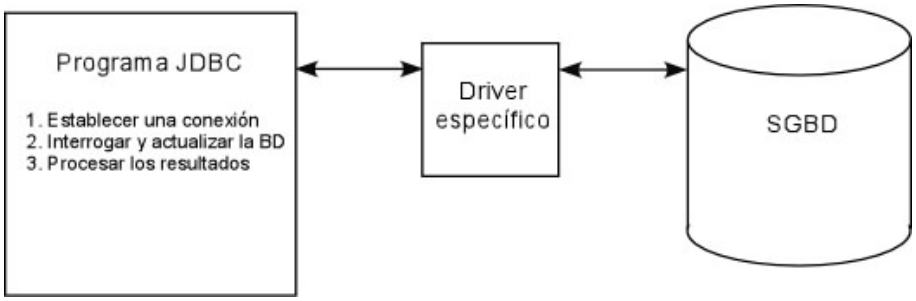
Introducción a JDBC. Consulta de una BD

Índice

1. Introducción a JDBC

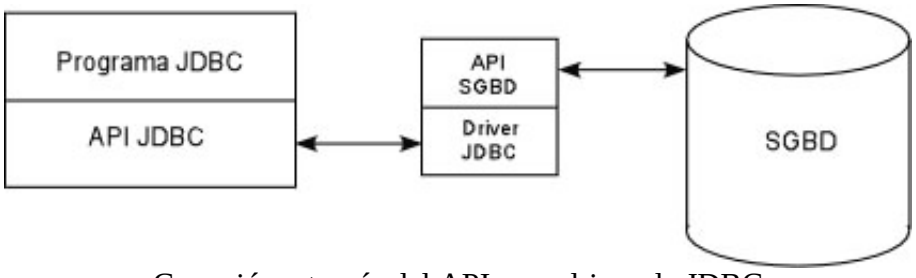
En la mayoría de las aplicaciones que nos vamos a encontrar, aparecerá una base de datos como fuente de información. JDBC nos va a permitir acceder a bases de datos (BD) desde Java. Con JDBC no es necesario escribir distintos programas para distintas BD, sino que un único programa sirve para acceder a BD de distinta naturaleza. Incluso, podemos acceder a más de una BD de distinta fuente (Oracle, Access, MySql, etc.) en la misma aplicación. Podemos pensar en JDBC como el puente entre una base de datos y nuestro programa Java. Un ejemplo sencillo puede ser un applet que muestra dinámicamente información contenida en una base de datos. El applet utilizará JDBC para obtener dichos datos.

El esquema a seguir en un programa que use JDBC es el siguiente:



Esquema general de conexión con una base de datos

Un programa Java que utilice JDBC primero deberá establecer una conexión con el SGBD. Para realizar dicha conexión haremos uso de un driver específico para cada SGBD que estemos utilizando. Una vez establecida la conexión ya podemos interrogar la BD con cualquier comando SQL (select, update, create, etc.). El resultado de un comando *select* es un objeto de la clase ResultSet, que contiene los datos que devuelve la consulta. Disponemos de métodos en *ResultSet* para manejar los datos devueltos. También podemos realizar cualquier operación en SQL (creación de tablas, gestión de usuarios, etc.).



Conexión a través del API y un driver de JDBC

Para realizar estas operaciones necesitaremos contar con un SGBD (sistema gestor de bases

de datos) además de un driver específico para poder acceder a este SGBD. Vamos a utilizar dos SGBD: MySQL (disponible para Windows y Linux, de libre distribución) y PostGres (sólo para Linux, también de libre distribución).

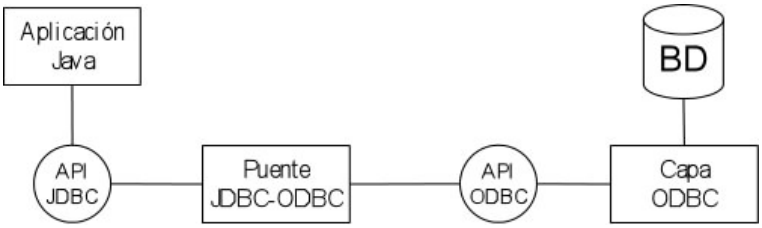
1.1. Drivers de acceso

Los drivers para poder acceder a cada SGBD no forman parte de la distribución de Java por lo que deberemos obtenerlos por separado. ¿Por qué hacer uso de un driver?. El principal problema que se nos puede plantear es que cada SGBD dispone de su propio API (la mayoría propietario), por lo que un cambio en el SGBD implica una modificación de nuestro código. Si colocamos una capa intermedia, podemos abstraer la conectividad, de tal forma que nosotros utilizamos un objeto para la conexión, y el driver se encarga de traducir la llamada al API. El driver lo suelen distribuir las propias empresas que fabrican el SGBD.

1.1.1. Tipos de drivers

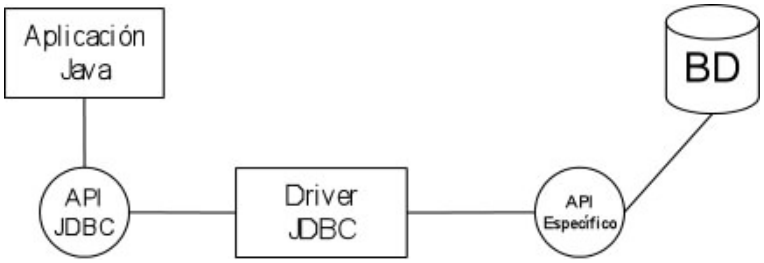
Existe un estándar establecido que divide los drivers en cuatro grupos:

- **Tipo 1: Puente JDBC-ODBC**. ODBC (Open Database Connectivity) fue creado para proporcionar una conexión a bases de datos en Microsoft Windows. ODBC permite acceso a bases de datos desde diferentes lenguajes de programación, tales como C y Cobol. El puente JDBC-ODBC permite enlazar Java con cualquier base de datos disponible en ODBC. No se aconseja el uso de este tipo de driver cuando tengamos que acceder a bases de datos de alto rendimiento, pues las funcionalidades están limitadas a las que marca ODBC. Cada cliente debe tener instalado el driver. J2SE incluye este driver en su versión Windows y Solaris.



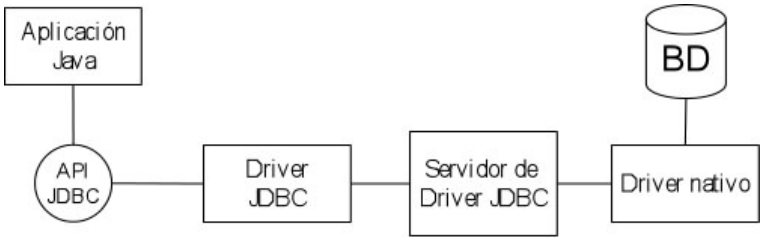
Configuración de un driver de tipo 1

- **Tipo 2: Parte Java, parte driver nativo**. Es una combinación de implementación Java y API nativo para el acceso a la base de datos. Este tipo de driver es más rápido que el anterior, pues no se realiza el paso por la capa ODBC. Las llamadas JDBC se traducen en llamadas específicas del API de la base de datos. Cada cliente debe tener instalado el driver. Tiene menor rendimiento que los dos siguientes y no se pueden usar en Internet, ya que necesita el API de forma local.



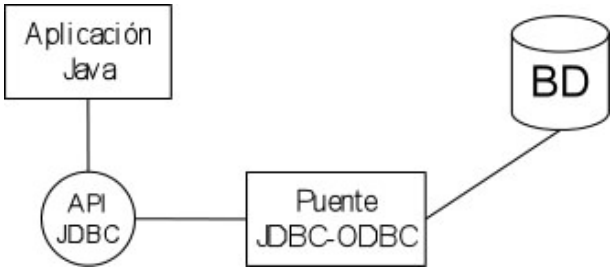
Configuración de un driver de tipo 2

- **Tipo 3: Servidor intermediario de acceso a base de datos**. Este tipo de driver proporciona una abstracción de la conexión. El cliente se conecta a los SGBD mediante un componente servidor intermedio, que actúa como una puerta para múltiples servidores. La ventaja de este tipo de driver es el nivel de abstracción. El servidor de aplicaciones WebLogic incorpora este tipo de driver.



Configuración de un driver de tipo 3

- **Tipo 4: Drivers Java**. Este es el más directo. La llamada JDBC se traduce directamente en una llamada de red a la base de datos, sin intermediarios. Proporcionan mejor rendimiento. La mayoría de SGBD proporcionan drivers de este tipo.



Configuración de un driver de tipo 4

1.1.2. Instalación de drivers

La distribución de JDBC incorpora los drivers para el puente JDBC-ODBC que nos permite acceder a cualquier BD que se gestione con ODBC. Para MySQL, deberemos descargar e instalar el SGBD y el driver, que puede ser obtenido en la dirección http://dev.mysql.com/doc/mysql/en/Java_Connector.html. El driver para PostGres se obtiene

en http://jdbc.postgresql.org

Para instalar el driver lo único que deberemos hacer es incluir el fichero JAR que lo contiene en el CLASSPATH. Por ejemplo, para MySQL:

```
export CLASSPATH=$CLASSPATH:
       /directorio-donde-este/mysql-connector-java-3.0.15-ga-bin.jar
```

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

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

El driver JDBC-ODBC se carga como se muestra a continuación:

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

Y de forma similar para PostGres:

```
Class.forName("org.postgresql.Driver");
```

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

Como hemos visto anteriormente, pueden existir distintos tipos de drivers para la misma base de datos. Por ejemplo, a una BD en MySQL podemos acceder mediante ODBC o mediante su propio driver. Podríamos pensar que la solución más sencilla sería utilizar ODBC para todos las conexiones a SGBD. Sin embargo, dependiendo de la complejidad de la aplicación a desarrollar esto nos podría dar problemas. Determinados SGBD permiten realizar operaciones (transacciones, mejora de rendimiento, escabilidad, etc.) que se ven mermadas al realizar su conexión a través del driver ODBC. Por ello es preferible hacer uso de driver específicos para el SGBD en cuestión.

El ejemplo más claro de problemas en el uso de drivers es con los *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.

En cuanto a las excepciones, debemos capturar la excepción *SQLException* en casi todas las operaciones en las que se vea involucrado algún objeto JDBC.

1.2. 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);
Connection con = DriverManager.getConnection(url, login, password);
```

La conexión a la BD está encapsulada en un objeto `Connection`. Para su creación debemos proporcionar la *url* de la BD y, si la BD está protegida con contraseña, el *login* y *password* para acceder a ella. El formato de la *url* variará según el driver que utilicemos. Sin embargo, todas las *url* tendrán la siguiente forma general: *jdbc*:<*subprotocolo*>:<*nombre*>, con *subprotocolo* indicando el tipo de SGBD y con *nombre* indicando el nombre de la BD y aportando información adicional para la conexión.

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

```
Connection con = DriverManager.getConnection("jdbc:odbc:bd");
```

En el caso de MySQL, si queremos conectarnos a una BD de nombre *bd* alojada en la máquina local (*localhost*) y con usuario *miguel* y contraseña *m++24*, lo haremos de la siguiente forma:

```
Connection con =
    DriverManager.getConnection("jdbc:mysql://localhost/bd", "miguel",
"m++24");
```

En el caso de PostGres (notar que hemos indicado un puerto de conexión, el 5432):

```
Connection con = DriverManager.getConnection(
    "jdbc:postgresql://localhost:5432/bd", "miguel", "m++24");
```

Podemos depurar la conexión y determinar qué llamadas está realizando JDBC. Para ello haremos uso de un par de métodos que incorpora **DriverManager**. En el siguiente ejemplo se indica que las operaciones que realice JDBC se mostrarán por la salida estándar:

```
DriverManager.setLogWriter(new PrintWriter(System.out, true));
```

Una vez realizada esta llamada también podemos mostrar mensajes usando:

```
DriverManager.println("Esto es un mensaje");
```

2. Consulta a una base de datos con JDBC

2.1. Creación y ejecución de sentencias SQL

Una vez obtenida la conexión a la BD, podemos utilizarla para realizar consultas, inserción y/o borrado de datos de dicha BD. Todas estas operaciones se realizarán mediante lenguaje SQL. La clase **Statement** es la que permite realizar todas estas operaciones. La instanciación de esta clase se realiza haciendo uso del siguiente método que proporciona el objeto **Connection**:

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

Podemos dividir las sentencias SQL en dos grupos: las que actualizan la BD y las que únicamente la consultan. En las siguientes secciones veremos cómo podemos realizar estas dos acciones.

2.2. Sentencias de consulta

Para obtener datos almacenados en la BD podemos realizar una consulta SQL (*query*). Podemos ejecutar la consulta utilizando el objeto **Statement**, pero ahora haciendo uso del 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 o bien que nos devuelva todos los campos de la tabla *alumnos*.

Veamos el funcionamiento de las consultas SQL mediante un ejemplo:

```
String query = "SELECT * FROM ALUMNOS WHERE sexo = 'M'";
ResultSet result = stmt.executeQuery(query);
```

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

exp	nombre	sexo
1286	Amparo	M
1287	Manuela	M

1288	Lucrecia	M

Estos datos nos los devolverá como un objeto **ResultSet**. A continuación veremos cómo podemos acceder a los valores de este objeto y cómo podemos movernos por los distintos registros.

El objeto **ResultSet** dispone de 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 estuviéramos 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. También debemos tener en cuenta que no podemos acceder al mismo campo dos veces seguidas en el mismo registro. Si lo hacemos nos dará una excepción.

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 (Devuelve **Date**)
getTime	Tipo hora (Devuelve **Time**)

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

```
int exp;
```

```
String nombre;
String sexo;

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

Cuando un campo de un registro de una tabla no tiene asignado ningún valor, la consulta de ese valor devuelve NULL. Esta situación puede dar problemas al intentar manejar ese dato. La clase **ResultSet** dispone de un método **wasNull** que llamado después de acceder a un registro nos dice si el valor devuelto fue NULL. Esto no sucede así para los datos numéricos, ya que devuelve el valor 0. Comprobemos qué sucede en el siguiente código:

```
String sexo;

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

La llamada al método **trim** devolverá una excepción si el objeto **nombre** es NULL. Por ello podemos realizar la siguiente modificación:

```
String sexo;

while(result.next()){
    exp = result.getInt("exp");
    System.out.print(exp + "\t");
    nombre = result.getString("nombre");
    if (result.wasNull()) {
        System.out.print("Sin nombre asignado");
    else
        System.out.print(nombre.trim());
    sexo = result.getString("sexo");
    System.out.println("\t" + sexo);
}
```