4. Acceso a bases de datos con JDBC

JDBC nos permitirá acceder a bases de datos desde Java. Para ello necesitaremos contar con un SGBD (sistema gestor de bases de datos) además de un driver específico para poder acceder a este SGBD. La ventaja de JDBC es que nos permitirá acceder a cualquier tipo de base de datos, siempre que contemos con un driver apropiado para ella.

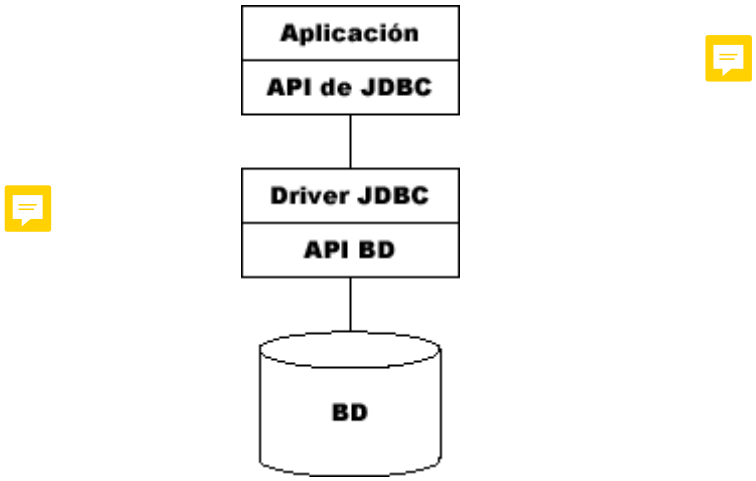

Figura 1. Arquitectura de JDBC

Como podemos ver en la Figura 1, cuando construimos una aplicación Java utilizando JDBC para el acceso a nuestra base de datos, en nuestra aplicación siempre utilizaremos la API estándar de JDBC, y la implementación concreta de la base de datos será transparente para nosotros.

Esto es importante ya que distintos SGBD tienen interfaces distintas (distintas APIs), por lo que si en algún momento quisiesemos migrar una aplicación que utiliza una de estas APIs a otro tipo de SGBD nos veríamos obligados a adaptar todo el código fuente de la aplicación a la nueva API, lo cual hace que la aplicación sea menos mantenible.

JDBC será quien se encargue por nosotros de tratar de forma interna las diferencias entre las APIs de distintos SGBDs, de forma que nosotros siempre utilizaremos la API de JDBC dentro de nuestra aplicación, y si cambiamos de SGBD no será necesario realizar ningún cambio.

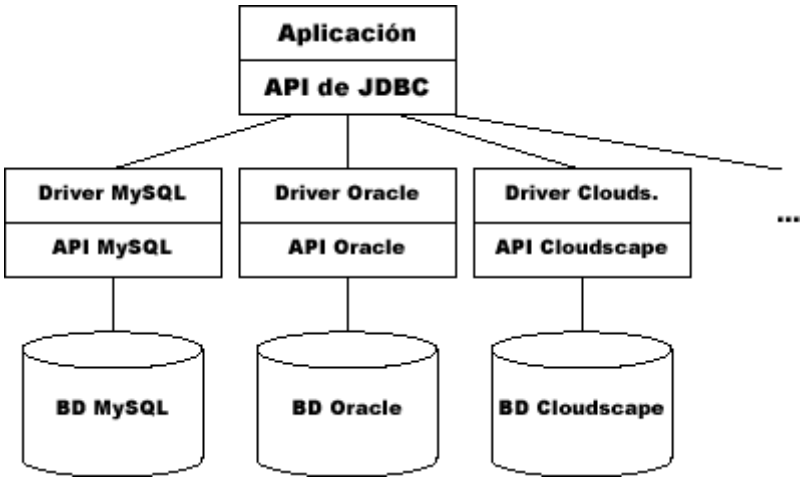

Figura 2. Conexión de JDBC a distintos SGBDs

El único cambio que tendremos que hacer será cargar al comienzo el driver apropiado para la BD que estemos utilizando. El resto de nuestro código permanecerá inalterado sea cual sea la BD a la que nos conectemos.

4.1. Instalación de drivers

Los drivers para poder acceder a cada SGBD no forman parte de la distribución de Java por lo que deberemos obtenerlos por separado. El único driver incluido es el puente JDBC-ODBC que nos permitirá acceder a cualquier fuente de datos ODBC instalada.

Si queremos trabajar directamente con un SGBD será necesario obtener el driver correspondiente. Vamos a utilizar MySQL, por lo que deberemos descargar e instalar el SGBD y el driver, que puede ser obtenido en la dirección http://www.mysql.com/downloads/api-jdbc.html.

Para instalar el driver lo único que deberemos hacer es incluir el fichero JAR que lo contiene en el CLASSPATH:

```
set CLASSPATH=%CLASSPATH%;c:\mmsql\mm.mysql-2.0.4-bin.jar
```

Con el driver instalado, podremos cargarlo desde nuestra aplicación simplemente cargando dinámicamente la clase correspondiente al driver:

```
Class.forName("org.gjt.mm.mysql.Driver");
```

Si quisiesemos cargar el driver JDBC-ODBC deberemos hacerlo como se muestra a continuación:

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

La carga del driver se deberá hacer siempre antes de conectar con la BD.

4.2. Conexión a la BD

Una vez cargado el driver apropiado para nuestro SGBD deberemos establecer la conexión con la BD. Para ello utilizaremos el siguiente método:

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

La conexión a la BD está encapsulada en un objeto **Connection**, y como hemos visto para su creación debemos proporcionar la *url* de la BD y el *login* y *password* para acceder a ella. El formato de la *url* variará según el driver que utilicemos.

En el caso de MySQL, si queremos conectarnos a una BD de nombre *paj* alojada en la máquina local (*localhost*), lo haremos de la siguiente forma:

```
Connection con = DriverManager.getConnection(
                 "jdbc:mysql://localhost/paj", null, null);
```

Para conectar a una fuente ODBC de nombre *paj* por ejemplo, utilizaremos la siguiente URL:

```
Connection con = DriverManager.getConnection(
                 "jdbc:odbc:paj", null, null);
```

En este caso no hemos especificado *login* ni *password* por lo que sólo se nos permitirá acceder si la BD no está protegida con contraseña.

4.3. Creación y ejecución de sentencias SQL

Una vez obtenida la conexión a la BD, podemos utilizarla para crear sentencias. Estas sentencias están encapsuladas en la clase **Statement**, y podemos crearlas de la siguiente forma:

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

Una vez obtenido este objeto podemos ejecutar sentencias utilizando su método **executeUpdate()** al que proporcionaremos una cadena con la sentencia SQL que queramos ejecutar:

```
stmt.executeUpdate(sentenciaSQL);
```

Estas sentencias pueden utilizarse para creación de tablas, y para la inserción, actualización y borrado de datos en ellas. Para ello utilizaremos las correspondientes sentencias SQL.

Vamos a ver a continuación un ejemplo de estas operaciones. Crearemos una tabla ALUMNOS en nuestra base de datos y añadiremos datos a la misma. La sentencia para la creación de la tabla será la siguiente:

```
String st_crea = "CREATE TABLE ALUMNOS (
                        exp INTEGER,
                        nombre VARCHAR(32),
                        nota VARCHAR(3) default 'NP',
                        PRIMARY KEY (exp)
                )";

stmt.executeUpdate(st_crea);
```

Una vez creada la tabla podremos insertar datos en ella como se muestra a continuación:

```
String st_inserta = "INSERT INTO ALUMNOS(exp, nombre)
                        VALUES(1285, 'Manu')";

stmt.executeUpdate(st_inserta);
```

Cuando tengamos datos dentro de la tabla, podremos modificarlos utilizando para ello una sentencia UPDATE:

```
String st_actualiza = "UPDATE FROM ALUMNOS
                        SET nota = 'NOT'
                        WHERE exp = 1285";

stmt.executeUpdate(st_actualiza);
```

Si queremos eliminar un registro de la tabla utilizaremos una sentencia DELETE como se muestra a continuación:

```
String st_borra = "DELETE FROM ALUMNOS
                        WHERE exp = 1285";

stmt.executeUpdate(st_borra);
```

Como hemos visto mediante el método **executeUpdate()** del objeto **Statement** podremos ejecutar cualquier sentencia SQL en la base de datos a la que nos hayamos conectado, pudiendo así realizar cualquier acción que nos permita SQL de manera sencilla.

```
int n = stmt.executeUpdate(sentencia);
```

Como resultado nos devolverá un entero que podremos consultar, y que nos dirá el número de registros a los que ha afectado la operación, en caso de sentencias INSERT, UPDATE y DELETE. En el caso de una sentencia DDL (Lenguaje de definición de datos), como puede ser por ejemplo la creación de una tabla, como no afecta a los datos contenidos en la BD nos devolverá siempre 0.

4.4. Obtención de datos

Para obtener datos almacenados en la BD podemos utilizar una consulta SQL (*query*). Podemos ejecutar la consulta utilizando el objeto **Statement**, con el método **executeQuery()** al que le pasaremos una cadena con la consulta SQL. Los datos resultantes nos los devolverá como un objeto **ResultSet**.

```
ResultSet result = stmt.executeQuery(query);
```

La consulta SQL nos devolverá una tabla, que tendrá una serie de campos y un conjunto de registros, cada uno de los cuales consistirá en una tupla de valores correspondientes a los campos de la tabla.

Los campos que tenga la tabla resultante dependerán de la consulta que hagamos, de los datos que solicitemos que nos devuelva. Por ejemplo, podemos solicitar que una consulta nos devuelva los campos *expediente* y *nombre* de los alumnos.

Además en la consulta impondremos unas restricciones a los datos que queremos ver, devolviéndonos únicamente aquellos registros que cumplan dichas restricciones. Por ejemplo, podemos solicitar que nos devuelva sólo los registros de aquellos alumnos cuya nota sea suspenso.

Veamos el funcionamiento de las consultas SQL mediante un ejemplo:

```
String query = "SELECT * FROM ALUMNOS
                        WHERE nota = 'SUS'
                )";

ResultSet result = stmt.executeQuery(query);
```

En esta consulta estamos solicitando todos los registros de la tabla ALUMNOS en los que la nota sea suspenso ('SUS'), pidiendo que nos devuelva todos los campos (indicado con *) de dicha tabla. Nos devolverá una tabla como la siguiente:

exp	nombre	nota
1286	Pedro	SUS
1304	Juana	SUS
1310	Jaume	SUS

Estos datos nos los devolverá como un objeto **ResultSet**. A continuación veremos el acceso a estos valores dentro de dicho objeto.

En el objeto **ResultSet** existe un *cursor* que estará situado en el registro que podemos consultar en cada momento. Este *cursor* en un principio estará situado en una posición anterior al primer registro de la tabla. Podemos mover el cursor al siguiente registro con el método **next()** del **ResultSet**. La llamada a este método nos devolverá **true** mientras pueda pasar al siguiente registro, y **false** en el caso de que ya estuviésemos en el último registro de la tabla. Para la consulta de todos los registros obtenidos utilizaremos normalmente un bucle como el siguiente:

```
while(result.next())
{
        // Leer registro
}
```

Ahora necesitamos obtener los datos del registro que marca el *cursor*, para lo cual podremos acceder a los campos de dicho registro. Esto lo haremos utilizando los métodos **getXXXX(campo)** donde **XXXX** será el tipo de datos de Java en el que queremos que nos devuelva el valor del campo. Hemos de tener en cuenta que el tipo del campo en la tabla debe ser convertible al tipo de datos Java solicitado. Para especificar el campo que queremos leer podremos utilizar bien su nombre en forma de cadena, o bien su índice que dependerá de la ordenación de los campos que devuelve la consulta.

Los tipos principales que podemos obtener son los siguientes:

getInt	Datos enteros
getDouble	Datos reales
getBoolean	Campos booleanos (si/no)
getString	Campos de texto
getDate	Tipo fecha

Si queremos imprimir todos los datos obtenidos de nuestra tabla ALUMNOS del ejemplo podremos hacer lo siguiente:

```
int exp;
String nombre;
int nota;

while(result.next())
{
        exp = result.getInt("exp");
        nombre = result.getString("nombre");
        nota = result.getString("nota");
        System.out.println(exp + "\t" + nombre + "\t" + nota);
}
```

Igual que los métodos **getXXXX** tenemos también métodos **updateXXXX** que nos permiten actualizar el campo especificado en el registro actual. La actualización se hace sobre los datos en memoria, si queremos que este cambio se aplique a la BD deberemos llamar a **updateRow()** tras modificar los datos del registro.

Existe un registro especial al que no se puede acceder como hemos visto anteriormente, que es el registro de inserción. Este registro se utiliza para insertar nuevos registros en la tabla. Para situarnos en él deberemos llamar al método **moveToInsertRow()**. Una vez situados en él deberemos asignar los datos y una vez hecho esto llamar a **insertRow()** para que el registro se inserte en la BD. Podemos volver al registro donde nos encontrábamos antes de movernos al registro de inserción llamando a **moveToCurrentRow()**. Si queremos eliminar el registro actual de la BD podemos llamar a **deleteRow()**.

En la interfaz **Statement** podemos observar un tercer método que podemos utilizar para la ejecución de sentencias SQL. Hasta ahora hemos visto como para la ejecución de sentencias que devuelven datos (consultas) debemos usar **executeQuery()**, mientras que para las sentencias INSERT, DELETE, UPDATE e instrucciones DDL utilizamos **executeUpdate()**. Sin embargo, puede haber ocasiones en las que no conozcamos de antemano el tipo de la sentencia que vamos a utilizar (por ejemplo si la sentencia la introduce el usuario). En este caso podemos usar el método **execute()**.

```
boolean hay_result = stmt.execute(sentencia);
```

Podemos ver que el método devuelve un valor *booleano*. Este valor será *true* si la sentencia ha devuelto resultados (uno o varios objetos **ResultSet**), y false en el caso de que sólo haya devuelto el número de registros afectados. Tras haber ejecutado la sentencia con el método anterior, para obtener estos datos devueltos proporciona una serie de métodos:

```
int n = stmt.getUpdateCount();
```

El método **getUpdateCount()** nos devuelve el número de registros a los que afecta la actualización, inserción o borrado, al igual que el resultado que devolvía **executeUpdate()**.

```
ResultSet rs = stmt.getResultSet();
```

El método **getResultSet()** nos devolverá el objeto **ResultSet** que haya devuelto en el caso de ser una consulta, al igual que hacía **executeQuery()**. Sin embargo, de esta forma nos permitirá además tener múltiples objetos **ResultSet** como resultado de una llamada. Eso puede ser necesario por ejemplo en el caso de una llamada a un procedimiento, que nos puede devolver varios resultados como veremos más adelante. Para movernos al siguiente **ResultSet** utilizaremos el siguiente método:

```
boolean hay_mas_results = stmt.getMoreResults();
```

La llamada a este método nos moverá al siguiente **ResultSet** devuelto, devolviendonos *true* en el caso de que exista, y *false* en el caso de que no haya más resultados. Si existe, una vez nos hayamos movido podremos consultar el nuevo **ResultSet** llamando nuevamente al método **getResultSet()**.

4.5. Optimización de sentencias

Cuando vamos a invocar una determinada sentencia repetidas veces, puede ser conveniente dejar esa sentencia preparada para que pueda ser ejecutada de forma más eficiente. Para hacer esto utilizaremos la interfaz **PreparedStatement**, que podrá obtenerse a partir de la conexión a la BD de la siguiente forma:

```
PreparedStatement ps = con.prepareStatement("UPDATE FROM alumnos
                                SET nota = 'NP'
                                WHERE exp>1200 AND exp<1300");
```

Vemos que a este objeto, a diferencia del objeto **Statement** visto anteriormente, le proporcionamos la sentencia SQL en el momento de su creación, por lo que estará preparado y optimizado para la ejecución de dicha sentencia posteriormente.

Sin embargo, lo más común es que necesitemos hacer variaciones sobre la sentencia, ya que normalmente no será necesario ejecutar repetidas veces la misma

sentencia exactamente, sino variaciones de ella. Por ello, este objeto nos permite parametrizar la sentencia. Estableceremos las posiciones de los parámetros con el carácter '?' dentro de la cadena de la sentencia, tal como se muestra a continuación:

```
PreparedStatement ps = con.prepareStatement("UPDATE FROM alumnos
                            SET nota = 'NP'
                            WHERE exp > ? AND exp < ?");
```

En este caso tenemos dos parámetros, que será el número de expediente mínimo y el máximo del rango que queremos actualizar. Cuando ejecutemos esta sentencia, la nota de los alumnos desde expediente inferior hasta expediente superior se establecerá a NO PRESENTADO ('NP').

Para dar valor a estos parámetros utilizaremos método **setXXX()** donde **XXX** será el tipo de los datos que asignamos al parámetro, indicando el número del parámetro (que empieza desde 1) y el valor que le queremos dar. Por ejemplo, para asignar valores enteros a los parámetros de nuestro ejemplo haremos:

```
ps.setInt(1,1200);
ps.setInt(2,1300);
```

Una vez asignados los parámetros, podremos ejecutar la sentencia llamando al método **executeUpdate()** del objeto **PreparedStatement**:

```
int n = ps.executeUpdate();
```

Igual que en el caso de los objetos **Statement**, podremos utilizar cualquier otro de los métodos para la ejecución de sentencias, **executeQuery()** o **execute()**, según el tipo de sentencia que vayamos a ejecutar.

4.6. Llamadas a procedimientos

Los procedimientos (PROCEDURES) son unidades de código que contienen un conjunto de sentencias SQL. Estos pueden servirnos para tener en nuestra BD una serie de acciones comunes predefinidas.

Por ejemplo, si cuando realizamos una venta tenemos que añadir información de la venta, y reducir el stock del producto vendido, podemos definir un procedimiento que haga esto de la siguiente forma:

```
String procedure = "CREATE PROCEDURE EFECTUAR_VENTA
    (IN cod_cliente INT, IN cod_producto INT, IN cantidad INT)
    LANGUAGE SQL
    BEGIN
                INSERT
                    INTO ventas(cliente, producto, cant)
                    VALUES(cod_cliente, cod_producto, cantidad);
                UPDATE productos
                    SET stock = stock - cantidad
                    WHERE producto = cod_producto;
        END";

stmt.executeUpdate(procedure);
```

Los procedimientos se crean mediante una sentencia DDL como la del ejemplo, ejecutándola de la misma forma que cualquier sentencia DDL. En este caso tomará tres parámetros de entrada y no devuelve ningún resultado.

En muchos SGBD podemos definir procedimientos también como se muestra a continuación:

```
String procedure = "CREATE PROCEDURE VER_VENTAS_CLIENTE
        AS SELECT cliente, sum(precio) FROM ventas, productos
                WHERE ventas.producto = productos.producto
                GROUP BY cliente";

stmt.executeUpdate(procedure);
```

En este caso este procedimiento no toma parámetros de entrada pero si que producirá resultados. Podremos tener procedimientos con distinto número de parámetros de entrada, parámetros de salida, y podrán generar incluso varios **ResultSet**.

Para llamar al procedimiento necesitaremos un tipo especial de interfaz llamada **CallableStatement**. Con esta interfaz podremos invocar sentencias para la ejecución de procedimientos como las que tenemos a continuación:

```
CallableStatement cs = con.prepareCall("{call VER_VENTAS_CLIENTE}");

ResultSet rs = cs.executeQuery();
```

Podremos pasar parámetros de entrada de la misma forma que los pasabamos con la interfaz **PreparedStatement**:

```
CallableStatement cs = con.prepareCall(
        "{call EFECTUAR_VENTA[?, ?, ?]}");

cs.setInt(1, 112);
cs.setInt(2, 3380);
cs.setInt(3, 1);

cs.executeUpdate();
```

Incluso podremos indicar parámetros de salida si el procedimiento nos proporciona alguno. En el caso de que el procedimiento genere múltiples **ResultSet**, podremos obtenerlos utilizando el método **execute()** tal como vimos anteriormente.

4.7. Transacciones

Muchas veces, cuando tengamos que realizar una serie de acciones, querremos que todas se hayan realizado correctamente, o bien que no se realice ninguna de ellas, pero no que se realicen algunas y otras no.

Podemos ver esto mediante un ejemplo, en el que se va a hacer una reserva de vuelos para ir desde Alicante a Osaka. Para hacer esto tendremos que hacer transbordo en dos aeropuertos, por lo que tenemos que reservar un vuelo Alicante-Madrid, un vuelo Madrid-Amsterdam, y un vuelo Amsterdam-Osaka.

Si cualquiera de estos tres vuelos estuviese lleno y no pudiesemos reservar, no querremos reservar ninguno de los otros dos porque no nos serviría de nada. Por lo tanto sólo nos interesa que la reserva se lleve a cabo si podemos reservar los tres vuelos.

Una transacción son un conjunto de sentencias que deben ser ejecutadas como una unidad, de forma que si una de ellas no puede realizarse, no se llevará a cabo ninguna.

Pero para hacer esto encontramos un problema. Pensemos en nuestro ejemplo de la reserva de vuelos, en la que necesitaremos realizar las siguiente inserciones (reservas):

```
try {
        stmt.executeUpdate("INSERT
                INTO RESERVAS(pasajero, origen, destino)
                VALUES('Paquito', 'Alicante', 'Madrid')");
        stmt.executeUpdate("INSERT
                INTO RESERVAS(pasajero, origen, destino)
                VALUES('Paquito', 'Madrid', 'Amsterdam')");
        stmt.executeUpdate("INSERT
                INTO RESERVAS(pasajero, origen, destino)
                VALUES('Paquito', 'Amsterdam', 'Osaka')");
} catch(SQLException e) {
        // ¿Dónde ha fallado? ¿Qué hacemos ahora?
}
```

En este caso, vemos que si falla la reserva de uno de los tres vuelos obtendremos una excepción, pero en ese caso, ¿cómo podremos saber donde se ha producido el fallo y hasta que acción debemos deshacer? Con la excepción lo único que sabemos es que algo ha fallado, pero no sabremos donde ha sido, por lo que de esta forma no podremos saber hasta que acción deberemos deshacer.

Para hacer esto de una forma limpia asegurando la consistencia de los datos, utilizaremos las operaciones de *commit* y *rollback*.

Cuando realicemos cambios en la base de datos, estos cambios se harán efectivos en ella de forma persistente cuando realicemos la operación *commit*. En el modo de operación que hemos visto hasta ahora, por defecto tenemos activado el modo *auto-commit*, de forma que siempre que ejecutamos alguna sentencia se realiza *commit* automáticamente. Sin embargo, en el caso de las transacciones con múltiples sentencias, no nos interesará hacer estos cambios persistentes hasta haber comprobado que todos los cambios se pueden hacer de forma correcta. Para ello desactivaremos este modo con:

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

Al desactivar este modo, una vez hayamos hecho las modificaciones de forma correcta, deberemos hacerlas persistentes mediante la operación *commit* llamando de forma explícita a:

```
con.commit();
```

Si por el contrario hemos obtenido algún error, no querremos que esas modificaciones se lleven a cabo finalmente en la BD, por lo que podremos deshacerlas llamando a:

```
con.rollback();
```

Por lo tanto, la operación *rollback* deshará todos los cambios que hayamos realizado para los que todavía no hubiesemos hecho *commit* para hacerlos persistentes, permitiéndonos de esta forma implementar estas transacciones de forma atómica.

Nuestro ejemplo de la reserva de vuelos debería hacerse de la siguiente forma:

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

```
            stmt.executeUpdate("INSERT
                    INTO RESERVAS(pasajero, origen, destino)
                    VALUES('Paquito', 'Alicante', 'Madrid')");
            stmt.executeUpdate("INSERT
                    INTO RESERVAS(pasajero, origen, destino)
                    VALUES('Paquito', 'Madrid', 'Amsterdam')");
            stmt.executeUpdate("INSERT
                    INTO RESERVAS(pasajero, origen, destino)
                    VALUES('Paquito', 'Amsterdam', 'Osaka')");
            // Hemos podido reservar los tres vuelos correctamente
            con.commit();
    } catch(SQLException e) {
            // Alguno de los tres ha fallado, deshacemos los cambios
            con.rollback();
    }
```

4.8. JDBC y Applets

Cuando utilicemos acceso a bases de datos mediante JDBC desde un *Applet*, deberemos tener en cuenta que el *Applet* se ejecuta en la máquina del cliente, por lo que si la BD está alojada en nuestro servidor tendrá que establecer una conexión remota.

Aquí encontramos el problema de que si el *Applet* es visible desde Internet, es muy posible que el puerto en el que escucha el servidor de base de datos puede estar cortado por algún *firewall*, por lo que el acceso desde el exterior no sería posible.

El uso del puente JDBC-ODBC tampoco es recomendable en *Applets*, ya que requiere que cada cliente tenga configurada la fuente de datos ODBC adecuada en su máquina. Esto podemos controlarlo en el caso de una intranet, pero en el caso de Internet será mejor utilizar otros métodos para la conexión.