UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DIVISIÓN DE INGENIERÍA ELÉCTRICA

PROGRAMA DE TECNOLOGÍA EN CÓMPUTO

MANUAL DEL CURSO

Java Web

INSTRUCTORES:

Javier Fernandez Ayala

Cuauhtemoc Hohman Sandoval

CONTENIDO .

Introducción a JDBC

Servlets y JSPs

JavaServer Pages (JSP) 1.0

Introducción a JDBC

SQL es un lenguaje usado para crear, manipular, examinar, y manejar bancos de datos relaciónales. SQL es un lenguaje de aplicación específica, por lo que un sólo "statement" o comando puede ser muy expresivo y realizar acciones de alto nivel, tales como ordenar o modificar datos. SQL fue estandarizado en 1992, para que un programa pudiera comunicarse con la mayoría de los sistemas de bases de datos, sin tener que cambiar los comandos SQL. Desafortunadamente, debes conectarte con la base de datos, y cada base tiene una interfase diferente, así como diferentes extensiones de SQL, y estos dependen de cada proveedor. Aquí es donde entra ODBC.

ODBC es una interfase basada en C, y provee una forma consistente para comunicarse con una base de datos basada en SQL. Gracias a ODBC, es posible conectarse a una base de datos y manipularla en una forma estándar.

SQL es un lenguaje que esta bien diseñado para realizar su labor (manejar bases de datos), sin embargo, por esa misma razón, es difícil realizar con el, programas de aplicación en otras áreas, como podría ser el procesamiento de los datos para la generación de un reporte o su despliegue gráfico. Se debe recurrir a otro lenguaje de programación para realizar este tipo de aplicaciones de carácter general, típicamente C++. Sin embargo, el lenguaje C y C++, son dependientes de la plataforma en que se compile, es decir que si un programa se compile en una PC, no correrá en una Macintosh. Y es aquí donde finalmente entra Java.

Un programa Java es totalmente independiente de la plataforma en la que se compile, por lo que puede correr en cualquier plataforma con la Java Virtual Machine (JVM) habilitada, sin tener que ser recompilado. Como sabemos, Java incluye varias librerías estándar, una de ellas es JDBC, que se puede pensar como una versión de ODBC en java. Por las ventajas de Java, JDBC esta siendo desarrollado rápidamente, y muchas compañías han creado y siguen ocupadas creando puentes entre JDBC API y sus sistemas particulares.

Por otra parte, JavaSoft, también nos ha provisto de un puente que traduce JDBC en ODBC, lo cual permite a un programa Java comunicarse a muchas bases de datos que no tienen ni la menor idea de que Java existe.

Es por eso que Java y el puente JDBC-ODBC, nos ofrece una muy buena solución al momento de escribir programas portables que requieran el uso de bases de datos, de aquí la importancia de estudiar el presente capitulo.

1 Empezar con JDBC

Lo primero que tenemos que hacer es asegurarnos de que disponemos de la configuración apropiada. Esto incluye los siguientes pasos.

1.1 Instalar Java y el JDBC en nuestra máquina.

Para instalar tanto la plataforma JAVA como el API JDBC, simplemente tenemos que seguir las instrucciones de descarga de la última versión del JDK (Java Development Kit). Junto con el JDK también viene el JDBC.. El código de ejemplo de demostración del API del JDBC 1.0 fue escrito para el JDK 1.1 y se ejecutará en cualquier versión de la plataforma Java compatible con el JDK 1.1, incluyendo el JDK1.2. Teniendo en cuenta que los ejemplos del API del JDBC 2.0 requieren el JDK 1.2 y no se podrán ejecutar sobre el JDK 1.1.

Podrás encontrar la última versión del JDK en la siguiente dirección: http://java.sun.com/products/JDK/CurrentRelease

1.2 Instalar un driver en nuestra máquina.

Nuestro Driver debe incluir instrucciones para su instalación. Para los drivers JDBC escritos para controladores de bases de datos específicos la instalación consiste sólo en copiar el driver en nuesta máquina; no se necesita ninguna configuración especial.

El driver "puente JDBC-ODBC" no es tan sencillo de configurar. Si descargamos las versiones Solaris o Windows de JDK 1.1, automáticamente obtendremos una versión del driver Bridge JDBC-ODBC, que tampoco requiere una configuración especial. Si embargo, ODBC, si lo necesita. Si no tenemos ODBC en nuestra máquina, necesitaremos preguntarle al vendedor del driver ODBC sobre su instalación y configuración.

1.3 Instalar nuestro Controlador de Base de Datos si es necesario.

Si no tenemos instalado un controlador de base de datos, necesitaremos seguir las instrucciones de instalación del vendedor. La mayoría de los usuarios tienen un controlador de base de datos instalado y trabajarán con un base de datos establecida.

1.4 Contar con una Base de Datos.

Adicionalmente, debemos contar con una base de datos compatible con ODBC que podamos usar. Se debe tener una fuente de datos ODBC "ODBC data source". Para esto se tienen varias opciones, por ejemplo, conectarse con una base de datos Oracle o Sybase. Ponte en contacto con tu administrador si necesitas ayuda a este respecto.

Nota: Para este capitulo se requieren nociones básicas del lenguaje SQL, por lo que si tienes poca experiencia a este respecto, te recomiendo que busques un libro o un manual donde puedas consultar la notación y los comandos utilizados para crear tablas (create), insertar datos (insert) y realizar búsquedas (select). Si no lo tienes a la mano, puedes seguir con el capitulo, ya que esta explicado lo que realizan los statements de SQL utilizados, sin embargo, para realizar tu propias aplicaciones, probablemente requerirás de más comandos y un conocimiento más profundo de las bases de datos.

2 Un ejemplo completo

Ver rápidamente un ejemplo, completo aunque simple, nos ayudara a captar fácilmente los conceptos de JDBC. Cuando intentamos escribir una aplicación Java, basada en una base de datos, en general, nos encontramos con los siguientes puntos:

- **Creando una base de datos**. Puedes crear la base de datos desde fuera de Java, utilizando las herramientas que incluya el vendedor de la base, o desde un programa Java que alimente la base con statements SQL.
- **Conectándose a un ODBC Data Source**. Un ODBC Data Source es una base de datos que esta registrada con el ODBC driver. En Java se puede usar tanto el puente de JDBC a ODBC, como el JDBC al puente específico del vendedor.
- **Introduciendo la información a la base de datos.** Aquí, nuevamente, se puede usar tanto las herramientas que provea el vendedor de la base, como los SQL statements que se manden desde un programa Java.
- **Obtener información de la base.** Aquí se usarán SQL statements para obtener resultados de la base de datos desde un programa Java, y luego se usará Java para desplegarlos o manipularlos.

2.1 Creando una Base de Datos

Para este ejemplo, considérese el interés por saber la cantidad de horas que asisten los programadores de la Facultad de Ingeniería al laboratorio de computación. Un reporte debe ser generado semanalmente que incluya el total de horas y la máxima cantidad de horas que pasa un programador en un día. Aquí se presentan los datos observados:

Horas de trabajo en el laboratorio de Computación, Facultad de Ingeniería

Programador	Día	# Horas
Gilbert	Lun	1
Wally	Lun	2
Edgar	Mar	8
Wally	Mar	2
Eugene	Mar	3
Josephine	Mie	2
Eugene	Jue	3
Gilbert	Jue	1
Clarence	Vie	9
Edgar	Vie	3
Josephine	Vie	4

Para este ejemplo supondremos que existe un ODBC data source con el nombre de LabComp.

Para crear esta base de datos, puedes alimentar el ODBC a través del puente JDBC-ODBC. Para ello debemos crear una aplicación Java que siga los siguientes pasos:

2.1.- Cargar el puente JDBC-ODBC. Debemos cargar el driver que le dirá a las clases JDBC como "hablar" con la fuente de datos. En este caso, se necesitará la clase JdbcOdbcDriver.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

En el caso de que se trate de otro driver el que vaya a ser utilizado, la documentación del driver nos dará el nombre de la clase a utilizar. Por ejemplo, si el nombre de la clase es **jdbc.DriverXYZ**, cargaríamos el driver con esta línea de código.

```
Class.forName("jdbc.DriverXYZ");
```

2.2.- Conectarse a una fuente de datos. Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos. Una URL es usada para conectarse a un JDBC data source particular.

```
Connection con= DriverManager.getConnection(url, "Login", "Password");
```

Este paso también es sencillo, lo más duro es saber qué suministrar para url. Si estamos utilizando el puente JDBC-ODBC, el JDBC URL empezará con **jdbc:odbc:**. el resto de la URL normalmente es la fuente de nuestros datos o el sistema de base de datos. Por eso, si estamos utilizando ODBC para acceder a una fuente de datos ODBC llamada "**Fred,**" por ejemplo, nuestro URL podría ser **jdbc:odbc:Fred**. En lugar de "**Login**" pondríamos el nombre utilizado para entrar en el controlador de la base de datos; en lugar de "**Password**" pondríamos nuestra password para el controlador de la base de datos. Por eso si entramos en el controlador con el nombre "**Fernando**" y la password of "**J8,**" estas dos líneas de código estableceran una conexión.

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernando", "J8");
```

En todo caso, la documentación del driver debe de proporcionarnos las guías para el resto de la URL de la JDBC.

Si uno de los drivers que hemos cargado reconoce la URL suministada por el método **DriverManager.getConnection**, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión detrás de la escena. A menos que estemos escribiendo un driver, posiblemente nunca utilizaremos ningún método del interface **Driver**, y el único método de **DriverManager** que realmente necesitaremos conocer es **DriverManager.getConnection**.

La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **con** es una conexión abierta, y se utilizará en los ejemplos posteriores.

2.3.- Mandar los statements SQL para crear la tabla. Para ello, primero hay que pedirle al objeto **conecction** (con), un objeto de tipo **statement** (stmt).

Statement stmt = con.createStatement();

Con lo que ya podemos ejecutar el siguiente statement SQL para crear una tabla de nombre LabComp:

```
create table LabComp (
  programador varchar (32),
  día char (3),
  horas integer,
  );
```

Para lo que se usa el codigo Java:

```
stmt.execute(    "create table LabComp ("+
          "programador varchar (32),"+
          "día char (3),"+
          "horas integer);"
      );
```

Después de crear la tabla, podemos insertar los valores apropiados, como son:

```
insert into LabComp values ('Gilbert', 'Lun', 1);
insert into LabComp values ('Wally', 'Lun', 2);
insert into LabComp values ('Edgar', 'Mar', 8);
... etc.
```

En seguida se muestra el código completo del programa:

```
import java.sql.*;  //importa todas las clases de JDBC

public class CreateLabComp {

    static String[] SQL = {
        "create table LabComp ("+
            "programador varchar (32),"+
            "dia varchar (3),"+
            "horas integer);",
        "insert into LabComp values ('Gilbert', 'Lun', 1);",
        "insert into LabComp values ('Wally', 'Lun', 2);",
        "insert into LabComp values ('Edgar', 'Mar', 8);",
        "insert into LabComp values ('Wally', 'Mar', 2);",
        "insert into LabComp values ('Eugene', 'Mar', 3);",
        "insert into LabComp values ('Josephine', 'Mie', 2);",
        "insert into LabComp values ('Eugene', 'Jue', 3);",
        "insert into LabComp values ('Gilbert', 'Jue', 1);",
        "insert into LabComp values ('Clarence', 'Vie', 9);",
        "insert into LabComp values ('Edgar', 'Vie', 3);",
        "insert into LabComp values ('Josephine', 'Vie', 4);",
    };

    public static void main(String[] args) {
        String URL = "jdbc:odbc:labComp";
        String username = "";
        String password = "";

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (Exception e) {
            System.out.println("Fallo la carga del driver JDBC/ODBC.");
            return;
        }

        Statement stmt = null;
        Connection con=null;
        try {
            con = DriverManager.getConnection (
```

```
                URL,
                username,
                password);
            stmt = con.createStatement();
        } catch (Exception e) {
            System.err.println("Hubo problemas al conectarse a: "+URL);
        }

        try {
           // Ejecuta el commando SQL para crear tables e insertar datos
            for (int i=0; i<SQL.length; i++) {
                stmt.execute(SQL[i]);
            }
            con.close();
        } catch (Exception e) {
            System.err.println("Problemas con el SQL mandado a "+URL+
                ": "+e.getMessage());
        }

    }
}
```

2.4 Obteniendo información de la Base de Datos

El comando select de SQL, se utiliza para buscar información en la base de datos, mediante el metodo executeQuery de Java, aplicado este, sobre un objeto de tipo statement. Los resultados son devueltos en un objeto de tipo ResultSet, los cuales pueden ser examinados renglón por renglón usando los metodos ResultSet.next y ResultSet.getXXX.

Consideremos ahora la necesidad de obtener el número máximo de tazas de café consumidas por un programador en un día. En terminos de SQL, una forma de hacerlo es ordenar la labra por la columna horas en orden descendente. Seleccionar posteriormente el nombre del programador del primer renglón e imprimirlo.

Esto se puede realizar mediante la sentencia Java:

```
ResultSet result = stmt.executeQuery(
  "SELECT programador, horas
  FROM labComp ORDER BY horas DESC;");
```

Lo que producirá un objeto de tipo **ResultSet** con los siguiente datos:

Clarence	**9**
Edgar	8
Josephine	4
Eugene	3
Eugene	3
Edgar	3
Wally	2
Wally	2
Josephine	2
Gilbert	1
Gilbert	1

Podemos entonces movernos al primer renglón de la tabla mediante:

```
result.next();
```

Y extraer el nombre del programador mediante:

```
String name = result.getString("Programador");
```

Y el número de horas que estuvo:

```
int horas = result.getInt("horas");
```

Que se puede desplegar fácilmente en pantalla por medio de:

```
System.out.println("El programador "+name+
  " estuvo en el laboratorio más tiempo: "+horas+" horas.");
```

resultando en la siguiente salida:

El programador Clarence estuvo en el laboratorio más tiempo: 9 horas.

Ahora, calcular el total de horas en la semana, es sólo cuestión de sumar los datos de la columna horas

Podemos primero seleccionar la columna horas mediante:

```
result = stmt.executeQuery(
  "SELECT horas FROM LabComp;");
```

Ahora nos movemos de renglón en renglón utilizando el método next, sumando los datos, hasta que devuelva falso, indicando que ya no hay más renglones en los datos.

```
horas = 0;
while(result.next()) {
  horas += result.getInt("horas");
}
```

Imprimimos el número total de horas que los programadores pasaron en el laboratorio en la semana:

```
System.out.println("Número total de horas: "+horas);
```

La salida debería de ser:

Número total de horas: 38

En seguida se muestra el programa completo:

```
import java.sql.*;
public class Reporte {
    public static void main (String args[]) {
        String URL = "jdbc:odbc:labComp";
        String username = "";
        String password = "";
         try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
        } catch (Exception e) {
            System.out.println("Fallo la carga del driver JDBC/ODBC.");
            return;
        }
        Statement stmt = null;
        Connection con=null;
        try {
            con = DriverManager.getConnection (URL, username, password);
            stmt = con.createStatement();
        } catch (Exception e) {
            System.err.println("Problemas conectando a: "+URL);
        }
        try {
            ResultSet result = stmt.executeQuery(
                "SELECT proramador, horas FROM labComp ORDER BY horas DESC;");
            result.next();  // se mueve a la primera fila
            String name = result.getString("programador");
            int horas = result.getInt("horas");
            System.out.println("El programador "+name+
                  " estuvo más tiempo en el laboratorio: "+horas+" horas.");
result = stmt.executeQuery(
                "SELECT horas FROM labComp;");    // para cada fila de datos
            horas = 0;
            while(result.next()) {
                horas += result.getInt("horas");
            }
            System.out.println("Total de horas: "+horas+" horas.");
          con.close();
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

2.5 Obteniendo otro tipo de datos de un resultado (MetaData)

Ocasionalmente se necesitará de cierto tipo de información acerca de la tabla de resultados que no son los datos contenidos en ella, sino de la forma en que esta construida, es decir, el número de columnas, el tipo de dato contenido en cada columna, etc.

Por ejemplo, para la sentencia:

```
SELECT * from LabComp
```

Regresará un **ResultSet** con el mismo número de columnas y renglones que la tabla LabComp. Si no conocemos el número de columnas que contiene esta tabla de antemano, podemos utilizar metadata, para averiguarlo.

Continuando con el el ejemplo anterior, obtengamos el número y tipo de columnas que nos regresa la sentencia ya utilizada:

```
ResultSet result = stmt.executeQuery(
    "SELECT programador,
    horas FROM LabComp ORDER BY horas DESC;");
```

Los metadatos de una tabla de resultados, se pueden obtener mediante la siguiente sentencia:

```
ResultSetMetaData meta = result.getMetaData();
```

La cual regresa un objeto de tipo **ResultSetMetaData** con todos los metadatos de la tabla.

Podemos entonces determinar el número de columnas en la tabla de resultados mediante el método `getColumnCount`

```
int columns = meta.getColumnCount();
```

y después ir de columna en columna, imprimiendo su nombre y tipo de dato que contiene mediante los métodos **getColumnLabel** y **getColumnTypeName** , respectivamente.

```
int numbers = 0;
for (int i=1;i<=columns;i++) {
  System.out.println (meta.getColumnLabel(i) + "\t"
                 + meta.getColumnTypeName(i));
  if (meta.isSigned(i)) { // Es un número signado?
        numbers++;
  }
}
System.out.println ("Columns: " +
  columns + " Numeric: " + numbers);
```

En seguida se muestra el código fuente del programa anterior:

```
import java.sql.*; public class JoltMetaData {
    public static void main (String args[]) {
        String URL = "jdbc:odbc:CafeJolt";
        String username = "";
        String password = "";
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (Exception e) {
            System.out.println("Fallo cargar el JDBC/ODBC driver.");
            return;
        }         Statement stmt = null;
        Connection con=null;
        try {
            con = DriverManager.getConnection (
                URL,
                username,
                password);
            stmt = con.createStatement();
        } catch (Exception e) {
            System.err.println("Problemas conectando a "+URL);
        }
        try {
            ResultSet result = stmt.executeQuery(
```

```
                "SELECT programador, horas FROM labComp ORDER BY horas
DESC;");
            ResultSetMetaData      meta      =      result.getMetaData();
int numbers = 0;
            int columns = meta.getColumnCount();
            for (int i=1;i<=columns;i++) {
                System.out.println (meta.getColumnLabel(i) + "\t"
                             + meta.getColumnTypeName(i));
                if (meta.isSigned(i)) { // is it a signed number?
                    numbers++;
                }
            }
             System.out.println("Columnas: " + columns + " Números: " +
numbers);
            con.close();
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

Servlets y JSPs

1. Introducción

1.1 ¿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 comúnmente 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 sitios 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. Instalación y Configuración del Servidor

2.1 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 archivo servlet. Las instrucciones de instalación del JSWDK explican esto, pero básicamente apuntan a poner los archivos **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 estándard del sistema. Si la seleccionamos nosotros mismos necesitamos estar seguros de incluir ".", que significa el directorio actual. Aquí tenemos un rápido sumario 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 está sentencia dentro de nuestro archivo **.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 directoriois, mientras que en Unix se usan dos puntos. Para hacer permanente esta configuración ponemos esta sentencias en el archivo **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.

2.2 Instalar un servidor Web con Capacidad para Servlets

Nuestro siguiente paso es obtener e instalar un servidor Web que soporte servlets, 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. 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. La opcion más popular es **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 redimiento, 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/.

3. Estructura Básica de un Servlet

3.1 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 {

    PrintWriter out = response.getWriter();
  }
}
```

Para ser un servlet, una clase debería extender HttpServlet y sobreescribir 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 peticiòn 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 envíar 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 HttpServlet, 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.

3.2 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.

```
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");
  }
}
```

3.3 Compilar e Instalar el Servlet

Debemos observar que los detalles específicos para instalar servlets varian de servidor en servidor. 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 sudirectorio llamado hall en el directorio **servlets**.

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, deberiamos 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 archivo autoexec.bat después de la línea que selecciona nuestro **CLASSPATH** para apuntar a servlet.jar y jsp.jar.

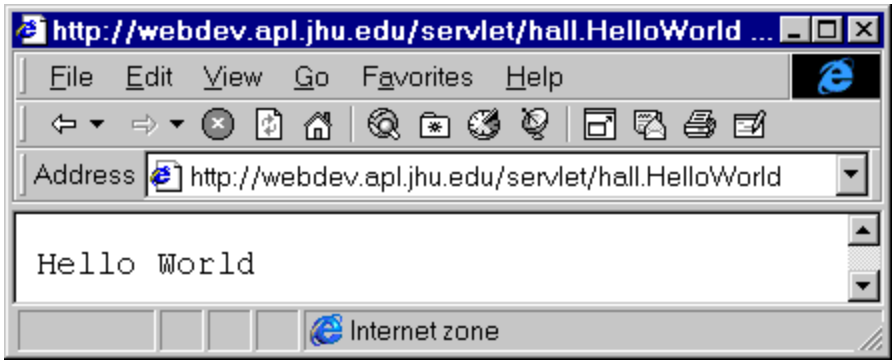
Una segunda forma de compilar las clases que están en paquetes es ir al directorio superior del que contiene los Servlets, y luego hacer "**javac directory\YourServlet.java**". Por ejemplo, supongamos de nuevo que nuestro directorio base es **C:\JavaWebServer\servlets** y que el nombre de nuestro paquete (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 archivos .class. y usar la opción "-d" de javac para instalarlos en la localización que espera el servidor Web.

3.4 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**.



3.5 Un Servlet que Genera HTML

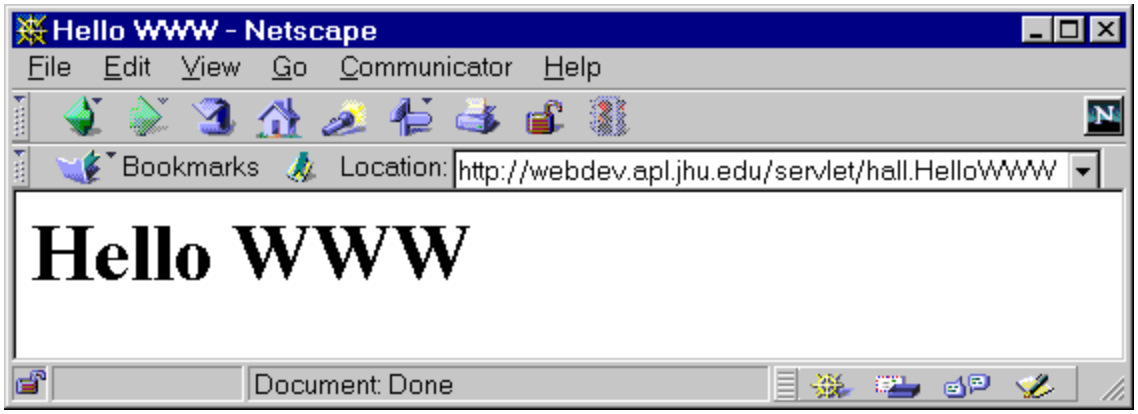
La mayoría de los Servlets generan HTML, no texto normal como el ejemplo anterior. Para hacer esto, necesitamos dos pasos adicionales, decirle al navegador que estámos devolviendo HTML. y modificar la sentencia println para construir una página Web legal. El primer paso se hace configurando la cabecera de respuesta Content-Type. En general, las cabeceras de respuesta se configuran mendiante el método

setHeader de HttpServletResponse, pero seleccionar el tipo de contenido es una tarea muy común y por eso tiene un método especial setContentType sólo para este propósito. Observa que necesitamos configurar las cabeceras de respuesta antes, de devolver algún contenido mediante PrintWriter. Aquí hay un ejemplo:

HelloWWW.java

```
package hall;

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

public class HelloWWW extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                                          "Transitional//EN\">\n" +
                "<HTML>\n" +
                "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Hello WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Resultado de HelloWWW



4. Manejar Datos de Formularios

4.1 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 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**.

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 las 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.

4.2 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 sobreescribiendo service (que llama a doGet, doPost, doHead, etc.). Esta es una buena práctica estándard, 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 estándard. 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 archivos 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.

ThreeParams.java

```
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);
  }
}
```

Salida de ThreeParams

4.3 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 vabr 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 estándard, 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.

ShowParameters.java

```
package hall;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
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);
  }
}
```

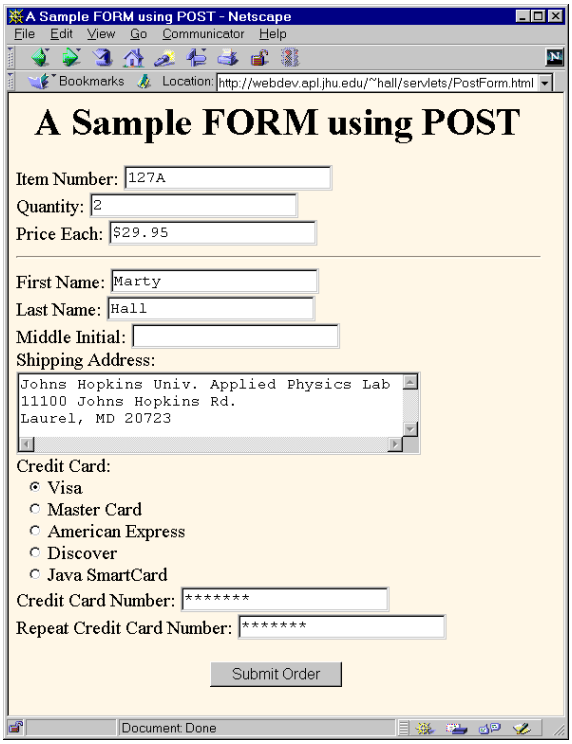
ShowParameters

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

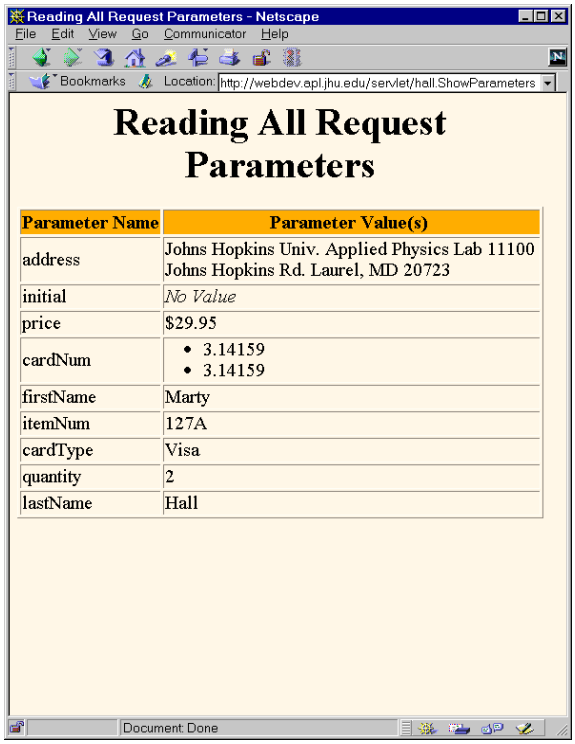
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>A Sample FORM using POST</TITLE>
</HEAD>

<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>
```

Resultados



Resultados de l envío



5 Leer Cabeceras de Solicitud HTTP

5.1 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 decoficarlos. Los servlets pueden checar 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 persisentes, ahorrando un tiempo significante para las páginas Web que incluyen muchas piezas pequeñas (imágenes o clases de applets). Para hacer esto, necesita envíar una cabecera Content-Length en la respuesta, que es fácimente conseguido escribiendo en un ByteArrayOutputStream, y preguntando por el tamaño antes de escribir la salida.
- **Content-Length** (para mensajes POST, cúantos 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 estándard envíadas 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/.

5.2 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 cebecera 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 su 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**".

5.3 Ejemplo: Imprimir todas las Cabeceras

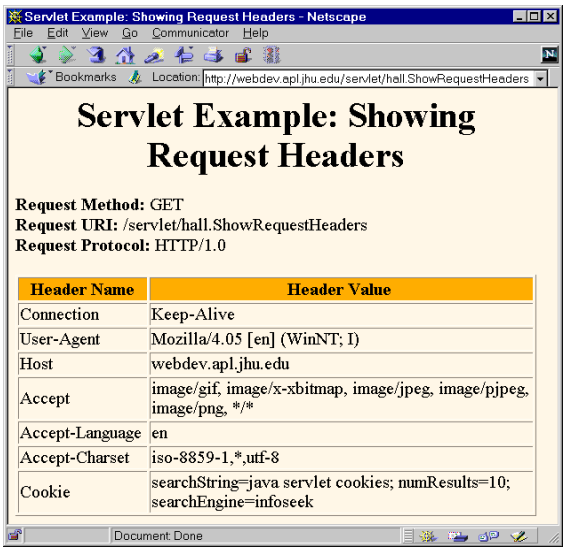
Aquí tenemos un servelt que simplemente crea una tabla con todas las cabeceras recibidas, junto con sus valores asociados. También imprime los tres componentes de la línea de petición principal (método, URI y protocolo).

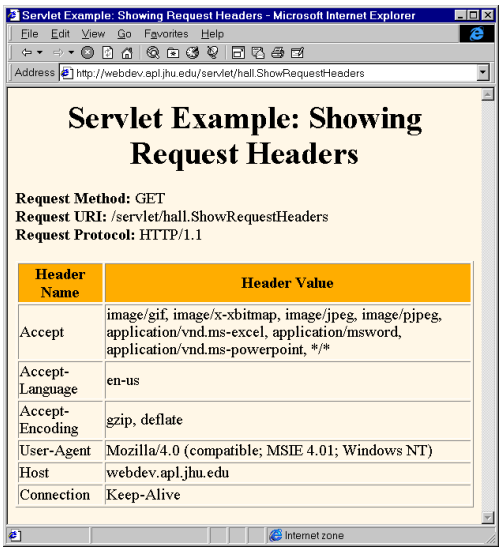
ShowRequestHeaders.java

```
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);
  }
}
```

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.





6 Acceder a Variables Estándards CGI

6.1 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).

6.2 Equivalentes Servlet a la Variables Estándards CGI

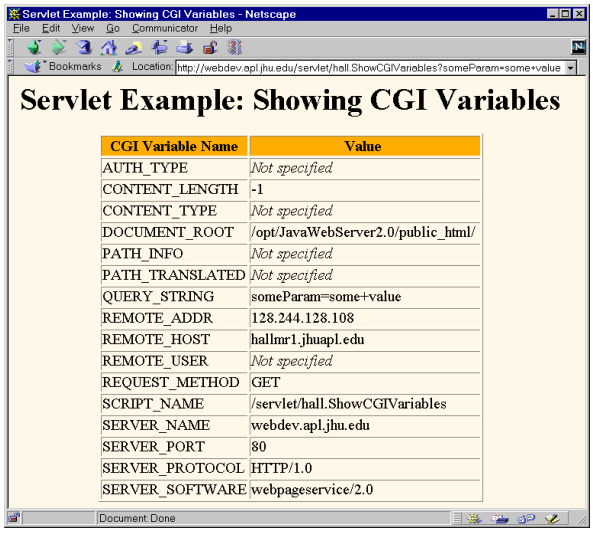
Aunque probablemente tiene más sentido pensar en diferentes fuentes de datos (datos de petición, datos de servidor, etc.) como distintas, los programadores experimentados en CGI podrían encontrar muy útil la siguiente tabla. Asumimos que request es el HttpServletRequest suministrado a los métodos doGet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

ShowCGIVariables.java

```
package hall;

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


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);
  }
}
```

Salida de ShowCGIVariables



7 Códigos de Estado HTTP

7.1 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
```

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 archivo Acrobat, o (más comúnmente) un archivo 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.

7.2 Especificar Códigos de Estado

Como se describe arriba, la línea de estado de respuesta HTPP 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 estándard 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 signifia 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.

7.3 Ejemplo: Motor de Búsqueda

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.

SearchEngines.java

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 {
```

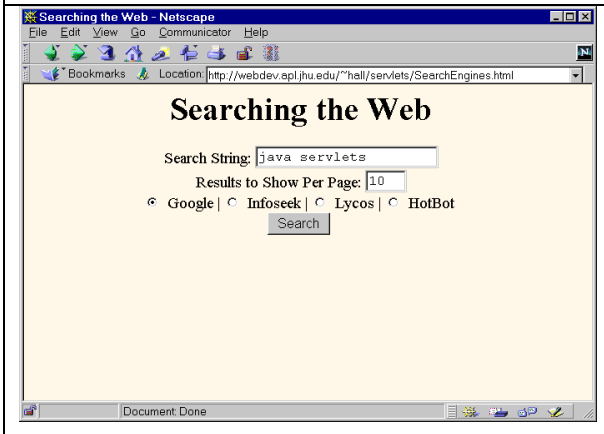
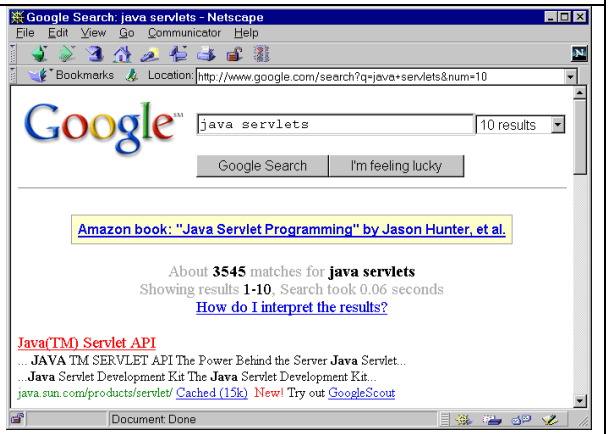
```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    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)) {
        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);
  }
}
```

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);
  }
}
```

SearchSpec.java

```
<!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>
```

Pantalla inicial	Resultados de la Busqueda
	

8 Especificar Cabeceras de Respuesta HTTP

8.1 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 archivo usando conexiones persisitentes, 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 envíar 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 ahora la incoveniencia 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 métodoaddCookie 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.

8.2 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 servelt 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 siemplemente 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).

PrimeNumbers.java

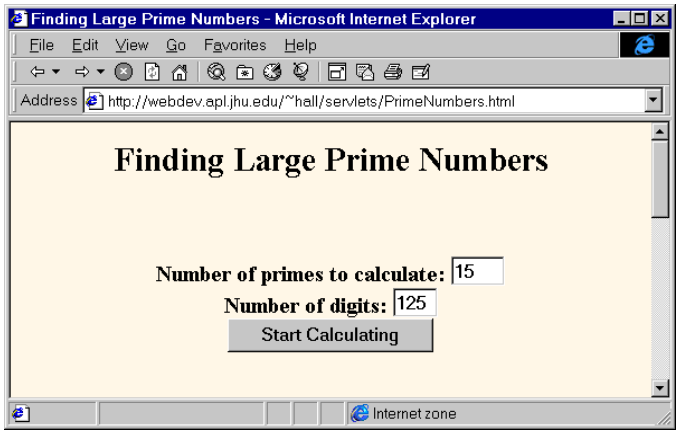
```
package hall;

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);
  }

  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);
    }
  }
}
```

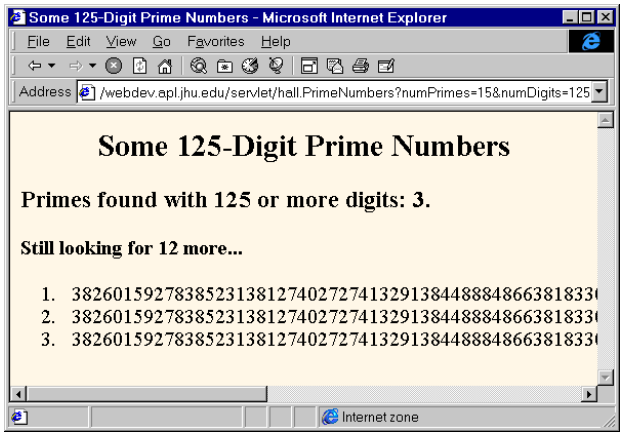
PrimeNumbers.html

```
<!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>
```

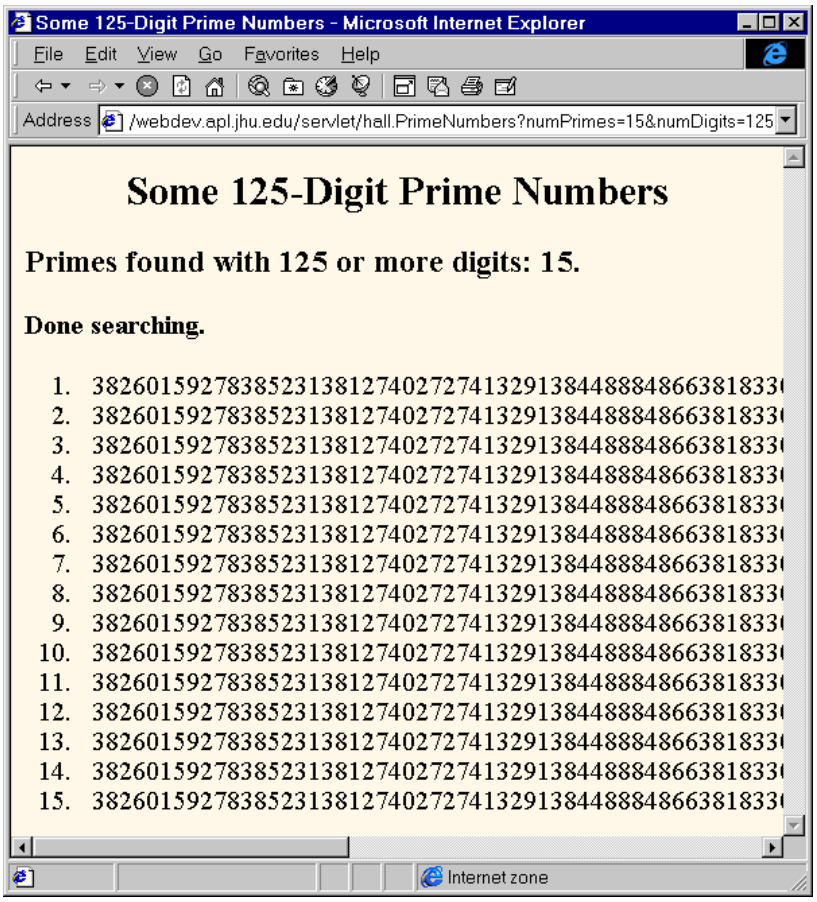
Inicio



Resultados intermedios



Resultado Final



9 Seguimiento de sesion

9.1. ¿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 transaciones 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), 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, depués 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 incluiran 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.

9.2. 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.

9.2.1 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);
```

9.2.2 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 asocidados. En la versión 2.1 y anteriores del API servlet, usamos getValue("key") para buscar un valor préviamente 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 infomación 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.

9.2.3 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 especificacioens 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));
  session.putValue("previousItems", previousItems);
```

9.3 Ejemplo: Mostrar Información de Sesión

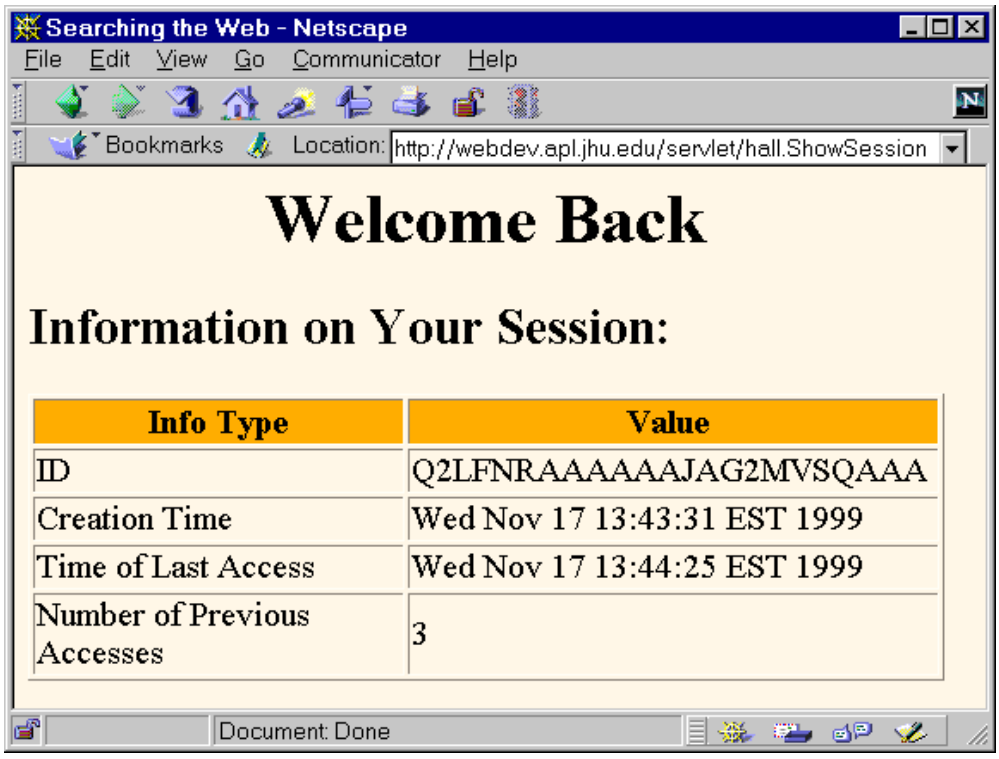
Aquí tenemos un sencillo ejemplo que genera una página Web mostrando alguna información sobre la sesión actual.

```
package hall;

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

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

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://localhost:8080/Orden.jsp?titulo=cien+años+de+soledad:`*

```
Gracias por ordenar:
<I><%= request.getParameter("titulo") %></I>
```

Normalmente daremos a nuestro archivo 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 archivo HTML normal en vez de un servlet, 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. Observa también, que muchos servidores Web nos permiten definir alias para que una URL que parece apuntar a un archivo HTML realmente apunte a un servlet o a una página JSP.

Además del HTML normal, hay tres tipos de construcciones JSP que incluiremos 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.

2. 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 *<%! codigo %>* que se insertan en el cuerpo de la clase del servlet, fuera de cualquier método existente.

2.1 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:

```
Hora actual: <%= 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 envíar la salida al cliente.

Aquí tenemos un ejemplo:

```
El nombre de tu computadora es: <%= 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.

2.2 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("Cadena GET: " + 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 scriplets. Por ejemplo, el siguiente fragmento JSP, contiene una mezcla de texto y scritplets:

```
<% if (Math.random() < 0.5) { %>
Usted Tiene un <B>buen</B> dia!
<% } else { %>
Usted tiene un <B>mal</B> dia!
<% } %>
```

Que se convertirá en algo como esto:

```
if (Math.random() < 0.5) {
  out.println("Usted tiene un<B>buen</B> dia!");
} else {
  out.println("Usted tiene un <B>mal</B> dia!");
}
```

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>
```

2.3 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; %> Numero de accesos al servidor:
<%= ++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>
```

3. 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 archivo dentro de la clase servlet en el momento que el archivo 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.

3.1 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 desacargase cuando esté lleno. Un valor de false, raramente utilizado, indica que se debe lanzar una excepción cuando el buffe 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 precuparnos 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.*" />
```

3.2 La directiva include JSP

Esta directiva nos permite incluir archivos 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 delservidor Web empezando la URL con una barra invertida. Los contenidos del archivo 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 desarroladores el mantenimiento engorroso de copiar realmente el HTML en cada archivo separado. Aquí tenemos un código representativo:

```
<HTML>
<HEAD>
<TITLE> Tutorial JSP 1.0</TITLE>
</HEAD>

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

</BODY>
</HTML>
```

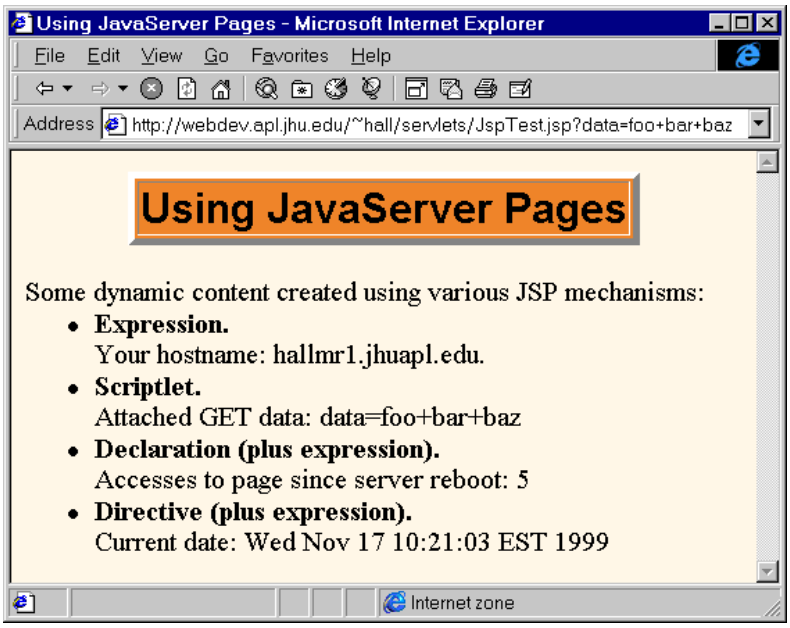
Observa que como la directiva include inserta los archivos en el momento en que la página es traducida, si la barra de navegación cambia, necesitamos re-traducir todas las páginas JSP que la refieren. Esto es un buen compromiso en una situación como esta, ya que las barras de navegación no cambian frecuentemente,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

4. Ejemplo: Usar Elementos de Script y Directivas

Aquí tenemos un sencillo ejemplo que muestra el uso de expresiones, scriptlets, declaraciones y directivas JSP.

```
<HTML>
<HEAD>
<TITLE>Using JavaServer Pages</TITLE>
</HEAD>
<BODY>
      Using JavaServer Pages
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:



5. Variables Predefinidas

Para simplificar el código en expresiones y scriplets JSP, tenemos ocho variables predefinidas, algunas veces llamadas objetos implícitos. Las variables disponibles son:

5.1 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.

5.2 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 envíada al cliente.

5.3 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.

5.4 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.

5.5 application.- Este es el ServletContext obtenido mediante getServletConfig().getContext().

5.6 config.- Este es el objeto ServletConfig para esta página.

5.7 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".

5.8 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.

6. Accciones.-

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

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

6.1 Acción jsp:include

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

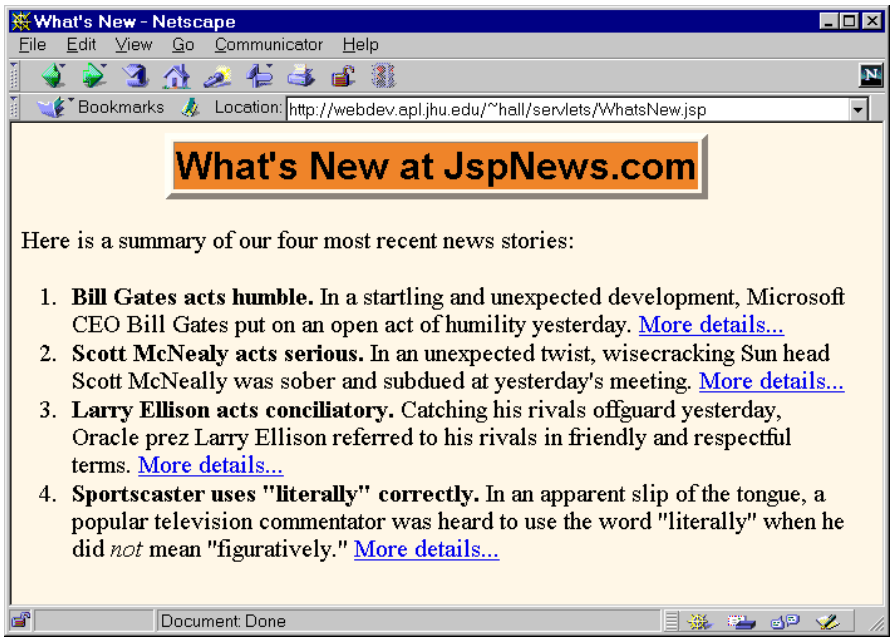
```
<jsp:include page="relative URL" flush="true" />
```

Al contrario que la directiva include, que inserta el archivo en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el archivo 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 significante 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 archivos, pero pueden dejar como estaba la página JSP principal.

WhatsNew.jsp

```
<!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:



6.2 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 síntaxis más simple para especificar que se debería usar un Bean es:

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

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 archivo 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

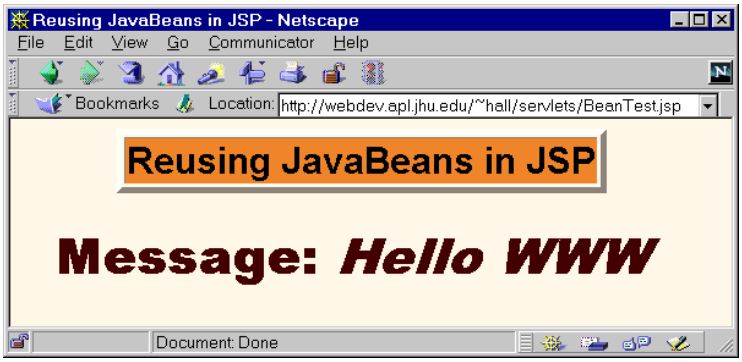
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <HEAD>
<TITLE>Reusing JavaBeans in JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="My-Style-Sheet.css"
      TYPE="text/css">
</HEAD><BODY>
<CENTER>
<table BORDER=5>
  <TR><TH CLASS="TITLE">
      Reusing JavaBeans in JSP</table>
</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.

```
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:



6.3 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: page, request, session, y application. El valor por defecto, page, indica que el bean estará sólo disponible para la página actual (almacenado en el PageContext de la página actual). Un valor de request indica que el bean sólo está disponible para la petición actual del cliente (almacenado en el objeto ServletRequest). Un valor de session indica que el objeto está disponible para todas las páginas durante el tiempo de vida de la HttpSession actual. Finalmente, un valor de application indica que está disponible para todas las páginas que compartan el mismo ServletContext. La razón de la importancia del ámbito es que una entrada jsp:useBean 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 jsp:setParameter u otras entradas entre las etiquetas de inicio jsp:useBean 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.

6.4 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 atibuto 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 estándard 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.

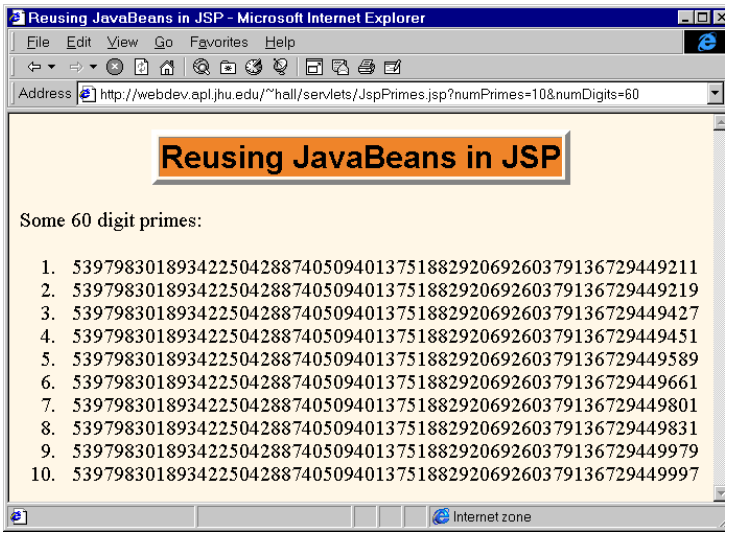
param	Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición actual no tiene dicho parámetro, no se hace nada: el sistema no pasa null al método seleccionador de la propiedad. Así, podemos dejar que el bean suministre los valores por defecto, sobreescribiendolos 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 perámetro numItems de la petición, si existe dicho parámetro, si no existe no se hace nada"
	`<jsp:setProperty name="orderBean"`
	`property="numberOfItems"`
	`param="numItems" />`
	
	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.

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Reusing JavaBeans in JSP</TITLE></HEAD>
<BODY>
      Reusing JavaBeans in JSP
<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:



6.5 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>
```

6.6 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 ejemplo:

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

APENDICES

Utilizar los métodos de ResultSet.getXXX para Recuperar tipos JDBC

Una "x" indica que el método **getXXX** se puede utilizar legalmente para recuperar el tipo JDBC dado.

Una "X" indica que el método **getXXX** está recomendado para recuperar el tipo JDBC dado

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR
getByte	X	x	x	x	x	x	x	x	x	x	x
getShort	x	X	x	x	x	x	x	x	x	x	x
getInt	x	x	X	x	x	x	x	x	x	x	x
getLong	x	x	x	X	x	x	x	x	x	x	x
getFloat	x	x	x	x	X	x	x	x	x	x	x
getDouble	x	x	x	x	x	X	X	x	x	x	x
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x
getBoolean	x	x	x	x	x	x	x	x	x	X	x
getString	x	x	x	x	x	x	x	x	x	x	X
getBytes											
getDate											X
getTime											X
getTimestamp											X
getAsciiStream											X
getUnicodeStream											X
getBinaryStream											
getObject	x	x	x	x	x	x	x	x	x	x	X

	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	VARCHAR
getByte	x	x						
getShort	x	x						
getInt	x	x						
getLong	x	x						
getFloat	x	x						
getDouble	x	x						
getBigDecimal	x	x						
getBoolean	x	x						
getString	X	x	x	x	x	x	x	x
getBytes			X	X	x			
getDate	x	x				X		x
getTime	x	x					X	x
getTimestamp	x	x				x	x	X
getAsciiStream	x	X	x	x	x			
getUnicodeStream	x	X	x	x	x			
getBinaryStream			x	x	X			
getObject	x	x	x	x	x	x	x	X

Sumario de Sintaxis

Elemento JSP	Síntaxis	Interpretación	Notas
Expresión JSP	<%= expression %>;	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 %>;	El código se inserta en el método service.	El equivalente XML es: **<jsp:scriptlet> code </jsp:scriptlet>**.
Declaración JSP	<%! 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	<%@ page att="val" %>	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): • import="**package.class**" • contentType="**MIME-Type**" • isThreadSafe="**true**|false" • session="**true**|false" • buffer="**size**kb|none" • autoflush="**true**|false" • extends="**package.class**" • info="**message**" • errorPage="**url**" • isErrorPage="true|**false**" • language="java"
Directiva include JSP	<%@ include file="url" %>	Un archivo 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 archivo en el momento de la petición en vez del momento de la tradución.
Comentario JSP	<%-- comment --%>	Comentario ignorado cuando se traduce la página JSP en un servlet.	Si queremos un comentario en el HTML resultante, usamos la síntaxis de comentario normal del HTML <-- comment -->.
Acción jsp:include	<jsp:include page="relative URL" flush="true"/>	Incluye un archivo en el momento en que la página es solicitada.	**Aviso:** en algunos servidores, el archivo incluido debe ser un archivo HTML o JSP, según determine el servidor (normalmente basado en la extensión del archivo).
Acción jsp:useBean	<jsp:useBean att=val*/> <jsp:useBean att=val*> ... </jsp:useBean>	Encuentra o crea un Java Bean.	Los posibles atributos son: • id="**name**" • scope="**page**|request|session|application" • class="**package.class**" • type="**package.class**" • beanName="**package.class**"
Acción jsp:setProperty	<jsp:setProperty att=val*/>	Selecciona las propiedades del bean, bien directamenteo o designando el valor que viene desde un	Los atributos legales son: • name="**beanName**" • property="**propertyName**|*"

		parámetro de la petición.	• param="**parameterName**" • value="**val**"
Acción jsp:getProperty	<jsp:getProperty name="propertyName" value="val"/>	Recupera y saca las propiedades del Bean.	
Acción jsp:forward	<jsp:forward page="relative URL"/>	Reenvía la petición a otra página.	
Acción jsp:plugin	<jsp:plugin attribute="value"*> ... </jsp:plugin>	Genera etiquetas OBJECT o EMBED, apropiadas al tipo de navegador, pidiendo que se ejecute un applet usando el Java Plugin.	

Equivalentes Servlet a la Variables Estándares CGI

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 envíados.	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 específica.	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 estándards 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	request.getQueryString()

	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.	
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.	Tecnicamente, 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()

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.

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	¿Cúando 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	¿Cúando (en milisegundos) debería perdir 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 reenvio 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 **response.setHeader("Set-Cookie", ...)**, pero en su lugar usan el método de propósito especial **addCookie** de **HttpServletResponse**.
WWW-Authenticate	¿Qué tipo de autorización y domino debería suministrar el cliente en su cabecera **Authorization**? Esta cabecera es necesaria en respuestas que tienen una línea de estado 401 (Unauthorized). Por ejemplo **response.setHeader("WWW-Authenticate", "BASIC realm=\"executives\"")**.

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

Códigos de Estado HTTP 1.1 y sus Significados

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 **Upgrade** 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 **setStatus**, obtendremos esto.
201	Created	El servidor creo un documento; la cabecera **Location** 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 contínua 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 **Refresh** o su equivalente **<META HTTP-EQUIV="Refresh" ...>**,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 **Range**, 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	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, automáticamentes las traducen a absolutas. 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. Técnicamente, se supone que los navegadores siguen automáticamente la redirección su la petición original era **GET**. Puedes ver la cabecera 307 para más detalles.
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 añádido 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 ambígua: 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 Síntaxis 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 estándard "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 envíar 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 reenvio. 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 prévia especificada en la petición era falsa (Nuevo en HTTP 1.1)
413	Request Entity	El documento pedido es mayor que lo que el servidor quiere manejar ahora. Si el servidor

	Too Large	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 demsiado 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 mentenimiento o sobrecarga. Por ejemplo, un servlet podría devolver esta cabecera si algún almacen 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)