CONECTIVIDAD A BASES DE DATOS. JDBC.

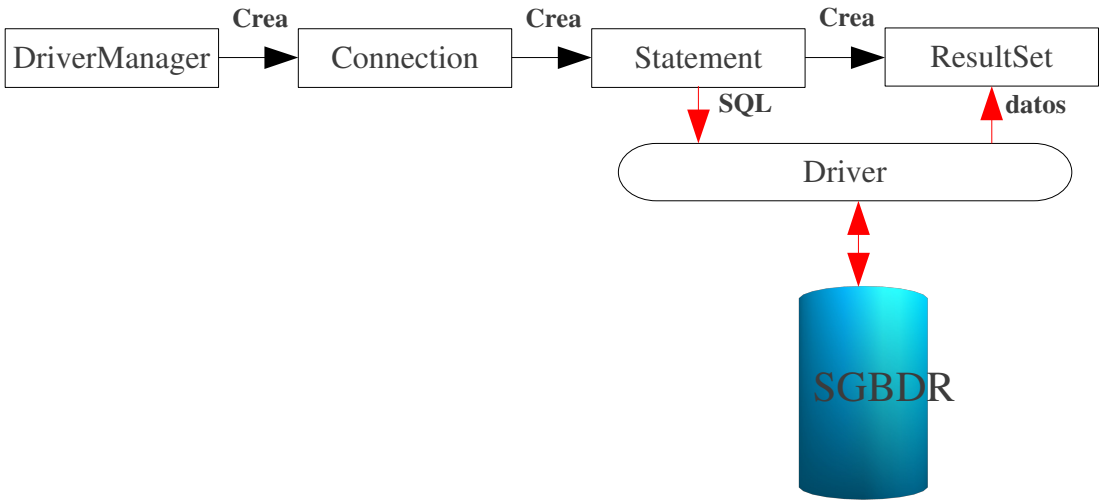
Introducción a JDBC

Se trata de una API (*Java DataBase Connectivity*) que permite conectarse a una base de datos (BD), realizar consultas mediante el lenguaje *SQL* y manipularla. Está ubicada en los paquetes *java.sql* (API estándar) y *javax.sql* (API extendida) . Admite todos los dialectos de SQL, y ofrece funcionalidades que van más allá de él. JDBC es además el mecanismo necesario para conectar con una BD, independientemente de la BD que se trate.

La arquitectura de JDBC está basada en un conjunto de interfaces y clases de Java que permiten conectar con bases de datos, crear y ejecutar sentencias SQL y recuperar y modificar datos de una base de datos.

La versión actual de JDBC es la 4.0 y los paquetes *java.sql* y *javax.sql* ya están incorporados en la *J2SE 1.4* y posteriores.

Esquema de funcionamiento



La clase *java.sql.DriverManager* es la responsable de establecer conexiones con los orígenes de datos, a través de los drivers JDBC. Los drivers de base de datos JDBC se definen mediante clases que implementan la interfaz *java.sql.Driver*. Un driver JDBC sabe cómo convertir las peticiones SQL para una base de datos concreta.

La primera acción que se debe llevar a cabo es cargar el driver adecuado para que el código Java pueda comunicarse con la base de datos empleada.

El procedimiento deberá ser el siguiente:

1. *Cargar la clase del driver.*
Lo primero será cargar la clase del driver y registrarla en el programa. Hay dos formas de hacerlo:

   a) Usar el método *Class.forName()*.
   b) Identificar la clase *Driver* en la propiedad del sistema *jdbc.drivers*[66].

66 Se emplearía la instrucción System.setProperty("jdbc.drivers","com.mysql.jdbc.Driver"); en lugar de usar Class.forName().

Nos centraremos en el primero. La sintaxis es la siguiente:

Class.forName("<nombre de la clase del driver>");

Cada clase *Driver* típicamente creará una instancia de sí misma cuando sea cargada, y registrará esta instancia automáticamente con una llamada al método *registerDriver()* de la clase *DriverManager*.
Ejemplo:

Class.forName("com.mysql.jdbc.Driver");

2. Crear la conexión.
Una vez esté cargado y registrado el driver, necesitamos establecer una conexión con la base de datos. El objeto *java.sql.Connection* representa una conexión con un origen de datos concreto. Para crear el objeto *Connection* se recurre al método *getConnection()* de *DriverManager*. El argumento es un *String* que define el URL donde está la base de datos. La URL de JDBC tiene el siguiente formato:

jdbc:<subprotocolo>:<identificador origen datos>

donde *subprotocolo* identifica el driver JDBC a usar, y el i*dentificador origen datos* dependerá del driver empleado. En el caso de un servidor de base de datos, el nombre del servidor debería estar en el URL. El URL también puede incluir información sobre el usuario y su clave. Ejemplo:

Fuente de datos: *jdbc:mysql://frolik:3306/usuarios*

Connection conDB= DriverManager.getConnection("jdbc:mysql://localhost:3306/usuarios","root","root");

Después de crear la conexión se obtiene un objeto *Connection* asociado a la misma, con el que se conformarán todos los procesos de ejecución de instrucciones SQL.

3. Crear el objeto Statement.
El objeto de la clase *java.sql.Statement* se emplea para enviar consultas y comandos a la base de datos. El código es:

Statement s = conDB.createStatement();

JDBC ofrece tres clases para enviar sentencias SQL a la BD y tres métodos en la interfaz *Connection* crean instancias de estas clases[67]:

- Un objeto *Statement* es un contenedor o mecanismo de transporte para enviar/ejecutar sentencias SQL y recuperar los resultados generados.
- Los objetos *PreparedStatement* posibilitan ejecutar sentencias SQL que toman parámetros como argumentos de entrada.
- *CallableStatement* se emplea para ejecutar procedimientos almacenados (grupos de sentencias SQL que son llamadas por un nombre y que se encuentran en la base de datos). Posee parámetros de entrada, salida y entrada/salida.

4. Ejecutar la sentencia SQL.
Para ejecutar una consulta o actualización puede emplearse el método *executeQuery()*. Este método devuelve un objeto de tipo *java.sql.ResultSet*. Ejemplo:

*ResultSet rs = s.executeQuery("select * from user");*

El método *executeQuery()* posibilita ejecutar consultas. Se debe emplear el método *executeUpdate()* para ejecutar sentencias SQL que modifican tablas o valores de filas y columnas en una tabla, y devuelve

67 Los métodos son: createStatement(), prepareStatement() y prepareCall().

el número de filas afectadas por los cambios. Y el método *execute()* puede ser empleado para ejecutar cualquier sentencia SQL.

5. *Procesamiento de ResultSet (mostrar el contenido de una consulta)*

La forma más sencilla de manejar los resultados es procesarlos una fila a la vez, con ayuda del método *next()* del objeto *ResultSet* para moverse por una tabla una fila a la vez. Dentro de una fila, *ResultSet* provee diversos métodos *getXXX()* que toman como argumento el índice o nombre de una columna, y devuelven el resultado en diversos tipos de datos de Java.

Si sólo se desea mostrar los resultados, se empleará el método *getString()* sin importar el tipo de columna real. Si se emplea con índices, éstos empiezan en 1, no en 0. Ejemplo:

```
while (rs.next()) {
   System.out.println(rs.getString(1)); //Hace referencia al primer campo de la definición de la tabla
}
```

o también

```
while (rs.next()) {
  System.out.println(“login”)); //Hace referencia a un nombre de campo concreto.
}
```

6. *Metadatos.*

El método *getMetaData()* permite obtener dinámicamente información específica sobre un objeto *ResultSet*. Se puede obtener la cantidad, nombre y tipos de las columnas en el *ResultSet*. Sin embargo, no existe un método en la clase *ResultSetMetaData* que informe del número total de filas. Ejemplo:

```
ResultSetMetaData rsmd = rs.getMetaData();
        int numCols = rsmd.getColumnCont();
        while (rs.next()) {
                for (int i = 1; i<= numCols; i++) {
                        System.out.print(rs.getString(1)+”\t”);
                        System.out.println();
                }
        }
```

7. *Cerrar la conexión.*

Si se cierra sobre el objeto *Statement* se liberan los recursos de la conexión. No se suele requerir, pues *Statement* se cierra automáticamente cuando se cierra el objeto *Connection* asociado. Si se cierra el objeto *Connection*, se liberan todos los recursos (como conexiones de red y bloqueos de bases de datos) asociados con la conexión. Se llama automáticamente a este método cuando se procede a la recolección de basura, sin embargo, es mejor cerrar manualmente una conexión una vez que se ha acabado con ella, pues puede dejar recursos abiertos. Este método cierra implícitamente cualquier sentencia y conjunto de resultados creados por esta conexión.

SQL Básico[68]

Cómo crear tablas mediante JDBC

Para crear una tabla se precisa emplear la sentencia SQL *CREATE TABLE*. Ejemplo:

68 Para saber mucho más:
Información básica:
http://es.wikipedia.org/wiki/SQL
Tutorial interactivo:
http://sqlzoo.net/
Tutoriales online:
http://www.w3schools.com/SQl/default.asp
http://www.learn-sql-tutorial.com/

create table datos (login varchar(8) not null primary key,
nombre varchar(25) not null,
password varchar(8) not null)

Se debe crear la cadena siguiente:

String creaTabla = "create table datos (login varchar(8) not null primary key,
nombre varchar(25) not null,
password varchar(8) not null)";

Los pasos serán:

1. Cargar el controlador.
2. Crear la conexión.
3. Crear el objeto Statement.
4. Ejecutar el método executeUpdate de la siguiente forma:

st.executeUpdate(creaTabla);

Cómo añadir datos mediante JDBC
Se emplea la sentencia SQL *INSERT INTO*. Ejemplo:

insert into datos (login, password, nombre) values ("alumno","alumno","Alumno de DFSI")

Se debe crear la siguiente cadena:

String insertaUsuario = "insert into datos (login, password, nombre) values (\"alumno\",\"alumno\",\"Alumno de DFSI\")";

Los pasos serán los mismos que en el apartado anterior.

Cómo modificar datos mediante JDBC
Se emplea la sentencia SQL *UPDATE*. Ejemplo:

update datos set nombre="Alumno genérico" where login="alumno"

El proceso es similar a los apartados anteriores.

Cómo eliminar datos mediante JDBC
Se emplea la sentencia SQL *DELETE FROM*. Ejemplo:

delete from datos where login="alumno"

Sentencias preparadas

El método *createStatement()* proporciona un objeto *Statement* que servirá como contenedor de ejecución para las instrucciones SQL. Sin embargo, este objeto requiere que se le proporcionen instrucciones SQL completamente especificadas, en otras palabras, no podría ejecutarse repetidamente una instrucción SQL cambiando en ella el valor o valores de uno o varios de sus elementos o cláusulas. Con los objetos instanciados mediante *createStatement()* se requiere *montar* la instrucción SQL cada vez que se vaya a ejecutar.

El método *createStatement()* genera el objeto *Statement* por medio de la interfaz *java.sql.Statement*. Existen otras dos interfaces que también proporcionan soporte para otros dos tipos de contenedores de

ejecución: *java.sql.PreparedStatement* y *java.sql.CallableStatement*.

PreparedStatement: Es una extensión de *java.sql.Statement* y permite que las instrucciones SQL posean parámetros, con objeto de ejecutar una sola instrucción de forma repetida con distintos valores en cada uno de sus parámetros.

CallableStatement: Extensión de la anterior y similar. Pero permite acceder a procedimientos almacenados de la base de datos. Los procedimientos almacenados son fragmentos de código asociados a la base de datos que mejoran el rendimiento y el tiempo de respuesta de los procesos asociados a ella.

La interfaz *java.sql.PreparedStatement* proporciona la construcción de sentencias SQL compiladas que pueden disponer de uno o varios parámetros de forma dinámica.

Cuando se crea un objeto *Statement*, no se precisa proporcionar la sentencia SQL, ya que se incluye en la llamada al método de ejecución, como en el caso del método *executeQuery()*. Sin embargo, con los objetos *PreparedStatement* es preciso proporcionar la sentencia SQL en el momento de crear este objeto. Ahora bien, ¿cómo se determinan los parámetros de la instrucción SQL cuyos valores se proporcionarán de manera dinámica? Veamos un ejemplo. Supongamos que deseamos diseñar una sentencia SQL que permita mostrar los registros de una tabla, y que esos registros se filtren dinámicamente mediante una condición que deberá cumplir uno de los campos. Entonces se escribirá el siguiente código:

```
1.  <!-- Directivas JSP -->
2.  <%@ page contentType="text/html;charset=UTF-8"%>
3.  <%@ page import="java.sql.*" %>
4.  <html>
5.    <head>
6.      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7.      <title>Ejemplo de consulta a BD con PreparedStatement</title>
8.    </head>
9.    <body>
10.
11.     <!-- Scriptlets JSP -->
12.     <%
13.       ResultSet rs = null;
14.       Connection conDB = null;
15.       try {
16.         //Se carga el controlador de MySQL
17.         Class.forName("com.mysql.jdbc.Driver"); //Puede lanzar ClassNotFoundException
18.         //System.setProperty("jdbc.drivers","com.mysql.jdbc.Driver");
19.         //Se establece la conexión. Puede lanzar un SQLException
20.         conDB = DriverManager.getConnection("jdbc:mysql://localhost:3306/bdtest","eclipse","eclipse");
21.         //Se prepara la sentencia SQL
22.         String consulta = "SELECT * FROM usuarios WHERE login like ?;";
23.         //Se crea el contenedor de ejecución de sentencias
24.         PreparedStatement ps = conDB.prepareStatement(consulta);
25.         //Se proporciona un valor al parámetro que almacena el nombre de la tabla
26.         String cond[] = {"ed%","al%","%vita%","%do"};
27.         //Se asigna el parámetro a la sentencia SQL mediante ciclo for
28.         for (int i=0;i<cond.length;i++) {
29.           ps.setString(1,cond[i]);
30.           //Se obtiene el resultado
31.           rs = ps.executeQuery();
32.     %>
33.         <center>
34.         <DIV>
35.           <table cellspacing="0" cellpadding="0" border="2" width="250">
36.             <tr><td>login de usuarios del sistema</td></tr>
37.     <%
38.               try {
39.                 while (rs.next()) {
40.     %>
41.                   <tr><td ><%= rs.getString("login")%></td></tr>
42.     <%
```

```
43.                 }
44.               }
45.               catch (SQLException sqle) {
46.                  System.out.println("Error en procesamiento de datos.\n"+sqle.toString());
47.               }
48.     %>
49.           </table>
50.         </DIV></center>
51.     <%
52.        }
53.        conDB.close();
54.      }
55.      catch (ClassNotFoundException cnfe) {
56.        System.out.println("Error: Clase no encontrada."+cnfe.toString());
57.      }
58.      catch (SQLException sqle) {
59.        System.out.println("Error: SQLException."+sqle.toString());
60.      }
61.      catch (Exception e) {
62.        System.out.println("Error durante el proceso.\n"+e.toString());
63.      }
64.     %>
65.   </body>
66. </html>
```

Comentemos cada uno de las instrucciones más importantes de ese código:

```
String consulta = "SELECT * FROM usuarios WHERE login like ?;";
```

Se construye una cadena (*consulta*) que contendrá la sentencia SQL que se pretende ejecutar, indicando con el símbolo ***?*** aquellos parámetros que recibirán los valores de forma dinámica. En el caso concreto que nos ocupa, será el valor de la condición *like* la que recibe el *?*. Pretendemos por tanto asignar de forma dinámica la condición que deben cumplir en el campo *login* los registros que se muestren de la tabla *dregistro*.

Estas interrogaciones reciben el nombre de *marcadores de posición* y no pueden utilizarse para ocupar el sitio del nombre de una tabla o del nombre de una columna (campo). Es decir, no pueden emplearse expresiones como:

select ? from ? ;

```
PreparedStatement ps = conDB.prepareStatement(consulta);
```

Una vez construida la cadena se proporciona como argumento al método *prepareStatement()* para obtener un objeto *PreparedStatement*.

```
String cond[] = {"ed%","al%","%vita%","%do"};
```

En este caso concreto se ha optado por crear una tabla con una lista de cadenas que representan las condiciones de filtrado que se aplicarán a la sentencia SQL. Pueden emplearse otros mecanismos, como por ejemplo, proporcionarlos en un archivo de texto, o en un parámetro recibido por medio de la petición (*request*).

```
for (int i=0;i<cond.length;i++) {
   ps.setString(1,cond[i]);
   rs = ps.executeQuery();
```

En un ciclo for se procede a asignar dinámicamente el valor de la condición al primer parámetro de la

sentencia SQL. Se emplea el método *setString()* que *ubicará* el valor del parámetro en la instrucción preparada. Para cada tipo de dato Java hay un método *setXXX()*. En nuestro caso, la condición es una cadena de caracteres, por lo que se recurre a *setString()*. Estos métodos requieren también un índice, que servirá para identificar el parámetro dentro de la sentencia. Como una sentencia SQL preparada puede disponer de más de un parámetro, cada uno de ellos se referencia por un índice, que empieza por 1, y que se asigna de izquierda a derecha dentro de la sentencia.

Por ejemplo, la siguiente sentencia SQL preparada posee tres parámetros:

INSERT INTO datos (login, password, nombre) VALUES (?,?,?);

El primer marcador de posición posee índice 1, el siguiente 2 y el último el 3.

Recorrer en ambos sentidos un ResultSet

Como se indicó en apartado anterior, los objetos *ResultSet* emplean un cursor que apunta al registro (fila) actual y se desplaza hacia abajo mediante el método *next()*. Al crearse el objeto, el puntero apunta a una posición inmediatamente anterior a la primera fila. Las filas se pueden ir recuperando secuencialmente de arriba hacia abajo mediante sucesivas llamadas al método *next()*.

Por defecto esta forma de recorrer el *ResultSet* era la única manera posible de hacerlo con los controladores diseñados para ejecutarse junto a la API JDBC 1.0. Con la API JDBC 2.0 ya se permite el desplazamiento en ambos sentidos. Para ello se dispondrán de los siguientes métodos:

- *previous()*
- *first()*
- *last()*
- *absolute()*
- *relative()*
- *afterLast()*
- *beforeFirst()*

El siguiente código recorre el *ResultSet* desde la última fila a la primera:

```
1.  <DIV align="center">
2.      <table cellspacing="0" cellpadding="0" border="2" width="250" align=center>
3.        <tr>
4.          <td>
5.          <td>Usuarios del sistema</td>
6.          <td>
7.        </tr>
8.        <tr>
9.          <td>
10.           <STRONG>login</STRONG>
11.         </td>
12.         <td>
13.           <STRONG>clave</STRONG>
14.         </td>
15.       </tr>
16.       <%
17.          rs.afterLast();
18.          try {
19.            while (rs.previous()) {
20.       %>
21.       <tr>
22.         <td ><%= rs.getString("login")%></td>
23.         <td ><%= rs.getString("clave")%></td>
24.       </tr>
25.       <%
26.           }
27.          conDB.close();
```

```
28.           }
29.           catch (SQLException sqle) {
30.             System.out.println("Error en procesamiento de datos.\n"+sqle.toString());
31.           }
32.       %>
33.     </table>
34.   </DIV>
```

Los métodos *previous()* y *next()* devuelven *null* cuando se alcanza más allá de la primera o de la última fila, respectivamente.

Acceder a los MetaDatos

Consideramos los metadatos como la información relativa a la estructura de los elementos que forman una base de datos: nombre de las tablas, nombre y tipo de los campos, condiciones especiales asociados a ellos, etc. Las herramientas gráficas como *MySQL Admin* o *phpMyAdmin* proporcionan acceso a esos metadatos, modificándolos cada vez que se altera algún elemento de esa estructura. Algunos de estos metadatos están almacenados en tablas específicas en los gestores de bases de datos[69]. Podemos entender entonces que los metadatos son datos sobre la base de datos. También se conoce como *diccionario de datos*. Las clases relacionadas con los metadatos son:

- *DatabaseMetaData*
- *ResultSetMetaData*
- *ParameterMetaData*

DatabaseMetaData

Los objetos de esta clase proporcionan información de la base de datos con la que se ha conectado la aplicación. Para obtener este objeto se ejecuta la siguiente instrucción:

DatabaseMetaData dbmd = con.getMetaData();

El objeto creado proporciona un conjunto de métodos que permitirán obtener información tal como:

- Esquemas, información de tablas y columnas.
- Parámetros de configuración de la base de datos.
- Privilegios de acceso. Usuarios reconocidos.

El siguiente código muestra información sobre los metadatos disponibles en una base de datos MySQL.

```
1.  <!-- Directivas JSP -->
2.  <%@ page contentType="text/html;charset=UTF-8"%>
3.  <%@ page import="java.sql.*" %>
4.  <html>
5.    <head>
6.      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7.      <title>Ejemplo de consulta a BD</title>
8.    </head>
9.    <body bgcolor="Gray">
10.
11.     <!-- Scriptlets JSP -->
12.     <%
13.       Connection conDB = null;
14.       String basedatos = "bdfotogramas";
15.       try {
16.         //Se carga el controlador de MySQL
17.         Class.forName("com.mysql.jdbc.Driver"); //Puede lanzar ClassNotFoundException
18.         //System.setProperty("jdbc.drivers","com.mysql.jdbc.Driver");
19.         //Se establece la conexión. Puede lanzar un SQLException
```

69 En MySQL la base de datos *mysql* contiene información de los usuarios reconocidos para el gestor y sus correspondientes privilegios.

```
20.       conDB = DriverManager.getConnection("jdbc:mysql://localhost:3306/"+basedatos,"eclipse","eclipse");
21.       DatabaseMetaData dbmd = conDB.getMetaData();
22.     %>
23.     <table bgcolor="Olive" cellspacing="0" cellpadding="0" border="1" width="550" align="center">
24.       <tr>
25.         <td colspan="2">
26.           <DIV align="center">
27.             <H5>Metadatos sobre la base de datos
28.               <FONT color="#ff0033">
29.                 <%=   basedatos%>
30.               </FONT>
31.             </H5>
32.             </DIV>
33.         </td>
34.       </tr>
35.       <tr>
36.         <td width="193">
37.           <DIV align="right">
38.             <H5>Nombre del sistema gestor:</H5>
39.           </DIV>
40.         </td>
41.         <td width="89">
42.           <H5>
43.             <%=  dbmd.getDatabaseProductName() %>
44.           </H5>
45.         </td>
46.       </tr>
47.       <tr>
48.         <td width="193">
49.           <DIV align="right">
50.             <H5>Versi&oacute;n del sistema gestor:</H5>
51.           </DIV>
52.         </td>
53.         <td width="89">
54.           <H5>
55.             <%=  dbmd.getDatabaseProductVersion()%>
56.           </H5>
57.         </td>
58.       </tr>
59.       <tr>
60.         <td width="193">
61.           <DIV align="right">
62.             <H5>Nombre del controlador:</H5>
63.           </DIV>
64.         </td>
65.         <td width="89">
66.           <H5>
67.             <%=  dbmd.getDriverName()%>
68.           </H5>
69.         </td>
70.       </tr>
71.       <tr>
72.         <td width="193">
73.           <DIV align="right">
74.             <H5>Versi&oacute;n del controlador:</H5>
75.           </DIV>
76.         </td>
77.         <td width="89">
78.           <H5>
79.             <%=  dbmd.getDriverVersion()%>
80.           </H5>
81.         </td>
82.       </tr>
83.       <tr>
84.         <td width="193">
85.           <DIV align="right">
86.             <H5>Soporte del ANSI92 completo:</H5>
87.           </DIV>
88.         </td>
```

```
89.       <td width="89">
90.         <H5>
91.           <%=  dbmd.supportsANSI92FullSQL()%>
92.         </H5>
93.       </td>
94.     </tr>
95.     <tr>
96.       <td width="193">
97.         <DIV align="right">
98.           <H5>Versi&oacute;n menor de JDBC que soporta el controlador:</H5>
99.         </DIV>
100.      </td>
101.      <td width="89">
102.        <H5>
103.          <%=  dbmd.getDriverMinorVersion()%>
104.        </H5>
105.      </td>
106.    </tr>
107.    <tr>
108.      <td width="193">
109.        <DIV align="right">
110.          <H5>Versi&oacute;n mayor de JDBC que soporta el controlador:</H5>
111.        </DIV>
112.      </td>
113.      <td width="89">
114.        <H5>
115.          <%=  dbmd.getDriverMajorVersion()%>
116.        </H5>
117.      </td>
118.    </tr>
119.    <tr>
120.      <td width="193">
121.        <DIV align="right">
122.          <H5>M&aacute;ximo n&uacute;mero de conexiones concurrentes:</H5>
123.        </DIV>
124.      </td>
125.      <H5>
126.        <%  if (dbmd.getMaxConnections()==0)
127.          {
128.      %>
129.      </H5>
130.            <td width="89">
131.              <H5>sin l&iacute;mite o desconocido</H5>
132.            </td>
133.            <H5>
134.              <%
135.          }
136.          else
137.          {
138.      %>
139.            </H5>
140.              <td>
141.                <H5>
142.                  <%=  dbmd.getMaxConnections()%>
143.                </H5>
144.              </td>
145.              <H5>
146.                <%
147.          }
148.      %>
149.            </H5>
150.    </tr>
151.    <tr>
152.      <td width="193">
153.        <DIV align="right">
154.          <H5>Tama&ntilde;o m&aacute;ximo de nombre de tabla:</H5>
155.        </DIV>
156.      </td>
157.      <td width="89">
```

```
158.          <H5>
159.            <%=  dbmd.getMaxTableNameLength()%>
160.          </H5>
161.        </td>
162.      </tr>
163.    </table>
164.    <%
165.      }
166.      catch (ClassNotFoundException cnfe) {
167.        System.out.println("Error: Clase de controlador no encontrada."+cnfe.toString());
168.      }
169.      catch (SQLException sqle) {
170.        System.out.println("Error: excepción SQL."+sqle.toString());
171.      }
172.    %>
173.  </body>
174.</html>
```

Y el siguiente código muestra la lista de tablas de una base de datos :

```
1.   <!-- Directivas JSP -->
2.   <%@ page contentType="text/html;charset=UTF-8"%>
3.   <%@ page import="java.sql.*, java.util.*" %>
4.   <html>
5.     <head>
6.       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7.       <title>Cómo obtener los nombres de las tablas de una BD</title>
8.     </head>
9.     <body bgcolor="Gray">
10.
11.      <%  /* Declaraciones */
12.         Connection conDB = null;
13.         ResultSet rs = null;
14.         DatabaseMetaData dbmd = null;
15.         Vector listaTablas = new Vector();
16.         String basedatos = "mysql";
17.         String[] tipoTabla = {"TABLE"};
18.      %>
19.      <!-- Scriptlets JSP -->
20.      <%
21.        try {
22.         String nombreTabla = null;
23.         //Se carga el controlador de MySQL
24.         Class.forName("com.mysql.jdbc.Driver"); //Puede lanzar ClassNotFoundException
25.         //System.setProperty("jdbc.drivers","com.mysql.jdbc.Driver");
26.         //Se establece la conexión
27.             conDB = DriverManager.getConnection("jdbc:mysql://localhost:3306/"+basedatos,"root","root"); //Puede lanzar un
SQLException
28.         dbmd = conDB.getMetaData();
29.          rs = dbmd.getTables(null,null,null,tipoTabla);
30.         while (rs.next()) {
31.           nombreTabla = rs.getString("TABLE_NAME");
32.          listaTablas.add(nombreTabla);
33.         }
34.      %>
35.      <table bgcolor="Olive" cellspacing="0" cellpadding="0" border="1" width="550" align="center">
36.        <tr>
37.         <td colspan="2">
38.          <DIV align="center">
39.           <H5>Nombres de tabla de
40.            <FONT color="#ff0033">
41.             <%=   basedatos%>
42.            </FONT>
43.           </H5>
44.           </DIV>
45.      <%
46.         for (int i = 0; i<listaTablas.size(); i++) {
```

```
47.    %>
48.          <tr><td align="center"><%= listaTablas.elementAt(i) %> </td>
49.    <%
50.       }
51.    %>
52.    </table>
53.    <%
54.     }
55.     catch (ClassNotFoundException cnfe) {
56.       System.out.println("Error: Clase de controlador no encontrada."+cnfe.toString());
57.     }
58.     catch (SQLException sqle) {
59.       System.out.println("Error: excepción SQL."+sqle.toString());
60.     }
61.    %>
62.  </body>
63. </html>
```

En este código el método *getTables()* permite obtener un objeto *ResultSet* que contiene los nombres de las tablas de la base de datos, además, en formato de tabla, razón por la cual se emplea esta clase. Para ello se deben proporcionar unos parámetros a este método. Para el caso de MySQL, los dos primeros parámetros pueden permanecer a *null*, el tercero representa a un nombre de tabla con el que deben coincidir las tablas que se van a mostrar (poner *null* significa mostrar todas las tablas) y el cuarto el tipo de tabla. Es una matriz que puede contener una o más de las siguientes cadenas: *TABLE*, *VIEW*, *SYSTEM TABLE*, *GLOBAL TEMPORARY*, *LOCAL TEMPORARY*, *ALIAS*, *SYNONYM*.

Finalmente, he aquí la forma de obtener el código SQL que permite crear la tabla referenciada:

1. Obtener los nombres de todas las tablas de la base de datos, tal como se mostró en el anterior código.
2. Para cada tabla, obtener el nombre y tipo de cada columna. Si el tipo de la columna tiene limitaciones de tamaño, obtener dicha información. Si esa columna no admite valores nulos, también se necesitará esa información.
3. Determinar las claves principales de cada tabla.
4. Emplear la información de los puntos 1 a 3 para generar una sentencia SQL *CREATE TABLE*.

Los métodos necesarios serán *getTables()*, *getColumns()* y *getPrimaryKeys()*.

ResultSetMetaData

Los objetos *ResultSet* también tienen asociados metadatos. Por ejemplo, los nombres y tipos de las columnas que los forman. El siguiente código muestra cómo obtenerlos:

```
1.  <!-- Directivas JSP -->
2.  <%@ page contentType="text/html;charset=UTF-8"%>
3.  <%@ page import="java.sql.*, java.util.*" %>
4.  <html>
5.    <head>
6.      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7.      <title>Acceso a los MetaDatos de un ResultSet</title>
8.    </head>
9.    <body>
10.
11.    <% /* Declaraciones */
12.       Connection conDB = null;
13.       ResultSet rs = null;
14.       ResultSetMetaData rsmd = null;
15.       Statement st = null;
16.       String basedatos = "bdfotogramas";
17.       String tabla = "usuarios";
18.       int numColumnas;
19.       String nombreColumna;
```

```
20.         String nombreTabla;
21.         String nombreEsquema;
22.         String nombreClase;
23.         int tipo; //Tipo definido en java.sql.Types
24.         String nombreTipo;
25.      %>
26.      <!-- Scriptlets JSP -->
27.      <%
28.       int i;
29.       try {
30.         //Se carga el controlador de MySQL
31.         Class.forName("com.mysql.jdbc.Driver"); //Puede lanzar ClassNotFoundException
32.         //Se establece la conexión
33.            conDB = DriverManager.getConnection("jdbc:mysql://localhost:3306/"+basedatos,"eclipse","eclipse"); //Puede lanzar
un SQLException
34.         st = conDB.createStatement();
35.         rs = st.executeQuery("select * from "+tabla);
36.          rsmd = rs.getMetaData();
37.      %>
38.      <table bgcolor="Olive" cellspacing="0" cellpadding="0" border="1" width="550" align="center">
39.        <tr>
40.          <td colspan=6>
41.          <DIV align="center">
42.            <H5>Algunos metadatos de la tabla
43.              <FONT color="#ff0033">
44.                <%= tabla%>
45.              </FONT>
46.            </H5>
47.            </DIV>
48.          <tr>
49.            <td align="center">Columna:
50.            <td align="center">Nombre:
51.            <td align="center">Nombre tabla:
52.            <td align="center">Nombre clase:
53.            <td align="center">Tipo columna:
54.            <td align="center">Nombre del tipo:
55.      <%
56.        numColumnas = rsmd.getColumnCount();
57.        for (i = 1; i <= numColumnas; i++) {
58.          nombreColumna= rsmd.getColumnName(i);
59.          nombreTabla= rsmd.getTableName(i);
60.          nombreClase= rsmd.getColumnClassName(i);
61.          tipo= rsmd.getColumnType(i);
62.          nombreTipo= rsmd.getColumnTypeName(i);
63.      %>
64.            <tr>
65.              <td align="center"><%= i %>
66.              <td align="center"><%= nombreColumna%>
67.              <td align="center"><%= nombreTabla%>
68.              <td align="center"><%= nombreClase%>
69.              <td align="center"><%= tipo%>
70.              <td align="center"><%= nombreTipo%>
71.      <%
72.         }
73.      %>
74.      </table>
75.      <%
76.       }
77.       catch (ClassNotFoundException cnfe) {
78.         System.out.println("Error: Clase de controlador no encontrada."+cnfe.toString());
79.       }
80.       catch (SQLException sqle) {
81.         System.out.println("Error: excepción SQL."+sqle.toString());
82.       }
83.      %>
84.    </body>
85. </html>
```

ParameterMetaData
Permite obtener información de los parámetros de una sentencia SQL preparada. Para más información, consultar la API. Está disponible desde la versión 1.4 del J2SE.

Configuración de un DataSource

¿Qué es un DataSource?
Un *DataSource* es una interfaz que se emplea para acceder de forma transparente a una fuente de datos. Esta interfaz se encuentra en *javax.sql.DataSource*.

Esta interfaz posibilita abstraer el acceso a una fuente de datos, y por tanto, nos evita tener que emplear *DriverManager* para la carga del controlador de la fuente correspondiente.

Ventajas:

1. Utilizaremos DataSources para acceder a fuentes de datos heterogéneas, por ejemplo a una base de datos relacional, a una base de datos orientada a objetos, a un conector para acceso a un sistema legacy, a un directorio de LDAP, etc.
2. Típicamente los DataSource nos ocultan aspectos de bajo nivel relativos a la gestión de las conexiones. Así, cuando nosotros accedemos a un DataSource, probablemente el servidor de base de datos o el servidor de aplicaciones se encargue de gestionar por nosotros el ciclo de vida (pools, cachés, etc.) de las conexiones del sistema al que queremos conectarnos, obteniendo de este modo un aumento de rendimiento considerable
3. La interfaz actúa como fachada, es decir, si por ejemplo queremos cambiar de base de datos, sólo tendremos que modificar la configuración del DataSource.
4. En lugar de cargar explícitamente las clases del gestor de drivers en el periodo de ejecución de la aplicación cliente, se emplea una búsqueda centralizada de servicio JNDI[70] para obtener un objeto *javax.sql.DataSource*.
5. Sustituye a la clase *java.sql.DriverManager*.

Este enfoque aísla las aplicaciones cliente de la responsabilidad de inicialización del driver de base de datos. Esto significa que las aplicaciones cliente no necesitan conocer las clases de driver de base de datos y el URL de base de datos. Estas tareas pasan ahora a ser responsabilidad del administrador del servidor de aplicaciones que configura las fuentes de datos. Esta separación también permite activar servicios adicionales como reserva de conexiones y apoyo a transacciones sin afectar a aplicaciones adicionales.

¿Qué es un pool (o reserva) de conexiones?
Las conexiones a una fuente de datos suele ser un recurso caro. Normalmente existe un número limitado de conexiones que puede mantener abiertas un determinado servidor, dependiendo de su memoria, su potencia, del sistema al que estamos accediendo, etc. Muchos sistemas que actúan como servidores necesitan mantener una gran escalabilidad y por lo tanto no pueden arriesgarse a que se agoten las conexiones. ¿Qué se puede hacer? Un pool de conexiones soluciona esta técnica. Se trata de un conjunto limitado de conexiones que se reutilizan constantemente para dar servicio a los diferentes clientes. De este modo se aumenta la escalabilidad y también el rendimiento, ya que no sería necesario abrir las conexiones constantemente. Normalmente un pool de conexiones se inicializa con un número de conexiones determinado. Casi todos los servidores modernos traen normalmente pools de conexiones integrados, aunque crearse uno propio no es demasiado difícil. En un pool de conexiones cada una de éstas es utilizada por múltiples clientes diferentes. Los clientes abren la conexión, acceden al servicio a través de ella y por último cierran dicha conexión. Es importante que el cliente abra y cierre la conexión en cada acceso al servicio ya que si no la cerrase no la estaría devolviendo al pool y por lo tanto correríamos el riesgo de agotar los recursos del servidor. Un driver de base de datos puede ofrecer soporte de pools de conexiones si implementa la interfaz *javax.sql.DataSource*. Este soporte es optativo

70 Java Naming and Directory Interface.

en JDBC 2.0 y obligatorio en JDBC 3.0 y siguientes.

Pues bien, el uso de DataSources en combinación con el mecanismo de reserva de conexiones posibilita acceder a las fuentes de datos desde aplicaciones cliente sobre servidores de aplicaciones como J2EE, de una forma sencilla y portable.

JNDI y fuentes de datos

Una fuente de datos es considerada un recurso de red, recuperado de un servicio JNDI. En un servicio JNDI, las aplicaciones pueden asociar objetos a nombres. Otras aplicaciones pueden recuperar estos objetos utilizando estos nombres. Ambas aplicaciones que asocian objetos a nombres en el servicio JNDI y las aplicaciones que buscan estos nombres en el servicio JNDI pueden ser remotas. El API JDBC permite a los vendedores de servidores de aplicaciones y a los vendedores de drivers construir recursos de base de datos basados en este enfoque.

Características clave del uso de fuentes de datos

1. No es necesario que cada aplicación cliente inicialice los drivers JDBC. En cambio será preciso que el servidor de aplicaciones convierta en disponibles los objetos fuente de datos en el servicio JNDI.
2. La aplicación cliente no tiene por qué conocer los detalles del driver (la clase del driver no se incorpora en su estructura de archivo de aplicación web). La única información requerida es un nombre lógico. Esto hace que el código de aplicación sea independiente de los drivers y de las URL de JDBC.
3. Puesto que este enfoque utiliza un servicio JNDI para localizar los objetos de fuente de datos utilizando nombres lógicos, este enfoque proporciona una búsqueda independiente de la posición de las fuentes de datos. Los objetos DataSource pueden ser creados, desplegados y gestionados en un servicio JNDI independiente de todos los clientes de aplicación.