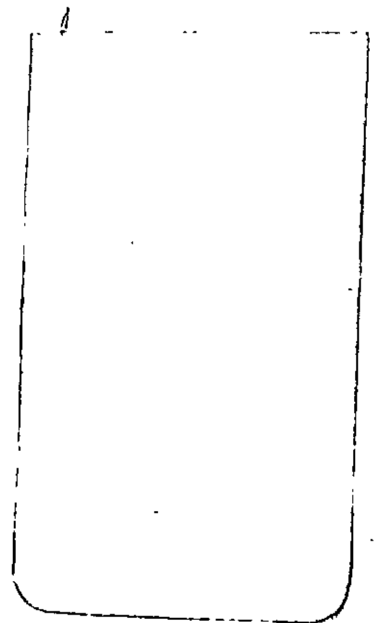
Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Educación Continua

Java Web

Duración: 40 hrs.



Instructor:

Ing. Edgar Eduardo García Cano Castillo.
CA180

Índice

1. JDBC 6
1.1 .... Introducción 6
1.2 Conceptos básicos 6
1.2.2 ¿Qué es JDBC? 6
1.2.3 ¿Qué hace JDBC? 6
1.2.4 JDBC frente a ODBC y otros APIs 7
1.3 Conectividad a base de datos con aplicaciones standalone 8
1.3.1 Drivers JDBC 8
1.3.2 Pasos para usar JDBC 8
1.3.3 Envío de Sentencias SQL 11
2 HTML 15
2.1 Introducción. 15
2.2 Etiquetas. 15
3. Servlets 24
3.1 Protocolo http 24
3.2 El Ciclo de Vida de un Servlet 26
3.2.1 Cargando e instanciando un servlet 27
3.2.2 Inicializar un Servlet 27
3.2.3 Interactuar con Clientes 27
3.2.4 Destruir un Servlet 27
3.2.5 Descargando el Servlet 27
3.3 Servlet API 28
3.3.1 Paquete javax.servlet 28
3.3.2 Paquete javax.servlet.http 28
3.3.3 Peticiones y respuestas. 29
3.4 Interface ServletConfig 31
3.4.1 Métodos de ServletConfig 31
3.5 Interfaz ServletContext 33
3.6 Interfaz RequestDispatcher 35
3.7 Descriptor de aplicaciones 37
3.7.1 Elemento <servlet> 37
3.7.2 Elemento <servlet-mapping> 38
3.8 Manejo de Sesiones 39
3.8.1 Obtener una Sesión 39

3.8.2 Almacenar y Obtener Datos desde la Sesión 39
3.8.3 Invalidar una sesión 40
4. Java Server Page (JSP) 41
4.1 Sintaxis de los JSPs 41
4.1.1 Directivas 41
4.1.2 Declaraciones 42
4.1.3 Scriptlets 42
4.1.4 Expresiones 42
4.1.5 Acciones 43
4.1.6 Comentarios 43
4.2 Atributos de la directiva page 44
4.3 Usando Scriptlets 45
4.4 Objetos implícitos 47
4.5 Alcances de los JSPs 48
4.5.1 Aplication 48
4.5.2 Session 48
4.5.3 Request 48
4.5.4 Page 48
5. JavaBeans 49
5.1 Utilización de JavaBeans en JSP's 49
5.2 Usando JavaBeans y JSPs 50
5.2.1 Declarando JavaBeans usando <jsp:useBean> 50
5.2.2 Inicializando las propiedades del bean 51
5.2.3 Uso de <jsp:setProperty> 52
5.2.3 Uso de <jsp:getProperty> 53
6 JDBC, JSP y Servlets 54
6.1 Integración de los JSPs ySevlets con JDBC 54
6.2 Transacciones 56
7 Patrón Modelo-Vista-Controlador (MVC) 57
7.1 Introducción a patrones 57
7.1.1 Historia de los patrones 57
7.1.2 Clasificación 58
7.2 Definición del Patrón MVC 59
8. Desarrollo de aplicaciones web seguras 60
8.1. Conceptos básicos 60
8.1.1 Autentificación (Authentication) 60

8.1.2 Autorización (Authorization) 60
8.1.3. Integridad de los datos (Data integrity). 60
8.1.4. Confidencialidad (Confidentiality). 60
8.1.5. Auditoria (Auditing). 60
8.1.6. Código malicioso. 61
8.2. Mecanismos de autentificación. 61
8.2.1 HTTP Basic Autentificación 61
8.2.2 HTTP Digest authentication. 62
8.2.3 HTTPS Client authentication. 63
8.2.4 FORM-based authentication. 63
8.2.5 Definición de mecanismos de autenticación para aplicaciones web. 63
8.2.6 Especificando el mecanismo de autentificación. 64
8.3 Haciendo una página Web segura. 65
8.3.1. display-name 66
8.3.2. web-resourse-collection 66
8.3.3. auth-constrainth 66
8.3.4. user-data-constraint 67
8.3.5. Identificando usuario. 68
10. API Java Mail 69
10.1 Introducción 69
10.1 Estructura de un mensaje de correo 70
10.2. Direcciones de correo electrónico. 71
10.3. Protocolos y extensiones relacionadas. 72
10.4. API de JavaMail. 74
10.5. Enviar mensajes 75
10.6 Recibir Mensajes. 80
10.7. Resumen de paquetes y clases. 83
11. XML 87
11.1. Introducción 87
10.2. El API JAXP 95
Apendice. 99
A. Instalación de Oracle 10g Express Edition 99

1. JDBC

1.1 Introducción

Java Database Connectivity (JDBC) es una interfase de acceso a bases de datos estándar SQL que proporciona un acceso uniforme a una gran variedad de bases de datos relacionales. JDBC también proporciona una base común para la construcción de herramientas y utilidades de alto nivel.

El paquete actual de JDK incluye JDBC y el puente JDBC-ODBC.

1.2 Conceptos básicos

1.2.2 ¿Qué es JDBC?

JDBC es el API para la ejecución de sentencias SQL. (Como punto de interés JDBC es una marca registrada y no un acrónimo, no obstante a menudo es conocido como "Java Database Connectivity"). Consiste en un conjunto de clases e interfases escritas en el lenguaje de programación Java. JDBC suministra un API estándar para los desarrolladores y hace posible escribir aplicaciones de base de datos usando un API puro Java.

Usando JDBC es fácil enviar sentencias SQL virtualmente a cualquier sistema de base de datos. En otras palabras, con el API JDBC, no es necesario escribir un programa que acceda a una base de datos Sybase, otro para acceder a Oracle y otro para acceder a Informix. Un único programa escrito usando el API JDBC y el programa será capaz de enviar sentencias SQL a la base de datos apropiada. Y, con una aplicación escrita en el lenguaje de programación Java, tampoco es necesario escribir diferentes aplicaciones para ejecutar en diferentes plataformas. La combinación de Java y JDBC permite al programador escribir una sola vez y ejecutarlo en cualquier entorno.

JDBC expande las posibilidades de Java. Por ejemplo, con Java y JDBC, es posible publicar una página web que contenga un applet que usa información obtenida de una base de datos remota. O una empresa puede usar JDBC para conectar a todos sus empleados (incluso si usan un conglomerado de máquinas Windows, Macintosh y UNIX) a una base de datos interna vía intranet.

1.2.3 ¿Qué hace JDBC?

Simplemente JDBC hace posible estas tres cosas:

- Establece una conexión con la base de datos.
- Envía sentencias SQL
- Procesa los resultados.

El siguiente fragmento de código nos muestra un ejemplo básico de estas tres cosas:

```
Connection con = DriverManager.getConnection (
        "jdbc:odbc:wombat", "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
   int x = rs.getInt("a");
   String s = rs.getString("b");
   float f = rs.getFloat("c");
}
```

1.2.4 JDBC frente a ODBC y otros APIs

En este punto, el ODBC de Microsoft (Open Database Connectvity), es probablemente el API más extendido para el acceso a bases de datos relacionales. Ofrece la posibilidad de conectar a la mayoría de las bases de datos en casi todas las plataformas. ¿Por qué no usar, entonces, ODBC, desde Java?

La respuesta es que se puede usar ODBC desde Java, pero es preferible hacerlo con la ayuda de JDBC mediante el puente JDBC-ODBC. La pregunta es ahora ¿por qué necesito JDBC? Hay varias respuestas a estas preguntas:

1.- ODBC no es apropiado para su uso directo con Java porque usa una interface C. Las llamadas desde Java a código nativo C tienen un número de inconvenientes en la seguridad, la implementación, la robustez y en la portabilidad automática de las aplicaciones.

2.- Una traducción literal del API C de ODBC en el API Java podría no ser deseable. Por ejemplo, Java no tiene punteros, y ODBC hace un uso copioso de ellos, incluyendo el notoriamente propenso a errores "void * ". Se puede pensar en JDBC como un ODBC traducido a una interfase orientada a objeto que es el natural para programadores Java.

3. ODBC es difícil de aprender. Mezcla características simples y avanzadas juntas, y sus opciones son complejas para 'querys' simples. JDBC por otro lado, ha sido diseñado para mantener las cosas sencillas mientras que permite las características avanzadas cuando éstas son necesarias.

4. Un API Java como JDBC es necesario en orden a permitir una solución Java "puro". Cuando se usa ODBC, el gestor de drivers de ODBC y los drivers deben instalarse manualmente en cada máquina cliente. Como el driver JDBC esta completamente escrito en Java, el código JDBC es automáticamente instalable, portable y seguro en todas las plataformas Java.

En resumen, el API JDBC es la interfase natural de Java para las abstracciones y conceptos básicos de SQL. JDBC retiene las características básicas de diseño de ODBC.

1.3 Conectividad a base de datos con aplicaciones standalone

Gracias al API de JDBC tenemos la oportunidad de poder hacer conexión con la base de datos y ocupar cualquier componente que ofrece la plataforma Java, ya sea un Servlet, un JSP, una GUI o simplemente usando una aplicación standalone como se verá a continuación.

1.3.1 Drivers JDBC

Para usar JDBC con un sistema gestor de base de datos en particular, es necesario disponer del driver JDBC apropiado que haga de intermediario entre ésta y JDBC. Dependiendo de varios factores, este driver puede estar escrito en Java puro, o ser una mezcla de Java y métodos nativos JNI (Java Native Interface).

Para cargar o registrar el driver, simplemente se siguen los siguientes pasos:

1. Se agrega a la variable CLASSPATH el archivo que contiene las clases del driver (con su ruta de directorios). Una alternativa a este paso es agregar el archivo JAR al directorio `jre\lib\ext` que se encuentra en el directorio raíz del JDK Standard Edition.

2. Se hace uso del método estático `forName()` de la clase `Class` del paquete `java.lang`:

   ```
   Class.forName("nombre_de_la_clase_del_driver");
   ```

   Por ejemplo, para cargar el driver propio de Sybase es:

   ```
   Class.forName("com.sybase.jdbc2.jdbc.SybDriver");
   ```

   y para cargar el driver de Oracle es:

   ```
   Class.forName("oracle.jdbc.driver.OracleDriver");
   ```

Y así como se ejemplificó para estos dos manejadores de bases de datos, así se realiza para los demás. Posteriormente, ya en el código, se requiere realizar la conexión a la base de datos.

1.3.2 Pasos para usar JDBC

A continuación se muestran los pasos que se tienen que seguir para hacer la conexión a una base de datos de terminada.

1) Crear una instancia del JDBC driver.

```
Class.forName("paquete.driver.nombreDriver");
```

2) Especificar la *url* de la base de datos.

```
String url = "jdbc:subprotocolo:subname"
```

- subprotocolo es un nombre corto que identifica al driver.
- subname es dependiente del DBMS, normalmente contiene el nombre del servidor y el nombre de la base de datos cuando menos.

3) Establecer una conexión usando el driver que crea el objeto *Connection*.

Para crear y establecer una conexión a una base de datos desde JDBC, se requiere utilizar el método estático `getConnection()` de la clase `DriverManager` del paquete `java.sql`:

```
Connection con =
 DriverManager.getConnection(String url, String user, String password);
```

donde el primer argumento es un String que le indica a JDBC el sistema en donde reside la base de datos. La sintaxis estándar del URL de JDBC es de la siguiente manera:

```
jdbc:subprotocolo:servidor.dominio:puerto{/|:}base_de_datos
```

- El método estático `getConnecion` usa el valor de la url como argumento.
- Si se establece la conexión sin problema, regresa un objeto de clase `Connection`.
- Si hay problemas, se genera una `SQLException`.
- El objeto clase `Connection` representa una sesión con una base de datos específica.

Por ejemplo, si se tiene una base de datos Sybase llamada `pubs` en el sistema `odin.fi-b.unam.mx` en el puerto `1521` y cuyo usuario tiene el login `dba` y el password `sql`, entonces la sentencia de conexión es:

```
con = DriverManager.getConnection(
"jdbc:sybase:Tds:odin.fi-b.unam.mx:1521/pubs", "dba","sql");
```

4) Crear un objeto *Statement,* usando *Connection.*

```
Statement stmt = con.createStatement();
```

- El método `createStatement` regresa un objeto de clase `Statement`.
- Si hay problemas, se genera una `SQLException`.
- El objeto clase `Statement` es usado para enviar postulados SQL a la Base de Datos.

5) Armar el postulado SQL y enviarlo a ejecución usando el *Statement*.

```
ResultSet rs = stmt.executeQuery("postulado Select SQL");
```

```
int num = stmst.ExecuteUpdate("otro postulado SQL");
```

• El método `executeQuery` regresa un objeto de clase `ResultSet` que contiene los resultados del query SQL.
• El objeto clase `ResultSet` es usado para interpretar el resultado del query de SQL.
• El método `executeUpdate` regresa un entero que contiene el número de renglones de la tabla relacional afectados por "otro postulado SQL" que puede ser UPDATE, INSERT, DELETE u otro postulado de SQL.
• Si hay problemas, se genera una `SQLException`.

6) Recibir los resultados en el objeto *ResultSet.*

```
while(rs.next()) {
      System.out.println("Columna 1: " + rs.getString(1));
      System.out.println("Columna 2: " + rs.getString(2));
}
```

• Los resultados del Query se almacenan en un conjunto de renglones en el objeto `ResultSet`.
• El objeto `ResultSet` inicialmente apunta a la primera fila.
• El método `next` de `ResultSet` mueve a la siguiente fila.
• Los métodos getXXX de ResultSet permiten acceso a las columnas de la fila apuntada.

Aquí se presentan los requisitos correspondientes para algunos manejadores:

- **DB2**
  - Clase Driver : COM.ibm.db2.jdbc.app.DB2Driver
  - URL de conexión:: jdbc:db2:<database>
  - Archivo .jar/.zip: db2java.zip
- **Sybase**
  - Clase Driver : com.sybase.jdbc2.jdbc.SybDriver
  - URL de conexión: jdbc:sybase:Tds:<host>:<port>/<database>
  - Archivo .jar/.zip: jconn2.jar
- **Oracle**
  - Clase Driver : oracle.jdbc.driver.OracleDriver
  - URL de conexión: jdbc:oracle:thin:@ <host>:<port>:<sid>
  - Archivo .jar/.zip: classes12.zip
- **SQLServer**
  - Clase Driver : com.microsoft.jdbc.sqlserver.SQLServerDriver
  - URL de conexión: jdbc:microsoft:sqlserver://localhost:1433
  - Archivo .jar/.zip: mssqlserver.jar, msbase.jar, msutil.jar
- **PostgreSQL**
  - Clase Driver: org.postgresql.Driver
  - URL de conexión: jdbc:postgresql://<server>:<port>/<database>
  - Archivo .jar/.zip: postgresql.jar

1.3.3 Envío de Sentencias SQL

Una vez que la conexión se haya establecido, se usa para pasar sentencias SQL a la base de datos subyacente. JDBC no pone ninguna restricción sobre los tipos de sentencias que pueden enviarse, esto da un alto grado de flexibilidad, permitiendo el uso de sentencias específicas de la base de datos o incluso sentencias no SQL. Se requiere de cualquier modo, que el usuario sea responsable de asegurarse que la base de datos subyacente sea capaz de procesar las sentencias SQL que le están siendo enviadas y soportar las consecuencias si no es así. Por ejemplo, una aplicación que intenta enviar una llamada a un procedimiento almacenado a una DBMS (sistema manejador de bases de datos) que no soporta procedimientos almacenados no tendrá éxito y generará una excepción. JDBC requiere que un driver cumpla al menos ANSI SQL-2 Entry Level para ser designado JDBC COMPLIANT. Esto significa que los usuarios pueden contar al menos con este nivel de funcionalidad.

JDBC suministra tres clases para el envío de sentencias SQL y tres métodos en la interfaz `Connection` para crear instancias de estas tres clases. Estas clases y métodos son los siguientes:

1.- **Statement** - El objeto de tipo `Statement` se utiliza para crear sentencias SQL estáticas y regresar el resultado que éstas producen. Por lo general, el objeto `Statement` se utiliza para la ejecución de sentencias SELECT y en ese caso, se invoca al método `executeQuery()` el cual regresa un objeto `ResultSet` que contiene los registros resultantes de la consulta:

```
Statement stm = con.createStatement();
ResultSet res =
      stm.executeQuery("select cve_emp,nom_emp from emp");
```

También se utiliza para realizar sentencias INSERT, UPDATE y DELETE y en ese caso, se utiliza el método `executeUpdate()` llevando como argumento el `String` con la sentencia SQL:

```
Statement stm = con.createStatement();
int res = stm.executeUpdate("insert into emp values('123','Juan)");
```

2.- **PreparedStatement** - El objeto de tipo `PreparedStatement` se utiliza para ejecutar sentencias SQL precompiladas (o dinámicas en cuanto a los valores de los parámetros en la condición de la sentencia SQL). Por lo general, `PreparedStatement` se utiliza para la ejecución de sentencias INSERT, DELETE y UPDATE ya que éstas últimas requieren condicionantes por lo que se crea el `String` con la sentencia SQL desde la obtención del objeto `PreparedStatement`. Para establecer el valor de una condicionante, se utilizan los diversos métodos `setXXX()` donde `XXX` representa a cada uno de los diversos tipos de datos que existen en Java (`int`, `byte`, `float`, `double`, `String`, etc) y posteriormente se invoca al método `executeUpdate()` el cual regresa un valor `int` que indica el número de registros afectados por la sentencia (ya sea INSERT, UPDATE o DELETE). Por ejemplo, utilizando el objeto `PreparedStatement`:

```
PreparedStatement pstmt =
```

```
   con.prepareStatement("update emp
                         set nom_emp=?
                          where cve_emp=?)");
pstmt.setString(1,"Juan");
pstmt.setInt(2,123);
int res = pstmt.executeUpdate();
```

Se observa que JDBC sustituye los signos de interrogación (?) por los valores según el orden establecido por el primer argumento del método `setXXX()`. También existe la posibilidad de crear una sentencia SELECT con argumentos asignados dinámicamente así que `PreparedStatement` también tiene el método `executeQuery()` que regresa un objeto `ResultSet`, con los registros resultantes:

```
PreparedStatement pstmt = con.prepareStatement("select cve_emp,nom_emp from emp where
cve_emp=?");
pstmt.setInt(1,123);
ResultSet res = pstmt.executeQuery();
```

3.- **`CallableStatement`** – El objeto de tipo `CallableStatement` se utiliza para ejecutar procedimientos almacenados (*stored procedures*) SQL. La API de JDBC provee una sintaxis para que todos los procedimientos almacenados sean llamados de la misma manera en los diferentes sistemas manejadores de bases de datos. Dicha sintaxis provee una manera que incluye un parámetro de resultado y otra manera que no incluye dicho parámetro. Si el parámetro es utilizado, debe ser registrado como un parámetro OUT. Los parámetros restantes pueden ser utilizados para entrada, salida o ambos. Los parámetros son referenciados a través de un número secuencial, empezando por 1. Las sintaxis son:

```
{?= call <nombre_del_procedimiento_almacenado>[<arg1>,<arg2>, ...]}
{call < nombre_del_procedimiento_almacenado >[<arg1>,<arg2>, ...]}
```

Los valores de los parámetros IN (de entrada) están establecidos por los métodos `setXXX()` heredados de `PreparedStatement`. Los tipos de todos los parámetros OUT (de salida) deben ser registrados ejecutando el procedimiento almacenado, sus valores son recuperados después de la ejecución a través de los métodos `getXXX()` que provee `CallableStatement`.

Un objeto `CallableStatement` también puede regresar un objeto `ResultSet` o múltiples objetos `ResultSet`. En este último caso, los múltiples objetos `ResultSet` son manejados por métodos heredados de `Statement`.

A continuación se muestra el código DBConector.java para hacer una inserción en la base de datos.

```
import java.sql.*;
import java.io.*;


public class DBConector
{

    private Connection conn;
    private Statement sqlStatement;
    private ResultSet rs;
    private ResultSetMetaData rsmd;
    private Statement stm = null;
    private static String sql=null;
    private int rowsAffected = 0;

    public DBConector(String className,String ip,String base,String susuario,String
spass)
    {
        try
        {
            Class.forName(className);
            conn =
DriverManager.getConnection(ip+base+"?user="+susuario+"&password="+spass);
        }
        catch(Exception e)
        {
            System.out.println("Error:"+e.getMessage());
        }
    }

    public void close()
    {
        try{
            conn.close();
        }catch(SQLException e){}
    }

    public int hacerInsert(String sql) throws SQLException
    {
        System.out.println(sql);
        try
        {
            stm = conn.createStatement();
            rowsAffected = stm.executeUpdate(sql);
        }
        catch(Exception e)
        {
            System.out.println("--------------------------------------");
            System.out.println("EXCEPTION GENERADA");
            System.out.println(e.toString());
            System.out.println("--------------------------------------");

        }
        return rowsAffected;
    }
```

```
    public static void main(String args[])
    {
        try
        {
            sql = "insert into alumnos values(DEFAULT,\'edgar\',\'garcia
cano\',\'castillo\')";
            DBConector c = new
DBConector("org.postgresql.Driver","jdbc:postgresql://127.0.0.1:5432/","curso","postgre
s","hola123");

            System.out.println("Resultdo de la inserción"+ c.hacerInsert(sql));
            c.close();
        }
        catch(Exception e)
        {
            System.out.println("Error:"+e);
        }
    }
}
```

A continuación se muestra el método para realizar selecciones sobre una tabla de la base de datos.

```
    public void hacerSelect(String sql) throws SQLException
        {

            try
            {

                sqlStatement = conn.createStatement();
                rs = sqlStatement.executeQuery(sql);

                while(rs.next())
                {
                        System.out.println(rs.getInt(1)+" "+rs.getString(2)+"
"+rs.getString(3)+" "+rs.getString(4));

                }

            }
            catch(SQLException e)
            {
                System.out.println("Error en select : "+e);
            }
            catch(NullPointerException e)
            {
                System.out.println("Error en select : "+e);
            }
        }
```

Para mandar llamar éste método en el main se escribiría lo siguiente:

```
        c.hacerSelect("select * from alumnos");
```

2 HTML

2.1 Introducción.

HTML (*HyperText Markup Language*) es un lenguaje muy sencillo que permite describir hipertexto, es decir, texto presentado de forma estructurada y agradable, con *enlaces* (*hyperlinks*) que conducen a otros documentos o fuentes de información relacionadas, y con *inserciones* multimedia (gráficos, sonido...).

2.2 Etiquetas.

Mi primer pagina

```
<HTML>
<HEAD >
<TITLE>Yo soy el titulo de tu página</TITLE>
</HEAD>
<BODY >
<center><h1>Hola mundo</h1></center>
<p> <p>
Hola bienvenido a tu primer pagina con html
</BODY>
</HTML>
```

Explicación del código fuente:

Al principio de la página

<HTML> lo ultimo es ***</HTML>***

Esto es una manera de decirle a tu navegador que lo que hay en medio de esas dos etiquetas es texto HTML.

De aquí podemos sacar ya varias ideas:

- Cada marca va siempre entre los signos < y >.
- Las Marcas siempre son dobles (casi siempre), una para indicar el principio y otra para el final.
- La marca que indica el final es igual que la del principio pero con el signo / delante.

Dentro de estas Etiquetas (Entre Etiqueta-Principio y Etiqueta-Final) encontramos otro que dice:

<HEAD> </HEAD>

Que indica la cabecera de la página, la cual sirve para meter otras cosas cómo:

<TITLE> Esto-Debería-Ser-El-Título-De-La-Pagina </TITLE>

Que es, como su propio nombre indica (en inglés), el titulo de la Pagina (El que aparece en la ventana del navegador).

Después viene el cuerpo de la página en sí: El texto, imágenes, links, etc. todo ello dentro de los siguientes Etiquetas:

<BODY> </BODY> Aquí va el cuerpo de la página.

<P> Sirve para delimitar un párrafo. Inserta una línea en blanco antes del texto.

<CENTER> </CENTER> Permite centrar todo el texto del párrafo.

<H1> ... </H1> Es la cabecera de nivel 1 y sirve para modificar de manera rápida el tamaño de alguna frase o párrafo.

Una vez hecho esto guarda el archivo con el nombre que quieras pero con extensión con extensión ***. html o *.htm** y cambiando el tipo de archivo de tipo de texto a todos los archivos.

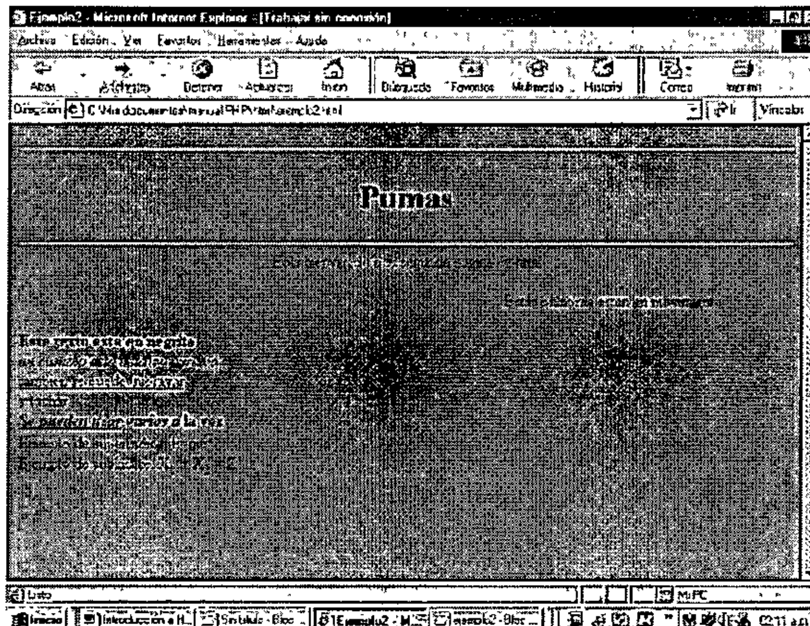
Abre cualquier explorador de Internet, Explorer o Netscape y abre tu archivo desde ahí, o si lo prefieres ve a la carpeta donde se encuentre tu primer pagina y ábrela directamente, si hiciste todo correctamente debe aparecerte con el icono del explorador que tengas instalado en tu maquina.

La salida de la primera página debe verse de esta manera:

Ahora veamos una página un poco menos plana:

```
<HTML>
<HEAD > <TITLE> Ejemplo2 </TITLE> </HEAD>
<BODY bgcolor=#345678>
<hr> <center> <h1> Pumas </h1> </center> <hr>
<center> Hola bienvenido a tu segunda pagina de html </center>
<p><p>
<marquee> Estas palabras están en movimiento</marque>
<B>Este texto esta en negrita</B>
```

```
<BR> <I>en cambio este esta en cursiva</I>
              <BR> <U>también se puede subrayar</U>
<BR> <S>y tachar</S>
<BR> <B><I><U>Se pueden usar</U> varios a</I> la vez</B><BR>
Ejemplo de superíndice: 6<SUP>2</SUP>= 36<BR>
Ejemplo de subíndice: X<SUB>1</SUB> + X<SUB>2</SUB> = Z
</BODY>
</HTML>
```



Explicación de las etiquetas para este ejemplo y algunas más:

Etiqueta	**Utilidad**
<B>... </B>	Pone el texto en negrita.
<I>....</I>	Representa el texto en cursiva.
<U>....</U>	Para subrayar algo.
<S>... </S>	Para tachar.
^{...}	Letra superíndice.
_{...}	Letra subíndice.
<HR> *	Inserta una barra horizontal
 *	Salto de línea
<P>*	Inicio de un párrafo
<marquee>...</marquee>	Desplaza el texto por toda la pantalla
<BIG>... </BIG>	Incrementa el tamaño del tipo de letra.
<SMALL> ... </SMALL>	Disminuye el tamaño del tipo de letra.

<P>	Sirve para delimitar un párrafo. Inserta una línea en blanco antes del texto.
<CENTER> ... </CENTER>	Permite centrar todo el texto del párrafo.
<DIV ALIGN=x> ... </DIV>	Permite justificar el texto del párrafo a la izquierda (ALIGN=LEFT), derecha (RIGHT), al centro (CENTER) o a ambos márgenes (JUSTIFY)
<BLOCKQUOTE> ... Para dejando </BLOCKQUOTE>	citar un texto ajeno. Se suele implementar márgenes tanto a izquierda como a derecha, razón por la que se usa habitualmente.

Etiqueta	**Resultado**
<H1> ... </H1>	Cabecera de nivel 1
<H2> ... </H2>	Cabecera de nivel 2
<H3> ... </H3>	Cabecera de nivel 3
<H4> ... </H4>	Cabecera de nivel 4
<H5> ... </H5>	Cabecera de nivel 5
<H6> ... </H6>	Cabecera de nivel 6

Las etiquetas marcadas con (*) no necesitan ser cerradas.

El “atributo” que se encuentra dentro de la etiqueta body nos permite modificar el color de fondo de pantalla

<body bgcolor=#345678>	Fondo de la página

El tipo de color puede agregarse en forma hexadecimal logrando así varias degradaciones de colores o por el nombre en ingles de dicho color (blue, green, red, gray, etc).

En html algunos caracteres como los acentos, diéresis y algunas letras especiales tiene que escribirse antes de cada letra como se muestra a continuación:

Formularios y links

Los formularios son el sistema del que nos provee el HTML para enviar información desde una página web a algún programa u otro recurso en un ordenador remoto. Esto quiere decir que un formulario no sirve de mucho si no tenemos un lugar al que enviarlo.

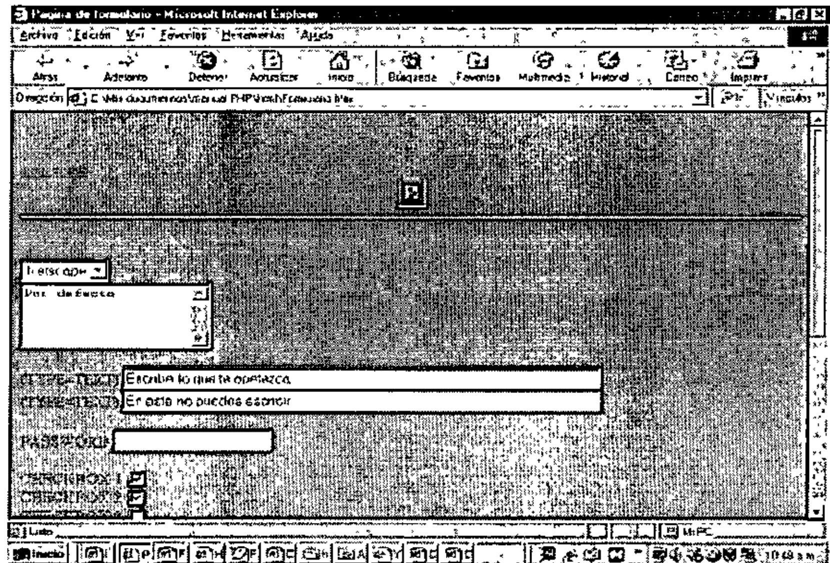
El funcionamiento de los formularios es muy simple: La persona que visita la página rellena el formulario con los datos que se quieran enviar y, al picar en un botón, estos son enviados al lugar que corresponda

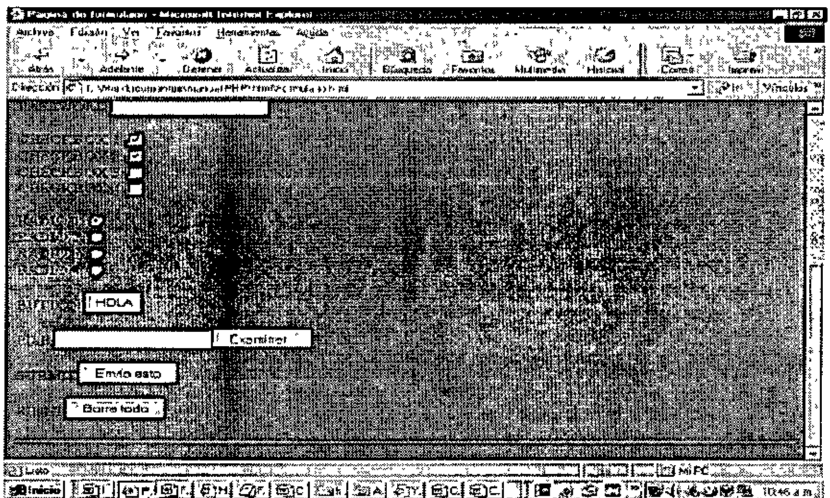
La primera etiqueta que debemos tener en cuenta es el que crea un formulario. Esta etiqueta deberá englobar en su interior a todos los elementos que formen parte de este:

```
<HTML>
<HEAD>
<TITLE>Página de formulario </TITLE>
</HEAD>
<BODY bgcolor=#234567>
<br>
<br>
<a href="ejemplo2.html"> soy un link </A>
<BR>
<CENTER>
<img scr=Dibujo.bmp>
</CENTER>
<hr>
<FORM METHOD="POST" ACTION="ejemplo1.html">
<br>
<SELECT NAME="Navegador">
<OPTION>Netscape
<OPTION>Explorer
<OPTION>Opera
<OPTION>Lynx
<OPTION>Otros
</SELECT>
<br>
<TEXTAREA rows=4 cols=20>Por defecto</TEXTAREA>
<P>(TYPE=TEXT) <INPUT TYPE="TEXT" VALUE="Escribe lo que te apetezca "
SIZE="65" NAME="Text1">
<BR>(TYPE=TEXT) <INPUT TYPE="TEXT" VALUE="En este no puedes
escribir" SIZE="65" NAME="Text2" READONLY>
<P>PASSWORD <INPUT TYPE="PASSWORD" NAME="Password">
<P>CHECKBOX 1 <INPUT TYPE="CHECKBOX" NAME="Checkbox">
<BR>CHECKBOX 2 <INPUT TYPE="CHECKBOX" CHECKED NAME="">
<BR>CHECKBOX 3 <INPUT TYPE="CHECKBOX" NAME="Checkbox">
<BR>CHECKBOX 4 <INPUT TYPE="CHECKBOX" NAME="Checkbox">
<P>RADIO 1 <INPUT TYPE="RADIO" CHECKED NAME="Radio">
```

```
<BR>RADIO 2 <INPUT TYPE="RADIO" NAME="Radio">
<BR>RADIO 3 <INPUT TYPE="RADIO" NAME="Radio">
<BR>RADIO 4 <INPUT TYPE="RADIO" NAME="Radio">
<P>BUTTON <INPUT TYPE="BUTTON" VALUE="HOLA" NAME="Boton">
<P>FILE <INPUT TYPE="FILE" NAME="File">
<P>SUBMIT <INPUT TYPE="SUBMIT" VALUE="Env&iacute;a esto">
<P>RESET <INPUT TYPE="RESET" VALUE="Borra todo">
</FORM>
<hr>
</BODY>
</HTML>
```

Se debe ver así.





Explicación de la página.

<A HREF="destino.htm">pica aquí mismo</A> La etiqueta <A> </A> indica que lo que hay en medio, en este caso la frase "Pica aquí mismo", es un Hipervínculo (Un Link) y HREF="Lo-Que-Sea" indica el destino al que te mandará cuando pique en el con el ratón, en este caso una página que está en el mismo directorio que en la que nos encontramos y con el nombre de destino.htm.

Pero, ¿y si, en vez de en el mismo, está en un subdirectorio? Pues le indicamos la ruta y en paz:

<A HREF="directorio/destino.htm">pica aquí mismo</A>

Si la página está en el directorio justamente superior (el directorio padre), se indicará la ruta de este modo:

<A HREF="../destino.htm">pica aquí mismo</A>

Y si está en el superior a este (algo así como el abuelo), se haría así:

<A HREF="../../destino.htm">pica aquí mismo</A>

Y si está en un directorio "paralelo":

<A HREF="../directoriohermano/destino.htm">pica aquí mismo</A>

<IMG> Esta etiqueta es muy simple, ni siquiera hay que cerrarla (o sea, que no existe </IMG>) y tiene una sola opción estrictamente necesaria que es:

<IMG SRC="nombre de la imagen"> donde "nombre de la imagen" es el nombre del archivo gráfico que se quiere insertar en este punto.

Todos los elementos de un formulario deben estar encerrados entre ***<FORM>*** y ***</FORM>.*** Como parámetros cabe destacar tres. ***ACTION*** define el URL que deberá gestionar el formulario. Puede ser una dirección de correo (precedida del inevitable mailto:, en cuyo caso deberemos añadir el parámetro ENCTYPE="text/plain" para que lo que recibamos resulte legible.

Por otro lado, tenemos el parámetro ***METHOD*** define la manera en que se mandará el formulario. Es recomendable utilizar ***POST***. En el caso de que estemos mandando el formulario a nuestra dirección de correo electrónico es obligado usarlo.

Ahora vamos a ver uno a uno todos los elementos que podemos incluir en un formulario. Veremos que todos ellos tienen algo en común. Como el resultado de cualquier formulario es una lista de variables y valores asignados a las mismas, todos ellos tendrán un atributo en común: el nombre de su variable. El parámetro también será común a todos: NAME.

Menú de selección.

Los parámetros que admite ***SELECT*** son las siguientes:

Parámetro	Utilidad
SIZE	El número de opciones que podremos ver. Si es mayor que 1 veremos una lista de selección y, si no, veremos una lista desplegable.

MULTIPLE	Si lo indicamos podremos elegir más de una opción.

Y ***OPTION*** estos:

Parámetro	**Utilidad**
VALUE	Este es el valor que asignará a la variable.
SELECTED	Si lo indicamos en una de las opciones esta será la seleccionada por defecto.

Campo de texto área.

<TEXTAREA>: Este comando permite al usuario meter más de una línea de texto. El campo es mostrado en caracteres de tamaño fijo y puede ser escroleado si es necesario. El texto mostrado es usado para inicializar el campo. Típicamente los valores de los atributos ROWS y COLS determinan la dimensión visible del campo de caracteres.

Cajas de texto.

Existen tres maneras de conseguir que el usuario introduzca texto en nuestro formulario. Las dos primeras se obtienen por medio de la etiqueta **<INPUT>:**

El primero nos dibujará una caja donde escribir un texto (de una sola línea). El segundo es equivalente, pero no veremos lo que tecleemos en él. Estos son los atributos para modificarlos:

Parámetro	**Utilidad**
SIZE	Tamaño de la caja de texto.
MAXLENGTH	Número máximo de caracteres que puede introducir el usuario.
VALUE	Texto por defecto que contendrá la caja.

"PASSWORD" hace lo mismo que "TEXT", pero los caracteres que se escriban no se verán en pantalla, sino que serán sustituidos por asteriscos.

Si lo que deseamos es que el usuario decida entre varias opciones podremos hacerlo de dos modos. El primero es el que vimos en el ejemplo inicial:

Código	Resultado
3<INPUT NAME="Respuesta" TYPE=RADIO VALUE="mal"> 4<INPUT NAME="Respuesta" TYPE=RADIO VALUE="bien"> 5<INPUT NAME="Respuesta" TYPE=RADIO VALUE="mal"> 	3 ○ 4 ○ 5 ○

Para asociar varios botones de radio a una misma variable les pondremos a todos ellos el mismo NAME. Aparte de esto acepta los siguientes parámetros:

Parámetro	Utilidad
VALUE	Este es el valor que asignará a la variable.
CHECKED	Si lo indicamos en una de las opciones esta será la que esté activada por defecto.

Puede que necesites que el usuario sencillamente nos confirme o niegue algo. Lo podremos conseguir por medio de controles de confirmación:

<INPUT NAME="Belleza" TYPE=CHECKBOX>Me considero guapo/a	☐ Me considero guapo/a

Si queremos que el control esté activado por defecto le añadiremos el parámetro CHECKED. El formulario asignará a la variable NAME el valor *on* u *off*.

Botones del formulario.

Existen dos: uno que se utiliza para mandar el formulario y otro que sirve para limpiar todo lo que haya rellenado el usuario:

<input type="submit" name="Entrar" value="Entrar" /> <input type="reset" name="Resetear" value="Resetear" />	Entrar Resetear

Podemos cambiar el texto que el navegador pone por defecto en esos botones utilizando el parámetro VALUE.

"BUTTON" crea un botón.

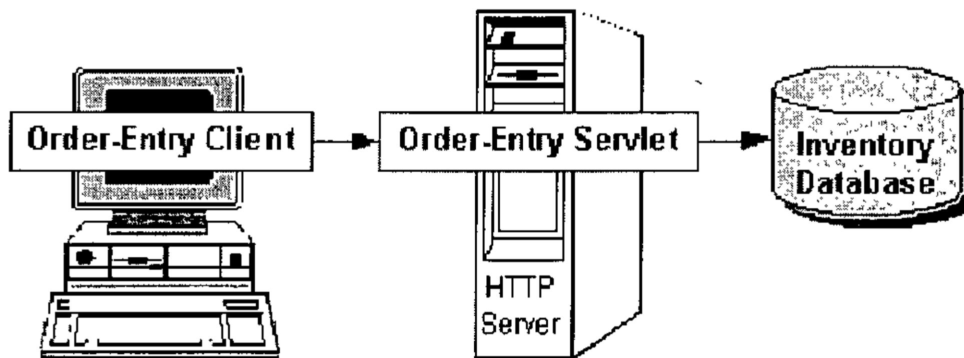
"FILE" sirve para crear un cuadro de diálogo mediante el que enviar un archivo desde tu disco duro. No está soportado en todos los navegadores.

3. Servlets

La tecnología de Servlets de Java es comúnmente usada para manejar la lógica de negocio de una aplicación web. Los Servlets pueden ser aplicados en cualquier protocolo, pero en la práctica, los Servlets son escritos para ser utilizados para el protocolo HTTP.

Los Servlets son módulos que extienden los servidores orientados a petición-respuesta, como los servidores web compatibles con Java. Por ejemplo, un Servlet podría ser responsable de tomar los datos de un formulario de entrada de pedidos en HTML y aplicarle la lógica de negocios utilizada para actualizar la base de datos de pedidos de la compañía.

Los paquetes de clases javax.servlet y javax.servlet.http proveen interfaces y clases que nos van a servir para escribir Servlets. Todos los Servlets deben implementar la interface Servlet, la cual define los métodos del ciclo de vida de los Servlets.



3.1 Protocolo http

El **protocolo de transferencia de hipertexto** (***HTTP***, *HyperText Transfer Protocol*) es el protocolo usado en cada transacción de la Web. El hipertexto es el contenido de las páginas web, y el protocolo de transferencia es el sistema mediante el cual se envían las peticiones de acceso a una página y la respuesta con el contenido. También sirve el protocolo para enviar 'información adicional en ambos sentidos, como formularios con campos de texto.

HTTP es un protocolo sin estado, es decir, que no guarda ninguna información sobre conexiones anteriores. Al finalizar la transacción todos los datos se pierden.

Un mensaje mandado por un cliente a un servidor es llamado HTTP request. La línea inicial de un HTTP request tiene tres partes separadas por espacios.

- Nombre del método.
- La ruta local.
- La versión de http.
-

Una típica línea es la siguiente:

```
GET /reports/sales/index.html HTTP/1.1
```

El nombre del método especifica la acción que el cliente quiere que el servidor ejecute. A continuación se enuncian los principales métodos.

GET

Éste método es usado para traer un recurso. Si se necesita algún parámetro, son pegados al URI.

HEAD

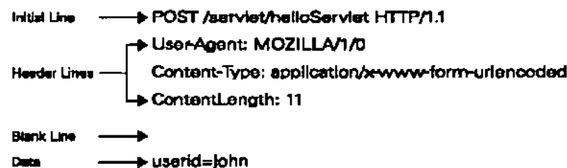
El método HEAD es igual que el método GET, salvo que el servidor no tiene que devolver el contenido, sólo las cabeceras. Estas cabeceras que se devuelven en el método HEAD deberían ser las mismas que las que se devolverían si fuese una petición GET.

Este método se puede usar para obtener información sobre el contenido que se va a devolver en respuesta a la petición. Se suele usar también para chequear la validez de *links*, accesibilidad y modificaciones recientes.

POST

El método POST se usa para hacer peticiones en las que el servidor destino acepta el contenido de la petición como un nuevo subordinado del recurso pedido. El método POST se creó para cubrir funciones como la de enviar un mensaje a grupos de usuarios, dar un bloque de datos como resultado de un formulario a un proceso de datos, añadir nuevos datos a una base de datos.

La función llevada a cabo por el método POST está determinada por el servidor y suele depender de la URI de la petición. El resultado de la acción realizada por el método POST puede ser un recurso que no sea identificable mediante una URI.



PUT

El método PUT permite guardar el contenido de la petición en el servidor bajo la URI de la petición. Si esta URI ya existe, entonces el servidor considera que esta petición proporciona una versión actualizada del recurso. Si la URI indicada no existe y es válida para definir un nuevo recurso, el servidor puede crear el recurso con esa URI. Si se crea un nuevo recurso, debe responder con un código 201 (creado), si se modifica se contesta con un código 200 (OK) o 204 (sin contenido). En caso de que no se pueda crear el recurso se devuelve un mensaje con el código de error apropiado.

La principal diferencia entre POST y PUT se encuentra en el significado de la URI. En el caso del método POST, la URI identifica el recurso que va a manejar en contenido, mientras que en el PUT identifica el contenido.

En general un navegador es un cliente HTTP y los recursos son páginas web, imágenes, Servlets y más. Cada recurso es identificado por un único URI (Uniform Resource Identifier o Identificador Uniforme de Recursos), se pueden encontrar 3 términos que pueden ser intercambiables entre sí: URI,URL y URN. Aunque son similares, tienen ciertas diferencias.

URI: Es una cadena que identifica cualquier recurso. Identificar un recurso (servicio, página, documento, dirección de correo electrónico, enciclopedia, etc), no indica que podamos traerlo, éste término es más general que URL y URN.

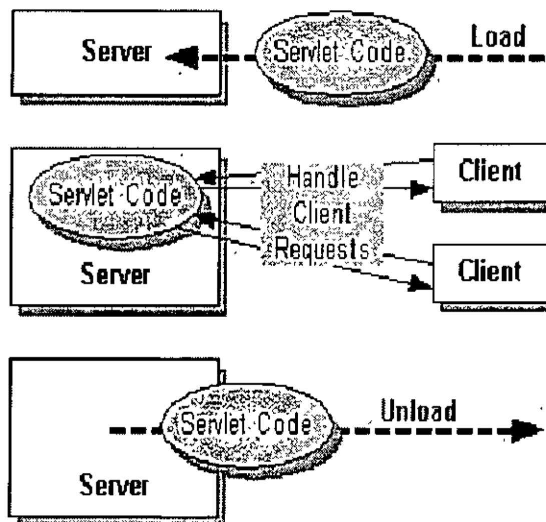
URL: (Uniform Resource Locator o Localizador Uniforme de Recurso) estas son URIs que especifican protocolos comunes como HTTP, FTP y malito. Usado para nombrar recursos, como documentos e imágenes en Internet, por su localización.

URN: (Uniform Resource Name o Nombre Uniforme de Recurso) es un identificador que únicamente identifica un recurso, pero no especifica como accesar a él. URNs son estandarizadas para instituciones oficiales para mantener un único recurso.

3.2 El Ciclo de Vida de un Servlet

Cada servlet tiene el mismo ciclo de vida.

- Un servidor carga e inicializa el servlet.
- El servlet maneja cero o más peticiones de cliente.
- El servidor elimina el servlet. (Algunos servidores sólo cumplen este paso cuando se desconectan).



3.2.1 Cargando e instanciando un servlet

Cuando el contenedor de Servlets se inicia, busca los archivos de configuración de las aplicaciones, también llamados descriptores. Cada aplicación web tiene su propio descriptor llamado `web.xml`, que incluye una descripción de cada Servlet que compone la aplicación. El contenedor de Servlets crea una instancia del Servlet usando `Class.forName(className).newInstance()`. Sin embargo, para hacer lo anterior el Servlet debe de tener un constructor público y sin argumentos, generalmente no se define ningún constructor en la clase.

3.2.2 Inicializar un Servlet

Cuando un servidor carga un Servlet, ejecuta el método `init(ServletConfig)` del Servlet. La inicialización se completa antes de manejar peticiones de clientes y antes de que el Servlet sea destruido. El objeto ServletConfig contiene todos los parámetros de inicialización que se especifican en el descriptor de la aplicación.
Aunque muchos Servlets se ejecutan en servidores multithread, los Servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método `init()`, cuando carga el Servlet, y no lo llamará de nuevo a menos que vuelva a recargar el servlet. El servidor no puede recargar un Servlet sin primero haber destruido el servlet llamando al método `destroy()`.

3.2.3 Interactuar con Clientes

Después de la inicialización, el Servlet puede manejar peticiones de clientes. Esta parte del ciclo de vida de un Servlet se pudo ver en la sección anterior. Después de que la instancia es propiamente inicializada, el Servlet esta listo para dar servicio. Cuando el contenedor recibe una petición para el Servlet, éste la despachará llamando a
`Servlet.service(ServletRequest,ServletResponse)`.

3.2.4 Destruir un Servlet

Los Servlets se ejecutan hasta que el servidor los destruye, por ejemplo, a petición del administrador del sistema. Cuando un servidor destruye un Servlet, ejecuta el método `destroy()` del propio Servlet. Este método sólo se ejecuta una vez. El servidor no ejecutará de nuevo el Servlet, hasta haberlo cargado e inicializado de nuevo.

3.2.5 Descargando el Servlet

Una vez destruido, la instancia puede ser recogida por el recolector de basura. Si el Servlet ha sido destruido porque el Servicio del contenedor se paro, el Servlet también es descargado.

3.3 Servlet API

La especificación de los Servlets de Sun proveen un estándar e independencia de la plataforma para poder realizar comunicación entre los Servlets y sus contenedores. Las grupo de clases e interfaces es llamado Servlet Application Programming Interfaces (Servlet API).

El Servlet API está dividido en dos paquetes: javax.servlet y javax.servlet.http.

3.3.1 Paquete javax.servlet

Este paquete contiene las interfaces y clases genéricas de los Servlets que son independientes de cualquier plataforma.

Interface javax.servlet.Servlet

Esta es la interface central en el API de los Servlets. Cada Servlet implementa directa o indirectamente esta interfaz y contiene los métodos:

- init()
- service()
- destroy()
- getServletConfig()
- getServletInfo()

Clase javax.servlet.GenericServlet

Es una clase abstracta que tiene implementación para todos los métodos excepto `service()`.

Interface javax.servlet.ServletRequest

Provee una vista genérica de la petición que fue enviada por un cliente. Define métodos para obtener la información de la petición.

Interface javax.servlet.ServletResponse

Provee una forma genérica de envíar respuestas. Posee métodos que permiten envíar respuestas al cliente.

3.3.2 Paquete javax.servlet.http

Este paquete provee la funcionalidad básica requerida para los HTTP Servlets. Las clases e interfaces de éste paquete tienen soporte para utilizar el protocolo HTTP.

Clase javax.servlet.http.HttpServlet

Es una clase abstracta que extiende de GenericServlet y asigna un método service con la siguiente firma.

```
protected void service(HttpServletRequest,HttpServletResponse) throws ServletException,
IOException
```

Interface javax.servlet.http.HttpServletRequest

Esta interface hereda de ServletRequest y prove formas específicas HTTP para las peticiones. Define métodos para obtener información como cabeceras o cookies en la petición.

Interface javax.servlet.http.HttpServletRequest

Esta interface hereda de `ServletResponse` y provee fomas específicas para de envío de respuestas HTTP. Define métodos para el envío de información como cabeceras y cookies en la respuesta.

3.3.3 Peticiones y respuestas.

Un Servlet HTTP maneja peticiones del cliente a través de su método `service`. Este método soporta peticiones estándar de cliente HTTP despachando cada petición a un método designado para manejar esa petición.

Los métodos de la clase `HttpServlet` que manejan peticiones de cliente toman dos argumentos.

- Un objeto `HttpServletRequest`, que encapsula los datos desde el cliente.
- Un objeto `HttpServletResponse`, que encapsula la respuesta hacia el cliente.

Objetos HttpServletRequest

Un objeto `HttpServletRequest` proporciona acceso a los datos de cabecera HTTP, como cualquier cookie encontrada en la petición, y el método HTTP con el que se ha realizado la petición. El objeto `HttpServletRequest` también permite obtener los argumentos que el cliente envía como parte de la petición.

Para acceder a los datos del cliente:

El método `getParameter` devuelve el valor de un parámetro nombrado. Si nuestro parámetro pudiera tener más de un valor, deberíamos utilizar `getParameterValues` en su lugar. El método `getParameterValues` devuelve un arreglo de valores del parámetro nombrado. (El método `getParameterNames` proporciona los nombres de los parámetros.

Objetos HttpServletResponse

Un objeto HttpServletResponse proporciona dos formas de devolver datos al usuario:

- El método getWriter devuelve un Writer.
- El método getOutputStream devuelve un ServletOutputStream.

Se utiliza el método getWriter para devolver datos en formato texto al usuario y el método getOutputStream para devolver datos binarios.

Usando PrintWriter.

Utilizando el método getWriter() regresa un objeto java.io.PrintWriter que es usado para envíar datos. PrintWriter es extensivamente usado por los Servlets para generar páginas HTML dinámicas.

Una vez que se ha analizado el request, el Servlet puede decidir si quiere redireccionar a algún recurso. Para realizqar éste trabajo HttpServletResponse provee el método sendRedirect() que se muestra a continuación.

```
if("companynews".equals(request.getParameter("news_category")))
{
//retrieve internal company news and generate
//the page dynamically
}
else
{
response.sendRedirect("http://www.cnn.com");
}
```

En el ejemplo se checa news_category para tomar la decisión de qué hacer. Lo que no se puede hacer con sendRedirect() es enviar un flujo de información ya que generará una IllegalStateException.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
      PrintWriter pw = res.getWriter();
      pw.println("<html><body>Hello World!</body></html>");
      pw.flush();        <- Envia la respuesta
      res.sendRedirect("http://www.cnn.com"); <- Trata de direccionar
}
```

En el ejemplo anterior se esta forzando al contenedor de Servlets enviar una cabecera y luego se direcciona, lo que hace que mande una excepción.

3.4 Interface ServletConfig

Como ya se vió en el ciclo de vida del servlet, el contenedor de Servlets pasa un objeto de tipo `ServletConfig` en el método `init(ServletConfig)`, a continuación se detallará la manera en que trabja `ServletConfig`. Hay que notar que `ServletConfig` sólo tiene métodos para traer los parámetros, no para envíar.

3.4.1 Métodos de ServletConfig

Método	Descripción
`String getInitParameter(String paramName)`	Regresa sólo un valor asociado con el parámetro dado o null si no esta disponible.
`Enumeration getInitParameterNames()`	Regresa un Enumeration de Strings de todos los nombres de los parámetros.
`ServletContext getServletContext()`	Regresa el ServletContext de éste servidor.
`String getServletName()`	Regresa el nombre del servlet especificado en el archivo de configuración.

A continuación se muestra un ejemplo de uso de `ServletConfig`.

a. Especificando el archivo web.xml

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


<web-app>
      <servlet>
           <servlet-name>TestServlet</servlet-name>
           <servlet-class>TestServlet</servlet-class>
           <init-param>
                 <param-name>driverclassname</param-name>
                 <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
           </init-param>
           <init-param>
                  <param-name>dburl</param-name>
                  <param-value>jdbc:odbc:MySQLODBC</param-value>
           </init-param>
           <init-param>
                  <param-name>username</param-name>
                  <param-value>testuser</param-value>
           </init-param>
           <init-param>
                  <param-name>password</param-name>
                  <param-value>test</param-value>
           </init-param>
           <load-on-startup>1</load-on-startup>
      </servlet>
</web-app>
```

b. Código de TestServlet.java

```
import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class TestServlet extends HttpServlet
{
    Connection dbConnection;

    public void init(){

    System.out.println(getServletName()+": Initializing...");
    ServletConfig config = getServletConfig();
    String driverClassName = config.getInitParameter("driverclassname");
    String dbURL = config.getInitParameter("dburl");
    String username = config.getInitParameter("username");
    String password = config.getInitParameter("password");

    //Load the driver class
    Class.forName(driverClassName);

    //get a database connection
    dbConnection = DriverManager.getConnection(dbURL,username,password);
    System.out.println("Initialized.");
    }

    public void service(HttpServletRequest req,HttpServletResponse res)throws
ServletException, java.io.IOException{

    //get the requested data from the database and
    //generate an HTML page.
    }

    public void destroy()
    {
    try
    {
        dbConnection.close();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    }
}
```

3.5 Interfaz ServletContext

Se puede pensar que la interface `ServletContext` es la ventana para que un Servlet vea su entorno. El Servlet puede usar esta interface para obtener información como la inicialización de parámetros. Cada aplicación web tiene un y sólo un `ServletContext`. El contexto es inicializado al mismo tiempo que la aplicación es cargada.

La información del contexto del servidor web está disponible en cualquier momento a través de un objeto `ServletContext`. Un Servlet puede obtener este objeto llamando al método `getServletContext()` del objeto `ServletConfig` y hay que recordar que este objeto se pasa al servlet en el método `init()`.

La interface `ServletContext` define varios métodos, como se puede ver a continuación:

Método	Descripción
`getAttribute()`	Obtiene información acerca del servidor en pares de atributos nombre/valor.
`getMimeType`	Regresa el tipo MIME de un archivo dado.
`getRealPath()`	Este método convierte una ruta relativa o virtual a una nueva ruta en relación con el directorio raíz de lós documentos HTML.
`getServerInfo`	Regresa el nombre y la versión de los servicios de red en los que está ejecutándose el servlet.
`getServlet()`	Regresa un objeto Servlet de un nombre dado. Útil cuando se desea tener acceso a los servicios de otro servlet.
`getServletNames()`	Regresa un objeto Enumeration con los nombres de los servlets disponibles en el actual espacio de nombres.
`log()`	Escribe información al archivo de bitácora. Ese archivo y su formato son específicos de cada servidor web.

El siguiente código de ejemplo muestra la manera en que un servlet utiliza el servidor para escribir un mensaje al archivo de bitácora cuando el servlet se inicializa:

```
private ServletConfig config;
public void init(ServletConfig config) {
  // Store config in an instance variable
  this.config = config;
  ServletContext sc = config.getServletContext();
  sc.log( "Started OK!" );
}
```

Los parámetros definidos para el contexto en el descriptor de la aplicación web, van dentro de las etiquetas <context-param>.

```
<web-app>
    ...
    <context-param>
        <param-name>dburl</param-name>
        <param-value>jdbc:databaseurl</param-value>
    </context-param>
    ...
<web-app>
```

Los métodos de ServletContext que sirven para obtener los parámetros de inicialización son.

Método	Descripción
String getInitParameter(String paramName)	Regresa sólo un valor asociado con el parámetro dado o null si no esta disponible.
Enumeration getInitParameterNames()	Regresa un Enumeration de Strings de todos los nombres de los parámetros.

Interface javax.servletServletContextListener

Ésta interface permite al desarrollador saber cuando el servlet context es inicializado o destruido. Por ejemplo se puede querer crear una conexión a una base de datos tan pronto como el contexto sea inicializado y cerrarla cuando el contexto sea destruido.

Método	Descripción
String contextDestroyed(ServletContextEvent sce)	Llamado cuando el contexto es destruido.
String contextInitialized(ServletContextEvent sce)	Llamado cuando el contexto es iniciado.

A continuación se muestra un ejemplo que implementa la interface ServletContextListener

```
package com.abcinc;
import javax.servlet.*;
import java.sql.*;

public class MyServletContextListener implements ServletContextListener
{
    public void contextInitialized(ServletContextEvent sce)
    {
        try
        {
            Connection c = //crea conexión a la base de datos;
            sce.getServletContext().setAttribute("connection", c);
        }catch(Exception e) { }
    }

    public void contextDestroyed(ServletContextEvent sce)
    {
        try
        {
            Connection c = (Connection)
```

```
        sce.getServletContext().getAttribute("connection");
        c.close();
    }catch(Exception e) { }
  }
}
```

Para agregar Listeners en el descriptor de la aplicación se hace lo siguiente:

```
<listener>
   <listener-class>
      com.abcinc.MyServletContextListener
</listener-class>
</listener>
```

3.6 Interfaz RequestDispatcher

Un componente web puede invocar a otros recursos web de 2 maneras: indirecta y directamente. Un componente web invoca indirectamente a otros recursos web cuando se agrega a su respuesta enviada al cliente un URL que apunta a otro componente web.

Un componente web en ejecución también puede invocar a otros recursos y existen 2 posibilidades en este caso: puede incluir el contenido de otro recurso o puede redireccionar una petición al otro recurso.

Para invocar un recurso disponible en el servidor en el que se está corriendo el componente web, primero se debe obtener un objeto `RequestDispatcher` utilizando el método `getRequestDispatcher("URL")`. Se puede obtener un objeto `RequestDispatcher` ya sea desde una petición o desde el contexto web, sin embargo, ambas alternativas se comportan de manera un poco diferente. El método `getRequestDispatcher` toma la ruta del recurso solicitado como un argumento. Una petición puede tomar una ruta relativa (esto es, una ruta que no comienza con una diagonal /), pero el contexto web requiere una ruta relativa. Si el recurso no está disponible, o si el servidor no ha implementado un objeto `RequestDispatcher` para ese tipo de recurso, el método `getRequestDispatcher` regresará null. Se aconseja que los servlets deban estar preparados para tratar esta situación.

Por lo general, es útil incluir otros recursos web en la respuesta regresada por un componente web, como por ejemplo texto o imágenes, un banner o información de los derechos de autor. Para incluir otros recursos, se invoca al método `include` de un objeto `RequestDispatcher`:

```
include(request, response);
```

Si el recurso es estático (como texto fijo o imágenes), el método `include` "incrusta" el recurso desde la ejecución del servlet en el servidor (server-side include). Si el recurso es un componente

web, el efecto del método es enviar la petición al componente web incluido, ejecuta el componente web y entonces incluye el resultado de esa ejecución en la respuesta del servlet contenedor. Un componente web incluido tiene acceso al objeto request, pero está limitado en lo que puede hacer con el objeto response:

- Puede escribir en el cuerpo de la respuesta (objeto response) y realizar la asignación (commit) de una respuesta.
- No puede establecer o configurar cabeceras ni llamar a algún método (por ejemplo `setCookie`) que afecte a las cabeceras de la respuesta.

En algunas aplicaciones se desea tener un componente web que realice procesamiento preliminar de una petición y tener otro componente que genere la respuesta. Para transferir el control a otro componente web, se debe invocar al método `forward` de un objeto `RequestDispatcher`. Cuando una petición es redireccionada, el URL de la petición se establece a la ruta de la nueva página a la que se direccionó la petición. Si el URL original es requerido para algún proceso, se puede salvar como un atributo más de la petición.

El método `forward` debe ser utilizado para dar a otro recurso la responsabilidad de responderle al cliente. Si el servlet ha accesado a un objeto `ServletOutputStream` o a un objeto `PrintWriter`, utilizar el método `forward` lanzará una excepción `IllegalStateException`.

3.7 Descriptor de aplicaciones

El descriptor de una aplicación, como su nombre lo indica describe la aplicación para el contendor de Servlets la entienda. A continuación se muestra un archivo `web.xml`.

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

<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" >

  <display-name>Test Webapp</display-name>

    <context-param>
      <param-name>author</param-name>
      <param-value>john@abc.com</param-value>
    </context-param>

  <servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>com.abc.TestServlet</servlet-class>
    <init-param>
      <param-name>greeting</param-name>
      <param-value>Good Morning</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>test</servlet-name>
    <url-pattern>/test/*</url-pattern>
  </servlet-mapping>

  <mime-mapping>
```

Declares the XML version and character set used in this file

Declares the schema definition for this file

Specifies a parameter for this web application

Specifies a servlet

Specifies a parameter for this servlet

Maps /test/* to test servlet

CENTRO DE INFORMACION
Y DOCUMENTACION
"ING. BRUNO MASCANZONI"

3.7.1 Elemento <servlet>

Cada `<servlet>` debajo de `<web-app>` define un servlet en la aplicación, en la siguiente imagen se muestra el uso de esta etiqueta.

```
<servlet>

  <servlet-name>us-sales</servlet-name>

  <servlet-class>com.xyz.SalesServlet</servlet-class>

  <init-param>
    <param-name>region</param-name>
    <param-value>USA</param-value>
  </init-param>

  <init-param>
```

The servlet name

The servlet class

The servlet parameters

El contenedor de servlets hace una instancia y lo asocia con el nombre del servlet.

<servlet-name> : Esta etiqueta define el nombre del servlet.
<servlet-class> : Esta etiqueta especifica la clase Java que será usada por el contenedor de Servlets.
<init-param>: Esta etiqueta sirve para agregar parámetros iniciales al servlet, se tiene que crear una de éstas por cada parámetro que se necesite. Contiene las etiquetas <param-name> que indica el nombre del parámetro y <param-value> que indica el calor del parámetro. En el servlet los parámetros se obtienen con `ServletConfig.getInitParameter("paramname")`.

3.7.2 Elemento <servlet-mapping>

Simplemente hay que especificar el URL que será manejada por el Servlet. El contenedor de Servlets utiliza este mapeo para invocar el Servlet apropiado.

El siguiente es un ejemplo que muestra cómo se hace el mapeo de un Servlet.

```
<servlet-mapping>
   <servlet-name>accountServlet</servlet-name>
   <url-pattern>/account/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
   <servlet-name>accountServlet</servlet-name>
   <url-pattern>/myaccount/*</url-pattern>
</servlet-mapping>
```

En estos mapeos se esta asociando /account y /myaccount con accountServlet.

3.8 Manejo de Sesiones

El seguimiento de sesión es un mecanismo que los Servlets utilizan para mantener el estado sobre la serie de peticiones desde un mismo usuario (esto es, peticiones originadas desde el mismo navegador) durante algún periodo de tiempo.

Las sesiones son compartidas por los Servlets a los que accede el cliente. Esto es conveniente para aplicaciones compuestas por varios Servlets.

Para utilizar el seguimiento de sesión debemos.

- Obtener una sesión (un objeto `HttpSession`) para un usuario.
- Almacenar u obtener datos desde el objeto `HttpSession`.
- Invalidar la sesión (opcional).

3.8.1 Obtener una Sesión

El método `getSession` del objeto `HttpServletRequest` devuelve una sesión de usuario. Cuando llamamos al método con su argumento `create` como `true`, la implementación creará una sesión si es necesario. Para mantener la sesión apropiadamente, debemos llamar a `getSession` antes de escribir cualquier respuesta. En el siguiente código se muestra la creación de una sesión.

```
public class CatalogServlet extends HttpServlet {
      public void doGet (HttpServletRequest request,HttpServletResponse response)
      throws ServletException, IOException
      {
       // Get the user's session and shopping cart
       HttpSession session = request.getSession(true);
       ...
       out = response.getWriter();
       ...
       }
}
```

3.8.2 Almacenar y Obtener Datos desde la Sesión

El Interface `HttpSession` proporciona métodos que almacenan y recuperan:

- Propiedades de Sesión estándar, como un identificador de sesión.
- Datos de la aplicación, que son almacenados como parejas nombre-valor, donde el nombre es un `string` y los valores son objetos del lenguaje de programación Java. Como varios Servlets pueden acceder a la sesión de usuario, deberemos adoptar una convención de nombrado para organizar los nombres con los datos de la aplicación.

```
public class CatalogServlet extends HttpServlet {

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

        HttpSession session = request.getSession(true);

        //obteniendo el shopping cart.

        ShoppingCart cart = (ShoppingCart)session.getValue(session.getId());

        if (cart == null) {
        cart = new ShoppingCart();
        session.putValue(session.getId(), cart);
    }
    ...
    }
}
```

Una sesión puede ser designada como nueva cuando hace que el método `isNew` de la clase `HttpSession` devuelva `true`, indicando que, por ejemplo, el cliente, todavía no sabe nada de la sesión. Una nueva sesión no tiene datos asociados.

3.8.3 Invalidar una sesión

Una sesión de usuario puede ser invalidada manual o automáticamente, dependiendo de donde se esté ejecutando el servlet. (Por ejemplo, el Java Web Server, invalida una sesión cuando no hay peticiones de página por un periodo de tiempo, unos 30 minutos por defecto). Invalidar una sesión significa eliminar el objeto `HttpSession` y todos sus valores del sistema.
Para invalidar manualmente una sesión, se utiliza el método `invalidate` de "session". Algunas aplicaciones tienen un punto natural en el que invalidar la sesión.

```
public class ReceiptServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, IOException {
    ...
    scart = (ShoppingCart)session.getValue(session.getId());
    ...
    // Clear out shopping cart by invalidating the session
    session.invalidate();
    // set content type header before accessing the Writer
    response.setContentType("text/html");
    out = response.getWriter();
    ...
    }
}
```

4. Java Server Page (JSP)

Una página JSP es una página Web que contiene código Java insertado dentro del código HTML. Todo el código Java es interpretado del lado del servidor, esto implica que el cliente no puede ver ningún tipo de código, lo único que ve es una página HTML.

Gracias a un JSP que tiene incrustado código Java se pueden hacer páginas dinámicas. Una página dinámica es aquella que no tiene un contenido fijo, si no, que éste puede variar con el tiempo. El contenido dinámico puede ser obtenido de una base de datos, de archivos, etc.

4.1 Sintaxis de los JSPs

Como cualquier otro lenguaje, los JSPs tienen su gramática bien definida. La siguiente tabla muestra los elementos pertenecientes a los JSPs.

Tipo de etiqueta	Descripción	Sintaxis
Directivas	Información acerca de los JSPs	<%@ Directivas %>
Declaraciones	Declara y define métodos y variables	<%! Declaraciones Java %>
Scriptlets	Permiten escribir código Java	<% Código Java %>
Expresiones	Sirve para imprimir variables	<%= Expresión %>
Acciones	Comandos al motor JSP	<jsp:acción />
Comentarios	Usado para documentar	<%-- Texto --%>

4.1.1 Directivas.

Poseen información acerca de las páginas JSPs a la máquina JSP. Existen tres tipos de directivas mencionadas a continuación.

page: Informa al motor acerca de las propiedades de un JSP.

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

include: Informa al motor que incluya los contenidos de otro archivo en la página actual.

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

taglib: Se usa para asociar un taglib.

```
<%@ taglib prefix="test" uri="taglib.tld" %>
```

4.1.2 Declaraciones.

Declaran y definen variables y métodos que pueden ser usados en el JSP. Las variables sólo son inicializadas una vez y retiene el valor para los siguientes request.

```
<%! int count = 0; %>
```

También se puede hacer lo siguiente:

```
1. <%!
      String color[] = {"red", "green", "blue"};
      String getColor(int i)
      {
            return color[i];
      }
   %>
```

```
2. <%! private int j = 0; %>
      <%! public int suma(int a, int b)
      {
            return a + b;
      }
      %>
```

Típicamente, las declaraciones son utilizadas para crear una variable a usar en JSP o para crear un método que se quiera utilizar en un JSP sin tener que estar declarándolo cada vez que se tenga que utilizar.

4.1.3 Scriptlets.

Son fragmentos de código Java, el scriptlet es ejecutado cada vez que la página es accesada.

```
<%
      out.print("<html><body>");
      count++;
      out.print("Welcome! You are visitor number " + count);
      out.print("</body></html>");
%>
```

4.1.4 Expresiones.

Sirve para imprimir el valor de una variable o expresión. La expresión es evaluada cada vez se accesa a la página. Prácticamente son una única línea de código y no incluyen operadores de control de flujo como `if`, `while`, `for`, etc.

```
<html><body>
<%@ page language="java" %>
<%! int count = 0; %>
  Bienvenido! Visitante numero:<%= ++count %>
</body></html>
```

Importante: Las expresiones no terminan con punto y coma (;) mientras que las declaraciones y los scriptlets sí llevan punto y coma. Además las expresiones son tomadas de manera interna por el motor de JSP como cadenas (como objetos de tipo `java.lang.String`).

4.1.5 Acciones.

Son comandos dados al motor de JSPs. Sirven para tareas específicas durante al ejecución de la página. Existen 6 acciones estándar:

- `jsp:include`
- `jsp:forward`
- `jsp:useBean`
- `jsp:setProperty`
- `jsp:getProperty`
- `jsp:pliguin`

Las dos primeras acciones `jsp:incluye` y `jsp:forward`, habilitan a los JSPs el reuso de componentes. Las siguientes `jsp:useBean`,`jsp:setProperty` y `jsp:getProperty` están relacionados con los javaBeans. Finalmete `jsp:plugin` indica a la motor de JSPs que genere el código apropiado para componentes embebidos como applets.

4.1.6 Comentarios.

Los comentarios no tienen ningún efecto sobre la salida de los JSPs, generalmente se utilizan para hacer documentación sobre la página.

```
<html><body>
Welcome!
<%-- JSP comment --%>
<% //Java comment %>
<!-- HTML comment -->
</body></html>
```

4.2 Atributos de la directiva page.

La directiva `page` informa al motor de JSPs sobre todas las propiedades de la página JSP. Existen 12 posibles atributos, pero en la práctica los que más se suelen usar son los siguientes cuatro.

`import`: Sirve para importar clases java y paquetes. Los valores que tiene por default son los siguientes:

- `java.lang.*;`
- `javax.servlet.*;`
- `javax.servlet.jsp*;`
- `javax.servlet.http.*;`

Y se usa de la siguiente manera:

```
<%@ page import="java.util.*," java.io.*, java.text.*,com.mycom.*, om.mycom.util.MyClass
" %>
```

O también se puede colocar en varias directivas:

```
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<%@ page import="java.text.*" %>
<%@ page import="com.mycom.*, com.mycom.util.MyClass" %>
```

`session`: Este atributo indica cuando el JSP es parte de la sesión HTTP y tiene por valor default `true`.

```
<%@ page session="false" %>
```

`errorPage`: se utiliza para especificar una URL de error, por defecto tiene valor `null`.

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

`isErrorPage`: Indica si el JSP es capaz de manejar errores. Su valor predeterminado es `false`.

```
<%@ page isErrorPage="true" %>
```

4.3 Usando Scriptlets.

Como todos los miembros definidos en un JSP y finalmente se convierten en un Servlet, no importa el orden.

```
<html>
<body>
Using pi = <%=pi%>, the area of a circle<br>
with a radius of 3 is <%=area(3)%>
<%!
double area(double r)
{
return r*r*pi;
}
%>
<%! final double pi=3.14159; %>
</body>
</html>
```

Las variables declaradas en un scriptlet se convierten en locales, por lo tanto el orden en que aparecen sí tiene importancia.

```
<html>
<body>
<% String s = s1+s2; %>            <- No se ha definido la variable s2
<%! String s1 = "hello"; %>        <- Variable s1
<% String s2 = "world"; %>         <- Variable s2
<% out.print(s); %>
</body>
</html>
```

Inicialización de variables.

En Java, las variables de instancia son automáticamente inicializadas con sus valores de default, mientras que las variables locales deben de ser inicializadas explícitamente cuando son usadas. Por lo tanto las variables declaradas en "declaraciones" son iniciadas automáticamente y las declaradas en scriptlets deben de ser inicializadas explícitamente.

```
<html>
<body>
<%! int i; %>
<% int j; %>
The value of i is <%= i++ %> <br>        <- El valor es 0.
The value of j is <%= j++ %> <br>        <- No ha sido inicializado.
</body>
</html>
```

Para que el código anterior compile:

```
<% int j=0; %>
```

Hay que recordar que las variables de instancia son creadas e inicializadas sólo una vez.

Como ya se mencionó, los scriptlets sirven para insertar todo tipo de código Java y se puede hacer una mezcla entre código Java y etiquetas HTML. La mayor ventaja de un scriptlet es poder abrir una llave ({) y poder cerrarla líneas de código del JSP más abajo (}), cómo por ejemplo:

```
1. <%
        if( i < 0 )
        {
          out.println("i es mayor que cero");
        }
   %>

2. <%
     if( nombre != null )
     {
      %>
      Bienvenido <%= nombre %>
      <%
     }
     %>
```

4.4 Objetos implícitos

Para los JSPs, Java cuenta con nueve objetos implícitos que están disponibles en cualquier lugar de un JSP y que por lo tanto, no necesitan crearse ni iniciarse:

Objetos Implícitos	Tipo	Alcance	Métodos más utilizados
request	Subclase de javax.servlet.ServletRequest	Petición (objeto Request)	getAttribute, getParameter, getParameterNames, getParameterValues
response	Subclase de javax.servlet.ServletResponse	Página	Comúnmente utilizada de manera interna y no por los autores de documentos JSP
pageContext	javax.servlet.jsp.PageContext	Página	findAttribute, getAttribute, getAttributesScope, getAttributeNamesInScope
session	javax.servlet.http.HttpSession	Sesión	getId, getValue, getValueNames, putValue
application	javax.servlet.ServletContext	Aplicación	getMimeType, getRealPath
out	javax.servlet.jsp.JspWriter	Página	clear, clearBuffer, flush, getBufferSize, getRemaining
config	javax.servlet.ServletConfig	Página	getInitParameter, getInitParameterNames
page	java.lang.Object	Página	Comúnmente utilizada de manera interna y no por los autores de documentos JSP
exception	java.lang.Throwable	Página	getMessage, getLocalizedMessage, printStackTrace, toString

4.5 Alcances de los JSPs.

El concepto de alcance significa la forma que los datos se comparten entre los Servlets usando los tres contenedores de objetos `ServletContext`, `HttpSession` y `ServletRequest`. Los alcances asociados con los tres alcances son respectivamente `aplication`, `session` y `request`. Para los JSPs existen los mismos alcances, pero se agrega otro: `page`, que esta dentro del objeto `PageContext`.

4.5.1 Aplication

En el alcance `aplication` los objetos son compartidos a través de todos los componentes de la aplicación web y son accesibles durante el tiempo de vida de la aplicación. Estos objetos son mantenidos en el `ServletContext`. Para compartir u obtener objetos de `ServletContext` se utilizan los métodos `setAttribute()` y `getAttribute()`.

4.5.2 Session

Los objetos en sesión son compartidos a través de todas las peticiones que haga un solo usuario y están accesibles mientras que la sesión esté activa. Para compartir objetos a este nivel se utiliza `session.setAttribute` y `session.getAttribute`.

4.5.3 Request

Los objetos en request son compartidos a través de los componentes que hayan realizado la misma petición y seguirán activos mientras siga la misma petición. Para obtener un objeto se utiliza: `request.setAttribute` y `request.getAttribute`.

4.5.4 Page

Los objetos en éste tipo de alcance son accesibles sólo y sólo en la traducción del JSP. Estos objetos están disponibles sólo a través de pageContext. También tiene los métodos `setAttribute()` y `getAttribute()`.

5. JavaBeans.

5.1 Utilización de JavaBeans en JSP's.

Los JavaBeans son componentes de software independiente que se usan para ensamblar otras aplicaciones. Simplemente es son datos encapsulados en forma de variables de instancia; éstas variables de instancia son referidas como propiedades del bean. La clase provee métodos para accesar y cambiar un valor.
Para utilizar JavaBeans en los JSP's es necesario conocer la etiqueta `<jsp:useBean>` la cual crea una instancia de un JavaBean. Se utilizan los JavaBeans porque:

1. En JSPs y servlets, los JavaBeans son utilizados principalmente porque sus propiedades pueden ser dinámicamente modificadas y accesadas en tiempo de ejecución.

2. Los JSPs por lo general utilizan JavaBeans que tienen un tiempo de vida de una única página o de una petición de usuario, por lo que no tiene mucho sentido almacenar el estado de un JavaBean. Si un JavaBean está siendo utilizado con el alcance de sesión, entonces será serializado automáticamente.

Para que una clase sea considerada un JavaBean en un JSP debe de seguir las siguientes características:

- La clase debe de tener un constructor público sin argumentos. Esto permita que la clase sea instanciada por el motor de JSPs.
- Por cada propiedad la clase debe contener dos métodos públicos que permiten obtener o cambiar el valor de la propiedad. El nombre de los métodos que accesan a un valor se llaman accesors y deben de ser `getXXX()` y el nombre de los métodos que cambian la propiedad se llaman mutators y deben de ser `setXXX()`, dónde XXX es el nombre de la propiedad. Por ejemplo:

```
public String getColor();
public void setColor(String);
```

El siguiente código es un ejemplo de un JavaBean llamado AddressBean.

```
public class AddressBean
{
//properties
private String street;
private String city;
private String state;
private String zip;
```

```
//setters or mutators
public void setStreet(String street){ this.street = street; }
public void setCity(String city) { this.city = city; }
public void setState(String state) { this.state = state; }
public void setZip(String zip) { this.zip = zip; }

//getters
public String getStreet(){ return this.street; }
public String getCity() { return this.city; }
public String getState() { return this.state; }
public String getZip() { return this.zip; }
}
```

El nombre de la clase trae pegada la palabra Bean, lo cual no es necesario pero se utiliza como una convención.

5.2 Usando JavaBeans y JSPs.

Hay tres acciones estándar que pueden usar los JavaBeans con los JSPs.

Acción	Descripción
`<jsp:useBean>`	Declara el uso del Javabean en una página JSP.
`<jsp:setProperty>`	Envío de valores a las propiedades de los JavaBeans.
`<jsp:getProperty>`	Obtención de los valores de las propiedades del JavaBean.

5.2.1 Declarando JavaBeans usando <jsp:useBean>

El `jsp:useBean` declara las variables en el JSP y asocia la instancia con en JavaBean. Este tipo de acción tiene cinco atributos:

Atributo	Descripción	Ejemplo
`id`	El nombre con el que el bean es identificado en el JSP.	Id="direccion"
`scope`	El alance del bean.	scope="session"
`class`	La clase Java del bean.	class="AddressBean"
`type`	Especifica el tipo de la variable que será usada para referirse al bean.	type="AdderssBean"
`beanName`	El nombre del bean	beanName="AddressBean"

De los atributos anteriores: id es forzoso, scope es opcional y class,type y beanName deben de ser usadas en las siguientes combinaciones:

- class
- type

- class y type
- beanName y type

La sintaxis de la etiqueta queda de la siguiente manera:

```
<jsp: useBean id=" beanInstanceName "
scope=" page |request| session| application"
{
 class=" package. class " [ type=" package. class " ]|
 type=" package. class "|
 beanName="{package. class | <%= expression %>}" type="package.class"
}
{
/>|> other elements </ jsp: useBean>
}
```

La sintaxis anterior nos indica que hay 3 maneras de instanciar un JavaBean y sigue la convención de Sun Microsystems de indicar una lista de opciones mutuamente excluyentes (se requiere solo una opción) que se encuentran entre llaves ({}). Una linea vertical (|) separa cada una de esas opciones excluyentes. Los atributos opcionales son listados entre corchetes ([]). Es importante notar que cuando el atributo beanName es utilizado, se evalúa una expresión para el tipo o nombre de archivo del Bean. Sin embargo, la forma más simple y común de instanciar un JavaBean es especificando un id para el Bean, dejar el alcance (scope) al de por defecto (page) y dar el nombre de la clase del JavaBean. Este es un ejemplo de un JavaBean que es un contador de accesos a una página:

```
<jsp:useBean id="miContador" class="HitCountBean" scope="session"/>
```

La anterior etiqueta es equivalente a:

```
HitCountBean miContador = (HitCountBean)
session.getAttribute("miContador ");
if (miContador == null)
{
      miContador = new HitCountBean ();
      session.setAttribute("miContador ", miContador);
}
```

5.2.2 Inicializando las propiedades del bean.

Dado que el Javabean es instanciado por el motor de JSPs usando un constructor sin argumentos, no se pueden inicializar las propiedades del bean; para resolver este problema se tiene la siguiente solución:

```
<jsp:useBean id="address" scope="session" class="AddressBean" >
```

```
<%
address.setStreet("123 Main St.");
%>
</jsp:useBean>
```

5.2.3 Uso de <jsp:setProperty>.

El uso de `jsp:setProperty` consiste en asignar nuevos valores a las propiedades del bean. Esta acción tiene cuatro propiedades:

Nombre	Descripción
`name`	El nombre del bean con que es identificado en el JSP.
`property`	El nombre de la propiedad del bean.
`value`	El nuevo valor asignado a la propiedad.
`param`	El nombre del parámetro disponible en `HttpServletRequest`, que será asignado como nuevo valor de la propiedad del bean.

Enseguida se muestra un ejemplo de cómo se utiliza esta acción.

```
<jsp:setProperty name="address" property="city" value="Albany" />
<jsp:setProperty name="address" property="state" value="NY" />
```

El anterior código es equivalente:

```
<%
address.setCity("Albany");
address.setState("NY");
%>
```

En el siguiente ejemplo se muestra la forma de utilizar al atributo `param` en vez de `value`.

```
<jsp:setProperty name="address" property="city" param="myCity" />
<jsp:setProperty name="address" property="state" param="myState"/>
```

En este caso el valor asignado a la propiedad del bean viene del request; esto también es equivalente a:

```
<jsp:setProperty name="address" property="city" value="<%= theCity %>" />
```

En un scriptlet también es equivalente a:

```
<%
address.setCity(request.getParameter("myCity"));
address.setState(request.getParameter("myState"));
%>
```

Las anteriores técnicas se utilizan cuando los nombres de los parámetros que se obtienen del request, no encajan con las propiedades de los beans. De lo contrario se puede utilizar:

```
<jsp:setProperty name="address" property="city" />
<jsp:setProperty name="address" property="state" />
```

Lo anterior es equivalente a:

```
<jsp:setProperty name="address" property="city" param="city" />
<jsp:setProperty name="address" property="state" param="state" />
```

Cuando todos los nombres de todos los parámetros recibidos del request son iguales a los de las propiedades del JavaBean, se puede hacer lo siguiente:

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

En vez de envíar propiedad por propiedad, se mandan todas al mismo tiempo.

5.2.3 Uso de <jsp:getProperty>.

El uso de esta acción es para traer los valores de las propiedades del bean. La sintaxis de ésta acción es muy sencilla:

```
<jsp:getProperty name="beanInstanceName" property="propertyName" />
```

Como se observa en la sentencia sólo se tienen dos atributos que no se pueden discriminar el uno del otro. El atributo `name` indica el nombre de la instancia del bean declarada en `<jsp:useBean>` y `property` especifica la propiedad cuyo valor será impreso.

Por ejemplo:

```
<jsp:getProperty name="address" property="state" />
<jsp:getProperty name="address" property="zip" />
```

Que es equivalente a:

```
<%
 out.print(address.getState());
 out.print(address.getZip());
%>
```

6 JDBC, JSP y Servlets.

6.1 Integración de los JSPs ySevlets con JDBC.

A continuación se muestra el código necesario para hacer la conexión entre un jsp y la clase java que se encarga de hacer la conexión a la base de datos.
El siguiente código hace la conexión con la base de datos (DBConector.java):

```
package database;
import java.sql.*;
import java.io.*;
import java.util.*;

public class DBConector
{
      private Connection conn;
      private Statement sqlStatement;
      String foo = "Not Connected";
      int bar = -1;

       public DBConector(String className,String ip,String base,String susuario,String
spass)
       {
             try
             {
                 Class.forName(className);
                 conn =
DriverManager.getConnection(ip+base+"?user="+susuario+"&password="+spass);
             }
             catch(Exception e)
             {
                 System.out.println("Error:"+e.getMessage());
             }
       }


       public void init() {
           try{
               if(conn != null){
                       foo = "Got Connection "+conn.toString();
               }

                     sqlStatement = conn.createStatement();
                     ResultSet rst = sqlStatement.executeQuery("select
categ_id,categ_definicion from au_categoria");
                     if(rst.next()) {
                        bar=rst.getInt(1);
                        foo=rst.getString(2);

                        }
               conn.close();
             System.out.println(" bar = "+bar+"\n foo = "+foo);
           }
           catch(Exception e) {
                     e.printStackTrace();
           }
       }
       public static void main(String...a){
```

```
                DBConector dbc = new
DBConector("org.postgresql.Driver","jdbc:postgresql://132.248.59.12:5432/","aula","post
gres","hola123");
                dbc.init();
        }
}
```

El siguiente código muestra el jsp que hace la llamada a la clase que se conecta a la base de datos (test.jsp).

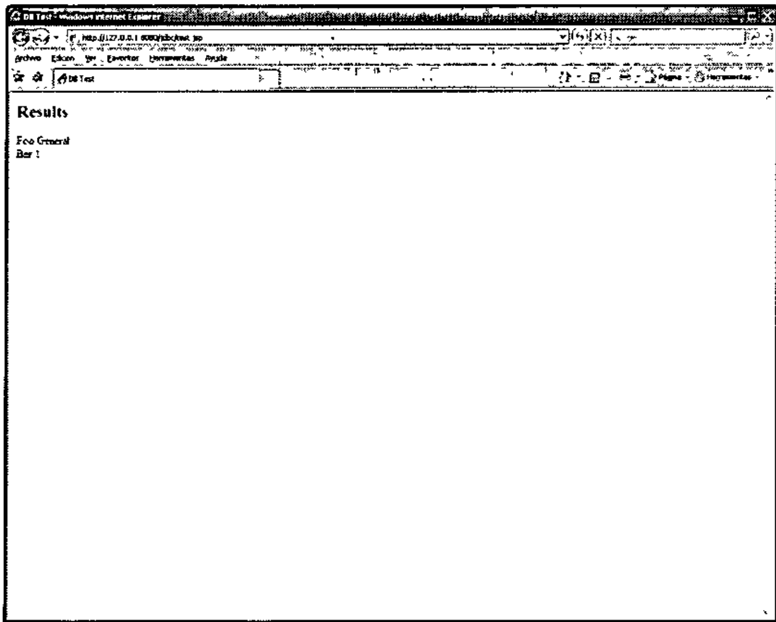
```
<html>
  <head>
    <title>DB Test</title>
  </head>
  <body>

  <%
    database.DBConector tst = new
      database.DBConector("org.postgresql.Driver","jdbc:postgresql://132.248.59.12:5432
    /","aula","postgres","hola123");
    tst.init();
  %>

  <h2>Results</h2>
    Foo <%= tst.getFoo() %><br/>
    Bar <%= tst.getBar() %>

  </body>
</html>
```

La salida del jsp debe de ser similar a a siguiente:



6.2 Transacciones.

Por defecto, una conexión funciona en modo *autocommit*, es decir, cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción, que sólo afecta a dicha sentencia. Es posible modificar esta opción mediante *setAutoCommit()*, mientras que *getAutoCommit()* indica si se está en modo *autocommit* o no. Si no se está trabajando en modo *autocommit* será necesario que se cierren explícitamente las transacciones mediante *commit()* si tienen éxito, o *rollback()*, si fallan, tras cerrar una transacción, la próxima vez que se ejecute una sentencia SQL se abrirá automáticamente una nueva, por lo que no existe ningún método del tipo que permita iniciar una transacción.

```
con.setAutoCommit(false);
```

Una vez que se han deshabilitado los commits, no se ejecutan las sentencias SQL hasta que se llame explícitamente al método commit(). Todas las sentencias que se hayan realizado antes de llamar a commit(), se van a realizar al momento de hacer la invocación del mismo.

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
pdateSales.setInt(1, 50);
pdateSales.setString(2, "Colombian");
pdateSales.executeUpdate();
reparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
pdateTotal.setInt(1, 50);
pdateTotal.setString(2, "Colombian");
pdateTotal.executeUpdate();
on.commit();
on.setAutoCommit(true);
```

Desde JDBC 3.0 se implementó el método Connection.setSavePoint, que guarda un punto específico en un transacción.

El siguiente ejemplo inserta una fila dentro de una tabla y se guarda un punto, después se inserta una segunda fila. Después de la ùtlima transacción, se deshace la inserción, pero la primera inserción se mantiene intacta, por lo tanto sólo se realiza la primera transacción.

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " +
                                              "(?FIRST?)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " +"VALUES (?SECOND?)");
...
conn.rollback(svpt1);
...
conn.commit();
```

7 Patrón Modelo-Vista-Controlador (MVC).

7.1 Introducción a patrones.

En el diseño orientado a objetos existen cierto tipo de problemas que se presentan con frecuencia. Para analizar y compartir el conocimiento sobre ellos, se ha desarrollado un tipo formal de documentación llamado “patrón”.

Un patrón es un conjunto de información que aporta la solución a un problema que se presenta en un contexto determinado. Para elaborarlo se aíslan sus aspectos esenciales y se añaden cuantos comentarios y ejemplos sean necesarios. En particular debemos identificar:

- El contexto en el que es posible aplicar el patrón.
- Las fuerzas (objetivos y restricciones) que intervienen en ese contexto.
- El diseño a aplicar para equilibrar objetivos y restricciones.

Las características de un buen patrón:

- Resuelve un problema cuya solución no es obvia.
- Ha sido revisado por una comunidad de desarrolladores.
- Ha sido experimentado en la práctica.
- Muestra como equilibrar restricciones y objetivos.

Los patrones son importantes por varios motivos:

- Encapsulan conocimiento detallado sobre un tipo de problema y sus soluciones.
- Proporcionan un vocabulario común.
- Estimula la reutilización del software (un patrón no es tal si no es reutilizable).
- Al aplicar soluciones probadas evitamos los riesgos y ahorramos el esfuerzo de reinventar nuestras soluciones.

7.1.1 Historia de los patrones.

El concepto de "patrón de diseño" que tenemos en Ingeniería del Software se ha tomado prestado de la arquitectura. En 1977 se publica el libro "A Pattern Language: Towns/Building/Construction", de Christopher Alexander.

Alexander comenta que “Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo siquiera dos veces de la misma forma”. El patrón es un esquema de solución que se aplica a un tipo de problema, esta aplicación del patrón no es mecánica, sino que requiere de adaptación y matices. Por ello, dice Alexander que los numerosos usos de un patrón no se repiten dos veces de la misma forma.
La idea de patrones de diseño estaba "en el aire", la prueba es que numerosos diseñadores se dirigieron a aplicar las ideas de Alexander a su contexto. El catálogo más famoso de patrones se

encuentra en "Design Patterns: Elements of Reusable Object-Oriented Software", de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, 1995, Addison-Wesley, también conocido como el libro GOF (Gang-Of-Four).

7.1.2 Clasificación.

Existen varios tipos de patrones, dependiendo del nivel de abstracción, del contexto particular en el cual aplican o de la etapa en el proceso de desarrollo. Algunos de estos tipos, de los cuales existen publicaciones al respecto son:

Patrones de creación.

Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo tratar las clases a crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas.

Según donde se tome dicha decisión podemos clasificar a los patrones de creación en patrones de creación de clase (la decisión se toma en los constructores de las clases y usan la herencia para determinar la creación de las instancias) y patrones de creación de objeto (se modifica la clase desde el objeto).

Algunos patrones de creación son los siguientes: Abstract Factory (Fábrica abstracta), Builder (Constructor), Factory Method (Método de fabricación), Prototype (Prototipo) y Singleton (Único).

Patrones estructurales.

Tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos y éstas están determinadas por las interfaces que soportan los objetos. Estudian como se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos.

Algunos patrones estructurales son: Adapter (Adaptador), Bridge (Puente), Composite (Compuesto), Facade (Fachada) y Proxy (Representante).

Patrones de comportamiento

Los patrones de comportamiento estudian las relaciones con el flujo de control de un sistema. Ciertas formas de organizar el control en un sistema pueden derivar en grandes beneficios para la eficiencia y el mantenimiento del sistema.

Algunos patrones de comportamiento son: Chain of responsability (Cadena de responsabilidades), Command (Comando), Interpreter (Intérprete), Observer (Observador), State (Estado), Strategy (Estrategia) y Visitor (Visitante)

Patrones de sistema

Los patrones de sistema constituyen el más diverso de los cuatro tipos de patrones. Llevan su aplicación al nivel más abstracto de su arquitectura. Los patrones de sistema pueden aplicarse a los procesos principales de una aplicación, o incluso entre aplicaciones.

Algunos patrones de sistema son: Model-View-Controler (Modelo-Vista-Controlador), Session (Sesión), Callback (Retrollamada), Seccessive Update (Actualización sucesiva) y Transaction (Transacción).

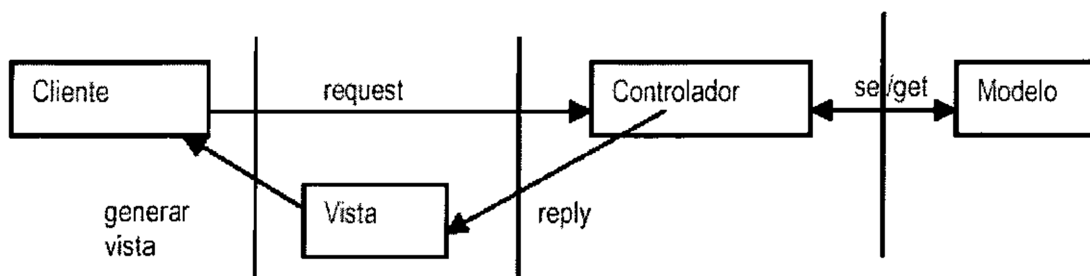
7.2 Definición del Patrón MVC.

El MVC se encuentra integrado dentro de los patrones de sistema y su principal objetivo es aislar tanto los datos de la aplicación, como el estado (modelo) de la misma y el mecanismo utilizado para representar (vista) dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo (controlador). Las aplicaciones MVC se dividen en tres grandes áreas funcionales:

- Vista: La presentación de los datos.
- Controlador: El que atenderá las peticiones y componentes para toma de decisiones de la aplicación. Controla la transición entre el procesamiento de los datos y su visualización.
- Modelo: Es la lógica del negocio o servicio y los datos asociados con la aplicación, representa sus datos y comportamientos.

El propósito del MVC es aislar los cambios. Es una arquitectura preparada para los cambios, que desacopla datos y lógica de negocio de la lógica de presentación, permitiendo la actualización y desarrollo independiente de cada uno de los componentes citados.

A continuación se muestra un esquema de este modelo:



Esquema del patrón MVC.

8. Desarrollo de aplicaciones web seguras.

8.1. Conceptos básicos.

La importancia de Internet hace que las compañías presten mucha atención en la seguridad de las aplicaciones que utilizan a través de Internet. La especificación de los Servlets provee métodos para la implementación en las aplicaciones web.

8.1.1 Autentificación (Authentication).

El fundamental requerimiento en seguridad es la autentificación, el cual es un proceso para identificar a una persona, sistema o aplicación para realizar un procedimiento, lo que significa que se hace una validación. Un ejemplo es el caso de mostrar el pasaporte cada vez que se viaja a otro país, éste identifica al viajero en cualquier parte del mundo. En Internet la típica forma de autentificarse es a través del usuario (login) y una contraseña (password).

8.1.2 Autorización (Authorization).

Una vez que el usuario ha sido autentificado, debe de ser autorizado. La autorización es el proceso de determinar cuando un usuario tiene permitido accesar aun recurso por el cual ha realizado una solicitud. Un ejemplo es que uno no puede accesar a una cuenta cualquiera de banco, a pesar de que se tenga una cuenta en ese banco. La autorización sirve para llevar una lista de control de acceso (ACL), ésta especifica el usuario y el tipo de acceso a los recursos.

8.1.3. Integridad de los datos (Data integrity).

Éste es el proceso de asegurarse que los datos no son manipulados mientras hay peticiones y respuestas. Por ejemplo, si se envía una petición para transferir $1000 de una cuenta a otra, el banco deberá obtener la petición por $1000 y no por $1000.

8.1.4. Confidencialidad (Confidentiality).

Es el proceso de asegurarse de que ninguno excepto el usuario es capaz de accesar a su información. Esto se logra a través de la encriptación de la información que sólo el usuario al que le pertenece es capaz de desencriptar; si la información es capturada, se asegura que al menos sea inservible.

8.1.5. Auditoria (Auditing).

Es el proceso grabar los eventos o procesos que se realicen en el sistema. La auditoria ayuda a determinar violaciones y es realizado dando mantenimientos a as bitácoras de errores que genera la aplicación.

8.1.6. Código malicioso.

Una parte de un código que cause daño a un sistema es llamado código malicioso, esto incluye virus, gusanos y trojanos. Además de llegar de afuera, algunas veces los desarrolladores dejan abierta una puerta dentro del software, lo cual puede dar la oportunidad de un uso indebido.

8.2. Mecanismos de autentificación.

La especificación de los Servlets define los siguientes cuatro mecanismos de autentificación:

- HTTP Basic autentificación.
- HTTP Digest autentificación.
- HTTPS Client autentificación.
- HTTP FORM-Based autentificación.

Todos estos mecanismos están basados en el mecanismo de usuario y contraseña, en donde el servidor mantiene una lista de usuarios y contraseñas o bien una lista de recursos que tiene que proteger.

8.2.1 HTTP Basic Autentificación.

Este tipo de autentificación definido en la versión 1.1 de HTTP, es la más simple y común de todos los mecanismos para proteger recursos. Cuando un navegador hace una petición, el servidor pregunta por el usuario y contraseña. Si el usuario ingresa un usuario y contraseña válidos, el servidor le envía los recursos. La secuencia de eventos es la siguiente:

1. El navegador envía una petición de un recurso protegido. En este momento el navegador no sabe si el recurso está protegido o no, sólo hace la petición:

```
GET /servlet/SalesServlet HTTP/1.1
```

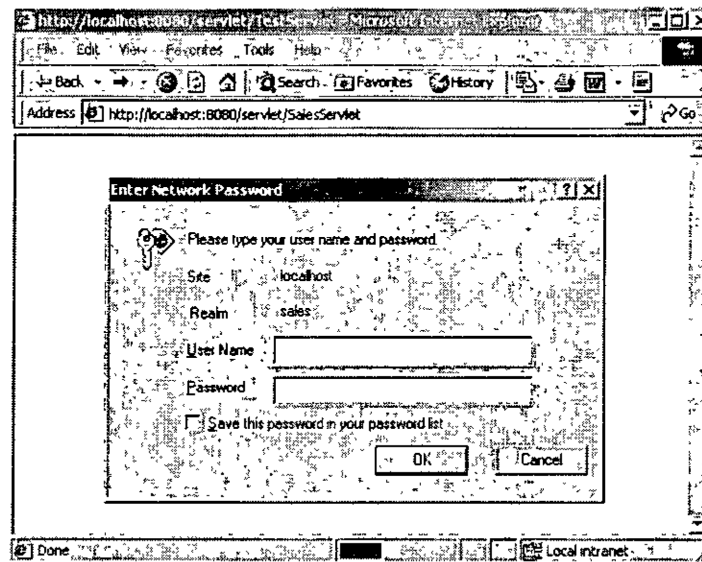
2. El servidor observa que el recurso esta protegido y en vez de enviar el recurso, manda un mensaje "401 Unauthorized". En el mensaje, se incluye una cabecera que le dice al navegador que es necesario hacer "Basic Authentication" para entrar al recurso.

```
HTTP/1.1 401 Unauthorized
Server: Tomcat/5.0.25
WWW-Authenticate: Basic realm="sales"
Content-Length=500
Content-Type=text/html

<html>
…detailed message
</html>
```

3. Al momento de recibir el mensaje, el navegador manda un cuadro de dialogo pidiendo el usuario y contraseña.
4. Una vez que se ingresó el usuario y contraseña, el navegador envía la petición y pasa el valor en el encabezado llamado Authorization:

```
GET /servlet/SalesServlet HTTP/1.1
Authorization: Basic am9objpqamo=
```



5. Cuando el servidor recibe la petición, hace la validación del usuario y contraseña. Si son válidos, envía el recurso, de otro modo envía el mensaje "401 Unauthorized" de nuevo.

Ventajas.

- Es fácil de implementar.
- Los navegadores la soportan.

Desventajas

- No es seguro porque el usuario y contraseña no esta encriptada.
- No se puede modificar la apariencia del cuadro de diálogo.

8.2.2 HTTP Digest authentication.

La diferencia con Basic authentication es que la contraeña es enviada de forma encriptada, lo cual la hace más segura.

Ventajas:

- Es más segura que Basic authentication.

Desventajas:

- Es sólo soportada por Internet Explorer.
- No es soportada por los contenedores de Servlets si la especificación no lo indica.

8.2.3 HTTPS Client authentication.

HTTPS es HTTP utilizando SSL (Secure Socket LAyer). SSL es un protocolo desarrollado por Netscape para asegurar que los datos que se mandan sean enviados en Internet sean de forma privada. La autentificación se realiza cuando la conexión SSL es establecida entre el servidor y el navegador. Toda la información es transmitida en forma encriptada usando una llave pública, que es manejada por el navegador y el contenedor de servlets.

Ventajas:

- Es la forma más segura de los cuatro tipos.
- Todos los navegadores tienen soporte para implementarla.

Desventajas:

- Se requiere un certificado de autorización.
- Es costosa la implementación y el mantenimiento.

8.2.4 FORM-based authentication.

Este mecanismo es similar a Basic authentication, sin embargo en vez de usar un pop-up con ventana, se utiliza una forma de HTML para caprutar el usuario y contraseña. Los desarrolladores deben crear la página HTML que contiene la forma, que les permite personalizar su aspecto. El único requisito de la forma es que se su acción debe ser atributo j_security_check y debe tener dos campos: j_username y j_password. Todo lo demás es personalizable.

Ventajas:

- Es muy sencillo de realizar.
- Todos los navegadores lo soportan.
- Se puede personalizar la apariencia de la forma.

Desventaja:

- No es seguro, ya que el usuario y contraseña no están encriptadas.

8.2.5 Definición de mecanismos de autenticación para aplicaciones web.

Con el fin de garantizar la portabilidad y facilidad de configuración, la autentificación se define en el descriptor de despliegue (web.xml) de la solicitud. Sin embargo, antes de especificar que los usuarios deben ser autentificados, debemos de configurar sus nombres de usuario y contraseñas. Este paso depende del contenedor servlet proveedor.

8.2.6 Especificando el mecanismo de autentificación.

Este mecanismo se especifica en el descriptor de la aplicación web usando el elemento <login-config>. La especificación define <login-config> con los siguientes elementos:

```
<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>
```

Veamos los subelementos:

- <auth-method>. Especifica cuál de los cuatro métodos de autenticación deben ser utilizados para validar el usuario: BASIC, DIGEST, CLIENT, o FORM.
- <realm-name>. Especifica el nombre de dominio que se utilizará en la autorización básica HTTP solamente.
- <form-login-config>. Especifica la URL de la página de inicio de sesión y la URL de la página de error. Este elemento sólo se utiliza con el método FORM.

Ejemplo:

```
<web-app>
...
<login-config>
<auth-method>BASIC</auth-method>
<realm-name>sales</realm-name>
</login-config>
...
<web-app>
```

El código anterior utiliza el mecanismo BASIC para autentificar a los usuarios. Si se quiere utilizar el mecanismo FORM, debemos escribir dos páginas HTML: una para capturar el nombre de usuario y contraseña, y otra para mostrar un mensaje de error si la conexión falla. Por último, necesitamos especificar estos archivos HTML en el elemento <form-login-config>, como se muestra aquí:

```
<web-app>
...
<login-config>
<auth-method>FORM</auth-method>
<!--realm-name not required for FORM based authentication -->
<form-login-config>
<form-login-page>/formlogin.html</form-login-page>
<form-error-page>/formerror.html</form-error-page>
</form-login-config>
</login-config>
...
<web-app>
```

La página formlogin.html queda de la siguiente forma:

```
<html>
<body>
<h4>Please login:</h4>
<form method="POST" action="j_security_check">
```

```
<input type="text" name="j_username">
<input type="password" name="j_password">
<input type="submit" value="OK">
</form>
</body>
</html>
```

La pàgina formerror.html queda de la siguiente forma:

```
<html>
<body>
<h4>Sorry, your username and password do not match.</h4>
</body>
</html>
```

Observe que el método FOMR, no tenemos que escribir ningún Servlet para procesar la forma. La acción la desencadena j_security_check en el contenedor Servlet para hacer la transformación.

8.3 Haciendo una página Web segura.

Es muy común que una aplicación Web ser desarrollado por un grupo de personas y, a continuación, desplegados por un muy diferente grupo de personas en otro lugar. Por ejemplo, muchas empresas vender aplicaciones Web como soluciones para las necesidades empresariales. Esto significa que el desarrollador debe ser capaz de transmitir fácilmente los requisitos de seguridad de la solicitud del cliente. El cliente también debe ser capaz de personalizar determinados aspectos de la aplicación de la seguridad sin modificar el código. El Servlet framework permite precisar las necesidades de seguridad de la aplicación en el el descriptor de despliegue. Esto se llama seguridad declarativa.

Por defecto, todos los recursos de una aplicación Web son accesibles a todo el mundo. Para restringir el acceso a los recursos, tenemos que identificar tres cosas:

- Colección de recursos Web: Identifica los recursos de la aplicación (es decir, Archivos HTML, servlets, etc) que debe ser protegida del acceso público. Un usuario debe tener la debida autorización para acceder a los recursos asignados.
- Autorización: Identifica las funciones a las que un usuario puede ser asignado. En vez de de especificar los permisos para los usuarios individuales, los permisos están asignados a funciones. Por ejemplo, un AdminServlet puede ser accesible a cualquier usuario que se encuentra en la función de administrador. En el tiempo de instalación, cualquiera de los usuarios reales se puede configurar como administrador.
- Datos del usuario: Especifica la forma en que los datos deben ser transmitidos entre el remitente y el receptor. Formula las políticas para el mantenimiento de la integridad de los datos y la confidencialidad. Por ejemplo, una aplicación puede requieren el uso de HTTPS como un medio de comunicación en lugar de HTTP.

Podemos configurar los tres aspectos en el descriptor de despliegue utilizando el elemento <security-constraint>. Este elemento, que cae directamente bajo el elemento de <web-app> web.xml, se define como sigue:

```
<!ELEMENTO limitación de seguridad (nombre de pantalla?, La web de recopilación de recursos +,auto-limitación?, el usuario de datos limitación?)>
```

8.3.1. display-name

Este es un elemento opcional. Especifica un nombre para la restricción de seguridad que sea fácilmente identificable.

8.3.2. web-resourse-collection

Especifica una colección de recursos para la que la seguridad de esta se aplica. Podemos definir una o más colecciones en <security-constraint> de la siguiente manera:

```
<!ELEMENT web-resource-collection (web-resource-name, description?,url-pattern*, http-method*)>
```

- web-resourse-name: Especifica el nombre del recurso.
- description: Indica la descripción del recurso.
- url-pattern: Especifica el url a través de la cual se especificará el acceso al recurso.
- http-method: Provee el control sobre las peticiones HTTP. Especifica el método HTTP que se va a aplicar.

```
<web-app>
...
<security-constraint>
<web-resource-collection>
<web-resource-name>reports</web-resource-name>
<url-pattern>/servlet/SalesReportServlet/*</url-pattern>
<url-pattern>/servlet/FinanceReportServlet/*</url-pattern>
<url-pattern>/servlet/HRReportServlet/*</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
...
</security-constraint>
...
</web-app>
```

8.3.3. auth-constrainth

Específica los roles que pueden acceder a un recurso.

```
<!ELEMENT auth-constraint (description?, role-name*)>
```

- description: descripción del rol de usuario
- role: Especifica los roles de los usuarios que pueden acceder a un recurso.

```
<web-app>
...
<security-role>
<role-name>supervisor</role-name>
</security-role>
<security-role>
<role-name>director</role-name>
</security-role>
<security-role>
<role-name>employee</role-name>
</security-role>
...
<security-constraint>

<auth-constraint>
<description>accessible to all supervisors and
directors</description>
<role-name>supervisor</role-name>
<role-name>director/role-name>
</auth-constraint>
...
</security-constraint>

</web-app>
```

8.3.4. user-data-constraint

Este elemento especifica cómo va a ser la comunicación entre el cliente y el servidor.

```
<!ELEMENT user-data-constraint (description?, transport-guarantee)>
```

- description: Se indica una descripción de cómo se esta usando este elemento.
- transport-guarantee: Contiene uno de los tres valores NONE, INTEGRAL y CONFIDENTIAL. NONE implica no necesita ninguna garantía de la integridad de los datos transmitidos. INTEGRAL Y CONFIDENTIAL implica que la aplicación tenga alguno de estos dos tipos de integridad. Usualmente cuando se una HTTP el valor es NONE y cuando se usa HTTPS el valor cambia a INTEGRAL y CONFIDENTIAL.

```
<web-app>
...
<security-constraint>
...
<user-data-constraint>
<description>requires the data transmission
to be integral</description>
<transport-guarantee>INTEGRAL</transport-guarantee>
</user-data-constraint>
...
</security-constraint>
...
</web-app>
```

8.3.5. Identificando usuario.

La especificación de los Servlets permite utilizar métodos para identificar usuarios, con lo cual se puede generar algún tipo de respuesta dependiendo el usuario.

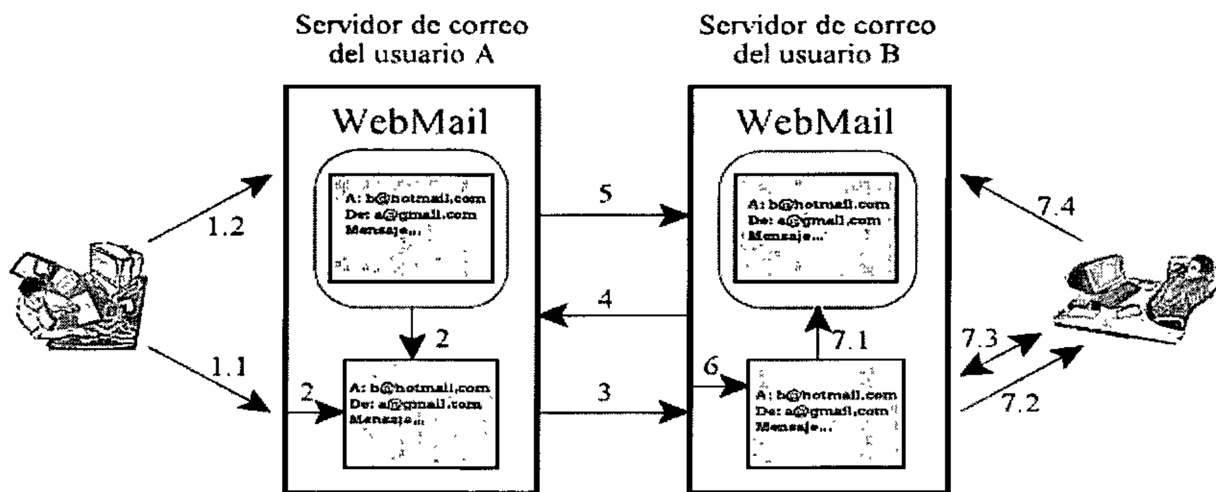
Método	Descripción.
String getRemoteUser()	Este método devuelve el nombre de inicio de sesión del usuario, si el usuario ha sido autentificado, o nulo si el usuario no se ha autentificado.
Principal getUserPrincipal()	Este método devuelve una java.security.Principal objeto que contiene el nombre del usuario autentificado. Devuelve null si el usuario no está autentificado.
boolean isUserInRole()	Este método devuelve un booleano que indica si el usuario autentificado está incluido en el rol especificado. Devuelve falso si el usuario no está autentificado.

10. API Java Mail.

10.1 Introducción.

El correo electrónico es un servicio de red que permite intercambiar mensajes entre distintos usuarios de manera asíncrona; estos mensajes pueden contener o no archivos adjuntos. Según la el diccionario de la RAE el correo electrónico se define como: «Sistema de comunicación personal por ordenador a través de redes informáticas».

La forma en funciona el correo electrónico es la siguiente:



1.- El usuario A se prepara para enviar un correo electrónico al usuario B. Este procedimiento se puede realizar de dos formas:

> 1.1.- El usuario A usa un programa de tipo MUA (*Mail User Agent* o Agente de Usuario de Correo), que son programas que se usan para crear y enviar (entre otras funciones) correos electrónicos, y entre los que se pueden destacar Microsoft Office Outlook , Mozilla Thunderbird, Eudora, Pegasus, etc. El usuario compone el mensaje y selecciona la opción de enviar. El MUA del usuario **A** le indica al servidor de correo de este usuario quién está enviando el mensaje y quiénes son los destinatarios, y a continuación envía el mensaje usando el protocolo SMTP (*Simple Mail Transfer Protocol* - Protocolo Simple de Transferencia de Correo Electrónico). Cuando el servidor recoge el mensaje y acepta transferirlo hasta su destino, el MUA informa al usuario **A** que el envió ha tenido éxito.
>
> 1.2.- El usuario **A** usa directamente su servidor de correo electrónico vía web. Esta forma de acceder al correo electrónico se conoce como *WebMail* o Correo Web. Se trata de un servicio que permite el acceso al correo mediante páginas web, usando para ello un navegador. Para ello, el usuario ha de autenticarse en el servidor introduciendo su nombre de usuario y contraseña; a continuación compone el mensaje y selecciona la opción de enviar. Entre las ventajas de este método se puede decir que los mensajes no se descargan al ordenador, es posible acceder desde cualquier máquina sin que tenga que tener ningún *software* específico (todos los sistemas operativos suelen incluir un navegador web) y la creación de una cuenta de correo es más sencilla que desde un programa MUA. La principal desventaja es que el espacio de almacenamiento de mensajes se limita a lo que ofrece el servidor de correo, en lugar de la capacidad de almacenamiento del propio disco duro que ofrece un programa MUA. Además requiere una conexión permanente a Internet mientras lo usemos y los mensajes ocupan bastante más espacio pues van embebidos en HTML, lo que implica que tardan más en enviarse.

2.- El mensaje es almacenado en el servidor de correo del usuario **A**.

3.- El servidor de correo del usuario **A** solicita al del usuario **B** el nombre del servidor al que tiene que enviar el mensaje. El servidor del usuario **B** se conoce gracias a la parte de la dirección de correo que sigue a la arroba, ej. @gmail.com.

4.- El servidor de correo del usuario **B** le responde al del usuario **A**, enviándole un registro MX (*MX record*), que es un registro de correo. En este registro constan las direcciones de correo que pertenecen a un dominio concreto y la máquina que se corresponde con cada dirección. Así cuando se envía un correo, se comprueba en la lista de registros MX de ese dominio si contiene esa dirección de correo, no aceptándola en caso contrario. En el registro MX pueden aparecer distintas máquinas, estableciéndose prioridades entre ellas o balanceando la carga de correo por las distintas máquinas. Por ejemplo un servidor le devolverá al otro una lista con una serie de máquinas en orden de prioridad. Si al enviar el correo a la primera máquina esta fallase o estuviese saturada, se enviaría a la segunda, y así sucesivamente.

5.- El servidor de correo del usuario **A** envía entonces el mensaje al servidor correspondiente a esa dirección de correo, usando el protocolo SMTP.

6.- El mensaje es almacenado en el servidor de correo del usuario **B** y se coloca en el buzón de correo del usuario.

7.- El último paso sería la lectura del mensaje por parte del usuario **B**, lo que se puede llevar a cabo de 3 formas:

7.1.- El usuario **B** accede directamente a su servidor de correo vía *webmail* y lee sus mensajes del servidor sin descargarlos.
7.2.- El usuario **B** accede a su correo a través de un programa de tipo MUA. Si lo hace de esta forma tiene dos opciones:
7.2.1.- Descargar los mensajes a su máquina usando el protocolo POP3 (*Post Office Protocol* - Protocolo de Oficina Postal versión 3).
7.2.2.- Descargar los mensajes a su máquina o leerlos directamente en el servidor usando el protocolo IMAP (*Internet Message AccessProtocol* - Protocolo de Internet para el Acceso a Mensajes Electrónicos).

10.1 Estructura de un mensaje de correo.

A continuación se detallan los principales campos del sobre o cabecera de un mensaje; el nombre de cada uno de estos campos está definido en el estándar en su forma inglesa:

• *From* (De): se trata de la dirección de correo del usuario que envía el mensaje. Este campo puede ser alterado por el emisor para que aparezca otra dirección. En este caso puede aparecer en la cabecera un mensaje de X-Autentication-Warning indicando que la dirección que aparece en el *From* puede no corresponderse con la que realmente ha enviado el correo.

• *To* (Para): ss la dirección y nombre (opcional) del usuario al que va dirigido el correo. En el campo *To* pueden aparecer varias direcciones.

• *Cc* (*Carbon Copy* - Copia en Papel Carbón): dirección y nombre (opcional) del usuario al que queremos que se envíe una copia del mensaje. En este campo pueden aparecer varias direcciones o ninguna. El término copia en papel carbón procede de las antiguas máquinas de escribir en las que, para sacar varias copias de un documento, se colocaba bajo el original un papel de carbón y otro papel en blanco (la copia), de manera que al ir escribiendo, los golpes de los martillos de la máquina imprimían en el original y, por presión, dejaban una marca de carboncillo con la forma de la letra en la copia que había por debajo.

• *Bcc* (*Blind Carbon Copy* - Copia Oculta en Papel Carbón): es equivalente al campo *Cc* con la diferencia de que la dirección que escribamos en este campo no será vista por el resto de usuarios que reciban el mensaje. Puede aparecer también como *Cco* (Copia de carbón oculta). En este campo pueden aparecer varias direcciones o ninguna.

• *Subject* (Asunto): breve descripción o título que queremos que figure en el mensaje, y que informa al destinatario de la naturaleza de éste. Este campo se puede dejar en blanco, pero no es recomendable.

Los campos que se explican a continuación no suelen ser mostrados por defecto por los MUA o al consultar el correo desde el servidor. Para verlos suele ser necesario seleccionar opciones como encabezado completo, ver todas las cabeceras,etc.:

• *Return-Path*: este campo es añadido por el último servidor por el que pasa el mensaje antes de ser entregado a su destino, y se usa para hallar la ruta de retorno en caso de devolución del mensaje.

• *Received*: en este campo están registrados todos los servidores por los que pasa el mensaje, por si se necesita realizar un seguimiento del mismo.

• *Message-ID*: este campo contiene un identificador que hace que el mensaje asociado a él sea único en todo Internet.

• *X-Priority*: indica la prioridad con la que el lector debe considerar al mensaje.

• *X-Mailer*: Nombre del *software* o aplicación utilizada para transferir el correo.

• *MIME-Version*: indica la versión de MIME que utiliza el mensaje. MIME son las siglas de *Multipurpose Internet Mail Extension* (Extensiones Multipropósito de Correo por Internet). Básicamente, permiten enviar información adicional junto con el texto de un correo, de manera que se pueda identificar correctamente cada uno de estos bloque de información y la naturaleza de su contenido.

• *Content-Type*: indica la naturaleza de los bytes que constituyen el contenido del mensaje para que el sistema receptor del mismo pueda tratar los datos de forma apropiada: imagen, audio, video, texto, etc.

• *Content-Transfer-Encoding*: indica el tipo de codificación que se ha usado con el mensaje, para que el sistema receptor sepa como decodificarlo.

Existen muchos más campos (no obligatorios en la mayoría de los casos) que forman la cabecera de un mensaje. Si se desean conocer todas las posibilidades puede consultar el RFC 2822 y los RFC que tratan sobre MIME: RFC 2045 a 2049.

10.2. Direcciones de correo electrónico.

Las direcciones de correo electrónico se corresponden de forma unívoca con un usuario, es decir, identifican de forma única a cada usuario dentro de una red. Así cuando se envía un correo electrónico se asegura que este llegará sólo al usuario deseado.

Las direcciones tienen la forma **usuario@dominio** donde el **usuario** indica la cuenta o buzón dentro del servidor del usuario al que va destinado el mensaje. A continuación aparece el **dominio**, es decir, el nombre de la máquina donde se encuentra dicha cuenta. Este nombre es único en todo Internet, por lo que el propietario del servidor de correo tendrá que haber registrado su dominio previamente.

10.3. Protocolos y extensiones relacionadas.

A continuación se comentan los protocolos más importantes que intervienen en el funcionamiento del correo electrónico, y que son los que JavaMail incluye por defecto. Además se comentan las extensiones MIME, que hoy día aparecen prácticamente en todas las aplicaciones de correo electrónico.

10.3.1. POP (Post Office Protocol)

El protocolo POP (Protocolo de Oficina Postal), actualmente en su versión 3 referido como POP3, se usa para que un usuario pueda descargar su correo desde el servidor a su máquina local. El estándar de este protocolo se encuentra en el documento RFC 1939 (http://www.rfc-es org/rfc/rfc1939-es.txt).

Se podría decir de forma somera, que POP3 se basa en 3 fases: la fase de autorización, la fase de transacción, y la fase de actualización, junto con una serie de comandos.

• El primer paso cuando un cliente se conecta a un servidor POP es establecer una conexión en el puerto 110 de éste. Una vez establecida la conexión, el cliente POP recibe una invitación y se produce una serie de intercambios de comandos POP entre el cliente y el servidor. Durante este intercambio el cliente debe identificarse, con nombre de usuario y contraseña, ante el servidor POP en lo que se conoce como fase de autorización o autentificación.

• La siguiente fase es la fase de transacción en la que se pueden usar los siguientes comandos:

LIST: para mostrar los mensajes.
RETR: para recuperar los mensajes.
DELE: para borrar mensajes.
STAT: que devuelve información sobre un buzón.
NOOP: el servidor no hace nada, sólo devuelve una respuesta positiva.
RSET: elimina las marcas de los mensajes marcados como borrados en el servidor.
QUIT: en esta fase hace que se pase a la fase de actualización.

• La última fase es la fase de actualización, en la que se eliminan los mensajes marcados y se eliminan los recursos asociados a la conexión.

Por último señalar que los servidores basados en POP3 sólo ofrecen al usuario acceso a un único buzón, mostrado normalmente a modo de carpeta por el servidor. En otras palabras, el servidor no mantiene ninguna estructura de carpetas, sino que ésta debe ser creada y mantenida por un programa MUA.

10.3.2. SMTP (Simple Mail Transfer Protocol)

El protocolo SMTP es un protocolo que se usa para el envío de correo electrónico (Protocolo Simple de Transferencia de Correo Electrónico). SMTP transfiere los mensajes desde la máquina cliente al servidor, quedándose este último con el mensaje, si se trata del destino final, o transfiriéndolo a otro servidor, también vía SMTP, para que el mensaje llegue al servidor del destinatario. El protocolo completo está descrito en los RFC 821 y 822, donde se define el funcionamiento de SMTP y el formato de mensaje respectivamente. Estos RFC han sido actualizados por los RFC 2821 y 2822 (http://www.faqs.org/rfcs/rfc2821.html y http://www.faqs.org/rfcs/rfc2822.html).

Algunas de las características de SMTP son las siguientes:

• Utiliza el puerto 25 para establecer la conexión.
• Se pueden comunicar entre sí servidores de plataformas distintas.
• Dos sistemas que intercambien correo mediante SMTP no necesitan una conexión activa, ya que posee sistema de almacenamiento y reenvío de mensajes, es decir, si el receptor no está disponible el mensaje será reenviado posteriormente.

Algunos de los principales problemas de SMTP en su forma original son:

• Falta de soporte para idiomas distintos del inglés pues solo acepta caracteres ASCII de 7 bits, lo que impedía usar caracteres propios de otros idiomas como la ñ o los acentos.

• Los datos no viajan de forma segura.
• Facilita la expansión del spam, pues no requiere autenticación en el servidor.

10.3.3. IMAP (Internet Message Access Protocol)

El protocolo IMAP (Protocolo de Internet para el Acceso a Mensajes Electrónicos), actualmente en su versión 4 primera revisión, es un protocolo que al igual que POP se utiliza para que un usuario pueda acceder a sus mensajes de correo electrónico contenidos en un servidor. IMAP está definido en el RFC 2060, aunque desde 2003 ha sido actualizado por el RFC 3501 (http://www.faqs.org/rfcs/rfc3501.html).

Algunas de las características principales de IMAP son las siguientes:

• Una de las ventajas más destacadas es la capacidad de un usuario de poder manejar varios buzones, llamados carpetas remotas de mensajes en el RFC, pudiendo crear, borrar y renombrar estos buzones. En POP3, como se comentó anteriormente, el usuario sólo tiene acceso a un único buzón.
• Se permite que varios usuarios accedan simultáneamente a un mismo buzón, a diferencia de POP3 donde cada buzón pertenece a un único usuario.
• Se pueden activar y desactivar banderines (*flags*) para controlar el estado de un mensaje, es decir, saber si ha sido borrado, leído, no leído, etc. POP3 no puede manejar banderines.
• Se pueden leer los mensajes desde el servidor sin necesidad de descargarlos, como ocurre con POP3.
• Se pueden realizar búsquedas en el servidor, con el objetivo de localizar los mensajes que cumplan ciertas características.

10.3.4. MIME (Multipurpose Internet Mail Extension)

Las extensiones MIME son una ampliación del RFC 822, definidas en los RFC 2045 a 2049 (http://www.rfc-es.org/rfc/rfc2045-es.txt, http://www.rfc-es.org/rfc/rfc2046-2046-es.tx, http://www.rfc-es.org/rfc/rfc2047-es.tx), pensadas para superar algunos de los problemas impuestos por las restricciones de SMTP y del formato de mensaje del RFC 822.

Algunos de estos problemas son los siguientes:

• El contenido del mensaje sólo puede ser texto por lo que no se pueden transmitir archivos ejecutables, binarios o de cualquier otro tipo.
• El conjunto de caracteres aceptado está limitado al US-ASCII, de 7 bits, por lo que no se admiten caracteres de otros idiomas, que necesitan de 8 bits para su representación.
• El tamaño total de los mensajes no puede superar un determinado tamaño.
• Problemas en la conversión entre distintos formatos.

MIME describe una serie de mecanismos que, cooperando entre sí, solucionan la mayoría de estos problemas sin añadir incompatibilidades a los correos basados en RFC 822. Estos mecanismos se basan en:

• Definición de cinco nuevos campos en la cabecera del mensaje:
• Versión MIME: este campo debe contener al menos la versión 1.0
• Tipo de contenido: indica el tipo y subtipo de los datos contenidos en el cuerpo del mensaje.
• Codificación para la transferencia del contenido: tipo de codificación que se ha usado con el cuerpo del mensaje.
• Identificador de Contenido y Descripción de contenido: amplían la descripción de la información contenida en el mensaje. Por ejemplo para añadir información sobre una imagen o un archivo de voz.
• Se definen varios tipos de contenido y dentro de estos tipos varios subtipos.

10.4. API de JavaMail.

JavaMail es una API opcional a la versión estándar de Java (J2SDK) que proporciona funcionalidades de correo electrónico, a través del paquete **javax.mail**. Permite realizar desde tareas básicas como enviar, leer y componer mensajes, hasta tareas más sofisticadas como manejar gran variedad de formatos de mensajes y datos, y definir protocolos de acceso y transporte. Aunque a primera vista pueda parecer que su utilidad se orienta a construir clientes de correo-e de tipo Outlook, ThunderBird, etc., su aplicación se puede generalizar a cualquier programa Java que necesite enviar y/o recibir mensajes, como por ejemplo, aplicaciones de *intranets*, páginas JSP, etc.

Este paquete va unido al uso del paquete *JavaBeans Activation Framework* (JAF), que es también un paquete opcional a la versión estándar de Java (J2SDK), aunque tanto éste como JavaMail vienen incorporados en la versión empresarial (J2EE) . Este paquete es el que se encarga de la manipulación de los canales (*streams*) de bytes de los tipos MIME, que pueden incluirse en un mensaje.

Las funciones más importantes se detallan a continuación:

- Componer y enviar un mensaje: el primer paso que ha de realizar cualquier aplicación que pretenda enviar un correo electrónico es componer el mensaje. Es posible crear un mensaje de texto plano, es decir, un mensaje sin adjuntos que contenga exclusivamente texto formado por caracteres ASCII; pero es igualmente sencillo componer mensajes más completos que contengan adjuntos. También se pueden componer mensajes que contengan código HTML e incluso imágenes embebidas en el texto.
- Descargar Mensajes: se pueden descargar los mensajes desde la carpeta que se indique ubicada en el servidor que en ese momento se esté usando. Estos mensajes pueden ser de texto plano, contener adjuntos, HTML, etc.
- Usar *flags* (banderines): para indicar, por ejemplo, que un mensaje ha sido leído, borrado, etc., en función de cuáles de estos *flags* estén soportados por el servidor.
- Establecer una conexión segura: actualmente algunos servidores de correo requieren de una conexión segura, ya sea para descargar el correo almacenado en ellos, o para enviar mensajes a través de ellos. JavaMail ofrece la posibilidad de establecer este tipo de conexiones, indicando que se trata de una conexión segura, además de poder indicar otros parámetros, en algunos casos necesarios, como el puerto donde establecer la conexión. Esta capacidad está disponible desde la versión de JDK 1.3.
- Autenticarse en un servidor: ciertos servidores requieren autenticación ya no sólo para leer sino también para enviar. JavaMail ofrece también la posibilidad de autenticación a la hora de enviar un correo.
- Definir protocolos: JavaMail soporta por defecto los protocolos de envío y recepción SMTP, IMAP, POP3, (y sus correspondientes seguros a partir de la versión de JDK 1.3.2), si bien puede implementar otros protocolos.
- Manejar adjuntos: JavaMail ofrece la posibilidad de añadir adjuntos a los mensajes de salida, así como obtener y manipular los adjuntos contenidos en un mensaje de entrada.
- Búsquedas: JavaMail ofrece la posibilidad de buscar mensajes dentro de la cuenta de correo en el propio servidor -sin necesidad de descargar todo el correo- que concuerden con un criterio de búsqueda previamente definido. Para ello dispone de varias clases que permiten buscar mensajes con un *subject* determinado, un *from* concreto, etc., devolviendo un *arreglo* con los mensajes que cumplan estos criterios.
- Acuse de recibo y prioridad: Se puede incluir un acuse de recibo en los mensajes de salida, para que el remitente sepa si un mensaje ha sido leído o no por el destinatario, si bien no todos los servidores soportan esta funcionalidad. Además se puede establecer la prioridad de un mensaje de forma muy sencilla.

JavaMail se descarga de la dirección http://java.sun.com/products/javamail. Se recomienda la versión 1.4, ya que incorpora algunas mejoras sobre el tratamiento de los componentes MIME. De esta manera se obtiene un archivo **.zip** llamado **javamail-version**, donde **version** se corresponde con la versión de JavaMail descargada.

Al descomprimir el archivo que contiene JavaMail (**javamail-1_4.zip** por ejemplo), se observa que contiene una serie de archivos de texto, un archivo llamado **mail.jar** y tres directorios. A continuación se comentan qué contienen y para qué sirven estos directorios y archivos y, sobre todo, el significado del resto de archivos **.jar** que incorpora.

10.4.1 Directorios

El directorio **\docs** contiene archivos de texto correspondientes a cada versión de JavaMail en los que se explica los cambios introducidos en cada una de ellas.

Contiene también dos archivos en formato Adobe Acrobat, llamados **javamail-version.pdf** y **providers.pdf** que son el tutorial oficial de JavaMail desarrollado por Sun y un tutorial sobre cómo desarrollar un proveedor de servicios, es decir, dar soporte a algún protocolo no recogido en la implementación estándar de JavaMail. Por último, contiene un directorio llamado **\docs\javadocs**, en el que está la ayuda sobre la API en formato HTML generado por **javadoc**.

El directorio **\demo** contiene una serie de ejemplos en los que se usa JavaMail y que nos pueden servir para hacernos una idea de sus posibilidades. En el archivo **readme.txt** es posible consultar una lista en la que, de forma muy somera, se comenta que hace cada uno de estos ejemplos.

10.4.2 Archivos .jar

Por último, es fundamental comentar los archivos **.jar** contenidos en el archivo de instalación. Por un lado se suministra un archivo llamado **mail.jar** y del directorio **\lib**, que contiene otros cuatro archivos **.jar** llamados **imap.jar**, **mailapi.jar**, **pop3.jar** y **smtp.jar**.

- El archivo **mail.jar** contiene la implementación completa de la API JavaMail y todos los proveedores de servicio de Sun, es decir, el soporte para los protocolos IMAP, POP3 y SMTP y sus correspondientes versiones seguras. Por tanto la mayoría de los usuarios sólo necesitan este archivo **.jar** aunque, en ciertas ocasiones, puede ser necesario el uso de alguno de los otros archivos **.jar**.
- El archivo **mailapi.jar** contiene también la implementación completa de la API de JavaMail, pero sin incluir los proveedores de servicio (nótese que el tamaño de este archivo es aproximadamente la mitad que el de **mail.jar**). Estos proveedores vienen independientemente empaquetados en los respectivos archivos **imap.jar**, **pop3.jar** y **smtp.jar**. En los casos en que se quiera que nuestra aplicación tenga el menor tamaño posible, se puede usar **mailapi.jar** y añadir sólo el soporte para el protocolo o los protocolos que necesitemos de forma separada, incluyendo el correspondiente **.jar**.

10.5. Enviar mensajes.

El proceso básico para enviar un correo consiste en:

- Crear una sesión a la que le pasan las propiedades de la conexión, entre las que se ha debido incluir el servidor SMTP que se quiere usar.
- A continuación se crea un mensaje y se rellenan sus campos: asunto, texto del mensaje, emisor y destinatario.
- Por último se envía el mensaje.

Lo primero que se debe hacer es importar las clases del paquete javax.mail y del paquete javax.mail.internet. El primero de ellos contiene clases que modelan un sistema de correo genérico, es decir, características que son comunes a cualquier sistema de correo, como pueden ser un mensaje o una dirección de correo. javax.mail.internet contiene clases específicas para modelar un sistema de correo a través de Internet, como por ejemplo clases para representar direcciones de Internet o usar características de las extensiones MIME. También será necesario usar la clase java.util.Properties, que representa un conjunto de propiedades, y que permitirá establecer el servidor SMTP como una propiedad de la conexión.

ombre	Descripción
mail.debug	Hace que el programa se ejecute en modo depurador, es decir, muestra por pantalla todas las operaciones que realiza y los posibles errores. Por defecto no está activada. P.ej. props.put ("mail.debug", "true")

mail.from	Añade la dirección del remitente en el mensaje. P.ej: props.put ("mail.from", "ignacio@gmail.com")
mail.protocolo.host	Dirección del servidor para el protocolo especificado, es decir, el que se va a usar para enviar o recibir. P.ej. props.put ("mail.smtp.host", "smtp.ono.com")
mail.protocolo.port	Número de puerto del servidor que hay que usar. P.ej. props.put ("mail.pop3.port", "995")
mail.protocolo.auth	Indica que vamos a autenticarnos en el servidor. P.ej. props.put ("mail.smtp.auth", "true")
mail.protocolo.user	Nombre de usuario con que conectarse al servidor. P.ej. props.put ("mail.pop3.user", "pruebasmail2008")

A continuación es necesario crear una sesión de correo usando para ello un objeto de la clase Session; para ello hay que hacer uso del objeto de tipo Properties creado en el paso anterior. Un objeto de la clase Session representa una sesión de correo y reúne sus principales características, de manera que cualquier operación posterior de envío/recepción de mensajes se hace en el contexto de una sesión. Dichos objetos se crean haciendo uso de dos métodos estáticos que devuelven un objeto de este tipo.

El primero de ellos devuelve la sesión por defecto, siempre la misma cuantas veces sea invocado este método. Este método recibe como parámetro el objeto de tipo Properties creado anteriormente y un objeto de la clase Authenticator.

El segundo método es similar al primero con la diferencia de que las aplicaciones no comparten esta sesión, pues cada vez que se invoca retorna una sesión diferente.

Una vez especificado el servidor SMTP y creada la sesión de correo-e, ya se puede proceder a la creación del mensaje que se quiere enviar. Para modelar un mensaje de correo electrónico existe la clase Message del paquete javax.mail, pero al ser esta clase abstracta es necesario crear una instancia de la única clase estándar que la implementa, la clase MimeMessage. Esta clase representa un mensaje de correo electrónico de tipo MIME.

Constructor	Comportamiento
public MimeMessage(Session session)	Es el constructor que se suele usar y devuelve un mensaje vacío.
public MimeMessage (MimeMessage source)	Crea un mensaje a partir de otro mensaje que se le pasa como parámetro. Es una copia del otro mensaje.
public MimeMessage(Session session, InputStream in)	Construye un mensaje a partir de los datos devueltos por in que serán el contenido y las cabeceras del mensaje.

protected MimeMessage(Folder folder, InputStream in, int msgnum)	Crea el mensaje leyendo los datos del canal de entrada in contenido en la carpeta folder, y que ocupa la posición msgnum dentro de ésta.
protected MimeMessage(Folder folder, int msgnum)	Crea un mensaje a partir del mensaje que ocupa la posición msgnum dentro de la carpeta folder.
protected MimeMessage(Folder folder, InternetHeaders headers, byte[] content, int msgnum)	Crea un mensaje a partir del mensaje que ocupa la posición msgnum dentro de la carpeta folder, leyendo las cabeceras desde headers y el contenido desde content.

A continuación hay que proceder a rellenar los atributos (asunto, fecha de envío, remitente, etc.) y el contenido del mensaje.

Método	Comportamiento
void setFrom(Address dirección)	Establece como dirección del remitente la que se pasa como parámetro.
void setSubject(String asunto)	Establece como asunto el String quese pasa como parámetro.
void setContent(Object obj, String type)	Establece el contenido del mensaje del tipo que le indicamos. Puede ser "text/plain" si el mensaje se rellena con un String sencillo, "text/html" si se hace con texto en formato HTML, o "multipart/alternative", "multipart/related", "multipart/mixed", si el mensaje se rellena con un objeto Multipart.
void setContent (Multipart mp)	Rellena el mensaje con el Multipart que se le pasa y cuyo tipo MIME habrá sido establecido al crearlo.
void setText(String texto)	Es equivalente a usar el método setContent insertando un String e indicando que se trata de texto plano ("text/plain").

A continuación se debe rellenar el atributo correspondiente al receptor del mensaje. Para ello se utiliza el método addRecipient de la clase Message al que se le pasan dos parámetros. El primero es un objeto del tipo Message.RecipientType (tipo de destinatario), que es una clase interna a la clase Message, que tiene tres constantes: TO, CC (*Carbon Copy*) y BCC (*Blind Carbon Copy*), que son los tres tipos de recipientes o destinatarios. Con esto se especifica si la dirección del destinatario se añade en el TO, el CC o el BCC del mensaje. El segundo parámetro es la dirección que se añadirá al recipiente y es del tipo InternetAddress, al igual que en el método setFrom.

Este apartado finaliza rellenando ahora el contenido del mensaje, para lo que se usa el método setText de la clase Message Este método además de establecer como contenido del mensaje el que se le especifique a través de un String, establece que el tipo de contenido es texto plano (*text/plain*).

Con todo esto, el mensaje ya está listo para ser enviado, lo que constituye el último paso de nuestro programa. Para enviarlo se usa la clase Transport que se encarga de esto de una forma muy sencilla mediante sus métodos estáticos.

Esta clase tiene sobrecargado el método send, de forma que también permite enviar un mensaje a varios destinatarios, mandándole a este método un *arreglo* de direcciones. De esta forma se ignoran las direcciones que pudiesen estar definidas en el mensaje y se envía sólo y exclusivamente a las especificadas en el *arreglo.*

Esta clase tiene sobrecargado el método send, de forma que también permite enviar un mensaje a varios destinatarios, mandándole a este método un *arreglo* de direcciones. De esta forma se ignoran las direcciones que pudiesen estar definidas en el mensaje y se envía sólo y exclusivamente a las especificadas en el *arreglo.*

10.5.1. Ejemplo de envío de mensajes.

```
import java.util.Properties;
import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendMail {

	private Properties initialProperties;

	private SMTPAuthenticator sMTPAuthenticator;
	private String SMTP_HOST_NAME = "";
	private String SMTP_AUTH_USER = "";
	private String SMTP_AUTH_PWD  = "";
	private String emailMsgTxt     = "";
	private String emailSubjectTxt  = "";
	private String emailFromAddress = "";
	private String emailToAddress = "";


	public SendMail() {
		this.initialProperties = new Properties();
	}


	public void setInitialProperties() throws Exception {

		SMTP_HOST_NAME = "132.248.59.1";
		SMTP_AUTH_USER = "principe";
		SMTP_AUTH_PWD = "hola123";
		emailFromAddress = "principe@fi-b.unam.mx";
		emailToAddress = "principe@fi-b.unam.mx";
		emailSubjectTxt = "titulo";
		emailMsgTxt = "Aqui va el cuerpo del mail";
		this.initialProperties.put("mail.smtp.host", SMTP_HOST_NAME);
		this.initialProperties.put("mail.smtp.auth", "true");
	}
```

```
    public boolean postMail() throws MessagingException {

        Authenticator auth = new SMTPAuthenticator();
        Session session = Session.getDefaultInstance( this.initialProperties, auth
);

        Message msg = new MimeMessage(session);

        InternetAddress addressFrom = new InternetAddress( emailToAddress );
        msg.setFrom(addressFrom);

        InternetAddress addressTo = new InternetAddress( emailFromAddress );
        msg.setRecipient(Message.RecipientType.TO, addressTo);

        msg.setSubject(emailSubjectTxt);
        msg.setContent(emailMsgTxt, "text/plain");
        Transport.send(msg);

        return true;
    }

    public SMTPAuthenticator getSMTPAuthenticator() {
        return sMTPAuthenticator;
    }

    public void setSMTPAuthenticator(SMTPAuthenticator authenticator) {
        sMTPAuthenticator = authenticator;
    }

    public Properties getInitialProperties() {
        return initialProperties;
    }

    private class SMTPAuthenticator extends javax.mail.Authenticator
    {
        public PasswordAuthentication getPasswordAuthentication()
        {
            String username = SMTP_AUTH_USER;
            String password = SMTP_AUTH_PWD;
            return new PasswordAuthentication(username, password);
        }
    }

    public static void main(String args[]) throws Exception
    {
        SendMail smtpMailSender = new SendMail();
        smtpMailSender.setInitialProperties();
        smtpMailSender.postMail();
        System.out.println("Mail enviado");
    }
}
```

10.6 Recibir Mensajes.

Los pasos básicos para recuperar los mensajes del servidor de correo son:

- Crear una sesión.
- Crear un almacén (Store) a partir de esta sesión y conectarse a él.
- Abrir una carpeta del almacén y recuperar los mensajes.
- Procesar estos mensajes convenientemente.

Los dos protocolos más usados para recuperar mensajes son POP3 e IMAP, siendo POP3 el más usado, El hecho de usar POP3 acarrea una serie de limitaciones que pueden conducir a errores: muchas de las capacidades de JavaMail no están soportadas por el protocolo POP3 y si se intenta usarlas se lanzará la excepción MethodNotSupportedException. En especial hay que mencionar las limitaciones con respecto al manejo de carpetas y de banderines, ya que POP3 sólo admite una única carpeta llamada **INBOX** y sólo permite trabajarcon el banderín que permite borrar los mensajes del servidor. Por último hay que indicar que POP3 no proporciona un mecanismo para obtener fecha y hora en que un mensaje es recibido, por lo que el método getReceivedDate devolverá null.

Los mismos paquetes y clases que en el caso del envío, asíque el primer paso es importar estos paquetes. Además es necesario importar el paquete de entrada java.io para el posterior manejo de excepciones.

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
```

A continuación hay que obtener las propiedades del sistema y con ellas crear una sesión de correo. La única iferencia con respecto al ejemplo anterior, es que en este caso no es necesario añadir ninguna propiedad a las ya obtenidas:

A continuación crearemos un objeto de la clase Store apropiado para este protocolo, usando la sesión creada anteriormente, y con el que es posible conectarse al servidor indicado nombre de usuario y contraseña. Conceptualmente un Store representa nuestro buzón de correo dentro del servidor, es decir, nuestra cuenta de correo. La clase Store permite acceder a una estructura de carpetas (clase Folder) y a los mensajes contenidos en cada una de ellas.

Una vez conectados a nuestro Store hay que abrir la carpeta **INBOX** que, como ya se comentó, es la única disponible en POP3. Para ello se debe crear un objeto Folder a partir de la clase Store y que haga referencia a esta carpeta. A continuación se usa el método open y se expecificael modo en que se desea abrirla:

- Si sólo queremos leer los mensajes, sin borrarlos, se debe usar el modo de sólo lectura READ_ONLY.
- En caso de que se quieran borrar los mensajes se debe usar el modo READ_WRITE.

Con la carpeta ya abierta resulta sumamente sencillo recuperamos los mensajes almacenados en ella. Para ello usaremos el método getMessages de la clase Folder, sin parámetros, que devuelve un *array* de mensajes de tipo Message. Este método también permite que se le pase como parámetro un arreglo de enteros (ordinales) con los números de los mensajes que queremos recuperar, o también que le pasemos dos enteros que indican que se desean recuperar los mensajes almacenados entre esas dos posiciones.

El último paso consiste en procesar los mensajes de alguna forma

Método	Comportamiento
Address[] getFrom()	Devuelve un array en el que hay tantas direcciones como remitentes, null si el campo from no está presente en el mensaje o un array vacío si el mensaje no contiene ninguna dirección en el from.

Date getSentDate()	Devuelve la fecha en la que el mensaje se envió.
String getSubject()	Devuelve el asunto del mensaje.
Object getContent()	Devuelve el contenido del mensaje Si éste es de tipo texto plano devolverá un String y si es de tipo Multipart devolverá un objeto de esta clase.
javax.activation.DataHandler getDataHandler()	Devuelve un DataHandler con el que poder manejar el contenido del mensaje. Con un DataHandler es posible manejar contenidos complejos a través de un canal de entrada.

El último paso consiste en cerrar el Folder y el Store creados. Al cerrar el Folder se le pasa como parámetro un valor boolean. Si este vale true se aplicarán los cambios realizados sobre los mensajes, es decir, si hemos marcado un mensaje como borrado entonces éste se borrará.

En cuanto a las excepciones, podrían producirse en los siguientes casos (todas de tipo MessagingException a no ser que se indique lo contrario):

- El método connect de la clase Store lanzará una excepción si falla al intentar conectarse al servidor.
- El método getStore de la clase Session lanzará una excepción si no puede manejar el protocolo especificado.
- El método getFolder de la clase Store lanzará una excepción si se intenta obtener una carpeta de un servidor al que no estamos conectados.
- El método open de la clase Folder lanzará una excepción si se intenta abrir una carpeta que no existe.
- El método getMessages de la clase Folder lanzará una excepción si se intenta obtener mensajes de una carpeta que no existe.
- Los métodos getSubject, getSentDate y getFrom de la clase Message lanzarán una excepción si fallan al intentar acceder al asunto o la fecha de un mensaje.
- El método getContent de la clase Message también puede generar una excepción, pero esta vez de tipo IOException. Esta excepción será lanzada cuando se produzca un error mientras se está recuperando el contenido del mensaje.
- El método close de la clase Folder lanzará una excepción si se intenta cerrar una carpeta que no está abierta.
- El método close de la clase Store lanzará una excepción si se produce algún error mientras se está cerrando el Store.

Por todo esto, el bloque de instrucciones que abarca desde que se obtiene el Store hasta que se cierra, debe incluirse dentro de un bloque **try-catch** que recoja los dos tipos de excepciones que se pueden dar.

10.6.1. Ejemplo de obtención de mensajes.

```
import java.util.Properties;
import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
```

```
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.mail.Store;
import javax.mail.Folder;
import java.io.*;

public class GetMail {

	private Properties initialProperties;

	private SMTPAuthenticator sMTPAuthenticator;
	private String SMTP_HOST_NAME = "";
	private String SMTP_AUTH_USER = "";
	private String SMTP_AUTH_PWD  = "";
	private String emailMsgTxt     = "";
	private String emailSubjectTxt = "";
	private String emailFromAddress = "";
	private String emailToAddress = "";
	  private int numberOfRecipients = 0;

	public GetMail() {
		this.initialProperties = new Properties();
	}

	public void setInitialProperties() throws Exception {

		SMTP_HOST_NAME = "132.248.59.1";
		SMTP_AUTH_USER = "principe";
		SMTP_AUTH_PWD = "hola123";
	}

	public boolean getMail() throws MessagingException {
		Session session = Session.getDefaultInstance( this.initialProperties, null
);

			try {

				Store store = session.getStore("pop3");
				store.connect(SMTP_HOST_NAME,SMTP_AUTH_USER,SMTP_AUTH_PWD);
			System.out.println("Conectado al servidor ?"+store.isConnected());

				Folder folder = store.getFolder("INBOX");
				folder.open(Folder.READ_ONLY);

				Message [] mensajes = folder.getMessages();

			System.out.println("numero de mensajes "+  mensajes.length);
				for (int i=0; i < mensajes.length; i++) {
					System.out.println("Mensaje " + i + ":\n" +
					"\tAsunto: " + mensajes[i].getSubject() + "\n" +
					"\tRemitente: " + mensajes[i].getFrom()[0] + "\n" +
					"\tFecha de Envio: " + mensajes[i].getSentDate() + "\n" +
					"\tContenido: " + mensajes[i].getContent() + "\n");
				}
			folder.close(false);
```

```
                store.close();
            } catch (MessagingException me) {
                System.out.println(me.toString());
            } catch (IOException ioe) {
                System.out.println(ioe.toString());
            }

        return true;
    }

    public SMTPAuthenticator getSMTPAuthenticator() {
        return sMTPAuthenticator;
    }

    public void setSMTPAuthenticator(SMTPAuthenticator authenticator) {
        sMTPAuthenticator = authenticator;
    }

    public Properties getInitialProperties() {
        return initialProperties;
    }

    private class SMTPAuthenticator extends javax.mail.Authenticator
    {
        public PasswordAuthentication getPasswordAuthentication()
        {
            String username = SMTP_AUTH_USER;
            String password = SMTP_AUTH_PWD;
            return new PasswordAuthentication(username, password);
        }
    }

    public static void main(String args[]) throws Exception
    {
        GetMail smtpMailGetter = new GetMail();
        smtpMailGetter.setInitialProperties();
        smtpMailGetter.getMail();
        System.out.println("Mails obtenido");
    }
}
```

10.7. Resumen de paquetes y clases.

En este apartado, y a modo de resumen, se muestran los paquetes que componen JavaMail y se profundiza en las principales clases que los integran. En la tabla pueden verse los paquetes que componen JavaMail y la descripción de cada uno de ellos, mientras que los siguientes epígrafes se centran en las clases más importantes y para qué se utilizan.

Paquete	Descripción
com.sun.mail.imap	Proporciona soporte para manejar el protocolo IMAP y poder acceder a un servidor de correo que use este protocolo.
com.sun.mail.pop3	Proporciona soporte para manejar el protocolo POP3 y poder acceder a un servidor de

	correo que use este protocolo.
com.sun.mail.smtp	Proporciona soporte para manejar el protocolo SMTP y poder usar un servidor basado en este protocolo para el envío de mensajes.
javax.mail	Contiene clases que modelan un sistema de correo, definiendo capacidades que son comunes a la mayoría de ellos. Este paquete es el más importante de los que componen JavaMail.
javax.mail.event	Define clases que se usan para el manejo de eventos producidos por las clases que componen JavaMail.
javax.mail.internet	Define características que son específicas de sistemas de correo que funcionan sobre la red Internet, y que se basan en las extensiones MIME.
javax.mail.search	Proporciona mecanismos para realizar búsquedas de mensajes según una serie de criterios, dentro de una carpeta.
javax.mail.util	Proporciona tres clases de utilidad que controlan distintos tipos de canales (streams).

10.7.1. La clase Message

Message es una clase abstracta que modela un mensaje de correo electrónico mediante un conjunto de atributos que definen su cabecera, y un contenido que define su cuerpo. Entre los atributos de la cabecera están los típicos de *from*, *fecha de envío*, *asunto*, etc., que se insertan y se recuperan mediante métodos. Para hacer referencia a los destinatarios de un mensaje, la clase Message proporciona la subclase Message.RecipientType que incluye constantes que hacen referencia a *to*, *Bcc* y *Cc*.

Por defecto, el contenido es de tipo String conteniendo texto plano, pero se puede enviar otro tipo de información haciendo uso de otras clases que heredan de Message. Una de ellas es MimeMessage, que implementa el RFC822 y los estándares MIME, y proporciona una gran potencia al permitir, incluso, construir mensajes constituídos por bloques que contienen información de diferente índole: texto, gráficos, música, documentos de aplicaciones ofimáticas (WordPerfect, StarOffice, Macromedia Flash, etc.), etc.

Esta clase se utiliza tanto para modelar los mensajes a enviar (que suelen crearse haciendo uso de MimeMessage), como para los que se reciben desde un buzón a través de la clase Folder.
Esta clase implementa la interfaz Part, que define un conjunto de características comunes a la mayoría de sistemas de correo, además de métodos para obtener y establecer el contenido del mensaje, y de algunos de los campos de la cabecera del mismo.

10.7.2. La clase Session.

Session define una sesión de conexión a una cuenta de correo. A este respecto, permite modificar opciones de configuración a la hora de conectarse y autenticarse con objeto de interactuar correctamente con el servidor de correo. Esta clase no tiene un constructor público, por lo que para obtener una sesión de correo tendremos que usar uno de los dos métodos estáticos que proporciona Session: getDefaultInstance y getInstance. El primer método devuelve

siempre el mismo objeto Session, esto es, no crea una sesión nueva cada vez por lo que facilita el que la sesión de correo sea compartida por varias aplicaciones. Es el método más utilizado porque consume pocos recursos y sumnistra la funcionalidad suficiente en la mayoría de los casos. El segundo método crea una sesión diferente cada vez que se invoca, por lo que no es compartida por distintas aplicaciones.

Sea cual sea el método usado, las características de configuración para la conexión han debido cargarse previamente en un objeto de tipo java.util.Properties, el cual hay que pasarle como primer parámetro de los dos que necesita. Estas propiedades se pueden obtener (y modificar posteriormente) a partir de las propiedades del sistema (usando el método System.getProperties() o creando un objeto vacío con el constructor Properties() y añadiéndole las propiedades que se necesiten. Desde un punto de vista pragmático, un objeto Properties no es más que una tabla de dispersión (tabla *hash*) que contiene pares (propiedad-valor). El segundo parámetro necesario para construir una sesión es un objeto de tipo Authenticator que puede ser null en caso de que el servidor no necesite autenticación.

10.7.3. Las clases Store y Folder.

Conceptualmente un objeto Store representa un buzón de correo dentro del servidor, es decir, la cuenta de correo a la que conectarsepara recibir y desde la que enviar mensajes. Esta clase permite manipular la estructura de carpetas (objetos de tipo Folder) y los mensajes contenidos en cada una de ellas.

La conexión a un objeto Store se hace a partir de un objeto Session utilizando los métodos getStore en sus diferentes versiones (la más usual consiste en indicar sencillamente el protocolo que se desea usar para la conexión con el servidor especificado en las propiedades de la sesión).

Una vez conectados a un Store podremos obtener una carpeta determinada (con el método getFolder()) u obtener la carpeta por defecto (con el método getDefaultFolder()). La carpeta por defecto es la raíz de la estructura de carpetas almacenadas en la cuenta de correo.

Una sesión de correo debe finalizar cerrando el objeto Store mediante el método close(). Por último comentar que una sesión puede producir eventos cuando se intenta crear, borrar o renombrar alguna de sus carpetas; estos eventos pueden ser capturados a través de un objeto delegado de tipo FolderListener.

Una vez obtenido un objeto Store, se hace posible acceder a sus carpetas con objeto de manejar los mensajes de correo que éstas contienen. Así, la clase Folder representa una carpeta almacenada en el buzón de correo, y puede contener mensajes de correo, otras carpetas o ambas cosas, según se especifique en sus propiedades. La carpeta principal o carpeta por defecto es la carpeta **INBOX**, que se corresponde con la bandeja de entrada, y todas las demás carpetas son descendientes -directa o indirectamente- de la carpeta **INBOX**.

Es importante señalar también que el manejo de una carpeta depende del protocolo usado. Por ejemplo, con el protocolo IMAP es posible acceder a cualquier carpeta dentro del buzón: bandeja de entrada (**INBOX**), papelera, enviados, etc., además de las carpetas definidas por el usuario. Estas carpetas se acceden siguiendo un método parecido al utilizado en los sistemas operativos, donde se utiliza una barra separadora entre el nombre de un directorio y el siguiente. En este caso se suele utilizar un punto ('.') como separador teniéndose, por ejemplo, **INBOX.Trash**.

Por otro lado, utilizando el protocolo POP3 sólo es posible acceder a la bandeja de entrada, es decir, a la carpeta **INBOX**.

Propiedad	Descripción
mail.debug	Hace que el programa se ejecute en modo depurador, es decir, muestre por pantalla todas las operaciones que realiza y los posibles errores. Por defecto no está activada. P.e. props.put ("mail.debug","true").

mail.from	Añade la dirección del remitente en el mensaje. P.e. props.put ("mail.from", lenin@gmail.com).
mail.protocolo.host	Dirección del servidor para el protocolo especificado, es decir, el que se va a usar para enviar o recibir. P.e. props.put ("mail.smtp.host" ,"smtp.oño.com").
mail.protocolo.port	Número de puerto por el que escucha el servidor. P.e. props.put ("mail.pop3.port", "995").
mail.protocolo.auth	Indica si vamos a autenticarnos en el servidor o no. P.e. props.put("mail.smtp.auth", "true").
mail.protocolo.user	Nombre de usuario cuando nos conectamos al servidor. P.e. props.put ("mail.pop3.user", "pruebasmail2005").

10.7.4. La clase Transport

Esta es la clase encargada de enviar el correo a su destinatario. En la mayoría de las ocasiones se usará el método estático Transport.send(), si bien podría obtenerse un objeto de esta clase usando el método getTransport() de la clase Session, sin parámetros, o indicándole el protocolo que usamos o la dirección del receptor del mensaje.

Para esto último, la clase Transport permite conectarse a un servidor y enviar los mensajes de la siguiente forma:

- Se obtiene un objeto Transport para el protocolo indicado.
- Nos conectamos a él.
- Se envía el mensaje.
- Cerrar el objeto.

11. XML

11.1. Introducción

XML hace los datos portables. La plataforma Java hace los datos portables. Los APIs Java para XML hacen fácil el uso del XML. Pongamos todo esto junto y tendremos la combinación perfecta: portabilidad de datos, portabilidad de codigo y facilidad de uso. De hecho, con los APIs de Java para XML, podremos obtener los beneficios del XML con un uso directo de XML muy pequeño.

XML (eXtensible Markup Language) no es, como su nombre podría sugerir, un lenguaje de marcado. XML es un meta-lenguaje que nos permite definir lenguajes de marcado adecuados a usos determinados.

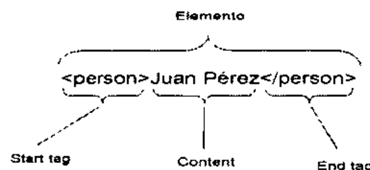
XML (eXtensible Markup Language) es un estándar industrial para representar datos, independiente del sistema. Al igual que HTML (HyperText Markup Language), XML encierra los datos en etiquetas, pero hay importantes diferencias entre los dos lenguajes de marcas. Primero, las etiquetas XML tiene relación con el significado del texto que encierran, mientras que las etiquetas HTML especifican cómo mostrar el texto encerrado. El siguiente ejemplo XML muestra una lista de precios con el nombre y el precio de dos cafés:

```
<priceList>
    <coffee>
        <name>Mocha Java</name>
        <price>11.95</price>
    </coffee>
    <coffee>
        <name>Sumatra</name>
        <price>12.50</price>
    </coffee>
</priceList>
```

Las etiquetas <coffee> y </coffee> le dicen al analizador que la información que hay entre ellas trata sobre un café. Las otras dos etiquetas dentro de las etiquetas <coffee> especifican que la información encerrada son el nombre del café y su precio. Como las etiquetas XML especifican el contenido y la estructura de los datos que encierran es posible hacer cosas como archivar o buscar datos.

Un segunda diferencia importante entre XML y HTML es que las etiquetas XML son extensibles, permiténdonos escribir nuestras propias etiquetas XML para describir nuestro contenido. Con HTML, estábamos limitados a usar sólo aquellas etiquetas que habían sido predefinidas en la especificación HTML.

Cómo se observa en el ejemplo anterior, un docuemnto xml esta fomado por:



Todos los documentos XML deben de tener un elemento raíz:

```
                    Start tag elemento raiz
<people>
    <person>Juan Pérez</person>
    <person>Pedro Mata</person>
    <person>Maria Bonita</person>
</people>
                    End tag elemento raiz
```

11.1.1. Modelado de datos.

En los atributos se utiliza la pareja de nombre valor dentro del start tag sepadaro por el signo de =. Cada nombre de atributo debe de ser único y no hay límite en el número de atributos de un elemento.

```
<people>
    <person>
        <name>Juan Perez</name>
        <pet species="cat">Fluffy</pet>
    </person>
    <person>
        <name>Maria Bonita</name>
        <pet species ="dog">Fido</pet>
    </person>
</people>
```

Usar elementos cuando:
- el contenido es más que unas cuantas palabras.
  - el orden es importante.
  - se requiere mostrar la estructura.

Usar atributos cuando:
- se requiere modificar el contenido de un elemento.
- se desee restringir valores de los datos.

Para las reglas de los nombres se deben de utilizar las siguientes caraterísticas:

- Letras A-Z y a-z.
- Números 0-9.
- Caracteres especiales:
- guión bajo (_).
- Guión (-).
- Punto (.).
- Deben empezar con letra o guión bajo.

Cuando se va a hacer un documento XML, no se requieren pero se usan comúnmente:

- Declaración: <?xml version="1.0" encoding="UTF-8" standalone="yes" ?> (debe ser la primera línea del archivo)
- Comentarios: <!-- Esto es un comentario -->

11.1.2. DTD: Document Type Definition.

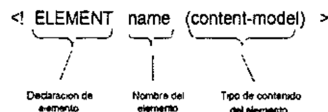
Con la extensibilidad que proporciona XML, podemos crear las etiquetas que necesitemos para un tipo de documento en particular. Definimos las etiquetas usando un esquema de lenguaje XML. Un esquema describe la estructura de un conjunto de documentos XML y puede usarse para limitar los contenidos de los documentos XML. Probablemente el lenguaje de esquema más ampliamente utilizado es el Document Type Definition. Un esquema escrito en este lenguaje se llama DTD. El siguiente DTD define las etiquetas usadas en el documento XML de la lista de precios. Especifica cuatro etiquetas (elementos) y además especifica qué etiquetas podrían ocurrir (o es necesario que ocurran) dentro de otras etiquetas. El DTD también define la estructura de árbol de un documento XML, incluyendo el orden en que deberían ocurrir las etiquetas:

```
<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```

La primera línea del ejemplo nos dá el elemento de más alto nivel priceList, lo que significa que todas las otras etiquetas del documento deben ocurrir entre las etiquetas <priceList> y </priceList>. La primera línea también dice que el elemento priceList debe contener uno o más elementos coffee (indicado por el signo más). La segunda línea especifica que cada elemento coffee debe contener un elemento name y un elemento price, en este orden. La tercera y cuarta línea especifican que los datos entre las etiquetas <name> y </name> y entre las etiquetas <price> y </price> son del tipo caracter y deberían analizarse. El name y el price de cada coffee son el texto real que componen la lista de precios.

Un DTD, como el del ejemplo anterior, es lo que le da al XML su portabilidad. Si una aplicación recibe un documento priceList en formato XML y tiene el DTD priceList, puede procesar el documento de acuerdo a las reglas especificadas en el DTD. Por ejemplo, dando el DTD priceList, un analizador conocerá la estructura y el contenido de cualquier documento XML basado en esa DTD. Si el analizador es un analizador validante, sabrá que el documento no es válido si contiene elementos que no están incluidos en la DTD, como <tea>, o si el elemento price precede al elemento name.

Por lo tanto, un DTD debe de contener:



El *content-model* puede ser:

- #PCDATA – Indica que se puede usar texto en este elemento.
- Elements – Indica que este elemento puede contener otro u otros elementos.
- EMPTY – Indica que este elemento no contiene texto ni otros elementos, sólo puede contener atributos.
- ANY – Puede contener cualquier cosa. No se recomienda usar.

El formato general que debe de serguir un DTD es el siguiente:

```
<!DOCTYPE root-element [external-ID] [public-id] [uri]
[
    <!-- Contenido del DTD -->
]>
```

- DOCTYPE y root-element son requeridos.
- external-ID indica que se debe leer un DTD en el documento, su valor puede ser:
    - PUBLIC - especifica que el DTD es público y debe seguir public-id y uri.
    - SYSTEM - indica un uri sin identificador público.
    - uri indica al parser XML donde encontrar el archivo DTD.
    - Se puede incluir contenido del DTD en el documento XML. Si se incluye tiene preferencia sobre el archivo externo.

En el siguente ejemplo se muestra un DTD interno:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE people [
<!-- DTD for People Example -->
<!ELEMENT people (person)>
<!ELEMENT person (#PCDATA)>
]>
<people>
<person>Jane Doe</person>
</people>
```

En el siguiente ejemplo se muestra un DTD externo:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE people SYSTEM "03_03.dtd" >
<people>
<person>Jane Doe</person>
</people>
```

(archivo XML)

```
<!-- DTD for People Example -->
<!ELEMENT people (person)>
<!ELEMENT person (#PCDATA)>
```

(archivo DTD)

A continuación se muestran ejemplos de los diferentes tipos de configuración de los archivos DTDs. En los ejemplos se utilizan los siguientes cuantificadores.

```
Cuantificadores:
? - Cero o uno.
* - Cero o más.
+ - Uno o más.
```

a. Especificación de cantidades:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE people SYSTEM "03_04.dtd" >
<people>
<person>Jane Doe</person>
</people>
```

```
<!-- DTD for People Example -->
<!ELEMENT people (person?)>
<!ELEMENT person (#PCDATA)>
```

b. Especificación de orden:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE people SYSTEM "03_05.dtd" >
<people>
<person>
<name>Jane Doe</name>
<pet>Fluffy</pet>
</person>
</people>
```