento de su implementación ocultando todos los detalles.

Los componentes son independientes y no se encuentran asociados a una única aplicación o uso. Esto permite que utilicemos los componentes para construir bloques de múltiples componentes que en un principio no tenían una relación entre sí.

Abstracción y reusabilidad son los dos conceptos principales del desarrollo de aplicaciones basadas en componentes, de esta forma a partir de una colección de componentes software independientes los ensamblaremos para formar una solución completa. Podemos pensar en los componentes como elementos de software reutilizables que podemos unir para dar lugar a una aplicación completa.

Existe una completa teoría en torno al desarrollo de aplicaciones basadas en arquitecturas de componentes software, nosotros en nuestro texto no vamos a entrar en más detalles teóricos entorno a los componentes, sino que vamos a empezar a comentar el modelo concreto que nos interesa, que no es otro que la arquitectura (o especificación) de componentes JavaBeans.

En el siguiente capítulo veremos las normas que define la especificación JavaBeans para desarrollar componentes, también desarrollaremos nuestros Beans para luego utilizarlos desde páginas JSP dentro de una aplicación Web.

## ¿Qué es un JavaBean/Bean/Bean de Java?

Para definir de forma sencilla lo que son los componentes JavaBeans, podemos decir que JavaBean se trata de una especificación que permite escribir componentes software en Java. Los componentes que siguen la arquitectura descrita por JavaBeans se denominan Beans o incluso JavaBeans o Beans de Java.

Para que una clase determinada pueda ser considerada un Bean debe seguir una serie de especificaciones descritas en el API JavaBeans. Por lo tanto un Bean va a ser una clase más de Java,

pero que debe seguir una serie de convenciones indicadas por la especificación JavaBeans, estas convenciones las veremos en el próximo capítulo, que es la continuación del actual, ya que en el actual vamos a comentar una serie de fundamentos sobre los componentes JavaBeans y también como podemos utilizarlos desde nuestras páginas JSP, aunque en este aspecto ya se ha adelantado bastante en el capítulo anterior.

Para que el lector se vaya familiarizando con los componentes JavaBeans, en el Código Fuente 182 se muestra el código de la clase de un Bean que utilizamos en el capítulo anterior para mostrar el uso de las acciones JSP relacionadas con los Beans, es decir, el Bean cuya clase era PersonaBean.

```
public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;

    public PersonaBean() {
        this.nombre="";
        this.apellidos="";
        this.edad=0;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre=nombre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos=apellidos;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad=edad;
    }
}
```

Código Fuente 182

Como se puede comprobar a simple vista es una clase normal del lenguaje Java, lo único que la hace especial es que sigue las normas de la especificación JavaBeans.

Se puede distinguir tres categorías de Beans:

- Componentes visuales (visual component Beans): estos Beans se utilizan como elementos dentro de interfaces de usuario gráficos (GUI, Graphical User Interface).
- Beans de datos (data Beans): estos componentes ofrecen acceso a una serie de información que puede estar almacenada en tablas de una base de datos.

- Beans de servicios (service Beans): también conocidos como Beans trabajadores (worker Beans), en este caso se encuentran especializados en la realización de una serie de tareas específicas o cálculos.

Esta separación o distinción de categorías de los Beans no es rígida, podemos tener Beans que pertenezcan a más de una de las categorías mencionadas. Desde las páginas JSP sólo podremos hacer uso de Beans que pertenecen a la categoría de datos o de servicios, pero no a los que pertenecen a la categoría de componentes visuales, debemos tener en cuenta que nos movemos siempre en el entorno del navegador Web.

El API JavaBeans ofrece un formato estándar para las clases Java. Las clases que quieran ser consideradas y tratadas como Beans deben seguir las reglas definidas por el API JavaBeans, como ya hemos dicho estas reglas que deben cumplir las clases de los Beans las veremos más adelante.

Al igual que ocurre con muchos componentes software los Beans encapsulan tanto su estado como comportamiento. Ya hemos visto que el aspecto que presenta la implementación un Bean es como el de una clase típica de Java, lo único que esta clase debe seguir las especificaciones que indica JavaBeans.

Los Beans los podemos utilizar tanto desde los servlets como desde las páginas JSP, aunque la forma de utilizarlos es distinta. Desde un servlet utilizaremos un Bean de la misma forma que utilizamos una instancia de una clase cualquiera de Java, sin embargo desde las páginas JSP, los Beans los utilizamos a través de varias acciones de la especificación JSP 1.1, que ya vimos en el capítulo anterior.

Mediante estas acciones podremos instanciar Beans, modificar las propiedades de los Beans creados y obtener también los valores de las mismas. La forma de hacer unos de componentes JavaBeans dentro de una página JSP es distinta a la de un servlet, debido a que una página JSP es un contenedor Bean, más adelante veremos que es exactamente un contenedor Bean.

## Fundamentos de JavaBeans

En el apartado anterior hemos realizado una primera aproximación a los que son los componentes JavaBeans y la especificación JavaBeans, también hemos tenido nuestra primera toma de contacto con la implementación de un Bean determinado. En este nuevo apartado vamos a comentar una serie de conceptos básicos que se encuentran relacionados con el modelo de componentes JavaBeans.

### Contenedores de Beans

El primero de estos conceptos, ya lo hemos nombrado en el apartado anterior, se trata de los contenedores Bean. Un contenedor Bean (Bean container) es una aplicación, entorno o lenguaje de programación que permite a los desarrolladores instanciar Beans, realizar llamadas a los mismos, configurarlos y acceder a su información y comportamiento.

Los contenedores Beans permiten a los desarrolladores trabajar con los Beans a un nivel conceptual mayor que el que se obtendría accediendo a los Beans desde una aplicación Java, ya que desde una aplicación típica de Java utilizaremos los Beans como si objetos de cualquier clase Java se trataran, no tienen un tratamiento especial. Esto es posible debido a que los Beans exponen sus características (atributos o propiedades) y comportamientos (métodos) al contenedor de Beans, permitiendo al desarrollador trabajar con los Beans de una manera más intuitiva.



Existen herramientas de desarrollo que contienen contenedores Bean que permiten trabajar con componentes Beans de una manera visual, algunas de estas herramientas son Bean Box de Sun, Visual Age for Java de IBM, Visual Café de Symantec o JBuilder de Borland. Con estas aplicaciones se pueden manipular los Beans arrastrándolos a una posición determinada y definiendo sus propiedades y comportamiento. La herramienta generará de manera más o menos automática todo el código Java necesario.

De forma similar a las herramientas de programación visual las páginas JSP permiten a los desarrolladores crear aplicaciones Web basadas en Java sin la necesidad de escribir código Java, debido a que desde JSP podemos interactuar con los Beans a través de las ya mencionadas etiquetas especiales que se alojan junto con el código HTML de la página, es decir, a través de las acciones JSP para la manipulación e instanciación de componentes JavaBeans.

Por lo tanto las herramientas de programación Visual, como puede ser JBuilder, y otros programas, como pueden ser las páginas JSP, pueden descubrir de manera automática la información sobre las clases que siguen la especificación JavaBeans y pueden crear y manipular los componentes JavaBeans sin que el desarrollador tenga que escribir código de manera explícita, ya hemos comentado que el grueso del código fuente de la implementación se trasladará desde las páginas JSP, que lo encontrábamos en forma de scriptlets, a las clases de los componentes JavaBeans correspondientes.

Por lo tanto podemos decir que una página JSP es un contenedor Bean que nos permite utilizar los componentes JavaBeans a través de las acciones JSP: `<jsp:useBean>`, `<jsp:getProperty>` y `<jsp:setProperty>`.

## Las propiedades de los Beans

El siguiente concepto que vamos a tratar es el de las propiedades de los Beans. Los contenedores Beans nos permiten trabajar con los componentes Beans en términos de propiedades. Una propiedad es un atributo de un componente JavaBean que tiene como función mantener su estado y controlar el comportamiento del Bean.

Un Bean se encuentra definido por sus propiedades, sin ellas no tendría mucha utilidad, ya que el contenedor Bean no se podría comunicar con él. Las propiedades de un Bean pueden ser modificadas en tiempo de ejecución por el contenedor de Beans para controlar aspectos específicos del comportamiento del componente. Estas propiedades es lo que el contenedor de Beans ofrece al desarrollador para poder manipular los Beans.

Como un ejemplo práctico podemos tener un componente llamado TiempoBean que conoce varios datos acerca de las condiciones atmosféricas y del tiempo meteorológico en general. Este componente puede obtener las variables meteorológicas correspondientes desde las máquinas del Instituto Nacional de Meteorología o desde una base de datos determinada, si n embargo el usuario del componente no tiene porque conocer el funcionamiento interno del Bean o la forma en la que el Bean obtiene la información. Lo único que le interesa realmente al usuario (desarrollador) es que el componente nos ofrece información acerca de la temperatura, precipitaciones, presión atmosférica, etc., cada uno de estos datos será ofrecido al contenedor de Beans, para que los pueda utilizar el desarrollador, en forma de propiedades que serán utilizadas por las acciones JSP correspondientes.

Cada Bean va a tener un conjunto determinado de propiedades dependiendo del tipo de información que contenga el componente, podremos personalizar un Bean estableciendo los valores de sus propiedades.

El creador (desarrollador) del Bean puede imponer restricciones a cada propiedad del Bean, controlando el acceso que realizamos al mismo. Una propiedad puede ser de sólo lectura, de sólo

escritura o de lectura y escritura. Este concepto de accesibilidad permite al diseñador del Bean imponer límites a la forma de utilizar los Beans, por ejemplo, volviendo al ejemplo anterior del Bean `TiempoBean`, no tiene sentido que el usuario del Bean pueda modificar la propiedad de la temperatura actual, esta propiedad debería ser de sólo lectura. Por otro lado si disponemos de una propiedad que permite indicar el código de la zona de la cual se quieren obtener los datos meteorológicos tiene sentido que esta propiedad sea de escritura y de lectura, para que el usuario del Bean pueda indicar de que zona desea los datos.

Hay algunas propiedades especiales denominadas propiedades desencadenantes o trigger (trigger properties), la lectura o modificación de una de estas propiedades enviará una señal al Bean que desencadenará un proceso o una actividad. Si seguimos con el mismo ejemplo, al cambiar la propiedad del código de la zona, se deberán volver a recalcular y actualizar los datos meteorológicos correspondientes, ya que se deben obtener para una zona distinta, también se deberían modificar el resto de las propiedades del Bean, como pueden ser la temperatura, lluvias, presión atmosférica, humedad, etc., al modificar la propiedad de código de zona se ven afectadas el resto de las propiedades, por lo que se dice que estas propiedades se encuentran enlazadas (linked properties).

También es posible que una única propiedad pueda almacenar una colección de valores, este tipo de propiedades se denominan propiedades indexadas (indexed properties), cada uno de los valores almacenados en la propiedad se puede recuperar utilizando su índice correspondiente, de la misma forma que accedemos al elemento de un array. Continuando con nuestro ejemplo, el Bean `TiempoBean` puede tener una propiedad indexada que contenga todas las temperaturas máximas que se han dado durante la semana anterior.

No todos los contenedores de Beans ofrecen un mecanismo sencillo para trabajar con las propiedades indexadas. Particularmente las páginas JSP no reconocen estas propiedades multivaluadas, en su lugar deberemos utilizar scriptlets que permitan manipular estas propiedades, es decir, no tenemos disponibles acciones de JSP para utilizar este tipo de propiedades, más adelante en este mismo capítulo veremos la forma de utilizar estas propiedades. Existe otra alternativa que consiste en construir nuestras etiquetas personalizadas para JSP, este mecanismo lo veremos en el capítulo correspondiente.

Las propiedades de los Beans pueden almacenar un amplio tipo de información distinta, atendiendo a las necesidades de cada componente. Cada propiedad de un Bean puede almacenar datos de un tipo determinado, como pueden ser enteros o texto. A cada propiedad de un Bean se le asigna un tipo de dato Java, este tipo de dato será utilizando internamente en la implementación del componente y en el código Java generado por el contenedor de Beans para instanciar el Bean y poder utilizar sus propiedades.

El tipo de los datos de los valores de las propiedades puede ser cualquiera de los tipos primitivos de Java como `int` o `double`, así como objetos Java como puede ser `String` o `Date`. Las propiedades pueden almacenar también clases definidas por nosotros e incluso otros Beans. Las propiedades indexadas almacenan un array de valores del mismo tipo de datos.

El contenedor de Beans determina como trabajaremos con los valores de las propiedades de los Beans, con los scriptlets de las páginas JSP y expresiones utilizaremos los valores de las propiedades con el tipo Java adecuado. Si una propiedad almacena valores enteros, nos devolverá valores enteros y le tendremos que asignar valores enteros.

Sin embargo cuando utilizamos las propiedades de los Beans desde las acciones estándar de JSP, se tratará cada propiedad como si fuera un dato de la clase `String`, cuando le asignamos el valor a una propiedad de un Bean se lo pasamos como texto. De la misma forma cuando recuperamos el valor de una propiedad de un Bean obtenemos texto, con independencia del tipo interno que tiene la propiedad. Este tratamiento de los datos como texto hace que la utilización de las acciones de JSP para

manipularlos Beans sea bastante sencilla, y además el tipo texto es el adecuado para la generación de contenidos HTML que se devolverán al cliente.

El contenedor JSP realiza automáticamente todas las conversiones que sean necesarias para recuperar y establecer los valores de las propiedades de los Beans utilizadas desde las acciones de JSP. Así cuando establecemos el valor de una propiedad cuyo tipo es un dato entero, el contenedor JSP realizará las llamadas Java para convertir los caracteres que hemos indicado en el número entero correspondiente. Eso sí, debemos indicarles los valores del texto adecuados para que el contenedor JSP pueda realizar la conversión de manera correcta.

Las propiedades de un Bean se suelen documentar en una tabla denominada hoja de propiedades, esta tabla lista todas las propiedades disponibles en el Bean, el nivel de acceso permitido a los usuarios y el tipo Java correspondiente. La Tabla 15 muestra la hoja de propiedades que se correspondería con el componente TiempoBean.

Nombre	Acceso	Tipo Java	Valor de ejemplo
codigoZona	Lectura/escritura	String	28041
tempActual	Sólo lectura	Int	11
tempMaxHoy	Sólo lectura	Int	14
tempMinHoy	Sólo lectura	Int	4
humedad	Sólo lectura	Float	0.62
tiempo	Sólo lectura	String[]	Soleado, lluvioso, nublado, caluroso, fresco, despejado
icono	Sólo lectura	URL	http://servidor/tiempo/nublado.gif

Tabla 15. Hoja de propiedades

Las hojas de propiedades permiten a los diseñadores de Beans describir las características de un Bean a sus usuarios, que pueden ser diseñadores de páginas JSP, programadores de servlets, etc., a partir de una hoja de propiedades un desarrollador puede determinar el tipo de información que el Bean puede contener y el comportamiento que puede ofrecer.

Una vez comentados los fundamentos principales de los componentes JavaBeans, en el siguiente apartado vamos a mostrar como utilizar los componentes JavaBeans desde las páginas JSP, se puede considerar que este apartado es una continuación o prolongación de la utilización de las acciones JSP para la utilización de Beans.

## Utilización de los componentes JavaBeans

En el capítulo anterior ya tratamos en detalle la utilización de las acciones `<jsp:useBean>`, `<jsp:getProperty>` y `<jsp:setProperty>`, en este apartado vamos a seguir comentando la utilización de los Beans dentro de las páginas JSP centrándonos en algunos aspectos que quedaron sin explicar o que tratamos de forma superficial.

Para refrescar la memoria se ofrece el Código Fuente 183 que muestra una página JSP que establece el valor de una propiedad de un Bean y recupera los valores de todas sus propiedades.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Acción jsp:setProperty II</title>
</head>

<body>
<%String valor=request.getParameter("nombre");%>
<jsp:useBean id="miBean" scope="page" class="PersonaBean"/>
<jsp:setProperty name="miBean" property="*" />
<jsp:setProperty name="miBean" property="nombre" value="<%=valor%>" />
Nombre:<b><jsp:getProperty name="miBean" property="nombre"/></b><br>
Apellidos:<b><jsp:getProperty name="miBean" property="apellidos"/></b><br>
Edad:<b><jsp:getProperty name="miBean" property="edad"/></b>

</body>
</html>
```

Código Fuente 183

En primer lugar vamos a comentar como podemos utilizar las propiedades indexadas que presentan algunos Beans. Si retomamos el Bean del ejemplo anterior cuya clase es *PersonaBean*, podemos pensar que este Bean puede presentar una propiedad llamada *aficiones*, que consiste en una colección de cadenas de caracteres para indicar las aficiones que tiene una persona determinada. En la clase se encontrará implementado esta propiedad como un array de objetos *String*.

Como ya hemos comentado en el apartado anterior, las acciones estándar que define la especificación JSP no permiten manipular las propiedades indexadas, por lo tanto deberemos utilizar los scriptlets de las páginas JSP. Debemos recordar que una vez realizada la referencia al componente *JavaBean* que deseamos utilizar en nuestra página JSP mediante la acción `<jsp:useBean>`, podremos acceder al Bean desde los scriptlets de nuestra página JSP utilizando el mismo identificador que se utilizó en la propiedad *id* de la acción `<jsp:useBean>`.

En este caso nuestro Bean ofrece un par de métodos para establecer el valor de la propiedad y recuperarlo, estos métodos poseen un parámetro que indica el índice que se quiere recuperar de la propiedad, es decir, tenemos que indicar que número de valor de la propiedad queremos establecer o recuperar. En el Código Fuente 184 se muestra una página JSP que a través de un scriptlet establece uno a uno los valores de la propiedad *aficiones*, indicando su valor y su índice. Más adelante, para recuperar los valores de la propiedad *aficiones* se recorren todos ellos y se van mostrando en el navegador. Para ejecutar esta página se supone que le pasamos los valores de las propiedades *nombre*, *apellidos* y *edad* del Bean a través de un formulario.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Propiedades Indexadas</title>
</head>
<body>
<jsp:useBean id="persona" scope="page" class="PersonaBean">
    <jsp:setProperty name="persona" property="*" />
</jsp:useBean>
```

```

<%persona.setAficion("Lectura",0);
persona.setAficion("Música",1);
persona.setAficion("Cine",2);%>
Soy <jsp:getProperty name="persona" property="nombre"/>
<jsp:getProperty name="persona" property="apellidos"/> y tengo
<jsp:getProperty name="persona" property="edad"/> años<br>
Mis aficiones son:
<ul>
<%for(int i=0;i<3;i++){%>
    <li><%=persona.getAficion(i)%></li>
<%}%>
</ul>
</body>
</html>

```

Código Fuente 184

Un ejemplo de ejecución de esta página JSP lo tenemos en la Figura 86.

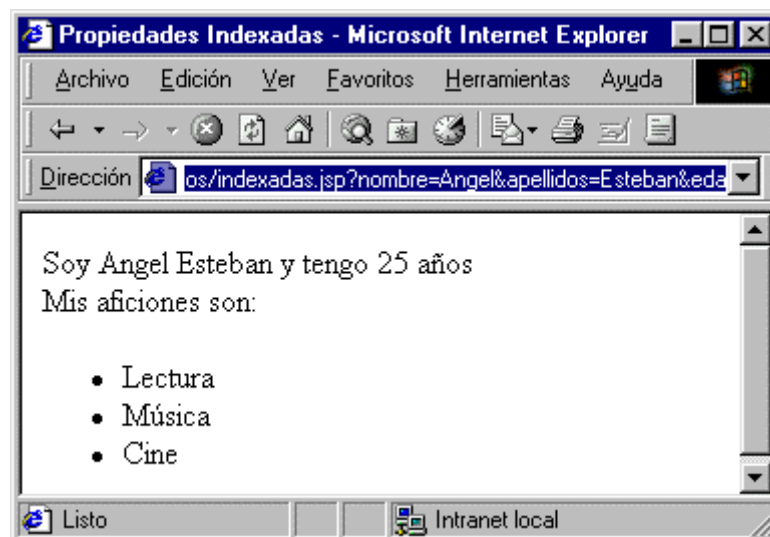


Figura 86. Utilizando propiedades indexadas

El código de la clase del componente JavaBean también es distinto, ya que hemos añadido una propiedad más, el nuevo aspecto se puede ver en el Código Fuente 185. El lector puede comprobar que el código fuente de este Bean es muy sencillo, pero debe cumplir unas normas para poder ser utilizado y considerado como un Bean, estas normas y la creación de Beans lo veremos en el siguiente capítulo, en este momento estamos viendo como utilizar las propiedades indexadas, y en el siguiente capítulo veremos como implementarlas para nuestros componentes JavaBeans.

```

public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;
    String [] aficiones;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
        this.aficiones=new String[3];
    }
}

```

```

    }

    public String getAficion(int indice){
        return aficiones[indice];
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }

    public String getApellidos(){
        return apellidos;
    }

    public void setAficion(String valor, int indice){
        aficiones[indice]=valor;
    }

    public void setApellidos(String apellidos){
        this.apellidos=apellidos;
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad=edad;
    }
}

```

Código Fuente 185

Para utilizar las propiedades indexadas también podemos tratarlas como un todo, esta aproximación consiste en construir una cadena de caracteres (String) con los valores que se desean establecer a una propiedad multivaluada separados por un delimitador (la coma por ejemplo), y al recuperar los valores se nos devolverá también una cadena con todos los valores separados por el mismo delimitador. Esta forma de tratar las propiedades nos permite utilizar las acciones JSP `<jsp:getProperty>` y `<jsp:setProperty>`, aunque como ya hemos dicho se trata la propiedad como un todo, no podemos indicar un valor, sino todos los valores de la propiedad, lo mismo pasa a la hora de recuperar los valores, no nos devuelve un valor de un índice determinado, sino que nos devuelve todos los valores que posee la propiedad.

En el Código Fuente 186 se ofrece una página JSP que utiliza esta aproximación para el tratamiento de propiedades indexadas.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Propiedades Indexadas II</title>
</head>
<body>
<jsp:useBean id="persona" scope="page" class="PersonaBean">
    <jsp:setProperty name="persona" property="*/>
    <jsp:setProperty name="persona"
        property="listaAficiones" value="Lectura,Música,Cine"/>

```

```
</jsp:useBean>
Soy <jsp:getProperty name="persona" property="nombre"/>
<jsp:getProperty name="persona" property="apellidos"/> y tengo
<jsp:getProperty name="persona" property="edad"/> años<br>
Mis aficiones son:
<jsp:getProperty name="persona" property="listaAficiones"/>
</body>
</html>
```

Código Fuente 186

Evidentemente el código de la clase de nuestro Bean ha cambiado, en este caso se han añadido los nuevos métodos de acceso que son `getListaAficiones()` y `setListaAficiones()`, que modifican la misma propiedad aficiones, pero en este caso se construye un objeto `String` con el delimitador de la coma, esto es lo que nos permite utilizar esta propiedad desde las acciones estándar de JSP. El nuevo código fuente de nuestro Bean (Código Fuente 187) se ofrece a continuación.

```
import java.util.*;

public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;
    String [] aficiones;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
        this.aficiones=new String[3];
    }

    public String getAficion(int indice){
        return aficiones[indice];
    }

    public String getListaAficiones(){
        String lista=new String();
        for(int i=0;i<aficiones.length;i++){
            if(i!=aficiones.length-1)
                lista+=aficiones[i]+",";
            else
                lista+=aficiones[i];
        }
        return lista;
    }

    public void setListaAficiones(String lista){
        StringTokenizer tok=new StringTokenizer(lista,",");
        int i=0;
        while(tok.hasMoreTokens()){
            aficiones[i]=(String)tok.nextToken();
            i++;
        }
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }
}
```

```

    }

    public String getApellidos(){
        return apellidos;
    }

    public void setAficion(String valor, int indice){
        aficiones[indice]=valor;
    }

    public void setApellidos(String apellidos){
        this.apellidos=apellidos;
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad=edad;
    }
}

```

Código Fuente 187

Y el resultado de la ejecución de esta página es el de la Figura 87.

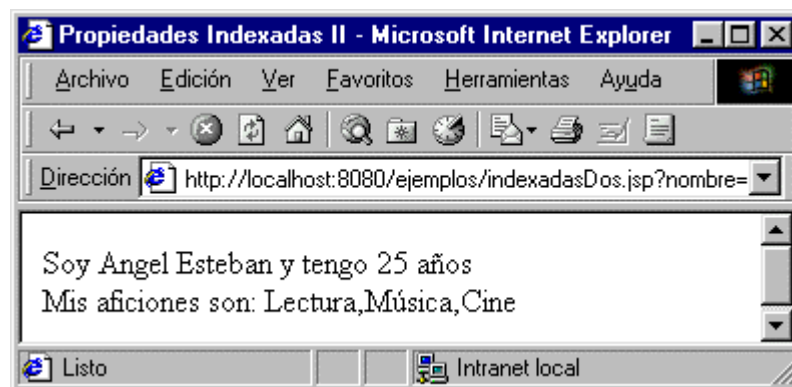


Figura 87. Otra aproximación al uso de propiedades indexadas

Y por último, en lo que a propiedades indexadas se refiere, podemos tener una solución distinta más, que consiste en tener un método getXXX() que devuelve el array con los valores de las propiedades, estos valores los podremos recorrer en el scriptlet correspondiente de nuestra página JSP. El Código Fuente 188 es una página JSP que utiliza esta otra aproximación.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Propiedades Indexadas III</title>
</head>
<body>
<jsp:useBean id="persona" scope="page" class="PersonaBean">
    <jsp:setProperty name="persona" property="*" />
    <%persona.setAficion("Lectura", 0);
    persona.setAficion("Música", 1);
    persona.setAficion("Cine", 2);%>

```



```

</jsp:useBean>

Soy <jsp:getProperty name="persona" property="nombre"/>
<jsp:getProperty name="persona" property="apellidos"/> y tengo
<jsp:getProperty name="persona" property="edad"/> años<br>
Mis aficiones son:
<ul>
<%String [] aficiones=persona.getAficiones();
for(int i=0;i<3;i++){%>
    <li><%=aficiones[i]%></li>
<%}%>
</ul>
</body>
</html>

```

Código Fuente 188

De nuevo el código de la clase del Bean se ha visto modificado, en este caso se ha añadido el método `getAficiones()` que devuelve el array de objetos `String` que representa a la propiedad `aficiones`, en el Código Fuente 189 se muestra el nuevo código del Bean, para simplificarlo hemos eliminado los métodos `getListaAficiones()` y `setListaAficiones()`, de todas formas en el siguiente [enlace](#) se ofrece la clase completa del componente `JavaBean`.

```

import java.util.*;

public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;
    String [] aficiones;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
        this.aficiones=new String[3];
    }

    public String getAficion(int indice){
        return aficiones[indice];
    }

    public String[] getAficiones(){
        return aficiones;
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }

    public String getApellidos(){
        return apellidos;
    }

    public void setAficion(String valor, int indice){
        aficiones[indice]=valor;
    }
}

```

```

    public void setApellidos(String apellidos){
        this.apellidos=apellidos;
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad=edad;
    }
}

```

Código Fuente 189

Otro aspecto de la utilización de los Beans dentro de las páginas JSP que quedó en el tintero fue los distintos ámbitos en los que se podían crear y sus consecuencias. Dentro de la acción `<jsp:useBean>` disponemos del atributo `scope` para indicar el ámbito en el que se debe utilizar el componente JavaBean que se va a instanciar, recordamos que existen cuatro ámbitos distintos dentro de nuestras páginas JSP: ámbito de página (`page`), ámbito de petición (`request`), ámbito de sesión (`session`) y ámbito de aplicación (`application`).

El ámbito indicará el tiempo de existencia del Bean, el valor por defecto del atributo `scope` es `page`, es decir, el ámbito de página, por lo que estos Beans únicamente existirán en la página actual. Este ámbito también determina la accesibilidad del Bean desde otras páginas JSP de la aplicación Web. En la Tabla 16 se muestra la relación existente entre los ámbitos de un Bean, su accesibilidad y su duración o existencia.

Ámbito	Accesibilidad	Existencia
<code>page</code>	Únicamente la página actual	Hasta que la página se ha terminado de mostrar o el control es redirigido hacia otro recurso.
<code>request</code>	Página actual y cualquiera que se incluya o redirija	Hasta que la petición se ha procesado completamente y la respuesta se ha enviado al usuario.
<code>session</code>	La petición actual y cualquiera de las siguientes peticiones que tengan lugar en la misma ventana (sesión) del navegador.	La vida de la sesión del usuario.
<code>application</code>	La petición actual y cualquiera de las siguientes peticiones que tengan lugar en la misma aplicación Web.	La vida de la aplicación Web

Tabla 16. Ámbitos de un componente JavaBean

Cuando creamos un Bean que se puede utilizar a través de varias páginas, cada vez que se utilice la acción `<jsp:useBean>` para instanciarlo, el contenedor JSP intenta localizarlo en el ámbito correspondiente, y si lo encuentra nos devuelve una referencia al mismo, en caso contrario se crea una nueva instancia del Bean.

Es posible definir una serie de sentencias de inicialización para que las propiedades de los Beans tengan valores iniciales. Para hacer esto se debe incluir el código de inicialización que sea necesario en el cuerpo de la etiqueta `<jsp:useBean>`, este código se ejecutará únicamente cuando la instancia del Bean se corresponde con una instancia nueva, y no cuando hace referencia a un Bean ya existente. En el Código Fuente 190 se muestra una página JSP que inicializa las propiedades del componente `PersonaBean`, conocido ya por todos.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Propiedades Indexadas III</title>
</head>
<body>
<jsp:useBean id="persona" scope="session" class="PersonaBean">
  <b>Se inicializa el Bean</b><br>
  <jsp:setProperty name="persona" property="nombre" value="Angel"/>
  <jsp:setProperty name="persona" property="apellidos" value="Esteban"/>
  <jsp:setProperty name="persona" property="edad" value="25"/>
  <%persona.setAficion("Lectura",0);
  persona.setAficion("Música",1);
  persona.setAficion("Cine",2);%>
</jsp:useBean>

Soy <jsp:getProperty name="persona" property="nombre"/>
<jsp:getProperty name="persona" property="apellidos"/> y tengo
<jsp:getProperty name="persona" property="edad"/> años<br>
Mis aficiones son:
<ul>
<%String [] aficiones=persona.getAficiones();
for(int i=0;i<3;i++){%>
  <li><%=aficiones[i]%></li>
<%}%>
</ul>
</body>
</html>
```

Código Fuente 190

En este caso se utiliza un Bean con ámbito de sesión, si el Bean existe no se ejecutará el código de inicialización, pero si se debe crear nuevo si que se ejecutará. En el código de inicialización podemos utilizar scriptlets, expresiones, acciones, etc.

En algún momento puede ser necesario realizar explícitamente la eliminación de un Bean. Al final de la vida del Bean, que se determina como hemos visto anteriormente a partir de su ámbito, se eliminan todas las referencias de forma automática y seleccionadas por el recolector de basura (garbage collector). Los Beans con ámbito de página o de petición se eliminarán y liberarán sus recursos al finalizar la petición correspondiente, pero los Beans con ámbito de sesión o aplicación pueden prevalecer durante más tiempo.

Si deseamos eliminar un Bean con ámbito de sesión o de aplicación, por lo motivos que sea, podemos utilizar el método `removeAttribute()` del objeto `session` o `application`, según el ámbito del Bean, pasándole por parámetro el identificador del Bean utilizado al instanciarlo en la acción `<jsp:useBean>`.

En el próximo capítulo vamos a adentrarnos en los detalles de implementación de los componentes JavaBeans y veremos las normas que define la especificación JavaBeans para sus componentes. En el presente capítulo hemos comentado y tratado los Beans desde el punto de vista del diseñador de

páginas JSP, es decir, el usuario que va a utilizar los componentes, pero en siguiente capítulo vamos a cambia de punto de vista y vamos a tratar los Beans desde el punto de vista del desarrollador de Beans.

Por lo tanto crearemos nuestros propios Beans a través de diversos ejemplos, para luego utilizarlos desde las páginas JSP de nuestra aplicación Web.

# Desarrollo de componentes JavaBeans

---

## Introducción

En este capítulo se va a mostrar como crear las clases de componentes JavaBeans, que deberán cumplir con las normas establecidas por el modelo de componentes JavaBeans. Para ello utilizaremos varios ejemplos de Beans que utilizaremos desde páginas JSP. No pretendemos crear un tutorial completo del desarrollo de componentes JavaBeans, sino que vamos a comentar los aspectos más relevantes que se encuentra relacionados con el tema que nos interesa, las páginas JSP.

Este capítulo es la continuación lógica del anterior, en el que introducíamos el concepto de componente y el concepto de Bean, también veíamos como podíamos utilizar los Beans desde las páginas JSP. Pero nos quedaba una parte pendiente, que es el desarrollo de Beans.

A lo largo de los siguientes apartados vamos a ir desglosando la creación de un sencillo componente JavaBean de ejemplo, para ir mostrando que convenciones deben cumplir las clases de estos componentes, y a continuación veremos dónde tenemos que situar la clase del componente JavaBean para poder utilizarlo desde una página JSP.

## Las clases de los Beans

Como ya hemos comentado en diversas ocasiones, un componente JavaBean o Bean, es un objeto de una instancia de una clase Java, lo único que tiene esta clase de particular para ser considerada un Bean, es que sigue una serie de convenciones.

Estas convenciones son definidas por el API JavaBeans, y son una serie de reglas que definen la forma en la que se debe crear el constructor del Bean y los métodos de acceso a sus propiedades.

Una instancia de un Bean la podemos tratar igual que un objeto cualquiera dentro de los scriptlets de nuestra página JSP, pero lo más común (y recomendable), es utilizar los Beans a través de las acciones estándar de JSP correspondientes.

Existe una convención para nombrar las clases de los Beans, no es un requerimiento obligatorio pero suele ser lo más común, y consiste en al nombre de la clase añadirle la terminación Bean, así si el nombre de la clase del Bean es Persona, la clase que va a representar al Bean se llamará PersonaBean. Esta forma de nombrar las clases de los componentes JavaBeans ayuda sobre todo a los desarrolladores a identificar que tipo de clase se trata.

Para averiguar las propiedades que nos ofrece un Bean el contenedor JSP utiliza el proceso de introspección (introspection), que permite a la clase del Bean exponer su métodos a la hora de realizar la petición. El proceso de introspección se produce en tiempo de ejecución y es controlado por el contenedor de páginas JSP.

El proceso de introspección se produce a través de un mecanismo denominado reflexión (reflection), que permite al contenedor de Beans examinar cualquier clase en tiempo de ejecución para determinar las cabeceras de sus métodos. El contenedor de Beans determina cuales son las propiedades que soporta un Bean a partir del análisis de sus métodos públicos, de esta forma comprobará la presencia de métodos que cumplan con el criterio de la especificación JavaBeans. Adelantamos que para que exista una propiedad de un Bean debe tener un método de acceso que permita obtener su valor, o bien un método que permita asignarle un valor o ambos. La presencia de estos métodos de acceso en la clase del Bean determinan las propiedades del Bean, aunque esto lo veremos en detalle más adelante.

## Los constructores de los Beans

La primera regla para construir un componente JavaBean es que la clase del Bean debe poseer un constructor sin argumentos, este será el constructor que el contenedor JSP utilizará para instanciar el componente a través de la acción `<jsp:useBean>`.

Normalmente el constructor de un Bean se utiliza para inicializar los atributos o variables de instancia del mismo, para que de esta forma el Bean una vez instanciado esté listo para su utilización.

Así por ejemplo el Código Fuente 191 muestra el constructor de la clase PersonaBean que inicializa cada uno de sus atributos.

```
public class PersonaBean{
    private String nombre;
    private String apellidos;
    private int edad;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
    }
}
```

Código Fuente 191

## Definiendo las propiedades

Ya hemos mencionado que las propiedades de un Bean se encuentran definidas a través de sus métodos de acceso, estos métodos se utilizarán para devolver el valor de una propiedad o establecer el valor de la misma. Normalmente los métodos de acceso utilizados para devolver los valores de las propiedades de un Beans se denominan métodos getter, y los métodos de acceso utilizados para establecer los valores de las propiedades de los Beans se denominan métodos setter.

Para definir las propiedades para un Bean crearemos simplemente un método público con el nombre de la propiedad precedido por el prefijo get (obtener) o set (establecer) según lo que corresponda en cada caso. Los métodos getter deben devolver el tipo correcto de la propiedad, y los métodos setter deben ser declarados como void aceptando un parámetro con el tipo adecuado para establecer el valor de la propiedad correspondiente. La sintaxis general para definir los métodos que determinan una propiedad es la siguiente.

```
Public TipoPropiedad getNombrePropiedad()  
Public void setNombrePropiedad(TipoPropiedad valor)
```

Así por ejemplo si queremos definir la propiedad nombre dentro de nuestro componente PersonaBean, deberemos indicar las siguientes cabeceras de los métodos de acceso, suponiendo que la propiedad nombre se de la clase String.

```
public String getNombre()  
public void setNombre(String nombre)
```

Si queremos que nuestra clase PersonaBean tenga las propiedades nombre, apellidos y edad, deberemos tener el siguiente código (Código Fuente 192) en la clase del Bean.

```
public class PersonaBean{  
    private String nombre;  
    private String apellidos;  
    private int edad;  
  
    public PersonaBean() {  
        this.nombre="";  
        this.apellidos="";  
        this.edad=0;  
    }  
  
    public String getNombre(){  
        return nombre;  
    }  
  
    public void setNombre(String nombre){  
        this.nombre=nombre;  
    }  
  
    public String getApellidos(){  
        return apellidos;  
    }  
  
    public void setApellidos(String apellidos){  
        this.apellidos=apellidos;  
    }  
  
    public int getEdad(){  
        return edad;  
    }  
}
```

```
public void setEdad(int edad){
    this.edad=edad;
}
}
```

Código Fuente 192

En este caso las tres propiedades definidas son de lectura y de escritura ya que presentan todas ellos métodos setter y getter, para modificar y obtener los valores, respectivamente.

Otro ejemplo de componente Bean puede ser el de un componente que almacena la hora actual, en este caso se tendrán dos propiedades llamadas horas y minutos, que serán de sólo lectura. El código de este Bean (Código Fuente 193) se muestra a continuación.

```
import java.util.*;

public class HoraBean{
    private int horas;
    private int minutos;

    public HoraBean(){
        Calendar ahora=Calendar.getInstance();
        this.horas=ahora.get(Calendar.HOUR_OF_DAY);
        this.minutos=ahora.get(Calendar.MINUTE);
    }

    public int getHoras(){
        return horas;
    }

    public int getMinutos(){
        return minutos;
    }
}
```

Código Fuente 193

Este dos componentes JavaBeans los podríamos utilizar ya dentro de una página JSP utilizando las acciones estándar adecuadas. Para poder utilizar cualquier Bean debemos compilarlo previamente con el compilador de Java como si se tratara de una clase cualquiera. Una vez compilado debemos copiar el fichero de la clase (.CLASS) al lugar adecuado del servidor Web, para que las páginas JSP puedan hacer uso del componente.

En el caso de Jakarta Tomcat, la clase del Bean se debe situar en el directorio WEB-INF\CLASSES dentro de la aplicación Web correspondiente, de la misma forma que hacíamos con las clases de los servlets. Dentro de este directorio se pueden crear la estructura pertinente de subdirectorios de acuerdo con la estructura de paquetes a los que pertenezca el componente correspondiente. En nuestro caso copiaremos los ficheros de clase PersonaBean.class y HoraBean.class.

En el Código Fuente 194 se muestra el código de una página JSP que utiliza estos componentes a través de las ya conocidas acciones estándar de JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```



```
<html>
<head>
    <title>JavaBeans</title>
</head>

<body>
<jsp:useBean id="persona" class="PersonaBean"/>
<jsp:useBean id="hora" class="HoraBean"/>
<jsp:setProperty name="persona" property="nombre" param="nom"/>
Hola <jsp:getProperty name="persona" property="nombre"/> son las
<jsp:getProperty name="hora" property="horas"/>:
<jsp:getProperty name="hora" property="minutos"/>
</body>
</html>
```

Código Fuente 194

El resultado de la ejecución de la página JSP se puede observar en la Figura 88.

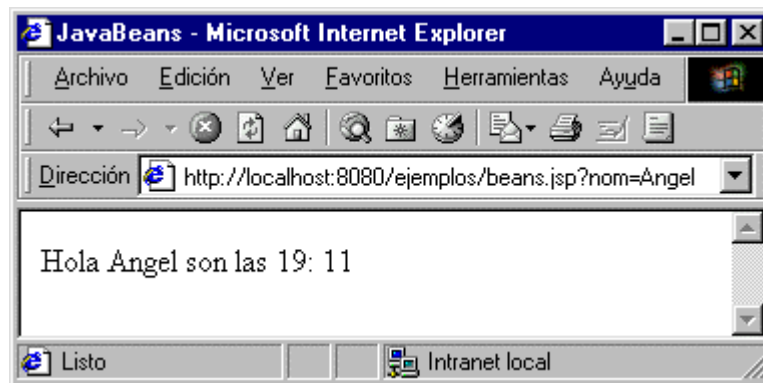


Figura 88. Utilizando componentes JavaBeans

No debemos confundir las propiedades con las variables de instancia o los atributos de una clase, aunque en muchos casos las variables de instancia se utilizan para contener los valores de las propiedades. Las propiedades de un Bean se encuentran definidas exclusivamente por el nombre de sus métodos de acceso, no por la implementación interna de los mismos. Esto permite que el desarrollador del Bean pueda modificar la estructura interna del mismo sin que se vea afectado al utilización del componente.

Un ejemplo de lo anterior puede ser un Bean que genera de forma dinámica los valores de sus propiedades, mediante la generación de números aleatorios, en lugar de devolver el valor de un atributo de la clase. El Código Fuente 195 muestra el código de la clase de este Bean, que ofrece dos propiedades, la primera es el valor de un número aleatorio entre 1 y 6, y la otra es el valor de un número aleatorio entre 2 y 12.

```
import java.util.*;

public class AleatorioBean{
    private Random aleatorio;

    public AleatorioBean(){
        aleatorio=new Random();
    }
}
```

```
public int getAleatorioUno() {  
    return aleatorio.nextInt(6)+1;  
}  
  
public int getAleatorioDos() {  
    return getAleatorioUno()+getAleatorioUno();  
}  
}
```

Código Fuente 195

El Código Fuente 196 es el ejemplo de utilización de este Bean desde una página JSP.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">  
  
<html>  
<head>  
    <title>JavaBeans</title>  
</head>  
  
<body>  
<jsp:useBean id="aleatorio" class="AleatorioBean"/>  
Primer número generado:  
<jsp:getProperty name="aleatorio" property="aleatorioUno"/><br>  
Segundo número generado:  
<jsp:getProperty name="aleatorio" property="aleatorioDos"/>  
</body>  
</html>
```

Código Fuente 196

Y la Figura 89 es el resultado de su ejecución.

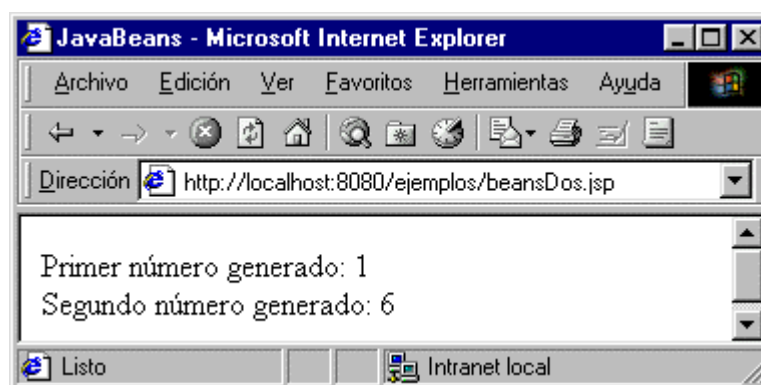


Figura 89. Bean que genera números aleatorios

Como hemos visto una propiedad determinada le corresponde un método get o set o ambos, en este caso el nombre de la propiedad siempre comenzará con una letra minúscula, aunque en el método get o set se utilice la primera letra mayúscula para la propiedad. Así a la propiedad nombre le podrá corresponder los métodos de acceso getNombre() y setNombre().

No es obligatorio crear los métodos `get` y `set`, si la propiedad es de lectura tendrá únicamente un método `get`, si es de sólo escritura presentará un método `set`, y si es de lectura y escritura presentará un método `get` y un método `set`.

A continuación vamos a comentar en un par de subapartados dos tipos de propiedades algo especiales, las propiedades indexadas y las propiedades booleanas.

## Propiedades indexadas

Ya comentábamos en el capítulo anterior que un Bean puede presentar propiedades multivaluadas, es decir, propiedades que pueden contener varios valores. Vimos la forma en la que se debían tratar estas propiedades en las páginas JSP, recordamos que las etiquetas estándar de JSP no son válidas para el tratamiento de estas propiedades, sino que debíamos utilizarlas desde scriptlets de la página JSP.

Ahora en este apartado vamos a ver las propiedades indexadas desde el punto de vista del desarrollador de componentes JavaBeans.

Normalmente las propiedades indexadas se tratan como un array de objetos de una clase determinada, así por ejemplo, si recordamos en el capítulo anterior, el componente `PersonaBean` contenía una propiedad llamada `aficiones` que puede contener las distintas aficiones de una persona.

En el capítulo anterior se realizaban varias aproximaciones, ahora en nuestro caso vamos a realizar también varias aproximaciones desde el punto de vista de la implementación. La primera de las soluciones consiste en utilizar métodos de acceso en los que se utiliza un parámetro para indicar el número de índice que se corresponde con uno de los valores de la propiedad. El ejemplo de cómo se implementaría en la clase del componente es muy sencillo y se puede observar en el Código Fuente 197.

```
import java.util.*;

public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;
    String [] aficiones;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
        this.aficiones=new String[3];
    }

    public String getAficion(int indice){
        return aficiones[indice];
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }

    public String getApellidos(){
        return apellidos;
    }
}
```

```

    }

    public void setAficion(String valor, int indice){
        aficiones[indice]=valor;
    }

    public void setApellidos(String apellidos){
        this.apellidos=apellidos;
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad=edad;
    }
}

```

Código Fuente 197

La limitación que se impone es que únicamente podemos indicar tres aficiones distintas. Para utilizar este componente en una página JSP utilizando su propiedad aficiones podríamos escribir el Código Fuente 198.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Propiedades Indexadas</title>
</head>
<body>
<jsp:useBean id="persona" scope="page" class="PersonaBean">
    <jsp:setProperty name="persona" property="*" />
</jsp:useBean>
<%persona.setAficion("Lectura", 0);
persona.setAficion("Música", 1);
persona.setAficion("Cine", 2);%>
Soy <jsp:getProperty name="persona" property="nombre"/>
<jsp:getProperty name="persona" property="apellidos"/> y tengo
<jsp:getProperty name="persona" property="edad"/> años<br>
Mis aficiones son:
<ul>
<%for(int i=0;i<3;i++){%>
    <li><%=persona.getAficion(i)%></li>
<%}%>
</ul>
</body>
</html>

```

Código Fuente 198

Para utilizar las propiedades indexadas también podemos tratarlas como un todo, esta aproximación consiste en construir una cadena de caracteres (String) con los valores que se desean establecer a una propiedad multivaluada separados por un delimitador (la coma por ejemplo), y al recuperar los valores se nos devolverá también una cadena con todos los valores separados por el mismo delimitador. Esta forma de tratar las propiedades nos permite utilizar las acciones JSP <jsp:getProperty> y <jsp:setProperty>, aunque como ya hemos dicho se trata la propiedad como un todo, no podemos indicar un valor, sino todos los valores de la propiedad, lo mismo pasa a la hora de recuperar los

valores, no nos devuelve un valor de un índice determinado, sino que nos devuelve todos los valores que posee la propiedad.

En este caso se debe variar también el código de nuestro componente para añadir dos métodos que se encarguen de tratar los valores de las propiedades como una lista de valores. En el método `get` se devuelve la lista de valores, por lo que se debe crear un objeto `String` con todos los valores de la propiedad aficiones. Y el método `set` recibe la lista de valores en forma de cadena de texto, por lo que tendrá que dividir esta cadena y asignar a cada elemento del array su valor correspondiente para actualizar la propiedad aficiones de forma adecuada. El Código Fuente 199 muestra la nueva implementación de la clase del componente `PersonaBean`.

```
import java.util.*;

public class PersonaBean{
    String nombre;
    String apellidos;
    int edad;
    String [] aficiones;

    public PersonaBean(){
        this.nombre="";
        this.apellidos="";
        this.edad=0;
        this.aficiones=new String[3];
    }

    public String getAficion(int indice){
        return aficiones[indice];
    }

    public String getListaAficiones(){
        String lista=new String();
        for(int i=0;i<aficiones.length;i++){
            if(i!=aficiones.length-1)
                lista+=aficiones[i]+",";
            else
                lista+=aficiones[i];
        }
        return lista;
    }

    public void setListaAficiones(String lista){
        StringTokenizer tok=new StringTokenizer(lista,",");
        int i=0;
        while(tok.hasMoreTokens()){
            aficiones[i]=(String)tok.nextToken();
            i++;
        }
    }

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombre){
        this.nombre=nombre;
    }

    public String getApellidos(){
        return apellidos;
    }

    public void setAficion(String valor, int indice){
```

```

        aficiones[indice]=valor;
    }

    public void setApellidos(String apellidos){
        this.apellidos=apellidos;
    }

    public int getEdad(){
        return edad;
    }

    public void setEdad(int edad){
        this.edad=edad;
    }
}

```

Código Fuente 199

En este caso se ha definido otra propiedad del Bean, llamada listaAficiones, que como se puede comprobar utiliza internamente el mismo atributo de la clase que se utilizaba anteriormente para almacenar el array de aficiones. En el Código Fuente 200 se muestra la utilización de este Bean desde una página JSP.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Propiedades Indexadas II</title>
</head>
<body>
<jsp:useBean id="persona" scope="page" class="PersonaBean">
    <jsp:setProperty name="persona" property="*" />
    <jsp:setProperty name="persona"
        property="listaAficiones" value="Lectura,Música,Cine" />
</jsp:useBean>
Soy <jsp:getProperty name="persona" property="nombre" />
<jsp:getProperty name="persona" property="apellidos" /> y tengo
<jsp:getProperty name="persona" property="edad" /> años<br>
Mis aficiones son:
<jsp:getProperty name="persona" property="listaAficiones" />
</body>
</html>

```

Código Fuente 200

Para tener un ejemplo más de propiedades indexadas se ofrece un componente JavaBeans que calcula una serie de operaciones sobre un conjunto de números, como pueden ser la media aritmética y la suma.

Los números se irán almacenando en una propiedad del Bean denominada numeros. En el constructor se inicializa el array de números a un array de únicamente dos elementos, pero mediante el método setListaNumeros(), que define la propiedad listaNumeros, podremos establecer en una cadena de caracteres separada por comas (,), el conjunto de números que vamos a utilizar para calcular la media y otras operaciones sobre los mismos.

El código fuente (Código Fuente 201) completo de este nuevo componente se ofrece a continuación, como se puede comprobar también se ofrecen dos propiedades de sólo lectura que se calculan de forma dinámica y que se llaman media y suma.

```
import java.util.*;

public class EstadisticasBean{

    private double[] numeros;

    public EstadisticasBean(){
        numeros=new double[2];
        numeros[0]=1;
        numeros[1]=2;
    }

    public double getMedia(){
        double suma=0;
        for(int i=0; i<numeros.length;i++)
            suma+=numeros[i];
        return suma/numeros.length;
    }

    public double getSuma(){
        double suma=0;
        for(int i=0; i<numeros.length;i++)
            suma+=numeros[i];
        return suma;
    }

    public double[] getNumeros(){
        return numeros;
    }

    public double getNumeros(int indice){
        return numeros[indice];
    }

    public void setNumeros(double[] numeros){
        this.numeros=numeros;
    }

    public void setListaNumeros(String valores){
        Vector vec=new Vector();
        StringTokenizer tok=new StringTokenizer(valores,"");
        while(tok.hasMoreTokens())
            vec.addElement(tok.nextToken());
        numeros=new double[vec.size()];
        for (int i=0;i<numeros.length;i++)
            numeros[i]=Double.parseDouble((String)vec.elementAt(i));
    }

    public String getListaNumeros(){
        String lista=new String();
        for(int i=0;i<numeros.length;i++){
            if(i!=numeros.length)
                lista+=numeros[i]+",";
            else
                lista+=numeros[i];
        }
        return lista;
    }
}
```

Código Fuente 201

Este Bean se puede utilizar de dos maneras distintas, ya que ofrece los métodos para ello. Podemos utilizar su propiedad indexada desde un scriptlet de una página JSP, como se puede ver en el Código Fuente 202, o bien podemos utilizar la acción correspondiente para asignar y obtener los valores de una propiedad indexada usando una cadena de caracteres separando con comas los distintos valores, esto lo hacemos gracias a la propiedad listaNumeros, y se puede observar en el Código Fuente 203.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>JavaBeans</title>
</head>
<body>
<jsp:useBean id="estadisticas" class="EstadisticasBean">
    <%double[] misNum={50,100,400,550,704};
    estadisticas.setNumeros(misNum);%>
</jsp:useBean>
La media de
<%double[] numeros=estadisticas.getNumeros();
for (int i=0;i<numeros.length;i++){
    if(i!=numeros.length)
        out.println(numeros[i]+",");
    else
        out.println(numeros[i]);
}%>
es igual a:
<jsp:getProperty name="estadisticas" property="media"/>, y su suma es
<jsp:getProperty name="estadisticas" property="suma"/>
</body>
</html>
```

Código Fuente 202

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>JavaBeans</title>
</head>
<body>
<jsp:useBean id="estadisticas" class="EstadisticasBean">
    <jsp:setProperty name="estadisticas" property="listaNumeros"
        value="50,100,400,550,704"/>
</jsp:useBean>
La media de
<jsp:getProperty name="estadisticas" property="listaNumeros"/>
es igual a:
<jsp:getProperty name="estadisticas" property="media"/>, y su suma es
<jsp:getProperty name="estadisticas" property="suma"/>
</body>
</html>
```

Código Fuente 203

En cualquier caso el resultado de la ejecución de ambas páginas es el mismo, y es el que se muestra en la Figura 90.

Estos Beans son todos bastante sencillos, ya que lo que queremos mostrar es la forma de desarrollar Beans y es más fácil comprender los conceptos a través de ejemplos sencillos, más adelante cuando



tratamos el acceso a bases de datos desde JSP utilizaremos algún componente JavaBean más complejo.

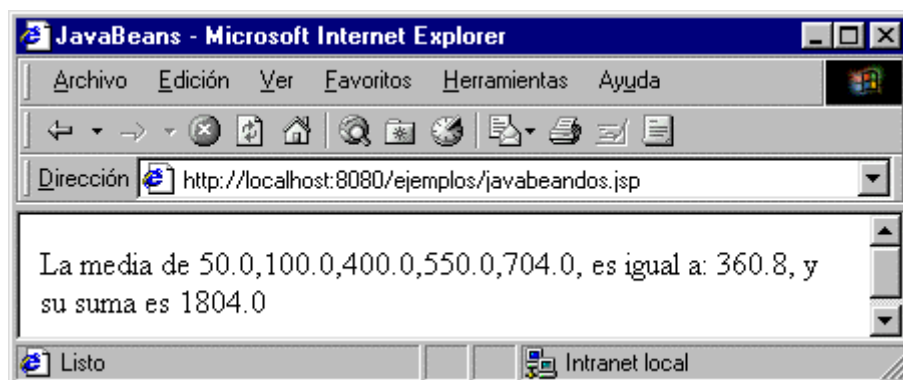


Figura 90. Resultado de la ejecución de las páginas anteriores

Con este ilustrativo ejemplo finalizamos la explicación sobre las propiedades indexadas, a continuación tratamos con otro tipo de propiedades.

## Propiedades booleanas

Las propiedades booleanas son aquellas que contienen un valor verdadero o falso (true/false). Esta propiedades presentan una particularidad en lo que a la especificación JavaBeans se refiere, no poseen un método `getNombrePropiedad()` correspondiente para obtener el valor de una propiedad, sino que utilizan el método `isNombrePropiedad()`. Por lo tanto la cabecera de los métodos que definen una propiedad booleana es la siguiente.

```
public boolean isNombrePropiedad()  
public void setNombrePropiedad(boolean valor)
```

Así por ejemplo podríamos tener el componente CasaBean con la propiedad booleana habitada, que devolverá verdadero o falso. A continuación se ofrece el sencillo código de este Bean (Código Fuente 204).

```
public class CasaBean{  
    private boolean habitada;  
  
    public CasaBean(){  
        this.habitada=false;  
    }  
  
    public boolean isHabitada(){  
        return this.habitada;  
    }  
  
    public void setHabitada(boolean valor){  
        this.habitada=valor;  
    }  
}
```

Código Fuente 204

Y en el Código Fuente 205 se ofrece la utilización de este Bean desde una página JSP, el resultado de la ejecución de esta página será el de la Figura 91.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>JavaBeans</title>
</head>
<body>
<jsp:useBean id="casa" class="CasaBean"/>
¿La casa está habitada?
<jsp:getProperty name="casa" property="habitada"/><br>
<jsp:setProperty name="casa" property="habitada" value="true"/>
¿La casa está habitada?
<jsp:getProperty name="casa" property="habitada"/>
</body>
</html>
```

Código Fuente 205

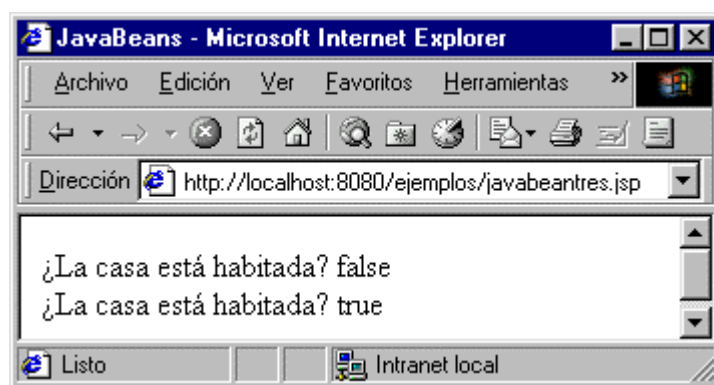


Figura 91. Utilizando propiedades booleanas

Dentro de este gran apartado de las propiedades de los Beans vamos a dedicar un subapartado a la conversión de tipos que tiene lugar en las acciones `<jsp:getProperty>` y `<jsp:setProperty>`.

## Conversión de tipos

Como ya hemos comentado los valores de las propiedades de los componentes JavaBeans no se encuentran limitados a valores de tipo String, pero es importante tener en cuenta que el valor devuelto por la acción `<jsp:getProperty>` siempre será de tipo String, es decir, internamente se da una conversión de tipos.

Sin embargo el método get correspondiente no tiene porque devolver un objeto de la clase String de forma explícita, es el contenedor JSP quien automáticamente convertirá el valor de retorno del método get correspondiente.

En la Tabla 17 se ofrece, para los tipos primitivos de Java, los métodos de conversión que se utiliza la acción `<jsp:getProperty>`.

Tipo de la propiedad	Conversión a tipo String
boolean	Boolean.toString(boolean valor)
byte	Byte.toString(byte valor)
char	Character.toString(char)
double	Double.toString(double)
int	Integer.toString(int valor)
float	Float.toString(float valor)
long	Long.toString(float valor)

Tabla 17. Conversiones realizadas por &lt;jsp:getProperty&gt;

A la hora de establecer el valor de una propiedad mediante la acción <jsp:setProperty> se deberá expresar el valor como un objeto String. De forma automática el contenedor de páginas JSP convertirá el valor String a su tipo correspondiente, que será el indicado en el método set de la propiedad.

Al igual que en el caso anterior, en la Tabla 18 se ofrece las conversiones de tipo, aplicadas a los tipos primitivos de Java, que utiliza internamente la acción <jsp:setProperty>.

Tipo de la propiedad	Conversión desde tipo String
Boolean o boolean	Boolean.valueOf(String valor)
Byte o byte	Byte.valueOf(String valor)
Character o char	Character.valueOf(String valor)
Double o double	Double.valueOf(String valor)
Integer o int	Integer.valueOf(String valor)
Float o float	Float.valueOf(String valor)
Long o long	Long.valueOf(String valor)

Tabla 18. Conversiones realizadas por &lt;jsp:setProperty&gt;

Con este subapartado damos por concluido el apartado dedicado a las propiedades de los componentes JavaBeans, en el siguiente apartado vamos a tratar un interfaz que pueden implementar los componentes JavaBeans, y que está muy relacionado con el objeto integrado session de los objetos integrados en JSP.

## El interfaz HttpSessionBindingListener

Este interfaz, que no es obligatorio implementar por los componentes JavaBeans, notificará al componente que lo implemente un par de eventos relacionados con el objeto integrado session de las páginas JSP, concretamente, indicará al componente cuando una instancia del mismo se ha almacenado en la sesión del usuario a través del objeto session, y también indicará cuando el objeto se elimina de la sesión.

Este interfaz, que pertenece al paquete `javax.servlet.http`, es decir, al API de los servlets, es muy simple y presenta únicamente un par de métodos que pasamos a comentar a continuación.

- `void valueBound(HttpSessionBindingEvent evento)`: este método se ejecutará cuando el Bean es almacenado por primera vez en el objeto integrado de JSP session, es decir, cuando se almacena en la sesión del usuario actual.
- `void valueUnbound(HttpSessionBindingEvent evento)`: este método se ejecutará cuando el Bean sea eliminado de la sesión correspondiente. Este evento se producirá cuando finalice la sesión del usuario o bien cuando de forma explícita se destruya la instancia del Bean.

La clase del evento lanzado, es decir, la clase `javax.servlet.http.HttpSessionBindingEvent`, posee un par de métodos relacionados con las páginas JSP que son los siguientes:

- `String getName()`: devuelve el nombre con el que se hace referencia al componente dentro de la sesión.
- `HttpSession getSession()`: devuelve una referencia a la sesión, objeto session, en la que se encuentra almacenada la instancia del componente.

Podemos modificar el último ejemplo de componente JavaBean del apartado anterior, el componente CasaBean, para que implemente el interfaz que nos ocupa. El Código Fuente 206 es el nuevo código de la clase del componente.

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

public class CasaBean implements HttpSessionBindingListener{

    private boolean habitada;

    public CasaBean(){
        this.habitada=false;
    }

    public boolean isHabitada(){
        return this.habitada;
    }

    public void setHabitada(boolean valor){
        this.habitada=valor;
    }

    public void valueBound(HttpSessionBindingEvent evento){
        HttpSession session=evento.getSession();
        session.setAttribute("componente",
            "Se ha añadido el componente a la sesión y posee el nombre "+
            evento.getName());
    }
}
```

```

    }

    public void valueUnbound(HttpSessionBindingEvent evento){
        HttpSession session=evento.getSession();
        session.setAttribute("componente","Se ha eliminado el componente de la
sesión");
    }
}

```

Código Fuente 206

En los métodos del interfaz `javax.servlet.http.HttpSessionBindingListener` se deberán indicar los procesos y acciones que se deseen realizar en cada caso. En este ejemplo se crea un atributo de sesión llamado `componente`, que nos va a permitir comunicarnos con la página JSP para indicar en que estado se encuentra el componente.

El Código Fuente 207 muestra una página JSP que utiliza el componente `CasaBean` con ámbito de sesión, esto ejecutará el método `valueBound()` de nuestro componente. En la propia página se elimina el componente de la sesión, con lo que se ejecutará el método `valueUnbound()` del componente `CasaBean`.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>JavaBeans</title>
</head>
<body>
<jsp:useBean id="casa" class="CasaBean" scope="session"/>
<%=session.getAttribute("componente")%><br>
<%session.removeAttribute("casa"); %>
<%=session.getAttribute("componente")%><br>
</body>
</html>

```

Código Fuente 207

El resultado de la ejecución de la página anterior se puede ver en la Figura 92.

Figura 92. Utilizando el interfaz `HttpSessionBindingListener`

Para terminar este apartado y este capítulo, vamos a recapitular sobre las normas que debe cumplir una clase para poder ser considerada y tratada como un componente JavaBean.

- La clase se suele nombrar de la siguiente forma: NombreClaseBean.
- La clase debe tener un constructor sin argumentos.
- Las propiedades del componente se encuentran definidas por los métodos de acceso correspondientes, de la forma genérica `getNombrePropiedad()` y `setNombrePropiedad()`.

De momento es todo lo que vamos a ver de los componentes JavaBeans, un poco más adelante cuando veamos el acceso a bases de datos desde las páginas JSP volveremos a retomar el uso de componentes JavaBeans, en este caos serán componentes más complejos.

En el siguiente capítulo se va a tratar un tema que teníamos pendiente desde varios capítulos atrás, se trata del tratamiento de errores desde las páginas JSP.

# Tratamiento de errores en JSP

---

## Introducción

En este capítulo vamos a retomar el objeto integrado `exception` para comentar el tratamiento de errores dentro de las páginas JSP. Mostraremos una página JSP estándar para el tratamiento de errores dentro de nuestra aplicación Web, y también veremos como se puede realizar un tratamiento de errores centralizado, que nos permite tratar de igual forma los errores dentro de nuestras páginas JSP y dentro de nuestros servlets.

Para utilizar el tratamiento de excepciones dentro de nuestras páginas JSP, principalmente debemos hacer uso del objeto integrado `exception` y del atributo `isErrorPage` de la directiva `page`.

## Tipos de errores y excepciones

Dentro de una página JSP nos podemos encontrar con dos tipos de excepciones: errores en tiempo de traducción y excepciones en tiempo de ejecución.

Los errores de compilación de las páginas JSP producen errores en tiempo de traducción, es decir, errores que se producen cuando se intenta generar la clase del servlet resultante de la página JSP. Este tipo de errores que pueden ser: olvidar importar un paquete, no declarar un objeto de forma correcta, sintaxis incorrecta de una directiva, etc, sólo se deben dar durante el ciclo de desarrollo de la aplicación Web, nunca debería verlos un cliente en producción.

Los errores en tiempo de traducción dan como resultado errores internos de servlets, que se suelen identificar mediante el código 500.

El error en tiempo de traducción que se muestra en el navegador es específico de cada contenedor de páginas JSP, en la Figura 93 se muestra un error en tiempo de traducción que se ha producido en una página JSP al no utilizar un delimitador de sentencia cuando era necesario.

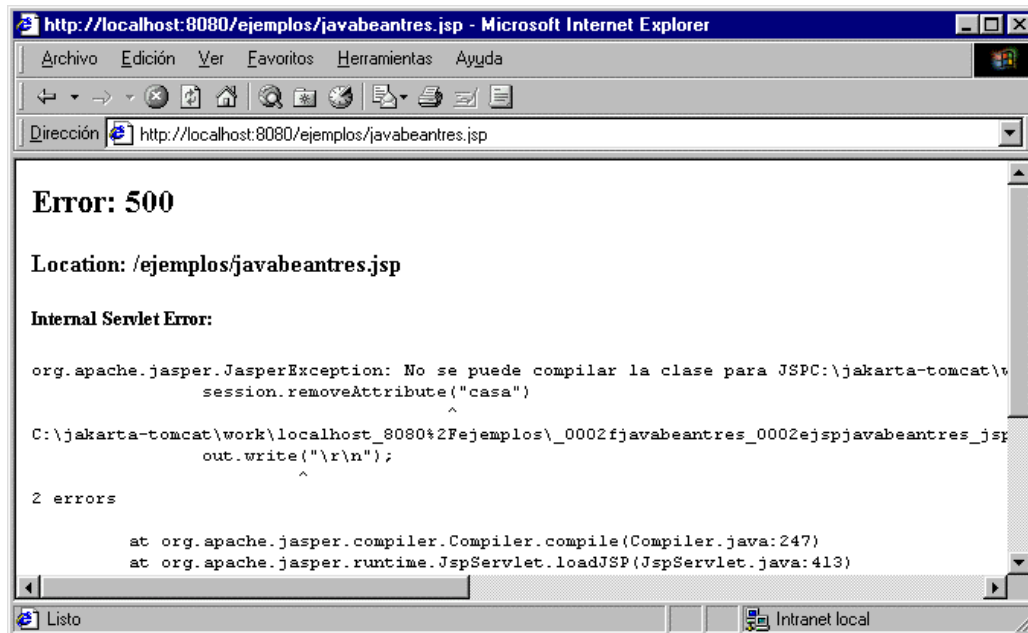


Figura 93. Error en tiempo de traducción

Como se puede comprobar en la zona superior de la página se indica el nombre del fichero en el que se ha producido el error de compilación, y además se muestra el error de compilación correspondiente, y en la parte inferior se puede apreciar un volcado de pila.

Los números de línea que aparecen indican el lugar del código fuente en el que se ha producido el error, no se corresponden con los de la página JSP, sino que hacen referencia al código del servlet que se ha generado a partir de la página JSP que se ha demandado.

Este tipo de errores se pueden dar en cualquier página JSP, incluso en las páginas de tratamiento de errores, que las veremos más adelante. Debemos prestar especial atención al nombre fichero indicado después de Location:, ya que así sabremos en qué página JSP se ha producido exactamente el error.

El servidor Jakarta Tomcat registra los errores en tiempo de traducción en el fichero de registro llamado TOMCAT.LOG, que se encuentra en el directorio logs del directorio de instalación de Tomcat. A continuación se ofrece un fragmento de la información que se suele almacenar en este fichero de registro.

```
Context log: path="/ejemplos" Error in jsp service() : No se puede
compilar la clase para JSPC:\jakarta-
tomcat\work\localhost_8080\ejemplos\0002fjavabeantres_0002ejspjav
abeantres_jsp_3.java:91: Invalid type expression.
    session.removeAttribute("casa")
                        ^
C:\jakarta-
tomcat\work\localhost_8080\ejemplos\0002fjavabeantres_0002ejspjav
abeantres_jsp_3.java:94: Invalid declaration.
    out.write("\r\n");
    ^
```



## 2 errors

```
org.apache.jasper.JasperException: No se puede compilar la clase
para JSPC:\jakarta-
tomcat\work\localhost_8080%2Fejemplos\_0002fjavabeantres_0002ejspjav
abeantres_jsp_3.java:91: Invalid type expression.
        session.removeAttribute("casa")
                          ^
C:\jakarta-
tomcat\work\localhost_8080%2Fejemplos\_0002fjavabeantres_0002ejspjav
abeantres_jsp_3.java:94: Invalid declaration.
        out.write("\r\n");
                ^
```

## 2 errors

```
at
org.apache.jasper.compiler.Compiler.compile(Compiler.java:247)
at
org.apache.jasper.runtime.JspServlet.loadJSP(JspServlet.java:413)
at
org.apache.jasper.runtime.JspServlet$JspServletWrapper.loadIfNecessa
ry(JspServlet.java:149)
```

El otro tipo de errores se denominaba errores en tiempo de ejecución, estos errores, como su nombre indica, se producen una vez que la página ya ha sido compilada en la clase del servlet correspondiente y se ha iniciado la ejecución de la misma por parte del usuario. Es el tipo de errores que se suelen tratar en el lenguaje Java utilizando los bloques try{}catch.

En el Código Fuente 208 se muestra una página JSP que realiza un tratamiento de errores utilizando los bloques try{}catch de la forma tradicional que se emplea en el lenguaje Java. En este caso se mostrará al usuario el mensaje “Se ha producido una excepción” seguido de la descripción de la excepción que se ha producido, como se puede apreciar en la Figura 94.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Genera excepción</title>
</head>
<body>
<%try{
    int i=1/0;
}catch(Exception ex){
    out.println("Se ha producido una excepción:"+ex);
}%>
</body>
</html>
```

Código Fuente 208

Este código fuente es bastante absurdo, ya que se trata de una página JSP con un scriptlet que siempre generará una excepción, ya que realiza una división por cero, pero para los conceptos que deseamos ilustrar es completamente válido.

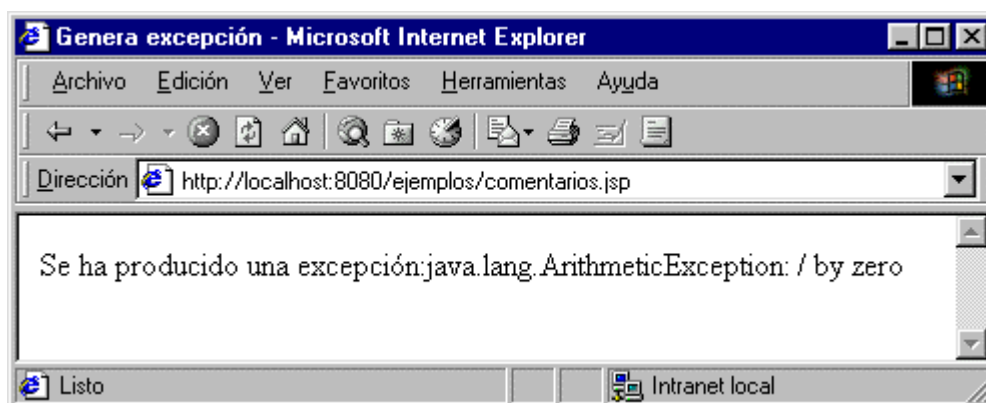


Figura 94. Tratando excepciones con try{}catch

Si en caso contrario no se atrapa una excepción, al usuario se le mostrará de nuevo la página de error por defecto del servidor Tomcat. Mostrándole un incomprensible volcado de pila. Así por ejemplo si eliminamos el bloque try{}catch del ejemplo anterior obtendremos un resultado como el de la Figura 95.

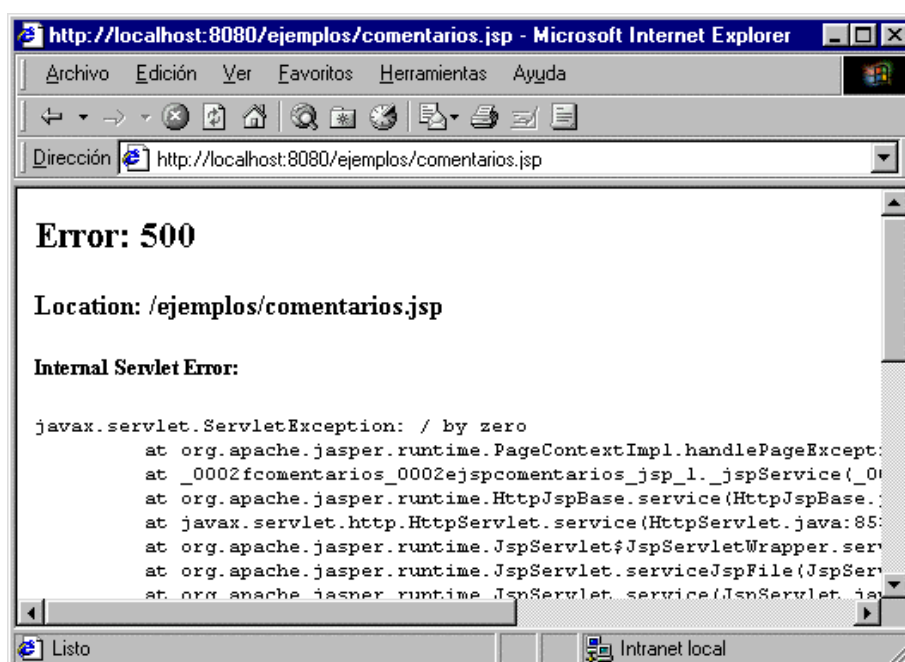


Figura 95. Excepción e tiempo de ejecución

Los errores en tiempo de ejecución se pueden tratar, además de la manera que ya conocíamos del lenguaje Java con los bloques try{}catch, con páginas JSP de error, utilizando para ello los atributos `errorPage` y `isErrorPage` de la directiva `page`, y el objeto integrado `exception`. En el siguiente apartado se indica como debemos utilizar la directiva `page` para tratar los errores con páginas JSP de error.

## Páginas JSP de error

Si utilizamos una página de error, en la página JSP en la que deseamos realizar este tratamiento de errores debemos indicar mediante el atributo `errorPage` de la directiva `page` la ruta a la página de

tratamiento de errores, así por ejemplo, si describimos la página JSP anterior que generaba una excepción de división por cero, tendríamos el siguiente código fuente(Código Fuente 209).

```
<%@ page errorPage="paginaErrorUno.jsp" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Genera excepción</title>
</head>
<body>
<%int i=1/0;%>
</body>
</html>
```

Código Fuente 209

Cuando se produce la excepción en esta página, de forma automática se vacía el búfer de salida y a continuación se muestra la página de error indicada en el atributo errorPage de la directiva page.

Pero se debe tener en cuenta que si el atributo autoFlush de la directiva page tiene el valor true, se generará un error interno del servidor, si el contenido del búfer ya ha sido enviado a la salida que recibe el cliente.

Este error se dará igualmente si se ha enviado ya algún contenido del búfer de salida por otro medio. De esta forma el Código Fuente 210 generará un error como el que se puede observar en la Figura 96.

```
<%@ page errorPage="paginaErrorUno.jsp" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Genera excepción</title>
</head>
<body>
<%out.flush();
int i=1/0;%>
</body>
</html>
```

Código Fuente 210

Una vez que ya hemos indicado la página de error que va a utilizarse en una página determinada, debemos diseñar la propia página de error.

Esta página de error puede ser tan sencilla como mostrar un simple mensaje en HTML informando al usuario que se ha producido un error y que debe ponerse en contacto con el administrador del sistema, así evitamos que el usuario se encuentre con un mensaje en inglés y con un ininteligible volcado de pila de la ejecución actual.

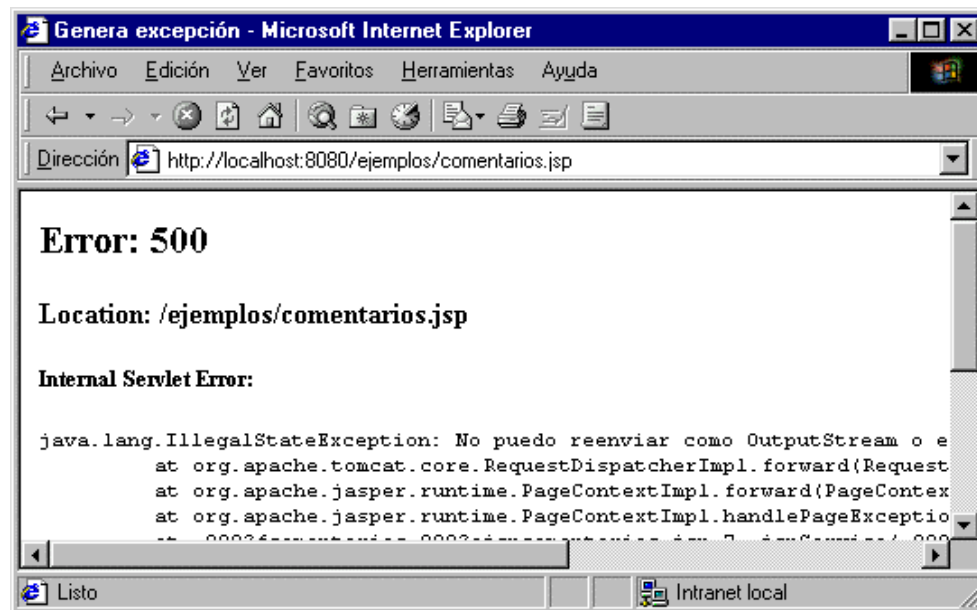


Figura 96. Error de redirección

En el Código Fuente 211 se muestra la página JSP que va a realizar la función de página de error.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Página de error</title>
</head>
<body>
  <%String email=application.getInitParameter("email");%>
  Se ha producido un error, póngase en contacto con el administrador<br>
  del sistema (<a href="mailto:<%=email%>"><%=email%></a>)
</body>
</html>
```

Código Fuente 211

Como se puede comprobar se ha utilizado un parámetro de inicialización de la aplicación Web, que en este caso contiene la dirección de correo electrónico del administrador del sistema. La definición de parámetros de inicialización que debe tener el fichero de configuración la aplicación Web, es decir, el fichero WEB.XML correspondiente, es el que se muestra en el Código Fuente 212.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <context-param>
    <param-name>email</param-name>
    <param-value>aesteban@eidos.es</param-value>
    <description>
      Dirección de correo del administrador del sistema
    </description>
```

```
</context-param>  
</web-app>
```

Código Fuente 212

Si ejecutamos la página JSP que genera la excepción, en su versión correcta, obtendremos el resultado de la Figura 97. Como se puede observar en la Figura 97 no aparece en la barra de direcciones del navegador la ruta de la página de error, sino que aparece la URL de la página JSP en la que se ha producido el error, por lo tanto el usuario nunca advierte que ha sido redirigido hacia una página de tratamiento de errores.



Figura 97. Mensaje de error generado por la página de error

La página de error no tiene porque ser una página JSP, sino que puede tratarse perfectamente de un fichero HTML estático que muestre un mensaje al usuario. En el caso de que la página de error sea una página JSP y queramos hacer uso del objeto integrado `exception` para mostrar información detallada acerca del error que se ha producido debemos asignar al atributo `isErrorPage` de la directiva `page` el valor `true`.

En una misma página JSP es posible utilizar los dos tratamientos de errores disponibles para JSP, es decir, los bloques `try{}catch` y la página de error, en este caso si se produce una excepción primero se comprueba si coincide con el tipo de excepciones que captura el bloque `try{}catch`, en caso afirmativo se ejecutará la rama del `catch` correspondiente, y en caso contrario se lanzará la excepción y se redirigirá hacia la página de tratamiento de errores.

En el siguiente apartado se muestra como utilizar el objeto integrado `exception` dentro de una página de error.

## El objeto `exception`

Instancia de la clase `java.lang.Throwable`, representa una excepción lanzada en tiempo de ejecución. Este objeto únicamente se encuentra disponible en las páginas de error, indicado mediante el atributo `isErrorPage` de la directiva `page`, por lo tanto no se encuentra disponible de forma automática en cualquier página JSP, sólo en aquellas que sean utilizadas para el tratamiento de errores.

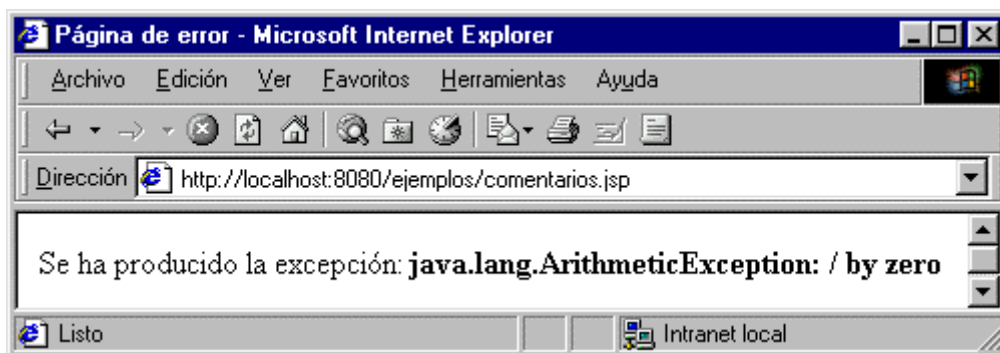
El objeto `exception` presenta los siguientes métodos, que son los métodos que presenta la clase `Throwable`:

- `Throwable fillInStackTrace()`: rellena la pila de ejecución, este método es útil cuando queremos relanzar una excepción que ya se ha producido.
- `String getLocalizedMessage()`: devuelve en forma de un objeto `String` el mensaje de error que representa a la excepción que se ha producido. Normalmente devuelve el mismo mensaje que el siguiente método.
- `String getMessage()`: devuelve el mensaje de error que se corresponde con la excepción.
- `void printStackTrace()`: imprime el objeto `Throwable` y su pila de ejecución en el canal (stream) de error estándar.
- `void printStackTrace(PrintStream s)`: imprime el objeto `Throwable` y su pila de ejecución en el canal de impresión indicado por parámetro.
- `void printStackTrace(PrintWriter s)`: imprime el objeto `Throwable` y su pila de ejecución en el canal `PrintWriter` indicado.
- `String toString()`: devuelve una breve descripción del objeto.

A continuación en el Código Fuente 213 vamos a mostrar el objeto `exception` en acción, este código se corresponde con el de una página JSP para el tratamiento de errores. La utilización que se hace del objeto `exception` es muy básica, consiste únicamente en mostrar en pantalla un breve descripción del objeto. Si modificamos la página JSP que genera la excepción de la división por cero para que utilice esta nueva página de error, obtendremos el resultado de la Figura 98.

```
<%@ page isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Página de error</title>
</head>
<body>
Se ha producido la excepción: <b><%=exception.toString()%></b>
</body>
</html>
```

Código Fuente 213

Figura 98. Utilización básica del objeto `exception`

Como se puede comprobar en este caso si que es necesario utilizar el atributo `isErrorPage` de la directiva `page`, para indicar que la página JSP actual es una página de error y así se nos permitirá utilizar el objeto integrado `exception`.

Pero podemos utilizar el objeto `exception` haciendo uso de otros métodos que posee para ofrecer de esta forma una descripción más detallada de la excepción que se ha producido, esto lo podemos observar en el Código Fuente 214, que se corresponde con el código de una nueva página de error. Y el nuevo resultado que nos devolverá esta página de error se puede ver en la Figura 99, como se puede comprobar nos muestra una descripción del error que se ha producido mucho más detallada.

```
<%@ page isErrorPage="true" import="java.io.*"%>
<html>
<head>
    <title>Página de error</title>
</head>
<body>
<h3>Se ha producido una excepción</h3>
<b>ERROR:</b> <%=exception.toString()%><br>
<b>MENSAJE:</b> <%=exception.getMessage()%><br>
<b>VOLCADO DE PILA:</b>
<%=StringWriter sSalida=new StringWriter();
PrintWriter salida=new PrintWriter(sSalida);
exception.printStackTrace(salida);%>
<%=sSalida%>
</body>
</html>
```

Código Fuente 214

Dentro de la página de error tenemos acceso a todos los atributos con ámbito de petición de la página que generó el error.

En el siguiente apartado vamos a mostrar un ejemplo algo más complejo del tratamiento de errores, a través de un componente JavaBean que realiza las funciones de una calculadora básica cuyos datos para operar se facilitan desde un formulario de una página JSP.

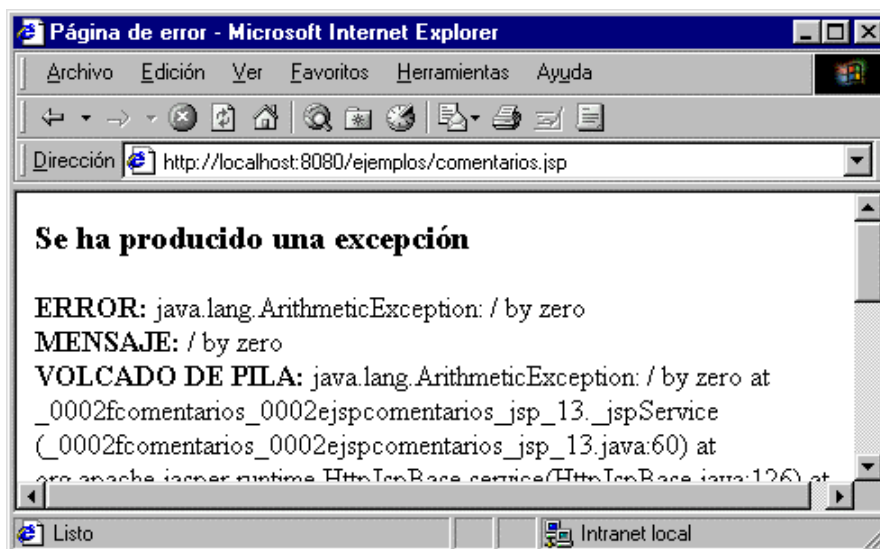


Figura 99. Excepción detallada

## Ejemplo de tratamiento de errores

En este apartado vamos a mostrar un ejemplo práctico del tratamiento de errores en las páginas JSP a través de la utilización de páginas de error.

Este ejemplo consiste en la utilización de un componente JavaBean representado por la clase `CalculadoraBean`, cuyo código fuente se muestra a continuación (Código Fuente 215).

Este Bean nos va a permitir realizar cuatro operaciones aritméticas básicas sobre dos números enteros. Para ello el componente posee tres propiedades de lectura y escritura que son `operando1`, `operando2` y `operacion`, y además ofrece una propiedad de sólo lectura llamada `solucion`. En la Tabla 19 se muestra la hoja de propiedades que le corresponde al componente `CalculadoraBean`.

```
public class CalculadoraBean{
    private int op1;
    private int op2;
    private String operacion;

    public void setOperando1(int op1){
        this.op1=op1;
    }

    public int getOperando1(){
        return op1;
    }

    public void setOperando2(int op2){
        this.op2=op2;
    }

    public int getOperando2(){
        return op2;
    }

    public void setOperacion(String valor){
        operacion=valor;
    }

    public String getOperacion(){
        return operacion;
    }

    public int getSolucion(){
        if(operacion.equals("Resta")){
            return op1-op2;
        }else if(operacion.equals("Suma")){
            return op1+op2;
        }else if(operacion.equals("Multiplicación")){
            return op1*op2;
        }else if(operacion.equals("División")){
            return op1/op2;
        }else return 0;
    }
}
```

Código Fuente 215



Nombre	Acceso	Tipo Java	Valor de ejemplo
operando1	Lectura/escritura	int	2
operando2	Lectura/escritura	int	11
operacion	Lectura/escritura	String	Suma
solucion	Sólo lectura	int	13

Tabla 19. Hoja de propiedades de CalculadoraBean

La página JSP que va a utilizar este Bean contiene un formulario que nos va a permitir indicar los valores de las propiedades del Bean para poder realizar las operaciones correspondientes y obtener los resultados. Una vez enviado el formulario se instanciará el Bean CalculadoraBean y se le asignarán los valores de sus propiedades que coinciden en nombre con el de los parámetros enviados por el formulario a través de objeto request, por lo tanto podemos asignárselos directamente los valores de las propiedades al Bean desde la petición realizada.

A continuación se obtienen los distintos valores de las propiedades para mostrar el resultado de la operación realizada por el componente. El Código Fuente 216 muestra el código completo de la página JSP incluyendo la indicación de la página de error que se va a utilizar, cuyo código veremos más adelante. En la Figura 100 se puede ver un ejemplo de ejecución de esta página.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page errorPage="paginaErrorCuatro.jsp" %>
<html>
<head>
    <title>Calculadora</title>
</head>

<body>
<form action="calculadora.jsp" method="GET">
Operando 1:<input type="text" name="operando1" size="4" value="">
Operación:<select name="operacion">
<option value="Resta">Resta
<option value="Suma">Suma
<option value="Multiplicación">Multiplicación
<option value="División">División
</select>
Operando 2:<input type="text" name="operando2" size="4" value="">
<input type="submit" name="enviar" value="Calcular">
</form>
<%if(request.getParameter("enviar")!=null){%>
    <jsp:useBean id="calculadora" class="CalculadoraBean"/>
    <jsp:setProperty name="calculadora" property="*" />
    El resultado de la operación <b><jsp:getProperty name="calculadora"
property="operacion"/></b>
sobre los operandos <b><jsp:getProperty name="calculadora"
property="operando1"/></b> y
<b><jsp:getProperty name="calculadora" property="operando2"/></b> es igual a
<jsp:getProperty name="calculadora" property="solucion"/>
<%} %>
</body>
</html>
```

Código Fuente 216



Figura 100. utilizando el Bean CalculadoraBean

La página de error que va utilizar la página JSP anterior, además de utilizar el objeto exception, también utiliza un parámetro de inicialización de la aplicación Web (el de correo electrónico del administrador del sitio Web) y como información adicional se muestran todos los parámetros que se encontraban presentes en la petición de la página JSP original. El Código Fuente 217 es el código de la página de error utilizada en este ejemplo. Nuestras páginas de error pueden ser todo lo sencillas o complejas que queramos.

```
<%@ page import="java.util.*,java.io.*" isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Página de error</title>
</head>
<body>
    <%String email=application.getInitParameter("email");%>
    Se ha producido un error, póngase en contacto con el administrador<br>
    del sistema (<a href="mailto:<%=email%>"><%=email%></a>)<br><hr>
    <h3>Información para personal técnico</h3>
    <TABLE BORDER="1" ALIGN="CENTER">
    <caption><b>Parámetros enviados</b></caption>
    <TR BGCOLOR="#FFAD00">
    <TH>Nombre parámetro</TH>
    <TH>Valor parámetro</TH>
    <%Enumeration nombresParametros= request.getParameterNames();
    while(nombresParametros.hasMoreElements()) {
        String nombre = (String)nombresParametros.nextElement();%>
        <TR><TD><%=nombre%></TD>
        <TD><%=request.getParameter(nombre)%></TD>
    <%}%>
    </TABLE>
    <b>ERROR:</b> <%=exception.toString()%><br>
    <b>MENSAJE:</b> <%=exception.getMessage()%><br>
    <b>VOLCADO DE PILA:</b>
    <%StringWriter sSalida=new StringWriter();
    PrintWriter salida=new PrintWriter(sSalida);
    exception.printStackTrace(salida);%>
    <%=sSalida%>
</body>
</html>
```

Código Fuente 217

Un posible error que se puede producir al utilizar el componente CalculadoraBean es realizar la operación de la división por cero. Si intentamos realizar esta operación se mostrará la página de error correspondiente, cuyo aspecto es el de la Figura 101.

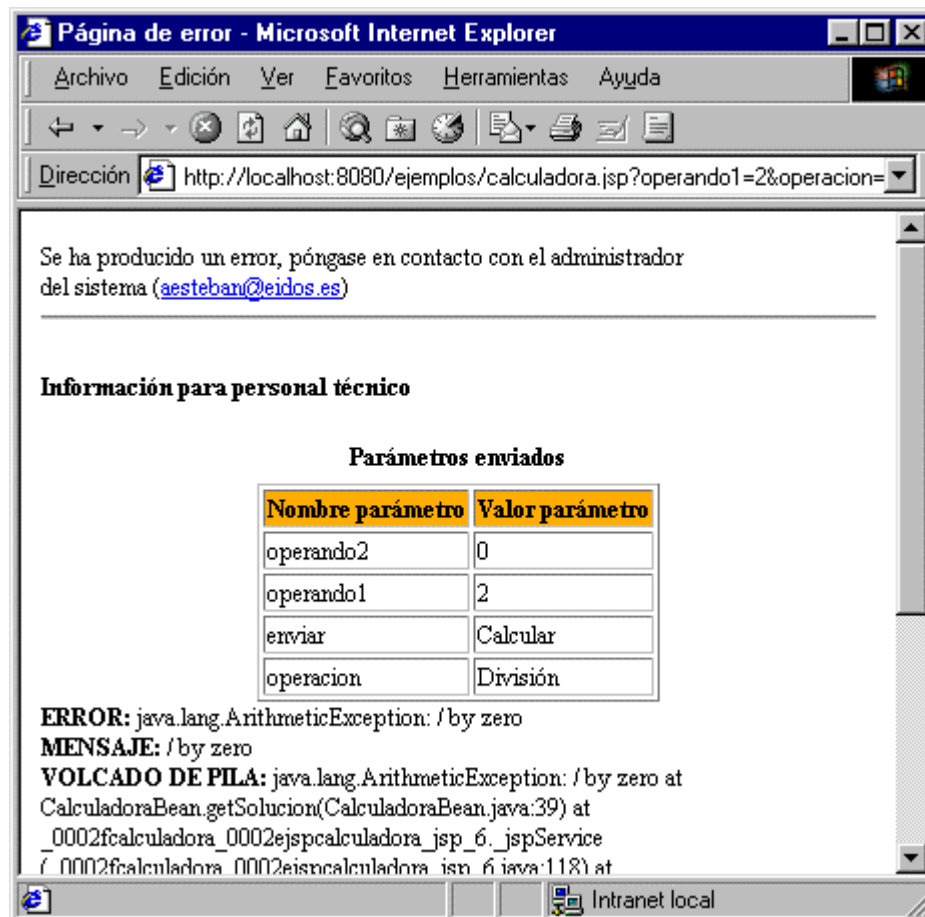


Figura 101. Capturando las excepciones de la calculadora

Como curiosidad comentaremos que en la petición que carga la página de error una vez se encuentra un atributo con ámbito de petición llamado `javax.servlet.jsp.jspException`, es decir, como el nombre de la clase de excepción de JSP, que contiene exactamente la misma información que el objeto integrado `exception`, en el siguiente apartado veremos que este atributo del objeto `request` es bastante útil para realizar el tratamiento de errores de los servlets a través de páginas de error JSP.

En el Código Fuente 218 se muestra una página JSP de error que no utiliza el objeto `exception`, sino que utiliza el atributo correspondiente del objeto `request`.

```
<%@ page isErrorPage="true"%>
<html>

<head>
    <title>Página de error</title>
</head>

<body>
<h3>Se ha producido una excepción</h3>
<b>ERROR:</b>
```

```
<%=request.getAttribute("javax.servlet.jsp.jspException")%>
</body>
</html>
```

Código Fuente 218

En este apartado además de ver un ejemplo completo de tratamiento de errores hemos repasado la utilización de los componentes JavaBeans desde las páginas JSP, y también hemos desarrollado un componente JavaBean, que fue lo que vimos en los dos capítulos anteriores.

En el siguiente y último apartado vamos a comentar como utilizar las páginas JSP de error desde los servlets, par así tener un tratamiento de errores centralizado, de esta forma siempre se cargarán las páginas de error indicadas con independencia de si nos encontramos en una página JSP o en un servlet.

## Tratamiento de errores en los servlets

Las páginas de error las podemos utilizar también en los servlet, debemos recordar que la relación entre páginas JSP y servlets es bastante estrecha, ya que una página JSP se traduce a un servlet equivalente a la hora de ejecutarse.

Por lo tanto podemos utilizar el mecanismo de las páginas de error JSP como un tratamiento de errores centralizado, que nos sirva de forma general en nuestra aplicación Web tanto para páginas JSP como para servlets, de esta forma no nos tenemos que preocupar en distinguir si un error se produce en una página JSP o bien en un servlet de la aplicación Web.

Para utilizar las páginas de error en los errores dentro de un servlet, primero debemos incluir las instrucciones que pueden generar una excepción dentro de un bloque try{}catch.

En el bloque de sentencias correspondientes a la rama de catch debemos establecer el valor del atributo javax.servlet.jsp.jspException con la excepción que se haya producido, y a continuación debemos obtener el objeto RequestDispatcher que va a representar a la página JSP de tratamiento de errores.

Recordamos que el interfaz javax.servlet.RequestDispatcher tiene la función de representar a un recurso dentro del servidor, que podrá ser una página JSP, un servlet, una página HTML, etc., y que nos va a permitir redirigir la petición de un cliente a ese recurso, o bien incluir el resultado de la ejecución del recurso o su contenido, dentro del servlet actual.

Una vez que ya hemos establecido el atributo del objeto request y tenemos la referencia a la página JSP de error a través de un objeto RequestDispatcher, redirigimos al ejecución del servlet actual hacia la página JSP de error utilizando el método forward() del interfaz RequestDispatcher.

Vamos a ilustrar este tratamiento de errores mediante un sencillo ejemplo que consulte en un servlet que genera una excepción de división por cero. El Código Fuente 219 es el código del servlet, y el código de la página JSP de error que va a utilizar el servlet es el Código Fuente 220.

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

public class ServletExcepcion extends HttpServlet {
```

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{

    try{
        int i=1/0;
    }catch(Exception ex){
        request.setAttribute("javax.servlet.jsp.jspException",ex);
        RequestDispatcher
rd=request.getRequestDispatcher("/paginaErrorCuatro.jsp");
        rd.forward(request,response);
    }
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException{
    doGet(request,response);
}
}

```

Código Fuente 219

```

<%@ page import="java.io.*" isErrorPage="true" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>

<head>
    <title>Página de error</title>
</head>

<body>
    <%String email=application.getInitParameter("email");%>
    Se ha producido un error, póngase en contacto con el administrador<br>
    del sistema (<a href="mailto:<%=email%>"><%=email%></a><br><hr>
    <h3>Información para personal técnico</h3>
    <b>ERROR:</b> <%=exception.toString()%><br>
    <b>MENSAJE:</b> <%=exception.getMessage()%><br>
    <b>VOLCADO DE PILA:</b>
    <%StringWriter sSalida=new StringWriter();
    PrintWriter salida=new PrintWriter(sSalida);
    exception.printStackTrace(salida);%>
    <%=sSalida%>
</body>

</html>

```

Código Fuente 220

Como se puede observar esta página es igual a las que hemos estado utilizando para tratar los errores que se producen dentro de las páginas JSP.

La ejecución del servlet anterior generará el resultado de la Figura 102.

Con este apartado concluye el capítulo dedicado al tratamiento de errores en las páginas JSP, en el siguiente apartado veremos el acceso a datos desde páginas JSP, para este acceso a datos se utilizará el API del lenguaje Java. Especializado en esta tarea, se trata del API JDBC.

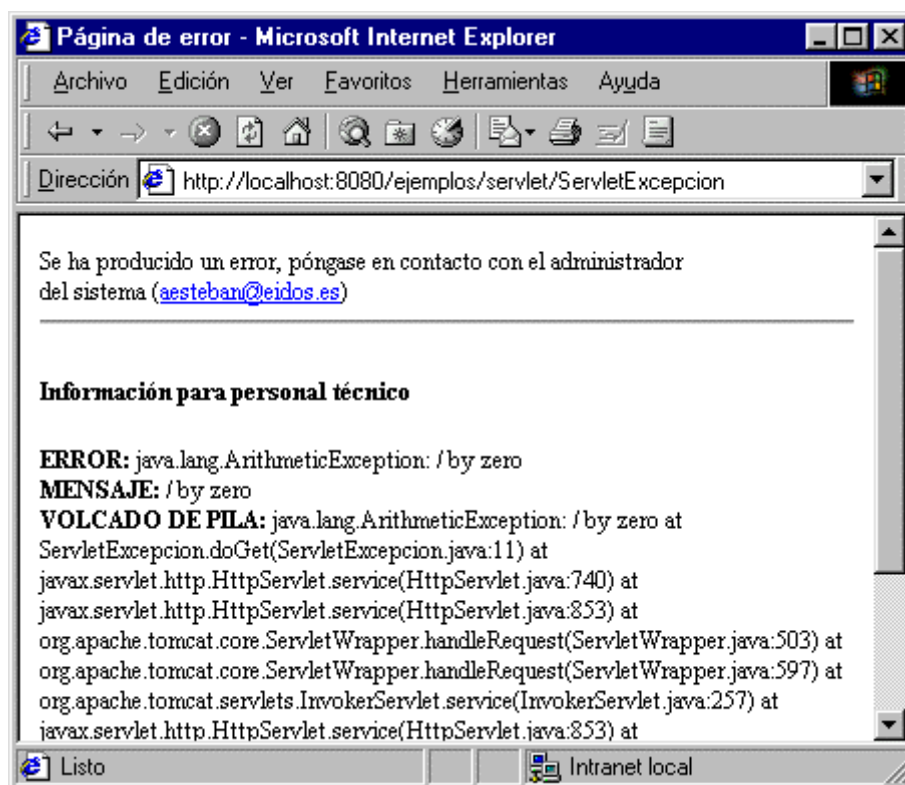


Figura 102. Excepción que se produce en un servlet

# Acceso a datos desde JSP I

---

## Introducción

En este capítulo y el siguiente vamos a tratar el acceso a bases de datos desde las páginas JSP, en la parte del texto dedicada a los servlet no se ofrecía ningún capítulo en el que se tratara el acceso a bases de datos, esto es debido a que tanto los servlets como las páginas JSP van a utilizar el API de acceso a bases de datos que ofrece el lenguaje Java, y este API es común a cualquier tecnología Java, ya sea a aplicaciones convencionales, applets, servlets o páginas JSP.

Se ha decidido explicar con las páginas JSP el acceso a datos debido a que ofrece mecanismos muy potentes a través de las acciones estándar para la utilización de componentes JavaBeans, que nos van a permitir utilizar Beans en las páginas JSP especializados en el acceso a datos.

El API que ofrece el lenguaje Java para el acceso a bases de datos es el API JDBC. El lector que ya conozca o se encuentre familiarizado con el acceso a bases de datos a través del API JDBC, comprobará que es idéntico la forma de utilizar JDBC en una aplicación Java convencional que en una página JSP. De todas formas vamos a comentar brevemente los aspectos más importantes del API JDBC, y veremos distintos ejemplos de utilización de JDBC desde páginas JSP, y también a través de componentes JavaBeans.

Como se puede comprobar, una de las virtudes de las páginas JSP es que se basan en el lenguaje Java, y por lo tanto pueden utilizar cualquier API del propio lenguaje. Aquí siempre entendemos como API a un conjunto de clases e interfaces relacionados entre sí que permiten una funcionalidad determinada.

## Introducción a JDBC

JDBC es un API incluido dentro del lenguaje Java para el acceso a bases de datos. Consiste en un conjunto de clases e interfaces escritos en Java que ofrecen un completo API para la programación de bases de datos, por lo tanto es la una solución 100% Java que permite el acceso a bases de datos, la primera aparición de JDBC (JDBC 1.0) se encuentra dentro del paquete `java.sql` que ha fue incorporado en la versión del JDK 1.1.x (Java Development Kit) correspondiente a la versión 1.1 del lenguaje Java, JDBC 2.0 sigue estando en el mismo paquete pero en las versiones JDK 1.2 y JDK 1.3 que se corresponden con la versión 2 del lenguaje Java, o también denominada plataforma Java 2 (Java 2 Platform).

JDBC es un especificación formada por una colección de interfaces y clases abstractas, que deben implementar todos los fabricantes de drivers que quieran realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver)

Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma. No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, o SQL Server. Además podrá ejecutarse en cualquier sistema que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma.

Básicamente el API JDBC hace posible la realización de las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular los datos.
- Procesar los resultados de la ejecución de las sentencias.

La columna vertebral de JDBC es el Driver Manager (gestor de drivers) que se encuentra representado por la clase `java.sql.DriverManager`. El gestor de drivers es pequeño y simple y su función primordial es la de seleccionar el driver adecuado para conectar la aplicación, applet, servlet o página JSP con una base de datos determinada, y acto seguido desaparece.

Se puede considerar que JDBC ofrece dos conjuntos de clases e interfaces bien diferenciados, aquellas de más alto nivel que serán utilizados por los programadores de aplicaciones para el acceso a bases de datos, y otras de más bajo nivel enfocadas hacia los programadores de drivers que permiten la conexión a una base de datos. En el presente texto nos vamos a centrar en el primer subconjunto, el de más alto nivel.

## El paquete `java.sql`

El API JDBC está formado por una serie de interfaces (es decir, la definición de sus métodos se encuentra vacía por lo que estos interfaces ofrecen simplemente plantillas de comportamiento que otras clases esperan implementar) que permiten al programador abrir conexiones con bases de datos, ejecutar sentencias SQL sobre las mismas, procesar y modificar los resultados.

Si consultamos al documentación del paquete `java.sql` podremos ver todos los interfaces y clases que forman parte del API para el acceso a bases de datos. A continuación podemos ver el índice del



paquete `java.sql` en el que se muestra todas las clases, interfaces y excepciones que se encuentran dentro de este paquete.

- Interfaces: `CallableStatement`, `Connection`, `DatabaseMetaData`, `Driver`, `PreparedStatement`, `ResultSet`, `ResultSetMetaData`, `Statement`, `SQLData`, `SQLInput`, `SQLOutput`, `Array`, `Blob`, `Clob`, `Ref`, `Struct`.
- Clases: `Date`, `DriverManager`, `DriverPropertyInfo`, `Time`, `Timestamp`, `Types`, `SQLPermission`
- Excepciones: `DataTruncation`, `SQLException`, `SQLWarning`, `BatchUpdateException`

De esta enumeración de interfaces, clases y excepciones los más importantes son:

- `DriverManager`: es la clase gestora de los drivers. Esta clase se encarga de cargar y seleccionar el driver adecuado para realizar la conexión con una base de datos determinada.
- `Connection`: representa una conexión con una base de datos.
- `Statement`: actúa como un contenedor para ejecutar sentencias SQL sobre una base de datos. Este interfaz tiene otros dos subtipos: `java.sql.PreparedStatement` para la ejecución de sentencias SQL precompiladas a las que se le pueden pasar parámetros de entrada; y `java.sql.CallableStatement` que permite ejecutar procedimientos almacenados de una base de datos.
- `ResultSet`: controla el acceso a los resultados de la ejecución de una consulta, es decir, de un objeto `Statement`, permite también la modificación de estos resultados.
- `SQLException`: para tratar las excepciones que se produzcan al manipular la base de datos, ya sea durante el proceso de conexión, desconexión u obtención y modificación de los datos.
- `BatchUpdateException`: excepción que se lanzará cuando se produzca algún error a la hora de ejecutar actualizaciones en batch sobre la base de datos.
- `Warning`: nos indica los avisos que se produzcan al manipular y realizar operaciones sobre la base de datos.
- `Driver`: este interfaz lo deben implementar todos los fabricantes de drivers que deseen construir un driver JDBC. Representa un driver que podemos utilizar para establecer una conexión con la base de datos.
- `Types`: realizan la conversión o mapeo de tipos estándar del lenguaje SQL a los tipos de datos del lenguaje Java.
- `ResultSetMetaData`: este interfaz ofrece información detallada relativa a un objeto `ResultSet` determinado.
- `DatabaseMetaData`: ofrece información detallada sobre la base de datos a la que nos encontramos conectados.

Todas estas clases e interfaces las podemos agrupar también atendiendo a las funciones que cumplen:

- Para establecer una conexión a un origen de datos: `DriverManager`, `DriverPropertyInfo` y `DriverConnection`.

- Envío de sentencias SQL a la base de datos: Statement, PreparedStatement y CallableStatement
- Obtención y modificación de los resultados de una sentencia SQL: ResultSet.
- Para el mapeo de datos SQL a tipos de datos del lenguaje Java: Array, Blob, Clob, Date, Ref, Struct, Time, Timestamp y Types.
- Mapeo de tipos SQL definidos por el usuario a clases del lenguaje Java: SQLData, SQLInput y SQLOutput
- Ofrecer información sobre la base de datos y sobre las columnas de un objeto ResultSet: DatabaseMetaData y ResultSetMetaData.
- Excepciones que se pueden producir en operaciones de JDBC: SQLException, SQLWarning, DataTruncation y BatchUpdateException.
- Ofrece seguridad: SQLPermission.

## Drivers JDBC

Los drivers nos permiten conectarnos con una base de datos determinada. Existen cuatro tipos de drivers JDBC, cada tipo presenta una filosofía de trabajo diferente, a continuación se pasa a comentar cada uno de los drivers:

1. JDBC-ODBC bridge plus ODBC driver (tipo 1): este driver fue desarrollado entre Sun e Intersolv y permite al programador acceder a fuentes de datos ODBC existentes mediante JDBC. El JDBC-ODBC Bridge (puente JDBC-ODBC) implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete sun.jdbc.odbc (que se encuentra incluido dentro del JDK a partir de la versión 1.1) y contiene librerías nativas para acceder a ODBC. Se debe señalar que en cada máquina cliente que utilice el driver es necesaria una configuración previa, es decir, deberemos definir la fuente de datos utilizando para ello el gestor de drivers ODBC que los podemos encontrar dentro del Panel de Control de Windows.
2. Native-API partly-Java driver (tipo 2): son similares a los drivers de tipo 1, en tanto en cuanto también necesitan una configuración en la máquina cliente. Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros Sistemas Gestores de Bases de Datos (SGBD).
3. JDBC-Net pure Java driver (tipo 3): este driver traduce las llamadas JDBC a un protocolo independiente del DBMS (DataBase Management System, Sistema Gestor de Bases de Datos SGBD) que será traducido por un servidor para comunicarse con un DBMS concreto. Con este tipo de drivers no se necesita ninguna configuración especial en el cliente, permiten el diálogo con un componente negociador encargado de dialogar a su vez con las bases de datos. Es ideal para aplicaciones con arquitectura basada en el modelo de tres capas. Un ejemplo de utilización de este driver puede ser un applet que se comunica con una aplicación intermediaria en el servidor desde el que se descargó y es esta aplicación intermediaria la encargada de acceder a la base de datos. Esta forma de trabajo hace a este tipo de drivers ideales para trabajar con Internet y grandes intranets, ya que el applet es independiente de la plataforma y puede ser descargado completamente desde el servidor Web. El intermediario puede estar escrito en cualquier lenguaje de programación, ya que se ejecutará en el servidor.

Pero si el intermediario se programa en Java se tendrá la ventaja de la independencia de la plataforma, además se debe tener en cuenta que el intermediario al ser una aplicación Java no posee las restricciones de seguridad de los applets, por lo que se podrá acceder a cualquier servidor, no solamente desde el que se cargó el applet. El problema que presentan este tipo de drivers es que su utilización es bastante compleja.

4. Native-protocol pure Java driver (tipo 4): esta clase de driver convierte directamente las llamadas en JDBC al protocolo usado por el DBMS. Esto permite una comunicación directa entre la máquina cliente y el servidor en el que se encuentra el DBMS. Son como los drivers de tipo 3 pero sin la figura del intermediario y tampoco requieren ninguna configuración en la máquina cliente. Estos drivers utilizan protocolos propietarios, por lo tanto serán los propios fabricantes de bases de datos los que ofrecerán estos drivers, ya existen diferentes fabricantes que se han puesto en marcha, en el presente texto se va a mostrar un driver de tipo 4 llamado i-net Sprinta 2000 de la compañía germana i-net software, la versión de evaluación de este driver se puede obtener en la dirección <http://www.inetsoftware.de>. Este driver implementa la versión 2.0 de JDBC. Mediante este driver podemos acceder a bases de datos en el servidor de bases de datos de Microsoft SQL Server 7.

La propia JavaSoft indica que es preferible el uso de drivers de tipo 3 y 4 para acceder a bases de datos a través de JDBC. Las categorías 1 y 2 deberán ser soluciones provisionales hasta que aparezca algún driver del tipo anterior. Las categorías 3 y 4 ofrecen todas las ventajas de Java, incluyendo la instalación automática, por ejemplo, cargando el driver JDBC cuando un applet necesite utilizarlo.

Existe un gran número de drivers distintos disponibles, los drivers disponibles se pueden consultar en la dirección <http://industry.java.sun.com/products/jdbc/drivers>, en esta dirección se ofrece un formulario de búsqueda que permite seleccionar el tipo de driver que estamos buscando junto con el sistema gestor de base de datos al que queremos acceder y la versión de JDBC que implementa el driver. Este punto es importante, debemos estar seguros que el driver que vamos a utilizar soporte la versión 2.0 de JDBC, cada vez existen más drivers disponibles que soportan la última versión de JDBC.

Para el presente texto se van a mostrar distintos ejemplos con dos tipos de drivers, el driver de tipo 1 JDBC-ODBC bridge, y el driver de tipo 4 Sprinta. Aunque los conceptos de JDBC y los distintos ejemplos serán válidos para cualquier tipo de driver, el tipo de driver únicamente interesa a la hora de establecer la conexión con la base de datos (en lo que se refiere a la cadena de conexión), el resto del código Java será exactamente igual para distintos tipos de drivers.

## URLs de JDBC

Una URL de JDBC ofrece un mecanismo para identificar una base de datos de forma que el driver adecuado la reconozca y establezca una conexión con ella. Al igual que las URLs generales sirven para identificar de un modo único un recurso en Internet, en este caso una URL de JDBC localizará un base de datos determinada. La sintaxis estándar de las URLs de JDBC es la siguiente:

```
jdbc:<subprotocolo>:<subnombre>
```

Siempre debe comenzar por jdbc pero es el fabricante del driver quién debe especificar la sintaxis exacta de la URL para su driver JDBC. El subprotocolo suele ser el nombre del driver o del mecanismo de conectividad con la base de datos y el subnombre es una forma de identificar a la base de datos concreta.

El subnombre puede variar ya que puede identificar una base de datos local, una base de datos remota, un número de puerto específico o una base de datos que requiera identificador de usuario y contraseña.

En el caso de utilizar el puente JDBC-ODBC (driver de tipo 1) se utilizará el subprotocolo odbc, que tiene la siguiente sintaxis:

```
jdbc:odbc:<nombre de la fuente de datos>[;<nombre atributo> = <valor atributo>]*
```

A continuación se muestran un par de ejemplos:

```
jdbc:odbc:FuenteBD  
jdbc:odbc:FuenteBD;ID=pepe;PWD=ssshhh
```

En el primer caso nos conectamos a una fuente de datos de ODBC llamada FuenteBD, y en el segundo caso también pero especificando el usuario (ID) y la contraseña (PWD) correspondiente.

En nuestros ejemplos, además de utilizar un driver de tipo 1 para acceder a fuentes de datos ODBC, vamos a utilizar un driver de tipo 4. En este caso es de la compañía i-net software y es denominado Sprinta. Este driver de tipo 4 lo vamos a utilizar para acceder directamente a bases de datos que se encuentren en servidores de bases de datos de Microsoft SQL Server 6.5/7 y 2000.

El driver Sprinta (que se puede obtener en la dirección <http://www.inetsoftware.de>) tiene una sintaxis diferente para las URLs de JDBC, como ya hemos comentado cada fabricante de driver posee una sintaxis, que ajustándose al esquema anterior, tiene una serie de variaciones y particularidades específicas.

La sintaxis para construir URLs de JDBC para el driver Sprinta se indica en la documentación que se ofrece junto con las clases del driver, por ejemplo, si queremos acceder a una base de datos llamada MiBD que reside en el servidor de bases de datos llamado [www.eidos.es](http://www.eidos.es) y que está escuchando en el puerto 1433 deberemos escribir la siguiente URL de JDBC:

```
jdbc:inetdae:www.eidos.es:1433?database=MiDB
```

Si a la base de datos anterior nos queremos conectar con el usuario pepe que posee la contraseña xxx, escribiríamos la siguiente URL:

```
jdbc:inetdae:www.eidos.es:1443?database=MiDB&user=pepe&password=xxx
```

La sintaxis general de para las URLs de JDBC de este driver es la siguiente:

```
jdbc:inetdae:servidor:puerto?[propiedad1=valorpropiedad&...propiedad  
n=valorpropiedadn]
```

Como se puede observar, la parte que corresponde al subprotocolo sería inetdae y la parte que corresponde al subnombre sería: servidor:puerto?propiedades.

## Realizando una conexión con la base de datos

La forma común de establecer una conexión con una base de datos es llamar al método `getConnection()` de la clase `DriverManager`, a este método se le debe pasar como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión. La ejecución de este método devolverá un objeto `Connection` que representará la conexión con la base de datos.

Este método está sobrecargado, le podemos pasar solamente la URL, la URL con el identificador de usuario y la contraseña o bien una URL y una lista de propiedades. La sintaxis general para realizar una conexión es la siguiente:

```
Connection conexion=DriverManager.getConnection(url);
```

Antes de realizar una conexión con una base de datos es necesario registrar las clases de los drivers correspondientes. La clase DriverManager mantiene una lista de clases Driver que se han registrado ellas mismas llamando al método registerDriver() de la clase DriverManager. Todas las clases Driver deberían estar escritas con una sección estática que crease una instancia de la clase y acto seguido se registrase con el método estático DriverManager.registerDriver(), este proceso se ejecuta automáticamente cuando se carga la clase del driver.

Por lo tanto gracias al mecanismo anteriormente mencionado, un programador normalmente no debería llamar directamente al método DriverManager.registerDriver(), sino que será llamado automáticamente por el driver cuando es cargado. Una clase Driver es cargada, y por lo tanto automáticamente registrada con el DriverManager, de dos maneras posibles.

La primera de ellas es llamando al método estático forName() de la clase Class, esto carga explícitamente la clase del driver.

La función del método Class.forName es cargar una clase de forma dinámica a partir del nombre completo de la clase que se le pasa como parámetro. Se recomienda esta forma de cargar los drivers. Así por ejemplo para cargar el driver cuya clase es acme.db.Driver, se escribiría el siguiente código:

```
Class.forName("acme.db.Driver");
```

Si el driver acme.db.Driver está implementado correctamente, al cargarse su clase lanzará una llamada al método DriverManager.registerDriver(), y de esta forma ya estará disponible en la lista de drivers del DriverManager para que se pueda realizar una conexión con él.

La segunda forma de registrar un driver de JDBC es creando una instancia de la clase del driver correspondiente, como muestra el siguiente código.

No existe ninguna diferencia a la hora de registrar un driver dentro de una página JSP, podemos utilizar cualquiera de ellas de manera indiferente.

En el Código Fuente 221 se muestra una página JSP que se conecta a una base de datos con el driver Sprinta. Para que el contenedor de páginas JSP encuentre las clases del driver debemos copiarlas al directorio de clases de la aplicación Web, es decir, al directorio WEB-INF\CLASSES.

```
<%@page import="java.sql.*"%>
<html>
<head><title>Conexión</title></head>
<body>
<%try{
    //Se registra el driver
    Class.forName("com.inet.tds.TdsDriver");
    //Driver driver=new com.inet.tds.TdsDriver();
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection
        ("jdbc:inetdae:212.100.2.3:1433?sql7=true&database=pubs&user=sa");
    out.println("Conexión realizada con éxito a: "+conexion.getCatalog());
    conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca%>
    <%= "Se produjo una excepción durante la conexión: "+ex%>
    <%}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
```

```

        <%= "Se produjo una excepción: "+ex%>
    <%}%>
</body>
</html>

```

Código Fuente 221

Si la conexión la realizamos con el driver de tipo 1 (JDBC-ODBC) la página JSP presentaría el Código Fuente 222, en este caso no es necesario copiar las clases del driver al directorio WEB-INF\CLASSES, ya que forman parte del lenguaje.

```

<%@page import="java.sql.*"%>
<html>
<head><title>Conexión</title></head>
<body>
<%try{
    //Se registra el driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    //Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection

    ("jdbc:odbc:FuenteBD","sa","");
    out.println("Conexión realizada con éxito a: "+conexion.getCatalog());
    conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca%>
    <%= "Se produjo una excepción durante la conexión: "+ex%>
    <%}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
    <%= "Se produjo una excepción: "+ex%>
    <%}%>
</body>
</html>

```

Código Fuente 222

En cualquier caso el resultado de la conexión es el de la Figura 103.



Figura 103. Conexión a una base de datos

## Tratando las excepciones

Como se ha comprobado en el apartado anterior, toda acción que realicemos sobre la base de datos (abrir la conexión, cerrarla, ejecutar una sentencia SQL, etc.) va a lanzar una excepción `SQLException` que deberemos atrapar o lanzar hacia arriba en la jerarquía de llamadas de métodos.

La clase `java.sql.SQLException` hereda de la clase `java.lang.Exception`. La clase `java.lang.Exception` recoge todas las excepciones que se producen dentro de un programa Java, pero gracias a la clase `java.sql.SQLException` podemos particularizar el tratamiento de excepciones para las que se producen en el acceso a bases de datos. En la Figura 104 se puede apreciar que lugar ocupa la clase `SQLException` dentro de la jerarquía de clases de Java.

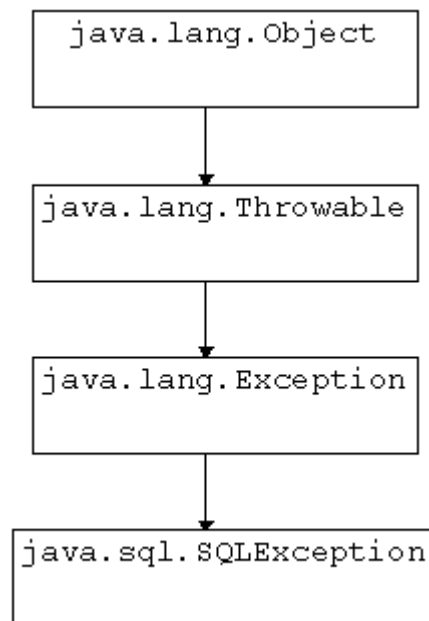


Figura 104. Jerarquía de la clase `SQLException`

Pero la clase `SQLException` no se comporta como el resto de las excepciones, sino que posee algunas particularidades, que vamos a pasar a comentar a continuación.

La principal de estas particularidades es que una excepción `SQLException` puede tener encadenados varios objetos `SQLException`, es decir, si se produce en un acceso a una base de datos tres excepciones, el objeto `SQLException` que se instancia posee una lista con las tres excepciones. Podemos considerar que tiene un "puntero" a la siguiente excepción que se ha producido. Este tipo de excepciones posee métodos que no tiene su clase padre `java.lang.Exception`.

Teniendo en cuenta lo anterior, si queremos obtener todas las excepciones que se han producido, nuestro tratamiento de excepciones se ocupará de recorrer la lista completa.

Los métodos principales de esta clase `SQLException` son los siguientes:

- `int getErrorCode()`: nos devuelve el código de error. Este código será específico de cada fabricante de SGBD.
- `String getSQLState()`: este método nos devuelve el estado SQL que se corresponde con el estándar XOPEN SQLstate.

- `SQLException getNextException()`: mediante este método obtenemos la siguiente excepción que se ha producido, es decir, se desplaza a la siguiente `SQLException` dentro de la lista de objetos `SQLException` existentes.
- `String getMessage()`: nos devuelve un objeto de la clase `String` que describe la excepción que se ha producido.

Todos los métodos anteriores son introducidos por la clase `SQLException`, menos el método `getMessage()` que lo hereda de la clase `Exception`.

Para tratar correctamente la excepción `SQLException` del ejemplo del apartado anterior, añadiremos unas líneas de código que consistirán en un bucle para recorrer todas las excepciones `SQLException` y mostrar información de cada una de ellas en pantalla. El nuevo código se ofrece en el Código Fuente 223.

```
<%@page import="java.sql.*"%>
<html>
<head><title>Conexión</title></head>
<body>
<%try{
    //Se registra el driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    //Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection

    ("jdbc:odbc:FuenteBD","sa","");
    out.println("Conexión realizada con éxito a: "+conexion.getCatalog());
    conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca
    out.println("Se han dado excepciones SQLException<br>");
    System.out.println("=====<br>");
    //Pueden existir varias SQLException encadenadas
    while(ex!=null){
        out.println("SQLState :"+ex.getSQLState()+"<br>");
        out.println("Mensaje :"+ex.getMessage()+"<br>");
        out.println("Código de error :"+ex.getErrorCode()+"<br>");
        ex=ex.getNextException();
        out.println("<br>");
    }
}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
    <%= "Se produjo una excepción: "+ex%>
<%}%>
</body>
</html>
```

Código Fuente 223

Si se produce un error, por ejemplo, si el servidor de base de datos no se encuentra ejecutándose, se producirá un error como el mostrado en la Figura 105.

Cuando diseñamos páginas JSP para el acceso a bases de datos a través de JDBC, además de producirse las ya conocidas excepciones `SQLException`, también se instancian objetos `SQLWarning`.



Aunque la clase `SQLWarning` hereda de la clase `SQLException` no la trataremos como una excepción, sino que se tratarán como advertencias (warnings) o avisos. Un objeto `SQLWarning` no se atrapará, ya que no se lanzan como las excepciones.

En el esquema de la Figura 106 se muestra el lugar que ocupa la clase `SQLWarning` dentro de la jerarquía de clases de Java.

Para poder tratar un aviso `SQLWarning` deberemos obtenerlos explícitamente a través de código, ya que los objetos `SQLWarning` que se vayan generando se irán añadiendo silenciosamente al objeto que las provocó.

Podemos decir que una excepción se lanza y se deberá atrapar y que un aviso `SQLWarning` se recupera.

Casi todos los interfaces del paquete `java.sql` poseen un par de métodos para manejar los avisos `SQLWarning`, estos métodos son `clearWarnings()` y `getWarnings()`. El primero de ellos elimina todas las advertencias, objetos de la clase `SQLWarning` que se han producido, y el segundo de ellos recupera estos objetos `SQLWarning`.

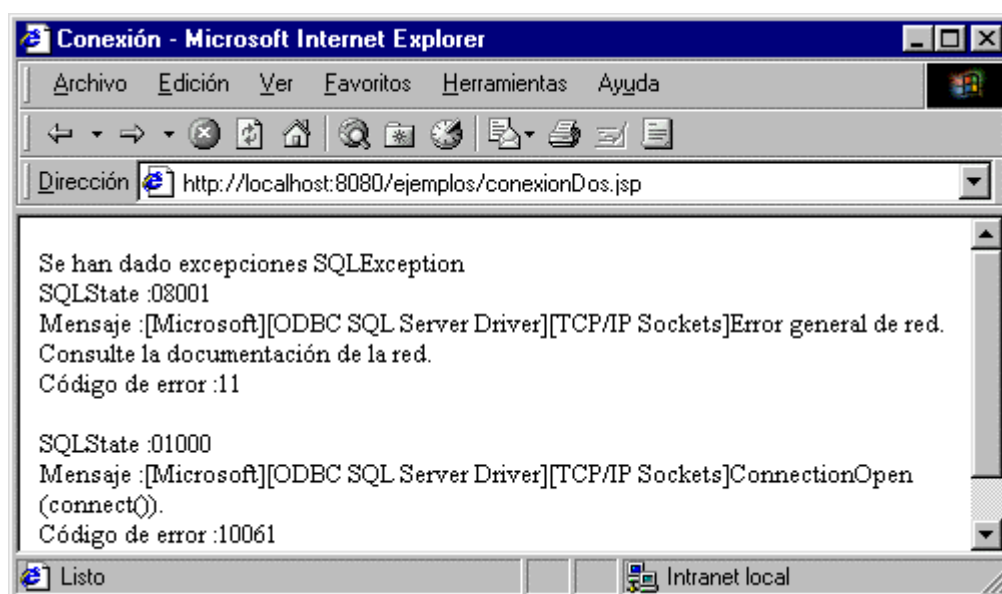
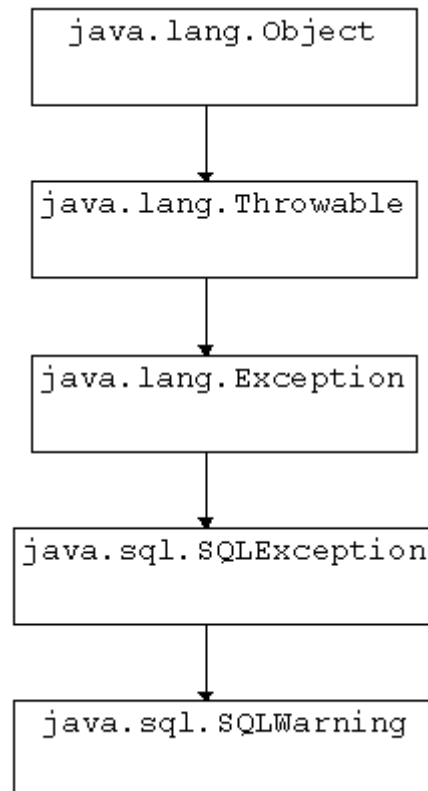


Figura 105. Tratamiento de excepciones

La clase `SQLWarning` básicamente posee los mismos métodos que su clase padre, `SQLException`, pero añade uno bastante útil llamado `getNextWarning()` que nos permite recuperar el siguiente objeto `SQLWarning`, es decir, con esta clase ocurre igual que con su clase padre, tenemos una serie de objetos `SQLWarning` encadenados que debemos recorrer si queremos recuperar la información de cada uno de ellos.

Si retomamos el ejemplo anterior, para mostrar las advertencias que se han producido al realizar la conexión con la base de datos lanzaremos el método `getWarnings()` sobre el objeto `Connection`. Este método nos devolverá el primer objeto `SQLWarning` que contendrá un enlace al siguiente.

El nuevo código del ejemplo es el Código Fuente 224.

Figura 106. Jerarquía de herencia de la clase `SQLWarning`

```

<%@page import="java.sql.*"%>
<html>
<head><title>Conexión</title></head>

<boby>
<%try{
    //Se registra el driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    //Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection

    ("jdbc:odbc:FuenteBD","sa","");
    out.println("Conexión realizada con éxito a: "+conexion.getCatalog()+"<br>");
    SQLWarning warn=conexion.getWarnings();
    if(warn!=null){
        out.println("Aviso(s) producido(s) al conectar<br>");
        out.println("=====<br>");
        while(warn!=null){
            out.println("SQLState : "+warn.getSQLState()+"<br>");
            out.println("Mensaje : "+warn.getMessage()+"<br>");
            out.println("Código de error: "+warn.getErrorCode()+"<br>");
            out.println( "<br>" );
            warn = warn.getNextWarning();
        }
    }
    conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca
    out.println("Se han dado excepciones SQLException<br>");
    System.out.println("=====<br>");
    //Pueden existir varias SQLException encadenadas

```

```

        while(ex!=null){
            out.println("SQLState : "+ex.getSQLState()+"<br>");
            out.println("Mensaje : "+ex.getMessage()+"<br>");
            out.println("Código de error : "+ex.getErrorCode()+"<br>");
            ex=ex.getNextException();
            out.println("<br>");
        }
    }catch(Exception ex){
        //Se captura cualquier tipo de excepción que se produzca%>
        <%= "Se produjo una excepción: "+ex%>
    <%}%>
</body>

</html>

```

Código Fuente 224

Y la Figura 107 muestra una ejemplo de ejecución del código anterior.

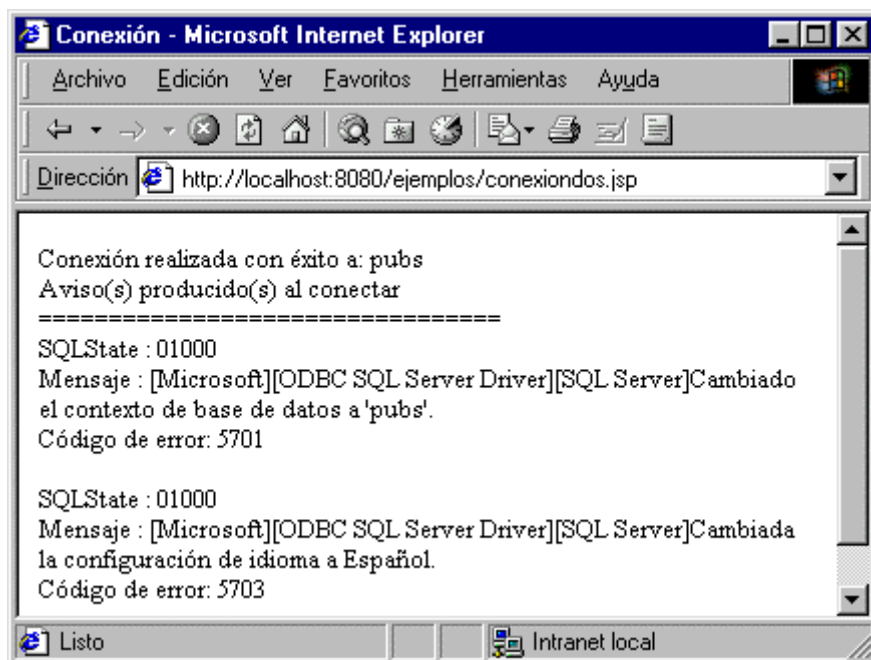


Figura 107. Tratamiento de los avisos

## Creación y ejecución de sentencias sencillas

Un objeto Statement es utilizado para enviar sentencias SQL a una base de datos. Existen tres tipos de objetos que representan una sentencia SQL, cada uno de ellos actúa como un contenedor para ejecutar un tipo determinado de sentencias SQL sobre una conexión a una base de datos, estos tipos son: Statement, PreparedStatement y CallableStatement. PreparedStatement hereda del interfaz Statement y CallableStatement del interfaz PreparedStatement.

La jerarquía de estos tres interfaces del paquete java.sql se puede observar en la Figura 108.

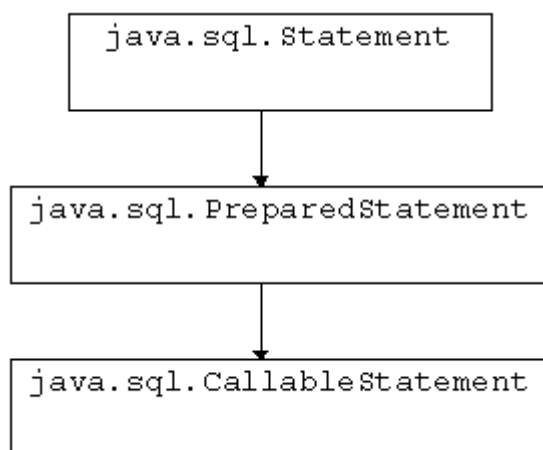


Figura 108. Jerarquía de los interfaces de sentencias

Una vez que se ha establecido una conexión a una base de datos determinada, esta conexión se puede utilizar para enviar sentencias SQL a la base de datos. Un objeto Statement se crea con el método `createStatement()` de la clase `Connection`, como se puede observar en el siguiente fragmento de código:

```
Connection conexion=DriverManager.getConnection(url, "pepe", "xxx");
Statement sentencia=conexion.createStatement();
```

La sentencia SQL que será enviada a la base de datos se indica a través de uno de los métodos para ejecutar objetos Statement, en este caso concreto se utiliza el método `executeQuery()`. En este caso ejecutamos la sentencia SQL y el resultado de la misma lo guardamos en un objeto `ResultSet` para su posterior consulta.

```
ResultSet rs=sentencia.executeQuery("SELECT nombre FROM empleados");
```

Como se puede comprobar, todas las clases e interfaces del API JDBC se encuentran íntimamente relacionadas, a partir de la clase `DriverManager` con el método `getConnection()` se establece una conexión y nos devuelve un objeto `Connection` que representa la conexión establecida. A continuación con el método `createStatement()` del objeto `Connection` creamos una sentencia SQL que se encuentra representada mediante el objeto `Statement`. Y al ejecutar este objeto `Statement`, con el método `executeQuery()` por ejemplo, obtenemos un objeto `ResultSet` que contiene los resultados (registros) de la ejecución del objeto `Statement`.

Veamos una sencilla página JSP que ejecuta una sentencia SQL sobre una base de datos determinada y devuelve una serie de características de la misma (Código Fuente 225).

```
<%@page import="java.sql.*"%>
<html>
<head><title>Conexión</title></head>
<body>
<%String url= "jdbc:odbc:FuenteBD;UID=sa";
//driver de tipo 1
try{
    Driver driver= new sun.jdbc.odbc.JdbcOdbcDriver();
    // Intentamos conectar a la base de datos
    Connection conexion=DriverManager.getConnection(url);
    out.println("Se ha establecido la conexión con: "+url+"<br>");
    Statement sentencia=conexion.createStatement();
```

```

        ResultSet resultado=sentencia.executeQuery("SELECT * FROM AUTHORS");
        out.println(
"Se ha ejecutado la sentencia SQL, que tiene las siguientes características<br>");
        out.println("Timeout de consulta: "+sentencia.getQueryTimeout()+"
sg"<br>");
        out.println("Número máximo de registros: "+sentencia.getMaxRows()+"<br>");
        out.println("Tamaño máximo de campo: "+sentencia.getMaxFieldSize()+"
bytes<br>");
        out.println("Número de registros devueltos cada vez: "+
sentencia.getFetchSize()+"<br>");
        //se liberan los recursos utilizados por la sentencia
        sentencia.close();
        out.println("Se ha cerrado la sentencia<br>");
        conexion.close();
        out.println("Se ha cerrado la conexión con: "+url);
    }catch(SQLException ex){
        out.println("Se han dado excepciones SQLException<br>");
        out.println("=====<br>");
        //Pueden existir varias SQLException encadenadas
        while(ex!=null){
            out.println("SQLState :"+ex.getSQLState()+"<br>");
            out.println("Mensaje :"+ex.getMessage()+"<br>");
            out.println("Código de error :"+ex.getErrorCode()+"<br>");
            ex=ex.getNextException();
            out.println("<br>");
        }
    }%>
</body>
</html>

```

Código Fuente 225

El resultado de la ejecución de la página anterior es el de la Figura 109.

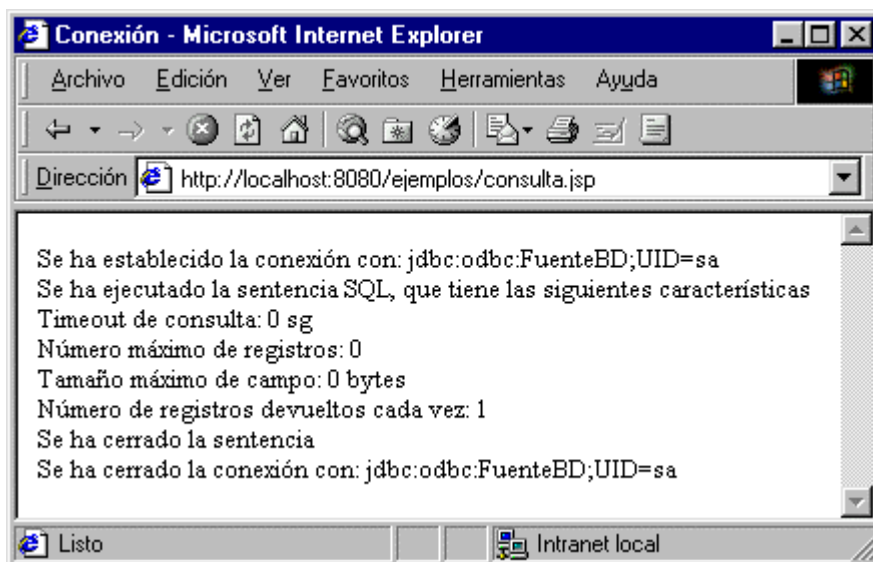


Figura 109. Características de una consulta

En método `createStatement()` del interfaz `Connection`, se encuentra sobrecargado, si utilizamos su versión sin parámetros, a la hora de ejecutar sentencias SQL sobre el objeto `Statement` que se ha creado, se obtendrá el tipo de objeto `ResultSet` por defecto, es decir, se obtendría un tipo de cursor de

sólo lectura y con movimiento únicamente hacia adelante. Pero la otra versión que ofrece el interfaz Connection del método createStatement() ofrece dos parámetros que nos permiten definir el tipo de objeto ResultSet que se va a devolver como resultado de la ejecución de una sentencia SQL.

En el siguiente ejemplo se utiliza la versión del método createStatement() con parámetros.

```
Connection conexion=DriverManager.getConnection(url,"pepe","xxx");
Statement sentencia=conexion.createStatement
    (ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
ResultSet rs=sentencia.executeQuery("SELECT * FROM Usuarios");
```

El primero de los parámetros indica el tipo de objeto ResultSet que se va a crear, y el segundo de ellos indica si el ResultSet es sólo de escritura o si permite modificaciones, este parámetro también se denomina tipo de concurrencia.

Si especificamos el tipo de objeto Resultset es obligatorio indicar si va a ser de sólo lectura o no. Si no indicamos ningún parámetro en el método createStatement(), se creará un objeto ResultSet con los valores por defecto.

Los tipos de ResultSet distintos que se pueden crear dependen del valor del primer parámetro, estos valores se corresponden con constantes definidas en el interfaz ResultSet. Estas constantes se describen a continuación.:

- TYPE\_FORWARD\_ONLY: se crear un objeto ResultSet con movimiento únicamente hacia delante (forward-only). Es el tipo de ResultSet por defecto.
- TYPE\_SCROLL\_INSENSITIVE: se crea un objeto ResultSet que permite todo tipo de movimientos. Pero este tipo de ResultSet, mientras está abierto, no será consciente de los cambios que se realicen sobre los datos que está mostrando, y por lo tanto no mostrará estas modificaciones.
- TYPE\_SCROLL\_SENSITIVE: al igual que el anterior permite todo tipo de movimientos, y además permite ver los cambios que se realizan sobre los datos que contiene.

Los valores que puede tener el segundo parámetro que define la creación de un objeto ResultSet, son también constantes definidas en el interfaz ResultSet y son las siguientes:

- CONCUR\_READ\_ONLY: indica que el ResultSet es sólo de lectura. Es el valor por defecto.
- CONCUR\_UPDATABLE: permite realizar modificaciones sobre los datos que contiene el ResultSet.

Así si queremos obtener un objeto ResultSet que permita desplazamiento libre, pero que no vea los cambios en los datos y que permita realizar modificaciones sobre los datos escribiremos lo siguiente:

```
Connection conexion=DriverManager.getConnection(url,"pepe","xxx");
Statement sentencia=conexion.createStatement
    (ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
ResultSet rs=sentencia.executeQuery("SELECT * FROM Usuarios");
```

La siguiente sentencia sería equivalente a utilizar el método createStatement() sin parámetros.

```
Statement sentencia=conexion.createStatement
    (ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
```

El interfaz `Statement` ofrece tres métodos diferentes para ejecutar sentencias SQL: `executeQuery()`, `executeUpdate()` y `execute()`. El uso de cada uno de ellos viene determinado por el resultado que ofrezca la sentencia SQL y por el contenido de la misma.

El método `executeQuery()` es apropiado para ejecutar sentencias que devuelven un único conjunto de resultados, tales como sentencias `SELECT`, este método devuelve un objeto `ResultSet` (este interfaz lo veremos en detalle en el siguiente capítulo).

El método `executeUpdate()` es utilizado para ejecutar sentencias `INSERT`, `UPDATE` o `DELETE` y también para sentencias DDL (Data Definition Language, lenguaje de definición de datos) de SQL, tales como `CREATE TABLE` y `DROP TABLE`.

El valor que devuelve `executeUpdate()` es un entero que indica el número de filas que han sido afectadas (este número se identifica con `update count` o cuenta de actualizaciones). Para sentencias tales como `CREATE TABLE` o `DROP TABLE`, que no operan con filas, el valor que devuelve `executeUpdate()` es siempre cero.

El método `execute()` es utilizado para ejecutar sentencias que devuelven más de un conjunto de resultados, más de un `update count` (cuenta de actualizaciones), o una combinación de los dos.

También puede ser utilizado para ejecutar dinámicamente sentencias SQL cuyo contenido desconocemos en tiempo de compilación. El método `execute()` devolverá un valor booleano, devolverá `true` si el resultado de la ejecución de la sentencia es un objeto `ResultSet` y `false` si es un entero (`int`) de Java.

Si `execute()` devuelve `false`, esto quiere decir que el resultado es un `update count` o que la sentencia ejecutada era un comando DDL (Data Definition Language).

Así si deseamos enviar una sentencia SQL que consiste en una simple `SELECT` utilizaríamos el método `executeQuery()` pasándole como parámetro un objeto `String` que contendría la consulta correspondiente. Este método nos devolvería un objeto `ResultSet` que contendría el resultado de la ejecución de la sentencia, el código para ejecutar esta consulta sería:

```
ResultSet rs=sentencia.executeQuery("SELECT nombre FROM empleados");
```

Y si deseamos enviar una sentencia SQL que consiste en un `INSERT`, utilizamos el método `executeUpdate()`, al que también se le pasa como parámetro una cadena que representa la sentencia SQL que se va a ejecutar.

```
int filasAfectadas=sentencia.executeUpdate
("INSERT INTO empleados (nombre, apellidos, id)"+
"VALUES ('Angel', 'Esteban', 3)");
```

Como se puede observar un objeto `Statement` no contiene una verdadera sentencia en sí mismo, la sentencia se le debe asignar como un argumento en los diferentes métodos `executeXXX()`, por lo tanto sobre una misma instancia de `Statement` podemos ejecutar distintas sentencias SQL tantas veces como sea necesario.

El siguiente esquema (Figura 110) resume la utilización de los tres métodos de ejecución de sentencias SQL, ofrecidos por el interfaz `Statement`.

Los objetos `Statement` los cerrará automáticamente el recolector de basura (garbage collector) de Java. Sin embargo, es recomendable cerrar explícitamente los objetos `Statement` una vez que ya no se vayan a necesitar más. Esto libera inmediatamente los recursos del DBMS y ayuda a evitar problemas

potenciales de memoria. Para cerrar una sentencia se deberá utilizar el método `close()` del interfaz `Statement`.

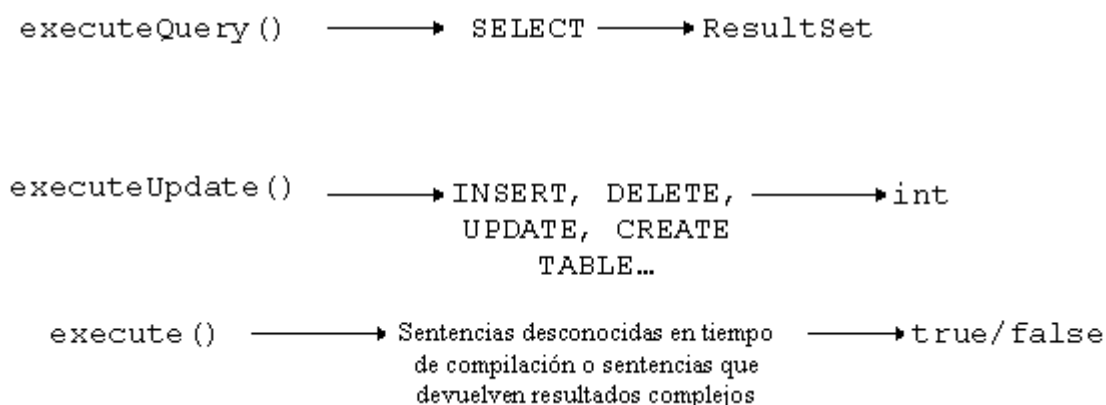


Figura 110. Ejecución de sentencias SQL

## Tratando los datos, el interfaz `ResultSet`

En un objeto `ResultSet` se encuentran los resultados de la ejecución de una sentencia SQL, por lo tanto, un objeto `ResultSet` contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece el acceso a los datos de las filas a través de una serie de métodos `getXXX` que permiten acceder a las columnas de la fila actual.

El aspecto que suele tener un `ResultSet` es una tabla con cabeceras de columnas y los valores correspondientes devueltos por una consulta.

Un `ResultSet` mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve hacia abajo cada vez que el método `next()` es lanzado. Inicialmente está posicionado antes de la primera fila, de esta forma, la primera llamada a `next()` situará el cursor en la primera fila, pasando a ser la fila actual. Las filas del `ResultSet` son devueltas de arriba a abajo según se va desplazando el cursor con las sucesivas llamadas al método `next()`. Un cursor es válido hasta que el objeto `ResultSet` o su objeto padre `Statement` es cerrado.

Los métodos `getXXX()` de este interfaz ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del `ResultSet`. Dentro de cada columna no es necesario que las columnas sean recuperadas utilizando un orden determinado, pero para una mayor portabilidad entre diferentes bases de datos se recomienda que los valores de las columnas se recuperen de izquierda a derecha y solamente una vez.

Para designar una columna podemos utilizar su nombre o bien su número de orden. Por ejemplo si la segunda columna de un objeto `rs` de la clase `ResultSet` se llama "título" y almacena datos de tipo `String`, se podrá recuperar su valor de las siguientes formas:

```
String valor=rs.getString(2);
String valor=rs.getString("título");
```

Se debe señalar que las columnas se numeran de izquierda a derecha empezando con la columna 1, y que los nombres de las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.



Cuando se lanza un método `getXXX()` determinado sobre un objeto `ResultSet` para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método `getString()` y el tipo del dato en la base de datos es `VARCHAR`, el driver JDBC convertirá el dato `VARCHAR` a un objeto `String` de Java, por lo tanto el valor de retorno de `getString()` será un objeto de la clase `String`.

Esta conversión de tipos se puede realizar gracias a la clase `java.sql.Types`. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL estándar.

La conversión que se realiza entre tipos JDBC y clases Java la podemos observar en la Tabla 20 que aparece a continuación.

Tipos JDBC	Tipos Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time

TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
DISTINCT	Mapeo de un tipo específico
STRUCT	Struct
REF	Ref
JAVA_OBJECT	Una clase de Java

Tabla 20. Tipos de datos JDBC y tipos de Java

Como ya se había mencionado anteriormente, el método `next()` del interfaz `ResultSet` lo vamos a utilizar para desplazarnos al registro siguiente dentro de un `ResultSet`. El método `next()` devuelve un valor booleano (tipo `boolean` de Java), `true` si el registro siguiente existe y `false` si hemos llegado al final del objeto `ResultSet`, es decir, no hay más registros. Este era el único método que ofrecía el interfaz `ResultSet` en la versión 1.0 de JDBC, pero en JDBC 2.0, además de tener el método `next()`, disponemos de los siguientes métodos para el desplazamiento y movimiento dentro de un objeto `ResultSet`:

- `boolean absolute(int registro)`: desplaza el cursor al número de registros indicado. Si el valor es negativo, se posiciona en el número registro indicado pero empezando por el final. Este método devolverá `false` si nos hemos desplazado después del último registro o antes del primer registro del objeto `ResultSet`. Para poder utilizar este método el objeto `ResultSet` debe ser de tipo `TYPE_SCROLL_SENSITIVE` o de tipo `TYPE_SCROLL_INSENSITIVE`, a un `ResultSet` que es de cualquiera de estos dos tipos se dice que es de tipo `scrollable`. Si a este método le pasamos un valor cero se lanzará una excepción `SQLException`
- `void afterLast()`: se desplaza al final del objeto `ResultSet`, después del último registro. Si el `ResultSet` no posee registros este método no tienen ningún efecto. Este método sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`, como sucede con muchos de los métodos comentados.
- `void beforeFirst()`: mueve el cursor al comienzo del objeto `ResultSet`, antes del primer registro. Sólo se puede utilizar sobre objetos `ResultSet` de tipo `scrollable`.
- `boolean first()`: desplaza el cursos al primer registro. Devuelve verdadero si el cursor se ha desplazado a un registro válido, por el contrario, devolverá `false` en otro caso o bien si el objeto `ResultSet` no contiene registros. Al igual que los métodos anteriores, sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`.
- `void last()`: desplaza el cursor al último registro del objeto `ResultSet`. Devolverá `true` si el cursor se encuentra en un registro válido, y `false` en otro caso o si el objeto `ResultSet` no tiene registros. Sólo es valido para objetos `ResultSet` de tipo `scrollable`, en caso contrario lanzará una excepción `SQLException`.

- `void moveToCurrentRow()`: mueve el cursor a la posición recordada, normalmente el registro actual. Este método sólo tiene sentido cuando estamos situados dentro del `ResultSet` en un registro que se ha insertado. Este método sólo es válido utilizarlo con objetos `ResultSet` que permiten la modificación, es decir, están definidos mediante la constante `CONCUR_UPDATABLE`.
- `boolean previous()`: desplaza el cursor al registro anterior. Es el método contrario al método `next()`. Devolverá `true` si el cursor se encuentra en un registro o fila válidos, y `false` en caso contrario. Sólo es válido este método con objetos `ResultSet` de tipo `scrollable`, en caso contrario lanzará una excepción `SQLException`.
- `boolean relative(int registros)`: mueve el cursor un número relativo de registros, este número puede ser positivo o negativo. Si el número es negativo el cursor se desplazará hacia el principio del objeto `ResultSet` el número de registros indicados, y si es positivo se desplazará hacia el final de objeto `ResultSet` correspondiente. Este método sólo se puede utilizar si el `ResultSet` es de tipo `scrollable`.

También existen otros métodos dentro del interfaz `ResultSet` que están relacionados con el desplazamiento:

- `boolean isAfterLast()`: indica si nos encontramos después del último registro del objeto `ResultSet`. Sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`.
- `boolean isBeforeFirst()`: indica si nos encontramos antes del primer registro del objeto `ResultSet`. Sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`.
- `boolean isFirst()`: indica si el cursor se encuentra en el primer registro. Sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`.
- `boolean isLast()`: indica si nos encontramos en el último registro del `ResultSet`. Sólo se puede utilizar en objetos `ResultSet` de tipo `scrollable`.
- `int getRow()`: devuelve el número de registro actual. El primer registro será el número 1, el segundo el 2, etc. Devolverá cero si no hay registro actual.

A continuación vamos a realizar una página JSP que se conecte a una base de datos, ejecute una sentencia SQL y muestre los resultados de la ejecución de la misma, en este ejemplo se devuelve el contenido de la tabla en sentido inverso, es decir, nos situamos en el último registro y empezamos a mostrar el contenido del `ResultSet` utilizando el método `movePrevious()`. La sentencia que se va a ejecutar va a ser una `SELECT` sobre una tabla llamada `authors` de una base de datos llamada `pubs`, que está en un servidor SQL Server cuyo nombre es `MiSer`. El usuario se llama `pepe` y su contraseña es `xxx`. El Código Fuente 226 muestra el código completo de esta página JSP.

```
<%@page import="java.sql.*"%>
<html>
<head><title>ResultSet</title></head>
<body>
<%try{
    //Se registra el driver
    Class.forName("com.inet.tds.TdsDriver");
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection(
        "jdbc:inetdae:MiSer:1433?sql7=true&database=pubs&user=pepe&password=xxx");
    //Creación de la consulta
```

```

Statement consulta=conexion.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
//Ejecución de la consulta
ResultSet rs=consulta.executeQuery("SELECT * FROM AUTHORS");
rs.afterLast();
boolean seguir=rs.previous();%>
<table align="center" border="1">
<%while(seguir){
    //Muestra el contenido de un registro del ResultSet%>
    <tr><td><%=rs.getString(1)+" "+rs.getString(2)+" " +
    rs.getString(3)+" "+rs.getString(4)%></td></tr>
    <%seguir=rs.previous();
}%>
</table>
<%//Se cierra el ResultSet, aunque no es necesario
rs.close();
//Se cierra la consulta
consulta.close();
//Se cierra la conexión con la BD
conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca%>
    <%= "Se produjo una excepción durante la consulta SQL: "+ex%>
    <%}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
    <%= "Se produjo una excepción: "+ex%>
<%}%>
</body>
</html>

```

Código Fuente 226

Un ejemplo de ejecución de esta página lo tenemos en la Figura 111.

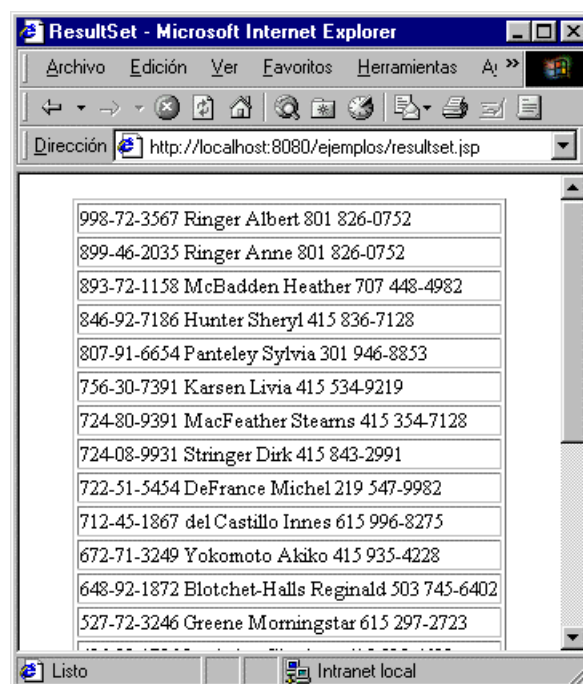


Figura 111. El contenido de un ResultSet

Para poder modificar los datos que contiene un `ResultSet` debemos crear un `ResultSet` de tipo modificable, para ello debemos utilizar la constante `ResultSet.CONCUR_UPDATABLE` dentro del método `createStatement()`.

Aunque un `ResultSet` que permite modificaciones suele permitir distintos desplazamientos, es decir, se suele utilizar la constante `ResultSet.TYPE_SCROLL_INSENSITIVE` o `ResultSet.TYPE_SCROLL_SENSITIVE`, pero no es del todo necesario ya que también puede ser del tipo sólo hacia adelante (forward-only).

Para modificar los valores de un registro existente se utilizan una serie de métodos `updateXXX()` del interfaz `ResultSet`. Las XXX indican el tipo del dato al igual que ocurre con los métodos `getXXX()` de este mismo interfaz.

El proceso para realizar la modificación de una fila de un `ResultSet` es el siguiente: nos situamos sobre el registro que queremos modificar y lanzamos los métodos `updateXXX()` adecuados, pasándole como argumento los nuevos valores. A continuación lanzamos el método `updateRow()` para que los cambios tengan efecto sobre la base de datos.

El método `updateXXX` recibe dos parámetros, el campo o columna a modificar y el nuevo valor. La columna la podemos indicar por su número de orden o bien por su nombre, igual que en los métodos `getXXX`.

Para modificar el campo Provincia del último registro de un `ResultSet` que contiene el resultado de una `SELECT` sobre la tabla de Provincias, escribiremos:

```
Statement sentencia=conexion.createStatement (
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs=sentencia.executeQuery("SELECT * FROM Provincias");
rs.last();
rs.updateString("Provincia","Soria");
//también podemos utilizar el número de columna:
rs.updateString(2,"Soria");
rs.updateRow();
```

Si nos desplazamos dentro del `ResultSet` antes de lanzar el método `updateRow()`, se perderán las modificaciones realizadas. Y si queremos cancelar las modificaciones lanzaremos el método `cancelRowUpdates()` sobre el objeto `ResultSet`, en lugar del método `updateRow()`.

Una vez que hemos invocado el método `updateRow()`, el método `cancelRowUpdates()` no tendrá ningún efecto. El método `cancelRowUpdates()` cancela las modificaciones de todos los campos de un registro, es decir, si hemos modificado dos campos con el método `updateXXX` se cancelarán ambas modificaciones.

Además de poder realizar modificaciones directamente sobre las filas de un `ResultSet`, con JDBC 2.0 también podemos añadir nuevas filas (registros) y eliminar las existentes. Para insertar un registro nuevo en JDBC 1.0 no nos quedaba otra opción que hacerlo a través de código SQL. Lo mismo ocurría si queríamos eliminar un registro:

Sin embargo JDBC 2.0 nos ofrece dos nuevos métodos dentro del interfaz `ResultSet` para realizar estas dos operaciones a directamente a través del lenguaje Java, estos nuevos métodos son: `moveToInsertRow()` y `deleteRow()`.

El primer paso para insertar un registro o fila en un `ResultSet` es mover el cursor (puntero que indica el registro actual) del `ResultSet`, esto se consigue mediante el método `moveToInsertRow()`.

El siguiente paso es dar un valor a cada uno de los campos que van a formar parte del nuevo registro, para ello se utilizan los métodos updateXXX adecuados. Para finalizar el proceso se lanza el método insertRow(), que creará el nuevo registro tanto en el ResultSet como en la tabla de la base de datos correspondientes. Hasta que no se lanza el método insertRow(), la fila no se incluye dentro del ResultSet, es una fila especial denominada fila de inserción (insert row) y es similar a un búfer completamente independiente del objeto ResultSet.

Si queremos dar de alta un registro en la tabla de Provincias, podemos escribir el siguiente código:

```
Statement sentencia=conexion.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=sentencia.executeQuery("SELECT * FROM Provincias");  
rs.moveToInsertRow();  
rs.updateInt("idProvin", 30);  
rs.updateString("Provincia", "Burgos");  
rs.insertRow();
```

En este caso el tipo de ResultSet utilizado es sensible a los cambios que se producen sobre los datos que contiene.

Si no facilitamos valores a todos los campos del nuevo registro con los métodos updateXXX, ese campo tendrá un valor NULL, y si en la base de datos no está definido ese campo para admitir nulos se producirá una excepción SQLException.

Cuando hemos insertado nuestro nuevo registro en el objeto ResultSet, podremos volver a la antigua posición en la que nos encontrábamos dentro del ResultSet, antes de haber lanzado el método moveToInsertRow(), llamando al método moveToCurrentRow(), este método sólo se puede utilizar en combinación con el método moveToInsertRow().

Además de insertar filas en nuestro objeto ResultSet también podremos eliminar filas o registros del mismo. El método que se debe utilizar para esta tarea es el método deleteRow(). Para eliminar un registro no tenemos que hacer nada más que movernos a ese registro, y lanzar el método deleteRow() sobre el objeto ResultSet correspondiente. Así por ejemplo, si queremos borrar el último registro de la tabla de Provincias este código nos podría ser útil:

```
Statement sentencia=conexion.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs=sentencia.executeQuery("SELECT * FROM Provincias");  
rs.last();  
rs.deleteRow();
```

Y para borrar el tercer registro:

```
rs.absolute(3);  
rs.deleteRow();
```

Con este apartado damos por finalizado el capítulo actual. En el siguiente capítulo, que es una continuación del actual, seguiremos tratando el API JDBC desde las páginas JSP, veremos dos tipos distintos de sentencias más que podemos utilizar en JDBC, y también la utilización de componentes JavaBeans que hacen uso del API JDBC.

# Acceso a datos desde JSP II

---

## Introducción

Como se propio título indica, este esta es la segunda parte del capítulo anterior, en el que vimos lo que era el API JDBC, como realizar una conexión, como ejecutar sentencias y como mostrar y manipular los resultados. En este nuevo capítulo vamos a tratar un par de sentencias más que nos permiten utilizar parámetros y procedimientos almacenados, además veremos como integrar los componentes JavaBeans con el acceso a datos a través de JDBC desde una página JSP.

## El interfaz PreparedStatement

Este interfaz, al igual que el interfaz Statement, nos permite ejecutar sentencias SQL sobre una conexión establecida con una base de datos. Pero en este caso vamos a ejecutar sentencias SQL más especializadas, estas sentencias SQL se van a denominar sentencias SQL precompiladas y van a recibir parámetros de entrada.

El interfaz PreparedStatement hereda del interfaz Statement y se diferencia de él de dos maneras:

- Las instancias de PreparedStatement contienen sentencias SQL que ya han sido compiladas. Esto es lo que hace a una sentencia "prepared" (preparada).
- La sentencia SQL que contiene un objeto PreparedStatement puede contener uno o más parámetros de entrada. Un parámetro de entrada es aquél cuyo valor no se especifica cuando la sentencia es creada, en su lugar la sentencia va a tener un signo de interrogación (?) por cada parámetro de entrada. Antes de ejecutarse la sentencia se debe especificar un valor para cada

uno de los parámetros a través de los métodos setXXX apropiados. Estos métodos setXXX los añade el interfaz PreparedStatement.

Debido a que las sentencias de los objetos PreparedStatement están precompiladas su ejecución será más rápida que la de los objetos Statement. Por lo tanto, una sentencia SQL que va a ser ejecutada varias veces se suele crear como un objeto PreparedStatement para ganar en eficiencia. También se utilizará este tipo de sentencias para pasarle parámetros de entrada a las sentencias SQL.

Al heredar del interfaz Statement, el interfaz PreparedStatement hereda todas funcionalidades de Statement. Además, añade una serie de métodos que permiten asignar un valor a cada uno de los parámetros de entrada de este tipo de sentencias.

Los métodos execute(), executeQuery() y executeUpdate() son sobrecargados y en esta versión, es decir, para los objetos PreparedStatement no toman ningún tipo de argumentos, de esta forma, a estos métodos nunca se les deberá pasar por parámetro el objeto String que representaba la sentencia SQL a ejecutar. En este caso, un objeto PreparedStatement ya es una sentencia SQL por sí misma, a diferencia de lo que ocurría con las sentencias Statement que poseían un significado sólo en el momento en el que se ejecutaban.

Para crear un objeto PreparedStatement se debe lanzar el método preparedStatement() del interfaz Connection sobre el objeto que representa la conexión establecida con la base de datos. En el siguiente ejemplo (Código fuente 227) se puede ver como se crearía un objeto PreparedStatement que representa una sentencia SQL con dos parámetros de entrada.

```
Connection conexion=DriverManager.getConnection(url,"pepe","xxxx");
PreparedStatement sentencia=conexion.prepareStatement("UPDATE MiTabla SET nombre =
?"+"WHERE clave =?");
```

Código fuente 227

El objeto sentencia contendrá la sentencia SQL indicada, la cual, ya ha sido enviada al DBMS y ya ha sido preparada para su ejecución. En este caso se ha creado una sentencia SQL con dos parámetros de entrada.

Antes de poder ejecutar un objeto PreparedStatement se debe asignar un valor para cada uno de sus parámetros. Esto se realiza mediante la llamada a un método setXXX, donde XXX es el tipo apropiado para el parámetro. Por ejemplo, si el parámetro es de tipo long, el método a utilizar será setLong().

El primer argumento de los métodos setXXX es la posición ordinal del parámetro al que se le va a asignar valor, y el segundo argumento es el valor a asignar. Por ejemplo el Código fuente 228 crea un objeto PreparedStatement y acto seguido le asigna al primero de sus parámetros un valor de tipo String y al segundo un valor de tipo int.

```
Connection conexion=DriverManager.getConnection(url);
PreparedStatement sentencia=conexion.prepareStatement("UPDATE MiTabla SET nombre =
? "+"WHERE clave =?");
sentencia.setString(1,"Pepe");
sentencia.setInt(2,157);
```

Código fuente 228



Una vez que se ha asignado unos valores a los parámetros de entrada de una sentencia, el objeto `PreparedStatement` se puede ejecutar múltiples veces, hasta que sean borrados los parámetros con el método `clearParameters()`, aunque no es necesario llamar a este método cuando se quieran modificar los parámetros de entrada, sino que al lanzar los nuevos métodos `setXXX` los valores de los parámetros serán reemplazados.

El Código fuente 229 muestra como una vez creado un objeto `PreparedStatement` se ejecuta varias veces y se le cambian los parámetros.

```
Connection conexion=DriverManager.getConnection(url);
PreparedStatement sentencia=conexion.prepareStatement("UPDATE MiTabla SET Id = ?"+
"WHERE num=?");

for (int i=0;i<20;i++){
    sentencia.setInt(1,i);
    sentencia.setInt(2,i);
    sentencia.executeUpdate();
}
```

Código fuente 229

Como ya se había comentado anteriormente XXX en un método `setXXX` es un tipo de Java. Es implícitamente un tipo JDBC (un tipo SQL genérico) ya que el driver transformará el tipo Java a su tipo JDBC correspondiente y acto seguido lo enviará a la base de datos. Así por ejemplo, el siguiente fragmento (Código Fuente 230) asigna a un parámetro de entrada el valor 44 del tipo Java `short`.

```
sentencia.setShort(2,44);
```

Código Fuente 230

El driver enviará 44 a la base de datos como un `SMALLINT` de JDBC, que es la transformación estándar para un `short` de Java. Se realiza la transformación inversa que realizaban los métodos `getXXX`.

Como ejemplo de utilización de una sentencia `PreparedStatement` vamos a realizar una página JSP que muestra un formulario y que va a permitir realizar el alta de un registro sobre una tabla de una base de datos determinada. La tabla se va a llamar `Provincias` y va a tener dos campos: `IDprovin` de tipo entero y `Provincia` de tipo cadena de caracteres. En el Código Fuente 231 se muestra el código completo de esta página JSP.

```
<%@page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>PreparedStatement</title>
</head>
<body>
<form action="prepared.jsp" method="GET">
    <input type="text" name="id" value="" size="3">
    <input type="text" name="provin" value="" size="25">
    <input type="submit" name="enviar" value="Realizar alta">
</form>
<%if (request.getParameter("enviar")!=null){
```

```

try{
    String sSentencia="INSERT INTO Provincias "+
                        "(idprovin,provincia) VALUES(?,?)";
    //Se registra el driver
    Class.forName("com.inet.tds.TdsDriver");
    //Driver driver=new com.inet.tds.TdsDriver();
    //Realización de la conexión
    Connection conexion=DriverManager.getConnection
        ("jdbc:inetdae:,miServ:1433?sql7=true&database=provincias&user=sa");
    //Se envía a la BD la consulta para que la compile
    PreparedStatement sentencia=conexion.prepareStatement(sSentencia);
    //Se pasan los dos parámetros de entrada
    sentencia.setInt(1,Integer.parseInt(request.getParameter("id")));
    sentencia.setString(2,request.getParameter("provin"));
    //Se realiza el alta
    sentencia.executeUpdate();
    out.print("<b><i>Alta realizada correctamente.</i></b>");
    sentencia.close();
    conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca%>
    <%= "Se produjo una excepción durante la conexión: "+ex%>
    <%>
}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
    <%= "Se produjo una excepción: "+ex%>
    <%>
}
}
</body>
</html>

```

Código Fuente 231

En la Figura 112 se puede ver un ejemplo de ejecución de esta página.

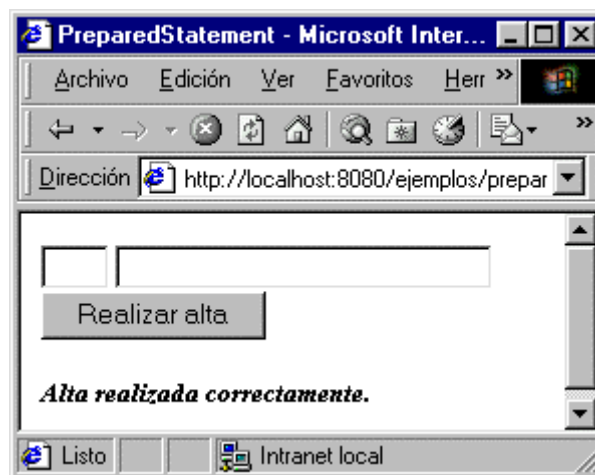


Figura 112. Realizando un alta

En el siguiente apartado vamos a tratar el tercer y último tipo de sentencias que podemos utilizar en el API JDBC.

## El interfaz CallableStatement

Este interfaz hereda del interfaz PreparedStatement y ofrece la posibilidad de manejar parámetros de salida y de realizar llamadas a procedimientos almacenados de la base de datos.

Un objeto CallableStatement ofrece la posibilidad de realizar llamadas a procedimientos almacenados de una forma estándar para todos los DBMS. Un procedimiento almacenado se encuentra dentro de una base de datos; la llamada a un procedimiento es lo que contiene un objeto CallableStatement. Esta llamada está escrita con un sintaxis de escape.

La sintaxis para realizar la llamada a un procedimiento almacenado es la siguiente:

```
{call nombre_del_procedimiento[(?, ?, ...)]}
```

El interfaz CallableStatement hereda los métodos del interfaz Statement, que se encargan de sentencias SQL generales, y también los métodos del interfaz PreparedStatement, que se encargan de manejar parámetros de entrada. En el esquema de la Figura 113 se puede ver la relación de herencia que existe entre los tres tipos de sentencias.

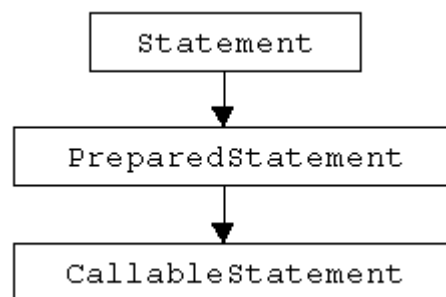


Figura 113. Jerarquía de los interfaces para sentencias

Los métodos que define el interfaz CallableStatement se encargan de manejar parámetros de salida: registrando los tipos JDBC de los parámetros de salida, recuperando los valores o comprobando si un valor devuelto es un JDBC NULL. Este interfaz supone un paso más en la especialización de sentencias. Una sentencia CallableStatement puede tener parámetros de entrada, de salida y de entrada/salida.

Para pasarle parámetros de entrada a un objeto CallableStatement, se utilizan los métodos setXXX que heredaba del interfaz PreparedStatement.

Si el procedimiento almacenado devuelve parámetros de salida, el tipo JDBC de cada parámetro de salida debe ser registrado antes de ejecutar el objeto CallableStatement correspondiente. Para registrar los tipos JDBC de los parámetros de salida se debe lanzar el método registerOutParameter() del interfaz CallableStatement.

Después de ejecutar la sentencia, se pueden recuperar los valores de estos parámetros llamando al método getXXX adecuado. El método getXXX debe recuperar el tipo Java que se correspondería con el tipo JDBC con el que se registró el parámetro. A los métodos getXXX se le pasará un entero que indicará el valor ordinal del parámetro a recuperar.

En el Código Fuente 232 se registran dos parámetros de salida, ejecuta el procedimiento llamado por el objeto sentencia y luego recupera el valor de los parámetros de salida.

```

Connection conexion=DriverManager.getConnection(url);
CallableStatement sentencia=conexion.prepareCall("{call compruebaDatos(?,?)}");
sentencia.registerOutParameter(1,java.sql.Types.TINYINT);
sentencia.registerOutParameter(2,java.sql.Types.DECIMAL);
sentencia.execute();
byte x=sentencia.getBytes(1);
java.math.BigDecimal n=sentencia.getBigDecimal(2,3);

```

Código Fuente 232

El Código Fuente 233 supone que existe un procedimiento almacenado llamado `revisarTotales` que posee un parámetro de entrada/salida.

```

CallableStatement sentencia=conexion.prepareCall("{call revisarTotales(?)}");
sentencia.setByte(1,25);
sentencia.registerOutParameter(1,java.sql.Types.TINYINT);
sentencia.executeUpdate();
byte x=sentencia.getBytes(1);

```

Código Fuente 233

Utilizando una sentencia `CallableStatement` vamos a describir el ejemplo del capítulo anterior que implementaba una página JSP que realizaba un alta sobre la tabla de Provincias.

Vamos a utilizar el sistema gestor de base de datos MS SQL Server, ya que soporta procedimientos almacenados. En nuestro caso vamos a utilizar un procedimiento almacenado llamado `AltaProvincia`. Este procedimiento recibe dos parámetros de entrada, que se van a utilizar para dar de alta un nuevo registro en la tabla de Provincias. El primero de estos parámetros es de tipo entero y el segundo de tipo cadena de caracteres. El código SQL que crea este procedimiento, lo podemos observar a continuación (Código Fuente 234).

```

CREATE PROCEDURE AltaProvincia
    @id int,
    @descrip varchar(40)
AS
    INSERT INTO Provincias VALUES(@id,@descrip)

```

Código Fuente 234

Las operaciones que realizamos con sentencias `PreparedStatement` las podemos realizar también con sentencias `CallableStatement` a partir del procedimiento almacenado adecuado. Así por ejemplo la operación de alta la podemos realizar con una sentencia `CallableStatement` que llama al procedimiento almacenado `AltaProvincia`.

El Código Fuente 235 sería un fragmento de como quedaría la página JSP al utilizar un objeto `CallableStatement`, sólo se ha mostrado lo que cambia de la página. El resultado de la ejecución sería exactamente igual al del ejemplo anterior.

```

String sSentencia="{call AltaProvincia(?,?)}";
//Se registra el driver

```

```

Class.forName("com.inet.tds.TdsDriver");
//Realización de la conexión
Connection conexion=DriverManager.getConnection
    ("jdbc:inetdae:miServ:1433?sql7=true&database=provincias&user=sa");
//Se envía a la BD la consulta para que la compile
CallableStatement sentencia=conexion.prepareCall(sSentencia);

```

Código Fuente 235

La siguiente página JSP de ejemplo va a utilizar parámetros de salida y se va a encargar de devolver el domicilio de una empresa, cuyo código le pasamos por a través de un formulario en la misma página.

En este ejemplo vamos a utilizar una nueva tabla, la tabla de Empresas. Esta tabla está en la base de datos BaseDatos. La tabla de Empresas tiene tres campos: Código de tipo entero, Empresa de tipo cadena de caracteres (varchar), y Domicilio, también de tipo varchar.

En la base de datos tendremos definido un procedimiento almacenado llamado devuelveDomicilio. Este procedimiento devolverá el domicilio de la empresa que se corresponda con el código de empresa que se pasa por parámetro. El procedimiento devuelveDomicilio posee dos parámetros; un parámetro de entrada, que será el código de empresa y un parámetro de salida que será el domicilio de la empresa correspondiente. El código SQL que define este procedimiento lo vemos aquí (Código Fuente 236):

```

CREATE PROCEDURE devuelveDomicilio
    @cod int,
    @domicilio varchar(50) output
AS
    SELECT @domicilio=domicilio FROM Empresas WHERE codigo=@cod

```

Código Fuente 236

A continuación ofrecemos el código (Código Fuente 237) de este nuevo ejemplo.

```

<%@page import="java.sql.*"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>CallableStatement</title>
</head>
<body>
<form action="callableDos.jsp" method="GET">
    Código de empresa:
    <input type="text" name="codigo" value="" size="3">
    <input type="submit" name="enviar" value="Obtener domicilio">
</form>
<%if (request.getParameter("enviar")!=null){
    try{
        String sSentencia="{call devuelveDomicilio(?,?)}";
        //Se registra el driver
        Class.forName("com.inet.tds.TdsDriver");
        //Driver driver=new com.inet.tds.TdsDriver();
        //Realización de la conexión
        Connection conexion=DriverManager.getConnection
            ("jdbc:inetdae:MiServ?sql7=true&database=BaseDatos&user=sa");
        //Se envía a la BD la consulta para que la compile
        CallableStatement sentencia=conexion.prepareCall(sSentencia);
        //Se asigna el valor al parámetro de entrada del procedimiento
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
%>

```

```

        sentencia.setInt(1,Integer.parseInt(request.getParameter("codigo")));
        //Se registra el parámetro de salida del procedimiento
        sentencia.registerOutParameter(2,Types.VARCHAR);
        //Se ejecuta la sentencia
        sentencia.execute();
        out.println("Resultado de la sentencia: ");
        //Se recupera el número de filas afectadas
        if(sentencia.getUpdateCount()==1)
            //Se recupera el valor del parámetro de salida
            out.println("<b><i>"+sentencia.getString(2)+"</i></b>");
        else
            out.println("<b><i>El código de empresa no existe</i></b>");
        //Se cierra la sentencia y la conexión
        sentencia.close();
        conexion.close();
    }
    catch(SQLException ex){
        //Se captura la excepción de tipo SQLException que se produzca%>
        <%= "Se produjo una excepción durante la conexión: "+ex%>
        <%>
    }
    catch(NumberFormatException ex){
        out.println("El parámetro debe ser un entero");
    }
    catch(Exception ex){
        //Se captura cualquier tipo de excepción que se produzca%>
        <%= "Se produjo una excepción: "+ex%>
    }
    <%>
}
</body>
</html>

```

Código Fuente 237

Como se puede observar, debemos pasar primero el parámetro de entrada mediante el método `setInt()`, pero antes debemos realizar la transformación de cadena de caracteres a entero, esta transformación lanzará una excepción `NumberFormatException` si la cadena de caracteres no se corresponde con un entero.

A continuación registramos el parámetro de salida indicando el tipo del mismo, para ello utilizamos la clase `Types`. Ejecutamos la sentencia con el método `execute()`. Para comprobar si se ha encontrado la empresa que se corresponde con el código pasado por parámetro, se lanza el método `getUpdateCount()` que nos devolverá el número de registros afectados por la ejecución de la sentencia. En este caso, si todo ha ido correctamente nos devolverá 1, ya que el código de empresa es la clave de la tabla de Empresas. Para obtener el valor del parámetro de salida lanzamos el método `getString()` sobre nuestro objeto `CallableStatement`. Mostramos el valor devuelto en pantalla y cerramos la sentencia y la conexión. La Figura 114 muestra un ejemplo de ejecución de esta página.

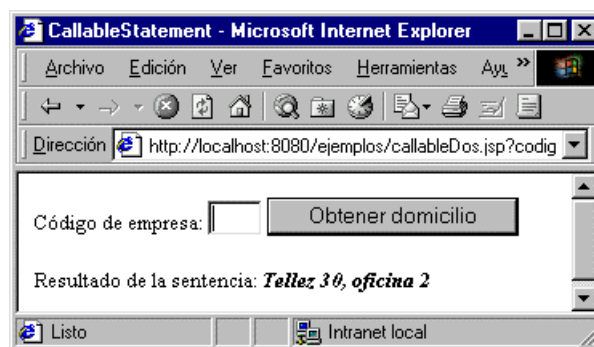


Figura 114. Utilizando parámetros de salida

En los dos apartados siguientes vamos a ver otros dos interfaces del API JDBC, en este caso estos interfaces no va a representar una sentencia SQL, sino que son interfaces que nos ofrecen información acerca de la base de datos a la que nos hemos conectado e información relativa al objeto ResultSet actual.

## El interfaz DatabaseMetaData

Este interfaz ofrece información sobre la base de datos a la que nos hemos conectado, trata a la base de datos como un todo. A partir de este interfaz vamos a obtener una gran cantidad de información sobre la base de datos con la que hemos establecido una conexión.

Para obtener esta información este interfaz aporta un gran número de métodos diferentes. Muchos de estos métodos devuelven objetos ResultSet conteniendo la información correspondiente, por lo tanto, deberemos usar los métodos getXXX para recuperar la información.

Estos métodos los podemos dividir en los que devuelven una información sencilla, tipos de Java, y los que devuelven una información más compleja, como puede ser un objeto ResultSet.

Algunos de las cabeceras de estos métodos toman como parámetro patrones de cadenas, en los que se pueden utilizar caracteres comodín. "%" identifica una cadena de 0 o más caracteres y "\_" se identifica con un sólo carácter, de esta forma, sólo los datos que se correspondan con el patrón dado serán devueltos por el método lanzado.

Si un driver no soporta un método del interfaz DatabaseMetaData se lanzará una excepción SQLException, y en el caso de que el método devuelva un objeto ResultSet, se obtendrá un ResultSet vacío o se lanzará una excepción SQLException.

Para obtener un objeto DatabaseMetaData sobre el que lanzar los métodos que nos darán la información sobre el DBMS se debe lanzar el método getMetaData() sobre el objeto Connection que se corresponda con la conexión a la base de datos de la que queremos obtener la información, como se puede observar en el Código Fuente 238

```
Connection conexion=DriverManager.getConnection(url);  
DatabaseMetaData dbmd=conexion.getMetaData();
```

Código Fuente 238

Todos los métodos del interfaz DatabaseMetaData lanzan excepciones SQLException. No vamos a mostrar y comentar cada uno de los métodos que contiene este interfaz, ya que contiene más de 50 métodos diferentes que nos ofrecen una información detallada sobre la base de datos. Pero lo que si vamos a hacer es ver algunos de estos métodos a través de un ejemplo.

Vamos a realizar una página JSP que se conecte a una base de datos de MS SQL Server a través del driver de tipo 1 JDBC-ODBC. Una vez conectados a la base de datos vamos a obtener algunos datos de interés sobre ella.

Una vez que se ha realizado la conexión, se obtiene un objeto DatabaseMetaData. Para ello lanzamos el método getMetaData() sobre el objeto Connection, con lo que ya dispondremos de un objeto DatabaseMetaData que nos va a ofrecer la información sobre la base de datos. En una primera parte la página JSP va a ofrecer información general sobre el sistema gestor de base de datos: su nombre,

versión, URL de JDBC, usuario conectado, características que soporta, etc, y a continuación se obtiene información sobre el driver que se utiliza para realizar la conexión, la información que se obtiene es: el nombre del driver, versión, versión inferior y superior.

En todos los métodos que se han utilizado del interfaz `DatabaseMetaData` se devuelven valores que se corresponden con objetos `String` o tipos `int` de Java. En la última parte de la página JSP, que nos devuelve información sobre las tablas de la base de datos, la información se devuelve dentro de un objeto `ResultSet` con un formato determinado, esto lo conseguimos a través del método `getTables()` del interfaz `DatabaseMetaData`.

La llamada al método `getTables()` es más complicada que las llamadas al resto de los métodos del interfaz `DatabaseMetaData` vistos hasta ahora. Los dos primeros parámetros del método `getTables()` se utilizan para obtener las tablas de un catálogo, en nuestro caso no vamos a utilizar esta opción y por lo tanto le pasamos un `null`, al igual que vamos a hacer con el segundo parámetro. El tercer parámetro es el patrón de búsqueda que se va a aplicar al nombre de las tablas, si utilizamos el carácter especial `%` obtendremos todas las tablas. El último de los parámetros indica el tipo de tabla que queremos obtener. Este parámetro es un array con todos los tipos de tablas de las que queremos obtener información, los tipos de tabla son: `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS` y `SYNONYM`.

En este ejemplo se quiere recuperar información sobre las tablas de usuario y de sistema, por lo que sólo tendremos un array con dos elementos, el primero de ellos contendrá la cadena `TABLE` y el segundo la cadena `SYSTEM TABLE`.

En el objeto `ResultSet` devuelto por el método `getTables()` encontraremos en cada fila del mismo información sobre cada tabla. Este `ResultSet` devuelto tiene un formato determinado, cada columna contiene una información determinada de la tabla que se corresponde con la fila actual del `ResultSet`. Los nombres de estas columnas son:

- `TABLE_CAT`: catálogo de la tabla.
- `TABLE_SCHEM`: esquema de la tabla.
- `TABLE_NAME`: nombre de la tabla.
- `TABLE_TYPE`: tipo de la tabla.
- `REMARKS`: comentarios acerca de la tabla.

Algunas bases de datos no facilitan toda esta información, por lo que algunas de estas columnas pueden estar vacías. Nuestra página JSP va a recuperar, por cada tabla, el contenido de la columna `TABLE_NAME` y `TABLE_TYPE`, es decir, el nombre y el tipo de la tabla. Para recuperar cualquier columna de este `ResultSet` utilizaremos el método `getString()` del interfaz `ResultSet`, ya que toda la información que contiene el `ResultSet` es de tipo cadena de caracteres. En el Código Fuente 239 se muestra el código completo de la página JSP.

```
<%@page import="java.sql.*"%>
<html>
<head><title>DatabaseMetaData</title></head>
<body>
<%try{
    //Se registra el driver
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    //Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();
```



```

//Realización de la conexión
Connection conexion=DriverManager.getConnection
    ("jdbc:odbc:FuenteBD","sa","");
DatabaseMetaData dbmd=conexion.getMetaData();
out.println("<h3>Información sobre el DBMS:</h3>");
String producto=dbmd.getDatabaseProductName();
String version=dbmd.getDatabaseProductVersion();
boolean soportaSQL=dbmd.supportsANSI92EntryLevelSQL();
boolean soportaConvert=dbmd.supportsConvert();
boolean usaFich=dbmd.usesLocalFiles();
boolean soportaGRBY=dbmd.supportsGroupBy();
boolean soportaMinSQL=dbmd.supportsMinimumSQLGrammar();
String nombre=dbmd.getUserName();
String url=dbmd.getURL();
out.println("Producto: "+producto+" "+version+"<br>");
out.println("Soporta el SQL ANSI92: "+soportaSQL+"<br>");
out.println("Soporta la función CONVERT entre tipos SQL: "+soportaConvert+
    "<br>");
out.println("Almacena las tablas en ficheros locales: "+usaFich+"<br>");
out.println("Nombre del usuario conectado: "+nombre+"<br>");
out.println("URL de la Base de Datos: "+url+"<br>");
out.println("Soporta GROUP BY: "+soportaGRBY+"<br>");
out.println("Soporta la mínima gramática SQL: "+soportaMinSQL);
out.println("<h3>Información sobre el driver:</h3>");
String driver=dbmd.getDriverName();
String driverVersion=dbmd.getDriverVersion();
int verMayor=dbmd.getDriverMajorVersion();
int verMenor=dbmd.getDriverMinorVersion();
out.println("Driver: "+driver+" "+driverVersion+"<br>");
out.println("Versión superior del driver: "+verMayor+"<br>");
out.println("Versión inferior del driver: "+verMenor);
out.println("<h3>Tablas existentes:</h3>");
String patron="%";//listamos todas las tablas
String tipos[]=new String[2];
tipos[0]="TABLE";//tablas de usuario
tipos[1]="SYSTEM TABLE";//tablas del sistema
ResultSet tablas=dbmd.getTables(null,null,patron,tipos);
boolean seguir=tablas.next();
while(seguir){
    //Por cada tabla obtenemos su nombre y tipo
    out.println("    Nombre:"+tablas.getString("TABLE_NAME")+
        "    Tipo:"+tablas.getString("TABLE_TYPE")+"<br>");
    seguir=tablas.next();
}
conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca
    out.println("Se han dado excepciones SQLException<br>");
    out.println("=====<br>");
    //Pueden existir varias SQLException encadenadas
    while(ex!=null){
        out.println("SQLState :"+ex.getSQLState()+"<br>");
        out.println("Mensaje :"+ex.getMessage()+"<br>");
        out.println("Código de error :"+ex.getErrorCode()+"<br>");
        ex=ex.getNextException();
        out.println("<br>");
    }
}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%
    <%= "Se produjo una excepción: "+ex%>
<%}%>
</body>
</html>

```

En la Figura 115 se ofrece un ejemplo de ejecución de la página.

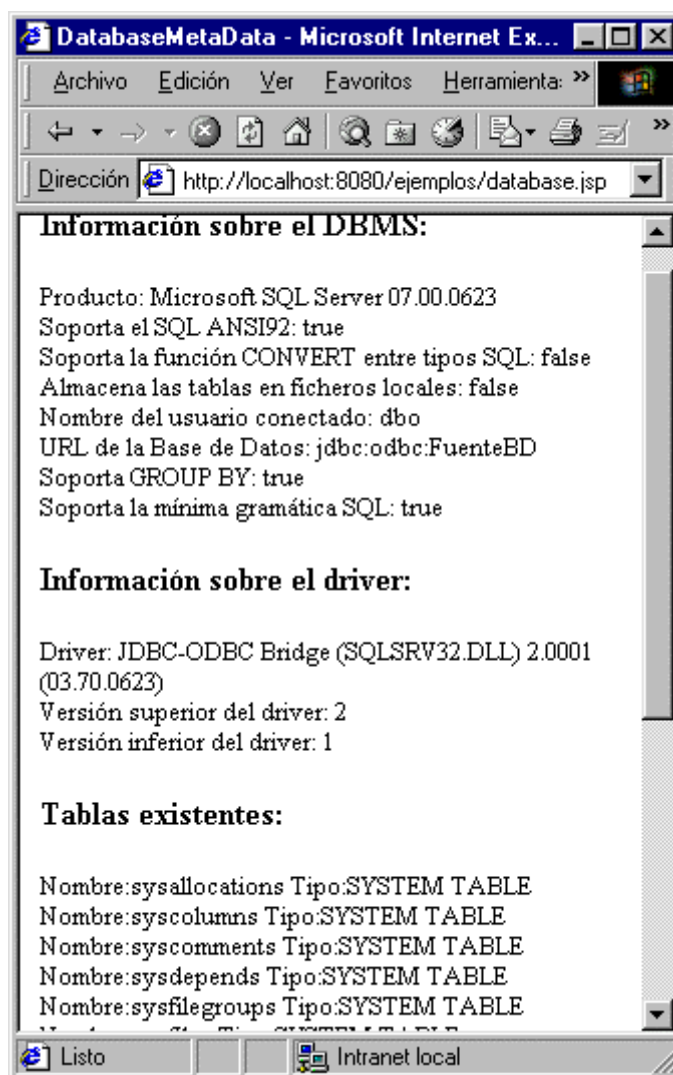


Figura 115: información sobre la base de datos

En el siguiente apartado vamos a tratar el interfaz `ResultSetMetaData`, que nos ofrecer información acerca de un `ResultSet` determinado.

## El interfaz `ResultSetMetaData`

El interfaz `ResultSetMetaData` nos ofrece una serie de métodos que nos permiten obtener información sobre las columnas que contiene el `ResultSet`. Es un interfaz más particular que el anterior, un objeto `DatabaseMetaData` es común a toda la base de datos, y cada objeto `ResultSetMetaData` es particular para cada `ResultSet`.

El número de métodos que ofrece este interfaz es mucho menor que el que ofrecía el interfaz `DatabaseMetaData`, veremos algunos de ellos a través de una página JSP de ejemplo.

Un objeto `ResultSetMetaData` lo obtendremos lanzando el método `getMetaData()` sobre el objeto `ResultSet` sobre la que queramos consultar la información de sus columnas, como se puede observar en el Código Fuente 240.

```
ResultSet rs=sentencia.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();
```

Código Fuente 240

Vamos a realizar una página JSP que nos va a ofrecer información de la columna que elija el usuario de una tabla que se encuentra dentro de la base de datos a la que previamente se ha debido conectar la página.

La página JSP debe ofrecer un interfaz bastante sencillo, que permita al usuario seleccionar la tabla y la columna. Es decir, se ofrecerá información sobre una columna (campo) dentro de una tabla.

El campo de la tabla se puede facilitar con su número de orden o bien con su nombre. La información sobre la columna aparecerá en la página.

Hay que tener en cuenta que el usuario puede facilitar el número del campo (1..n) o bien el nombre del campo. Si se indica el nombre del campo se calculará el número que le corresponde con el método `findColumn()` del interfaz `ResultSet`. También se obtiene un objeto `ResultSetMetaData`, que será el que nos facilite la información sobre el campo que nos ha indicado el usuario.

En nuestro ejemplo recuperamos alguna información del campo como puede ser: el nombre del campo, su tamaño, si es sólo de lectura, si distingue entre mayúsculas y minúsculas, etc.

En el Código Fuente 241 se puede observar el código de la página de ejemplo.

```
<%@page import="java.sql.*"%>
<html>
<head><title>ResultSetMetaData</title></head>
<body>
<form action="resmeta.jsp" method="GET">
    Tabla:<input type="text" name="tabla" value="" size="25"><br>
    Campo:<input type="text" name="columna" value="" size="25">
    <input type="submit" name="enviar" value="Información">
</form>
<%if (request.getParameter("enviar")!=null){
    try{
        String consulta="SELECT * FROM ";
        Statement sentencia;
        //Se registra el driver
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        //Driver driver=new sun.jdbc.odbc.JdbcOdbcDriver();
        //Realización de la conexión
        Connection conexion=DriverManager.getConnection

        ("jdbc:odbc:FuenteBD","sa","");
        //Se ejecuta la sentencia añadiéndole el nombre de la tabla.
        //Es necesaria la ejecución de esta sentencia para obtener un
        //objeto ResultSet con todo el contenido de la tabla seleccionada
        sentencia=conexion.createStatement();
        ResultSet rs=sentencia.executeQuery(consulta+
            request.getParameter("tabla"));
```

```

//Se obtiene el objeto ResultSetMetaData que nos ofrecerá
//información sobre el ResultSet
ResultSetMetaData rsmd=rs.getMetaData();
//Se recupera la columna de la que se debe mostrar
//la información. Este dato puede ser el número de columna o el nombre
//de la columna, por lo tanto de deben tener en cuenta las dos opciones.
int numCol;
try{
    numCol=Integer.parseInt(request.getParameter("columna"));
}catch(NumberFormatException ex){
    numCol=rs.findColumn(request.getParameter("columna"));
}
out.println("<h3>Información sobre la columna "+
    request.getParameter("columna")+" de la tabla "+
    request.getParameter("tabla")+"</h3>");
//Tipo JDBC de la columna
int jdbcType=rsmd.getColumnType(numCol);
//Nombre del tipo en el DBMS
String name=rsmd.getColumnTypeName(numCol);
//Nombre de la columna
String colName=rsmd.getColumnName(numCol);
//Es "case sensitive"
boolean caseSen=rsmd.isCaseSensitive(numCol);
//Es sólo de lectura
boolean readOnly=rsmd.isReadOnly(numCol);
//Tamaño máximo de la columna en caracteres
int tam=rsmd.getColumnDisplaySize(numCol);
//Título de la columna
String titulo=rsmd.getColumnLabel(numCol);
String clase=rsmd.getColumnClassName(numCol);
int precision=rsmd.getPrecision(numCol);
boolean auto=rsmd.isAutoIncrement(numCol);
boolean moneda=rsmd.isCurrency(numCol);
out.println("Nombre de la columna: "+colName+"<br>");
out.println("Número de orden de la columna: "+numCol+"<br>");
out.println("Tipo JDBC: " + jdbcType+"<br>");
out.println("Nombre del tipo en el DBMS: "+name+"<br>");
out.println("Es \"case sensitive\": "+caseSen+"<br>");
out.println("Es solamente de lectura: "+readOnly+"<br>");
out.println("Título de la columna: "+titulo+"<br>");
out.println("Tamaño máximo en caracteres: "+tam+"<br>");
out.println("Clase Java correspondiente: "+clase+"<br>");
out.println("Precisión de la columna: "+precision+"<br>");
out.println("Es autoincremental: "+auto+"<br>");
out.println("Es valor monetario: "+moneda+"<br>");
//Se cierra la sentencia
rs.close();
sentencia.close();
conexion.close();
}
catch(SQLException ex){
    //Se captura la excepción de tipo SQLException que se produzca
    out.println("Se han dado excepciones SQLException<br>");
    out.println("=====<br>");
    //Pueden existir varias SQLException encadenadas
    while(ex!=null){
        out.println("SQLState :"+ex.getSQLState()+"<br>");
        out.println("Mensaje :"+ex.getMessage()+"<br>");
        out.println("Código de error :"+ex.getErrorCode()+"<br>");
        ex=ex.getNextException();
        out.println("<br>");
    }
}
catch(Exception ex){
    //Se captura cualquier tipo de excepción que se produzca%>
    <%="Se produjo una excepción: "+ex%>
<% }
}%>

```

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

Código Fuente 241

La Figura 116 es un ejemplo de ejecución de esta página.

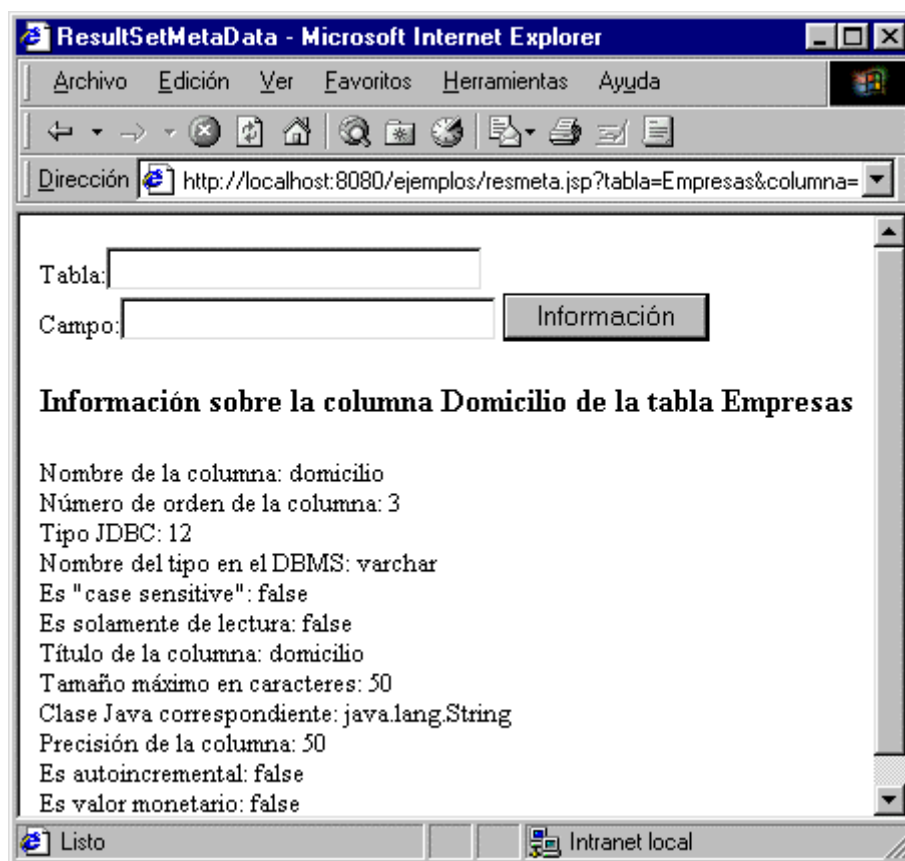


Figura 116. Obteniendo información de las columnas de un ResultSet

Con este apartado damos por finalizada la visión general de JDBC que pretendíamos realizar, en el resto del capítulo vamos a comprobar como podemos utilizar el API JDBC a través de componentes JavaBeans dentro de nuestras páginas JSP.

## Los componentes JavaBeans y JDBC

En este apartado vamos a comentar mediante un ejemplo una de las posibles formas que tenemos de utilizar a través de los componentes JavaBeans el API JDBC. Para ello vamos a retomar el ejemplo anterior en el que se devolvía el domicilio de una empresa determinado, y lo vamos a modificar para que dado el código de una empresa se devuelva toda la información relativa a dicha empresa.

En este caso vamos a utilizar un componente JavaBean que va a representar un registro de la tabla de empresas, y por lo tanto toda la lógica del acceso a la base de datos la vamos a desplazar desde la página JSP al Bean.

Además vamos a utilizar el Bean como si fuera un Bean de sesión, por lo que no se conectará y desconectará de a base de datos cada vez que queramos obtener los datos de una empresa. La desconexión se producirá cuando se destruya el componente, por lo tanto deberemos implementar el interfaz `javax.servlet.http.HttpSessionBindingListener`, para detectar cuando se agrega el Bean a la sesión y cuando se elimina de la misma. Lo mismo ocurrirá con la sentencia, se creará una única vez y luego se irá utilizando a lo largo de la vida del componente.

Para obtener la información de una empresa determinada se ha utilizado un procedimiento almacenado llamado `DevuelveEmpresa` y cuyo código se ofrece a continuación (Código Fuente 242).

```
CREATE PROCEDURE DevuelveEmpresa
    @codigo int
AS
    select * from empresas where codigo=@codigo
```

Código Fuente 242

Internamente el componente utiliza una sentencia `CallableStatement` para realizar la llamada al procedimiento almacenado que nos devolverá los datos de la empresa. Esta llamada se realizará cuando modifiquemos la propiedad `codigo` del componente, esta propiedad sería una propiedad desencadenante, ya que produce una actualización en los valores de las propiedades relacionadas como son `empresa` y `domicilio`.

El constructor del componente inicializa la propiedad `código` a cero, ya que si este vale cero indicará que no se ha especificado todavía ningún valor para la propiedad o bien que el valor establecido no ha sido encontrado en la base de datos. Precisamente en los métodos que devuelve el valor de la propiedad `domicilio` y `empresa` se comprueba previamente si el valor de la propiedad `codigo` es mayor que cero, si no es mayor que cero se devolverán valores nulos a la hora de recuperar los valores de las propiedades `empresa` y `domicilio`.

En la Tabla 21 se ofrece la hoja de propiedades del componente `EmpresaBean`, como se puede comprobar también se indican los valores que se asignan por defecto a algunas de las propiedades.

Nombre	Acceso	Tipo Java	Valor por defecto
Codigo	Sólo escritura	int	0
Conexion	Sólo escritura	String	jdbc:odbc:FuenteBD;UID=Sa;PWD=
Empresa	Sólo lectura	String	null
Domicilio	Sólo lectura	String	null

Tabla 21. Hoja de propiedades de `EmpresaBean`

Como ya hemos comentado las propiedades `empresa` y `domicilio` dependen de la propiedad `codigo`, y además el resto de las propiedades depende de la propiedad `conexión`, ya que al establecer la propiedad `conexión` se deberán reestablecer el resto de las propiedades a sus valores por defecto.

A continuación en el Código Fuente 243 se ofrece el código completo de la clase del componente `JavaBean`.

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

public class EmpresaBean implements HttpSessionBindingListener{

    private CallableStatement sentencia;
    private Connection conexion;
    private String sSentencia="{call DevuelveEmpresa(?)}";
    private int codigo;
    private String empresa;
    private String domicilio;

    public EmpresaBean(){
        codigo=0;
        empresa=null;
        domicilio=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            conexion=DriverManager.getConnection("jdbc:odbc:FuenteBD"
                                                , "sa", "");
            sentencia=conexion.prepareCall(sSentencia);
        }catch(Exception ex){
            System.out.println("Se ha producido una excepcion al crear "+
                               "el componente:"+ex);
        }
    }

    //propiedad de sólo escritura
    public void setConexion(String sConex){
        codigo=0;
        empresa=null;
        domicilio=null;
        try{
            conexion=DriverManager.getConnection(sConex);
            sentencia=conexion.prepareCall(sSentencia);
        }catch(Exception ex){
            System.out.println("Se ha producido una excepcion al realizar "+
                               "la conexión:"+ex);
        }
    }

    //propiedad de sólo escritura
    public void setCodigo(int cod){
        codigo=0;
        try{
            ResultSet resultado;
            sentencia.setInt(1,cod);
            resultado=sentencia.executeQuery();
            if (resultado!=null){
                if(resultado.next()){
                    empresa=resultado.getString("empresa");
                    domicilio=resultado.getString("domicilio");
                    codigo=cod;
                }
            }
            resultado.close();
            sentencia.clearParameters();
        }catch(Exception ex){
            codigo=0;
            System.out.println("Se ha producido una excepcion al "+
                               "establecer la propiedad codigo:"+ex);
        }
    }
}
```

```

        //propiedad de sólo lectura
        public String getEmpresa(){
            if(codigo>0)
                return empresa;
            else
                return null;
        }

        //propiedad de sólo lectura
        public String getDomicilio(){
            if(codigo>0)
                return domicilio;
            else
                return null;
        }

        public void valueBound(HttpSessionBindingEvent evento){}

        public void valueUnbound(HttpSessionBindingEvent evento){
            try{
                conexion.close();
            }catch(Exception ex){
                System.out.println("Se ha producido una excepcion al cerrar "+
                                   "la conexión:"+ex);
            }
        }
    }
}

```

Código Fuente 243

Este componente lo vamos a utilizar desde una página JSP y va a tener ámbito de sesión. La página JSP presenta un formulario que nos va a permitir facilitar el código de la empresa cuya información deseamos obtener. Se establecerá la propiedad `codigo` con el valor del parámetro del objeto `request` correspondiente y a continuación se obtendrá el valor de las propiedades `domicilio` y `empresa`.

Además del formulario que permite indicar el código de empresa, existe un segundo formulario dentro de la página JSP que permite modificar la cadena de conexión. Hay que tener en cuenta que sólo podemos utilizar fuentes de datos ODBC para establecer la conexión, ya que el componente `JavaBean` únicamente ha registrado el driver de tipo 1, si queremos hacer una conexión con otro tipo de driver se debería modificar el código del componente para que lo registre en su constructor.

En el Código Fuente 244 se puede ver el aspecto del código de esta página JSP que nos permite utilizar el componente `EmpresaBean`.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>JDBC/Bean</title>
</head>
<body>
<form action="BeanJDBC.jsp" method="GET">
    Código de empresa:
    <input type="text" name="codigo" value="" size="3">
    <input type="submit" name="enviar" value="Obtener datos">
</form><br><br>
<form action="BeanJDBC.jsp" method="GET">
    Cadena conexión:
    <input type="text" name="conex" value="" size="25">
    <input type="submit" name="conectar" value="Modificar conexión">
</form>

```



```

<%if (request.getParameter("conectar")!=null ||
request.getParameter("enviar")!=null){%>
    <jsp:useBean id="empresa" scope="session" class="EmpresaBean"/>
<%}
if (request.getParameter("conectar")!=null){%>
    <jsp:setProperty name="empresa" property="conexion" param="conex"/>
<%}
if (request.getParameter("enviar")!=null){%>

    Los datos de la empresa con código
    <i><%=request.getParameter("codigo")%></i> son:<br>
    <jsp:setProperty name="empresa" property="*" />
    <ul>
    <li>Nombre empresa:<b><i>
    <jsp:getProperty name="empresa" property="empresa"/></i></b>
    <li>Dirección:<b><i>
    <jsp:getProperty name="empresa" property="domicilio"/></i></b>
    </ul>
<%}%>
</body>
</html>

```

Código Fuente 244

Y en la Figura 117 se puede ver un resultado de la ejecución de la página.

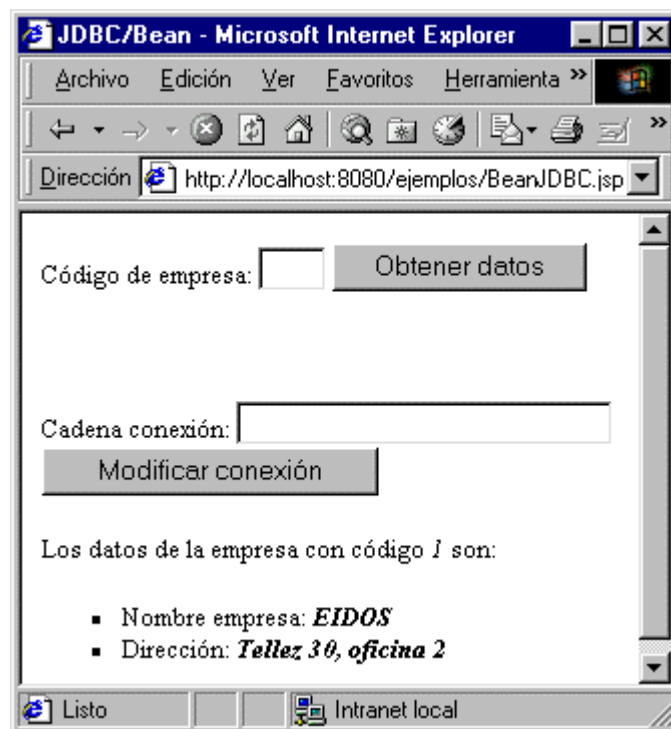


Figura 117. Utilizando el componente EmpresaBean

Con este apartado concluimos el presente capítulo y también finalizamos la parte del texto dedicado a JDBC y su utilización desde JSP. En el siguiente capítulo vamos a tratar el mecanismo que nos permite definir nuevas etiquetas para nuestras páginas JSP, es decir, veremos como crear y utilizar nuestras propias librerías de etiquetas personalizadas para ampliar de esta forma las acciones estándar que presenta la especificación JSP.



# Etiquetas personalizadas

---

## Introducción

En este capítulo vamos a comentar el mecanismo que nos ofrece la especificación JSP para poder ampliar las acciones disponibles en las páginas JSP. Se recuerda al lector que las acciones son una serie de etiquetas ofrecidas por la especificación JSP para realizar una tarea determinada, como puede ser el transferir la ejecución de una página JSP a otra página JSP o recurso.

Existen siete acciones estándar dentro de la especificación JSP. Estas etiquetas afectan le comportamiento de la ejecución de las páginas JSP y afectan también a la respuesta devuelta al cliente. Las acciones estándar ofrecen al desarrollador o autor de páginas JSP una funcionalidad básica.

Las acciones estándar son:

- `<jsp:forward>`
- `<jsp:include>`
- `<jsp:plugin>`
- `<jsp:param>`
- `<jsp:useBean>`
- `<jsp:setProperty>`

- `<jsp:getProperty>`

Pero no tenemos que quedarnos restringidos a estas siete acciones, sino que nosotros mismos podemos definir las acciones que sean necesarias para nuestras páginas JSP, es decir podemos definir nuestras propias etiquetas, esto se consigue mediante un mecanismo incorporado en la versión 1.1 de la especificación JSP denominado librerías de etiquetas personalizadas o simplemente etiquetas personalizadas. En este capítulo comentaremos tanto el uso como la creación de estas etiquetas personalizadas, que nos permiten aumentar la potencia de nuestras páginas JSP a las que les podremos asignar nuevas acciones, en este caso personalizadas.

Gracias a las etiquetas personalizadas podemos eliminar código Java de nuestras páginas JSP, por lo que conseguimos una mayor separación y distinción entre la lógica y la presentación. Nuestro objetivo debe ser mantener las páginas JSP lo mas sencillas posible, esto lo conseguíamos también con la utilización de las acciones para el uso de componentes JavaBeans, es decir, con las acciones `<jsp:useBean>`, `<jsp:getProperty>` y `<jsp:setProperty>`. Utilizando estas acciones eliminamos gran cantidad de scriptlets de nuestras páginas JSP, pero sólo disponemos de estas tres etiquetas estándar para poder utilizar código Java localizado en componentes.

Con la utilización del mecanismo de etiquetas personalizadas podemos definir nuestras propias etiquetas que realizarán las funciones deseadas, y nuestras páginas JSP las utilizarán de forma muy similar a como utilizamos las acciones estándar ofrecidas por la especificación JSP.

Por lo tanto mediante las etiquetas personalizadas vamos a añadir funcionalidad a las páginas JSP, de esta forma podremos encapsular una funcionalidad específica. Las etiquetas personalizadas se suelen utilizar para generar HTML de forma dinámica o para controlar la página de alguna forma. Los JavaBeans se suelen utilizar para mantener información de la sesión o implementar la lógica de la aplicación, se puede decir que las etiquetas personalizadas se encuentran en un término medio entre la página JSP y el componente JavaBean.

Las etiquetas suelen estar más informadas del entorno en el que se están ejecutando que los Beans, tiene acceso al contexto de la página JSP en la que se ejecutan. Es posible especificar atributos para las etiquetas personalizadas, que actuarán como parámetros que pueden modificar el comportamiento de las mismas. También estas etiquetas pueden contener cuerpo con código para ser procesado por las etiquetas o presentar el cuerpo vacío, al igual que ocurría con la acción estándar `<jsp:useBean>`. Incluso las etiquetas pueden interactuar entre sí intercambiando información que atiende a una jerarquía de anidación de las mismas.

La implementación de las etiquetas personalizadas se encuentra realizada a través de objetos Java, para definir el comportamiento y funcionalidad de la etiqueta deberemos implementar una clase Java que podrá acceder fácilmente a todo el API disponible dentro del lenguaje.

Para poder utilizar una etiqueta personalizada determinada en una página JSP se debe cargar la librería que contiene la etiqueta, y esto lo haremos utilizando la directiva `taglib`.

## La directiva `taglib`

Esta directiva de la especificación JSP, que ya adelantamos hace unos cuantos capítulos cuando tratábamos sobre los elementos de las páginas JSP, es utilizada para indicar al contenedor de páginas JSP que la página JSP actual utiliza una librería de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas personalizadas que extienden la funcionalidad de una página JSP. Una vez que se ha utilizado esta directiva para indicar la librería de etiquetas que se van a utilizar, todas las etiquetas personalizadas definidas en la librería están a nuestra disposición para hacer uso de ellas en nuestra página JSP actual.

Más adelante en este mismo capítulo veremos las reglas para definir los descriptores de las librerías de etiquetas, un descriptor librería de etiquetas va a consistir en un fichero en formato XML que va a poseer la extensión TLD y que va a indicar toda la información necesaria para utilizar una librería de etiquetas.

La sintaxis de la directiva taglib es la siguiente:

```
<%@ taglib uri="URLLibreria" prefix="prefijoEtiquetas"%>
```

Y como el resto de las directivas de la especificación JSP tiene su equivalente en XML:

```
<jsp:directive.taglib uri="URLLibreria" prefix="prefijo"/>
```

Esta directiva posee dos atributos, el primero de ellos, llamado uri, se utiliza para indicar la localización del descriptor de la librería de etiquetas (TLD, Tag Library Descriptor), y el segundo atributo, llamado prefix indica el prefijo que se va a utilizar para hacer referencia a las etiquetas personalizadas.

El descriptor de la librería de etiquetas es un fichero especial que indica las clases que implementan y forman una librería, también indica el nombre de las etiquetas y los atributos que poseen.

La siguiente página JSP (Código Fuente 108) utiliza dos librerías de etiquetas, que se encuentran definidas en los descriptores de etiquetas LIBRERIAETIQUETAS.TLD Y UTILIDADES.TLD, a la primera librería se le asigna el prefijo libreria y a la segunda el prefijo utils. Dentro de estas librerías se utilizan dos etiquetas, la etiqueta tam para obtener el tamaño de los ficheros que se indican en su atributo uri, y la etiqueta buffer que muestra información relativa al búfer que se ha utilizado y el tanto por ciento que queda libre.

```
<html>
<title>Directivas</title>
<body>
<%@taglib uri="tlds/libreriaEtiquetas.tld" prefix="libreria"%>
<%@taglib uri="tlds/utilidades.tld" prefix="utils"%>
Tamaños de ficheros:<br>
El fichero fich1.jpg tiene el tamaño de <libreria:tam uri="fich1.jpg"/><br>
El fichero fich2.java tiene el tamaño de <libreria:tam uri="fich2.java"/><br>
<br>
Información del búfer:
<br>
<utils:buffer/>
</body>
</html>
```

Código Fuente 245

El resultado de la ejecución de esta página JSP se puede ver en la Figura 57. Como se puede apreciar la utilización de las etiquetas personalizadas es bastante sencilla, las utilizaremos igual que cualquier otra acción estándar de la especificación JSP, pero indicando previamente mediante la directiva taglib los datos relativos a la librería de etiquetas.

En este ejemplo además de mostrar como se utilizan las etiquetas personalizadas se ha mostrado la funcionalidad que pueden ofrecer estas etiquetas, una de ellas indica el tamaño del fichero que le pasamos por parámetro, y otra de ellas ofrece información relativa a la utilización del búfer de salida de la página, como se puede ver son funcionalidades sencillas que devuelven código HTML para mostrar en la página resultante, más adelante veremos como implementar las clases que nos permiten

definir estas funcionalidades, ya que detrás de cada etiqueta existe una clase Java que realiza la implementación de la misma.

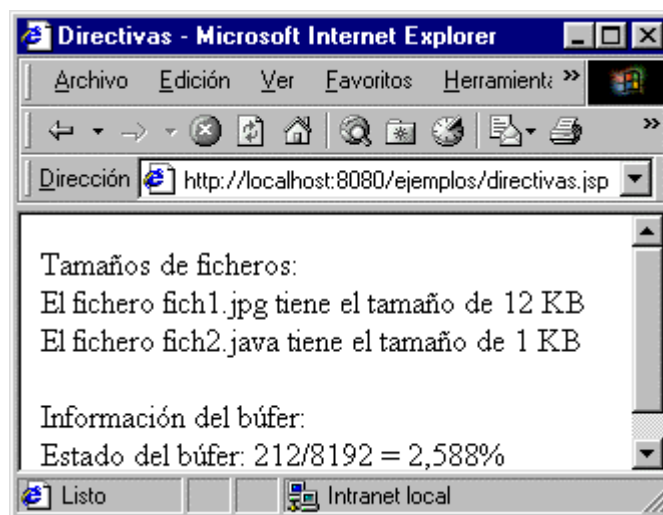


Figura 118. utilizando la directiva taglib

Cuando el contenedor de páginas JSP carga una página JSP que contiene etiquetas personalizadas, la directiva taglib determinará si necesita cargar la librería o no. Si es la primera vez que se ha realizado la petición para la librería desde una la página JSP el contenedor JSP debe leer el fichero TLD indicado en el atributo uri de la directiva taglib, sin embargo si la librería ya había sido demandada anteriormente por otra página no se volverá a cargar.

A la hora de compilar una página el contenedor de páginas JSP necesita únicamente examinar le fichero descriptor de la librería de etiquetas (fichero TLD), para validar la utilización de las etiquetas personalizadas dentro de la página JSP. Cada etiqueta personaliza que se encuentre en la página se compara con la especificación correspondiente del descriptor de la librería.

En el siguiente apartado vamos a tratar los descriptors de librerías de etiquetas, comentaremos cada uno de los elementos que contiene.

## Descriptor de librería de etiquetas

Un fichero descriptor de una librería de etiquetas es un documento en formato XML que posee la extensión TLD (Tag Library Descriptor) y que contiene información relativa a cada etiqueta que contiene la librería, como puede ser las clases que implementan y forman una librería o el nombre de las etiquetas y los atributos que poseen.

Un fichero TLD debe incluir una cabecera estándar de información XML, que suele tener el aspecto del Código Fuente 246.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

Código Fuente 246

Un descriptor de librería de etiquetas va a contener una serie de elementos (etiquetas XML) que le permiten especificar y definir la información de las distintas etiquetas que contiene. El primer elemento que aparece dentro de un fichero TLD es la etiqueta `<taglib>`, esta etiqueta es utilizada para describir la librería de manera genérica, para ello se sirve de cinco subelementos o subetiquetas, que son las que vamos a comentar a continuación.

- `<tlibversion>`: etiqueta obligatoria que indica el número de versión de la librería de etiquetas. Según se vayan realizando nuevas versiones de la librería se debería actualizar el valor de esta etiqueta, que se indicará en el cuerpo de la misma. Los valores para las subetiquetas de la etiqueta `<taglib>` se indican siempre en el cuerpo de las mismas, ya que ninguno de estos elementos poseen atributos.
- `<jspversion>`: etiqueta opcional que indica la versión de la especificación JSP con la que es compatible la librería de etiquetas, su valor por defecto es 1.1, ya que la versión 1.0 de la especificación JSP no soporta las etiquetas personalizadas.
- `<shortname>`: etiqueta obligatoria que indica el nombre abreviado de la librería de etiquetas, este será el nombre que se utilizará dentro de la página JSP correspondiente para utilizar la librería de etiquetas, se utilizará como el prefijo por defecto para el espacio de nombres cuando se utilice la librería dentro de páginas JSP.
- `<info>`: etiqueta opcional que muestra una descripción de la librería de etiquetas, se suele utilizar para auto documentar la librería, su valor por defecto es el de una cadena vacía.
- `<uri>`: etiqueta también opcional que indica una fuente de documentación adicional para la librería, contiene una URL que apunta a la documentación existente acerca de la librería.

A continuación, en el Código Fuente 247, se ofrece el aspecto que tendría el elemento `<taglib>` de un fichero descriptor de librería de etiquetas.

```
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>libreria</shortname>
  <uri>http://angel.com/librerias/lib.tld</uri>
  <info>
    Un ejemplo de librería de etiquetas
  </info>
  .....
</taglib>
```

Código Fuente 247

Un elemento `<taglib>` requiere que se definan subelementos `<tag>`, cada elemento `<tag>` define cada etiqueta personalizada que contiene la librería. La etiqueta `<tag>` posee también cinco elementos o subetiquetas para definir una etiqueta personalizada que forma parte de la librería actual, estos subelementos se describen a continuación, al igual que ocurría con los elementos anteriores, no poseen atributos y sus valores se indican en el cuerpo de cada una de las etiquetas.

- `<name>`: etiqueta obligatoria que indica el nombre utilizado para identificar la etiqueta, se utilizará dentro de la página JSP correspondiente en combinación con el prefijo de la librería en la que se encuentra definida.

- `<tagclass>`: etiqueta obligatoria que indica la clase que implementa esta etiqueta, estas clases se denominan manejadores de etiquetas (tag handler). Más adelante veremos como crear estas clases que son las que implementan la funcionalidad ofrecida por la etiqueta.
- `<teiclass>`: etiqueta opcional que especifica la clase de la clase de ayuda (helper class) para esta etiqueta. Una clase de ayuda será utilizada para validar los valores y dependencias de los distintos atributos que puede tener una etiqueta personalizada.
- `<bodycontent>`: etiqueta opcional que indica el tipo de contenido que puede aparecer en el cuerpo de la etiqueta personalizada, es decir, entre la etiqueta de apertura y de cierre. Existen tres valores válidos para este elemento, que son `empty`, `JSP` y `tagdependent`. El valor `empty` indica que el cuerpo de la etiqueta no tendrá ningún contenido, el valor `JSP` indica que el cuerpo de la etiqueta puede contener elementos JSP adicionales, incluyendo otras etiquetas personalizadas, y el valor `tagdependent` indica que el cuerpo de la etiqueta debe ser interpretado por la propia etiqueta.
- `<info>`: este elemento opcional es utilizado para documentar la etiqueta, su valor por defecto es una cadena vacía.

En el Código Fuente 248 se puede observar un ejemplo de utilización de un elemento `<tag>` para definir una etiqueta.

```
<tag>
  <name>buffer</name>
  <tagclass>EstadoBuffer</tagclass>
  <bodycontent>empty</bodycontent>
  <info>Muestra la utilización del buffer de la página actual</info>
  .....
</tag>
```

Código Fuente 248

Para finalizar con los elementos que podemos encontrar en un fichero TLD vamos a comentar la etiqueta `<attribute>` que se utilizará como un subelemento de la etiqueta `<tag>` cuando una etiqueta personalizada utilice atributos. A su vez la etiqueta `<attribute>` posee tres subelementos, que son los que se describen a continuación:

- `<name>`: etiqueta requerida que se utiliza para identificar el atributo correspondiente, representa la cadena que se deberá usar cuando el atributo se quiera utilizar en la etiqueta personalizada correspondiente.
- `<required>`: etiqueta opcional que indica si el atributo de la etiqueta personalizada es requerido o no, puede presentar los valores `true` o `false`, siendo su valor por defecto `false`.
- `<rtexprvalue>`: este elemento opcional indica si una expresión de JSP se puede utilizar para especificar el valor del atributo, los valores que puede tener este elemento son `true` o `false`, y su valor por defecto es `false`.

En el Código Fuente 249 se muestra un ejemplo de la definición de un atributo para una etiqueta personalizada, y el Código Fuente 250 muestra un ejemplo de utilización de todos los elementos principales que hemos visto hasta ahora es decir, de las etiquetas `<taglib>`, `<tag>` y `<attribute>`, sería el contenido completo de un fichero TLD que define una librería con una única etiqueta.



```

<attribute>
  <name>uri</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>

```

Código Fuente 249

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>libreria</shortname>
  <uri>libreria1</uri>
  <info>
    Un ejemplo de librería
  </info>

  <tag>
    <name>tam</name>
    <tagclass>EtiquetaTam</tagclass>
    <info>Muestra el tamaño de un fichero</info>
    <attribute>
      <name>uri</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>

```

Código Fuente 250

Una vez definido nuestro fichero TLD debemos registrar la librería de etiquetas en nuestra aplicación Web, para poder utilizarla desde las páginas JSP forman parte de la aplicación. Para ellos debemos acceder al fichero de configuración de la aplicación Web, es decir, al fichero WEB.XML correspondiente y añadir una etiqueta <taglib> que identifique la librería de etiquetas que deseamos utilizar en la aplicación Web.

La etiqueta <taglib> dentro del fichero WEB.XML presenta dos subelementos distintos a los del fichero descriptor de librerías de etiquetas, ya que aquí la función de esta etiqueta es distinta, en este caso deseamos registrar la librería de etiquetas en una aplicación Web. Los subelementos que la etiqueta <taglib> presenta son los comentados a continuación:

- <taglib-uri>: esta etiqueta contiene una URL que va a identificar a la librería de etiquetas, no se trata de una URL real, sino la que se va a utilizar en el atributo de la directiva taglib para identificar una librería determinada.
- <taglib-location>: esta etiqueta indica la ruta real dónde se localiza el fichero TLD.

En el Código Fuente 251 se muestra un fragmento del fichero WEB.XML de una aplicación Web que va a utilizar una librería de etiquetas.

```
<taglib>
  <taglib-uri>
    libreriaEtiquetas.tld
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/libreriaEtiquetas.tld
  </taglib-location>
</taglib>
```

Código Fuente 251

Los ficheros TLD se suelen situar en el directorio WEB-INF/TLDS de la aplicación Web.

Ya sabemos como utilizar una etiqueta personalizada desde una página JSP y también como definir el fichero descriptor de una librería y como registrar dicha librería de etiquetas en una aplicación Web, únicamente nos queda mostrar como implementar la etiqueta a través de su clase correspondiente, denomina manejador de etiqueta.

## Manejadores de etiquetas

Este nombre los reciben las clases que se utilizan para implementar las funcionalidades que ofrecen las etiquetas personalizadas. Los objetos manejadores de etiquetas son objetos que realizan la acción asociada con una etiqueta personalizada. Cuando una petición de una página JSP que contiene etiquetas personalizadas es recibida por el contenedor de páginas JSP, cada vez que se encuentra una etiqueta personalizada se crea una instancia del manejador de la etiqueta correspondiente.

El manejador de la etiqueta se inicializará con los valores de los atributos especificados en la etiqueta, si es que los tiene, y varios métodos del manejador de sentencias se invocarán para realizar la acción correspondiente, una vez que se ha finalizado la acción correspondiente el manejador de la etiqueta se devuelve a un conjunto de recursos para su posterior reutilización.

Los métodos que el manejador de la etiqueta debe ofrecer son definidos por los interfaces `javax.servlet.jsp.tagext.Tag` y `javax.servlet.jsp.tagext.BodyTag`, pero para simplificar el desarrollo de los manejadores de etiquetas podemos heredar de las clases `javax.servlet.jsp.tagext.TagSupport` y `javax.servlet.jsp.tagext.BodyTagSupport`. Estas clases ofrecen una implementación por defecto de todos los métodos del interfaz correspondiente, y por lo tanto sólo se deberá redefinir en nuestra clase manejadora aquellos métodos que sean necesarios para implementar la acción deseada de la etiqueta personalizada.

Las etiquetas personalizadas que tengan que procesar el contenido de su cuerpo de alguna manera, estarán implementadas por clases manejadoras que hereden de la clase `BodyTagSupport`, y aquellas etiquetas en las que no sea necesario interpretar el contenido de su cuerpo poseerán una clase manejadora que heredará de la clase `TagSupport`.

Cuando una clase manejadora de etiquetas hereda de la clase `TagSupport` sigue un ciclo de vida determinado. El primero de los pasos de este ciclo de vida es obtener una instancia de la clase apropiada, ya sea desde un conjunto de recursos o mediante la creación de una nueva instancia. A continuación el método `setContext()` es lanzado por el contenedor de página JSP para asignarle un objeto `pageContext` al objeto manejador de etiquetas, de esta forma podrá acceder al contexto de la página en la que se está ejecutando la etiqueta personalizada correspondiente. En ese momento el contenedor JSP llama al método `setParent()`, esto permite acceder a una instancia de una etiqueta padre si es que existe, que incluye a la etiqueta actual.

Después de que se han establecido estas propiedades, se establecen los valores de los atributos de la etiqueta, si es que los hemos definido. Los atributos de las etiquetas son tratados de la misma forma que las propiedades de los componentes JavaBeans, es decir, a través de los métodos de acceso con una nomenclatura específica, estos métodos habrá que definirlos en la clase manejadora de la etiqueta. Así por ejemplo si una etiqueta determina poseer el atributo `id` que es de lectura y escritura, la clase manejadora de la etiqueta debe ofrecer un método `getId()` y un método `setId()`, es decir, como si fuera una propiedad de un Bean.

Llegados a este punto el contenedor de páginas JSP llamará al método `doStartTag()`, hasta ahora toda la información ha sido ofrecida de forma automática por el contenedor de página JSP, es en el método `doStartTag()` donde debemos ofrecer la implementación de la acción correspondiente asociada con la etiqueta personalizada a la que representa la clase actual.

Por lo tanto nuestra clase manejadora de etiquetas debe implementar el método `doStartTag()` la acción que se quiera llevar a cabo en la etiqueta.

El método `doStartTag()` devolverá un entero que coincidirá con la constante `Tag.SKIP_BODY` o `Tag.EVAL_BODY_INCLUDE`.

El primer valor indica al contenedor de páginas JSP que se deben ignorar los contenidos del cuerpo de la etiqueta personalizada, y el segundo valor indica que los contenidos del cuerpo de la etiqueta deben ser procesados normalmente.

En cualquier caso el siguiente paso del ciclo de vida de nuestra clase manejadora de etiquetas es que el contenedor de páginas JSP lance el método `doEndTag()`.

En este método también podremos incluir las acciones que sean necesarias para la etiqueta, es decir, si nos interesa podemos dar una implementación a este método dentro de nuestra clase.

El método `doEndTag()` devuelve también un valor entero que se corresponde con otras dos constantes llamadas `Tag.SKIP_PAGE` y `Tag.EVAL_PAGE`.

Si se devuelve el valor `Tag.SKIP_PAGE` en el método `doEndTag()` se indicará al contenedor de páginas JSP que finalice el procesamiento de la página JSP Actual, es decir, se ignorará el contenido restante de la página, ya sean scriptlets o elementos estáticos, finaliza la ejecución de la página JSP.

Por el contrario si el método `doEndTag()` devuelve el valor `Tag.EVAL_PAGE`, el resto de la página se procesará de forma usual.

Con independencia del valor devuelto por el método `doEndTag()` la fase final del ciclo de vida de la clase manejadora de etiquetas es la invocación por parte del contenedor de páginas JSP del método `release()`.

Este método ofrece la oportunidad al manejador de la etiqueta de realizar labores de limpieza, antes de que la instancia del manejador de la etiqueta personalizada sea enviado a un conjunto de recursos para ser reutilizado.

En la Figura 119 se ofrece un esquema del ciclo de vida que sigue un manejador de una etiqueta personalizada que hereda de la clase `TagSupport`.

Si el manejador de la etiqueta hereda de la clase `BodyTagSupport` el ciclo de vida del manejador es muy similar al anterior, difiere cuando se devuelve el resultado del método `doStartTag()`. En este caso los valores que puede devolver el método `doStartTag()` son `Tag.SKIP_BODY` y `BodyTag.EVAL_BODY_TAG`.

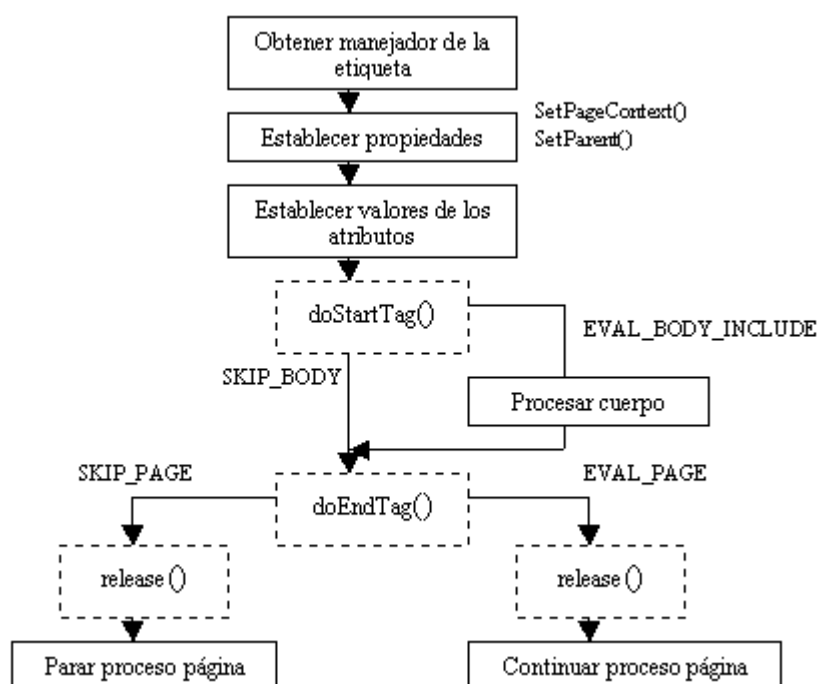


Figura 119. Ciclo de vida de un manejador de tipo TagSupport

Al igual que en el caso anterior el valor `Tag.SKIP_BODY` indica que se el contenido del cuerpo de la etiqueta se debe ignorar, sin embargo si el método `doStartTag()` devuelve el valor `BodyTag.EVAL_BODY_TAG` indicará al contenedor que el contenido del cuerpo de la etiqueta se debe procesar y los resultados de su ejecución se deben almacenar para manipulaciones futuras por parte del manejador de la etiqueta.

Los resultados del procesamiento del cuerpo de la etiqueta se almacenan a través de la clase `BodyContent`, que es una subclase de la clase `JspWriter`. De esta forma el manejador de la etiqueta puede decidir si el resultado de la ejecución del cuerpo de la etiqueta se va a enviar al navegador del usuario o no

Para procesar el cuerpo de una etiqueta el primer paso es obtener una instancia de a clase `BodyContent`, esto lo hace de forma automática el contenedor de páginas JSP llamando al método `setBodyContent()`. El contenedor llamará a continuación al método `doInitBody()`, para ofrecer al manejador de la etiqueta la oportunidad de realizar alguna labor de inicialización después de que la instancia de la clase `BodyContent` ha sido asignada. En ese momento el cuerpo de la etiqueta es procesado y toda su salida es asignada a la instancia de la clase `BodyContent`.

Una vez procesado el contenido del cuerpo, el contenedor JSP invoca el método `doAfterBody()` del manejador de la etiqueta. Las acciones que se realicen en este método dependen de la implementación específica de la etiqueta personalizada correspondiente, y normalmente contiene interacciones con el objeto `BodyContent` de la etiqueta.

El método `doAfterBody()` puede devolver los valores que se corresponden con las constantes `Tag.SKIP_BODY` y `BodyTag.EVAL_BODY_TAG`, si se devuelve el segundo valor el contenido del cuerpo será evaluado una vez más, y después se volverá a llamar al método `doAfterBody()`. El procesamiento repetido del cuerpo de la etiqueta tendrá lugar hasta que el método `doAfterBody()` devuelva el valor `Tag.SKIP_BODY`. Este tipo de procesamiento se suele utilizar para etieutas que necesitan procesar su cuerpo de forma iterativa.

Cuando el método `doAfterBody()` ha devuelto el valor `Tag.SKIP_BODY` se ejecuta el método `doEndTag()` de la misma forma que se hacía con el manejador de etiqueta que heredaba de la clase `TagSupport`, siguiendo en este momento su mismo ciclo de vida.

El ciclo de vida completo de este tipo de manejadores de etiquetas, es decir, aquellos que procesan el cuerpo de la etiqueta se puede ver en el esquema mostrado en la Figura 120.

Una vez visto el ciclo de vida que poseen los manejadores de etiquetas, en el siguiente apartado vamos a ver varios ejemplos de implementación de etiquetas personalizadas con sus correspondientes clases manejadoras.

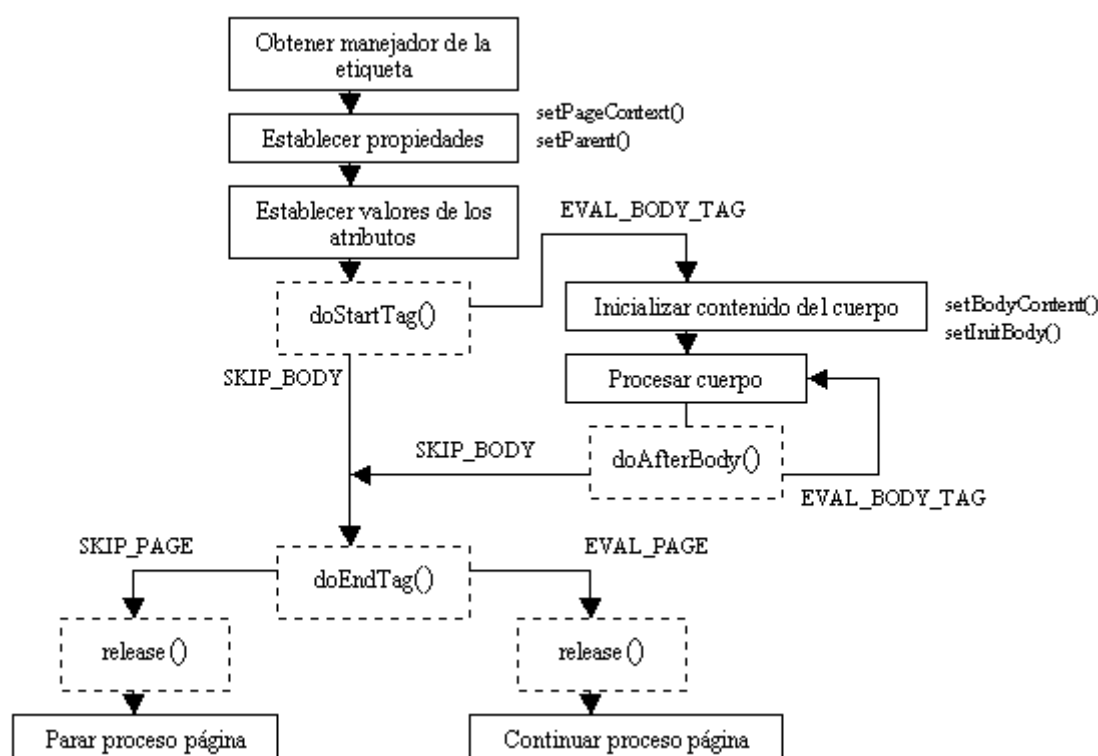


Figura 120. Ciclo de vida de un manejador del tipo `BodySupport`.

## Creación de etiquetas personalizadas

En este ejemplo vamos a aplicar todos los conceptos vistos hasta ahora en el presente capítulo a un sencillo ejemplo que consiste en la creación de una librería de etiquetas personalizadas.

La primera etiqueta personalizada que va a contener nuestra librería de etiquetas, se va a tratar de una etiqueta muy sencilla cuya labor es informar del estado actual del búfer de salida de la página JSP en la que se utiliza dicha etiqueta.

Es una etiqueta muy sencilla ya que no va a poseer atributos únicamente va a devolver información relativa a la utilización del búfer.

En un primer lugar debemos definir el fichero descriptor de la librería de etiquetas, es decir, el fichero TLD correspondiente. El código XML necesario (Código Fuente 252) para definir la librería de etiquetas y la etiqueta que nos ocupa es el que se muestra a continuación.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>libreria</shortname>
  <info>
    Un ejemplo de librería
  </info>

  <tag>
    <name>bufer</name>
    <tagclass>EstadoBuffer</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Muestra la utilización del buffer de la página actual</info>
  </tag>
</taglib>

```

Código Fuente 252

Como se puede ver a la etiqueta se le va a asignar el nombre bufer y a la librería en cuestión el nombre libreria, estos nombres se utilizarán para hacer referencia a la etiqueta dentro de la página JSP que utilice la librería que estamos definiendo. Nuestra etiqueta no va a tener cuerpo ni atributos, ya que para la funcionalidad que perseguimos no van a ser necesarios.

El fichero TLD lo nombraremos LIBRERIAETIQUETAS.TLD y lo guardaremos en el directorio WEB\_INF\TLDS de la aplicación Web en la que queramos utilizar la librería de etiquetas personalizadas.

El siguiente paso es implementar el manejador de la etiqueta, que del código anterior extraemos que su clase debe llamarse EstadoBuffer. Por lo tanto deberemos crear una clase llamada EstadoBuffer que herede de la clase TagSupport, ya que no vamos a procesar el cuerpo de la etiqueta, y que implemente el método doStartTag(). Será en el método doStartTag() dónde incluyamos las sentencias necesarias para mostrar la información de búfer al usuario.

Vamos a ver en un primer lugar el código fuente (Código Fuente 253) completo del manejador de la etiqueta, que resulta bastante sencillo, y a continuación lo comentaremos.

```

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.text.NumberFormat;

public class EstadoBuffer extends TagSupport{

    public int doStartTag() throws JspException{
        JspWriter out=pageContext.getOut();
        int total=out.getBufferSize();
        int disponible=out.getRemaining();
        int utilizado=total-disponible;
        try{
            out.print("Estado del búfer: ");
            out.print(Integer.toString(utilizado));
        }
    }
}

```

```

        out.print("/"+Integer.toString(total)+" = ");
        NumberFormat tantoPorCiento=NumberFormat.getInstance();
        tantoPorCiento.setMinimumFractionDigits(1);
        tantoPorCiento.setMaximumFractionDigits(3);
        out.print(tantoPorCiento.format((100D*utilizado)/total));
        out.print("%");
    }catch (IOException ex){
        throw new JspException(ex.getMessage());
    }

    return SKIP_BODY;
}
}

```

Código Fuente 253

La clase manejadora de la etiqueta implementa únicamente el método `doStartTag()`, dentro de este método recupera una referencia al objeto integrado de JSP `out`, este objeto es necesario para obtener la información relativa a la cantidad de búfer que se encuentra utilizada hasta este momento y el búfer que queda libre, y también para mostrar la información calculada.

La clase debe importar los paquetes que nos permiten utilizar la especificación JSP y también el mecanismo de etiquetas personalizadas. En este ejemplo el método `doStartTag()` devuelve el valor `Tag.SKIP_BODY`, ya que la etiqueta no va a presentar ningún cuerpo.

Una vez compilada la clase manejadora de la etiqueta deberemos copiarla al directorio común de todas las clases de las aplicaciones Web, como si se tratara de un componente `JavaBean` o de un `servlet`, es decir, debemos copiar el fichero de clase en el directorio `WEB-INF\CLASSES` de la aplicación Web que utilice la librería de etiquetas.

Ahora únicamente nos resta registrar la librería de etiquetas, en este caso de una única etiqueta, en la aplicación Web correspondiente. Para ello debemos modificar el fichero de configuración de la aplicación Web, es decir, el fichero `WEB.XML` y añadir la información que identificará a la etiqueta, el código que se debe añadir en este caso se puede ver en el Código Fuente 254.

```

<taglib>
  <taglib-uri>
    libreriaEtiquetas.tld
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/libreriaEtiquetas.tld
  </taglib-location>
</taglib>

```

Código Fuente 254

En este momento ya podemos utilizar la etiqueta personalizada en una página JSP, para ello deberemos utilizar dentro de la página JSP correspondiente la directiva `taglib` para hacer referencia a la librería de etiquetas que acabamos de definir. En el Código Fuente 255 se puede ver una página JSP que utiliza la librería de etiquetas.

```

<html>
<head>
  <title>Etiquetas personalizadas</title>

```

```

</head>
<body>
<%@taglib uri="libreriaEtiquetas.tld" prefix="libreria"%>
<h1>Información Buffer</h1>
<br>
<libreria:bufer/>
</body>
</html>

```

Código Fuente 255

Como se puede apreciar se ha elegido utilizar para el espacio de nombres de la librería el prefijo *libreria*, y el atributo *uri* de la directiva *taglib* coincide con la etiqueta *<taglib-uri>* utilizada en el fichero WEB.XML de la aplicación Web.

En la Figura 121 se puede ver el resultado de la ejecución de esta página JSP.

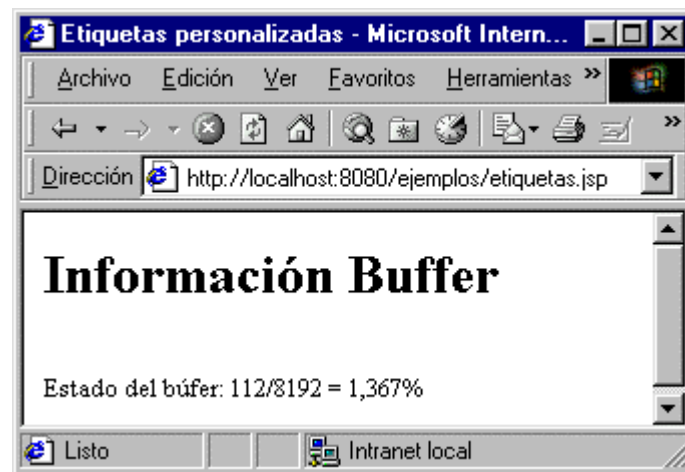


Figura 121. utilizando una etiqueta personalizada

Vamos a ver ahora otro ejemplo más de etiqueta personalizada, vamos a añadir una nueva etiqueta a la librería de etiquetas que ya tenemos definida. En este caso la etiqueta que vamos a desarrollar nos va a devolver el tamaño en kilobytes del fichero que le pasamos como parámetro. Por lo tanto lo novedoso de esta etiqueta es que va a poseer un atributo de sólo escritura que va a ser el nombre del fichero del cual deseamos obtener su tamaño.

Al presentar un atributo esta nueva etiqueta, deberemos definir el atributo dentro de la librería de etiquetas, para ello deberemos hacer uso de la etiqueta *<attribute>*. Nuestro fichero TLD tendría el nuevo aspecto que se muestra en el Código Fuente 256.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>libreria</shortname>
  <info>
    Un ejemplo de librería
  </info>
</taglib>

```



```

</info>

<tag>
  <name>bufer</name>
  <tagclass>EstadoBuffer</tagclass>
  <bodycontent>empty</bodycontent>
  <info>Muestra la utilización del buffer de la página actual</info>
</tag>

<tag>
  <name>tam</name>
  <tagclass>EtiquetaTam</tagclass>
  <info>Muestra el tamaño de un fichero</info>
  <attribute>
    <name>uri</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
</taglib>

```

Código Fuente 256

La clase manejadora de la etiqueta es la clase EtiquetaTam, la etiqueta va a recibir el nombre tam, y posee un atributo obligatorio llamado uri, que va a ser el nombre del fichero del que se va obtener su tamaño.

El código de la clase manejadora de esta etiqueta (Código Fuente 257) es muy similar al del ejemplo anterior, se diferencia en que define un método de acceso al estilo de las propiedades de los componentes JavaBeans, en este caso se trata del método setUri() para establecer el valor del atributo uri de la etiqueta. El método doStartTag() obtiene el camino real al fichero y obtiene el tamaño del mismo.

```

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class EtiquetaTam extends TagSupport{

    private String uri;
    public int doStartTag() throws JspException{

        StringBuffer html=new StringBuffer();
        String caminoReal=pageContext.getServletContext().getRealPath(uri);
        File fich=new File(caminoReal);
        long tam=fich.length()/1024;
        html.append(tam);
        html.append(" KB");
        try{
            pageContext.getOut().write(html.toString());
        }catch (IOException ex){
            throw new JspException(ex.getMessage());
        }
        return SKIP_BODY;
    }
    public void setUri(String valor){
        this.uri=valor;
    }
}

```

Código Fuente 257

Recuerdo al lector que el fichero de clase del manejador de la etiqueta se debe copiar al directorio WEB-INF\CLASSES de la aplicación Web en la que se va a utilizar la etiqueta personalizada.

Para utilizar la nueva etiqueta en la página JSP del ejemplo anterior la podemos escribir directamente, ya que se encuentra en la misma librería de etiquetas, y por lo tanto ya la tenemos disponible a través de la directiva taglib. El Código Fuente 258 muestra la nueva versión de la página JSP que utiliza las dos etiquetas personalizadas.

```
<html>
<head>
  <title>Etiquetas personalizadas</title>
</head>
<body>
  <%@taglib uri="libreriaEtiquetas.tld" prefix="libreria"%>
  El fichero fich1.jpg tiene el tamaño de <libreria:tam uri="fich1.jpg"/><br>
  El fichero fich2.java tiene el tamaño de <libreria:tam uri="fich2.java"/><br>
  <h1>Información Buffer</h1>
  <br>
  <libreria:bufer/>
</body>
</html>
```

Código Fuente 258

En la Figura 122 se puede ver el resultado de la ejecución de la página.

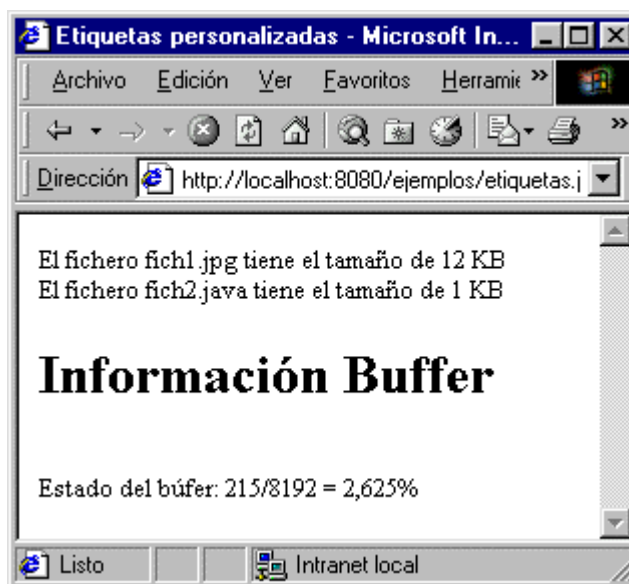


Figura 122. utilizando otra etiqueta personalizada

Con este ejemplo finalizamos el capítulo dedicado a las etiquetas personalizadas, en el siguiente capítulo comentaremos en profundidad la configuración del servidor Web que estamos utilizando como contenedor de servlets y de páginas JSP, es decir, el servidor Jakarta Tomcat. Con este capítulo pretendemos ofrecer al lector una referencia acerca de las distintas etiquetas y ficheros de configuración de Tomcat. También se comentará la estructura de directorios que presenta el servidor.

# Jakarta Tomcat

---

## Introducción

En este capítulo vamos a comentar la instalación y configuración del servidor Web Jakarta Tomcat. A lo largo del texto hemos ido comentando en diversos apartados la utilización de Jakarta Tomcat para poder probar los ejemplos del texto, el presente capítulo pretende aglutinar todo lo visto acerca de Jakarta Tomcat ampliándolo en algunos aspectos.

Hemos utilizado Jakarta Tomcat 3.2 como servidor Web independiente y como contenedor de servlets y de páginas JSP, es decir Jakarta Tomcat nos ha ofrecido la implementación de la especificación Java servlet 2.2 y de la especificación JavaServer Pages 1.1.

Existen otros servidores Web y motores que implementan las especificaciones servlet 2.2 y JSP 1.1, aunque el más extendido, que es reconocido como la implementación oficial, es el servidor Web Jakarta Tomcat 3.1.

## Instalación de Jakarta Tomcat

El servidor Web Jakarta Tomcat se puede utilizar como un servidor Web de prueba para ejecutar servlets y de páginas JSP (JavaServer Pages), es decir, además de implementar la especificación Java servlet 2.2 implementa la especificación JavaServer Pages 1.1. Jakarta Tomcat también puede utilizarse como un añadido para el servidor Apache, es decir, permite que el servidor Web Apache pueda ejecutar servlets y páginas JSP. Esta última utilización es la que se le debería dar a Jakarta Tomcat en un entorno de producción real.

Para poder seguir los ejemplos de este texto nosotros hemos utilizado Jakarta Tomcat como un servidor Web completo, es decir, como un servidor Web que soporta servlets y no como un módulo añadido a un servidor Web existente. Además Jakarta Tomcat está preparado para poder ejecutarse sin problemas en una máquina en la que ya se encuentre instalado un servidor Web, ya que por defecto el servidor Web Jakarta Tomcat utiliza el puerto 8080 para el protocolo HTTP, en lugar del estándar, es decir, el puerto 80.

Jakarta Tomcat es un servidor Web gratuito ofrecido por Apache, y que podemos obtener en la siguiente dirección <http://jakarta.apache.org>.

Jakarta Tomcat se ofrece en forma binaria, es decir, compilado y listo para utilizar, y también se ofrece el código fuente para que sea el desarrollador quien lo compile. Por sencillez vamos a utilizar la primera opción. Lo que obtenemos del sitio Web de Jakarta Tomcat es un fichero comprimido en formato ZIP (*jakarta-tomcat.zip*), que debemos descomprimir en nuestra máquina.

Al descomprimir el fichero, se nos generará una estructura de directorios que contiene los distintos elementos del servidor Web Jakarta Tomcat, en el apartado correspondiente detallaremos esta estructura de directorios y que funcionalidad tiene cada uno de ellos. El directorio en el que se encuentra toda la estructura de Jakarta Tomcat se llama *jakarta-tomcat*.

La configuración del servidor Web Jakarta Tomcat no resulta nada intuitiva ni fácil de realizar, al igual que ocurre con el servidor Web Apache. En este apartado vamos a comentar únicamente la configuración mínima necesaria que se necesita para poder ejecutar Tomcat.

Dos ficheros que se adjuntan junto Jakarta Tomcat, y a los que hay que prestar una atención especial son: *STARTUP.BAT* y *SHUTDOWN.BAT*. Al ejecutar el primero de ellos iniciaremos la ejecución del servidor Web Tomcat, y al ejecutar el segundo se finalizará la ejecución del servidor Jakarta Tomcat. Estos dos ficheros se encuentran en el directorio *jakarta-tomcat\bin*.

Para poder ejecutar el servidor Web Jakarta Tomcat debemos modificar el fichero de inicio del mismo, es decir, el fichero *STARTUP.BAT*. Esta modificación consiste en añadir una línea que especifique un parámetro que indicará la localización de la implementación de la plataforma Java 2, es decir, la localización del Java 2 SDK (Software Development Kit), que se supone ya tenemos instalado.

En el fichero *STARTUP.BAT* se debe especificar la variable *JAVA\_HOME* después de los comentarios que parecen en el fichero (varias líneas comenzando por *rem*). Tanto si hemos instalado la plataforma J2EE o la plataforma J2SE, deberemos indicar el directorio de instalación del Java 2 SDK Estándar Edition, que por defecto es *c:\jdk1.3\*, de todas formas el lector puede comprobar fácilmente el directorio raíz en el que ha instalado el Java 2 SDKSE. Por lo tanto la línea que debemos añadir en *STARTUP.BAT* es:

```
SET JAVA_HOME=c:\jdk1.3\
```

Un ejemplo de fichero *STARTUP.BAT* completo se ofrece a continuación:

```
@echo off
rem $Id: startup.bat,v 1.7 2000/03/31 19:40:02 craigmcc Exp $
rem Startup batch file for tomcat servner.

rem This batch file written and tested under Windows NT
rem Improvements to this file are welcome

SET JAVA_HOME=C:\jdk1.3\

if not "%TOMCAT_HOME%" == "" goto start
```

```

SET TOMCAT_HOME=.
if exist %TOMCAT_HOME%\bin\tomcat.bat goto start

SET TOMCAT_HOME=..
if exist %TOMCAT_HOME%\bin\tomcat.bat goto start

SET TOMCAT_HOME=
echo Unable to determine the value of TOMCAT_HOME.
goto eof

:start
call %TOMCAT_HOME%\bin\tomcat start %1 %2 %3 %4 %5 %6 %7 %8 %9

:eof

```

Una vez realizado este cambio ya estamos preparados para iniciar la ejecución de Jakarta Tomcat. Para iniciar la ejecución de Jakarta Tomcat simplemente debemos realizar doble clic sobre el fichero STARTUP.BAT, y aparecerá una ventana muy similar a la de la Figura 1.

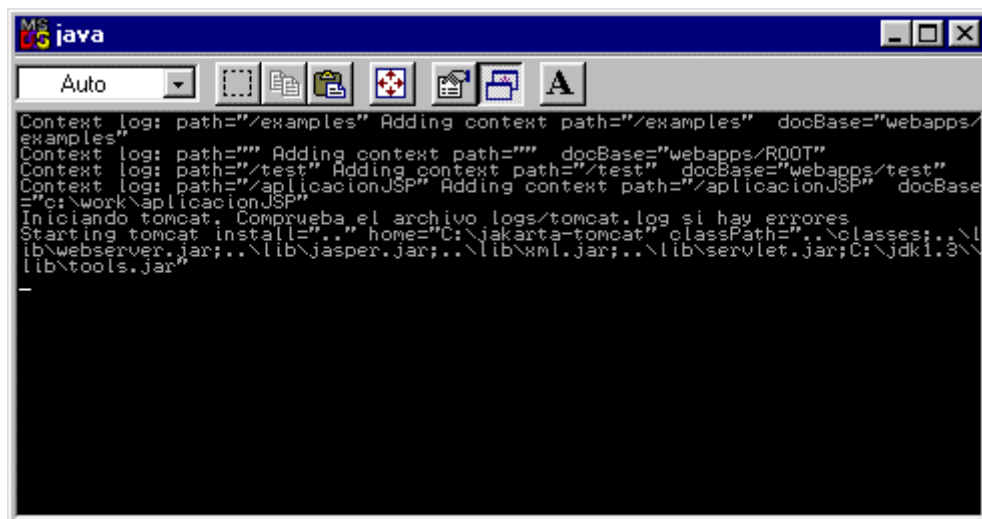


Figura 123. Inicio de Jakarta Tomcat

Para verificar que está todo correcto y que Jakarta Tomcat puede ejecutar servlets y páginas JSP sin ningún problema vamos a realizar una sencilla comprobación. Debemos ejecutar un navegador Web, ya sea Internet Explorer o Netscape Communicator, y debemos especificar la siguiente URL <http://localhost:8080/examples>. Esta URL se corresponde con los ejemplos de servlets y páginas JSP, que se ofrecen en la instalación de Jakarta Tomcat, el aspecto de esta página se puede comprobar en la Figura 2.

Para comprobar el correcto funcionamiento de un servlet en el servidor Tomcat pulsaremos sobre el enlace servlets. Este enlace nos lleva a la página de ejemplos de servlets, en la que al pulsar cada uno Así por ejemplo si pulsamos sobre el primer ejemplo, llamado Hello World, ejecutaremos el servlet HelloWorldExample que nos mostrará la página de la Figura 3.

Si deseamos detener Tomcat no tenemos nada más que ejecutar el fichero SHUTDOWN.BAT. Al ejecutar este fichero se detendrá el servidor Web Jakarta Tomcat y se cerrará la ventana del intérprete de comandos que se había abierto al iniciarse su ejecución.

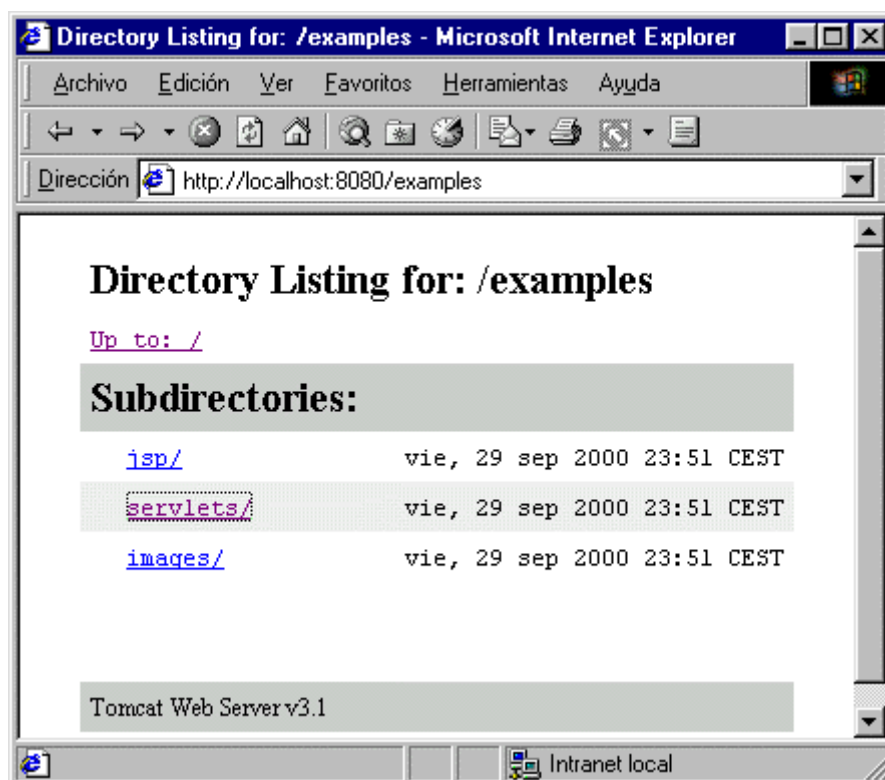


Figura 124. Ejemplos disponibles en Tomcat

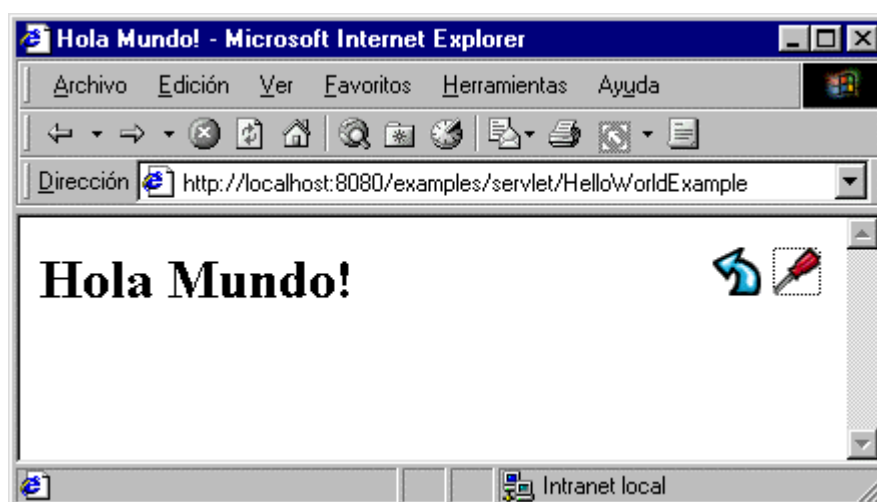


Figura 125. Ejecución de un servlet de ejemplo

## Estructura de directorios de Jakarta Tomcat

Al instalar Jakarta Tomcat, se crea un directorio llamado jakarta-tomcat como directorio raíz de la aplicación, dentro de este directorio encontramos diversos subdirectorios que son los que pasamos a comentar a continuación:

- BIN: contiene los scripts de arranque y parada del servidor Web, es decir, los ficheros por lotes STARTUP.BAT y SHUTDOWN.BAT respectivamente.

- **CONF:** contiene varios ficheros de configuración del servidor, los principales los **SERVER.XML** y **WEB.XML**. El primero de estos ficheros es el fichero de configuración principal del servidor, y el segundo de ellos establece los valores por defecto de las distintas aplicaciones Web del servidor. La estructura de estos dos directorios la comentaremos más adelante.
- **DOC:** contiene la documentación de Jakarta Tomcat.
- **LIB:** contiene varios ficheros JAR de Tomcat. Se ofrece un fichero JAR (Java Archive), llamado **SERVLET.JAR** que contiene los paquetes de la especificación Java servlet 2.2 y de la especificación **JavaServer Pages 1.1**, y también los ficheros **JASPER.JAR** y **WEBSEVER.JAR**, que contienen las clases que forman parte del propio contenedor de servlets y páginas JSP.
- **LOGS:** en este directorio se encuentran los ficheros de registro de Tomcat.
- **SRC:** contiene los ficheros fuente de la especificación Java servlet y de la especificación **JavaServer Pages**.
- **WEBAPPS:** contiene las aplicaciones Web de ejemplo que se distribuyen con Tomcat.
- **WORK:** es el directorio de trabajo de Tomcat, es dónde tiene lugar la traducción de las páginas JSP a los servlets correspondientes. Existirá un subdirectorio del directorio **WORK** por cada aplicación Web, así si nuestra aplicación Web se llama **ejemplos**, su directorio de trabajo será **c:\jakarta-tomcat\work\localhost\_8080\ejemplos**. El directorio de trabajo de la aplicación encontraremos un fichero **.JAVA** y otro fichero **.CLASS** que contienen en su nombre la cadena **paginaJSP**, es decir, el nombre de la página JSP a la que corresponden y a partir de la que se han generado. En este caso han generado los ficheros **\_0002fpaginaJSP\_0002ejsppaginaJSP\_jsp\_0.java** y **\_0002fpaginaJSP\_0002ejsppaginaJSP.class**. Si abrimos el fichero fuente del servlet, podemos ver el código Java que ha generado el contenedor de página JSP para crear el servlet equivalente a la página.

## Ficheros de configuración

Como ya hemos comentado dentro del directorio **CONF** se encuentran los dos ficheros de configuración de Tomcat, que son **SERVER.XML** y **WEB.XML**.

### Fichero **SERVER.XML**

El fichero **SERVER.XML** es el fichero general de configuración del servidor Web, y una de las funciones principales es la de definir las aplicaciones Web del servidor.

Este fichero de configuración está definido en formato XML (Extensive Markup Language), no vamos a entrar a comentar el lenguaje XML, sino que simplemente vamos a comentar y describir las etiquetas que nos permiten definir una nueva aplicación en el servidor Tomcat.

Antes de seguir comentando como definir una aplicación en Tomcat en el fichero **SERVER.XML**, vamos a mostrar en el Código Fuente 12 el contenido del fichero **SERVER.XML** que viene por defecto junto con la instalación del servidor Jakarta Tomcat.

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<Server>
  <!-- Debug low-level events in XmlMapper startup -->
  <xmlmapper:debug level="0" />

  <!-- This is quite flexible; we can either have a log file per
        module in Tomcat (example: ContextManager) or we can have
        one for Servlets and one for Jasper, or we can just have
        one tomcat.log for both Servlet and Jasper.

        If you omit "path" there, then stderr should be used.

        verbosityLevel values can be:
            FATAL
            ERROR
            WARNING
            INFORMATION
            DEBUG
        -->

  <Logger name="tc_log"
    path="logs/tomcat.log"
    customOutput="yes" />

  <Logger name="servlet_log"
    path="logs/servlet.log"
    customOutput="yes" />

  <Logger name="JASPER_LOG"
    path="logs/jasper.log"
    verbosityLevel = "INFORMATION" />

  <!-- Add "home" attribute if you want tomcat to be based on a different
directory
        "home" is used to create work and to read webapps, but not for libs or
CLASSPATH.
        Note that TOMCAT_HOME is where tomcat is installed, while ContextManager
home is the
        base directory for contexts, webapps/ and work/
    -->
  <ContextManager debug="0" workDir="work" >
    <!-- ContextInterceptor className="org.apache.tomcat.context.LogEvents" / -
->
    <ContextInterceptor className="org.apache.tomcat.context.AutoSetup" />
    <ContextInterceptor className="org.apache.tomcat.context.DefaultCMSetter"
/>
    <ContextInterceptor
className="org.apache.tomcat.context.WorkDirInterceptor" />
    <ContextInterceptor className="org.apache.tomcat.context.WebXmlReader" />
    <ContextInterceptor
className="org.apache.tomcat.context.LoadOnStartupInterceptor" />
    <!-- Request processing -->
    <RequestInterceptor className="org.apache.tomcat.request.SimpleMapper"
debug="0" />
    <RequestInterceptor
className="org.apache.tomcat.request.SessionInterceptor" />
    <RequestInterceptor className="org.apache.tomcat.request.SecurityCheck" />
    <RequestInterceptor className="org.apache.tomcat.request.FixHeaders" />

    <Connector className="org.apache.tomcat.service.SimpleTcpConnector">
      <Parameter name="handler"
value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
      <Parameter name="port" value="8080"/>
    </Connector>

```



```

        <Connector className="org.apache.tomcat.service.SimpleTcpConnector">
            <Parameter name="handler"
value="org.apache.tomcat.service.connector.Ajp12ConnectionHandler"/>
            <Parameter name="port" value="8007"/>
        </Connector>

        <!-- example - how to override AutoSetup actions -->
        <Context path="/examples" docBase="webapps/examples" debug="0"
reloadable="true" >
        </Context>
        <!-- example - how to override AutoSetup actions -->
        <Context path="" docBase="webapps/ROOT" debug="0" reloadable="true" >
        </Context>

        <Context path="/test" docBase="webapps/test" debug="0" reloadable="true" >
        </Context>

    </ContextManager>
</Server>

```

Código Fuente 259. Contenido del fichero SERVER.XML

No debemos asustarnos con tanto código XML, por ahora sólo nos vamos a fijar en las etiquetas `<Context>` que aparecen al final del fichero SERVER.XML. Para definir una aplicación dentro del fichero SERVER.XML debemos utilizar la etiqueta `<Context>`, existirán tantas etiquetas `<Context>` como aplicaciones definidas en el servidor Tomcat. Debido a esto, las aplicaciones Web en el entorno de Tomcat también se pueden denominar contextos.

El aspecto de la etiqueta `<Context>` lo podemos ver en el Código Fuente 13 para la aplicación que se ofrece de ejemplo (examples) con el servidor Jakarta Tomcat.

```

<Context path="/examples" docBase="webapps/examples" debug="0" reloadable="true" >

```

Código Fuente 260

Vamos a comentar las propiedades principales que posee la etiqueta `<Context>`:

- **path:** la propiedad `path` indica el nombre de directorio que va a recibir la aplicación en su URL correspondiente, así por ejemplo, para acceder a la aplicación `examples` debemos escribir la siguiente URL en el navegador Web: `http://localhost:8080/examples`.
- **docBase:** la propiedad `docBase` de la etiqueta `<Context>` indica la localización física de los ficheros que forman parte de la aplicación. En este caso los ficheros que forman la aplicación se identifican con un camino relativo, que es un subdirectorio del directorio de instalación de Tomcat y que es dónde por defecto se encuentran todas las aplicaciones Web de Tomcat, este directorio se llama `c:\jakarta-tomcat\webapps`, por lo tanto la ruta física completa del directorio físico de la aplicación `examples` es `c:\jakarta-tomcat\webapps\examples`.
- **debug:** mediante `debug` indicamos el nivel de detalle, de cero a nueve, de los diferentes mensajes de depuración que se registren, cero es el menor nivel de detalle y nueve es el que ofrece mayor información.
- **reloadable:** la propiedad `reloadable` de la etiqueta `<Context>` puede tomar un valor booleano, que indicará si el servidor Tomcat detecta automáticamente las modificaciones que se realicen

en los servlets de la aplicación. Es muy recomendable dar el valor de verdadero (true) a esta propiedad en servidores de desarrollo, ya que de esta forma podremos realizar modificaciones sobre los servlets sin tener que reiniciar el servidor de nuevo. De todas formas esta operación es costosa y como tal requiere un tiempo extra en la ejecución del servidor.

De esta forma, si queremos crear una aplicación nueva llamada ejemplos, cuyos ficheros se encuentran en la ruta física c:\work\ejemplos, y además queremos utilizar el menor detalle en la depuración y que se recarguen los servlets de forma automática al modificase, añadiremos la siguiente etiqueta <Context> (Código Fuente 14) en el fichero SERVER.XML

```
<Context path="/ejemplos" docBase="c:\work\ejemplos" debug="0" reloadable="true" >
</Context>
```

Código Fuente 261

Como ya hemos visto en el código del fichero SERVER.XML, además de la etiqueta <Context>, existen otra serie de etiquetas XML que nos van a permitir configurar nuestro servidor Web.

- <Server>: es la etiqueta de mayor jerarquía y representa al servidor Tomcat, normalmente no debemos realizar ningún tipo de configuración ni tarea con este elemento.
- <Logger>: este elemento define un objeto de registro (logger object). Cada objeto de registro va a poseer un nombre y una ruta en la que se situará el fichero de registro en el que se irá almacenando la información correspondiente. Existen tres objetos de registro, una para los servlets, otro para las páginas JSP y otro para el propio Tomcat. En el Código Fuente 262 se pueden ver los objetos de registro que se definen por defecto.

```
<Logger name="tc_log"
      path="logs/tomcat.log"
      customOutput="yes" />

<Logger name="servlet_log"
      path="logs/servlet.log"
      customOutput="yes" />

<Logger name="JASPER_LOG"
      path="logs/jasper.log"
      verbosityLevel = "INFORMATION" />
```

Código Fuente 262

- <ContextManager>: esta etiqueta define la estructura para un conjunto de otras etiquetas, y permite especificar el directorio de trabajo de Tomcat, así como el nivel empleado en los mensajes de depuración.
- <ContextInterceptor> y <RequestInterceptor>: estas etiquetas denominadas de interceptores, tiene la función de atender a distintos eventos del ContextManager, como puede ser el inicio y finalización del servidor, cuyo encargado es el ContextInterceptors, y las peticiones que se realizan, que son seguidas por el RequestInterceptor. Normalmente no debemos realizar ningún tipo de tarea con estas etiquetas.

- **<Conector>**: esta etiqueta representa una conexión con el usuario, en esta etiqueta podemos especificar el puerto TCP/IP en el que se va a ejecutar el servidor, por defecto es el puerto 8080.

Normalmente la etiqueta que más se suele utilizarse del fichero SERVER.XML es la etiqueta **<Context>**. Pasemos ahora al siguiente fichero de configuración, el fichero WEB.XML.

## Fichero WEB.XML

Existe un fichero WEB.XML dentro del directorio CONF que define una serie de valores por defecto para todas las aplicaciones Web del servidor, pero cada aplicación WEB, en su directorio WEB-INF, puede tener su propio fichero WEB.XML, y así poder sobrescribir estos valores por defecto o añadir nuevos aspectos a la configuración de la aplicación Web correspondiente.

El fichero de configuración WEB.XML posee las siguientes funciones:

- Ofrecer una descripción de la aplicación Web.
- Realizar una correspondencia de nombre de servlets con URLs.
- Establecer la configuración de la sesión.
- Mapeo de librerías de etiquetas.
- Informar de los tipos MIME soportados.
- Establecer la página por defecto de la aplicación Web.
- Definir los parámetros de inicialización de un servlet.

De momento vamos a comentar como definir los parámetros de inicialización de un servlet, pero antes de esto tenemos que ver como asignar un nombre a un servlet, ya que los parámetros de inicialización se le asignan a un servlet a través de su nombre.

Primero debemos asociar un nombre con el fichero de la clase del servlet, una vez hecho esto asociamos el nombre del servlet con el parámetro de inicialización. Todas las etiquetas XML que vamos a utilizar se encuentran englobadas por una etiqueta general llamada **<web-app>** y que contiene toda la configuración para la aplicación Web en la que se encuentra el fichero WEB.XML. La estructura inicial de este fichero es la que se muestra en el Código Fuente 16.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>

</web-app>
```

Código Fuente 263

Dentro de una etiqueta `<servlet>` indicaremos la correspondencia entre el nombre que se le va a dar al servlet y el fichero que contiene la clase del mismo y los parámetros de inicialización que se van a utilizar en el servlet, además de otros aspectos de configuración del servlet. Es precisamente dentro de esta etiqueta dónde podremos indicar si el servlet se debe instanciar cuando se inicie la ejecución del servidor Tomcat.

Mediante la etiqueta `<servlet-name>` indicamos el nombre que se le va a asignar al servlet. El valor que se indique en esta etiqueta puede ser utilizado en la URL que invoca al servlet, así por ejemplo si al servlet `MuestraMensaje` le asignamos el nombre `mensaje`, podremos invocarlo desde el navegador con la URL `http://localhost:8080/ejemplos/servlet/mensaje`. Como se puede comprobar debemos seguir utilizando el directorio `servlet`.

Para indicar el servlet al que vamos a asignar el nombre especificado en la etiqueta `<servlet-name>`, utilizaremos la etiqueta `<servlet-class>`. En esta etiqueta se indica el nombre de la clase del servlet, así si tomamos el ejemplo anterior, deberíamos especificar la clase `MuestraMensaje`.

Tanto la etiqueta `<servlet-name>` como `<servlet-class>` son etiquetas obligatorias que forman parte de la etiqueta `<servlet>`.

De esta forma ya tendríamos un servlet determinado identificado mediante un nombre. En este momento el fichero `WEB.XML` tiene el aspecto del mostrado en el Código Fuente 17.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>
      mensaje
    </servlet-name>
    <servlet-class>
      MuestraMensaje
    </servlet-class>
  </servlet>
</web-app>
```

Código Fuente 264

Para asignar ahora los parámetros de inicialización al servlet hacemos uso de otras etiquetas llamadas `<init-param>`, `<param-name>` y `<param-value>`. La etiqueta `<init-param>` indica que se va a definir un parámetro para el servlet actual, es decir, en servlet que se encuentra definido entre las etiquetas `<servlet>``</servlet>`.

La etiqueta `<init-param>` posee varios subelementos que comentamos a continuación:

- `<param-name>`: etiqueta en la que indicamos el nombre del parámetro de inicialización del servlet. El nombre del parámetro es case-sensitive, es decir, se distingue entre minúsculas y mayúsculas, esta norma la podemos aplicar como general a todos los elementos identificativos de los ficheros XML de configuración del servidor Tomcat y sus aplicaciones.

- `<param-value>`: etiqueta en la que se indica el valor del parámetro. Estos valores siempre se van a recuperar como objeto String, el servlet será encargado de convertir el valor del parámetro al tipo correspondiente.
- `<description>`: permite especificar una descripción del parámetro, esta etiqueta se utiliza a nivel de documentación.

De esta forma si queremos inicializar el servlet asignándole al parámetro repeticiones el valor de 4 y al parámetro mensaje el valor de la cadena “Hola, que tal”, el fichero de configuración de la aplicación Web, tendrá el aspecto del Código Fuente 265.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>
      mensaje
    </servlet-name>
    <servlet-class>
      MuestraMensaje
    </servlet-class>
    <init-param>
      <param-name>repeticiones</param-name>
      <param-value>4</param-value>
      <description>Número de veces que se repite el mensaje</description>
    </init-param>
    <init-param>
      <param-name>mensaje</param-name>
      <param-value>Hola, que tal</param-value>
      <description>Texto del mensaje</description>
    </init-param>
  </servlet>
</web-app>
```

Código Fuente 265

Dentro de la etiqueta `<servlet>` también podemos indicar que el servlet se cree, y ejecute el método `init()`, al iniciar la ejecución del servidor Tomcat. Para ello utilizaremos la etiqueta `<load-on-startup>`, si a este etiqueta no se le facilita ningún valor, el servidor instanciará y cargará el servlet cuando le sea posible al iniciarse. Pero si deseamos que se instancie el servlet atendiendo a una prioridad con respecto al resto de los servlets de la aplicación Web, especificaremos un entero como valor de la etiqueta. Cuanto mayor sea este número menor será la prioridad de creación del servlet, de esta forma primero se crearán los servlets con prioridad 1, luego los de prioridad 2, y así con el resto.

En siguiente aspecto que vamos a comentar del fichero WEB.XML es la forma en la que nos permite la correspondencia de los servlets con patrones de URLs.

La etiqueta `<servlet-mapping>` nos permite asignar a un servlet, identificado por su nombre asignado en la subetiqueta `<servlet-name>` de la etiqueta `<servlet>`, una URL determinada. La etiqueta `<servlet-mapping>` contiene dos subetiquetas `<servlet-name>` y `<url-pattern>`, que pasamos a comentar a continuación:

- `<servlet-name>`: en esta etiqueta indicamos el nombre del servlet al que queremos asociar con una URL determinada, o bien con un patrón de URL, como veremos más adelante.
- `<url-pattern>`: en esta etiqueta especificamos el patrón de la URL que queremos asignar al servlet. El contenido del cuerpo de esta etiqueta puede ser una cadena sencilla que indique una ruta más o menos compleja de la URL que se va a asociar con el servlet correspondiente, o bien una cadena con caracteres especiales que sirvan como un patrón.

Así por ejemplo si queremos invocar al servlet `MuestraMensaje` utilizando la URL `http://localhost:8080/ejemplos/miServlet`, deberemos añadir el Código Fuente 20 en el fichero WEB.XML de la aplicación `ejemplos`, justamente después del cierre de la etiqueta `<servlet>` que habíamos utilizado para definir el nombre del servlet y sus parámetros de inicialización.

```
<servlet-mapping>
  <servlet-name>mensaje</servlet-name>
  <url-pattern>/miServlet</url-pattern>
</servlet-mapping>
```

Código Fuente 266

Como se puede comprobar ya no se utiliza el directorio `servlet` en ningún lugar de la URL que utilizamos para invocar el servlet.

En el ejemplo anterior la etiqueta `<url-pattern>` contenía en su cuerpo únicamente una cadena que identificaba un directorio a partir de la URL de la aplicación actual, pero esto puede ser más complejo teniendo la posibilidad de haber especificado una estructura de directorios más compleja, como podría haber sido `miServlet/primeros/uno`. De esta forma el servlet `MuestraMensaje` podría haber sido invocado mediante la URL `http://localhost:8080/ejemplos/miServlet/primeros/uno`.

También es posible el uso de asteriscos (\*) como caracteres especiales. El asterisco se puede utilizar como un comodín en la parte final de la URL que se va a asignar al servlet, así por ejemplo si en la etiqueta `<url-pattern>`, utilizamos la cadena `miServlet/mensajes/*`, todas las URLs que comiencen de la forma `http://localhost:8080/ejemplos/miServlet/mensajes`, invocarán el servlet `MuestraMensaje`. De esta forma si utilizamos la URL `http://localhost:8080/ejemplos/miServlet/mensajes/la/898989/8898`, invocaremos al ya conocido servlet `MuestraMensaje`.

Otro uso avanzado que podemos realizar de la etiqueta `<url-pattern>` es la de asignar los ficheros con una extensión determinada a un servlet determinado. Para ello en el cuerpo de la etiqueta `<url-pattern>` utilizaremos una cadena compuesta de un asterisco, un punto y el nombre de la extensión de los ficheros que queremos que se correspondan con la invocación del servlet. Así por ejemplo, si queremos que al utilizar una URL con un fichero que posea la extensión `.ser` se invoque al servlet `MuestraMensaje` deberemos añadir el Código Fuente 22 al fichero WEB.XML.

```
<servlet-mapping>
  <servlet-name>mensaje</servlet-name>
  <url-pattern>*.ser</url-pattern>
</servlet-mapping>
```

Código Fuente 267

Este es el mecanismo utilizado para invocar las JavaServer Pages (JSP), todos los ficheros con extensión .jsp se mapean o se corresponden con el servlet que realiza las funciones de compilador de páginas JSP.

Otra funcionalidad del fichero de configuración WEB.XML es la de indicar las librerías de etiquetas personalizadas que se puede utilizar en la aplicación Web actual. Para ellos disponemos de la etiqueta <taglib>.

La etiqueta <taglib> dentro del fichero WEB.XML presenta dos subelementos que nos permiten registrar las librerías de etiquetas personalizadas necesarias en una aplicación Web. Los subelementos que la etiqueta <taglib> presenta son los comentados a continuación:

- <taglib-uri>: esta etiqueta contiene una URL que va a identificar a la librería de etiquetas, no se trata de una URL real, sino la que se va a utilizar en el atributo de la directiva taglib para identificar una librería determinada.
- <taglib-location>: esta etiqueta indica la ruta real dónde se localiza el fichero TLD.

En el Código Fuente 251 se muestra un fragmento del fichero WEB.XML de una aplicación Web que va a utilizar una librería de etiquetas.

```
<taglib>
  <taglib-uri>
    libreriaEtiquetas.tld
  </taglib-uri>
  <taglib-location>
    /WEB-INF/tlds/libreriaEtiquetas.tld
  </taglib-location>
</taglib>
```

Código Fuente 268

Los ficheros TLD se suelen situar en el directorio WEB-INF/TLDS de la aplicación Web, de todas formas, en el siguiente apartado comentaremos la estructura de directorios que suelen presentar las aplicaciones Web dentro de Jakarta Tomcat.

Dentro del fichero WEB.XML podemos mapear extensiones MIME al tipo de fichero correspondiente, en el fichero WEB.XML que se encuentra en el directorio CONF, y que define la configuración por defecto de todas las aplicaciones Web del servidor, se encuentran mapeadas una serie de extensiones, las más comunes que podemos encontrar en el entorno de Internet, a sus tipos de ficheros correspondientes.

En el Código Fuente 269 se puede ver un fragmento del fichero WEB.XML en el que se realiza la correspondencia de la extensión mov con los tipos de fichero quicktime, los tipos MIME definidos son los que va a saber interpretar de manera satisfactoria el servidor Web.

```
<mime-mapping>
  <extension>
    mov
  </extension>
  <mime-type>
    video/quicktime
  </mime-type>
```

```
</mime-mapping>
```

Código Fuente 269

Como se deduce del código anterior, existe una etiqueta principal llamada `<mime-mapping>` que permite realizar la correspondencia del tipo de fichero con la extensión. La extensión se indica con la subetiqueta `<extension>`, y el tipo MIME que se corresponde con la extensión se especifica mediante la subetiqueta `<mime-type>`.

Otro aspecto que nos permite configurar para cada aplicación Web el fichero WEB.XML es la página de inicio de inicio o página por defecto de la aplicación. Esta página será la que se cargará si no especificamos ninguna página concreta de la aplicación Web en la URL, es decir, si utilizamos una dirección del tipo `http://localhost:8080/ejemplos/` se mostrará la página o documento por defecto que hayamos indicado en el fichero WEB.XML.

Para indicar la página o páginas por defecto de la aplicación Web utilizaremos la etiqueta `<welcome-file-list>`. Esta etiqueta permite definir varias páginas por defecto, si no encuentra una de ellas buscará la siguiente el servidor, para definir cada una de las páginas se utiliza la subetiqueta `<welcome-file>`, que contendrá el nombre del fichero en cuestión.

En el Código Fuente 270 se ofrece un fragmento del fichero WEB.XML en el que se define la lista de páginas por defecto de las aplicaciones del servidor.

```
<welcome-file-list>
  <welcome-file>
    index.jsp
  </welcome-file>
  <welcome-file>
    index.html
  </welcome-file>
  <welcome-file>
    index.htm
  </welcome-file>
</welcome-file-list>
```

Código Fuente 270

En el siguiente apartado vamos a comentar la estructura de directorios que presenta una aplicación Web en el servidor Jakarta Tomcat.

## Estructura de directorios de una aplicación Web

Una aplicación Web en Tomcat, quedaba definida a partir del directorio indicado en la etiqueta `<Context>` del fichero SERVER.XML del servidor.

A lo largo del texto hemos ido comentando los distintos directorios que puede presentar una aplicación Web en el servidor Jakarta Tomcat y la finalidad de cada uno de ellos, ahora en este apartado vamos a realizar un resumen de todos ellos.

Vamos a suponer que tenemos definida una aplicación Web, según el Código Fuente 271, llamada `ejemplos`, y a partir de ella vamos a comentar la estructura de directorios que presenta.



```
<Context path="/ejemplos" docBase="c:\work\ejemplos" debug="0" reloadable="true" >  
</Context>
```

Código Fuente 271

En el directorio raíz de la aplicación Web, es decir, el directorio C:\WORK\EJEMPLOS, podemos incluir todos los ficheros HTML, JSP, imágenes, etc., de los que consta la aplicación Web, así como distintos subdirectorios de contenidos que pueden contener más páginas JSP o HTML.

A partir del directorio raíz existe un subdirectorio llamado WEB-INF que contendrá el fichero de configuración de la aplicación Web, es decir, el fichero WEB.XML.

Dentro del directorio WEB-INF existe un subdirectorio llamado CLASSES, que contendrá todas las clases de los servlets y componentes JavaBeans de la aplicación Web actual. El directorio CLASSES puede contener múltiples subdirectorios para organizar las distintas clases, ya sean servlets o Beans, en paquetes.

También dentro del directorio WEB-INF encontremos el subdirectorio TLDS en el que debemos situar los ficheros descriptores de las librerías de etiquetas personalizadas, vistos en el capítulo anterior. Además podemos encontrar también dentro del directorio WEB-INF el subdirectorio LIB, que contendrá ficheros JAR con diversas clases utilizadas en la aplicación Web.

Con este apartado damos por finalizado este capítulo, en el siguiente capítulo compararemos algunos aspectos de la especificación JSP con la tecnología de Microsoft de las páginas activas de servidor, es decir, con ASP (Active Server Pages).



# JSP/ASP, una comparativa

---

## Introducción

En este capítulo vamos a realizar una comparativa entre la especificación JSP y la tecnología de Microsoft Active Server Pages (ASP), este capítulo está dedicado a aquellos lectores que ya conocen las páginas activas de servidor, por lo tanto no vamos a realizar ningún comentario acerca de ASP. Con este capítulo se pretende que los lectores que ya conozcan ASP se sienta más cómodos con JSP y puedan entender algún concepto que no hay quedado lo suficientemente claro.

Aquellos lectores que no conozcan ASP no deben preocuparse, pueden leer este capítulo para ver la comparativa entre las dos tecnologías a nivel de curiosidad, incluso les puede servir también si pretender adentrarse en la tecnología ASP.

Vamos a centrar nuestra comparativa en los objetos integrados de cada una de las tecnologías.

Este capítulo se puede tomar también como una breve orientación de cómo podemos convertir una aplicación Web basada en ASP a una aplicación Web basada en JSP, o viceversa, comentado todo a rasgos muy generales. No pretendemos ofrecer una aquí completa de migración entre las distintas soluciones de páginas de servidor, sólo se pretende dar una orientación, y sobretodo facilitar a los programadores de páginas ASP su paso a la programación de páginas JSP.

En los siguientes apartados vamos a comentar las similitudes y diferencias entre los distintos objetos integrados de ASP y JSP. Para cada objeto integrado se va a utilizar una tabla, en la primera columna se describe el aspecto que se está comparando del objeto, en la segunda columna aparece lo aplicable a ASP y en la tercera lo que se correspondería en JSP. En algunos casos no es aplicable alguno de los aspectos a las dos tecnologías.

## Objeto application

Pasemos en un primer lugar con la Tabla 22 que va comparar el objeto application de la especificación JSP con el objeto integrado Application del modelo de objetos de ASP. El objeto application de JSP representa a la aplicación Web (contexto) en la que se encuentra una aplicación JSP, es una instancia del interfaz `javax.servlet.ServletContext`, y por lo tanto posee la misma funcionalidad que es la de permitir comunicarlos con el contenedor de páginas JSP, y permitir mantener un estado más general que el de la sesión para que se comparta entre todos los usuarios de la aplicación Web.

Funcionalidad	ASP	JSP
Nombre del objeto	Application	application
Clase del objeto	N/A (no aplicable)	<code>javax.servlet.ServletContext</code>
Almacenamiento de una variable	<code>Application(nombreVar)="valor"</code>	<code>setAttribute(String nombreVar, Object objeto)</code>
Almacenamiento de un objeto	<code>Set Application(nombreVar) = Server.CreateObject("ProgID")</code>	Igual que en el caso anterior.
Obteniendo una variable	<code>variable=Application(nombreVar)</code>	<code>getAttribute(String nombreVar)</code>
Obteniendo un objeto	<code>Set objeto = Application (nombreVar)</code>	Igual que en el caso anterior
Eliminando un objeto o variable	<code>Contents.Remove(nombre)</code>	<code>removeAttribute(String nombre)</code>
Colección de contenidos	Contents	<code>getAttributeNames()</code>
Bloqueo y desbloqueo	Mediante los métodos Lock y UnLock	Se deben implementar mecanismos de sincronización.
Determinado el tipo de contenedor utilizado	N/A	<code>getServerInfo()</code>
Determinando los números de versión del API servlet	N/A	<code>getMajorVersion()</code> y <code>getMinorVersion()</code>
Determinado el tipo MIME de un fichero	N/A	<code>getMimeType(String fichero)</code>
Escribiendo en el fichero de registro del contenedor	N/A <code>(Response.AppendToLog(mensaje))</code>	<code>log(String mensaje)</code>
Obteniendo el camino real de un fichero	N/A <code>(Server.MapPath(ruta))</code>	<code>getRealPath(String rutaVirtual)</code>

Obteniendo la URL de un recurso	N/A	getResource(String ruta)
---------------------------------	-----	--------------------------

Tabla 22. Objeto application

## Objeto config

En la Tabla 23 se ofrece la comparativa correspondiente del objeto config de JSP, pero en realidad este objeto no se puede comparar con ningún objeto de ASP, ya que no hay un objeto equivalente. Este objeto integrado es una instancia del interfaz `javax.servlet.ServletConfig`, y su función es la de almacenar información de configuración del servlet generado a partir de la página JSP correspondiente. Normalmente las páginas JSP no suelen interactuar con parámetros de inicialización del servlet generado, por lo tanto el objeto config en la práctica no se suele utilizar. El objeto config pertenece a la categoría de objetos integrados relacionados con los servlets.

Funcionalidad	ASP	JSP
Nombre del objeto	ASP no posee un objeto similar	config
Clase del objeto	N/A	<code>javax.servlet.ServletConfig</code>
Nombre del servlet	N/A	<code>getServletName()</code>
Referencia al contexto del servlet	N/A	<code>getServletContext()</code>
Devuelve los nombre de los parámetros de inicialización del servlet	N/A	<code>getInitParametesNames()</code>
Obtener el valor de un parámetro de inicialización	N/A	<code>getInitParamater(String nombreParam)</code>

Tabla 23. El objeto config

## Objeto exception

Pasamos ahora en la Tabla 24 a comparar los objetos encargados del tratamiento de errores, en el caso de ASP se trata del objeto `ASPError`, y en el caso de JSP del objeto `exception`. El objeto `exception` es una instancia de la clase `java.lang.Throwable`, representa una excepción lanzada en tiempo de ejecución.

Este objeto únicamente se encuentra disponible en las páginas de error (indicado mediante la directiva `page`), por lo tanto no se encuentra disponible de forma automática en cualquier página JSP, sólo en aquellas que sean utilizadas para el tratamiento de errores. Este es un objeto que no pertenece a ninguna categoría de objetos integrados, sino que el mismo es un tipo de objeto para el tratamiento de errores.

Funcionalidad	ASP	JSP
Nombre del objeto	ASPError	exception
Clase del objeto	N/A	java.lang.Throwable
Notas de interés	Objeto nuevo de ASP 3.0, se llama al método Server.GetLastError para obtener un referencia al mismo	Sólo se puede acceder a este objeto si la página se ha declarado mediante al directiva page como página de error
Mensaje de error	Description()	getMessage()
Traza de errores	N/A	printStackTrace(salida)
Error completo	ASPDescription()	toString()
Posición del error	Line y Column	N/A

Tabla 24. El objeto exception y el objeto ASPError

## Objeto out

A continuación en la Tabla 25 se compara el objeto Response de ASP con el objeto out de JSP. Este objeto de JSP representa el flujo de salida que se envía al cliente y que forma parte del cuerpo de la respuesta HTTP, esta salida utiliza un búfer intermedio que podemos activar o desactivar e indicar su tamaño utilizando la directiva page. Este objeto pertenece a la categoría de los objetos integrados de entrada/salida, y en este caso se trata de un objeto que representa la salida enviada en el cuerpo de la respuesta enviada al cliente.

Funcionalidad	ASP	JSP
Nombre del objeto	Response	out
Clase del objeto	N/A	javax.servlet.jsp.JspWriter
Escribiendo datos en el búfer de salida	Write datos	print(datos)
Vaciando el búfer	Clear	clearBuffer()
Enviando el búfer actual al cliente	Flush	flush()
Finaliza el proceso de la página	End	close(), se comporta de forma diferente a End, ya que se puede seguir con el procesamiento de la página aunque se haya cerrado el búfer de salida.

Tabla 25. El objeto out y el objeto Response

## Objeto page

La Tabla 26 muestra otro caso de objeto de la especificación JSP que no tiene correspondencia en ASP, se trata del objeto page. El objeto page es una instancia de la clase `java.lang.Object`, y representa la página JSP actual, o para ser más exactos, una instancia de la clase del servlet generado a partir de la página JSP actual.

Funcionalidad	ASP	JSP
Nombre del objeto	ASP no presenta un objeto similar	page
Clase del objeto	N/A	<code>java.lang.Object</code>
Notas		No se suele utilizar directamente

Tabla 26. El objeto page

## Objeto pageContext

Un objeto más de JSP, en la Tabla 27, que no se corresponde con ningún otro de ASP, se trata del objeto pageContext. El objeto pageContext encapsula el contexto para una página JSP particular, a través de este objeto tenemos acceso a el resto de los objetos implícitos de JSP. Este objeto pertenece a la categoría de objetos que representan un contexto, en este caso el contexto de una página JSP.

Funcionalidad	ASP	JSP
Nombre del objeto	ASP no tiene un objeto equivalente	pageContext
Clase del objeto	N/A	<code>javax.servlet.jsp.PageContext</code>

Tabla 27. El objeto pageContext

## Objeto request

En la Tabla 28 se puede observar la comparativa entre los objetos request. Este objeto representa una petición realizada a una página JSP y es una instancia del interfaz `javax.servlet.http.HttpServletRequest`. Este objeto cumple con el cometido del interfaz `HttpServletRequest`, que si recordamos era el de permitir obtener la información que envía el usuario al realizar una petición de un servlet (en este caso a una página JSP), esta información puede ser muy variada, puede ser desde encabezados de petición del protocolo HTTP, cookies enviadas por el usuario o los datos de un formulario. Este objeto entra en la categoría de los objetos que realizan funciones de entrada/salida, en este caso funciones de entrada.

Funcionalidad	ASP	JSP
Nombre del objeto	Response	response

Clase del objeto	N/A	javax.servlet.http.HttpServletRequest
Detalles de un certificado	ClientCertificate(Clave[Campo])	N/A
Detalles de una cookie	Cookies(cookie)[(clave).atributo]	getCookies()
Obteniendo datos del formulario	Form(campo)	getParameter(campo)
Obteniendo datos de la cadena de consulta.	QueryString(campo)	getQueryString(), devuelve la cadena de consulta completa
Cabeceras HTTP enviadas por el cliente	ServerVaRiables	getHeaderNames()

Tabla 28. El objeto request

## Objeto response

En la Tabla 29 se muestra la comparativa entre los objetos response. El objeto response es una instancia del interfaz javax.servlet.http.HttpServletResponse y encapsula la respuesta que le es enviada al usuario como el resultado de la petición de una página JSP. El objeto response ofrece la funcionalidad el interfaz HttpServletResponse, que era la siguiente: el interfaz HttpServletResponse ofrece funcionalidad específica para que los servlets HTTP puedan generar un respuesta del protocolo HTTP para el cliente que realizó la petición del servlet. En nuestro caso, se debe sustituir servlet por página JSP, aunque realmente es finalmente un servlet quién realiza la tarea indicada en la página JSP.

Funcionalidad	ASP	JSP
Nombre del objeto	Response	response
Clase del objeto	N/A	javax.servlet.jsp.HttpServletResponse
Búfer de salida de la página	Buffer	<%@page buffer%>
Caché del servidor proxy	CacheControl	setHeader("Cache-Control","no-cache")
Añadir cookies	Cookies(cookie)=valor	addCookie(cookie)
Añadir cabeceras HTTP	AddHeader nombre,valor	setHeader(nombre,valor)
Redirección del cliente	Redirect URL	sendRedirect(URL)



Enviar un error al cliente	N/A	sendError(int codigo, String mensaje)
Codificar una URL	N/A (Server.URLEncode)	encodeURL(URL)
Establecer el tipo MIME de salida	ContentType="Tipo MIME"	SetContentType("Tipo MIME")

Tabla 29. El objeto response

## Objeto Server de ASP

A continuación en la Tabla 30 se ofrece la comparativa del objeto Server de ASP con JSP, que no presenta un objeto similar, pero si las funcionalidades del mismo, distribuidas entre diferentes objetos.

Funcionalidad	ASP	JSP
Nombre del objeto	Server	N/A
Clase del objeto	N/A	N/A
Creación de un objeto en el servidor	CreateObject(ProgID)	Sintaxis estándar de Java para al creación de objetos
Codificar una cadena HTML	HTMLEncode(cadena)	N/A
Rutas reales de ficheros	MapPath(ruta)	application.getRealPath(ruta)
Pasar el control a otra página	Transfer	Acción forward
Codificar una URL	URLEncode(cadena)	response.encodeURL(cadena)
Tiempo de espera para un script.	ScriptTimeout	N/A

Tabla 30. El objeto Server

## El objeto session

En la siguiente tabla, la Tabla 31, se muestra la relación entre el objeto session de JSP y el Session de ASP. Este objeto integrado de JSP representa una sesión de un usuario con nuestra aplicación Web, y es una instancia del interfaz `javax.servlet.http.HttpSession`. Las sesiones son creadas de forma automática, a no ser que se indique lo contrario dentro de la directiva `page` de la página JSP. El objeto session ofrece toda la funcionalidad que ofrecía el interfaz `HttpSession`, que recordamos era: mantener a través de nuestros servlets información a lo largo de una sesión del cliente con nuestra aplicación Web. Cada cliente tendrá su propia sesión, y dentro de la sesión podremos almacenar cualquier tipo de objeto.

Funcionalidad	ASP	JSP
Nombre del objeto	Session	session
Nota de interés	Utiliza cookies para mantener la sesión	Puede utilizar cookies y reescritura de URLs
Dstrucción de la sesión	Abandon	invalidate()
Almacenado una variable de sesión	Session(nombreVar)=valor	setAttribute(String nombre, Object objeto)
Almacenando un objeto se sesión	Set Session(nombreObj)= Server.CreateObject(ProgID)	Igual que en el caso anterior.
Obteniendo una variable de sesión	var=Session(nombreVar)	getAttribute(String nombre)
Obteniendo un objeto de la sesión	Set objr=Session(nombreObj)	Igual que el caso anterior
Eliminando una variable se sesión	Contents.Remove(nombre)	removeAttribute(String nombre)
Colección de contenidos	Contents	getAttributeNames()
El identificador de la sesión	SessionID	GetId()
Estableciendo el tiempo máximo de espera.	Timeout=Minutos	setMaxInactiveInterval(int segundos)
Deshabilitando la sesión	<%@EnableSessionState=False%>	<%@page session="false"%>

Tabla 31. El objeto session

Con este objeto finalizamos la comparativa de JSP y ASP, y también el presente texto.

## Bibliografía

---

- Web Development with JavaServer Pages. Duane K. Fields, Mark A. Kolb. Manning 2000.
- Professional JSP. VV. AA. Wrox 2000.
- Core Servlets and JavaServer Pages. Marty Hall. Prentice Hall 2000.





Si quiere ver más textos en este formato, visítenos en: <http://www.lalibreriadigital.com>.

Este libro tiene soporte de formación virtual a través de Internet, con un profesor a su disposición, tutorías, exámenes y un completo plan formativo con otros textos. Si desea inscribirse en alguno de nuestros cursos o más información visite nuestro campus virtual en: <http://www.almagesto.com>.

Si quiere información más precisa de las nuevas técnicas de programación puede suscribirse gratuitamente a nuestra revista *Algoritmo* en: <http://www.algoritmodigital.com>. No deje de visitar nuestra revista *Alquimia* en <http://www.eidos.es/alquimia> donde podrá encontrar artículos sobre tecnologías de la sociedad del conocimiento.

Si quiere hacer algún comentario, sugerencia, o tiene cualquier tipo de problema, envíelo a la dirección de correo electrónico [lalibreriadigital@eidos.es](mailto:lalibreriadigital@eidos.es).

---

© Grupo EIDOS

<http://www.eidos.es>

