

Servlets y JavaServer Pages (JSP) 1.0

Indice de Contenidos Servlets

- [Introducción a los Servlets y JSP](#)
 - ¿Qué es un Servlet Java?
 - ¿Qué ventajas tienen los Servlets Java frente al CGI tradicional?
 - ¿Qué es JSP?
 - ¿Cuáles son las Ventajas de JSP?
- [Instalación y Configuración](#)
 - Obtener e Instalar las clases Servlet y JSP
 - Obtener un Servidor Web Compatible con Servlets o Motor de Servlets
- [Primeros Servlets](#)
 - Estructura Básica de un Servlet
 - Un Sencillo Servlet que Genera Texto Normal
 - Compilar e Invocar al Servlet
 - Un Sencillo Servlet que Genera HTML
 - Algunas utilidades HTML sencillas para Servlets
- [Procesar la Solicitud: Formulario de Datos](#)
 - Introducción (Format, URL-encoding, GET, POST)
 - Ejemplo: Leer Parámetros Específicos
 - Ejemplo: Hacer una Tabla de todos los Parámetros
- [Procesar la Solicitud: Cabeceras de Solicitud HTTP](#)
 - Cabeceras de Solicitud Comunes y sus Significados
 - Leer Cabeceras desde Servlets
 - Ejemplo: Hacer una Tabla con todas las Cabeceras
- [Acceder a Variables CGI Estándards](#)
 - Las Variables CGI, sus Significados y sus Equivalentes Servlet
 - Ejemplo: Hacer una Tabla con todas las Variables CGI
- [Generar la Respuesta: Códigos de Estado HTTP](#)
 - Introducción: Códigos de Estado y Mensajes
 - Seleccionar Códigos de Estado desde Servlets
 - Códigos de Estado HTTP 1.1 y sus Significados

- Ejemplo: Un Motor de Búsqueda
- [Generar la Respuesta: Cabeceras de Respuesta HTTP](#)
 - Introducción
 - Cabeceras de Respuesta Comunes y sus Significados
 - Ejemplo: Arrancar un cálculo de larga duración, mostrar resultados parciales y periódicamente mostrar actualizaciones con nuevos datos
- [Manejar Cookies](#)
 - Introducción: Como usar (y abusar de) las cookies
 - El API Servlet Cookie
 - Algunas utilidades de Cookies
 - Ejemplo: Un Interface de Motor de Búsqueda Personalizado
- [Seguimiento de Sesión](#)
 - Introducción al Seguimiento de Sesión
 - El API Session Tracking
 - Ejemplo
- [JavaServer Pages \(JSP\)](#)
 - Introducción
 - Sumario de Sintaxis
 - Plantilla de Texto (HTML Dinámico)
 - Elementos de Script JSP: Expresiones, Scriptlets, y Declaraciones
 - Directivas JSP
 - Ejemplo de uso de Elementos Script y Directivas
 - Variables Predefinidas
 - Acciones JSP
 - Convenciones de Comentarios y Caracteres de Escape JSP

Indice de Contenidos JSP

- [Introducción a JSP](#)
- [Directivas JSP](#)
- [El Principio](#)
- [Manejar Formularios HTML](#)
- [Usar Elementos de Script](#)
- [Manejar Excepciones](#)

Introducción

1. [¿Qué son los Servlets Java?](#)
 2. [¿Qué Ventajas tienen los Servlets Frente al CGI "Tradicional"?](#)
 3. [¿Qué es JSP?](#)
 4. [¿Qué Ventajas Tiene JSP?](#)
-

1. ¿Qué son los Servlets Java?

Los Servlets son la respuesta de la tecnología Java a la programación CGI. Son programas que se ejecutan en un servidor Web y construyen páginas Web. Construir páginas Web al vuelo es útil (y comunmente usado) por un número de razones:

- La página Web está basada en datos enviados por el usuario. Por ejemplo, las páginas de resultados de los motores de búsqueda se generan de esta forma, y los programas que procesan pedidos desde sites de comercio electrónico también.
- Los datos cambian frecuentemente. Por ejemplo, un informe sobre el tiempo o páginas de cabeceras de noticias podrían construir la página dinámicamente, quizás devolviendo una página previamente construida y luego actualizándola.
- Las páginas Web que usan información desde bases de datos corporativas u otras fuentes. Por ejemplo, usaríamos esto para hacer una página Web en una tienda on-line que liste los precios actuales y el número de artículos en stock.

2. ¿Cuáles son las Ventajas de los Servlets sobre el CGI "Tradicional"?

Los Servlets Java son más eficientes, fáciles de usar, más poderosos, más portables, y más baratos que el CGI tradicional y otras muchas tecnologías del tipo CGI. (y lo que es más importante, los desarrolladores de servlets cobran más que los programadores de Perl :-).

- Eficiencia.
Con CGI tradicional, se arranca un nuevo proceso para cada solicitud HTTP. Si

el programa CGI hace una operación relativamente rápida, la sobrecarga del proceso de arrancada puede dominar el tiempo de ejecución. Con los Servlets, la máquina Virtual Java permanece arrancada, y cada petición es manejada por un thread Java de peso ligero, no un pesado proceso del sistema operativo. De forma similar, en CGI tradicional, si hay N peticiones simultáneas para el mismo programa CGI, el código de este problema se cargará N veces en memoria. Sin embargo, con los Servlets, hay N threads pero sólo una copia de la clase Servlet. Los Servlet también tienen más alternativas que los programas normales CGI para optimizaciones como los cachés de cálculos previos, mantener abiertas las conexiones de bases de datos, etc.

- **Conveniencia.**

Hey, tu ya sabes Java. ¿Por qué aprender Perl? Junto con la conveniencia de poder utilizar un lenguaje familiar, los Servlets tienen una gran infraestructura para análisis automático y decodificación de datos de formularios HTML, leer y seleccionar cabeceras HTTP, manejar cookies, seguimiento de sesiones, y muchas otras utilidades.

- **Potencia.**

Los Servlets Java nos permiten fácilmente hacer muchas cosas que son difíciles o imposibles con CGI normal. Por algo, los servlets pueden hablar directamente con el servidor Web. Esto simplifica las operaciones que se necesitan para buscar imágenes y otros datos almacenados en situaciones estándares. Los Servlets también pueden compartir los datos entre ellos, haciendo las cosas útiles como almacenes de conexiones a bases de datos fáciles de implementar. También pueden mantener información de solicitud en solicitud, simplificando cosas como seguimiento de sesión y el caché de cálculos anteriores.

- **Portable.**

Los Servlets están escritos en Java y siguen un API bien estandarizado. Consecuentemente, los servlets escritos, digamos en el servidor I-Planet Enterprise, se pueden ejecutar sin modificarse en Apache, Microsoft IIS, o WebStar. Los Servlets están soportados directamente o mediante plug-in en la mayoría de los servidores Web.

- **Barato.**

Hay un número de servidores Web gratuitos o muy baratos que son buenos para el uso "personal" o el uso en sites Web de bajo nivel. Sin embargo, con la excepción de Apache, que es gratuito, la mayoría de los servidores Web comerciales son relativamente caros. Una vez que tengamos un servidor Web, no importa el coste del servidor, añadirle soporte para Servlets (si no viene preconfigurado para soportarlos) es gratuito o muy barato.

3. ¿Qué es JSP?

Java Server Pages (JSP) es una tecnología que nos permite mezclar HTML estático con HTML generado dinámicamente. Muchas páginas Web que están construidas con programas CGI son casi estáticas, con la parte dinámica limitada a muy pocas localizaciones. Pero muchas variaciones CGI, incluyendo los servlets, hacen que generemos la página completa mediante nuestro programa, incluso aunque la mayoría de ella sea siempre lo mismo. JSP nos permite crear dos partes de forma separada. Aquí tenemos un ejemplo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Welcome to Our Store</TITLE></HEAD>
<BODY>
<H1>Welcome to Our Store</H1>
<SMALL>Welcome,
<!-- User name is "New User" for first-time visitors -->
<% out.println(Utils.getUserNameFromCookie(request)); %>
To access your account settings, click
<A HREF="Account-Settings.html">here.</A></SMALL>
<P>
Regular HTML for all the rest of the on-line store's Web page.
</BODY></HTML>
```

4. ¿Cuáles son las Ventajas de JSP?

- **Contra Active Server Pages (ASP).**
ASP es una tecnología similar de Microsoft. Las ventajas de JSP están duplicadas. Primero, la parte dinámica está escrita en Java, no en Visual Basic, otro lenguaje específico de MS, por eso es mucho más poderosa y fácil de usar. Segundo, es portable a otros sistemas operativos y servidores Web
- **Contra los Servlets.**
JSP no nos da nada que no pudieramos en principio hacer con un servlet. Pero es mucho más conveniente escribir (y modificar!) HTML normal que tener que hacer un billón de sentencias println que generen HTML. Además, separando el formato del contenido podemos poner diferentes personas en diferentes tareas: nuestros expertos en diseño de páginas Web pueden construir el HTML, dejando espacio para que nuestros programadores de servlets inserten el contenido dinámico.
- **Contra Server-Side Includes (SSI).**
SSI es una tecnología ampliamente soportada que incluye piezas definidas externamente dentro de una página Web estática. JSP es mejor porque nos permite usar servlets en vez de un programa separado para generar las

partes dinámicas. Además, SSI, realmente está diseñado para inclusiones sencillas, no para programas "reales" que usen formularios de datos, hagan conexiones a bases de datos, etc.

- **Contra JavaScript.**

JavaScript puede generar HTML dinámicamente en el cliente. Esta es una capacidad útil, pero sólo maneja situaciones donde la información dinámica está basada en el entorno del cliente. Con la excepción de las cookies, el HTTP y el envío de formularios no están disponibles con JavaScript. Y, como se ejecuta en el cliente, JavaScript no puede acceder a los recursos en el lado del servidor, como bases de datos, catálogos, información de precios, etc.

Instalación y Configuración del Servidor

1. [Obtener e Instalar los Kits de Desarrollo de Servlets y JSP](#)
2. [Instalar un Servidor Web compatible con Servlets](#)

1. Obtener e Instalar los Kits de Desarrollo de Servlets y JSP

Nuestro primer paso es descargar el software que implementa las especificaciones Java Servlet 2.1 o 2.2 y Java Server Pages 1.0 ó 1.1. Podemos obtener una versión gratuita de Sun, conocida como "JavaServer Web Development Kit" (JSWDK), en <http://java.sun.com/products/servlet/>.

Luego, necesitamos decirle a javac dónde encontrar las clases Servlets y JSP cuando compilemos nuestro fichero servlet. Las instrucciones de instalación del JSWDK explican esto, pero básicamente apuntan a poner los ficheros servlet.jar y jsp.jar (que vienen con el JSWDK) en nuestro CLASSPATH. Si nunca antes has tratado con el CLASSPATH, es la variable de entorno que especifica donde Java busca la clases. Si no es especificada, Java busca en el directorio actual y en las librerías estándar del sistema. Si la seleccionamos nosotros mismos necesitamos estar seguros de incluir ".", que significa el directorio actual. Aquí tenemos un rápido resumen de cómo seleccionarla en un par de plataformas:

Unix (C Shell)

```
setenv CLASSPATH .:servlet_dir/servlet.jar:servlet_dir/jsp.jar
```

Añadimos ":\$CLASSPATH" al final de la línea setenv si nuestro CLASSPATH ya está configurado, y queremos añadirle más directorios, no reemplazarlo. Observa que se usan dos puntos ":" para separar directorios, mientras que Windows usa puntos y coma. Para hacer permanente esta configuración ponemos esta sentencia dentro de nuestro fichero .cshrc.

Windows 95/98/NT

```
set CLASSPATH=.;servlet_dir/servlet.jar;servlet_dir/jsp.jar
```

Añadimos `";% CLASSPATH% "` al final de la línea anterior si nuestro CLASSPATH ya está configurado. Observa que usamos puntos y coma ";" para separar directorios, mientras que en Unix se usan dos puntos. Para hacer permanente esta configuración ponemos esta sentencia en el fichero autoexec.bat. En Windows NT, vamos al menú Start, seleccionamos Settings y luego Control Panel, seleccionamos System, y Environment y luego introducimos la variable y el valor.

Finalmente, como veremos en [La siguiente sección](#), queremos poner nuestros servlets en paquetes para evitar conflictos de nombres con los servlets escritos por otras personas para la misma aplicación Web o servidor. En este caso, podríamos encontrar conveniente añadir el directorio de más alto nivel de nuestro paquete al CLASSPATH. Puedes ver la sección [Primeros Servlets](#) para más detalles.

2. Instalar un servidor Web con Capacidad para Servlets

Nuestro siguiente paso es obtener e instalar un servidor Web que soporte servlets Java, o instalar el paquete Servlet en nuestro servidor Web existente. Si estamos usando un servidor Web actualizado, hay muchas posibilidades de que ya tengamos todo lo que necesitamos. Debemos chequear la documentación de nuestro servidor o ver la última lista de servidores que soportan servlets en <http://java.sun.com/products/servlet/industry.html>. Aunque eventualmente queramos desarrollar en un servidor de calidad comercial, cuando estamos aprendiendo es útil tener un sistema gratuito que podemos instalar en nuestra máquina para propósitos de desarrollo y prueba. Aquí están algunas de las opciones más populares:

- [Apache Tomcat.](#)

Tomcat es la implementación de referencia oficial para las especificaciones Servlet 2.2 y JSP 1.1. Puede ser usado como pequeño servidor para probar páginas JSP y servlets, o puede integrarse en el servidor Web Apache. Tomcat, al igual que el propio Apache es gratuito. Sin embargo, también al igual que Apache (que es muy rápido, de gran rendimiento, pero un poco difícil de configurar e instalar), Tomcat requiere significativamente más esfuerzo para configurarlo que los motores de servlets comerciales. Para más detalles puedes ver <http://jakarta.apache.org/>.

- [JavaServer Web Development Kit \(JSWDK\).](#)

El JSWDK es la implementación de referencia oficial para las especificaciones Servlet 2.1 y JSP 1.0. Se usaba como pequeño servidor para probar servlets y páginas JSP antes de desarrollar un completo servidor Web que soporta estas tecnologías. Es gratuito y potente, pero necesita un poco de esfuerzo para instalarlo y configurarlo. Para más detalles puedes ver <http://java.sun.com/products/servlet/download.html>.

- [Allaire JRun.](#)

JRun es un motor servlet y JSP que puede conectarse dentro de los servidores Netscape Enterprise o FastTrack, de los servidores Web IIS, Microsoft Personal Web Server, viejas versiones de Apache, O'Reilly's WebSite, o StarNine WebSTAR. Hay una versión gratuita limitada a cinco conexiones simultáneas; la versión comercial elimina esta restricción y añade capacidades como una consola de administración remota. Para más detalles puedes ver <http://www.allaire.com/products/jrun/>.

- [New Atlanta's ServletExec.](#)

ServletExec es un rápido motor de servlets y páginas JSP que puede ser conectado dentro de los servidores web más populares para Solaris, Windows, MacOS, HP-UX y Linux. Podemos descargarlo y usarlo de forma gratuita, pero muchas de sus características avanzadas y utilidades de administración estarán desactivadas hasta que compremos una licencia. El nuevo Atlanta también proporciona un depurador de servlets gratuito que funciona con mucho de los IDEs más populares de Java. Para más detalles puedes ver <http://newatlanta.com/>.

- [Gefion's LiteWebServer \(LWS\).](#)

LWS es un pequeño servidor Web gratuito que soporta Servlets versión 2.2 y JSP 1.1. También tienen un plug-in gratuito llamado [WAICoolRunner](#) que añade soporte para Servlets 2.2 y JSP 1.1 a los servidores Netscape FastTrack y Enterprise. También venden un gran número de Servlets personalizados y componentes JSP en un paquete llamado [InstantOnline](#). Para más detalles puedes ver <http://www.gefionsoftware.com/>.

- [Sun's Java Web Server.](#)

Este servidor está escrito enteramente en Java y fue uno de los primeros servidores en soportar completamente las especificaciones servlet 2.1 y JSP 1.0. Aunque ya no estará bajo desarrollo activo porque Sun se está concentrando en el servidor Netscape/I-Planet, todavía es una opción popular para aprender servlets y JSP. Para un versión de prueba gratuita puedes ver <http://www.sun.com/software/jwebserver/try/>. Para obtener una versión gratuita que no expira para propósitos de enseñanza en instituciones académicas, puedes ver <http://freeware.thesphere.com/>.

Primeros Servlets

1. [Estructura Básica de un Servlet](#)
 2. [Un sencillo Servlet que Genera Texto Normal](#)
 3. [Un Servlet que Genera HTML](#)
 4. [Utilidades de Construcción de HTML Sencillo](#)
-

1. Estructura Básica de un Servlet

Aquí tenemos un servlet básico que maneja peticiones GET. Las peticiones GET, para aquellos que no estemos familiarizados con HTTP, son peticiones hechas por el navegador cuando el usuario teclea una URL en la línea de direcciones, sigue un enlace desde una página Web, o rellena un formulario que no especifica un METHOD. Los Servlets también pueden manejar peticiones POST muy fácilmente, que son generadas cuando alguien crea un formulario HTML que especifica METHOD= "POST". Los discutiremos en una sección posterior.

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

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

        // Use "request" to read incoming HTTP headers (e.g. cookies)
        // and HTML form data (e.g. data the user entered and submitted)

        // Use "response" to specify the HTTP response line and headers
        // (e.g. specifying the content type, setting cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser
    }
}
```

(Descarga [la plantilla de código fuente](#) -- pulsa con el botón derecho del ratón sobre el enlace o mantén pulsada la tecla SHIFT mientras pulsas sobre el enlace).

Para ser un servlet, una clase debería extender [HttpServlet](#) y sobrescribir doGet o doPost (o ambos), dependiendo de si los datos están siendo enviados mediante GET o

POST. Estos métodos toman dos argumentos: un [HttpServletRequest](#) y un [HttpServletResponse](#).

El `HttpServletRequest` tiene métodos que nos permiten encontrar información entrante como datos de un FORM, cabeceras de petición HTTP, etc. El `HttpServletResponse` tiene métodos que nos permiten especificar líneas de respuesta HTTP (200, 404, etc.), cabeceras de respuesta (Content-Type, Set-Cookie, etc.), y, todavía más importante, nos permiten obtener un [PrintWriter](#) usado para enviar la salida de vuelta al cliente. Para servlets sencillos, la mayoría del esfuerzo se gasta en sentencias `println` que generan la página deseada. Observamos que `doGet` y `doPost` lanzan dos excepciones, por eso es necesario incluirlas en la declaración. También observamos que tenemos que importar las clases de los paquetes `java.io` (para `PrintWriter`, etc.), `javax.servlet` (para `HttpServletRequest`, etc.), y `javax.servlet.http` (para `HttpServletRequest` y `HttpServletResponse`). Finalmente, observamos que `doGet` y `doPost` son llamados por el método `service`, y algunas veces queremos sobrescribir directamente el método `service`, por ejemplo, para un servlet que maneje tanto peticiones GET como POST.

2. Un Sencillo Servlet que Genera Texto Normal

Aquí tenemos un servlet que sólo genera texto normal. La siguiente sección mostrará el caso más usual donde se generará HTML.

2.1 HelloWorld.java

También puedes descargar el [código fuente](#)

```
package hall;

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

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```

2.2 Compilar e Instalar el Servlet

Debemos observar que los detalles específicos para instalar servlets varían de servidor en servidor. Los ejemplos se han probado sobre Java Web Server (JWS) 2.0, donde se espera que los servlets estén en un directorio llamado `servlets` en el árbol de instalación del JWS. Sin embargo, hemos situado este servlet en un paquete separado (`hall`) para evitar conflictos con otros servlets del servidor; querrás hacer lo mismo si usas un

servidor Web que es usado por otras personas y no tiene buena estructura para "servlets virtuales" para evitar automáticamente estos conflictos. Así, HelloWorld.java realmente va en un subdirectorio llamado hall en el directorio servlets. Observa que la configuración de la mayoría de los servidores, y los ejemplos de este tutorial también se han probado usando BEA WebLogic e IBM WebSphere 3.0. WebSphere tiene un excelente mecanismo para servlets virtuales, y no es necesario usar paquetes, sólo para evitar conflictos de nombres con otros usuarios.

Una forma de configurar nuestro CLASSPATH es apuntar al directorio superior al que contiene realmente nuestros servlets. Entonces podemos compilar normalmente desde dentro del directorio. Por ejemplo, si nuestro directorio base es C:\JavaWebServer\servlets y el nombre de nuestro paquete es (y por lo tanto el del subdirectorio) es hall, y trabajamos bajo Windows, deberíamos hacer:

```
DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer\servlets\hall
DOS> javac YourServlet.java
```

La primea parte, configura el CLASSPATH, probablemente querremos hacerlo permanentemente, en vez de hacerlo cada que arrancamos una nueva ventana del DOS. En Windows 95/98 pondremos la sentencia "set CLASSPATH=..." en algún lugar de nuestro fichero autoexec.bat después de la línea que selecciona nuestro CLASSPATH para apuntar a servlet.jar y jsp.jar.

Una segunda forma de compilar las clases que están en paquetes es ir al directorio superior del que contiene los Servlets, y luego hacer "javac **directory**\YourServlet.java". Por ejemplo, supongamos de nuevo que nuestro directorio base es C:\JavaWebServer\servlets y que el nombre de nuestro paquete (y del directorio) es hall, y que estamos trabajando en Windows. En este caso, haremos los siguiente:

```
DOS> cd C:\JavaWebServer\servlets
DOS> javac hall\YourServlet.java
```

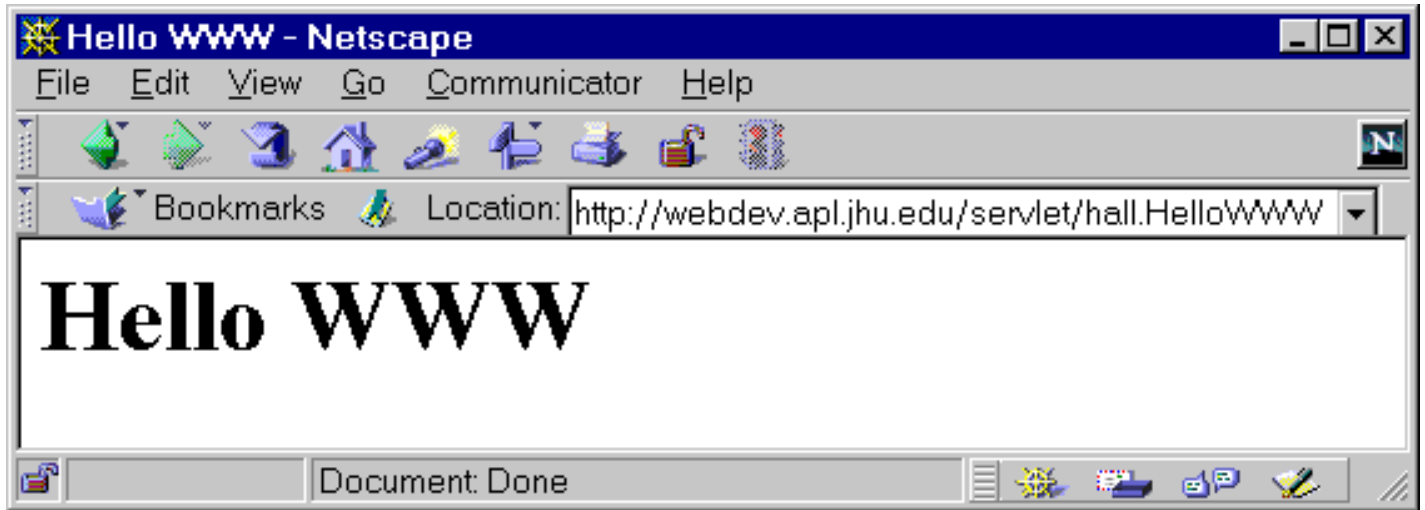
Finalmente otra opción avanzada es mantener el código fuente en una localización distinta de los ficheros .class. y usar la opción "-d" de javac para instalarlos en la localización que espera el servidor Web.

2.3 Ejecutar el Servlet

Con el Java Web Server, los servlets se sitúan en el directorio servlets dentro del directorio principal de la instalación del JWS, y son invocados mediante http://host/servlet/ServletName. Observa que el directorio es servlets, plural, mientras que la referencia URL es servlet, singular. Como este ejemplo se situó en el paquete hall, sería invocado mediante http://host/servlet/hall.HelloWorld. Otros servidores podrían tener convenciones diferentes sobre donde instalar los servlets y como invocarlos. La mayoría de los servidores nos permiten definir alias para nuestros servlets, para que un servlet pueda ser invocado mediante http://host/any-path/any-file.html.

}

3.2 Resultado de HelloWWW



4. Utilidades de Construcción de HTML Sencillo

Es un poco aburrido generar HTML con sentencias `println`. La solución real es usar Java Server Pages (JSP), que se describen [más adelante](#). Sin embargo, para Servlets estándares, hay dos partes de la página Web que no cambian (DOCTYPE y HEAD) y que podría beneficiarnos el incluirlas en un fichero de utilidades.

La línea DOCTYPE es técnicamente requerida por la especificación HTML, y aunque la mayoría de los navegadores Web la ignoran, es muy útil cuando se envían páginas a validadores de formato HTML. Estos validadores comparan la sintaxis HTML de las páginas comparándolas con la especificación formal del HTML, y usan la línea DOCTYPE para determinar la versión de HTML con la que comparar.

En muchas páginas web, la línea HEAD no contiene nada más que el TITLE, aunque los desarrolladores avanzados podrían querer incluir etiquetas META y hojas de estilo. Pero para el caso sencillo, crearemos un método que crea un título y devuelve las entradas DOCTYPE, HEAD, y TITLE como salida. Aquí está el código:

4.1 ServletUtilities.java (Descarga [el código fuente](#))

```
package hall;

public class ServletUtilities {
    public static final String DOCTYPE =
        "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">";

    public static String headWithTitle(String title) {
        return(DOCTYPE + "\n" +
            "<HTML>\n" +
            "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
    }
}
```

```
}

// Other utilities will be shown later...
}
```

4.2 HelloWWW2.java (Descarga [el código fuente](#))

Aquí tenemos una nueva versión de la clase HelloWWW que usa esto.

```
package hall;

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

public class HelloWWW2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println(ServletUtilities.headWithTitle("Hello WWW") +
                    "<BODY>\n" +
                    "<H1>Hello WWW</H1>\n" +
                    "</BODY></HTML>");
    }
}
```

Manejar Datos de Formularios

1. [Introducción](#)
2. [Ejemplo: Leer Tres Parámetros](#)
3. [Ejemplo: Listar Todos los Datos del Formulario](#)

1. Introducción

Si alguna vez has usado un motor de búsqueda Web, visitado un tienda de libros on-line, etc., probablemente habrás encontrado URLs de búsqueda divertidas como `http://host/path?user=Marty+Hall&origin=bwi&dest=lax`. La parte posterior a la interrogación (`user=Marty+Hall&origin=bwi&dest=lax`) es conocida como datos de formulario, y es la forma más común de obtener datos desde una página Web para un programa del lado del servidor. Puede añadirse al final de la URL después de la interrogación (como arriba) para peticiones GET o enviada al servidor en una línea separada, para peticiones POST.

Extraer la información necesaria desde estos datos de formulario es tradicionalmente una de las partes más tediosas de la programación CGI.

Primero de todo, tenemos que leer los datos de una forma para las peticiones GET (en CGI tradicional, esto se hace mediante `QUERY_STRING`), y de otra forma para peticiones POST (normalmente leyendo la entrada estándar).

Segundo, tenemos que separar las parejas de los ampersands, luego separar los nombres de los parámetros (a la izquierda de los signos igual) del valor del parámetro (a la derecha de los signos igual).

Tercero, tenemos que decodificar los valores. Los valores alfanuméricos no cambian, pero los espacios son convertidos a signos más y otros caracteres se convierten como `%XX` donde `XX` es el valor ASCII (o ISO Latin-1) del carácter, en hexadecimal. Por ejemplo, si alguien introduce un valor de "`~hall, ~gates, y ~mcnealy`" en un campo de texto con el nombre "users" en un formulario HTML, los datos serían enviados como "`users= % 7Ehall% 2C+ % 7Egates% 2C+ and+ % 7Emcnealy`".

Finalmente, la cuarta razón que hace que el análisis de los datos de formulario sea tedioso es que los valores pueden ser omitidos (por ejemplo, `param1=val1¶m2= ¶m3=val3`) y un parámetro puede tener más de un valor y que el mismo parámetro puede aparecer más de una vez (por ejemplo: `param1=val1¶m2=val2¶m1=val3`).

Una de las mejores características de los servlets Java es que todos estos análisis de formularios son manejados automáticamente. Simplemente llamamos al método `getParameter` de `HttpServletRequest`, y suministramos el nombre del parámetro como un argumento. Observa que los nombres de parámetros son sensibles a la mayúsculas. Hacemos esto exactamente igual que cuando los datos son enviados mediante GET o como si los enviáramos mediante POST. El valor de retorno es un `String` correspondiente al valor uudecode de la primera ocurrencia del parámetro. Se devuelve un `String` vacío si el parámetro existe pero no tiene valor, y se devuelve `null` si no existe dicho parámetro. Si el parámetro pudiera tener más de un valor, como en el ejemplo anterior, deberíamos llamar a `getParameterValues` en vez de a `getParameter`. Este devuelve un array de strings. Finalmente, aunque en aplicaciones reales nuestros servlets probablemente tengan un conjunto específico de nombres de parámetros por los que buscar. Usamos `getParameterNames` para esto, que devuelve una `Enumeration`, cada entrada puede ser forzada a `String` y usada en una llamada a `getParameter`.

2. Ejemplo: Leer Tres Parámetros

Aquí hay un sencillo ejemplo que lee tres parámetros llamados `param1`, `param2`, y `param3`, listando sus valores en una lista marcada. Observamos que, aunque tenemos que especificar selecciones de respuesta (content type, status line, otras cabeceras HTTP) antes de empezar a generar el contenido, no es necesario que leamos los parámetros de petición en un orden particular.

También observamos que podemos crear fácilmente servlets que puedan manejar datos GET y POST, simplemente haciendo que su método `doPost` llame a `doGet` o sobrescribiendo `service` (que llama a `doGet`,

doPost, doHead, etc.). Esta es una buena práctica estándar, ya que requiere muy poco trabajo extra y permite flexibilidad en el lado del cliente. Si hemos usado la aproximación CGI tradicional cuando leemos los datos POST mediante la entrada estándar. Deberíamos observar que hay una forma similar con los Servlets llamando primero a `getReader` o `getInputStream` sobre `HttpServletRequest`. Esto es una mala idea para parámetros normales, pero podría usarse para ficheros descargados o datos POST que están siendo enviados por clientes personales en vez de formularios HTML. Observa, sin embargo, que si leemos los datos POST de esta manera, podrían no ser encontrados por `getParameter`.

2.1 ThreeParams.java

También puedes [descargar el código fuente](#). Nota: también usa [ServletUtilities.java](#), mostrado anteriormente.

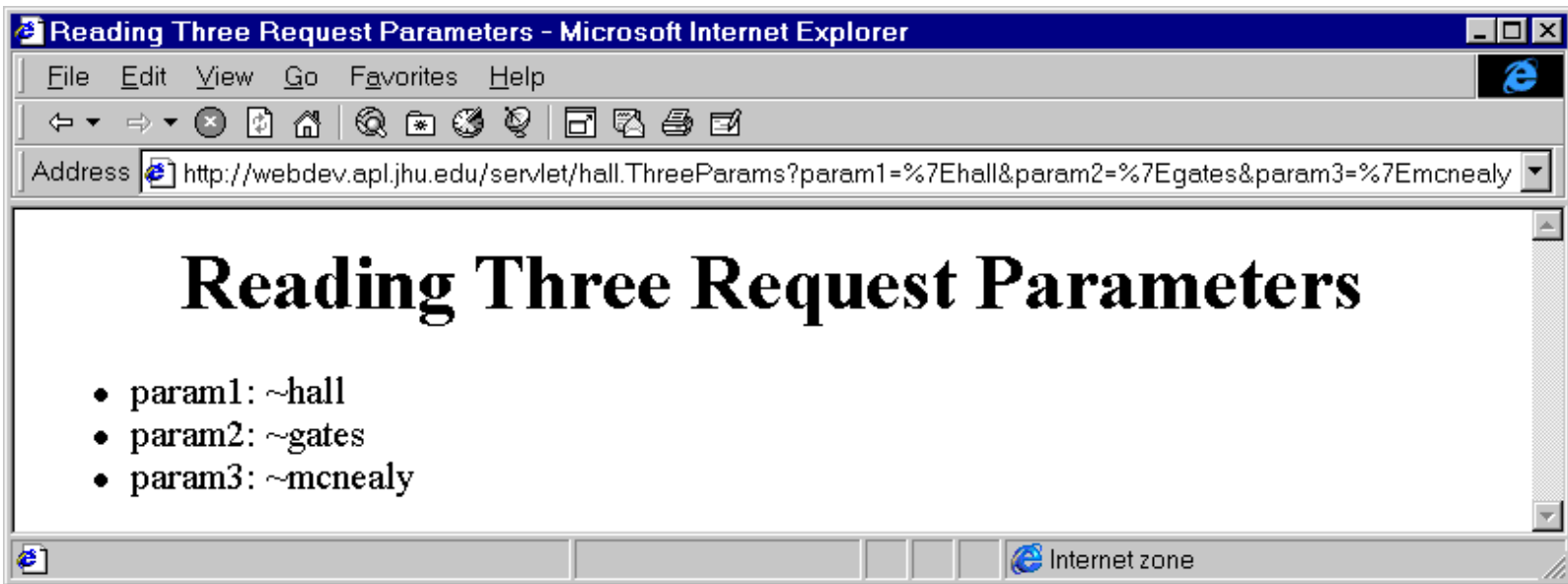
```
package hall;

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

public class ThreeParams extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY>\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<UL>\n" +
            "  <LI>param1: "
            + request.getParameter("param1") + "\n" +
            "  <LI>param2: "
            + request.getParameter("param2") + "\n" +
            "  <LI>param3: "
            + request.getParameter("param3") + "\n" +
            "</UL>\n" +
            "</BODY></HTML>" );
    }

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

2.2 Salida de ThreeParams



3. Ejemplo: Listar todos los Datos del Formulario

Aquí hay un ejemplo que busca todos los nombres de parámetros que fueron enviados y los pone en una tabla. Ilumina los parámetros que tienen valor cero así como aquellos que tienen múltiples valores.

Primero busca todos los nombres de parámetros mediante el método `getParameterNames` de `HttpServletRequest`. Esto devuelve una `Enumeration`. Luego, pasa por la `Enumeration` de la forma estándar, usando `hasMoreElements` para determinar cuando parar y usando `nextElement` para obtener cada entrada. Como `nextElement` devuelve un `Object`, fuerza el resultado a `String` y los pasa a `getParameterValues`, obteniendo un array de `Strings`. Si este array sólo tiene una entrada y sólo contiene un string vacío, el parámetro no tiene valores, y el servlet genera una entrada "No Value" en *itálica*. Si el array tiene más de una entrada, el parámetro tiene múltiples valores, y se muestran en una lista bulleteada. De otra forma, el único valor principal se sitúa en la tabla.

3.1 ShowParameters.java

También puedes [descargar el código fuente](#). Nota: también usa [ServletUtilities.java](#), mostrado anteriormente.

```
package hall;

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

/** Shows all the parameters sent to the servlet via either
 *  GET or POST. Specially marks parameters that have no values or
 *  multiple values.
 *
 *  Part of tutorial on servlets and JSP that appears at
 *  http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
 *  1999 Marty Hall; may be freely used or adapted.
 */

public class ShowParameters extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading All Request Parameters";
```

```

out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +
            "<TR BGCOLOR=\"#FFAD00\">\n" +
            "<TH>Parameter Name<TH>Parameter Value(s)");
Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) {
    String paramName = (String)paramNames.nextElement();
    out.println("<TR><TD>" + paramName + "\n<TD>");
    String[] paramValues = request.getParameterValues(paramName);
    if (paramValues.length == 1) {
        String paramValue = paramValues[0];
        if (paramValue.length() == 0)
            out.print("<I>No Value</I>");
        else
            out.print(paramValue);
    } else {
        out.println("<UL>");
        for(int i=0; i<paramValues.length; i++) {
            out.println("<LI>" + paramValues[i]);
        }
        out.println("</UL>");
    }
}
out.println("</TABLE>\n</BODY></HTML>");
}

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

```

3.2 ShowParameters

Aquí tenemos un formulario HTML que envía un número de parámetros a este servlet. Pulsa con el botón derecho sobre el [enlace al código fuente](#) para descargar el HTML.

Usa POST para enviar los datos (como deberían hacerlo todos los formularios que tienen entradas PASSWORD), demostrando el valor de que los servlets incluyan tanto doGet como doPost.

PostForm.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>A Sample FORM using POST</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>

<FORM ACTION="/servlet/hall.ShowParameters"
      METHOD="POST">
    Item Number:
    <INPUT TYPE="TEXT" NAME="itemNum"><BR>
    Quantity:
    <INPUT TYPE="TEXT" NAME="quantity"><BR>
    Price Each:
    <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>

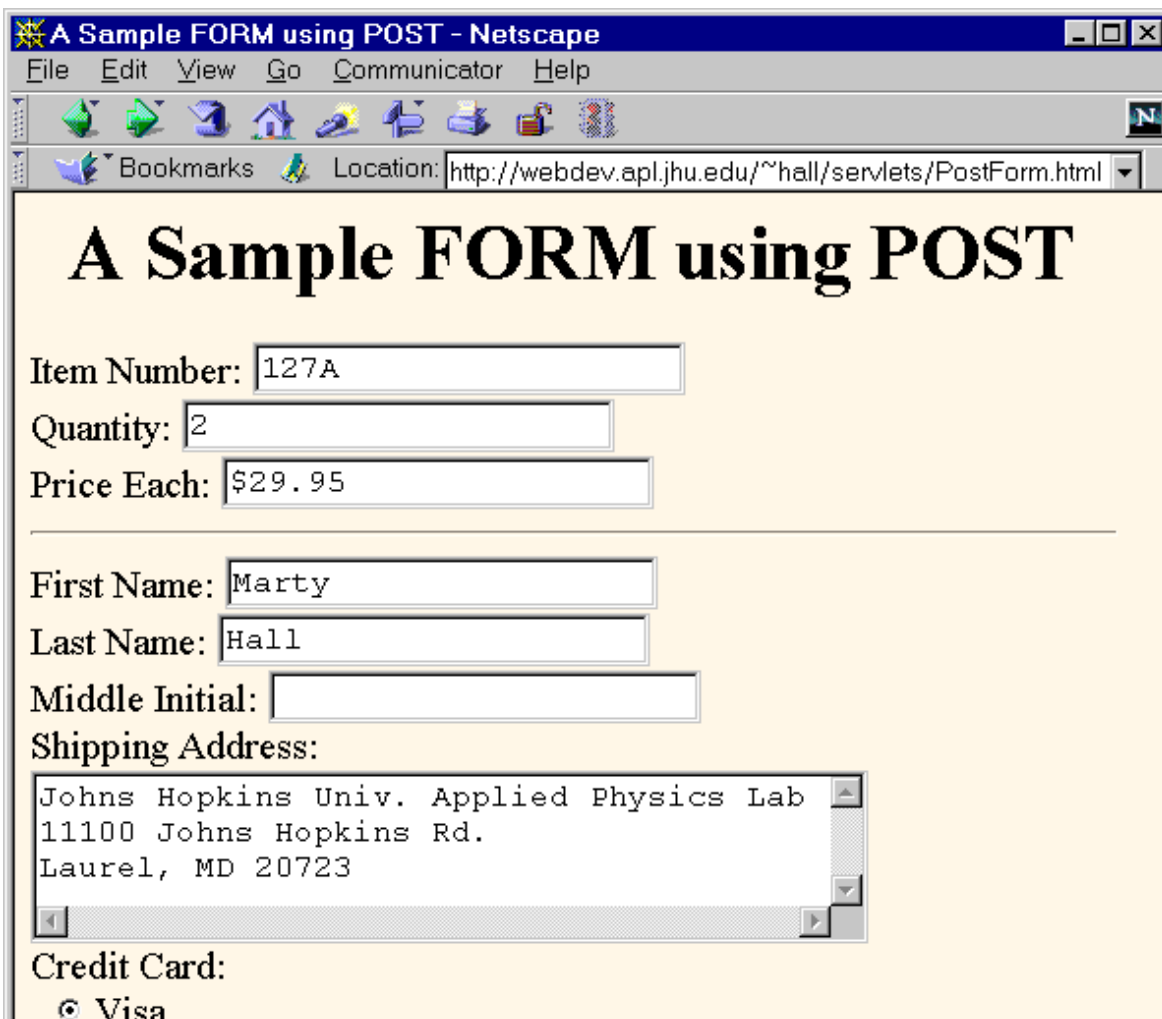
```

```

<HR>
First Name:
<INPUT TYPE="TEXT" NAME="firstName"><BR>
Last Name:
<INPUT TYPE="TEXT" NAME="lastName"><BR>
Middle Initial:
<INPUT TYPE="TEXT" NAME="initial"><BR>
Shipping Address:
<TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
Credit Card:<BR>
  <INPUT TYPE="RADIO" NAME="cardType"
    VALUE="Visa">Visa<BR>
  <INPUT TYPE="RADIO" NAME="cardType"
    VALUE="Master Card">Master Card<BR>
  <INPUT TYPE="RADIO" NAME="cardType"
    VALUE="Amex">American Express<BR>
  <INPUT TYPE="RADIO" NAME="cardType"
    VALUE="Discover">Discover<BR>
  <INPUT TYPE="RADIO" NAME="cardType"
    VALUE="Java SmartCard">Java SmartCard<BR>
Credit Card Number:
<INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
Repeat Credit Card Number:
<INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
<CENTER>
  <INPUT TYPE="SUBMIT" VALUE="Submit Order">
</CENTER>
</FORM>

</BODY>
</HTML>

```



A Sample FORM using POST

Item Number:

Quantity:

Price Each:

First Name:

Last Name:

Middle Initial:

Shipping Address:

Credit Card:

- ☐ Master Card
- ☐ American Express
- ☐ Discover
- ☐ Java SmartCard

Credit Card Number:

Repeat Credit Card Number:

Document: Done

3.3 Resultados del envío

Reading All Request Parameters - Netscape

File Edit View Go Communicator Help

Bookmarks Location: <http://webdev.apl.jhu.edu/servlet/hall.ShowParameters>

Reading All Request Parameters

Parameter Name	Parameter Value(s)
address	Johns Hopkins Univ. Applied Physics Lab 11100 Johns Hopkins Rd. Laurel, MD 20723
initial	<i>No Value</i>
price	\$29.95
cardNum	<ul style="list-style-type: none"> 3.14159 3.14159
firstName	Marty
itemNum	127A
cardType	Visa
quantity	2
lastName	Hall

Document: Done

Leer Cabeceras de Solicitud HTTP

1. [Introducción a las Cabeceras de Solicitud](#)
 2. [Leer Cabeceras de Solicitud desde Servlets](#)
 3. [Ejemplo: Imprimir todas la Cabeceras](#)
-

1. Introducción a las Cabeceras de Solicitud

Cuando un cliente HTTP (por ejemplo, un navegador) envía una petición, se pide que suministre una línea de petición (normalmente GET o POST). Si se quiere también puede enviar un número de cabeceras, que son opcionales excepto Content-Length, que es requerida sólo para peticiones POST. Aquí tenemos las cabeceras más comunes:

- Accept Los tipos MIME que prefiere el navegador.
- Accept-Charset El conjunto de caracteres que espera el navegador.
- Accept-Encoding Los tipos de codificación de datos (como gzip) para que el navegador sepa como decodificarlos. Los servlets pueden chequear explícitamente el soporte para gzip y devolver páginas HTML comprimidas con gzip para navegadores que las soportan, seleccionando la cabecera de respuesta Content-Encoding para indicar que están comprimidas con gzip. En muchos casos, esto puede reducir el tiempo de descarga por un factor de cinco o diez.
- Accept-Language El idioma que está esperando el navegador, en el caso de que el servidor tenga versiones en más de un idioma.
- Authorization Información de autorización, usualmente en respuesta a una cabecera WWW-Authenticate desde el servidor.
- Connection ¿Usamos conexiones persistentes? Sí un servlet obtiene un valor Keep-Alive aquí, u obtiene una línea de petición indicando HTTP 1.1 (donde las conexiones persistentes son por defecto), podría ser posible tomar ventaja de las conexiones persistentes, ahorrando un tiempo significativo para las páginas Web que incluyen muchas piezas pequeñas (imágenes o clases de applets). Para hacer esto, necesita enviar una cabecera Content-Length en la respuesta, que es fácilmente conseguido escribiendo en un ByteArrayOutputStream, y preguntando por el tamaño antes de escribir la salida.
- Content-Length (para mensajes POST, cuántos datos se han añadido)
- Cookie (una de las cabeceras más importantes, puedes ver la sección independiente de esta tutorial dedicada a los [Cookies](#)) .
- From (dirección email del peticionarios; sólo usado por aceleradores Web, no por clientes personalizados ni por navegadores)
- Host (host y puerto escuchado en la URL original)

- If-Modified-Since (sólo devuelve documentos más nuevos que éste, de otra forma se envía una respuesta 304 "Not Modified" response)
- Pragma (el valor no-cache indica que el servidor debería devolver un documento nuevo, incluso si es un proxy con una copia local)
- Referer (la URL de la página que contiene el enlace que el usuario siguió para obtener la página actual)
- User-Agent (tipo de navegador, útil si el servlets está devolviendo contenido específico para un navegador)
- UA-Pixels, UA-Color, UA-OS, UA-CPU (cabeceras no estándar enviadas por algunas versiones de Internet Explorer, indicando el tamaño de la pantalla, la profundidad del color, el sistema operativo, y el tipo de CPU usada por el sistema del navegador)

Para ver todos los detalles sobre las cabeceras HTTP, puedes ver las especificaciones en <http://www.w3.org/Protocols/>.

2. Leer Cabeceras de Solicitud desde Servlets

Leer cabeceras es muy sencillo, sólo llamamos al método `getHeader` de `HttpServletRequest`, que devuelve un `String` si se suministró la cabecera en esta petición, y `null` si no se suministró. Sin embargo, hay un par de cabeceras que se usan de forma tan común que tienen métodos de acceso especiales. El método `getCookies` devuelve el contenido de la cabecera `Cookie`, lo analiza y lo almacena en un array de objetos `Cookie`. Los métodos `getAuthType` y `getRemoteUser` dividen la cabecera `Authorization` en sus componentes. Los métodos `getDateHeader` y `getIntHeader` leen la cabecera específica y la convierten a valores `Date` e `int`, respectivamente.

En vez de buscar una cabecera particular, podemos usar el `getHeaderNames` para obtener una `Enumeration` de todos los nombres de cabecera de esta petición particular.

Finalmente, además de buscar las cabeceras de petición, podemos obtener información sobre la propia línea de petición principal. El método `getMethod` devuelve el método de petición principal (normalmente `GET` o `POST`, pero son posibles cosas como `HEAD`, `PUT`, y `DELETE`). El método `getRequestURI` devuelve la URI (la parte de la URL que viene después del host y el puerto, pero antes de los datos del formulario). El `getRequestProtocol` devuelve la tercera parte de la línea de petición que generalmente es `"HTTP/1.0"` o `"HTTP/1.1"`.

3. Ejemplo: Imprimir todas las Cabeceras

Aquí tenemos un servlet que simplemente crea una tabla con todas las cabeceras recibidas, junto con sus valores asociados. También imprime los tres componentes de la línea de petición principal (método, URI y protocolo).

3.1 ShowRequestHeaders.java

También puedes [descargar el código fuente](#)

```
package hall;

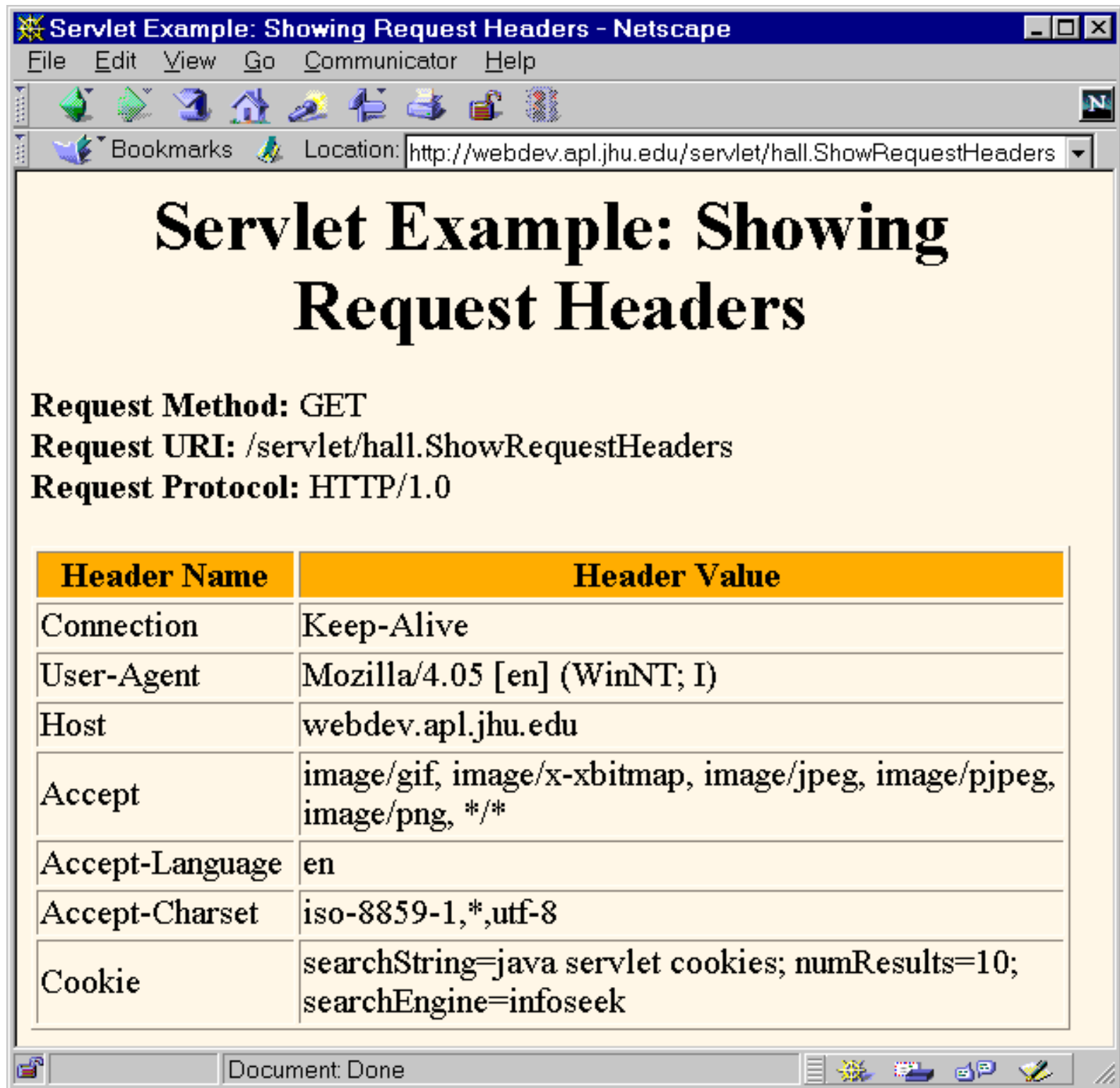
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ShowRequestHeaders extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Servlet Example: Showing Request Headers";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<B>Request Method: </B>" +
            request.getMethod() + "<BR>\n" +
            "<B>Request URI: </B>" +
            request.getRequestURI() + "<BR>\n" +
            "<B>Request Protocol: </B>" +
            request.getProtocol() + "<BR><BR>\n" +
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +
            "<TR BGCOLOR=\"#FFAD00\">\n" +
            "<TH>Header Name<TH>Header Value");
        Enumeration headerNames = request.getHeaderNames();
        while(headerNames.hasMoreElements()) {
            String headerName = (String)headerNames.nextElement();
            out.println("<TR><TD>" + headerName);
            out.println("    <TD>" + request.getHeader(headerName));
        }
        out.println("</TABLE>\n</BODY></HTML>");
    }

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


3.2 Salida de ShowRequestHeaders

Aquí están los resultados de dos peticiones típicas, una de Netscape y otra de Internet Explorer. Veremos la razón por la que Netscape muestra una cabecera Cookie cuando lleguemos a la sección [Cookies](#).



Servlet Example: Showing Request Headers

Request Method: GET
Request URI: /servlet/hall.ShowRequestHeaders
Request Protocol: HTTP/1.0

Header Name	Header Value
Connection	Keep-Alive
User-Agent	Mozilla/4.05 [en] (WinNT; I)
Host	webdev.apl.jhu.edu
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Language	en
Accept-Charset	iso-8859-1,*,utf-8
Cookie	searchString=java servlet cookies; numResults=10; searchEngine=infoseek

Document: Done

Servlet Example: Showing Request Headers - Microsoft Internet Explorer

File Edit View Go Favorites Help

Address <http://webdev.apl.jhu.edu/servlet/hall.ShowRequestHeaders>

Servlet Example: Showing Request Headers

Request Method: GET
Request URI: /servlet/hall.ShowRequestHeaders
Request Protocol: HTTP/1.1

Header Name	Header Value
Accept	image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/msword, application/vnd.ms-powerpoint, */*
Accept-Language	en-us
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/4.0 (compatible; MSIE 4.01; Windows NT)
Host	webdev.apl.jhu.edu
Connection	Keep-Alive

Internet zone

Acceder a Variables Estándards CGI

1. [Introducción a las Variables CGI](#)
2. [Equivalentes Servlet a Variables Estándards CGI](#)
3. [Ejemplo: Leer las Variables CGI](#)

1. Introducción a las Variables CGI

Si llegamos a los servlets Java desde CGI tradicional, probablemente usaremos la idea de "Variables CGI". Estas son una forma ecléctica de colección de información sobre la petición. Algunas se derivan de la línea de petición HTTP y las cabeceras, otras están derivadas desde el propio socket (como el nombre y la dirección IP del host peticionario), y otras derivadas de los parámetros de instalación del servidor (como el mapeo de URLs a los paths actuales).

2. Equivalentes Servlet a la Variables Estándards CGI

Aunque probablemente tiene más sentido pensar en diferentes fuentes de datos (datos de petición, datos de servidor, etc.) como distintas, los programadores experimentados en CGI podrían encontrar muy útil la siguiente tabla. Asumimos que request es el HttpServletRequest suministrado a los métodos doGet y doPost.

Variable CGI	Significado	Acceso desde doGet o doPost
AUTH_TYPE	Si se suministró una cabecera Authorization, este es el esquema especificado (basic o digest)	request.getAuthType()
CONTENT_LENGTH	Sólo para peticiones POST, el número de bytes enviados.	Tecnicamente, el equivalente es String.valueOf(request.getContentLength()) un String) pero probablemente querremos sólo llamar a request.getContentLength(), que devuelve un int.
CONTENT_TYPE	El tipo MIME de los datos adjuntos, si se especifica.	request.getContentType()
DOCUMENT_ROOT	Path al directorio que corresponde con http://host/	getServletContext().getRealPath("/") Observa que era request.getRealPath("/") en especificaciones servlet anteriores.
HTTP_XXX_YYY	Acceso a cabeceras arbitrarias HTTP	request.getHeader("Xxx-Yyy")
PATH_INFO	Información de Path adjunto a la URL. Como los servlets, al contrario que los programas estándares CGI, pueden hablar con el servidor, no necesitan tratar esto de forma separada. La información del path podría ser enviada como parte normal de los datos de formulario.	request.getPathInfo()

PATH_TRANSLATED	La información del path mapeado al path real en el servidor. De nuevo, los Servlets no necesitan tener un caso especial para esto.	request.getPathTranslated()
QUERY_STRING	Para peticiones GET, son los datos adjuntos como un gran string, con los valores codificados. Raramente querremos una fila de datos en los servlets; en su lugar, usaremos request.getParameter para acceder a parámetros individuales.	request.getQueryString()
REMOTE_ADDR	La dirección IP del cliente que hizo la petición, por ejemplo "192.9.48.9".	request.getRemoteAddr()
REMOTE_HOST	El nombre de dominio totalmente cualificado (por ejemplo "java.sun.com") del cliente que hizo la petición. Se devuelve la dirección IP si no se puede determinar.	request.getRemoteHost()
REMOTE_USER	Si se suministró una cabecera Authorization, la parte del usuario.	request.getRemoteUser()
REQUEST_METHOD	El tipo de petición, que normalmente es GET o POST, pero ocasionalmente puede ser HEAD, PUT, DELETE, OPTIONS, o TRACE.	request.getMethod()
SCRIPT_NAME	Path del servlet.	request.getServletPath()
SERVER_NAME	Nombre del Servidor Web.	request.getServerName()
SERVER_PORT	Puerto por el que escucha el servidor.	Técnicamente, el equivalente es String.valueOf(request.getServerPort()), que devuelve un String. Normalmente sólo querremos llamar a request.getServerPort(), que devuelve un int.
SERVER_PROTOCOL	Nombre y versión usada en la línea de petición (por ejemplo HTTP/1.0 o HTTP/1.1).	request.getProtocol()
SERVER_SOFTWARE	Información identificativa del servidor Web.	getServletContext().getServerInfo()

3. Ejemplo: Leer las Variables CGI

Aquí tenemos un servelt que crea una tabla que muestra los valores de todas las variables CGI distintas a HTTP_XXX_YYY, que son sólo cabeceras de petición HTTP que se mostraron en la página anterior.

3.1 ShowCGIVariables.java

También puedes [descargar el código fuente](#).

```
package hall;

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

/** Creates a table showing the values of all the CGI variables.
 *
 * Part of tutorial on servlets and JSP that appears at
 * http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
 * 1999 Marty Hall; may be freely used or adapted.
 */

public class ShowCGIVariables 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() }
            };

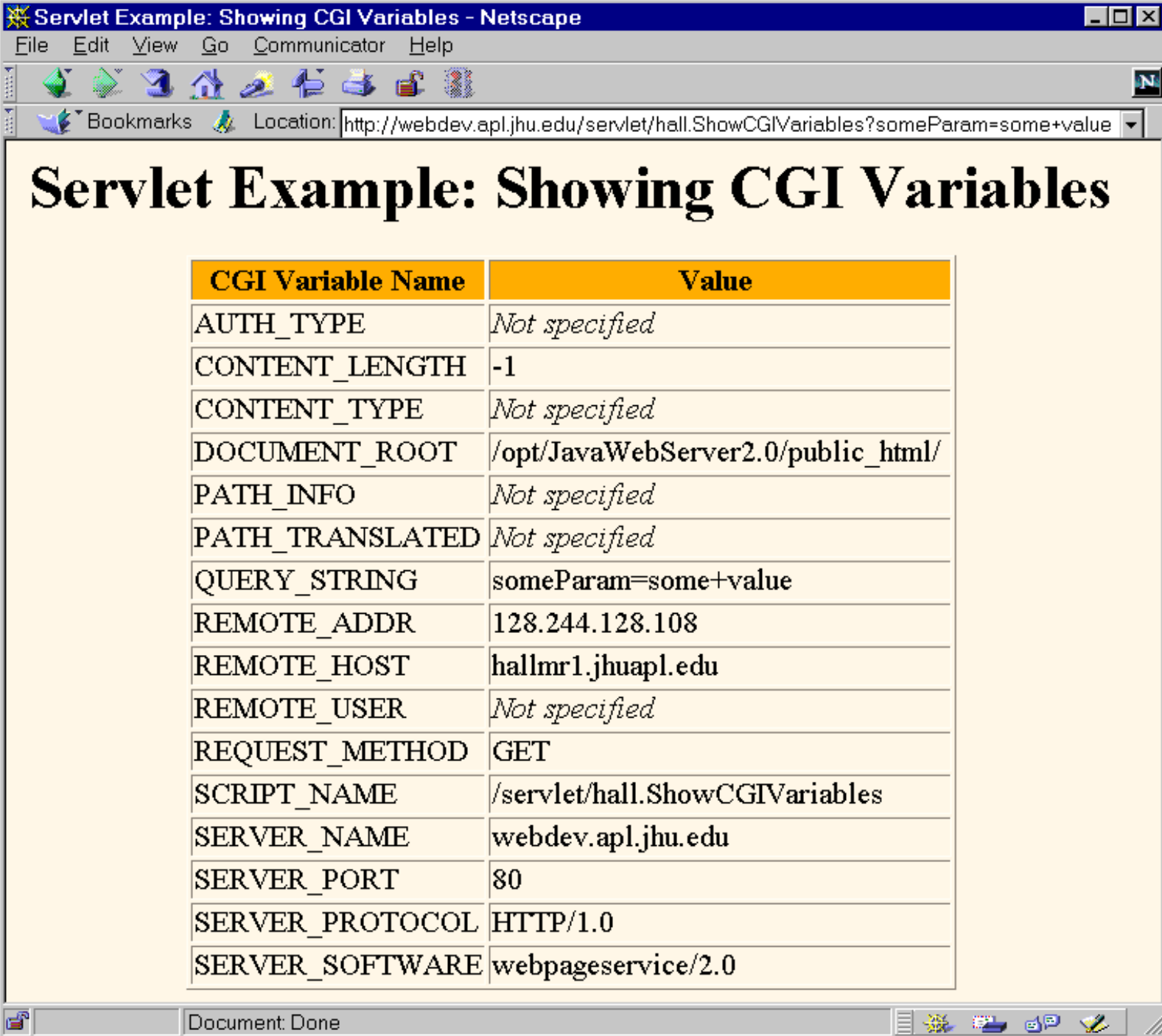
        String title = "Servlet Example: Showing CGI Variables";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +
            "<TR BGCOLOR=\"#FFAD00\">\n" +
            "<TH>CGI Variable Name<TH>Value");
        for(int i=0; i<variables.length; i++) {
            String varName = variables[i][0];
            String varValue = variables[i][1];
            if (varValue == null)
                varValue = "<I>Not specified</I>";
            out.println("<TR><TD>" + varName + "<TD>" + varValue);
        }
        out.println("</TABLE></BODY></HTML>");
    }
}
```

```

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

```

3.2 Salida de ShowCGIVariables



Servlet Example: Showing CGI Variables

CGI Variable Name	Value
AUTH_TYPE	<i>Not specified</i>
CONTENT_LENGTH	-1
CONTENT_TYPE	<i>Not specified</i>
DOCUMENT_ROOT	/opt/JavaWebServer2.0/public_html/
PATH_INFO	<i>Not specified</i>
PATH_TRANSLATED	<i>Not specified</i>
QUERY_STRING	someParam=some+value
REMOTE_ADDR	128.244.128.108
REMOTE_HOST	hallmr1.jhuapl.edu
REMOTE_USER	<i>Not specified</i>
REQUEST_METHOD	GET
SCRIPT_NAME	/servlet/hall.ShowCGIVariables
SERVER_NAME	webdev.apl.jhu.edu
SERVER_PORT	80
SERVER_PROTOCOL	HTTP/1.0
SERVER_SOFTWARE	webpageservice/2.0

Document: Done

Códigos de Estado HTTP

1. [Introducción](#)
 2. [Especificar Códigos de Estado](#)
 3. [Códigos de Estado HTTP 1.1 y sus Significados](#)
 4. [Ejemplo: Motor de Búsqueda](#)
-

1. Introducción

Cuando un servidor Web responde a una petición de un navegador u otro cliente Web, la respuesta consiste típicamente en una línea de estado, algunas cabeceras de respuesta, una línea en blanco, y el documento. Aquí tenemos un ejemplo mínimo:

```
HTTP/1.1 200 OK
Content-Type: text/plain
```

```
Hello World
```

La línea de estado consiste en la versión HTTP, y un entero que se interpreta como código de estado, y un mensaje muy corto que corresponde con el código de estado. En la mayoría de los casos, todas las cabeceras son opcionales excepto Content-Type, que especifica el tipo MIME del documento que sigue. Aunque muchas respuestas contienen un documento, algunas no lo tienen. Por ejemplo, las respuestas a peticiones HEAD nunca deberían incluir un documento, y hay una gran variedad de códigos de estado que esencialmente indican fallos, y o no incluyen un documento o sólo incluyen un pequeño "mensaje de error de documento".

Los servlets pueden realizar una variedad de tareas manipulando la línea de estado y las cabeceras de respuesta. Por ejemplo, reenviar al usuario a otros sites; indicar que el documento adjunto es una imagen, un fichero Acrobat, o (más comunmente) un fichero HTML; decirle al usuario que se requiere una password para acceder al documento; etc. Esta sección explica varios códigos de estados diferentes y como se pueden conseguir, mientras que [la página siguiente](#) describe la cabeceras de respuesta.

2. Especificar Códigos de Estado

Como se describe arriba, la línea de estado de respuesta HTTP consiste en una versión HTTP, un código de estado, y un mensaje asociado. Como el mensaje está asociado directamente con el código de estado y la versión HTTP está determinada por el servidor, todo lo que servidor tiene que hacer es seleccionar el código de estado. La forma de hacer esto es mediante el método setStatus de HttpServletResponse. El método setStatus toma un int (el código de estado) como argumento, pero en vez de usar los número explícitamente, es más claro y legible usar las constantes definidas en HttpServletResponse. El nombre de cada constante está derivado del mensaje estándar HTTP 1.1 para cada constante, todo en mayúsculas con un prefijo SC (por Status Code) y los espacios se cambian por subrayados. Así, como el mensaje para 404 es Not Found, la constante equivalente en HttpServletResponse es SC_NOT_FOUND. Sin embargo, hay dos excepciones. Por alguna razón oculta la constante para el código 302 se deriva del

mensaje HTTP 1.0, no de HTTP 1.1, y la constante para el código 307 no existe tampoco.

Seleccionar el código de estado no siempre significa que no necesitemos devolver un documento. Por ejemplo, aunque la mayoría de los servidores generarán un pequeño mensaje "File Not Found" para respuestas 404, un servlet podría querer personalizar esta respuesta. Sin embargo, si hacemos esto, necesitamos estar seguros de llamar a `response.setStatus` antes de enviar el contenido mediante `PrintWriter`.

Aunque el método general de selección del código de estado es simplemente llamar a `response.setStatus(int)`, hay dos casos comunes para los que se proporciona un método atajo en `HttpServletResponse`. El método `sendError` genera un respuesta 404 junto con un mensaje corto formateado dentro de un documento HTML. Y el método `sendRedirect` genera una respuesta 302 junto con una cabecera `Location` indicando la URL del nuevo documento.

3. Códigos de Estado HTTP 1.1 y sus Significados

Aquí hay una lista de todas los códigos de estado disponibles en HTTP 1.1 junto con sus mensajes asociados y su interpretación. Deberíamos ser cuidadosos al utilizar los códigos de estado que están disponibles sólo en HTTP 1.1, ya que muchos navegadores sólo soportan HTTP 1.0. Si tenemos que usar códigos de estado específicos para HTTP 1.1, en la mayoría de los casos queremos chequear explícitamente la versión HTTP de la petición (mediante el método `getProtocol` de `HttpServletRequest`) o reservarlo para situaciones donde no importe el significado de la cabecera HTTP 1.0.

Código de Estado	Mensaje Asociado	Significado
100	Continue	Continúa con petición parcial (nuevo en HTTP 1.1)
101	Switching Protocols	El servidor cumplirá con la cabecera <code>Upgrade</code> y cambiará a un protocolo diferente. (Nuevo en HTTP 1.1)
200	OK	Todo está bien; los documentos seguidos por peticiones GET y POST. Esto es por defecto para los Servlets, si no usamos <code>setStatus</code> , obtendremos esto.
201	Created	El servidor creo un documento; la cabecera <code>Location</code> indica la URL.
202	Accepted	La petición se está realizando, el proceso no se ha completado.
203	Non-Authoritative Information	El documento está siendo devuelto normalmente, pero algunas cabeceras de respuesta podrían ser incorrectas porque se está usando una copia del documento (Nuevo en HTTP 1.1)
204	No Content	No hay un documento nuevo; el navegador continúa mostrando el documento anterior. Esto es útil si el usuario recarga periódicamente una página y podemos determinar que la página anterior ya está actualizada. Sin embargo, esto no funciona para páginas que se recargan automáticamente mediante cabeceras de respuesta <code>Refresh</code> o su equivalente <code><META HTTP-EQUIV="Refresh" ...></code> , ya que al devolver este código de estado se pararán futuras recargas.
205	Reset Content	No hay documento nuevo, pero el navegador debería resetear el documento. Usado para forzar al navegador a borrar los contenidos de los campos de un formulario CGI (Nuevo en HTTP 1.1)
206	Partial Content	El cliente envía una petición parcial con una cabecera <code>Range</code> , y el servidor la ha completado. (Nuevo en HTTP 1.1)

300	Multiple Choices	El documento pedido se puede encontrar en varios sitios; serán listados en el documento devuelto. Si el servidor tiene una opción preferida, debería listarse en la cabecera de respuesta Location .
301	Moved Permanently	El documento pedido está en algún lugar, y la URL se da en la cabecera de respuesta Location. Los navegadores deberían seguir automáticamente el enlace a la nueva URL.
302	Found	<p>Similar a 301, excepto que la nueva URL debería ser interpretada como reemplazada temporalmente, no permanentemente. Observa: el mensaje era "Moved Temporarily" en HTTP 1.0, y la constante en HttpServletResponse es SC_MOVED_TEMPORARILY, no SC_FOUND. Cabecera muy útil, ya que los navegadores siguen automáticamente el enlace a la nueva URL. Este código de estado es tan útil que hay un método especial para ella, sendRedirect. Usar response.sendRedirect(url) tiene un par de ventajas sobre hacer response.setStatus(response.SC_MOVED_TEMPORARILY) y response.setHeader("Location", url) . Primero, es más fácil. Segundo, con sendRedirect, el servlet automáticamente construye una página que contiene el enlace (para mostrar a los viejos navegadores que no siguen las redirecciones automáticamente). Finalmente, sendRedirect puede manejar URLs relativas, automáticamente las traducen a absolutas.</p> <p>Observa que este código de estado es usado algunas veces de forma intercambiada con 301. Por ejemplo, si erróneamente pedimos http://host/~user (olvidando la última barra), algunos servidores enviarán 301 y otros 302.</p> <p>Técnicamente, se supone que los navegadores siguen automáticamente la redirección su la petición original era GET. Puedes ver la cabecera 307 para más detalles.</p>
303	See Other	Igual que 301/302, excepto que si la petición original era POST, el documento redirigido (dado en la cabecera Location) debería ser recuperado mediante GET. (Nuevo en HTTP 1.1)
304	Not Modified	El cliente tiene un documento en el caché y realiza una petición condicional (normalmente suministrando una cabecera If-Modified-Since indicando que sólo quiere documentos más nuevos que la fecha especificada). El servidor quiere decirle al cliente que el viejo documento del caché todavía está en uso.
305	Use Proxy	El documento pedido debería recuperarse mediante el proxy listado en la cabecera Location. (Nuevo en HTTP 1.1)

307	Temporary Redirect	Es idéntica a 302 ("Found" o "Temporarily Moved"). Fue añadido a HTTP 1.1 ya que muchos navegadores siguen erróneamente la redirección de una respuesta 302 incluso si el mensaje original fue un POST, y sólo se debe seguir la redirección de una petición POST en respuestas 303. Esta respuesta es algo ambigua: sigue el redireccionamiento para peticiones GET y POST en el caso de respuestas 303, y en el caso de respuesta 307 sólo sigue la redirección de peticiones GET. Nota: por alguna razón no existe una constante en HttpServletResponse que corresponda con este código de estado. (Nuevo en HTTP 1.1)
400	Bad Request	Mala Sintaxis de la petición.
401	Unauthorized	El cliente intenta acceder a una página protegida por password sin las autorización apropiada. La respuesta debería incluir una cabecera WWW-Authenticate que el navegador debería usar para mostrar la caja de diálogo usuario/password, que viene de vuelta con la cabecera Authorization.
403	Forbidden	El recurso no está disponible, si importar la autorización. Normalmente indica la falta permisos de fichero o directorios en el servidor.
404	Not Found	No se pudo encontrar el recurso en esa dirección. Esta la respuesta estándar "no such page". Es tan común y útil esta respuesta que hay un método especial para ella en HttpServletResponse: sendError(message). La ventaja de sendError sobre setStatus es que, con sendErr, el servidor genera automáticamente una página que muestra un mensaje de error.
405	Method Not Allowed	El método de la petición (GET, POST, HEAD, DELETE, PUT, TRACE, etc.) no estaba permitido para este recurso particular. (Nuevo en HTTP 1.1)
406	Not Acceptable	El recurso indicado genera un tipo MIME incompatible con el especificado por el cliente mediante su cabecera Accept. (Nuevo en HTTP 1.1)
407	Proxy Authentication Required	Similar a 401, pero el servidor proxy debería devolver una cabecera Proxy-Authenticate. (Nuevo en HTTP 1.1)
408	Request Timeout	El cliente tarda demasiado en enviar la petición. (Nuevo en HTTP 1.1)
409	Conflict	Usualmente asociado con peticiones PUT; usado para situaciones como la carga de una versión incorrecta de un fichero. (Nuevo en HTTP 1.1)
410	Gone	El documento se ha ido; no se conoce la dirección de reenvío. Difiere de la 404 en que se sabe que el documento se ha ido permanentemente, no sólo está indisponible por alguna razón desconocida como con 404. (Nuevo en HTTP 1.1)
411	Length Required	El servidor no puede procesar la petición a menos que el cliente envíe una cabecera Content-Length. (Nuevo en HTTP 1.1)
412	Precondition Failed	Alguna condición previa especificada en la petición era falsa (Nuevo en HTTP 1.1)

413	Request Entity Too Large	El documento pedido es mayor que lo que el servidor quiere manejar ahora. Si el servidor cree que puede manejarlo más tarde, debería incluir una cabecera Retry-After. (Nuevo en HTTP 1.1)
414	Request URI Too Long	La URI es demasiado larga. (Nuevo en HTTP 1.1)
415	Unsupported Media Type	La petición está en un formato desconocido. (Nuevo en HTTP 1.1)
416	Requested Range Not Satisfiable	El cliente incluyó una cabecera Range no satisfactoria en la petición. (Nuevo en HTTP 1.1)
417	Expectation Failed	No se puede conseguir el valor de la cabecera Expect. (Nuevo en HTTP 1.1)
500	Internal Server Error	Mensaje genérico "server is confused". Normalmente es el resultado de programas CGI o servlets que se quedan colgados o retornan cabeceras mal formateadas.
501	Not Implemented	El servidor no soporta la funcionalidad de rellenar peticiones. Usado, por ejemplo, cuando el cliente envía comandos como PUT que el cliente no soporta.
502	Bad Gateway	Usado por servidores que actúan como proxies o gateways; indica que el servidor inicial obtuvo una mala respuesta desde el servidor remoto.
503	Service Unavailable	El servidor no puede responder debido a mantenimiento o sobrecarga. Por ejemplo, un servlet podría devolver esta cabecera si algún almacén de threads o de conexiones con bases de datos están llenos. El servidor puede suministrar una cabecera Retry-After.
504	Gateway Timeout	Usado por servidores que actúan como proxies o gateways; indica que el servidor inicial no obtuvo una respuesta a tiempo del servidor remoto. (Nuevo en HTTP 1.1)
505	HTTP Version Not Supported	El servidor no soporta la versión de HTTP indicada en la línea de petición. (Nuevo en HTTP 1.1)

4. Ejemplo: Motor de Búsqueda

Aquí tenemos un ejemplo que hace uso de los dos códigos de estado más comunes distintos de 200: 302 y 404. El código 302 se selecciona mediante el método `sendRedirect`, y 404 se selecciona mediante `sendError`.

En esta aplicación, primero un formulario HTML muestra una página que permite al usuario elegir una cadena de búsqueda, el número de los resultados por página, y el motor de búsqueda a utilizar. Cuando se envía el formulario, el servlet extrae estos tres parámetros, construye una URL con los parámetros embebidos en una forma apropiada para el motor de búsqueda seleccionado, y redirige al usuario a esa dirección. Si el usuario falla al elegir el motor de búsqueda o envía un nombre de motor de búsqueda no conocido, se devuelve una página de error 404 diciendo que no hay motor de búsqueda o que no se conoce.

4.1 SearchEngines.java (Descarga [el código fuente](#))

Nota: hace uso de la clase `SearchSpec`, mostrada abajo, que incorpora información sobre como construir URLs para realizar búsquedas en varios buscadores.

```

package hall;

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

public class SearchEngines extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        // The URLEncoder changes spaces to "+" signs and other
        // non-alphanumeric characters to "%XY", where XY is the
        // hex value of the ASCII (or ISO Latin-1) character.
        // The getParameter method decodes automatically, but since
        // we're just passing this on to another server, we need to
        // re-encode it.
        String searchString =
            URLEncoder.encode(request.getParameter("searchString"));
        String numResults =
            request.getParameter("numResults");
        String searchEngine =
            request.getParameter("searchEngine");
        SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
        for(int i=0; i<commonSpecs.length; i++) {
            SearchSpec searchSpec = commonSpecs[i];
            if (searchSpec.getName().equals(searchEngine)) {
                // encodeURL is just planning ahead in case this servlet
                // is ever used in an application that does session tracking.
                // If cookies are turned off, session tracking is usually
                // accomplished by URL rewriting, so all URLs returned
                // by servlets should be sent through encodeURL.
                String url =
                    response.encodeURL(searchSpec.makeURL(searchString,
                                                            numResults));

                response.sendRedirect(url);
                return;
            }
        }
        response.sendError(response.SC_NOT_FOUND,
                          "No recognized search engine specified.");
    }

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

```

4.2 SearchSpec.java

```
package hall;

class SearchSpec {
    private String name, baseURL, numResultsSuffix;

    private static SearchSpec[] commonSpecs =
        { new SearchSpec("google",
                        "http://www.google.com/search?q=",
                        "&num="),
          new SearchSpec("infoseek",
                        "http://infoseek.go.com/Titles?qt=",
                        "&nh="),
          new SearchSpec("lycos",
                        "http://lycospro.lycos.com/cgi-bin/pursuit?query=",
                        "&maxhits="),
          new SearchSpec("hotbot",
                        "http://www.hotbot.com/?MT=",
                        "&DC=")
        };

    public SearchSpec(String name,
                      String baseURL,
                      String numResultsSuffix) {
        this.name = name;
        this.baseURL = baseURL;
        this.numResultsSuffix = numResultsSuffix;
    }

    public String makeURL(String searchString, String numResults) {
        return(baseURL + searchString + numResultsSuffix + numResults);
    }

    public String getName() {
        return(name);
    }

    public static SearchSpec[] getCommonSpecs() {
        return(commonSpecs);
    }
}
```

4.3 SearchEngines.html

Pulsa con el botón derecho sobre el [enlace al código fuente](#) para descargar el fichero fuente.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>Searching the Web</TITLE>
</HEAD>
```

```

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Searching the Web</H1>

<FORM ACTION="/servlet/hall.SearchEngines">
  <CENTER>
    Search String:
    <INPUT TYPE="TEXT" NAME="searchString"><BR>
    Results to Show Per Page:
    <INPUT TYPE="TEXT" NAME="numResults"
      VALUE=10 SIZE=3><BR>
    <INPUT TYPE="RADIO" NAME="searchEngine"
      VALUE="google">

    Google |
    <INPUT TYPE="RADIO" NAME="searchEngine"
      VALUE="infoseek">

    Infoseek |
    <INPUT TYPE="RADIO" NAME="searchEngine"
      VALUE="lycos">

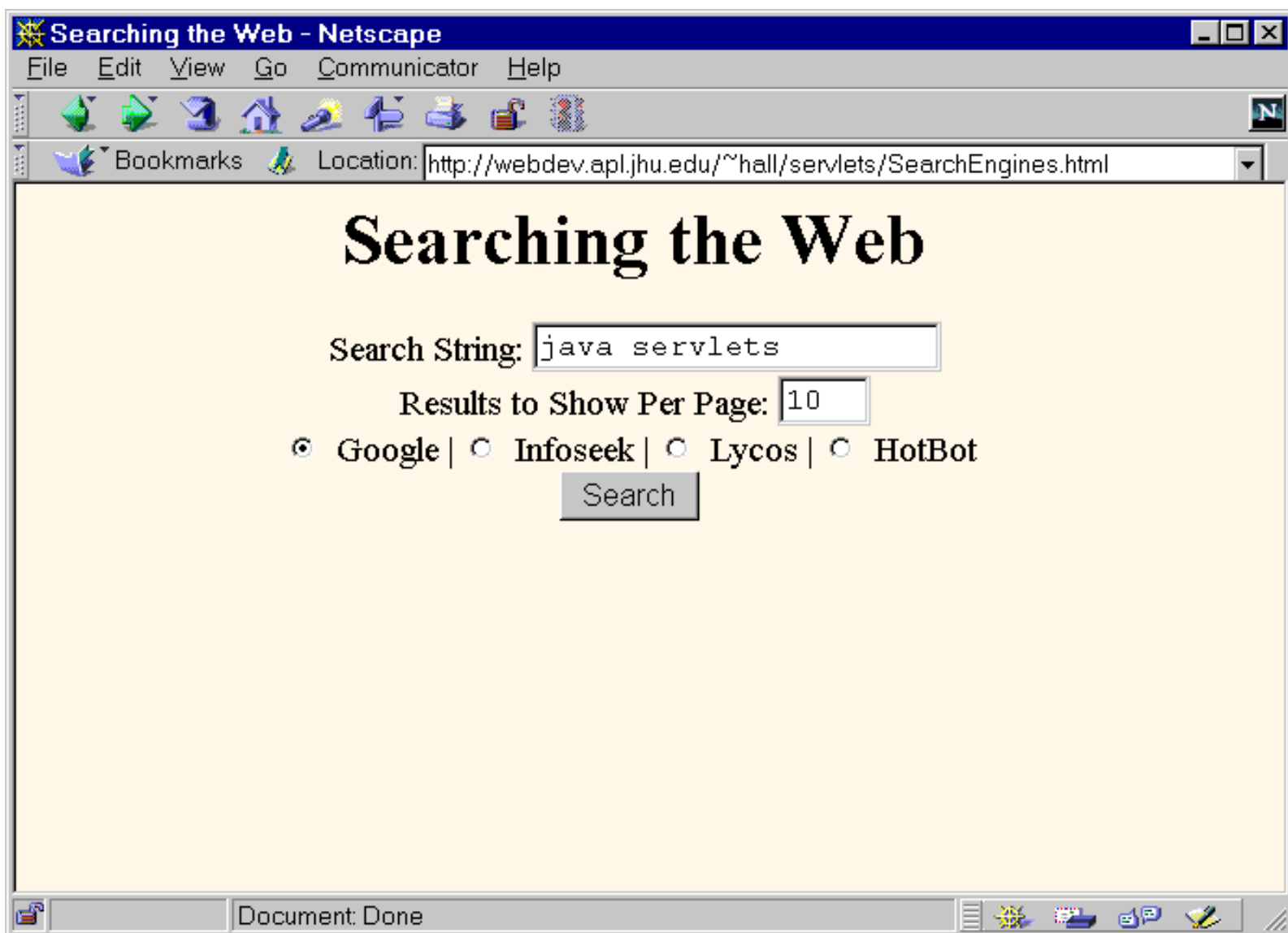
    Lycos |
    <INPUT TYPE="RADIO" NAME="searchEngine"
      VALUE="hotbot">

    HotBot
    <BR>
    <INPUT TYPE="SUBMIT" VALUE="Search">
  </CENTER>
</FORM>

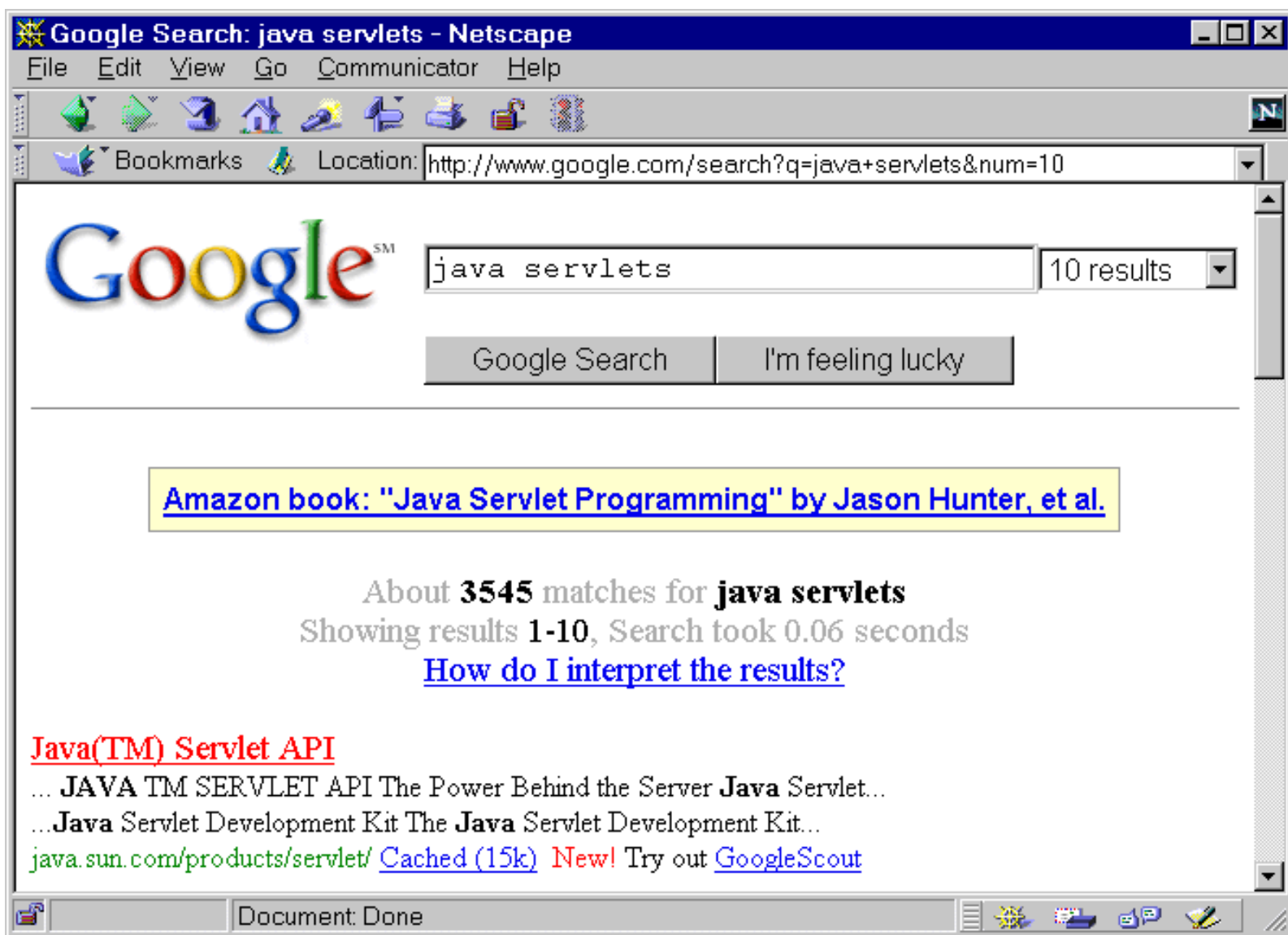
</BODY>
</HTML>

```

4.4 Pantalla inicial



4.5 Resultados de la Búsqueda



Especificar Cabeceras de Respuesta HTTP

1. [Introducción](#)
2. [Cabeceras de Respuesta más comunes y sus Significados.](#)
3. [Ejemplo: Recarga Automática de Páginas con Content Changes](#)

1. Introducción

Una respuesta desde un servidor Web normalmente consiste en una línea de estado, una o más cabeceras de respuesta, una línea en blanco, y el documento. Seleccionar las cabeceras de respuesta normalmente va mano con mano con la selección de códigos de estado en la línea de estado. Por ejemplo, muchos de los códigos de estado "document moved" tienen una cabecera Location de acompañamiento, y un 401 (Unauthorized) debe incluir una cabecera WWW-Authenticate.

Sin embargo, especificar las cabeceras puede jugar un rol muy útil cuando se selecciona códigos de estado no usuales. Las cabeceras de respuesta se pueden usar para especificar cookies, para suministrar la fecha de modificación (para el caché), para instruir al navegador sobre la recarga de la página después de un intervalo designado, para decir cuanto tiempo va a estar el fichero usando conexiones persistentes, y otras muchas tareas.

La forma más general de especificar cabeceras es mediante el método `setHeader` de `HttpServletResponse`, que toma dos strings: el nombre de la cabecera y el valor de ésta. Igual que la selección de los códigos de estado, esto debe hacerse antes de enviar cualquier documento.

Hay también dos métodos especializados para seleccionar cabeceras que contienen fechas (`setDateHeader`) y enteros (`setIntHeader`). La primera nos evita el problema de tener que traducir una fecha Java en milisegundos (como al devuelta por los métodos `System.currentTimeMillis` o `getTime` aplicados a un objeto `Date`) en string GMT. El segundo nos ahorra la inconveniencia menor de convertir un `int` a un `String`.

En el caso de una cabecera cuyo nombre ya exista, podemos añadir una nueva cabecera en vez de seleccionarla de nuevo. Usamos `addHeader`, `addDateHeader`, y `addIntHeader` para esto. Si realmente nos importa si una cabecera específica se ha seleccionado, podemos usar `containsHeader` para comprobarlo.

Finalmente, `HttpServletResponse` también suministra unos métodos de conveniencia para especificar cabeceras comunes:

- El método `setContentType` selecciona la cabecera `Content-Type`, y se usa en la mayoría de los Servlets.
- El método `setContentLength` selecciona la cabecera `Content-Length`, útil si el navegador soporta conexiones HTTP persistentes (keep-alive).
- El método `addCookie` selecciona un cookie (no existe el correspondiente `setCookie`, ya que es normal que haya varias líneas `Set-Cookie`).

- Y como se especificó en la [página anterior](#), el método sendRedirect selecciona la cabecera Location así como se selecciona el código de estado 302.

2. Cabeceras de Respuesta más Comunes y sus Significados

Cabecera	Interpretación/Propósito
Allow	¿Qué métodos de petición (GET, POST, etc.) soporta el servidor?
Content-Encoding	¿Qué método se utilizó para codificar el documento? Necesitamos decodificarlo para obtener el tipo especificado por la cabecera Content-Type.
Content-Length	¿Cuántos bytes han sido enviados? Esta información es sólo necesaria si el navegador está usando conexiones persistentes. Si queremos aprovecharnos de esto cuando el navegador lo soporte, nuestro servlet debería escribir el documento en un ByteArrayOutputStream, preguntar su tamaño cuando se haya terminado, ponerlo en el campo Content-Length, luego enviar el contenido mediante byteArrayStream.writeTo(response.getOutputStream()).
Content-Type	¿Cuál es el tipo MIME del siguiente documento? Por defecto en los servlets es text/plain, pero normalmente especifican explícitamente text/html. Seleccionar esta cabecera es tan común que hay un método especial en HttpServletResponse para el: setContentType.
Date	¿Cuál es la hora actual (en GMT)? Usamos el método setDateHeader para especificar esta cabecera.
Expires	¿En qué momento debería considerarse el documento como caducado y no se pondrá más en el caché?
Last-Modified	¿Cuándo se modificó el documento por última vez? El cliente puede suministrar una fecha mediante la cabecera de petición If-Modified-Since. Esta es tratada como un GET condicional, donde sólo se devuelven documentos si la fecha Last-Modified es posterior que la fecha especificada. De otra forma se devuelve una línea de estado 304 (Not Modified). De nuevo se usa el método setDateHeader para especificar esta cabecera.
Location	¿Dónde debería ir cliente para obtener el documento? Se selecciona indirectamente con un código de estado 302, mediante el método sendRedirect de HttpServletResponse.
Refresh	¿Cuándo (en milisegundos) debería perder el navegador una página actualizada? En lugar de recargar la página actual, podemos especificar otra página a cargar mediante setHeader("Refresh", "5; URL=http://host/path"). Nota: esto se selecciona comunmente mediante <META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://host/path"> en la sección HEAD de la página HTML, mejor que una cabecera explícita desde el servidor. Esto es porque la recarga o el reenvío automático es algo deseado por los autores de HTML que no tienen accesos a CGI o servlets. Pero esta cabecera significa "Recarga esta página o ve a URL especificada en n segundos". No significa "recarga esta página o ve la URL especificada cada n segundos". Por eso tenemos que enviar una cabecera Refresh cada vez. Nota: esta cabecera no forma parte oficial del HTTP 1.1, pero es una extensión

	soportada por Netspace e Internet Explorer
Server	¿Qué servidor soy? Los servlets normalmente no usan esto; lo hace el propio servidor.
Set-Cookie	Especifica una Cookie asociada con la página. Los servlets no deberían usar <code>response.setHeader("Set-Cookie", ...)</code> , pero en su lugar usan el método de propósito especial <code>addCookie</code> de <code>HttpServletResponse</code> .
WWW-Authenticate	¿Qué tipo de autorización y dominio debería suministrar el cliente en su cabecera Authorization? Esta cabecera es necesaria en respuestas que tienen una línea de estado 401 (Unauthorized). Por ejemplo <code>response.setHeader("WWW-Authenticate", "BASIC realm= \"executives\"")</code> .

Para más detalles sobre cabeceras HTTP, puedes ver las especificaciones en <http://www.w3.org/Protocols/>.

3. Ejemplo: Recarga Automática de Páginas como Cambio de Contenido

Aquí tenemos un ejemplo que nos permite pedir una lista de grandes números primos. Como esto podría tardar algún tiempo para números muy largos (por ejemplo 150 dígitos), el servlet devuelve los resultados hasta donde haya llegado, pero sigue calculando, usando un thread de baja prioridad para que no degrade el rendimiento del servidor Web. Si los cálculos no se han completado, instruye al navegador para que pida una nueva página en unos pocos segundos enviando una cabecera Refresh.

Además de ilustrar el valor de las cabeceras de respuesta HTTP, este ejemplo muestra otras dos capacidades de los servlets. Primero, muestra que el mismo servlet puede manejar múltiples conexiones simultáneas, cada una con su propio thread. Por eso, mientras un thread está finalizando los cálculos para un cliente, otro cliente puede conectarse y todavía ver resultados parciales.

Segundo, este ejemplo muestra lo fácil que es para los servlets mantener el estado entre llamadas, algo que es engorroso de implementar en CGI tradicional y sus alternativas. Sólo se crea un ejemplar del Servlet, y cada petición simplemente resulta en un nuevo thread que llama al método `service` del servlet (que a su vez llama a `doGet` o `doPost`). Por eso los datos compartidos sólo tienen que ser situados en una variable normal de ejemplar (campo) del servlet. Así el servlet puede acceder la cálculo de salida apropiado cuando el navegador recarga la página y puede mantener una lista de los resultados de las N solicitudes más recientes, retornándolas inmediatamente si una nueva solicitud especifica los mismo parámetros que otra reciente. Por supuesto, que se aplican las mismas reglas para sincronizar el acceso multi-thread a datos compartidos.

Los servlets también pueden almacenar datos persistentes en el objeto `ServletContext` que está disponible a través del método `getServletContext`. `ServletContext` tiene métodos `setAttribute` y `getAttribute` que nos permiten almacenar datos arbitrarios asociados con claves especificadas. La diferencia entre almacenar los datos en variables de ejemplar y almacenarlos en el `ServletContext` es que éste es compartido por todos los servlets en el motor servlet (o en la aplicación Web, si nuestro servidor soporta dicha capacidad).

3.1 PrimeNumbers.java (Descargar [el código fuente](#))

Nota: También usa [ServletUtilities.java](#), mostrado anteriormente, [PrimeList.java](#) para crear un vector de números primos en un thread de segundo plano, y [Primes.java](#) para generar grandes números aleatorios del tipo BigInteger y chequear si son primos:

```
package hall;

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

public class PrimeNumbers extends HttpServlet {
    private static Vector primeListVector = new Vector();
    private static int maxPrimeLists = 30;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        int numPrimes =
            ServletUtilities.getIntParameter(request, "numPrimes", 50);
        int numDigits =
            ServletUtilities.getIntParameter(request, "numDigits", 120);
        PrimeList primeList =
            findPrimeList(primeListVector, numPrimes, numDigits);
        if (primeList == null) {
            primeList = new PrimeList(numPrimes, numDigits, true);
            synchronized(primeListVector) {
                if (primeListVector.size() >= maxPrimeLists)
                    primeListVector.removeElementAt(0);
                primeListVector.addElement(primeList);
            }
        }
        Vector currentPrimes = primeList.getPrimes();
        int numCurrentPrimes = currentPrimes.size();
        int numPrimesRemaining = (numPrimes - numCurrentPrimes);
        boolean isLastResult = (numPrimesRemaining == 0);
        if (!isLastResult) {
            response.setHeader("Refresh", "5");
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Some " + numDigits + "-Digit Prime Numbers";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
            "<H3>Primes found with " + numDigits +
            " or more digits: " + numCurrentPrimes + ".</H3>");
        if (isLastResult)
            out.println("<B>Done searching.</B>");
    }
}
```

```

else
    out.println("<B>Still looking for " + numPrimesRemaining +
        " more<BLINK>...</BLINK></B>");
out.println("<OL>");
for(int i=0; i<numCurrentPrimes; i++) {
    out.println("  <LI>" + currentPrimes.elementAt(i));
}
out.println("</OL>");
out.println("</BODY></HTML>");
}

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

// See if there is an existing ongoing or completed calculation with
// the same number of primes and length of prime. If so, return
// those results instead of starting a new background thread. Keep
// this list small so that the Web server doesn't use too much memory.
// Synchronize access to the list since there may be multiple simultaneous
// requests.

private PrimeList findPrimeList(Vector primeListVector,
                                int numPrimes,
                                int numDigits) {
    synchronized(primeListVector) {
        for(int i=0; i<primeListVector.size(); i++) {
            PrimeList primes = (PrimeList)primeListVector.elementAt(i);
            if ((numPrimes == primes.numPrimes()) &&
                (numDigits == primes.numDigits()))
                return(primes);
        }
        return(null);
    }
}
}

```

3.3 PrimeNumbers.html

Nota: pulsa con el botón derecho sobre el [enlace al código fuente](#).

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
</HEAD>

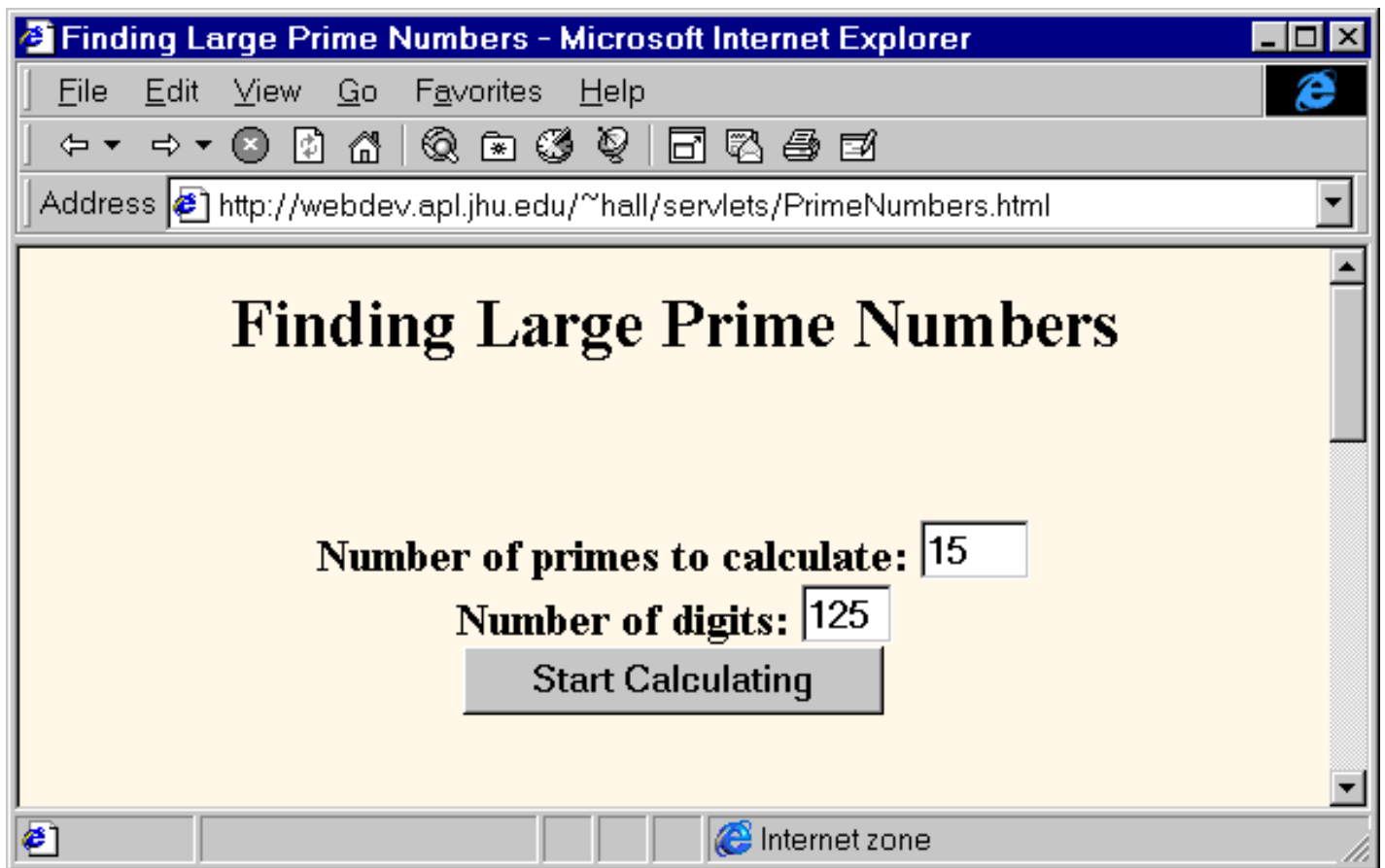
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Finding Large Prime Numbers</H2>
<BR><BR>

```

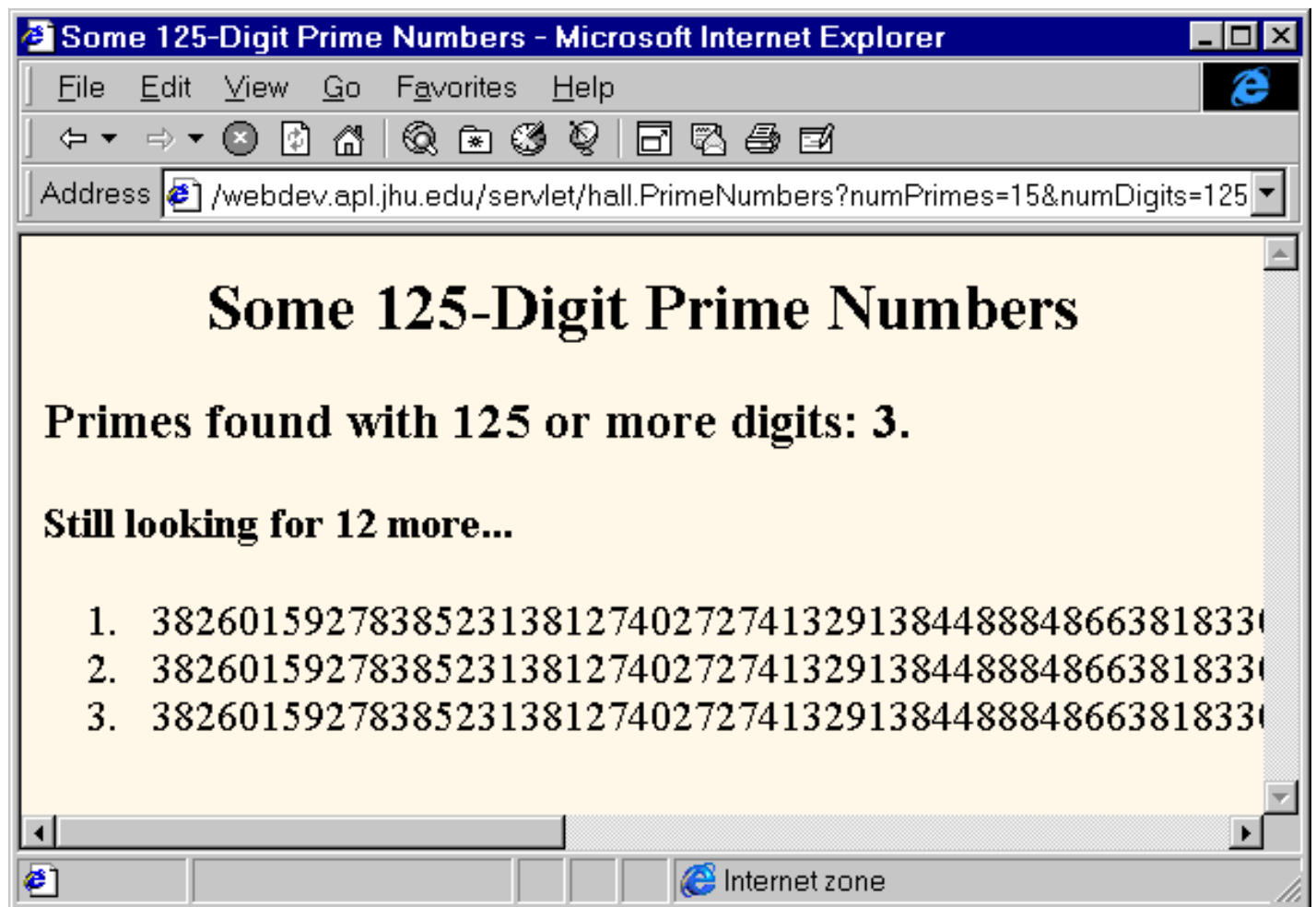
```
<CENTER>
<FORM ACTION="/servlet/hall.PrimeNumbers">
  <B>Number of primes to calculate:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Number of digits:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>

</BODY>
</HTML>
```

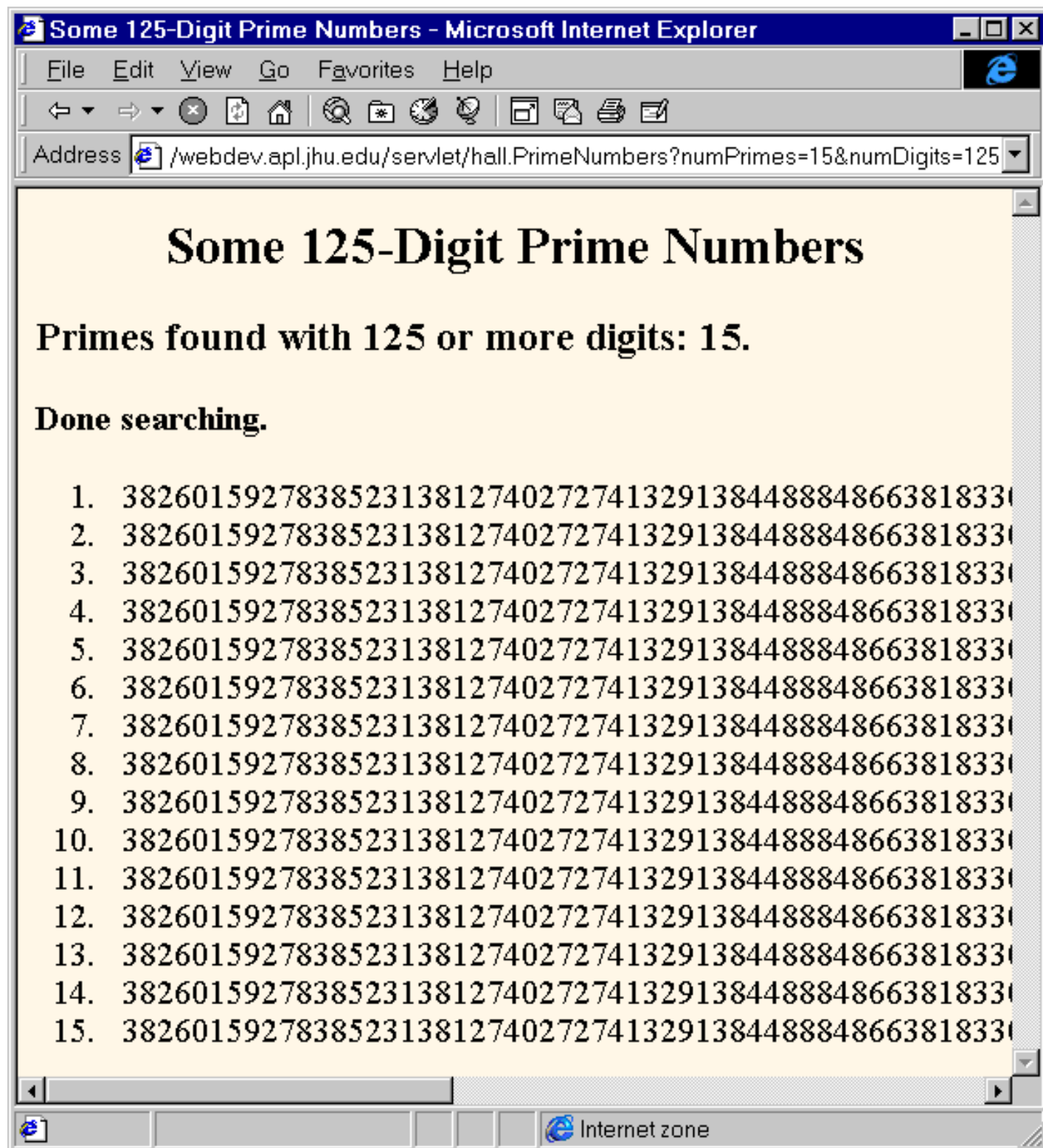
3.4 Inicio



3.5 Resultados Intermedios



3.6 Resultado Final



Manejar Cookies

1. [Introducción a los Cookies](#)
2. [El API Servlet Cookie](#)
3. [Algunas utilidades menores de Cookies](#)
4. [Ejemplo: Un Interface de Motor de Búsqueda Personalizado](#)

1. Introducción a los Cookies

Los Cookies son pequeños trozos de información textual que el servidor Web envía al navegador y que el navegador devuelve sin modificar cuando visita más tarde la misma site o dominio. Dejando que el servidor lea la información enviada previamente al cliente, la site puede proporcionar a los visitantes un número de conveniencias:

- Identificar a un usuario durante una sesión de comercio electrónico. Muchas tiendas on-line usan una "carta de compra" a la que se añaden los artículos que selecciona el usuario, luego continúa comprando. Como la conexión HTTP se cierra después de enviar cada página, cuando el usuario selecciona un nuevo ítem para su carta. ¿cómo sabe la tienda que él es el mismo usuario que añadió el ítem anterior a su carta? Los cookies son una buena forma de conseguir esto. De hecho, son tan útiles que los servlets tienen un API específico para esto, y los autores de servlets no necesitan manipular las cookies directamente haciendo uso de él. Esto se explica en la [página siguiente](#).
- Evitar el nombre de usuario y la password. Muchas grandes sites requieren registrarse para poder usar sus servicios, pero es un inconveniente recordar el nombre de usuario y la password. Los cookies son una buena alternativa para sites de baja seguridad. Cuando un usuario se registra, se envía una cookie con un único ID de usuario. Cuando el cliente se reconecta más tarde, se devuelve el ID de usuario, el servidor los busca, determina que pertenece a un usuario registrado, y no requiere explícitamente el nombre de usuario y la password.
- Personalizar una Site. Muchos "portales" nos permiten personalizar el aspecto de la página principal. Usan cookies para recordar lo que queremos, para que obtengamos el mismo resultado inicial la próxima vez.
- Publicidad enfocada. Los motores de búsqueda cobran más a sus clientes por mostrar anuncios "directos" que por anuncios "aleatorios". Es decir, si hacemos una búsqueda sobre "Java Servlets", un motor de búsqueda cobra más por un anuncio de un entorno de desarrollo de Servlets que por el de una agencia de viajes on-line. El problema es que tienen que enseñarnos un anuncio aleatorio la primera vez que llegamos y no hemos realizado la búsqueda, así como cuando buscamos algo que no corresponde con las categorías de anuncios. Los cookies les permiten recordar "Oh, esta es la persona que buscó por esto y esto anteriormente" y muestra un anuncio apropiado (lease "caro") en vez de uno aleatorio (lease "barato").

Ahora, añadir conveniencias al usuario y añadir valor al propietario de la site es el propósito que hay detrás de las cookies. Y no te creas todas las informaciones, las cookies no son un serio problema de seguridad. Las cookies nunca son interpretadas o ejecutadas de ninguna forma, y no pueden usarse para insertar virus o atacar nuestro sistema. Además, como los navegadores sólo aceptan 20 cookies por site y 300 cookies en total, cada cookie está limitada a 4Kb, las cookies no se pueden usar para llenar el disco duro o lanzar otros ataques de denegación de servicio.

Sin embargo, aunque no presentan un serio problema de seguridad, sí que presentan un significativo problema de privacidad. Primero, a algunas personas no les gusta que los motores de búsqueda

puedan recordar que ellos son las personas que usualmente buscan por uno u otro tópico. Incluso peor, dos motores de búsqueda pueden compartir datos sobre usuarios cargando pequeñas imágenes de una tercera parte que usa los cookies y comparte los datos con los dos motores de búsqueda. (Sin embargo, Netscape tiene una bonita característca que permite rechazar las cookies de otros sites distintos del que nos conectamos, pero sin desactivar las cookies totalmente). Este truco puede incluso ser explotado mediante email si usamos un programa de correo compatible HTML que soporte Cookies, como lo hace Outlook Express. Así, la gente podría enviar un email que cargue imágenes, adjuntar cookies a esas imágenes, y luego identificar nuestra dirección email y todo cuando vamos a su site.

O, una site que podría tener un nivel de seguridad muy superior al estándar podría permitir a los usuarios saltarse el nombre y la password mediante cookies. Por ejemplo, algunas de las grandes librerías on-line usan cookies para recordar a sus usuarios, y nos permite hacer pedidos sin tener que reintroducir nuestra información personal. Sin embargo, no muestran realmente el número completo de nuestra tarjeta de crédito, y sólo permiten enviar libros a la dirección que fue introducida cuando introducimos el número completo de la tarjeta de crédito o el nombre de usuario y la password. Y como resultado, alguien que use nuestro ordenador (o que robe el fichero cookie) lo único que podría hacer sería un enorme pedido de libros con nuestra tarjeta de crédito que nos llegaría a casa, donde podrían ser rechazados. Sin embargo, las pequeñas compañías no suelen ser tan cuidadosas, y el acceso de alguien a nuestro ordenador o al fichero de cookies puede resultar en una pérdida de información personal importante. Incluso peor, las sites incompetentes podrían embeber el propio número de tarjeta de crédito y otra información sensible directamente dentro de cookies, en vez de usar identificadores inocuos que sólo se enlazan con los usuarios reales en el servidor.

El punto de todo esto es doble. Primero, debido a los problemas reales de privacidad, algunos usuarios desactivan las cookies. Por eso, incluso aunque usemos cookies para dar valor añadido a nuestra site, nuestra site no debería depender de ellas. Segundo, como autor de Servlets que podemos usar cookies, no deberíamos confiar a los cookies información particularmente sensible, ya que el usuario podría correr el riesgo de que alguien accediera a su ordenador o a sus ficheros de cookies.

2. El API Servlet Cookie

Para enviar cookies al cliente, un servlet debería crear uno o más cookies con los nombres y valores apropiados mediante `new Cookie(name, value)` seleccionar cualquier atributo opcional mediante `cookie.setXxx`, y añadir los cookies a la cabecera de respuesta mediante `response.addCookie(cookie)`. Para leer cookies entrantes, llamamos `request.getCookies()`, que devuelve un array de objetos `Cookie`. En la mayoría de los casos, recorreremos el array hasta encontrar aquella cuyo nombre (`getName`) corresponda con el nombre que tenemos en mente. luego llamamos a `getValue` sobre ese `Cookie` para ver el valor asociado con ese nombre.

2.1 Crear Cookies

Un objeto [Cookie](#) se crea llamando al constructor `Cookie`, que toma dos strings: el nombre y el valor del cookie. Ni el nombre ni el valor deberían contener espacios en blanco y ninguno de estos caracteres:

[] () = , " / ? @ : ;

2.2 Leer y Especificar Atributos de Cookie

Antes de añadir el cookie a la cabecera saliente, podemos buscar o seleccionar atributos del cookie. Aquí hay un sumario:

`getComment/setComment`

Obtiene/Selecciona un comentario asociado con este cookie.

getDomain/setDomain

Obtiene/Selecciona el dominio al que se aplica el cookie. Podemos usar este método para instruir al navegador a que las devuelva a otros host en el mismo dominio. Observa que el dominio debe empezar por un punto (por ejemplo .prenhall.com), y debe contener dos puntos para dominios que no sean de países como .com, .edu, y .gov, y tres puntos para dominios de país como .co.uk y .edu.es.

getMaxAge/setMaxAge

Obtiene/Selecciona el tiempo (en segundos) que debe pasar hasta que expire el cookie. Si no lo seleccionamos, el cookie sólo vivirá durante la sesión actual (es decir, hasta que el usuario salga del navegador), y no será almacenado en disco.

getName/setName

Obtiene/Selecciona el nombre del cookie. El nombre y el valor son dos piezas con las que siempre tenemos que tener cuidado. Como el método getCookies de HttpServletRequest devuelve un array de objetos Cookie, es muy común recorrer el array hasta que tengamos un nombre particular, luego chequeamos el valor con getValue.

getPath/setPath

Obtiene/Selecciona el path al que se aplica este cookie. Si no especificamos esto el cookie se devuelve para todas las URLs del mismo directorio que la página actual, así como para todos sus subdirectorios. Este método puede usarse para especificar algo más general. Por ejemplo someCookie.setPath("/") especifica que todas las páginas del servidor deberían recibir el cookie. Observa que el path especificado debe incluir el directorio actual.

getSecure/setSecure

Obtiene/Selecciona el valor boolean indicando si el código sólo debería enviarse sobre conexiones encriptadas (SSL).

getValue/setValue

Obtiene/Selecciona el valor asociado con el cookie. En algunos casos el nombre se usa como una bandera booleana, y su valor es ignorado. (es decir, la existencia del nombre significa True).

getVersion/setVersion

Obtiene/Selecciona la versión del protocolo del cookie. La versión 0, por defecto, se adhiere a la especificación original Netscape. La versión 1, todavía no soportada muy ampliamente, se adhiere a la [RFC 2109](#).

2.3 Situar los Cookies en las cabeceras de respuesta

El cookie se añade a la cabecera de respuesta Set-Cookie por medio del método addCookie de HttpServletResponse. Aquí hay un ejemplo:

```
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);
```

2.4 Leer las Cookies desde el cliente

Para enviar el cookie al cliente, creamos un Cookie luego usamos addCookie para enviar una cabecera de respuesta HTTP Set-Cookie. Para leer los cookies desde el cliente, llamamos a getCookies sobre el HttpServletRequest. Esto devuelve un array de objetos Cookie correspondiendo con los valores que vinieron de la cabecera de petición HTTP Cookie. Una vez que tengamos este array, lo recorreremos, llamando al método getName sobre cada Cookie hasta encontrar uno que corresponda con el nombre que buscamos. Luego llamamos a getValue sobre la cookie correspondiente, haciendo algún precesamiento específico con el valor resultante.

3. Algunas utilidades menores de Cookies

Aquí tenemos algunas sencillas pero útiles utilidades para tratar con cookies.

3.1 Obtener el Valor de un Cookie con un Nombre Específico

Aquí tenemos una sección de [ServletUtilities.java](#) que simplifica ligeramente la recuperación del valor de una cookie dando un nombre de cookie, recorriendo el array de objetos Cookie disponible, devolviendo el valor de cualquier Cookie cuyo nombre corresponda con la entrada. Si no hay ningún nombre que corresponda, se devolverá el valor por defecto.

```
public static String getCookieValue(Cookie[] cookies,
                                    String cookieName,
                                    String defaultValue) {
    for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName()))
            return(cookie.getValue());
    }
    return(defaultValue);
}
```

3.2 LongLivedCookie.java (Descarga [el código fuente](#))

Aquí tenemos una pequeña clase que podemos usar en lugar de Cookie si queremos que la cookie persista automáticamente cuando el cliente salga de su navegador:

```
package hall;

import javax.servlet.http.*;

public class LongLivedCookie extends Cookie {
    public static final int SECONDS_PER_YEAR = 60*60*24*365;

    public LongLivedCookie(String name, String value) {
        super(name, value);
        setMaxAge( SECONDS_PER_YEAR );
    }
}
```

4. Ejemplo: Un Interface de Motor de Búsqueda Personalizado

Aquí tenemos una variante del ejemplo SearchEngines mostrado en [la página anterior](#). En esta versión, el inicio es generado dinámicamente en lugar de venir de un fichero HTML estático. Luego, el servlet que lee los parámetros y los reenvía al motor de búsqueda apropiado también devuelve cookies al cliente que lista estos valores. La siguiente vez que el cliente visita la site, se usan los valores del cookie para precargar los campos del formulario con las entradas usadas más recientemente.

4.1 SearchEnginesFrontEnd.java

Este servlet construye la página del formulario para el servlet del motor de búsqueda. A primera vista, se parece a la página vista en [la página HTML estática](#) presentada en [la página anterior](#). Sin embargo, aquí, se recuerdan los valor seleccionados en cookies (seleccionado por el servlet

CustomizedSearchEngines a los que está página envía los datos), por eso si el usuario vuelve a la misma página después de algún tiempo (incluso después de apagar el navegador), la página se inicializa con los valores de la búsqueda anterior.

También puedes descargar [el código fuente](#).

```
package hall;

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

public class SearchEnginesFrontEnd extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        Cookie[] cookies = request.getCookies();
        String searchString =
            ServletUtilities.getCookieValue(cookies,
                                           "searchString",
                                           "Java Programming");

        String numResults =
            ServletUtilities.getCookieValue(cookies,
                                           "numResults",
                                           "10");

        String searchEngine =
            ServletUtilities.getCookieValue(cookies,
                                           "searchEngine",
                                           "google");

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Searching the Web";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=#FDF5E6>\n" +
            "<H1 ALIGN=CENTER>Searching the Web</H1>\n" +
            "\n" +
            "<FORM ACTION=/servlet/hall.CustomizedSearchEngines>\n" +
            "<CENTER>\n" +
            "Search String:\n" +
            "<INPUT TYPE=TEXT NAME=searchString\n" +
            "      VALUE=" + searchString + "><BR>\n" +
            "Results to Show Per Page:\n" +
            "<INPUT TYPE=TEXT NAME=numResults\n" +
            "      VALUE=" + numResults + " SIZE=3><BR>\n" +
            "<INPUT TYPE=RADIO NAME=searchEngine\n" +
            "      VALUE=google\n" +
            checked("google", searchEngine) + ">\n" +
            "Google |\n" +
            "<INPUT TYPE=RADIO NAME=searchEngine\n" +
            "      VALUE=infoseek\n" +
            checked("infoseek", searchEngine) + ">\n" +
            "Infoseek |\n" +
            "<INPUT TYPE=RADIO NAME=searchEngine\n" +
            "      VALUE=lycos\n" +
            checked("lycos", searchEngine) + ">\n" +
```

```

        "Lycos |\n" +
        "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\" \n" +
        "        VALUE=\"hotbot\" \" +
        checked("hotbot", searchEngine) + ">\n" +
        "HotBot\n" +
        "<BR>\n" +
        "<INPUT TYPE=\"SUBMIT\" VALUE=\"Search\">\n" +
        "</CENTER>\n" +
        "</FORM>\n" +
        "\n" +
        "</BODY>\n" +
        "</HTML>\n");
    }

    private String checked(String name1, String name2) {
        if (name1.equals(name2))
            return(" CHECKED");
        else
            return("");
    }
}

```

4.2 CustomizedSearchEngines.java

El servlet anterior SearchEnginesFrontEnd envía sus datos al servlet CustomizedSearchEngines. En muchos aspectos, es igual que el servlet SearchEngines mostrado en [la página anterior](#). Sin embargo, además de construir una URL para un motor de búsqueda y enviar una redirección de respuesta al cliente, el servlet también envía cookies recordando los datos del usuario. Estos cookies, serán utilizados por el servlet para construir la página para inicializar las entradas del formulario HTML.

```

package hall;

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

/** A variation of the SearchEngine servlet that uses
 *  cookies to remember users choices. These values
 *  are then used by the SearchEngineFrontEnd servlet
 *  to create the form-based front end with these
 *  choices preset.
 *
 *
 *  Part of tutorial on servlets and JSP that appears at
 *  http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
 *  1999 Marty Hall; may be freely used or adapted.
 */

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

```

```

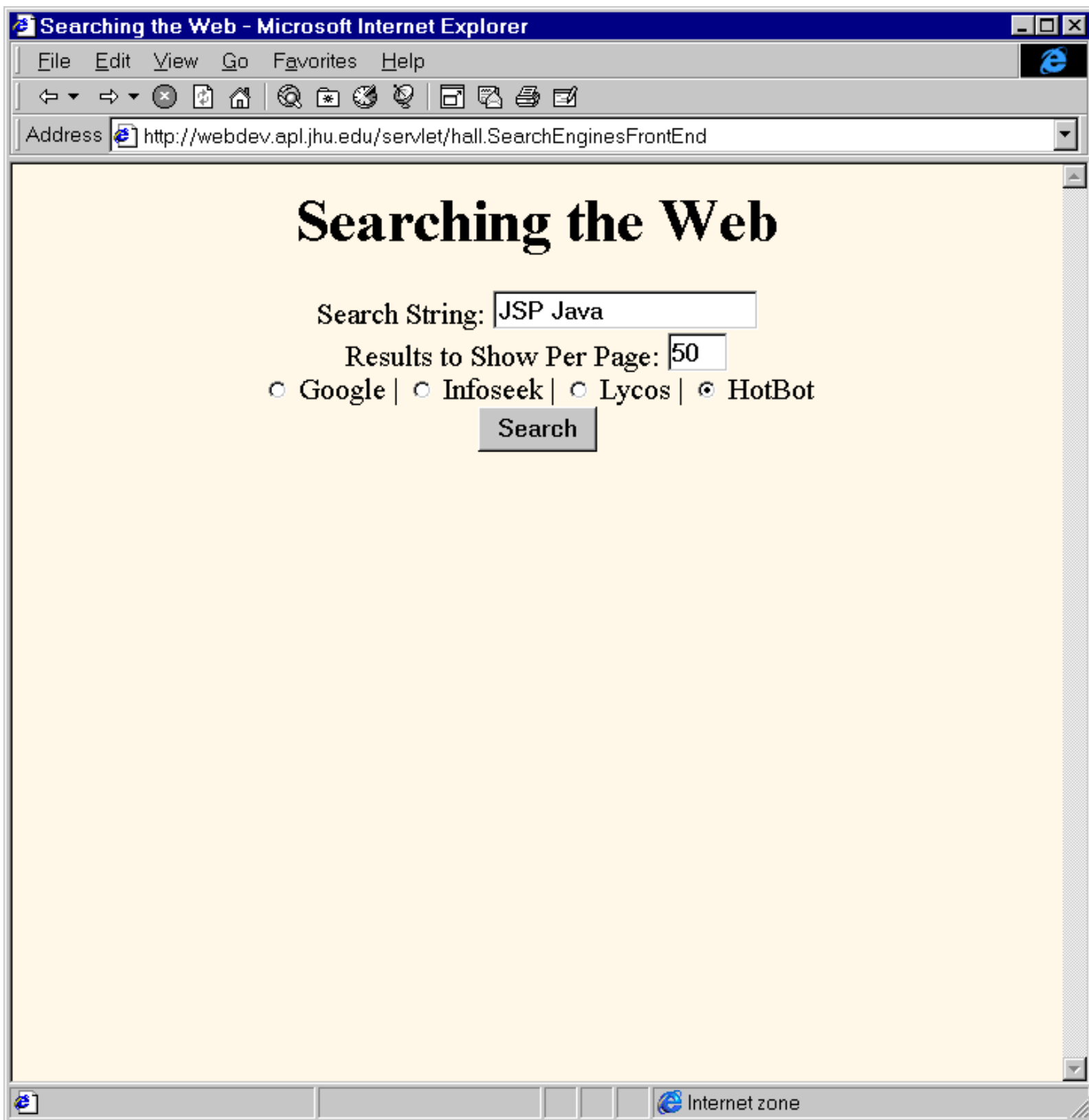
String searchString = request.getParameter("searchString");
Cookie searchStringCookie =
    new LongLivedCookie("searchString", searchString);
response.addCookie(searchStringCookie);
searchString = URLEncoder.encode(searchString);
String numResults = request.getParameter("numResults");
Cookie numResultsCookie =
    new LongLivedCookie("numResults", numResults);
response.addCookie(numResultsCookie);
String searchEngine = request.getParameter("searchEngine");
Cookie searchEngineCookie =
    new LongLivedCookie("searchEngine", searchEngine);
response.addCookie(searchEngineCookie);
SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
for(int i=0; i<commonSpecs.length; i++) {
    SearchSpec searchSpec = commonSpecs[i];
    if (searchSpec.getName().equals(searchEngine)) {
        String url =
            searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
    }
}
response.sendError(response.SC_NOT_FOUND,
    "No recognized search engine specified.");
}

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

```

4.3 Salidas de SearchEnginesFrontEnd

Aquí tenemos la vista de la página de nuestro buscador después de que el usuario haya vuelto a la misma página después de hacer una consulta:



4.4 CustomizedSearchEngines Output

HotBot results: JSP Java (1+) - Microsoft Internet Explorer

File Edit View Go Favorites Help

Address <http://www.hotbot.com/?MT=JSP+Java&DC=50>

Free Services EMAIL · HOMEPAGES · MESSAGE BOARDS · CHAT · PERSONAL GUIDE

Click here for Motorola

  **WIN** a pair of TalkAbouts! 

Results for **SEARCH** Search within these results ☐

[REVISE SEARCH](#)
[NEW SEARCH](#)
[ADVANCED SEARCH](#)

Search Resources
[Yellow Pages](#)
[White Pages](#)
[Email Addresses](#)
[Discussion Groups](#)
[Free Downloads](#)
[Research Service](#)
[Road Maps](#)
[Stock Quotes](#)
[Jobs & Résumés](#)
[News Headlines](#)
[FTP Search](#)

Shopping Resources
[Shopping Directory](#)
[Books](#)
[Hardware](#)
[Classifieds](#)
[Travel](#)
[New & Used Cars](#)

Search Partners

- Find books on "[JSP Java](#)" at [bn.com](#).
- Get information for "[JSP Java](#)" at [eHow.com](#).
- Research "[JSP Java](#)" at [AtHand Yellow Pages](#).

WEB RESULTS 1,320 Matches **1 - 50** [next](#) >>

Get the [Top 10 sites for "JSP Java"](#)

1. [java.sun.com - The Source for Java \(TM\) Technology](#)
Sun Microsystems' Java Technology Home Page - Visit this site to get the latest Java technologies, news, and products
99% 9/29/99 <http://java.sun.com/>
See results from [this site only](#).

2. [JSP -- Java Server Pages JSP Tutorial and JDBC Homepage](#)

 [CLICK TO SHOP](#)
[Fogdog Sports](#)

 [uBid Online Auctions](#)

 [Search thousands of loans](#)

Internet zone

Seguimiento de Sesión

1. [¿Qué es el Seguimiento de Sesión?](#)
 2. [El API Session Tracking](#)
 3. [Ejemplo: Mostrar Información de Sesión](#)
-

1. ¿Qué es el Seguimiento de Sesión?

Hay un número de problemas que vienen del hecho de que HTTP es un protocolo "sin estado". En particular, cuando estamos haciendo una compra on-line, es una molestia real que el servidor Web no puede recordar fácilmente transacciones anteriores. Esto hace que las aplicaciones como las cartas de compras sean muy problemáticas: cuando añadimos una entrada en nuestra carta, ¿cómo sabe el servidor que es realmente nuestra carta? Incluso si los servidores no retienen información contextual, todavía tendríamos problemas con comercio electrónico. Cuando nos movemos desde la página donde hemos especificado que queremos comprar (almacenada en un servidor Web normal) a la página que toma nuestro número de la tarjeta de crédito y la dirección de envío (almacenada en un servidor seguro que usa SSL), ¿cómo recuerda el servidor lo que hemos comprado?

Existen tres soluciones típicas a este problema:

1. Cookies. Podemos usar cookies HTTP para almacenar información sobre una sesión de compra, y cada conexión subsecuente puede buscar la sesión actual y luego extraer la información sobre esa sesión desde una localización en la máquina del servidor. Esta es una excelente alternativa, y es la aproximación más ampliamente utilizada. Sin embargo, aunque los servlets tienen un [Interface de alto nivel para usar cookies](#), existen unos tediosos detalles que necesitan ser controlados:
 - Extraer el cookie que almacena el identificador de sesión desde los otros cookies (puede haber muchos, después de todo),
 - Seleccionar un tiempo de expiración apropiado para el cookie (las sesiones interrumpidas durante 24 horas probablemente deberían ser reseteadas), y
 - Asociar la información en el servidor con el identificador de sesión (podría haber demasiada información que se almacena en el cookie, pero los datos sensibles como los números de las tarjetas de crédito nunca deben ir en cookies).
2. Reescribir la URL. Podemos añadir alguna información extra al final de cada URL que identifique la sesión, y el servidor puede asociar ese identificador de sesión con los datos que ha almacenado sobre la sesión. Esta también es una excelente solución, e incluso tiene la ventaja que funciona con navegadores que no soportan cookies o cuando el usuario las ha desactivado. Sin embargo, tiene casi los mismos problemas que los cookies, a saber, que los programas del lado del servidor tienen mucho proceso que hacer, pero tedioso. Además tenemos que ser muy cuidadosos con que cada URL que le devolvamos al usuario tiene añadida la información extra. Y si el usuario deja la sesión y vuelve mediante un bookmark o un enlace, la información de sesión puede perderse.
3. Campos de formulario ocultos. Los formularios HTML tienen una entrada que se

parece a esto: `<INPUT TYPE= "HIDDEN" NAME= "session" VALUE= "...">`. Esto significa que, cuando el formulario se envíe, el nombre y el valor especificado se incluirán en los datos GET o POST. Esto puede usarse para almacenar información sobre la sesión. Sin embargo, tiene la mayor desventaja en que sólo funciona si cada página se genera dinámicamente, ya que el punto negro es que cada sesión tiene un único identificador.

Los servlets proporcionan una solución técnica. Al API `HttpSession`. Este es un interface de alto nivel construido sobre las cookies y la reescritura de URL. De hecho, muchos servidores, usan cookies si el navegador las soporta, pero automáticamente se convierten a reescritura de URL cuando las cookies no son soportadas o están desactivadas. Pero el autor de servlets no necesita molestarse con muchos detalles, no tiene que manipular explícitamente las cookies o la información añadida a la URL, y se les da automáticamente un lugar conveniente para almacenar los datos asociados con cada sesión.

2. El API de Seguimiento de Sesión

Usar sesiones en servlets es bastante sencillo, envolver la búsqueda del objeto sesión asociado con la petición actual, crear un nuevo objeto sesión cuando sea necesario, buscar la información asociada con una sesión, almacenar la información de una sesión, y descartar las sesiones completas o abandonadas.

2.1 Buscar el objeto [HttpSession](#) asociado con la petición actual.

Esto se hace llamando al método `getSession` de `HttpServletRequest`. Si devuelve null, podemos crear una nueva sesión, pero es tan comunmente usado que hay una opción que crea automáticamente una nueva sesión si no existe una ya. Sólo pasamos `true` a `getSession`. Así, nuestro primer paso normalmente se parecerá a esto:

```
HttpSession session = request.getSession(true);
```

2.2 Buscar la Información Asociada con un Sesión.

Los objetos `HttpSession` viven en el servidor; son asociados automáticamente con el peticionario mediante un mecanismo detrás de la escena como los cookies o la reescritura de URL. Estos objetos sesión tienen una estructura de datos interna que nos permite almacenar un número de claves y valores asociados. En la versión 2.1 y anteriores del API servlet, usamos `getValue("key")` para buscar un valor previamente almacenado. El tipo de retorno es `Object`, por eso tenemos que forzarlo a un tipo más específico de datos. El valor de retorno es null si no existe dicho atributo. En la versión 2.2 `getValue` está obsoleto en favor de `getAttribute`, por el mejor nombrado correspondiente con `setAttribute` (el correspondiente para `getValue` es `putValue`, no `setValue`), y porque `setAttribute` nos permite usar un [HttpSessionBindingListener](#) asociado para monitorizar los valores, mientras que `putValue` no. Aquí tenemos un ejemplo representativo, asumiendo que `ShoppingCart` es alguna clase que hemos definido nosotros mismos y que almacena información de ítems para su venta

```
HttpSession session = request.getSession(true);
ShoppingCart previousItems =
    (ShoppingCart)session.getValue("previousItems");
if (previousItems != null) {
```

```

        doSomethingWith(previousItems);
    } else {
        previousItems = new ShoppingCart(...);
        doSomethingElseWith(previousItems);
    }
}

```

En la mayoría de los casos, tenemos un nombre atributo específico en mente, y queremos encontrar el valor (si existe) ya asociado con él. Sin embargo, también podemos descubrir todos los nombres de atributos en una sesión dada llamando a `getValueNames`, que devuelve un array de `String`. La versión 2.2, usa `getAttributeNames`, que tienen un nombre mejor y que es más consistente ya que devuelve una `Enumeration`, al igual que los métodos `getHeaders` y `getParameterNames` de `HttpServletRequest`.

Aunque los datos que fueron asociados explícitamente con una sesión son la parte en la que debemos tener más cuidado, hay otras partes de información que son muy útiles también.

- `getId`. Este método devuelve un identificador único generado para cada sesión. Algunas veces es usado como el nombre clave cuando hay un sólo valor asociado con una sesión, o cuando se uso la información de logging en sesiones anteriores.
- `isNew`. Esto devuelve `true` si el cliente (navegador) nunca ha visto la sesión, normalmente porque acaba de ser creada en vez de empezar una referencia a un petición de cliente entrante. Devuelve `false` para sesión preexistentes.
- `getCreationTime`. Devuelve la hora, en milisegundos desde 1970, en la que se creo la sesión. Para obtener un valor útil para impresión, pasamos el valor al constructor de `Date` o al método `setTimeInMillis` de `GregorianCalendar`.
- `getLastAccessedTime`. Esto devuelve la hora, en milisegundos desde 1970, en que la sesión fue enviada por última vez al cliente.
- `getMaxInactiveInterval`. Devuelve la cantidad de tiempo, en segundos, que la sesión debería seguir sin accesos antes de ser invalidada automáticamente. Un valor negativo indica que la sesión nunca se debe desactivar.

2.3 Asociar Información con una Sesión

Cómo se describió en la sección anterior, leemos la información asociada con una sesión usando `getValue` (o `getAttribute` en la versión 2.2 de las especificaciones Servlets). Observa que `putValue` reemplaza cualquier valor anterior. Algunas veces esto será lo que queremos pero otras veces queremos recuperar un valor anterior y aumentarlo. Aquí tenemos un ejemplo:

```

HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart previousItems =
    (ShoppingCart)session.getValue("previousItems");
if (previousItems == null) {
    previousItems = new ShoppingCart(...);
}
String itemID = request.getParameter("itemID");
previousItems.addEntry(Catalog.getEntry(itemID));
// You still have to do putValue, not just modify the cart, since
// the cart may be new and thus not already stored in the session.
session.putValue("previousItems", previousItems);

```

3. Ejemplo: Mostrar Información de Sesión

Aquí tenemos un sencillo ejemplo que genera una página Web mostrando alguna información sobre la sesión actual. También puedes [descargar el código fuente](#).

```
package hall;

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

/** Simple example of session tracking. See the shopping
 *  cart example for a more detailed one.
 *  <P>
 *  Part of tutorial on servlets and JSP that appears at
 *  http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/
 *  1999 Marty Hall; may be freely used or adapted.
 */

public class ShowSession extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        HttpSession session = request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Searching the Web";
        String heading;
        Integer accessCount = new Integer(0);
        if (session.isNew()) {
            heading = "Welcome, Newcomer";
        } else {
            heading = "Welcome Back";
            Integer oldAccessCount =
                // Use getAttribute, not getValue, in version
                // 2.2 of servlet API.
                (Integer)session.getValue("accessCount");
            if (oldAccessCount != null) {
                accessCount =
                    new Integer(oldAccessCount.intValue() + 1);
            }
        }
        // Use putAttribute in version 2.2 of servlet API.
        session.putValue("accessCount", accessCount);

        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
```

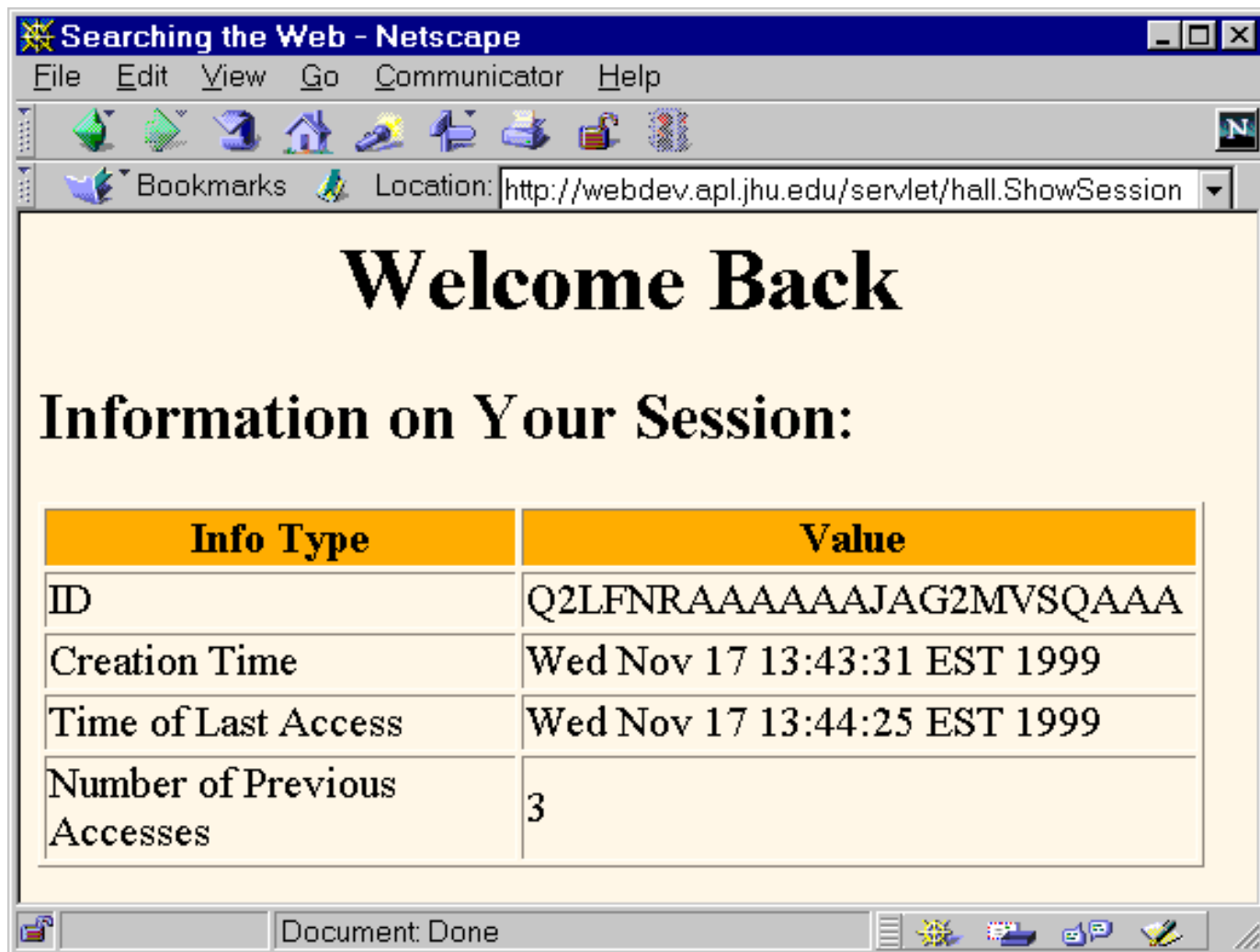
```

        "<H2>Information on Your Session:</H2>\n" +
        "<TABLE BORDER=1 ALIGN=CENTER>\n" +
        "<TR BGCOLOR=\"#FFAD00\">\n" +
        "    <TH>Info Type<TH>Value\n" +
        "<TR>\n" +
        "    <TD>ID\n" +
        "    <TD>" + session.getId() + "\n" +
        "<TR>\n" +
        "    <TD>Creation Time\n" +
        "    <TD>" + new Date(session.getCreationTime()) + "\n" +
        "<TR>\n" +
        "    <TD>Time of Last Access\n" +
        "    <TD>" + new Date(session.getLastAccessedTime()) + "\n" +
        "<TR>\n" +
        "    <TD>Number of Previous Accesses\n" +
        "    <TD>" + accessCount + "\n" +
        "</TABLE>\n" +
        "</BODY></HTML>" );
    }

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

```

Aquí tenemos un resultado típico, después de visitar la página varias veces sin salir del navegador entre medias:



JavaServer Pages (JSP) 1.0

1. [Introducción](#)
 2. [Sumario de sintaxis](#)
 3. [Plantilla de Texto \(HTML estático\)](#)
 4. [Elementos de Script JSP: Expresiones, Scriptlets, y Declaraciones](#)
 5. [Directivas JSP](#)
 6. [Ejemplo: usar Elementos de Script y Directivas](#)
 7. [Variables Predefinidas](#)
 8. [Acciones JSP](#)
 - [8.1 jsp:include](#)
 - [8.2 jsp:useBean \(introducción\)](#)
 - [8.3 jsp:useBean \(detalles\)](#)
 - [8.4 jsp:setProperty](#)
 - [8.5 jsp:getProperty](#)
 - [8.6 jsp:forward](#)
 - [8.7 jsp:plugin](#)
 9. [Convenciones sobre comentarios y caracteres de Escape en JSP](#)
-

1. Introducción

Las JavaServer Pages (JSP) nos permiten separar la parte dinámica de nuestras páginas Web del HTML estático. Simplemente escribimos el HTML regular de la forma normal, usando cualquier herramienta de construcción de páginas Web que usemos normalmente. Encerramos el código de las partes dinámicas en unas etiquetas especiales, la mayoría de las cuales empiezan con "< %" y terminan con "% >". Por ejemplo, aquí tenemos una sección de una página JSP que resulta en algo así como "Thanks for ordering Core Web Programming" para una URL como `http://host/OrderConfirmation.jsp?title= Core+ Web+ Programming:`

```
Thanks for ordering
<I><%= request.getParameter("title") %></I>
```

Normalmente daremos a nuestro fichero una extensión .jsp, y normalmente lo instalaremos en el mismo sitio que una página Web normal. Aunque lo que escribamos frecuentemente se parezca a un fichero HTML normal en vez de un servlet, detrás de la escena, la página JSP se convierte en un servlet normal, donde el HTML estático simplemente se imprime en el stream de salida estándar asociado con el método `service` del servlet. Esto normalmente sólo se hace la primera vez que se solicita la página, y los desarrolladores pueden solicitar la página ellos mismos cuando la instalan si quieren estar seguros de que el primer usuario real no tenga un retardo momentáneo cuando la página JSP sea traducida a un servlet y el servlet sea compilado y cargado. Observa también, que muchos servidores Web nos permiten definir alias para que una URL que parece apuntar a un fichero HTML realmente apunte a un servlet o a una página JSP.

Además de el HTML normal, hay tres tipos de construcciones JSP que embeberemos en una página:

elementos de script, directivas y acciones. Los elementos de script nos permiten especificar código Java que se convertirá en parte del servlet resultante, las directivas nos permiten controlar la estructura general del servlet, y las acciones nos permiten especificar componentes que deberían ser usados, y de otro modo controlar el comportamiento del motor JSP. Para simplificar los elementos de script, tenemos acceso a un número de variables predefinidas como request del fragmento de código anterior.

Nota: este tutorial cubre la versión 1.0 de la especificación JSP 1.0.

2. Sumario de Síntaxis

Elemento JSP	Síntaxis	Interpretación	Notas
Expresión JSP	<code><%= expression %></code>	La Expresión es evaluada y situada en la salida.	El equivalente XML es <jsp:expression> expression </jsp:expression> . Las variables predefinidas son request, response, out, session, application, config, y pageContext.
Scriptlet JSP	<code><% code %></code>	El código se inserta en el método service.	El equivalente XML es: <jsp:scriptlet> code </jsp:scriptlet> .
Declaración JSP	<code><%! code %></code>	El código se inserta en el cuerpo de la clase del servlet, fuera del método service.	El equivalente XML es: <jsp:declaration> code </jsp:declaration> .
Directiva page JSP	<code><%@ page att="val" %></code>	Dirige al motor servlet sobre la configuración general.	El equivalente XML es: <jsp:directive.page att="val"\> . Los atributos legales son (con los valores por defecto en negrita): <ul style="list-style-type: none">● import="package.class"● contentType="MIME-Type"● isThreadSafe="true false"● session="true false"● buffer="sizekb none"● autoflush="true false"● extends="package.class"● info="message"● errorPage="url"● isErrorPage="true false"● language="java"
Directiva include JSP	<code><%@ include file="url" %></code>	Un fichero del sistema local se incluirá cuando la página se traduzca a un Servlet.	El equivalente XML es: <jsp:directive.include file="url"\> . La URL debe ser relativa. Usamos la acción jsp:include para incluir un fichero en el momento de la petición en vez del momento de la traducción.
Comentario JSP	<code><%-- comment --%></code>	Comentario ignorado cuando se traduce la página JSP en un servlet.	Si queremos un comentario en el HTML resultante, usamos la sintaxis de comentario normal del HTML <-- comment --> .

Acción jsp:include	<code><jsp:include page="relative URL" flush="true"/></code>	Incluye un fichero en el momento en que la página es solicitada.	Aviso: en algunos servidores, el fichero incluido debe ser un fichero HTML o JSP, según determine el servidor (normalmente basado en la extensión del fichero).
Acción jsp:useBean	<code><jsp:useBean att=val*/> o <jsp:useBean att=val*> ... </jsp:useBean></code>	Encuentra o construye un Java Bean.	Los posibles atributos son: <ul style="list-style-type: none"> ● id= "name" ● scope= "page request session application" ● class= "package.class" ● type= "package.class" ● beanName= "package.class"
Acción jsp:setProperty	<code><jsp:setProperty att=val*/></code>	Selecciona las propiedades del bean, bien directamente o designando el valor que viene desde un parámetro de la petición.	Los atributos legales son: <ul style="list-style-type: none"> ● name= "beanName" ● property= "propertyName *" ● param= "parameterName" ● value= "val"
Acción jsp:getProperty	<code><jsp:getProperty name="propertyName" value="val"/></code>	Recupera y saca las propiedades del Bean.	
Acción jsp:forward	<code><jsp:forward page="relative URL"/></code>	Reenvía la petición a otra página.	
Acción jsp:plugin	<code><jsp:plugin attribute="value"*> ... </jsp:plugin></code>	Genera etiquetas OBJECT o EMBED, apropiadas al tipo de navegador, pidiendo que se ejecute un applet usando el Java Plugin.	

3. Plantilla de Texto: HTML estático

En muchos casos, un gran porcentaje de nuestras páginas JSP consistirá en HTML estático, conocido como plantilla de texto. En casi todos los aspectos, este HTML se parece al HTML normal, sigue las mismas reglas de sintaxis, y simplemente "pasa a través" del cliente por el servlet creado para manejar la página. No sólo el aspecto del HTML es normal, puede ser creado con cualquier herramienta que usemos para generar páginas Web. Por ejemplo, yo he usado Homesite de Allaire, para la mayoría de las páginas de este tutorial.

La única excepción a la regla de que "la plantilla de texto se pasa tal y como es" es que, si queremos tener "< %" en la salida, necesitamos poner "< \% " en la plantilla de texto.

4. Elementos de Script JSP

Los elementos de script nos permiten insertar código Java dentro del servlet que se generará desde la página JSP actual. Hay tres formas:

1. Expresiones de la forma `< % = expresión % >` que son evaluadas e insertadas en la salida.
2. Scriptlets de la forma `< % código % >` que se insertan dentro del método service del servlet, y
3. Declaraciones de la forma `< % ! código % >` que se insertan en el cuerpo de la clase del servlet, fuera de cualquier método existente.

4.1 Expresiones JSP

Una expresión JSP se usa para insertar valores Java directamente en la salida. Tiene la siguiente forma:

```
<%= expresión Java %>
```

La expresión Java es evaluada, convertida a un string, e insertada en la página. Esta evaluación se realiza durante la ejecución (cuando se solicita la página) y así tiene total acceso a la información sobre la solicitud. Por ejemplo, esto muestra la fecha y hora en que se solicitó la página:

```
Current time: <%= new java.util.Date() %>
```

Para simplificar estas expresiones, hay un gran número de variables predefinidas que podemos usar. Estos objetos implícitos se describen más adelante con más detalle, pero para el propósito de las expresiones, los más importantes son:

- request, el `HttpServletRequest`;
- response, el `HttpServletResponse`;
- session, el `HttpSession` asociado con el request (si existe), y
- out, el `PrintWriter` (una versión con buffer del tipo `JspWriter`) usada para enviar la salida al cliente.

Aquí tenemos un ejemplo:

```
Your hostname: <%= request.getRemoteHost() %>
```

Finalmente, observa que los autores de XML pueden usar una sintaxis alternativa para las expresiones JSP:

```
<jsp:expression>  
Expresión Java  
</jsp:expression>
```

Recuerda que los elementos XML, al contrario que los del HTML, son sensibles a las mayúsculas; por eso asegúrate de usar minúsculas.

4.2 Scriptlets JSP

Si queremos hacer algo más complejo que insertar una simple expresión, los scriptlets JSP nos permiten insertar código arbitrario dentro del método servlet que será construido al generar la página. Los Scriptlets tienen la siguiente forma:

```
<% Código Java %>
```

Los Scriptlets tienen acceso a las mismas variables predefinidas que las expresiones. Por eso, por ejemplo, si queremos que la salida aparezca en la página resultante, tenemos que usar la variable out:

```
<%  
String queryData = request.getQueryString();  
out.println("Attached GET data: " + queryData);  
%>
```

Observa que el código dentro de un scriptlet se insertará exactamente como está escrito, y cualquier HTML estático (plantilla de texto) anterior o posterior al scriptlet se convierte en sentencias print. Esto significa que los scriptlets no necesitan completar las sentencias Java, y los bloques abiertos pueden afectar al HTML estático fuera de los scriptlets. Por ejemplo, el siguiente fragmento JSP, contiene una mezcla de texto y scriptlets:

```
<% if (Math.random() < 0.5) { %>  
Have a <B>nice</B> day!  
<% } else { %>  
Have a <B>lousy</B> day!  
<% } %>
```

que se convertirá en algo como esto:

```
if (Math.random() < 0.5) {  
    out.println("Have a <B>nice</B> day!");  
} else {  
    out.println("Have a <B>lousy</B> day!");  
}
```

Si queremos usar los caracteres "% > " dentro de un scriptlet, debemos poner "% \> ". Finalmente, observa que el equivalente XML de <% Código %> es

```
<jsp:scriptlet>
Código
</jsp:scriptlet>
```

4.3 Declaraciones JSP

Una declaration JSP nos permite definir métodos o campos que serán insertados dentro del cuerpo principal de la clase servlet (fuera del método service que procesa la petición). Tienen la siguiente forma:

```
<%! Código Java%>
```

Como las declaraciones no generan ninguna salida, normalmente se usan en conjunción con expresiones JSP o scriptlets. Por ejemplo, aquí tenemos un fragmento de JSP que imprime el número de veces que se ha solicitado la página actual desde que el servidor se arrancó (o la clase del servlet se modificó o se recargó):

```
<%! private int accessCount = 0; %>
Acceses to page since server reboot:
<%= ++accessCount %>
```

Como con los scriptlet, si queremos usar los caracteres "% > ", ponemos "% \> ". Finalmente, observa que el equivalente XML de <%! Código %> es:

```
<jsp:declaration>
Código
</jsp:declaration>
```

5. Directivas JSP

Una directiva JSP afecta a la estructura general de la clase servlet. Normalmente tienen la siguiente forma:

```
<%@ directive attribute="value" %>
```

Sin embargo, también podemos combinar múltiples selecciones de atributos para una sola directiva, de esta forma:

```
<%@ directive attribute1="value1"
      attribute2="value2"
      ...
      attributeN="valueN" %>
```

Hay dos tipos principales de directivas: page, que nos permite hacer cosas como importar clases, personalizar la superclase del servlet, etc. e include, que nos permite insertar un fichero dentro de la clase servlet en el momento que el fichero JSP es traducido a un servlet. La especificación también menciona la directiva taglib, que no está soportada en JSP 1.0, pero se pretende que permita que los autores de JSP definan sus propias etiquetas. Se espera que sea una de las principales contribuciones a JSP 1.1.

5.1 La directiva page

La directiva page nos permite definir uno o más de los siguientes atributos sensibles a las mayúsculas:

- import= "**package.class**" o import= "**package.class1,...,package.classN**".
Esto nos permite especificar los paquetes que deberían ser importados. Por ejemplo:
<% @ page import= "java.util.*" %>
El atributo import es el único que puede aparecer múltiples veces.
- contentType= "**MIME-Type**" o
contentType= "**MIME-Type**; charset= **Character-Set**"
Esto especifica el tipo MIME de la salida. El valor por defecto es text/html. Por ejemplo, la directiva:
<% @ page contentType= "text/plain" %>
tiene el mismo valor que el scriptlet
<% response.setContentType("text/plain"); %>
- isThreadSafe= "true|false".
Un valor de true (por defecto) indica un procesamiento del servlet normal, donde múltiples peticiones pueden procesarse simultáneamente con un sólo ejemplar del servlet, bajo la suposición que del autor sincroniza las

variables de ejemplar. Un valor de false indica que el servlet debería implementar SingleThreadModel, con peticiones enviadas serialmente o con peticiones simultáneas siendo entregadas por ejemplares separados del servlet.

- session= "true|false".
Un valor de true (por defecto) indica que la variable predefinida session (del tipo HttpSession) debería unirse a la sesión existente si existe una, si no existe se debería crear una nueva sesión para unirla. Un valor de false indica que no se usarán sesiones, y los intentos de acceder a la variable session resultarán en errores en el momento en que la página JSP sea traducida a un servlet.
- buffer= "**size**kb|none".
Esto especifica el tamaño del buffer para el JspWriter out. El valor por defecto es específico del servidor, debería ser de al menos 8kb.
- autoflush= "true|false".
Un valor de true (por defecto) indica que el buffer debería desacargarse cuando esté lleno. Un valor de false, raramente utilizado, indica que se debe lanzar una excepción cuando el buffer se sobrecargue. Un valor de false es ilegal cuando usamos buffer= "none".
- extends= "**package.class**".
Esto indica la superclase del servlet que se va a generar. Debemos usarla con extrema precaución, ya que el servidor podría utilizar una superclase personalizada.
- info= "**message**".
Define un string que puede usarse para ser recuperado mediante el método getServletInfo.
- errorPage= "**url**".
Especifica una página JSP que se debería procesar si se lanzará cualquier Throwable pero no fuera capturado en la página actual.
- isErrorPage= "true|false".
Indica si la página actual actúa o no como página de error de otra página JSP. El valor por defecto es false.
- language= "java".
En algunos momentos, esto está pensado para especificar el lenguaje a utilizar. Por ahora, no debemos preocuparnos por él ya que java es tanto el valor por defecto como la única opción legal.

La sintaxis XML para definir directivas es:

```
<jsp:directive.TipoDirectiva atributo=valor />
```

Por ejemplo, el equivalente XML de:

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

es:

```
<jsp:directive.page import="java.util.*" />
```

5.2 La directiva include JSP

Esta directiva nos permite incluir ficheros en el momento en que la página JSP es traducida a un servlet. La directiva se parece a esto::

```
<%@ include file="url relativa" %>
```

La URL especificada normalmente se interpreta como relativa a la página JSP a la que se refiere, pero, al igual que las URLs relativas en general, podemos decirle al sistema que interpreta la URL relativa al directorio home del servidor Web empezando la URL con una barra invertida. Los contenidos del fichero incluido son analizados como texto normal JSP, y así pueden incluir HTML estático, elementos de script, directivas y acciones.

Por ejemplo, muchas sites incluyen una pequeña barra de navegación en cada página. Debido a los problemas con los marcos HTML, esto normalmente se implementa mediante una pequeña tabla que cruza la parte superior de la página o el lado izquierdo, con el HTML repetido para cada página de la site. La directiva include es una forma natural de hacer esto, ahorrando a los desarrolladores el mantenimiento engorroso de copiar realmente el HTML en cada fichero separado. Aquí tenemos un código representativo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Servlet Tutorial: JavaServer Pages (JSP) 1.0</TITLE>
<META NAME="author" CONTENT="webmaster@somesite.com">
<META NAME="keywords" CONTENT="...">
```

```

<META NAME="description" CONTENT="...">
<LINK REL=STYLESHEET
      HREF="Site-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<%@ include file="/navbar.html" %>

<!-- Part specific to this page ... -->

</BODY>
</HTML>

```

Observa que como la directiva `include` inserta los ficheros en el momento en que la página es traducida, si la barra de navegación cambia, necesitamos re-traducir todas las páginas JSP que la refieren. Esto es un buen compromiso en una situación como esta, ya que las barras de navegación no cambian frecuentemente, y queremos que el proceso de inclusión sea tan eficiente como sea posible. Si, sin embargo, los ficheros incluidos cambian de forma más frecuente, podríamos usar la acción `jsp:include` en su lugar. Esto incluye el fichero en el momento en que se solicita la página JSP, como se describe en la [sección 8](#).

6. Ejemplo: Usar Elementos de Script y Directivas

Aquí tenemos un sencillo ejemplo que muestra el uso de expresiones, scriptlets, declaraciones y directivas JSP. También puedes [descargar el código fuente](#)

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Using JavaServer Pages</TITLE>

<META NAME="author" CONTENT="Marty Hall -- hall@apl.jhu.edu">
<META NAME="keywords"
      CONTENT="JSP,JavaServer Pages,servlets">
<META NAME="description"
      CONTENT="A quick example of the four main JSP tags.">
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
    Using JavaServer Pages</TH></TR></TABLE>
</CENTER>
<P>

Some dynamic content created using various JSP mechanisms:
<UL>
  <LI><B>Expression.</B><BR>
    Your hostname: <%= request.getRemoteHost() %>.
  <LI><B>Scriptlet.</B><BR>
    <% out.println("Attached GET data: " +
      request.getQueryString()); %>
  <LI><B>Declaration (plus expression).</B><BR>
    <%! private int accessCount = 0; %>
    Accesses to page since server reboot: <%= ++accessCount %>
  <LI><B>Directive (plus expression).</B><BR>
    <%@ page import = "java.util.*" %>
    Current date: <%= new Date() %>
</UL>

```

```
</BODY>  
</HTML>
```

Aquí tenemos un resultado típico:



7. Variables Predefinidas

Para simplificar el código en expresiones y scriptlets JSP, tenemos ocho variables definidas automáticamente, algunas veces llamadas objetos implícitos. Las variables disponibles son: request, response, out, session, application, config, pageContext, y page.

7.1 request

Este es el HttpServletRequest asociado con la petición, y nos permite mirar los parámetros de la petición (mediante getParameter), el tipo de petición (GET, POST, HEAD, etc.), y las cabeceras HTTP entrantes (cookies, Referer, etc.). Estrictamente hablando, se permite que la petición sea una subclase de ServletRequest distinta de HttpServletRequest, si el protocolo de la petición es distinto del HTTP. Esto casi nunca se lleva a la práctica.

7.2 response

Este es el HttpServletResponse asociado con la respuesta al cliente. Observa que, como el stream de salida (ver out más abajo) tiene un buffer, es legal seleccionar los códigos de estado y cabeceras de respuesta, aunque no está permitido en los servlets normales una vez que la salida ha sido enviada al cliente.

7.3 out

Este es el PrintWriter usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto response (ver la sección anterior), esta es una versión con buffer de PrintWriter llamada JspWriter. Observa que podemos ajustar el tamaño del buffer, o incluso desactivar el buffer, usando el atributo buffer de la directiva page. Esto se explicó en la [Sección 5](#). También observa que out se usa casi exclusivamente en scriptlets ya que las expresiones JSP

obtienen un lugar en el stream de salida, y por eso raramente se refieren explícitamente a out.

7.4 session

Este es el objeto HttpSession asociado con la petición. Recuerda que las sesiones se crean automáticamente, por esto esta variable se une incluso si no hubiera una sesión de referencia entrante. La única excepción es usar el atributo session de la directiva page (ver la [Sección 5](#)) para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable session causarán un error en el momento de traducir la página JSP a un servlet.

7.5 application

Este es el ServletContext obtenido mediante `getServletConfig().getContext()`.

7.6 config

Este es el objeto ServletConfig para esta página.

7.7 pageContext

JSP presenta una nueva clase llamada PageContext para encapsular características de uso específicas del servidor como JspWriters de alto rendimiento. La idea es que, si tenemos acceso a ellas a través de esta clase en vez directamente, nuestro código seguirá funcionando en motores servlet/JSP "normales".

7.8 page

Esto es sólo un sinónimo de this, y no es muy útil en Java. Fue creado como situación para el día que el los lenguajes de script puedan incluir otros lenguajes distintos de Java.

8. Accciones

Las acciones JSP usan construcciones de sintaxis XML para controlar el comportamiento del motor de Servlets. Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, o generar HTML para el plug-in Java. Las acciones disponibles incluyen:

- `jsp:include` - Incluye un fichero en el momento de petición de esta página. Ver la [Sección 8.1](#).
- `jsp:useBean` - Encuentra o ejemplariza un JavaBean. Ver la [Sección 8.2](#) para una introducción, y la [Sección 8.3](#) para los detalles.
- `jsp:setProperty` - Selecciona la propiedad de un JavaBean. Ver la [Sección 8.4](#).
- `jsp:getProperty` - Inserta la propiedad de un JavaBean en la salida. Ver la [Sección 8.5](#).
- `jsp:forward` - Reenvía al peticionario a una nueva página. Ver la [Sección 8.6](#).
- `jsp:plugin` - Genera código específico del navegador que crea una etiqueta OBJECT o EMBED para el plug-in Java. Ver la [Sección 8.7](#).

Recuerda que, como en XML, los nombre de elementos y atributos son sensibles a las mayúsculas.

8.1 Acción jsp:include

Esta acción nos permite insertar ficheros en una página que está siendo generada. La sintaxis se parece a esto:

```
<jsp:include page="relative URL" flush="true" />
```

Al contrario que la [directiva include](#), que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada. Esto se paga un poco en la eficiencia, e imposibilita a la página incluida de contener código JSP general (no puede seleccionar cabeceras HTTP, por ejemplo), pero se obtiene una significativa flexibilidad. Por ejemplo, aquí tenemos una página JSP que inserta cuatro puntos diferentes dentro de una página Web "What's New?". Cada vez que cambian las líneas de cabeceras, los autores sólo tienen que actualizar los cuatro ficheros, pero pueden dejar como estaba la página JSP principal.

WhatsNew.jsp

También puedes [descargar el código fuente](#).

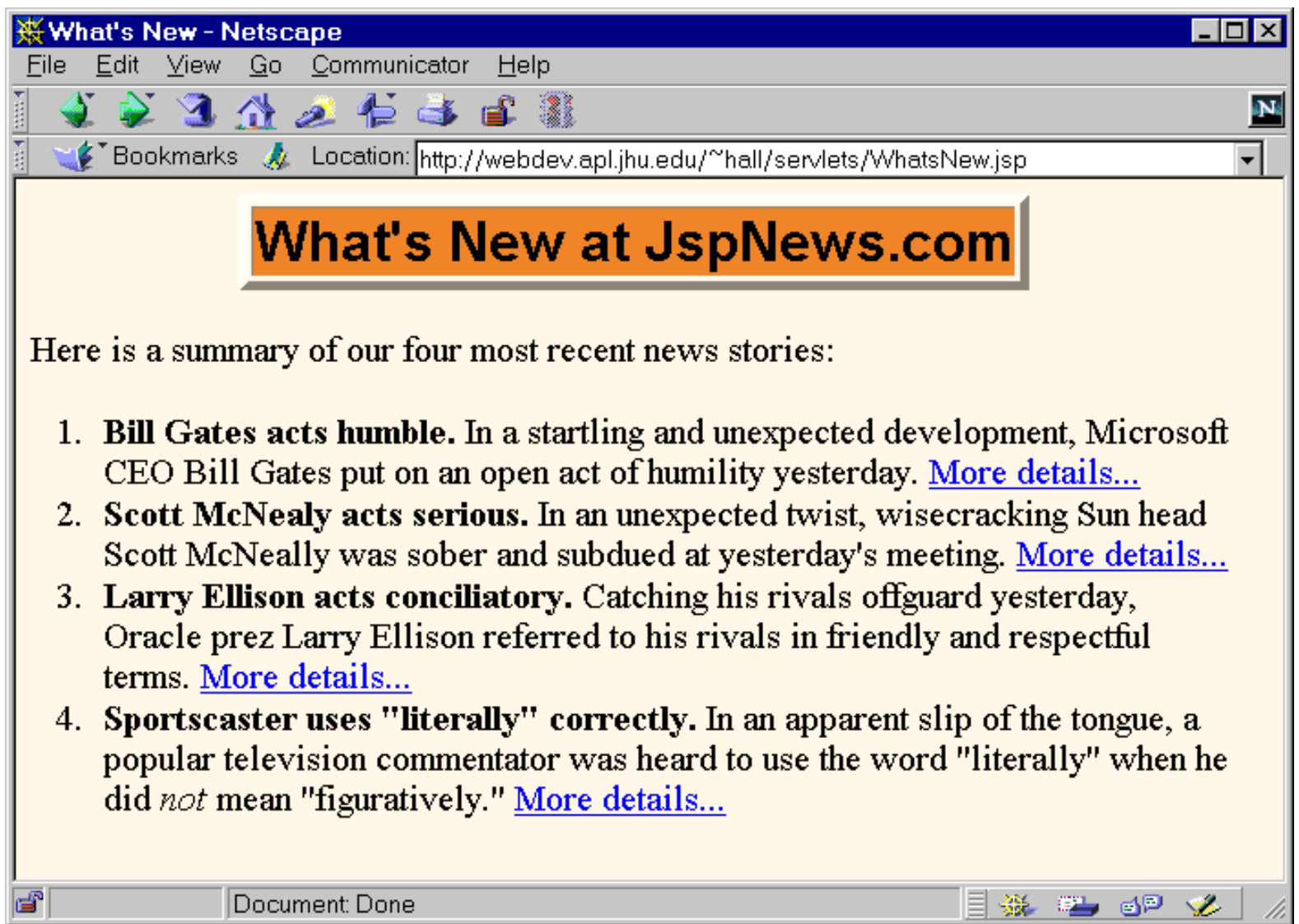
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>What's New</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY BGCOLOR="#FDF5E6" TEXT="#000000" LINK="#0000EE"
      VLINK="#551A8B" ALINK="#FF0000">

<CENTER>
<TABLE BORDER=5 BGCOLOR="#EF8429">
  <TR><TH CLASS="TITLE">
    What's New at JspNews.com</TABLE>
</CENTER>
<P>

Here is a summary of our four most recent news stories:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true"/>
  <LI><jsp:include page="news/Item2.html" flush="true"/>
  <LI><jsp:include page="news/Item3.html" flush="true"/>
  <LI><jsp:include page="news/Item4.html" flush="true"/>
</OL>
</BODY>
</HTML>
```

Aquí tenemos un resultado típico:



8.2 Acción jsp:useBean

Esta acción nos permite cargar y utilizar un JavaBean en la página JSP. Esta es una capacidad muy útil porque nos permite utilizar la reusabilidad de las clases Java sin sacrificar la conveniencia de añadir JSP sobre servlets solitarios. El sintaxis más simple para especificar que se debería usar un Bean es:

```
<jsp:useBean id="name" class="package.class" />
```

Esto normalmente significa "ejemplariza un objeto de la clase especificada por class, y unelo a una variable con el nombre especificado por id". Sin embargo, también podemos especificar un atributo scope que hace que ese Bean se asocie con más de una sola página. En este caso, es útil obtener referencias a los beans existentes, y la acción jsp:useBean especifica que se ejemplarizará un nuevo objeto si no existe uno con el mismo nombre y ámbito.

Ahora, una vez que tenemos un bean, podemos modificar sus propiedades mediante jsp:setProperty, o usando un scriptlet y llamando a un método explícitamente sobre el objeto con el nombre de la variable especificada anteriormente mediante el atributo id. Recuerda que con los beans, cuando decimos "este bean tiene una propiedad del tipo X llamada foo", realmente queremos decir "Esta clase tiene un método getFoo que devuelve algo del tipo X, y otro método llamado setFoo que toma un X como un argumento". La acción jsp:setProperty se describe con más detalle en la siguiente sección, pero ahora observemos que podemos suministrar un valor explícito, dando un atributo param para decir que el valor está derivado del parámetro de la petición nombrado, o sólo lista las propiedades para indicar que el valor debería derivarse de los parámetros de la petición con el mismo nombre que la propiedad. Leemos las propiedades existentes en una expresión o scriptlet JSP llamando al método **getXxx**, o más comunmente, usando la acción jsp:getProperty.

Observa que la clase especificada por el bean debe estar en el path normal del servidor, no en la parte reservada que obtiene la recarga automática cuando se modifican. Por ejemplo, en el Java Web Server, él y todas las clases que usa deben ir en el directorio classes o estar en un fichero JAR en el directorio lib, no en el directorio servlets.

Aquí tenemos un ejemplo muy sencillo que carga un bean y selecciona y obtiene un sencillo parámetro String.

BeanTest.jsp

También puedes [descargar el código fuente](#).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
    Reusing JavaBeans in JSP</TABLE>
</CENTER>
<P>

<jsp:useBean id="test" class="hall.SimpleBean" />
<jsp:setProperty name="test"
                  property="message"
                  value="Hello WWW" />

<H1>Message: <I>
<jsp:getProperty name="test" property="message" />
</I></H1>

</BODY>
</HTML>
```

SimpleBean.java

Aquí está el código fuente usado para el Bean usado en la página BeanTest. También puedes [descargar el código fuente](#).

```
package hall;

public class SimpleBean {
  private String message = "No message specified";

  public String getMessage() {
    return(message);
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

Aquí tenemos un resultado típico:



8.3 Más detalles de jsp:useBean

La forma más sencilla de usar un Bean es usar:

```
<jsp:useBean id="name" class="package.class" />
```

para cargar el Bean, luego usar `jsp:setProperty` y `jsp:getProperty` para modificar y recuperar propiedades del bean. Sin embargo, tenemos dos opciones. Primero, podemos usar un formato de contenedor, llamado:

```
<jsp:useBean ...>  
Body  
</jsp:useBean>
```

Para indicar que la porción Body sólo se debería ejecutar cuando el bean es ejemplarizado por primera vez, no cuando un bean existente se encuentre y se utilice. Como se explica abajo, los bean pueden ser compartidos, por eso no todas las sentencias `jsp:useBean` resultan en la ejemplarización de un Bean. Segundo, además de `id` y `class`, hay otros tres atributos que podemos usar: `scope`, `type`, y `beanName`.

Atributo	Uso
id	Da un nombre a la variable que referenciará el bean. Se usará un objeto bean anterior en lugar de ejemplarizar uno nuevo si se puede encontrar uno con el mismo id y scope.
class	Designa el nombre completo del paquete del bean.
scope	Indica el contexto en el que el bean debería estar disponible. Hay cuatro posibles valores: <code>page</code> , <code>request</code> , <code>session</code> , y <code>application</code> . El valor por defecto, <code>page</code> , indica que el bean estará sólo disponible para la página actual (almacenado en el <code>PageContext</code> de la página actual). Un valor de <code>request</code> indica que el bean sólo está disponible para la petición actual del cliente (almacenado en el objeto <code>ServletRequest</code>). Un valor de <code>session</code> indica que el objeto está disponible para todas las páginas durante el tiempo de vida de la <code>HttpSession</code> actual. Finalmente, un valor de <code>application</code> indica que está disponible para todas las páginas que compartan el mismo <code>ServletContext</code> . La razón de la importancia del ámbito es que una entrada <code>jsp:useBean</code> sólo resultará en la ejemplarización de un nuevo objeto si no había objetos anteriores con el mismo id y scope. De otra forma, se usarán los objetos existentes, y cualquier elemento <code>jsp:setParameter</code> u otras entradas entre las etiquetas de inicio <code>jsp:useBean</code> y la etiqueta de final, serán ignoradas.
type	Especifica el tipo de la variable a la que se referirá el objeto. Este debe corresponder con el nombre de la clase o ser una superclase o un interface que implemente la clase. Recuerda que el nombre de la variable se designa mediante el atributo id.

beanName	Da el nombre del bean, como lo suministraríamos en el método instantiate de Beans. Esta permitido suministrar un type y un beanName, y omitir el atributo class.
----------	--

8.4 Acción jsp:setProperty

Usamos jsp:setProperty para obtener valores de propiedades de los beans que se han referenciado anteriormente. Podemos hacer esto en dos contextos. Primero, podemos usar antes jsp:setProperty, pero fuera de un elemento jsp:useBean, de esta forma:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
                 property="someProperty" ... />
```

En este caso, el jsp:setProperty se ejecuta sin importar si se ha ejemplarizado un nuevo bean o se ha encontrado uno ya existente. Un segundo contexto en el que jsp:setProperty puede aparecer dentro del cuerpo de un elemento jsp:useBean, de esta forma:

```
<jsp:useBean id="myName" ... >
...
  <jsp:setProperty name="myName"
                  property="someProperty" ... />
</jsp:useBean>
```

Aquí, el jsp:setProperty sólo se ejecuta si se ha ejemplarizado un nuevo objeto, no si se encontró uno ya existente.

Aquí tenemos los cuatro atributos posibles de jsp:setProperty:

Atributo	Uso
name	Este atributo requerido designa el bean cuya propiedad va a ser seleccionada. El elemento jsp:useBean debe aparecer antes del elemento jsp:setProperty.
property	Este atributo requerido indica la propiedad que queremos seleccionar. Sin embargo, hay un caso especial: un valor de "*" significa que todos los parámetros de la petición cuyos nombres correspondan con nombres de propiedades del Bean serán pasados a los métodos de selección apropiados.
value	Este atributo opcional especifica el valor para la propiedad. Los valores string son convertidos automáticamente a números, boolean, Boolean, byte, Byte, char, y Character mediante el método estándar valueOf en la fuente o la clase envolvente. Por ejemplo, un valor de "true" para una propiedad boolean o Boolean será convertido mediante Boolean.valueOf, y un valor de "42" para una propiedad int o Integer será convertido con Integer.valueOf. No podemos usar value y param juntos, pero si está permitido no usar ninguna.
param	<p>Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición actual no tiene dicho parámetro, no se hace nada: el sistema no pasa null al método seleccionador de la propiedad. Así, podemos dejar que el bean suministre los valores por defecto, sobrescribiendolos sólo cuando el parámetro dice que lo haga. Por ejemplo, el siguiente código dice "selecciona el valor de la propiedad numberOfItems a cualquier valor que tenga el parámetro numItems de la petición, si existe dicho parámetro, si no existe no se hace nada"</p> <pre><jsp:setProperty name="orderBean" property="numberOfItems" param="numItems" /></pre> <p>Si omitimos tanto value como param, es lo mismo que si suministramos un nombre de parámetro que corresponde con el nombre de una propiedad. Podremos tomar esta idea de automaticidad usando el parámetro de la petición cuyo nombre corresponde con la propiedad suministrada un nombre de propiedad de "*" y omitir tanto value como param. En este caso, el servidor itera sobre las propiedades disponibles y los parámetros de la petición, correspondiendo aquellas con nombres idénticos.</p>

Aquí tenemos un ejemplo que usa un bean para crear una tabla de números primos. Si hay un parámetro llamado numDigits en los datos de la petición, se pasa dentro del bean a la propiedad numDigits. Al igual que en numPrimes.

JspPrimes.jsp

Para descargar el código JSP, pulsa con el botón derecho sobre [el enlace](#).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD>

<BODY>

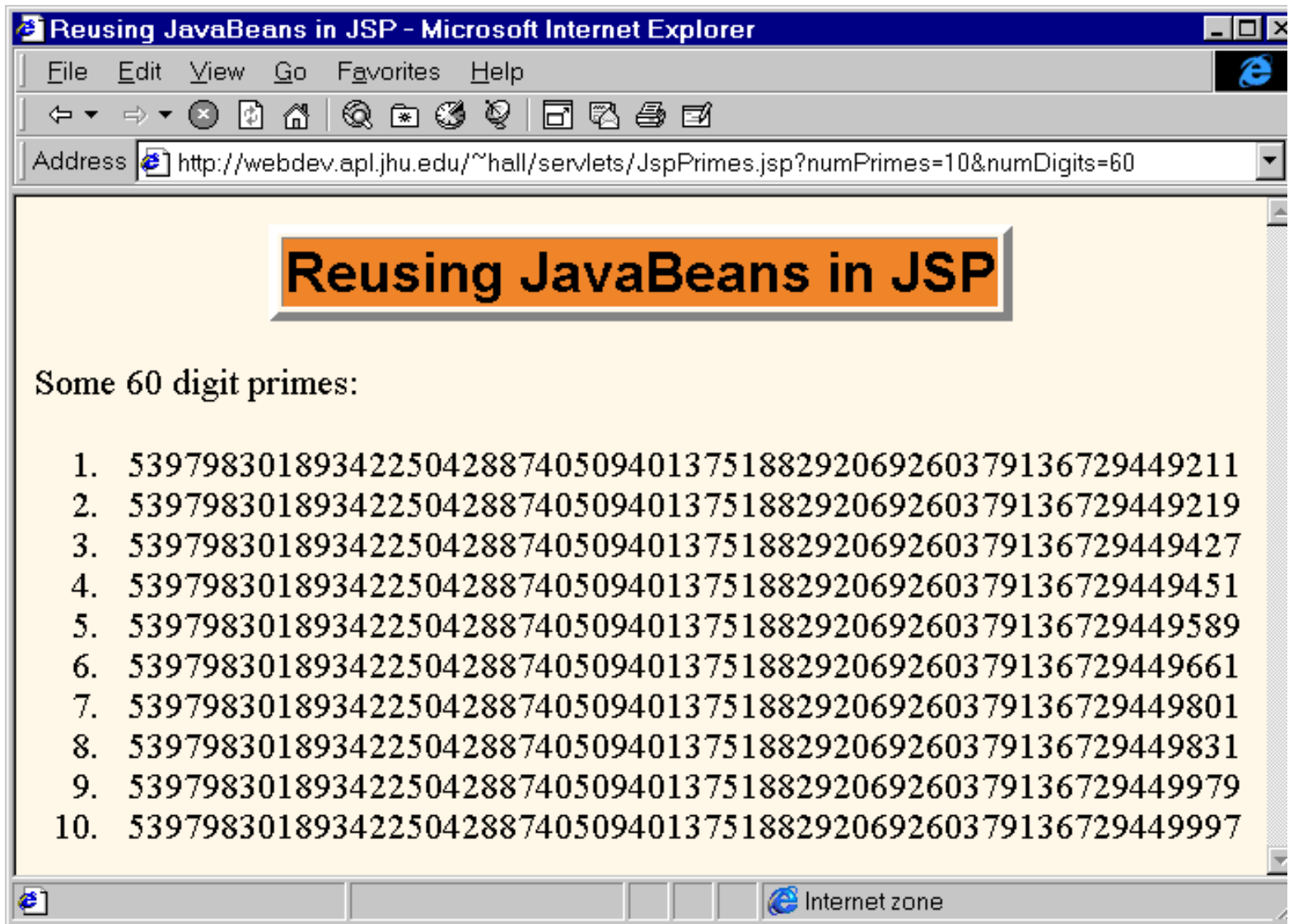
<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
    Reusing JavaBeans in JSP</TABLE>
</CENTER>
<P>

<jsp:useBean id="primeTable" class="hall.NumberedPrimes" />
<jsp:setProperty name="primeTable" property="numDigits" />
<jsp:setProperty name="primeTable" property="numPrimes" />

Some <jsp:getProperty name="primeTable" property="numDigits" />
digit primes:
<jsp:getProperty name="primeTable" property="numberedList" />

</BODY>
</HTML>
```

Aquí tenemos un resultado típico:



8.5 Acción jsp:getProperty

Este elemento recupera el valor de una propiedad del bean, lo convierte a un string, e inserta el valor en la salida. Los dos atributos requeridos son name, el nombre de un bean referenciado anteriormente mediante jsp:useBean, y property, la propiedad cuyo valor debería ser insertado. Aquí tenemos un ejemplo:

```

<jsp:useBean id="itemBean" ... />
...
<UL>
  <LI>Number of items:
    <jsp:getProperty name="itemBean" property="numItems" />
  <LI>Cost of each:
    <jsp:getProperty name="itemBean" property="unitCost" />
</UL>

```

8.6 Acción jsp:forward

Esta acción nos permite reenviar la petición a otra página. Tiene un sólo atributo, page, que debería consistir en una URL relativa. Este podría ser un valor estático, o podría ser calculado en el momento de la petición, como en estos dos ejemplos:

```

<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpression %>" />

```

8.7 Acción jsp:plugin

Esta acción nos permite insertar un elemento OBJECT o EMBED específico del navegador para especificar que el navegador debería ejecutar un applet usando el Plug-in Java.

9. Convenciones de Comentarios y Caracteres de Escape

Hay un pequeño número de construcciones especiales que podemos usar en varios casos para insertar comentarios o caracteres que de otra forma serían tratados especialmente:

Síntaxis	Propósito
<%-- comment --%>	Un comentario JSP. Ignorado por el traductor JSP-a-scriptlet. Cualquier elemento de script, directivas o acciones embebidas son ignorados.
<!-- comment -->	Un comentario HTML. Se pasa al HTML resultante. Cualquier elemento de script, directivas o acciones embebidas se ejecutan normalmente.
<%	Usado en plantillas de texto (HTML estático) donde realmente queremos "< % ".
%\>	Usado en elementos de script donde realmente queremos "% > ".
\'	Una sola comilla en un atributo que usa comillas simples. Sin embargo, recuerda que podemos usar comillas dobles o simples, y que otros tipos de comillas serán caracteres regulares.
\"	Una doble comilla un un atributo que usa comillas dobles. Sin embargo, recuerda que podemos usar comillas dobles o simples, y que otros tipos de comillas serán caracteres regulares.
%\>	% > en un atributo.
<%	< % en un atributo.



Introducción a JSP

Las JavaServer Pages (JSP) están basadas en la tecnología de servlets. Cuando se combina con el uso de componentes JavaBeans, JSP promete proporcionar una capacidad que es al menos tan poderosa como los Servlets, posiblemente más que un servlet en crudo, y potencialmente mucho más fácil de usar.

La creación y compilación del Servlet es automática

Cada página JSP es compilada automáticamente en un servlet por el motor JSP. (Sólo podemos usar JSP en servidores que sean compatibles con JSP).

La creación y compilación automática del servlet ocurre la primera vez que se accede a la página. Dependiendo del comportamiento del servidor web, el servlet será grabado durante algún periodo de tiempo para utilizarlo una y otra vez sin necesidad de recrearlo y recompilarlo.

Por eso, la primera vez que se accede a la página, podría haber una pausa mientras que el servidor web crea y compila el servlet. Después de esto, los accesos a la página serán muchos más rápidos.

¿Cómo se usan las página JSP?

Hay muchas formas diferentes de combinar JAP, Beans y Servlets. Como se menciono antes, el uso juicioso de JSP hace posible combinar las mejores capacidades del HTML con los componentes de software reutilizables para crear aplicaciones del lado del servidor.

Esto hace muy práctico separar la lógica del negocio de la representación de los datos. Con esto, los programadores especializados en escribir JavaBeans que implementen la lógica del negocio, y los diseñadores de páginas especializados en HTML pueden embeber llamadas a asos Beans desde el HTML sin sin necesidad de convertirse en expertos programadores Java.

¿Por donde Empezar?

Como en el caso de los Servlts y JDBC, para probar nuestros ficheros JSP, necesitaremos algo más que un compilador Java y una máquina virtual.

Además de estas dos cosas, necesitamos un servidor web compatible con JSP, en

el que probar los programas de ejemplo.

Podremos descargar la implementación de referencia de Sun del JSP desde:
<http://java.sun.com/products/jsp/download.html>.

Aquí tenemos lo que Sun dice sobre su servidor web:

JavaServer (TM) Web Development Kit (JSWDK) 1.0.1
Reference Implementation - Final Release

El JavaServer Web Development Kit (JSWDK) 1.0.1 combina la implementación de referencia para JavaServer Pages(TM) 1.0.1 y el API Java(TM) Servlet (2.1). ...

El JSWDK ofrece una forma simplificada de crear páginas web dinámicas que son independientes del servidor web y de plataformas de sistemas operativos.

Instalación y Configuración

Cuando descarguemos el JSWDK, encontraremos que se nos proporciona mucha información para ayudarnos a instalar y configurar el servidor. En algún momento deberemos leer esta documentación. Sin embargo, para que obtengas un arranque rápido vamos a mostrarte como configurar el servidor web en una máquina WinNT. Deberías poder configurar de una forma similar tu máquina y ver las primeras demostraciones de JSP muy rápidamente.

Instalación del JSWDK

El JSWDK se descarga como un fichero zip. Para instalarlo, simplemente tenemos que extraer los ficheros (preservando la estructura de directorios) desde el fichero zip.

Situar el árbol de directorios que contiene todos los ficheros en algún lugar de tu disco duro. Yo elegí el directorio Program Files en mis disco duro D.

Como resultado, mi path al directorio de más alto nivel que contiene el arbol de directorios JSWDK es:

`d:\Program Files\jswdk-1.0.1`

El nombre del directorio más alto en el árbol de directorios es jswdk-1.0.1. Con el tiempo, con la liberación de nuevas versiones, el nombre de este directorio seguro que cambiará.

¿Qué pasa con el JDK?

Cuando usamos nuestro navegador para solicitar una página JSP desde el servidor, éste debe crear y compilar un servlet. El servidor jswdk no contiene un compilador ni una máquina virtual. Por lo tanto, si no lo tenemos ya instalado, debemos instalar el JDK para que el servidor lo use para compilar y ejecutar el servlet.

En el momento de escribir esto, tengo instalada la versión JDK 1.2.2 en mi máquina. El Path al directorio de más alto nivel del JDK es:

```
d:\Program Files\jdk1.2.2
```

Mi instalaciónes completamente estándar (según Sun) excepto en que la tengo instalada en el disco D en lugar del disco C.

¿Qué pasa con el classpath?

Cuando el servidor intenta compilar el servlet, debe saber cómo localizar los ficheros class del JDK. Por lo tanto, es necesaria la variable de entorno classpath. Aquí puedes ver una parte de mi classpath:

```
d:\Program Files\jdk1.2.2\lib\tools.jar
```

Este elemento identifica la localización del fichero JAR que contiene las librerías de clases estándares del JDK.

Classpath para servlets

Como también uso el JSWDK como motor de servlet (independiente del JDP), también necesito el siguiente elemento en el classpath:

```
d:\Program Files\jswdk-1.0.1\lib\servlet.jar
```

Este elemento identifica la localización del fichero jar que contiene los distintos componentes de software necesarios para compilar y ejecutar servlets.

Instalar los Ficheros HTML, JSP, y de Servlets

una vez instalado el JSWDK, veremos que el árbol de directorios resultante es bastante complejo con varias ramas diferentes.

Para usar el JSWDK en su configuración por defecto, debemos instalar los ficheros HTML, JSP, y de servlets en los siguientes directorios:

Situamos los ficheros HTML y JSP en el siguiente directorio:

```
d:\Program Files\jswdk-1.0.1\webpages
```

Situamos los ficheros class de los servlets en el siguiente directorio:

```
d:\Program Files\jswdk-1.0.1\examples\Web-inf\servlets
```

(la parte inicial de tu path podría ser diferente, dependiendo de donde situaras el JSWDK en tu disco duro).

Arrancar y Parar el Servidor

Podemos arrancar el servidor ejecutando el siguiente fichero batch:

```
d:\Program Files\jswdk-1.0.1\startserver.bat
```

Deberíamos parar el servidor ejecutando el siguiente fichero batch:

```
d:\Program Files\jswdk-1.0.1\stopserver.bat
```

He situado accesos directos a estos ficheros batch en mi escritorio para arrancar y parar el servidor fácilmente.

Ficheros Temporales

Cuando arrancamos el servidor y accedemos a una página JSP, si prestamos atención al árbol de directorios, veremos que se han creado varios ficheros temporales en un directorio llamado work. Este directorio es un subdirectorio del directorio de más alto nivel del motor JSP llamado jswdk-1.0.1. Cuando paramos el servidor, estos ficheros temporales se borran automáticamente.

Acceder a Ficheros HTML y JSP

Acceder a un Fichero JSP

Habiendo realizado la instalación y configuración descrita arriba, y habiendo instalado un fichero JSP (llamado jsp001.jsp) en el directorio adecuado, debemos poder acceder a ese fichero JSP introduciendo la siguiente URL en la ventana de nuestro navegador:

```
http://localhost:8080/jsp001.jsp
```

Podríamos necesitar estar online para que esto funcione. En caso de que no funcione, necesitaremos estudiar la documentación del JSWDK para aprender otras formas alternativas para direccionar el servidor.

Acceder a un Fichero HTML

De forma similar, deberíamos poder acceder a un fichero HTML llamado jsp001.htm introduciendo la siguiente URL en nuestra ventana de navegador:

`http://localhost:8080/jsp001.htm`

Ozito



Directivas JSP

¿Qué es la Directiva Include?

La directiva include se usa para insertar un fichero dentro de una página JSP cuando se compila la página JSP. El texto del fichero incluido se añade a la página JSP (ver la descripción de un fichero estático más adelante en esta página)

¿Qué clases de ficheros se pueden incluir?

El fichero incluido puede ser un fichero JSP, un fichero HTML, o un fichero de texto. También ser un fichero de código escrito en lenguaje Java.

Hay que ser cuidadoso en que el fichero incluido no contenga las etiquetas `<html>`, `</html>`, `<body>`, or `</body>`. Porque como todo el contenido del fichero incluido se añade en esa localización del fichero JSP, estas etiquetas podrían entrar en conflicto con las etiquetas similares del fichero JSP.

Incluir Ficheros JSP

Si el fichero incluido es un fichero JSP, las etiquetas JSP son analizadas y sus resultados se incluyen (junto con cualquier otro texto) en el fichero JSP.

Sólo podemos incluir ficheros estaticos. Esto significa que el resultado analizado del fichero incluido se añade al fichero JSP justo donde está situada la directiva. Una vez que el fichero incluido es analizado y añadido, el proceso continúa con la siguiente línea del fichero JSP llamante.

¿Qué es un fichero Estático?

Un include estático significa que el texto del fichero incluido se añade al fichero JSP.

Además en conjunción con otra etiqueta JSP, `<jsp:include>`: podemos incluir ficheros estáticos o dinámicos:

Un fichero estático es analizado y si contenido se incluye en la página JSP llamante.

Un fichero dinámico actúa sobre la solicitud y envía de vuelta un resultado que es incluido en la página JSP.

¿Cuál es la Sintaxis para Incluir un Fichero?

Podemos incluir un fichero en la localización específica del fichero JSP usando la directiva `include` con la siguiente sintaxis:

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

Aquí la URL puede ser una URL relativa indicando la posición del fichero a incluir dentro del servidor.

¿Qué es la Directiva Page?

La directiva `Page` se usa para definir atributos que se aplican a una página JSP entera.

La directiva `page` se aplica a una página JSP completa, y a cualquier fichero estático que incluya con la directivas `include` o `<jsp:include>`, que juntas son llamadas una unidad de traducción.

Observa que la directiva `page` no se aplica a cualquier fichero dinámico incluido.

Una directiva `page` puede usarse para establecer valores para distintos atributos que se pueden aplicar a la página JSP. Podemos usar la directiva `page` más de una vez en una página JSP (unidad de traducción). Sin embargo, (excepto para el atributo `import`), sólo podemos especificar un valor para atributo una sola vez.

El atributo `import` es similar a la directiva `import` en un programa Java, y por eso podemos usarlar más de una vez.

Podemos situar el directiva `page` en cualquier lugar de la unidad de traducción y se aplicará a toda la unidad de traducción. Por calridad y facilidad de entendimiento, el mejor lugar podría ser al principio del fichero JSP principal.

¿Cuál es la Síntaxis de la Directiva page

Aquí podemos ver la sintaxis de la directiva `page`. Los valores por defecto se muestran en negrita. Los corchetes (`[...]`) indican un término opcional. La barra vertical (`|`) proporciona una elección entre dos valores como `true` y `false`.

```
<%@ page
  [ language="java" ]
  [ extends="package.class" ]
  [ import= "{ package.class|package.*}, ..." ]
  [ session="true|false" ]
  [ buffer="none|8kb|sizekb" ]
  [ autoFlush="true|false" ]
```

```
[ isThreadSafe="true|false" ]
[ info="text" ]
[ errorPage="URLrelativa" ]
[ contentType="mimeType[ ;charset=characterSet]" |
    "text/html; charset=ISO-8859-1" ]
[ isErrorPage="true|false" ]
%>
```

Ejemplos de Directivas Page

```
<% @ page import="java.util.Date" % >
<% @ page import="java.awt.*" % >
<% @ page info="Información sobre la página" % >
```

Atributos de la Directiva Page

language="java"

Este atributo define el lenguaje de script usado en los scriptlets, declaraciones y expresiones en el fichero JSP y en cualquier fichero incluido. En JSP 1.0 el único lenguaje permitido es Java.

extends="package.class"

Este atributo especifica un nombre totalmente cualificado de una superclase que será extendida por la clase Java en el fichero JSP. Se recomienda que usemos este atributo con caute, ya puede limitar la habilidad del motor del JSP a proporcionar la superclase especializada que mejora la calidad del fichero compilado.

import= "{ package.class | package.* }, ..."

Esta lista especifica una lista separada por comas de uno o más paquetes o clases que el fichero JSP debería importar. Las clases de los paquetes se ponen a disposición de los scriptlets, expresiones, declaraciones y etiquetas dentro del fichero JSP.

Como cabría esperar, el atributo import debe aparecer antes de cualquier etiqueta que refiera la clase importada. Para importar varios paquetes, podemos usar una lista separada por comas, más de una directiva import o una combinación de ambas.

session="true|false"

Todo cliente debe unirse a una sesión HTTP para poder usar una página JSP. Si el valor es true, el objeto session se refiere a la sesión actual o a una nueva sesión. Si el valor es false, no podemos utilizar el objeto session en el fichero JSP. El valor por defecto es true.

buffer="none|8kb|sizekb"

Este atributo especifica el tamaño del buffer en kilobytes que será usado por el objeto out para manejar la salida enviada desde la página JSP compilada hasta el navegador cliente. El valor por defecto es 8kb.

`autoFlush= "true|false"`

Este atributo especifica si la salida sería enviada o no cuando el buffer esté lleno. Por defecto, el valor es true, el buffer será descargado. Si especificamos false, se lanzará una excepción cuando el buffer se sobrecargue.

`isThreadSafe= "true|false"`

Este atributo especifica si la seguridad de threads está implementada en el fichero JSP. El valor por defecto, true, significa que el motor puede enviar múltiples solicitudes concurrentes a la página.

Si usamos el valor por defecto, varios threads pueden acceder a la página JSP. Por lo tanto, debemos sincronizar nuestros métodos para proporcionar seguridad de threads.

Con false, el motor JSP no envía solicitudes concurrentes a la página JSP. Probablemente no queremos forzar esta restricción en servidores de gran volumen porque puede dañar la habilidad del servidor de enviar nuestra página JSP a múltiples clientes.

`info= "text"`

Este atributo nos permite especificar una cadena de texto que es incorporada en el página JSP compilada. Podemos recuperar el string más tarde con el método `getServletInfo()`.

`errorPage= "URLrelativa"`

Este atributo especifica un path a un fichero JSP al que este fichero JSP envía excepciones. Si el path empieza con una "/", el path es relativo al directorio raíz de documentos de la aplicación JSP y es resuelto por el servidor Web. Si no, el path es relativo al fichero JSP actual.

`isErrorPage= "true|false"`

Este atributo especifica si el fichero JSP muestra una página de error. Si es true, podemos usar el objeto `exception`, que contiene una referencia a la excepción lanzada, en el fichero JSP. Si es false (el valor por defecto), significa que no podemos usar el objeto `exception` en el fichero JSP.

`contentType= " mimeType [; charset= characterSet]" | "text/html; charset= ISO-8859-1"`

Este atributo especifica el tipo MIME y la codificación de caracteres que use el fichero JSP cuando se envía la respuesta al cliente. Podemos usar cualquier tipo MIME o conjunto de caracteres que sean válidos para el motor JSP.

El tipo MIME por defecto es `text/html`, y el conjunto de caracteres por defecto es `ISO-8859-1`.



El Principio de JSP

Esta figura muestra lo que quizás sea la aplicación JSP más sencilla que uno podría escribir.

Duke Dice Hello



El Banner de Duke (dukebanner.html)



```
<table border="0" width="400" cellspacing="0"
        cellpadding="0">

<tr>
<td height="150" width="150"> &nbsp; </td>
<td width="250"> &nbsp; </td>
</tr>

<tr>
<td width="150"> &nbsp; </td>
<td align="right" width="250">

         </td>
</tr>

</table>
<br>
```

La página JSP (helloworld.jsp)



```

<%@ page info="a hello world example" %>

<html>
<head><title>Hello, World</title></head>
<body bgcolor="#ffffff" background="background.gif">

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

<table>
<tr>
<td width=150> &nbsp; </td>
<td width=250 align=right> <h1>Hello, World!</h1> </td>
</tr>
</table>

</body>
</html>

```

La Directiva Page

La directiva Page es una etiqueta JSP que usaremos encasos todos los ficheros fuente JSP que escribamos. En helloworld.jsp, es la línea que se parece a esto:

```

<%@ page info="a hello world example" %>

```

Esta directiva da instrucciones al motor JSO que aplica a todo el fichero fuente JSP. En este ejemplo, está directiva especifica un comentario informativo que formará parte del fichero JSP compilado. En otros casos, podría especificar el lenguaje de script usado en el fichero fuente JSP, los paquetes de ficheros fuentes que serán importados, o la página de error que se llamará si ocurren errores o excepciones.

Podemos usar la directiva page en cualquier lugar del fichero JSP, pero es un buen estilo de codificación situarlo en la parte superior del fichero. como es una etiqueta JSP, podemos situarla antes de la etiqueta de apertura <html> tag.

La Directiva Include

La directiva include inserta el contenido de otro fichero en el fichero principal JSP, donde está situada la directiva. Es útil para incluir informaciónde copuright, ficheros de lenguaje de script, p cualquier cosa que podríamos querer reutilizar en otras aplicaciones. En este ejemplo, el fichero incluido es una tabla qu crea un banner gráfico.

Podemos ver el contenido del fichero incluido viendo la página fuente del fichero principal JSP mientras estamos ejecutando Hello, World. El fichero incluido no contiene etiquetas <html> o <body>, porque podrían generar conflictos con las mismas etiquetas del fichero JSP llamante.

Una Nota sobre las Etiquetas JSP

Cuando uses los ejemplos de este capítulo, recuerda que las etiquetas JSP son sensibles a las mayúscula. Si, por ejemplo, tecleamos `<jsp:usebean>` en lugar de `<jsp:useBean>`, nuestra etiqueta no será reconocida, y la implementación de referencia JSP 1.0 lanzará una excepción. Algunos de los atributos de las etiquetas toman nombres de clases, nombres de paquetes, pathnames o otros valores también sensibles a las mayúsculas.

¿Cómo ejecutar la aplicación de ejemplo

Las instrucciones dadas aquí usan una pathname del estilo UNIX. Si estamos trabajando en Windows, usamo el mismo pathname pero con el separador de directorios apropiado

1. Creamos el directorio (o carpeta)
../jswdk-1.0/examples/jsp/tutorial/helloworld.
2. Situamos los siguientes ficheros en el directorio ../tutorial/hello:
background.gif, duke.waving.gif, dukebanner.html, y helloworld.jsp.
3. Desde la línea de comandos, arrancamos la implementación de referencia JSP de Sun: `cd ../jswdk-1.0 startserver`
4. Abrimos un navegador Web y vamos a
`http://yourMachineName:8080/examples/jsp/tutorial/helloworld/helloworld.jsp`



Manejar Formularios HTML

Una de las partes más comunes de una aplicación de comercio electrónico es un formulario HTML en el que un usuario introduce alguna información. La información podría ser un nombre de cliente y su dirección, una palabra o frase introducida para un motor de búsqueda, o un conjunto de preferencias lanzadas como datos del mercado.

¿Qué le sucede a los datos del Formulario?

La información que el usuario introduce en el formulario se almacena en el objeto request, que se envía desde el cliente al motor JSP. ¿Qué sucede luego?

La siguiente figura representa como fluyen los datos entre el cliente y el servidor (al menos cuando usamos la implementación JSP de Sun, otros motores JSP podrían trabajar de forma diferente).

¿Cómo se pasan los datos entre el cliente y el servidor?



El motor JSP envía el objeto solicitado a cualquier componente del lado del servidor (JavaBeans™, servlet, o bean enterprise) que especifica el JSP. El componente maneja la solicitud, posiblemente recuperando datos desde una base de datos u otros datos almacenados, y pasa un objeto respuesta de vuelta al motor JSP, donde los datos se formatean de acuerdo al diseño de la página HTML. El motor JSP y el servidor Web entonces envían la página JSP revisada de vuelta al cliente, donde el usuario puede ver el resultado, en el navegador Web. Los protocolos de comunicación usados entre el cliente y el servidor pueden ser HTTP, o cualquier otro protocolo.

Los objetos request y response están siempre implícitamente disponibles para nosotros como autores de ficheros fuentes JSP. El objeto request se explica con más detalle más adelante en este tutorial.

¿Cómo Crear un Formulario?

Normalmente se define un formulario HTML en un fichero fuente JSP, usando etiquetas JSP para pasar los datos entre el formulario y algún tipo de objeto del

lado del servidor (usualmente un Bean). En general, haremos las siguientes cosas en nuestra aplicación JSP:

1. Empezaremos a escribir un fichero fuente JSP, creando un formulario HTML y dándole un nombre a cada elemento.
2. Escribimos el Bean en un fichero .java, definiendo las propiedades, los métodos get y set para los nombres de los elementos del formulario (a menos que querramos seleccionar explícitamente un valor propiedad).
3. Devolvemos el fichero fuente JSP. Añadimos una etiqueta `<jsp:useBean>` para crear y asignar un ejemplar del Bean.
4. Añadimos una etiqueta `<jsp:setProperty>` para seleccionar propiedades del Bean desde el formulario HTML (el Bean necesita un método set correspondiente).
5. Añadimos una etiqueta `<jsp:getProperty>` para recuperar los datos desde el Bean (el Bean necesita un método get correspondiente).
6. Si necesitamos realizar más procesos sobre los datos del usuario, usamos el objeto request desde dentro de un scriptlet.

El ejemplo Hello, User hará estos pasos más claros.

Una sencilla aplicación "Hello"

El usuario introduce un nombre, y Duke dice Hello!



Código de Ejemplo

El Banner Duke (dukebanner.html)



```
<table border="0" width="400" cellspacing="0"
        cellpadding="0">
<tr>
<td height="150" width="150"> &nbsp; </td>
<td width="250"> &nbsp; </td>
</tr>
<tr>
<td width="150"> &nbsp; </td>
<td align="right" width="250">
     </td>
</tr>
</table>
<br>
```

El fichero principal JSP (hellouser.jsp)



```
<%@ page import="hello.NameHandler" %>

<jsp:useBean id="mybean" scope="page"
             class="hello.NameHandler" />
<jsp:setProperty name="mybean" property="*" />

<html>
<head><title>Hello, User</title></head>
<body bgcolor="#ffffff" background="background.gif">

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

<table border="0" width="700">
<tr>
<td width="150"> &nbsp; </td>
<td width="550">
<h1>My name is Duke. What's yours?</h1>
</td>
</tr>
<tr>
<td width="150" &nbsp; </td>
<td width="550">
```



```

<form method="get">
<input type="text" name="username" size="25">
<br>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</td>
</tr>
</form>
</table>

<%
    if ( request.getParameter("username") != null ) {
%>

<%@ include file="response.jsp" %>

<%
    }
%>

</body>
</html>

```

El fichero de Respuesta (response.jsp)



```

<table border="0" width="700">
<tr>
<td width="150"> &nbsp; </td>

<td width="550">

<h1>Hello, <jsp:getProperty name="mybean"
    property="username" />!
</h1>

</td>
</tr>
</table>

```

El Bean que Maneja los Datos del Formulario (namehandler.java)

```

package hello;

```

```

public class NameHandler {

    private String username;

    public NameHandler() {
        username = null;
    }

    public void setUsername( String name ) {
        username = name;
    }

    public String getUsername() {
        return username;
    }

}

```

Construir el Formulario HTML

Un formulario HTML tiene tres partes principales: las etiquetas de apertura y cierre `<form>`, los elementos de entrada, y el botón Submit que envía los datos al servidor. En una página normal HTML, la etiqueta de apertura `<form>` normalmente se parecerá a algo como esto:

```
<form method=get action=someURL>
```

En otras aplicaciones web, el atributo `action` especifica un script CGI u otro programa que procese los datos del usuario. En un fichero JSP, podemos omitir el atributo `action` si queremos que los datos se envíen al Bean especificado en la etiqueta `<jsp:useBean>` o especificar otro fichero JSP.

El resto del formulario se construye igual que un formulario estándar HTML, con elementos de entrada, un botón Submit, y quizás un botón Reset. Debemos asegurarnos de dar a cada elemento un nombre, como este:

```
<input type="text" name="username">
```

Usar los Métodos GET y POST

Los métodos HTTP GET y POST envían datos al servidor. En una Aplicación JSP, GET y POST envían los datos a un Bean, servlet, u otro componente del lado del servidor que está manejando los datos del formulario.

En teoría, GET es para obtener datos desde el servidor y POST es para enviar datos. Sin embargo, GET añade los datos del formulario (llamada una query string (string de solicitud)) a una URL, en la forma de parejas clave/valor desde el formulario HTML, por ejemplo, `name=john`. En el String de solicitud las parejas

de clave/valor se separarán por caracteres &, los espacios se convierten en caracteres +, y los caracteres especiales se convierten a sus correspondientes hexadecimales. Como el String de solicitud está en la URL, la página puede ser añadida a los bookmarks o enviada por e-mail con el string de solicitud. Este string normalmente está limitado a un número relativamente pequeño de caracteres.

Sin embargo, el método POST, pasa los datos de una longitud ilimitada como un cuerpo de solicitud de cabecera HTTP hacia el servidor. El usuario que trabaja en el navegador cliente no puede ver los datos que están siendo enviados, por eso la solicitud POST es ideal para enviar datos confidenciales (como el número de una tarjeta de crédito) o grandes cantidades de datos al servidor.

Escribir el Bean

Si nuestra aplicación usa un Bean, podemos escribir el Bean de acuerdo a los patrones de diseño explicados en la Especificación del API de JavaBeans, recordamos estos puntos generales:

- Si usamos una etiqueta `<jsp:getProperty>` en nuestro fichero fuente JSP, necesitamos el correspondiente método `get` en el Bean.
- Si usamos una etiqueta `<jsp:setProperty>` en nuestro fichero fuente JSP, necesitamos uno o más métodos `set` correspondientes.

Seleccionar las propiedades en unas propiedades obtenidas desde un Bean se explica un poco más en la siguiente sección:

Obtener los Datos desde el Fomulario hacia el Bean

Seleccionar las propiedades en un Bean desde un formulario HTML es una tarea en dos partes:

- Crear o localizar el ejemplar del Bean con `lt;jsp:useBean>`
- Seleccionar los valores de la propiedad en el Bean con `<jsp:setProperty>`

El primer paso es ejemplarizar o localizar un Bean con una etiqueta `<jsp:useBean>` antes de seleccionar los valores de las propiedades en el bean. En un fichero fuente JSP, la etiqueta `<jsp:useBean>` debe aparecer sobre la etiqueta `<jsp:setProperty>`. La etiqueta `<jsp:useBean>` primero busca un ejemplar de Bean con un nombre que especifiquemos, pero si no encuentra el Bean, lo ejemplariza. Esto nos permite crear el Bean en un fichero JSP y usarlo en otro, siempre que el Bean tenga un ámbito suficientemente grande.

El segundo paso es seleccionar el valor de la propiedad en el Bean con una etiqueta `<jsp:setProperty>`. La forma más fácil de usar `<jsp:setProperty>` es definir propiedades en el Bean que correspondan con los nombres de los elementos del formulario. También deberíamos definir los correspondientes métodos `set` para cada propiedad. Por ejemplo, si el elemento del formulario se llama `username`, deberíamos definir una propiedad `username` u los métodos `getUsername` y

setUsername en el Bean.

Si usamos nombres diferentes para el elemento del formulario y la propiedad del Bean, todavía podríamos seleccionar el valor de la propiedad con `<jsp:setProperty>`, pero sólo podemos seleccionar una valor a la vez.

Chequear el Objeto Request

Los datos que el usuario introduce se almacenan en un objeto request, que usualmente implementa `javax.servlet.HttpServletRequest` (o si nuestra implementación usa un protocolo diferente, otro interface que sea una subclase de `javax.servlet.ServletRequest`).

Podemos acceder al objeto request directamente desde dentro de un scriptlet. Los scriptlet son fragmentos de código escritos en un lenguaje de scripts y situado dentro de los caracteres `<% y %>`. En JSP 1.0, debemos usar el lenguaje de programación Java como lenguaje de Script.

Podríamos encontrar útiles algunos de estos métodos para trabajar con objetos request:

Método	Definido en	Trabajo Realizado
<code>getRequest</code>	<code>javax.servlet.jsp.PageContext</code>	Devuelve el Objeto request actual
<code>getParameterNames</code>	<code>javax.servlet.ServletRequest</code>	Devuelve los nombres de los parámetros contenidos actualmente en request
<code>getParameterValues</code>	<code>javax.servlet.ServletRequest</code>	Devuelve los valores de los parámetros contenidos actualmente en request
<code>getParameter</code>	<code>javax.servlet.ServletRequest</code>	Devuelve el valor de un parámetro su proporcionamos el nombre

También podemos encontrar otros métodos definidos en `ServletRequest`, `HttpServletRequest`, o cualquier otra sobclase de `ServletRequest` que implemente nuestra aplicación

El motor JSP siempre usa el objeto request detrás de la escena, incluso si no la llamamos explícitamente desde el fichero JSP.

Obtener Datos desde el Bean a la Página JSP

Una vez que los datos del usuario han sido enviados al Bean, podríamos querer recuperar los datos y mostrarlos en la página JSP. Para hacer esto, usamos la etiqueta `<jsp:getProperty>`, dándole el nombre del Bean y el nombre de la propiedad:

```
<h1>Hello, <jsp:getProperty name="mybean"
    property="username" />!
```

Los nombres de Beans que usemos en las etiquetas `<jsp:useBean>`, `<jsp:setProperty>`, y `<jsp:getProperty>` deben ser iguales, por ejemplo:

hellouser.jsp:

```
<jsp:useBean id="mybean" scope="session"
            class="hello.NameHandler" />
<jsp:setProperty name="mybean" property="*" />
```

response.jsp:

```
<h1>Hello, <jsp:getProperty name="mybean"
                        property="username" />!
```

En este ejemplo, las etiquetas están en dos ficheros, pero los nombres de los Bean son iguales. Si no fuera así, la implementación de referencia del JSP lanzará un error.

La respuesta que el motor JSP devuelve al cliente está encapsulada en el objeto `response` implícito, que crea el motor JSP.

¿Cómo Ejecutar el Ejemplo

Las instrucciones dadas usan un path al estilo de Unix, si trabajamos vbajo Windows, usaremos el mismo path pero con el separador apropiado:

1. Creamos el directorio (o carpeta)
../jswdk-1.0/examples/jsp/tutorial/hellouser.
2. Situamos los siguientes ficheros en el directorio (o carpeta) ../tutorial/hellouser: background.gif, duke.waving.gif, dukebanner.html, hellouser.jsp, y response.jsp.
3. Creamos el directorio (o carpeta)
../jswdk-1.0/examples/WEB-INF/jsp/beans/hello. Observa que el directorio se llama hello y no hellouser.
4. Situamos los ficheros NameHandler.java y NameHandler.class en el directorio ../beans/hello.
5. Arrancamos la implementación de referencia de JSP: `cd ../jswdk-1.0 startserver`
6. Abrimos un navegador Web y vamos a
`http://yourMachineName:8080/examples/jsp/tutorial/hellouser/hellouser.jsp`



Usar Elementos de Scripting

En el algún momento, probablemente querramos añadir algún viejo código a nuestras ficheros JSP. Las etiquetas JSP son poderosas y encapsulan tareas que serían difíciles o llevarían mucho tiempo de programar. Pero por eso, probablemente querramos usar fragmentos de lenguaje de script para suplementar las etiquetas JSP.

Los lenguajes de script que tenemos disponibles dependen del motor JSP que estemos usando. Con la implementación JSP de Sun, debemos usar el lenguaje Java como lenguaje de script, pero otros motores de JSP de terceras partes podrían incluir soporte para otros lenguajes.

¿Cómo añadir Scripting?

Primero, necesitaremos conocer una pocas reglas generales sobre la adición de elementos script a un fichero fuente JSP:

1. Usamos una directiva page para definir el lenguaje de script utilizado en la página JSP (a menos que estemos usando el lenguaje Java, que es el valor por defecto).
2. La declaración `<% ! .. %>` declara variables o métodos.
3. La expresión `<% = .. %>` define una expresión del lenguaje script y fuerza el resultado a un String.
4. El scriptlet `<% .. %>` puede manejar declaraciones, expresiones o cualquier otro tipo de fragmento de código válido en el lenguaje script de la página.
5. Cuando escribimos un scriptlet, lo terminamos con `%>` antes de cambiar a HTML, texto u otra etiqueta JSP.

La Diferencia entre `<%`, `<%=`, y `<%!`

Declaraciones, expresiones y scriptlets tienen una sintaxis y un uso similar, pero también tienen diferencias importantes.

Declaraciones (entre las etiquetas `<% !` y `%>`) contiene una o más declaraciones de variables o métodos que terminan o están separadas por puntos y comas:

```
<%! int i = 0; %>
<%! int a, b; double c; %>
```

```
<%! Circle a = new Circle(2.0); %>
```

Debemos declarar una variable o método en una página JSP antes de usarla en la página. El ámbito de la declaración normalmente es el fichero JSP, pero si el fichero JSP incluye otros ficheros con la directiva include, el ámbito se expande para incluir también estos ficheros incluidos.

Expresiones (entre las etiquetas `<%=` y `%>`) pueden contener cualquier expresión del lenguaje que sea válida en el lenguaje de script de la página, pero sin punto y coma:

```
<%= Math.sqrt(2) %>
<%= items[i] %>
<%= a + b + c %>
<%= new java.util.Date() %>
```

La definición de una expresión válida depende del lenguaje de script. Cuando usamos lenguaje Java, lo que hay entre las etiquetas de expresión puede ser cualquier expresión definida en la Especificación del Lenguaje Java. Las partes de la expresión se evalúan de izquierda a derecha. Una diferencia clave entre expresiones y scriptlets es que dentro de las etiquetas de expresión no se permiten puntos y comas, incluso si la misma expresión lo requiere.

Scriptlets (entre las etiquetas `<%` y `%>`) nos permite escribir cualquier número de sentencias del lenguaje Script, de esta forma:

```
<%
    String name = null;
    if (request.getParameter("name") == null) {
%>
```

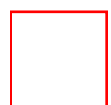
Recuerda que en un scriptlet debemos terminar una sentencia de lenguaje con un punto y coma si el lenguaje los requiere.

Cuando escribamos un scriptlet podemos usar cualquiera de los objetos o clases implícitos de JSP importados por la directiva page, declarada en una declaración, o nombrada en una etiqueta `<jsp:useBean>` .

The Number Guess Game

El juego Number Guess es divertido y hace buen uso de scriptlets y expressions, así como de los conocimientos de los formularios HTML que aprendimos en la página anterior.

Sobre "Guess a Number":



Código de Ejemplo

Mostrar la pantalla de "Number Guess" (numguess.jsp)



```
<!--
    Number Guess Game
    Written by Jason Hunter, CTO, K&A Software
    jasonh@kasoftware.com, http://www.servlets.com
    Copyright 1999, K&A Software
    Distributed by Sun Microsystems with permission
-->

<%@ page import = "num.NumberGuessBean" %>

<jsp:useBean id="numguess" class="num.
    NumberGuessBean" scope="session" />
<jsp:setProperty name="numguess" property="*" />

<html>
<head><title>Number Guess</title></head>
<body bgcolor="white">
<font size=4>

<% if (numguess.getSuccess() ) { %>
    Congratulations!  You got it.
    And after just <%= numguess.getNumGuesses() %>
    tries.<p>

        <% numguess.reset(); %>
        Care to <a href="numguess.jsp">try again</a>?

<% } else if (numguess.getNumGuesses() == 0) { %>

    Welcome to the Number Guess game.<p>
    I'm thinking of a number between 1 and 100.<p>

    <form method=get>
    What's your guess? <input type=text name=guess>
    <input type=submit value="Submit">
    </form>

<% } else { %>
```


Good guess, but nope. Try **<%= numguess.
getHint() %>**.
You have made **<%= numguess.getNumGuesses() %>**
guesses.

I'm thinking of a number between 1 and 100.<p>

```
<form method=get>
What's your guess? <input type=text name=guess>
<input type=submit value="Submit">
</form>
```

```
<% } %>
</font>
</body>
</html>
```

Manejar el Guess (NumberGuessBean.java)

```
// Number Guess Game
// Written by Jason Hunter, CTO, K&A Software
// jasonh@kasoftware.com, http://www.servlets.com
// Copyright 1999, K&A Software
// Distributed by Sun Microsystems with permission
```

```
package num;
```

```
import java.util.*;
public class NumberGuessBean {

    int answer;
    boolean success;
    String hint;
    int numGuesses;

    public NumberGuessBean() {
        reset();
    }

    public void setGuess(String guess) {
        numGuesses++;

        int g;
        try {
            g = Integer.parseInt(guess);
```

```

    }
    catch (NumberFormatException e) {
        g = -1;
    }
    if (g == answer) {
        success = true;
    }
    else if (g == -1) {
        hint = "a number next time";
    }
    else if (g < answer) {
        hint = "higher";
    }
    else if (g > answer) {
        hint = "lower";
    }
}

public boolean getSuccess() {
    return success;
}

public String getHint() {
    return "" + hint;
}

public int getNumGuesses() {
    return numGuesses;
}

public void reset() {
    answer = Math.abs(new Random().nextInt() % 100)
        + 1;
    success = false;
    numGuesses = 0;
}
}

```

Usar Elementos Script en un fichero JSP

El fichero numguess.jsp es un ejemplo interesante del uso de elementos script, porque está estructurado como nosotros estructuraríamos un fichero fuente con una larga sentencia if ... else dentro de las etiquetas scriptlet. La diferencia es que el cuerpo de cada clausula de sentencia están escritas en HTML y etiquetas JSP, en vez del lenguaje de programación.

No es necesario que escribamos scriptlets mezclados con HTML y etiquetas

JSP, como se muestra en numguess.jsp. Entre las etiquetas `<%` y `%>`. Podemos escribir cuantas línea de código script creamos necesarias. En general, hacer menos proceso en los scriptlets y más en los componentes como servlets o Beans haremos el código de nuestra aplicación más reutilizable y portable. No obstante, podemos escribir nuestra aplicación JSP como queramos, y la implementación de referencia de JSP 1.0 de Sun no especifica límite a la longitud del scriptlet.

Mezclar Sentencias Scripting con Etiquetas

Cuando mezclamos elementos scripting con etiquetas HTML y JSP, siempre debemos terminar un elemento de scripting antes de empezar a usar etiquetas y luego reabrir el elemento de scripting, de esta forma:

```
<% } else { %>
  <!-- Cerrar el escript antes de empezar las etiquetas-->

... siguen las etiquetas...

<% } %>
  <!-- reabrimos el scriptlet para cerrar el bloque de lenguaje-->
```

Al principio, esto podría parecer un poco extraño, pero así se asegura de que los elementos de scripting son transformados correctamente cuando se compila el fichero fuente JSP.

¿Cuándo se ejecutan los elementos de Scripting?

Un fichero fuente JSP se procesa en dos estado - traducción HTTP y procesamiento de solicitud.

Durante la traducción HTTP, que ocurre cuando el usuario carga por primera vez una página JSP, el fichero fuente JSP es compilado a una clase Java, normalmente un Servlet Java. Las etiquetas HTML así como muchas etiquetas JSP son procesadas en este estado, antes de que el usuario haga una petición.

El procesamiento de solicitud ocurre cuando el usuario pulsa en la página JSP para hacer un solicitud. La solicitud es enviada desde el cliente al servidor mediante el objeto request. Entonces el motor JSP ejecuta el fichero JSP compilado, o servlet, usando los valores del request enviado por el usuario.

Cuando usamos elementos de scripting en un fichero JSP, deberíamos saber cuando son evaluadas. Las declaraciones son procesadas durante la traducción HTTP y están disponibles para otras declaraciones, expresiones y scriptlets en el fichero compilador JSP. La expresiones también se evalúan durante la traducción a HTTP. El valor de cada expresión se convierte a un String y es insertado en su lugar del fichero compilador JSP. Sin embargo, los scriptlets son evaluados durante

el proceso de la solicitud, usando las declaraciones y expresiones que se han puesto a su disposición.

¿Cómo ejecutar el Ejemplo?

Las instrucciones dadas aquí usan path al estilo Unix, si utilizamos Windows, usaremos el mismo path pero con el separador apropiado.

1. El ejemplo Number Guess ya está instalado en la implementación de referencia JSP.
2. Los ficheros .jsp y .html están en el directorio ../jswdk-1.0/examples/jsp/num .
3. Los ficheros .java y .class están en el directorio ../jswdk-1.0/examples/WEB-INF/jsp/beans/num .
4. Abrimos un navegador web y vamos a:
<http://host/examples/jsp/num/numguess.jsp>



Manejar Excepciones

¿Que sucedió la última vez que usamos una aplicación JSP e introdujimos algo incorrectamente? Si la aplicación estaba bien escrita, probablemente lanzaría una excepción y mostraría una página de error. Las excepciones que ocurren durante la ejecución de una aplicación JSP se llaman excepciones en tiempo de ejecución y se describen en este tutorial.

Al igual que en una aplicación Java, una excepción es un objeto que es un ejemplar de `java.lang.Throwable` o de una de sus subclases. `Throwable` tiene dos subclases estándares -`java.lang.Exception`, que describe excepciones, y `java.lang.Error`, que describe errores.

Los errores son diferentes de las excepciones. Los errores normalmente indican problemas de enlaces o de la máquina virtual de los que nuestra aplicación Web podría no recuperarse, como errores de memoria. Sin embargo, las excepciones son condiciones que pueden capturarse y recuperarse de ellas. Estas excepciones podrían ser, por ejemplo, un `NullPointerException` o un `ClassCastException`, que nos dicen que se ha pasado un valor nulo o un dato del tipo erróneo a nuestra aplicación mientras se estaba ejecutando.

Las excepciones en tiempo de ejecución son fáciles de manejar en una aplicación JSP, porque están almacenadas una cada vez en el objeto implícito llamado `exception`. Podemos usar el objeto `exception` en un tipo especial de página JSP llamado página de error, donde mostramos el nombre de la clase `exception`, su seguimiento de pila, y un mensaje informativo para el usuario.

Las excepciones en tiempo de ejecución son lanzadas por el fichero JSP compilado, el fichero class Java que contiene la versión traducida de nuestra página JSP. Esto significa que nuestra aplicación ha sido compilada y traducida correctamente. (Las excepciones que ocurren mientras un fichero está siendo compilado o traducido no son almacenadas en el objeto `exception` y tienen sus mensajes mostrados en la ventana de comandos, en vez de en la página de error. Estas no son el tipo de excepciones descritas en este tutorial.)

Este tutorial describe cómo crear una sencilla aplicación JSP con varias páginas, un componente `JavaBean` y una página de error que ofrece mensajes informativos al usuario. En este ejemplo, el `Bean` sigue la pista sobre la página en la que estaba trabajando el usuario cuando se lanzó la excepción, que nos da a nosotros, el desarrollador, información útil para que podamos mostrar un mensaje informativo. Este es un simple mecanismo de seguimiento de error.

¿Cómo Añadir Páginas de Error?

Aunque las llamemos páginas de error, las páginas especializadas JSP que describimos aquí realmente muestran información sobre excepciones. Para añadir páginas de error que muestren información de excepciones a una aplicación web, seguimos estos pasos:

- Escribimos nuestro Bean (o bean enterprise, servlet, u otro componente) para que lance ciertas excepciones bajo ciertas condiciones.
- Usamos un sencillo mecanismo de seguimiento en nuestro componente para ayudarnos a obtener información sobre lo que estaba haciendo el usuario cuando la excepción fue lanzada. (Si nos movemos en el desarrollo de aplicaciones J2EE, nuestra aplicación podrá grabar el estado, que es la mejor forma de proporcionar información).
- El fichero JSP usa una directiva page con `errorPage` que selecciona el nombre de un fichero JSP que mostrará un mensaje al usuario cuando ocurre una excepción.
- Escribir un fichero de página de error, usando una directiva page con `isErrorPage= "true"`.
- En el fichero de la página de error, usa el objeto `exception` para obtener información sobre la excepción.
- Usamos mensajes informativos, en nuestra página de error o incluida desde otros ficheros, para darle al usuario un mensaje relevantemente informativo sobre lo que el usuario estaba haciendo cuando se lanzó la excepción.

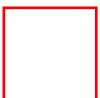
Ejemplo de Buscador de Direcciones de Email

Este ejemplo, llamado `email`, almacena nombres y direcciones de e-mail en un fichero map basado en la clase `java.util.TreeMap` definida en el JDK 1.2. La clase `TreeMap` crea una estructura de datos llamada "red-black tree". En el árbol, los datos son almacenados con una clave y un valor. En este ejemplo, el nombre es la clave y la dirección email el valor.

Cuando añadimos una entrada al fichero map, introducimos tanto un nombre (la clave) como una dirección email (el valor). Podemos buscar o borrar una dirección email introduciendo sólo un nombre. El nombre no puede ser null porque es una clave. Si un usuario intenta introducir un nombre null, la aplicación lanza una excepción y muestra una página de error.

¿Entonces que es un Red-Black Tree?

Para aquellos que seamos curiosos sobre algoritmos, un árbol rojo-negro es un árbol binario extendido que se parece a algo similar a esto (conceptualmente, al menos):



Si estas viendo este documento en la pantalla, veras que algunos nodos son rojos y otros son negros.

El árbol rojo-negro tiene nodos que pueden ser ramas u hojas. Los nodos hojas son

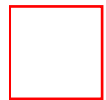
los nodos que hay al final de una línea, mientras que los nodos ramas son los nodos más grandes que conectan con dos o más líneas. Los nodos se almacenan en una estructura compensada en el árbol, usando las siguientes condiciones:

- Cada nodo tiene dos hijos o es una hoja.
- Cada nodo está coloreado en rojo en negro.
- Cada nodo hoja está coloreado en negro.
- Si un nodo es rojo, sus dos hijos son negros.
- Cada camino desde el raíz hasta una hoja contiene el mismo número de nodos negros.

La ventaja de un árbol, para nosotros, los desarrolladores Web, es que podemos crear un fichero map que almacena datos en orden ascendente (ordenados por claves) y que tiene tiempos de búsqueda rápidos.

¿Cómo está Estructurado el Ejemplo?

El ejemplo email tiene tres páginas con formularios HTML, dos ficheros de respuesta, una página de error, y un componente JavaBean. Podemos visualizar la estructura de ficheros en algo como esto:



- Map.java es un componente JavaBeans que crea el fichero map.
- email.jsp es una página JSP que muestra un formulario donde el usuario introduce un nombre y una dirección email.
- lookup.jsp es una página JSP que permite al usuario buscar una dirección email que corresponda con un nombre.
- lookupresponse.jsp está incluido en lookup.jsp y muestra la entrada que el usuario quiere buscar.
- delete.jsp es una página JSP que permite al usuario borrar una dirección email que corresponde con un nombre.
- deleteresponse.jsp está incluido en delete.jsp y muestra la entrada que fue borrada del fichero map.
- error.jsp es una página de error que muestra información sobre manejo de excepciones que ocurren durante la adicción, búsqueda o borrado de entradas en el fichero map.

Añadir un Nombre y una Dirección Email (*email.jsp*)

```
<%@ include file="copyright.html" %>
```

```
<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>
```

```
<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />
<jsp:setProperty name="mymap" property="email" param="email" />
```

```
<% mymap.setAction( "add" ); %>
```

```
<html>
```

```
<head><title>Email Finder</title></head>
```

```
<body bgcolor="#ffffff" background="background.gif" link="#000099">
```

```
<!-- the form table -->
```

```
<form method="get">
```

```
<table border="0" cellspacing="0" cellpadding="5">
```

```
<tr>
```

```
<td width="120"> &nbsp; </td>
```

```
<td align="right"> <h1>Email Finder</h1> </td>
```



```
</tr>
```

```
<tr>
```

```
<td width="120" align="right"><b>Name</b></td>
```

```
<td align="left"><input type="text" name="name" size="35"></td>
```

```
</tr>
```

```
<tr>
```

```
<td width="120" align="right"><b>Email Address</b></td>
```

```
<td align="left"><input type="text" name="email" size="35"></td>
```

```
</tr>
```

```
<tr>
```

```
<td width="120"> &nbsp; </td>
```

```
<td align="right">
```

```
Please enter a name and an email address.
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td width="120"> &nbsp; </td>
```

```
<td align="right">
```

```
<input type="submit" value="Add">
```

```
</td>
```

```
</tr>
```

```
<!-- here we call the put method to add the  
      name and email address to the map file -->
```

```
<%
```

```
String rname = request.getParameter( "name" );
```

```
String remail = request.getParameter( "email" );
```

```
if ( rname != null ) {
```

```
    mymap.put( rname, remail );
```

```
}
```

```
%>
```

```
<tr>
```

```
<td width="120"> &nbsp; </td>
```

```
<td align="right">
```

```
The map file has <font color="blue"><%= mymap.size() %>
```

```
</font> entries.
```

```
</font>
```

```
</td>
```

```
</tr>
```

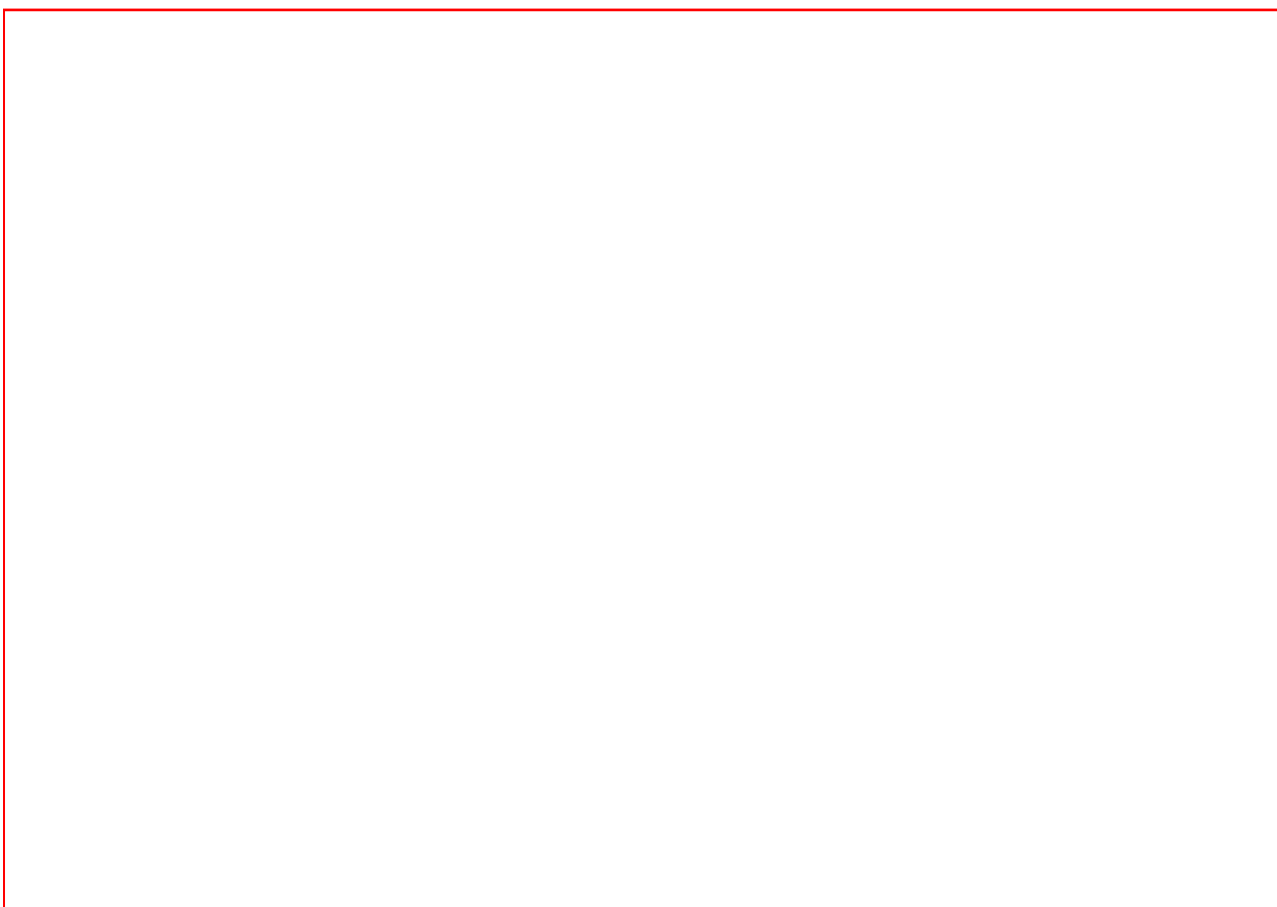
```
<tr>
```

```
<td width="120"> &nbsp; </td>
<td align="right">
<a href="lookup.jsp">Lookup</a>&nbsp; | &nbsp;
    <a href="delete.jsp">Delete</a>
</td>
</tr>

</table>
</form>

</body>
</html>
```

Buscar un Nombre en el Fichero Map (*lookup.jsp*)



```
<%@ include file="copyright.html" %>

<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>

<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />

<% mymap.setAction( "lookup" ); %>
```

```

<html>
<head><title> Email Finder </title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">

<form method="get">
<table border="0" cellspacing="0" cellpadding="5">

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <h1>Email Finder</h1> </td>
</tr>

<tr>
<td width="120" align="right"><b>Name</b></td>
<td align="left"><input type="text" name="name" size="35"></td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
Please enter a name for which
<br>
you'd like an email address.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
The map file has <font color="blue"> <%= mymap.size() %></font>
entries.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <input type="submit" value="Lookup"> </td>
</tr>

<% if ( request.getParameter( "name" ) != null ) { %>
    <%@ include file="lookupresponse.jsp" %>
<% } %>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">

```

```
<a href="email.jsp">Add</a> &nbsp; | &nbsp;  
    <a href="delete.jsp">Delete</a>  
</td>  
</tr>  
</table>  
  
</body>  
</html>
```

Mostrar la Respuesta a la Búsqueda (*lookupresponse.jsp*)



```
<%@ page import="java.util.*, email.Map"    %>  
  
<tr>  
<td width="120"> &nbsp; </td>  
<td align="right">  
<b> Success! </b>  
</td>  
</tr>  
  
<tr>  
<td width="120"> &nbsp; </td>  
<td align="right">  
<jsp:getProperty name="mymap" property="name" />
```

```
<br>
<jsp:getProperty name="mymap" property="email" />
</td>
</tr>
```

Borrar una Dirección Email (*delete.jsp*)



```
<%@ include file="copyright.html" %>

<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>

<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />

<!-- tags the JSP page so that we can display
    the right exception message later -->

<% mymap.setAction( "delete" ); %>

<html>
<head><title> Email Finder </title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">
```

```

<form method="get">
<table border="0" cellspacing="0" cellpadding="5">

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <h1>Email Finder</h1> </td>
</tr>

<tr>
<td width="120" align="right"><b>Name</b></td>
<td align="left"> <input type="text" name="name" size="35"> </td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
Please enter a name you would like to delete.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
The map file has <font color="blue"> <%= mymap.size() %></font>
entries.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <input type="submit" value="Delete"> </td>
</tr>

<!-- display the name and email address, then
      delete them from the map file -->

<%  if ( request.getParameter( "name" ) != null ) {  %>
      <%@ include file="deleterespone.jsp" %>
<%
      mymap.remove( request.getParameter("name") ) ;
    }
%>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<a href="email.jsp">Add</a> &nbsp; | &nbsp;

```

```
        <a href="lookup.jsp">Lookup</a>
    </td>
</tr>

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

Mostrar la Respuesta de Borrado (*deleterespense.jsp*)



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

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <b>Success!</b> </td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
<br><p>
```

has been deleted from the map file.

</td>

</tr>

Mostrar Mensajes de Excepción (*error.jsp*)

```
<%@ include file="copyright.html" %>
```

```
<%@ page isErrorPage="true" import="java.util.*, email.Map" %>
```

```
<jsp:useBean id="mymap" scope="session" class="email.Map" />
```

```
<html>
```

```
<head><title>Email Finder</title></head>
```

```
<body bgcolor="#ffffff" background="background.gif" link="#000099">
```

```
<table border="0" cellspacing="0" cellpadding="5">
```

```
<tr>
```

```
<td width="150" align="right"> &nbsp; </td>
```

```
<td align="right" valign="bottom"> <h1> Email Finder </h1> </td>
```

```
</tr>
```

```
<tr>
```

```
<td width="150" align="right"> &nbsp; </td>
```

```
<td align="right"> <b>Oops! an exception occurred.</b> </td>
```



```
</tr>
```

```
<tr>
```

```
<td width="150" align="right"> &nbsp; </td>
```

```
<td align="right">The name of the exception is  
    <%= exception.toString() %>.
```

```
</td>
```

```
</tr>
```

```
<tr>
```

```
<td width="150" align="right"> &nbsp; </td>
```

```
<td align="right"> &nbsp; </td>
```

```
</tr>
```

```
<% if (mymap.getAction() == "delete" ) { %>
```

```
    <tr>
```

```
        <td width=150 align=right> &nbsp; </td>
```

```
        <td align=right>
```

```
            <b>This means that ...</b>
```

```
            <p>The entry you were trying to
```

```
            <font color="blue">delete</font> is not in the map file <br>
```

```
            <b><i>or</i></b>
```

```
            <br>
```

```
            you did not enter a name to delete.
```

```
            <p>
```

```
            Want to try <a href="delete.jsp">again</a>?
```

```
        </td>
```

```
    </tr>
```

```
<% }
```

```
else if (mymap.getAction() == "lookup" ) { %>
```

```
    <tr>
```

```
        <td width="150" align="right"> &nbsp; </td>
```

```
        <td align="right">
```

```
            <b><i>This means that ...</b></i>
```

```
            <p>the entry you were trying to
```

```
            <font color="blue">look up</font>
```

```
            is not in the map file, <b><i>or</i></b>
```

```
            <br>
```

```
            you did not enter a name to look up.
```

```
            <p>
```

```
            Want to try <a href="lookup.jsp">again</a>?
```

```
        </td>
```

```
    </tr>
```

```
<% }
```

```

else if (mymap.getAction() == "add" ) { %>
    <tr>
    <td width="150" align="right"> &nbsp; </td>
    <td align="right">
    <b><i>This means that ...</b></i>
    <p>You were trying to <font color="blue">add</font>
    an entry with a name of null.
    <br>
    The map file doesn't allow this.
    <p>
    Want to try <a href="email.jsp">again</a>?
    </td>
    </tr>

<% } %>

</table>

```

Crear el Fichero Map (*Map.java*)

```

package email;
import java.util.*;

public class Map extends TreeMap {

    // In this treemap, name is the key and email is the value

    private String name, email, action;
    private int count = 0;

    public Map() { }

    public void setName( String formName ) {
        if ( formName != "" ) {
            name = formName;
        }
    }

    public String getName()
        return name;
    }

    public void setEmail( String formEmail ) {
        if ( formEmail != "" ) {
            email = formEmail;
        }
    }
}

```

```

        System.out.println( name ); // for debugging only
        System.out.println( email ); // for debugging only
    }
}

public String getEmail() {
    email = get(name).toString();
    return email;
}

public void setAction( String pageAction ) {
    action = pageAction;
}

public String getAction() {
    return action;
}
}

```

Manejar Excepciones en el Bean

En este ejemplo, el código que lanza excepciones es la clase `TreeMap`, que extiende nuestro `email.Map`, por eso no tenemos que escribir código que lance excepciones en el Bean.

Los métodos que hemos usado de `TreeMap` son estos con sus excepciones:

- `public Object get(Object key)` throws `ClassCastException`, `NullPointerException`- recupera una entrada de un fichero map.
- `public Object put(Object key, Object value)` throws `ClassCastException`, `NullPointerException`-añade una entrada al fichero map.
- `public Object remove(Object key)` throws `ClassCastException`, `NullPointerException`- elimina una entrada del fichero map.
- `int size()` - devuelve el número de entradas del fichero map.

Por supuesto, si necesitamos más información sobre estos métodos, podemos buscarlos en el API Javadoc por `java.util.TreeMap`.

La clase `TreeMap` lanza una `ClassCastException` cuando el usuario trata de introducir un dato del tipo erróneo en un fichero map, por ejemplo, un `int` donde el fichero map está esperando un `String`. Tengamos en cuenta que la clase `TreeMap` también se usa en aplicaciones cliente Java. En nuestra aplicación JSP, esta aplicación no ocurrirá, porque el usuario introduce un nombre y una dirección email en un formulario HTML, que siempre pasa los datos al Bean como strings. Incluso si el usuario teclea 6 como un nombre, el valor enviado es un `String`.

Sin embargo, los métodos `get`, `put`, y `remove` lanzan una `NullPointerException` si el usuario no introduce nada o se pasa un valor `null` al Bean. Esta la excepción más comun que necesita manejar la aplicación email. Esta excepción podría ocurrir siempre

que el usuario intente añadir, buscar o eliminar una entrada del fichero map. Recuerda que la clave, (en este caso el nombre) no puede ser null.

Cuando el Usuario Intenta Añadir un Valor Null

El primer caso, cuando el usuario intenta añadir un nombre o dirección de email nulos, es manejado por un sencillo código en el Bean y en email.jsp. (Aquí null significa que el usuario no ha introducido nada en la caja de texto del formulario. No maneja el caso en que el usuario teclee uno o dos espacio en blanco, y luego pulsa Return).

El código que maneja la adicción de valores null está en los métodos setName y setEmail de Map.java y en un scriptlet en email.jsp:

Capturar un Valor Null Durante la Adicción

Map.java:

```
public void setName( String formName ) {
    if ( formName != "" ) {
        name = formName;
    }
}
public void setEmail( String formEmail ) {
    if ( formEmail != "" ) {
        email = formEmail;
        System.out.println( name ); // for debugging only
        System.out.println( email ); // for debugging only
    }
}
```

email.jsp:

```
<%
    String rname = request.getParameter( "name" );
    String remail = request.getParameter( "email" );
    if ( rname != null ) {
        mymap.put( rname, remail );
    }
%>
```

Tanto setName como setEmail chequean si el usuario ha introducido un valor en el formulario antes de seleccionar sus respectivas propiedades. Si el formulario es un valor null, el Bean no selecciona ninguna propiedad, el método put no añade nada al fichero map, y no se lanza ninguna excepción.

Cuando el Usuario Intenta Buscar un Valor Null

Pero si vamos a las páginas Lookup o Delete del ejemplo e intentamos buscar o borrar una entrada que no está en el fichero map, la aplicación email lanza una

NullPointerException y muestra una página de error.

Capturar un Valor Null durante la Búsqueda

lookup.jsp:

```
<% if ( request.getParameter( "name" ) != null ) { %>
    <%@ include file="lookupresponse.jsp" %>
<% } %>
```

lookupresponse.jsp:

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<font face="helvetica" size="-2">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
</font>
</td>
</tr>
```

Este ejemplo tiene dos piezas de código que trabajan juntas. La página lookup.jsp, donde introducimos un nombre por el que queremos buscar en el fichero map, tiene un scriptlet que chequea si el usuario ha introducido un nombre en el formulario o no. Si el usuario no ha introducido un nombre o introduce uno que no existe en el fichero map, el Bean lanza una NullPointerException y la aplicación muestra una página de error -- que es el comportamiento deseado! En este caso, podemos estar felices porque se muestra la página de error.

Podríamos haber observado que las líneas del fichero lookupresponse.jsp usan la etiqueta <jsp:getProperty> para recuperar el nombre y la dirección email desde el Bean. También podríamos intentar recuperar la dirección email usando expresiones, algo como esto:

```
<%= request.getParameter( "name" ) %>
<br>
<%= mymap.get( request.getParameter( "name" ) ) %>
```

Si usamos estas líneas, el comportamiento de la aplicación sería un poco diferente. En vez de lanzar una NullPointerException y mostrar una página de error, mostraría el nombre que introdujo el usuario, con la palabra null debajo en la página JSP. En la implementación JSP de Sun, la etiqueta <jsp:getProperty> maneja intencionadamente los valores null de forma diferente que los scriptlets o las expresiones. La forma de manejar los valores Null depende del motor JSP utilizado.

Cuando el Usuario Intenta Borrar un Valor Null

Manejar el caso de un usuario que intenta borrar un valor null es muy similar a manejar la búsqueda de un valor null.

Capturar un Valor Null durante el Borrado

delete.jsp:

```
<% if ( request.getParameter( "name" ) != null ) { %>
    <%@ include file="deleterespone.jsp" %>
<%
    mymap.remove( request.getParameter( "name" ) ) ;
    }
%>
```

deleterespone.jsp:

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<font face="helvetica" size="-2">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
<br><p>
has been deleted from the map file.
</font>
</td>
</tr>
```

Llamar a una Página de Error desde otra Página

Para hacer que las páginas muestren una página de error, cada página de la aplicación email usa una directiva page con el atributo errorPage, de esta forma:

```
<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>
```

En los ejemplos de código, los ficheros que usan esta directiva son email.jsp, lookup.jsp, y delete.jsp. Sólo podemos especificar un página de error por cada página JSP.

Esto significa que podemos diseñar una aplicación JSP para que cada página JSP llame a una página de error diferente, o que varias páginas JSP llamen a un misma página de error. En la aplicación email, varias páginas JSP llaman a un página de error, a sí simplificamos el número de ficheros que necesitamos para mantener una aplicación.

Deberíamos usar al menos una página de error en una aplicación JSP. Si no especificamos una página de error, los mensajes de excepción y el seguimiento de pila

se mostrarán en la ventana de comandos desde la que se arrancó el motor JSP, mientras que el navegador Web mostrará un mensaje de error HTTP no informativo, por ejemplo, un mensaje 404 o 501. Esta definitivamente no es una manera adecuada de manejar excepciones.

Escribir una Página de Error

Una página de error es diferente a una página normal JSP. En una página de error, debemos seleccionar explícitamente el atributo `isErrorPage` de la directiva `page` como `true`. También tendremos acceso al objeto `exception`, que nos dará información sobre la excepción.

Primero, veamos un ejemplo de la directiva `page` de una página de error:

```
<%@ page isErrorPage="true" import="java.util.*, email.Map" %>
```

Una vez que hemos seleccionado `isErrorPage` a `true`, podemos usar el objeto `exception`. `exception` es del tipo `java.lang.Throwable`, por eso podemos usar cualquier método definido en `Throwable` con `exception` en un scriptlet o una expresión, por ejemplo:

- `<% = exception.toString() %>`
- `<% exception.printStackTrace(); %>`

La expresión `exception.toString()` muestra el nombre de la clase de la excepción, por ejemplo, `java.lang.NullPointerException`, mientras que `exception.printStackTrace()` muestra el seguimiento de pila de la excepción. El nombre de la clase y el seguimiento de pila son probablemente muy útiles para nuestro usuario. Para evitar esto, podríamos querer escribir algún tipo de mecanismo de seguimiento para proporcionar información que nos ayude a darle un mensaje informativo a nuestro usuario.

Escribir un Sencillo Mecanismo de Pila

El ejemplo `email` usa una propiedad llamada `action` en `Map.java` para seguir la página en la que el usuario estaba trabajando cuando se lanzó la excepción. Esto nos da información importante para ayudarnos a escribir un mensaje de error informativo para el usuario. El Bean tiene una variable llamada `action`, un método `getAction`, y un método `setAction`. Las declaraciones de variable y métodos en el Bean se parecen a esto:

```
private String action;

public void setAction( String pageAction ) {
    action = pageAction;
}

public String getAction() {
    return action;
}
```

```
}
```

Cada una de las páginas email.jsp, lookup.jsp, y delete.jsp seleccionan el valor de action con una línea como esta (que viene desde email.jsp):

```
<% mymap.setAction( "add" ); %>
```

Si ocurre una excepción, error.jsp chequea el valor de action e incluye el mensaje apropiado para cada valor, usando líneas como estas:

```
<% if (mymap.getAction() == "delete" ) { %>
.. text message here ..
else if (mymap.getAction() == "lookup" ) { %>
.. text message here ..
else if (mymap.getAction() == "add" ) { %>
.. text message here ..
<% } %>
```

Por supuesto, esta es una forma sencilla de implementar seguimiento. Si nos movemos dentro del desarrollo de aplicaciones J2EE con beans enterprise, podemos escribir aplicaciones que graben el estado.

¿Cómo ejecutar el Ejemplo?

Para poder ejecutar este ejemplo, necesitamos tener instalado el JDK 1.2 (si no lo tienes, puedes ir a http://java.sun.com/products/OV_jdkProduct.html.)

Los paths dados aquí son para un sistema UNIX, si estás usando Windows, deberás usar los mismos paths pero con el separador de directorios invertido:

1. Creamos el directorio (o carpeta)
../jswdk-1.0/examples/jsp/tutorial/email.
2. Situamos los siguientes ficheros en el directorio ../tutorial/email:
background.gif, delete.jsp, deleteresponse.jsp, email.jsp, error.jsp, lookup.jsp, lookupresponse.jsp.
3. Creamos el directorio (o carpeta)
../jswdk-1.0/examples/WEB-INF/jsp/beans/email.
4. Situamos los ficheros Map.class y Map.java en el directorio ../beans/email.
5. Arrancamos la implementación de referencia JSP de Sun: cd
../jswdk-1.0startserver
6. Abrimos un navegador Web y vamos a:
<http://yourMachineName:8080/examples/jsp/tutorial/email/email.jsp>