

# 2

## Manejo de conectores

### OBJETIVOS DEL CAPÍTULO

- ✓ Valorar las ventajas e inconvenientes de utilizar conectores.
- ✓ Establecer conexiones, modificaciones y consultas sobre una base de datos usando conectores.
- ✓ Gestionar transacciones mediante conectores.

Se llama conector al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Desde los lenguajes propios de los sistemas gestores de bases de datos se pueden gestionar los datos mediante lenguajes de consulta y manipulación propios de esos sistemas. Sin embargo, cuando se quiere acceder a los datos desde lenguajes de programación de una misma manera con independencia del sistema gestor que contenga los datos, entonces es necesario utilizar conectores que faciliten estas operaciones. Los conectores dan al programador una manera homogénea de acceder a cualquier sistema gestor (preferiblemente relacional u objeto-relacional).

En este capítulo se hace una primera introducción a los conceptos básicos que hay detrás de los conectores. Seguidamente se muestran ejemplos sobre cómo realizar las operaciones básicas sobre una base de datos MySQL usando conectores con Java.

## 2.1 EL DESFASE OBJETO-RELACIONAL

El problema del desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la programación orientada a objetos, con la que se desarrollan aplicaciones, y la base de datos, con las que se almacena la información. Estos aspectos se pueden presentar relacionados cuando:

- ✓ Se realizan actividades de programación, donde el programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- ✓ Se especifican los tipos de datos. En las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, suelen ser tipos simples, mientras que en la programación orientada a objetos se utilizan tipos de datos complejos.
- ✓ En el proceso de elaboración del software se realiza una traducción del modelo orientado a objetos al modelo entidad-relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que el desarrollador tenga que diseñar dos diagramas diferentes para el diseño de la aplicación.

La discrepancia objeto-relacional surge porque en el modelo relacional se trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, en el modelo de programación orientada a objetos se trabaja con objetos y las asociaciones entre ellos. Al problema se le denomina desfase objeto-relacional, o sea, el conjunto de dificultades técnicas que aparecen cuando una base de datos relacional se usa conjuntamente con un programa escrito con lenguajes de programación orientada a objetos.

En el Capítulo 3 se volverá a hacer hincapié sobre esta idea, destacando los problemas del desfase objeto-relacional en programación y mostrando la solución que son los sistemas gestores orientados a objetos (SGBDOO) para solventar este problema.

## 2.2 PROTOCOLOS DE ACCESO A BASES DE DATOS: CONECTORES

Muchos servidores de bases de datos utilizan protocolos de comunicación específicos que facilitan el acceso a los mismos, lo que obliga a aprender un lenguaje nuevo para trabajar con cada uno de ellos. Es posible reducir esa diversidad de protocolos mediante alguna interfaz de alto nivel que ofrezca al programador una serie de métodos para acceder a la base de datos (véase la Figura 2.1).

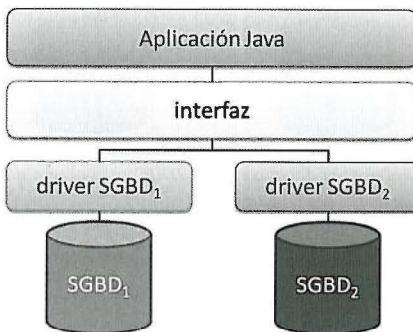


Figura 2.1. Estructura general de acceso a distintas bases de datos desde Java

Estas interfaces de alto nivel ofrecen facilidades para:

- ✓ Establecer una conexión a una base de datos.
- ✓ Ejecutar consultas sobre una base de datos.
- ✓ Procesar los resultados de las consultas realizadas.

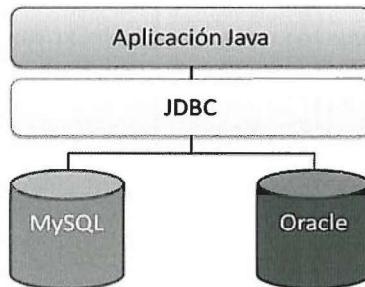
Las tecnologías disponibles, aunque muy diversas, abstraen la complejidad subyacente de cada producto y proporcionan una interfaz común, basada en el lenguaje de consulta estructurado (SQL), para el acceso homogéneo a los datos. Algunos ejemplos representativos son JDBC (*Java DataBase Connectivity*) de Sun y ODBC (*Open DataBase Connectivity*) de Microsoft.

Al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos se le denomina *conector* o *driver*. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Cuando se construye una aplicación de base de datos, el conector oculta los detalles específicos de cada base de datos, de modo que el programador solo debe preocuparse de los aspectos relacionados con su aplicación, olvidándose de otras consideraciones. La mayoría de los fabricantes ofrecen conectores para acceder a sus bases de datos.

Un ejemplo de conector muy extendido es el mencionado con anterioridad, el conector JDBC. Este conector es una capa software intermedia (véase la Figura 2.2) situada entre los programas Java y los sistemas de gestión de bases de datos relacionales que utilizan SQL. Dicha capa es independiente de la plataforma y del gestor de bases de datos utilizado.

Con el conector JDBC no hay que escribir un programa para acceder, por ejemplo, a una base de datos Access y otro programa distinto para acceder a una base de datos Oracle, etc., sino que se puede escribir un único programa utilizando el API JDBC, y es ese programa el que se encarga de enviar las consultas a la base de datos utilizada en cada caso.



*Figura 2.2. Localización de un conector JDBC en el acceso a bases de datos*

Otro ejemplo de conector es el conector de Microsoft ODBC. La diferencia entre JDBC y ODBC está en que ODBC tiene una interfaz C. En este sentido, ODBC es simplemente otra opción respecto a JDBC, ya que la mayoría de sistemas gestores de bases de datos disponen de *drivers* para trabajar con ODBC y JDBC. Además, también existe en Java un *driver* JDBC-ODBC, para convertir llamadas JDBC a ODBC y poder acceder a bases de datos que ya tienen un *driver* ODBC y todavía no tienen un conector JDBC.<sup>23</sup>

Seguidamente, y para ser coherentes con el resto de contenidos de este libro, el capítulo se centrará en el conector JDBC, sus componentes y principales características.

### 2.2.1 COMPONENTES JDBC

El conector JDBC incluye cuatro componentes principales:

- La propia API JDBC, que facilita el acceso desde el lenguaje de programación Java a bases de datos relacionales y permite que se puedan ejecutar sentencias de consulta en la base de datos. Dicha API está disponible en los paquetes *java.sql* y *javax.sql*, Java Standard Edition (Java SE) / Java Enterprise Edition (Java EE) respectivamente.
- El gestor del conector JDBC (*driver manager*), que conecta una aplicación Java con el *driver* correcto de JDBC. Se puede realizar por conexión directa (*DriverManager*) o a través de un *pool* de conexiones, vía *DataSource*.
- La *suite* de pruebas JDBC, encargada de comprobar si un conector (*driver*)<sup>24</sup> cumple con los requisitos JDBC.
- El *driver* o puente JDBC-ODBC, que permite que se puedan utilizar los *drivers* ODBC como si fueran de tipo JDBC.

<sup>23</sup> Esta solución, aunque muy versátil, es la que peor rendimiento ofrece, ya que obliga a varias conversiones entre API.

<sup>24</sup> A lo largo del capítulo, conector o *driver* se usarán como sinónimos indistintamente.

## 2.2.2 TIPOS DE CONECTORES JDBC

En función de los componentes anteriores, en un conector JDBC existen cuatro tipos de controladores JDBC. La denominación de estos controladores está asociada a un número de 1 a 4 y viene determinada por el grado de independencia respecto de la plataforma, prestaciones, etc. Seguidamente se muestran cada uno de estos tipos de conectores JDBC.

**Driver tipo 1:** utilizan una API nativa estándar, donde se traducen las llamadas de JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo (Figura 2.3).

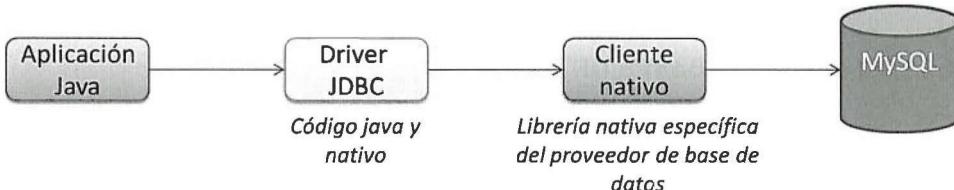


Figura 2.3. Driver tipo 1

**Driver tipo 2:** utilizan una API nativa de la base de datos, es decir son *drivers* escritos parte en Java y parte en código nativo. El *driver* usa una librería cliente nativa, específica de la base de datos con la que se desea conectar. No es un *driver* 100 % Java. La aplicación Java hace una llamada a la base de datos a través del *driver* JDBC y este traduce la petición a invocaciones a la API del fabricante de la base de datos (Figura 2.4).



Figura 2.4. Driver tipo 2

**Driver tipo 3:** utilizan un servidor remoto con una API genérica, es decir son *drivers* que usan un cliente Java puro que se comunica con un *middleware server* usando un protocolo independiente de la base de datos (por ejemplo, TCP/IP). Este tipo de *drivers* convierte las llamadas en un protocolo que puede utilizarse para interactuar con la base de datos (Figura 2.5).



Figura 2.5. Driver tipo 3

**Driver tipo 4:** es el método más eficiente de acceso a base de datos. Este tipo de *drivers* son suministrados por el fabricante de la base de datos y su finalidad es convertir llamadas JDBC en un protocolo de red comprendido por la base de datos. Este tipo de *driver* es el que se trabajará en los ejemplos incluidos en este capítulo (Figura 2.6).



Figura 2.6. Driver tipo 4

### 2.2.3 MODELOS DE ACCESO A BASES DE DATOS

A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple.

En el modelo de dos capas, la aplicación que accede a la base de datos reside en el mismo lugar que el *driver* de la base de datos. Sin embargo, la base de datos puede estar en otra máquina distinta, con lo que el cliente se comunica por red. Esta es la configuración llamada cliente-servidor, y en ella toda la comunicación a través de la red con la base de datos será manejada por el conector de forma transparente a la aplicación Java. Este modelo de acceso se muestra gráficamente en la Figura 2.7.

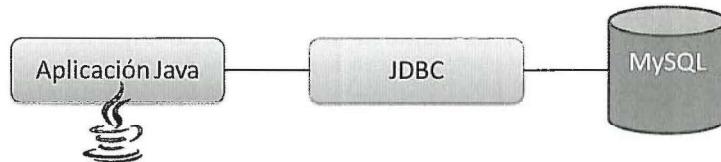


Figura 2.7. Modelo de acceso a base de datos cliente-servidor (dos capas)

Alternativamente al modelo de dos capas, en el modelo de tres capas los comandos se envían a la capa intermedia de servicios, que envía las consultas a la base de datos (Figura 2.8). Esta las procesa y envía los resultados de vuelta a la capa intermedia, para que más tarde sean enviados al cliente. En este modelo una aplicación o *applet* de Java se está ejecutando en una máquina y accediendo a un *driver* de base de datos situado en otra máquina. Ejemplos de puesta en práctica de este modelo de acceso se dan en los siguientes casos:

- Cuando se tiene un *applet* accediendo al *driver* a través de un servidor web.
- Cuando una aplicación accede a un servidor remoto que comunica localmente con el *driver*.
- Cuando una aplicación, que está en comunicación con un servidor de aplicaciones, accede a la base de datos por nosotros.

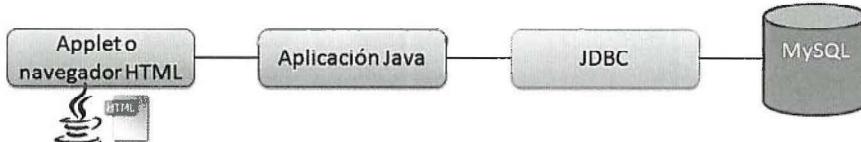


Figura 2.8. Modelo de acceso basado en la existencia de una capa intermedia de servicio (tres capas)

### 2.2.4 ACCESO A BASES DE DATOS MEDIANTE UN CONECTOR JDBC

Las dos ventajas que ofrece JDBC pasan por proveer una interfaz para acceder a distintos motores de base de datos y por definir una arquitectura estándar con la que los fabricantes puedan crear conectores que permitan a las aplicaciones Java acceder a los datos. Este apartado se centra en la primera de esas ventajas.

A continuación se muestra cómo acceder a una base de datos utilizando JDBC. Lo primero que se debe hacer para poder realizar consultas en una base de datos es, obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas ellas, así que se optará por una en concreto. La elegida es una base de datos MySQL (véase el esquema mostrado en la Figura 2.9). Se ha elegido este gestor de bases de datos porque es gratuito y por funcionar en diferentes plataformas.

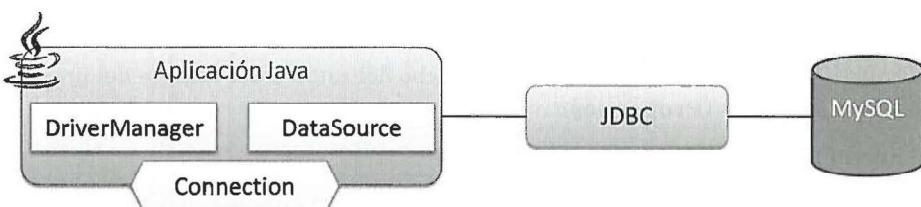


Figura 2.9. Estructura general de acceso a una base de datos MySQL desde una aplicación Java

Para acceder a MySQL con JDBC se deben seguir los pasos que se detallan a continuación (véase la Figura 2.10):

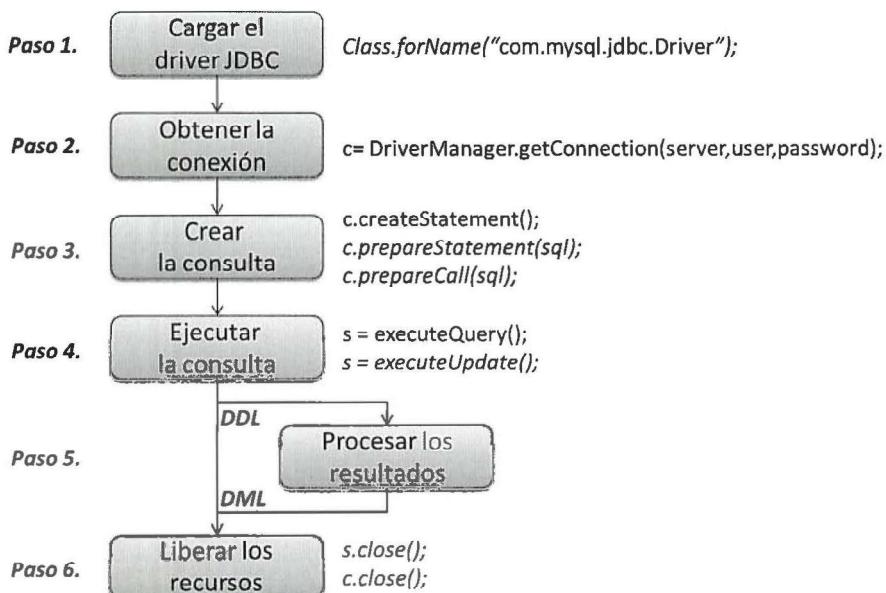


Figura 2.10. Pasos para acceder a una base de datos MySQL

Antes de cargar el *driver JDBC* y obtener una conexión con la base de datos se deben hacer dos pasos:

Lo primero que se necesita para conectar a una base de datos es un objeto *conector*. Ese *conector* es el que sabe cómo interactuar con la base de datos. El lenguaje Java no viene con todos los conectores de todas las posibles bases de datos del mercado. Por tanto, se debe recurrir a Internet para obtener el conector que se necesite en cada caso.

En los ejemplos de este capítulo, se necesitará el conector de MySQL.<sup>25</sup> Una vez descargado el fichero *mysql-connector-java-5.1.xx.zip*, se descomprime y se localiza el fichero *mysql-connector-java-5.1.21-bin.jar* (donde *xx* hace referencia a la versión más actual del mismo), que viene incluido en el fichero *.zip*. En ese otro archivo un fichero con extensión *.jar* ofrece la clase conector que nos interesa.

Para incluir el fichero *mysql-connector-java-5.1.21-bin.jar* en cada proyecto se deberán seguir los siguientes pasos:

1. En la carpeta raíz del proyecto, crear la carpeta */lib*.
2. Copiar el fichero *mysql-connector-java-5.1.21-bin.jar* en la carpeta */lib* que se acaba de crear.
3. Desde *NetBeans IDE 7.1.2*,<sup>26</sup> pulsar con el botón derecho del ratón en el nombre del proyecto y seleccionar la opción de menú propiedades (**Properties**).
4. En el árbol lateral pulsar en **Libraries**.
5. En el botón de la izquierda pulsar en **Add JAR/Folder**.
6. Seleccionar el fichero *mysql-connector-java-5.1.21-bin.jar* que se encuentra en la carpeta */lib* del proyecto y pulsar **Abrir**.
7. Pulsar **OK**. La Figura 2.11 muestra cómo quedarían las propiedades del proyecto con la librería *mysql-connector-java-5.1.21-bin.jar* incorporada en tiempo de compilación.

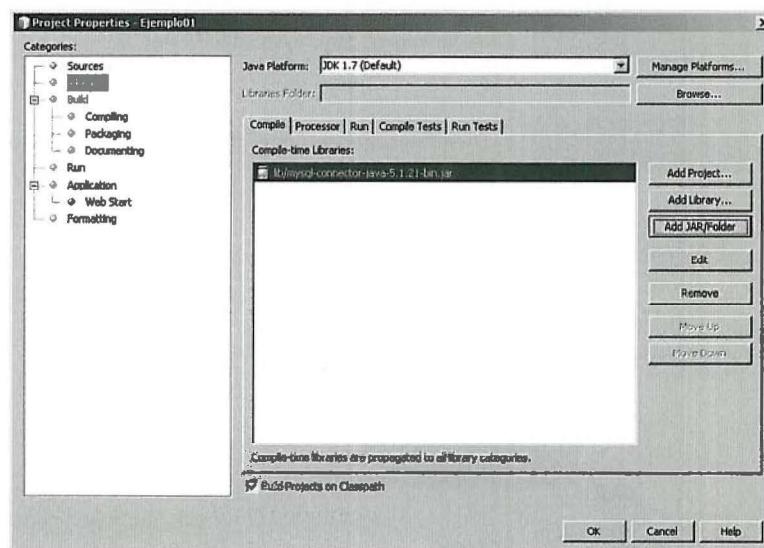


Figura 2.11. Librería añadida al proyecto

25 <http://dev.mysql.com/downloads/connector/j/>

26 Netbeans IDE 7.1.2 es la versión usada en los proyectos de este capítulo.

La segunda acción para poder utilizar el conector sin problemas es localizarlo en el sistema operativo identificando su ruta en la variable de entorno CLASSPATH, siempre que nuestro IDE (Eclipse, Netbeans, etc.) utilice esa variable. Desde consola el comando para lograr este propósito sería el siguiente:

```
$ set CLASSPATH=<PATH_DEL_JAR>\mysql-connector-java-5.1.21-bin.jar
```

Una vez localizado el *driver* (conector) en el sistema será posible cargarlo desde cualquier aplicación Java (Figura 2.10, pasos 1 y 2).

Un ejemplo es el siguiente código. Lo que hace es acceder a una base de datos llamada *discográfica* que se ha creado en MySQL. Esta base de datos tiene una tabla ALBUMES. Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico, TITULO, tipo VARCHAR(30), y AUTOR, tipo VARCHAR(30).

```
public Gestor_conexion() { //Constructor
    // crea una conexión
    Connection conn1 = null;
    try {
        String url1 = "jdbc:mysql://localhost:3306/discografica";
        String user = "root";
        String password = "";//no tiene clave
        conn1 = DriverManager.getConnection(url1, user, password);
        if (conn1 != null) {
            System.out.println("Conectado a discográfica...");
        }
    } catch (SQLException ex) {
        System.out.println("ERROR:dirección no válida o usuario/clave");
        ex.printStackTrace();
    }
}
```

El procedimiento de conexión con el controlador de la base de datos, independientemente de la arquitectura, es siempre muy similar. En primer lugar se carga el conector. Cualquier *driver* JDBC, independientemente de su tipo, debe implementar la interfaz *java.sql.Driver*. La carga del *driver* se realizaba originalmente con *Class.forName(driver)*. Sin embargo, al poner el *jar* del *driver* (MySQL en nuestro caso) en la carpeta *lib* del proyecto (o en el *classpath* del programa), cuando la clase *DriverManager* se inicializa, busca esta propiedad en el sistema y detecta que se necesita el *driver* elegido.

Una vez cargado el *driver*, el programador puede crear una conexión (paso 2 en la Figura 2.10). El objetivo es conseguir un objeto del tipo *java.sql.Connection* a través del método *DriverManager.getConnection (String url)*. En el código anterior se muestra el uso de este método.

La línea *url1 = "jdbc:mysql://localhost:3306/discografica";* indica que se desea acceder a una base de datos MySQL mediante JDBC, que la base de datos está localizable en el localhost (127.0.0.1) por el puerto 3306 y que su nombre es *discográfica* (sin tilde).

Si todo va bien, cuando se ejecute la sentencia donde el objeto *DriverManager* invoca al método *getConnection()* se crea una conexión a una base de datos MySQL. Si esa invocación fuera mal, se informará de una excepción gracias al uso de las sentencias *try-catch* utilizadas (captura de excepciones).

No hay que olvidar que, después de usar una conexión, ésta debe ser cerrada con el método *close()* de *Connection*. El siguiente código muestra un ejemplo de cómo hacerlo.

```
public void cerrar_Conexion (Connection conn1){  
    try {  
  
        conn1.close();  
  
    } catch (SQLException ex) {  
        System.out.println("ERROR:al cerrar la conexión");  
        ex.printStackTrace();  
    }  
}
```

### Pool de conexiones

La manera mostrada de obtener una conexión está bien para aplicaciones sencillas, donde únicamente se establece una conexión con la base de datos. Sin embargo, hay un pequeño problema con esta alternativa: varios hilos de ejecución no pueden usar una misma conexión física con la base de datos simultáneamente, ya que la información enviada o recibida por cada uno de los hilos de ejecución se entremezcla con la de los otros, haciendo imposible una escritura o lectura coherente en dicha conexión. Hay varias posibles soluciones para este problema:

- Abrir y cerrar una conexión cada vez que la necesitemos. De esta forma, cada hilo de ejecución tendrá la suya propia. Esta solución en principio no es eficiente, puesto que establecer una conexión real con la base de datos es un proceso costoso. El hecho de andar abriendo y cerrando conexiones con frecuencia puede hacer que el programa vaya más lento de lo debido.
- Usar una única conexión y sincronizar el acceso a ella desde los distintos hilos. Esta solución es más o menos eficiente, pero requiere cierta disciplina al programar, ya que es necesario poner siempre *synchronized* antes de hacer cualquier transacción con la base de datos. También tiene la pega de que los hilos deben esperar entre ellos.
- Finalmente, también existe la posibilidad de tener varias conexiones abiertas (*pool de conexiones*), de forma que cuando un hilo necesite una, la pida, y cuando termine, la deje para que pueda ser usada por los demás hilos, todo ello sin abrir y cerrar la conexión cada vez. De esta forma, si hay conexiones disponibles, un hilo no tiene que esperar a que otro acabe. Esta solución es en principio la ideal y es la que se conoce como *pool de conexiones*.

Apostando por el tercero de los escenarios anteriores, en Java, un *pool de conexiones* es una clase que tiene abiertas varias conexiones a bases de datos. Cuando alguien necesita una conexión a base de datos, en vez de abrirla directamente con *DriverManager.getConnection()*, se pide al *pool* usando su método *pool.getConnection()*. El *pool* coge una de las conexiones que ya tiene abierta, la marca para saber que está asignada y la devuelve. La siguiente llamada a este método *pool.getConnection()* buscará una conexión libre para marcarla como ocupada y ofrecerla.

El código Java asociado a esta forma de trabajar es el siguiente:

```
public Connection crearConexion() {  
    BasicDataSource bdSource = new BasicDataSource();  
    bdSource.setUrl ("jdbc:mysql://localhost:3306/dicografica");  
    bdSource.setUsername("root");  
    bdSource.setPassword("");  
    Connection con = null;  
    try {  
        if (con != null) {  
            System.out.println("No se puede crear la conexión");  
        } else {  
            //DataSource reserva una conexión y la devuelve para ser usada  
            con = bdSource.getConnection();  
            System.out.println("Conexión creada ");  
        }  
    } catch (Exception e) {  
        System.out.println("Error: " + e.toString());  
    }  
    return con;  
}
```

Esta aplicación necesita de la librería *commons-dbcp-all-1.3.jar*. Para usar esta librería se necesita seguir el mismo proceso que el mostrado para el *driver MySQL*, es decir, incluir el *commons-dbcp-all-1.3.jar* en la carpeta */lib* del proyecto, y seguidamente añadirla a la librería de NetBeans.<sup>27</sup>

En la librería *commons-dbcp-all-1.3.jar* se dispone de una implementación sencilla de un *pool de conexiones*, ya que al ser *DataSource* una interfaz se debe facilitar una implementación para poder instanciar objetos de esa clase. Esta librería necesita a su vez la librería *commons-pool*, por lo que también es necesario descargarla si se quiere usar este *pool*. Una vez configurado, para usar *BasicDataSource* solo es necesario hacer un *new* de esa clase y pasarle los parámetros adecuados de nuestra conexión con los métodos *set()* disponibles para ello.

## ACTIVIDADES 2.1



- Utilizar el contenido de la sección para configurar el entorno de desarrollo (IDE Netbeans si es posible) para incluir las librerías JDBC que dan acceso a MySQL.



### PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado.

<sup>27</sup> En el código asociado a este proyecto está disponible la librería *apache commons-dbcp* completa, dentro de la cual se encuentra *commons-dbcp-1.4.jar*.

### 2.2.5 CLASES BÁSICAS DEL API JDBC

En la sección anterior se ha mostrado cómo hacer una conexión con una base de datos MySQL. Como se ha comentado anteriormente, la interfaz del conector JDBC reside en los paquetes `java.sql` y `javax.sql`. Lo que se ofrece en esos paquetes son en su mayoría interfaces, ya que la implementación específica de cada una de ellas es fijada por cada proveedor según su protocolo de bases de datos. En cualquier caso, en la interfaz hay distintos tipos de objetos que se deben tener presentes, por ejemplo `Connection`, `Statement` y `ResultSet`. El resto de objetos necesarios se mostrarán en próximas secciones.

- Los objetos de la clase `Connection`<sup>28</sup> ofrecen un enlace activo a una base de datos a través del cual un programa en Java puede leer y escribir datos, así como explorar la estructura de la base de datos y sus capacidades. Se crea con una llamada a `DriverManager.getConnection()` o a `DataSource.getConnection()` (en JDBC 2.0). En la sección anterior se han mostrado ejemplos asociados donde se utilizan ambas llamadas.
- La interfaz `DriverManager`,<sup>29</sup> complementaria de la clase `Connection`. Con ella se registran los controladores JDBC y se proporcionan las conexiones que permiten manejar las URL específicas de JDBC. Se consigue con el método `getConnection()` de la propia clase.
- La clase `Statement`<sup>30</sup> proporciona los métodos para que las sentencias, utilizando el lenguaje de consulta estructurado (SQL), sean ejecutadas sobre la base de datos y se pueda recuperar el resultado de su ejecución. Hay tres tipos de sentencias `Statement`,<sup>31</sup> cada una especializa a la anterior. Estas sentencias se verán en las siguientes secciones.
- Además de las clases anteriores, el API JDBC ofrece también la posibilidad de gestionar excepciones con la clase `SQLException`. Dicha clase es la base de las excepciones de JDBC. La mayor parte de las operaciones que proporciona el API JDBC lanzarán la excepción `java.sql.SQLException` en caso de que se produzca algún error en la base de datos (por ejemplo, errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc.). Por este motivo es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques `try/catch`.

## ACTIVIDADES 2.2



- Crear en MySQL una base de datos de ejemplo para una *discográfica*: una tabla con *canción* (título (`Varchar()`), duración (`Varchar()`), letra (`Varchar()`) y *álbum* (id (`Int`), título (`Varchar()`), año en el que se publicó (`Varchar()`)). La relación entre las tablas *álbum* y *canción* es uno a muchos: un álbum está compuesto por muchas canciones.

28 Más información sobre esta clase se puede encontrar en la siguiente dirección: <http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html>

29 Más información sobre esta clase se puede encontrar en la siguiente dirección: <http://docs.oracle.com/javase/7/docs/api/java/sql/DriverManager.html>

30 Más información sobre esta clase se puede encontrar en la siguiente dirección: <http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html>

31 Estos tres tipos serán detallados en las siguientes secciones.

### 2.2.6 CLASES ADICIONALES DEL API JDBC

Además de las clases básicas anteriores, el API JDBC también ofrece la posibilidad de acceder a los metadatos de una base de datos. Con ellos se puede obtener información sobre la estructura de la base de datos y, gracias a ello, se pueden desarrollar aplicaciones independientemente del esquema que tenga la base de datos. Las principales clases asociadas a metadatos de una base de datos son las siguientes: *DatabaseMetaData* y *ResultSetMetaData*.

- Los objetos de la clase *DatabaseMetaData* ofrecen la posibilidad de operar con la estructura y capacidades de la base de datos. Se instancian con *connection.getMetaData()*. Los metadatos son datos acerca de los datos, es decir, datos que explican la naturaleza de otros datos. La interfaz *DatabaseMetaData* contiene más de 150 métodos para recuperar información de una base de datos (catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, procedimientos almacenados, vistas etc.), así como información sobre algunas características del controlador JDBC que se esté utilizando. Estos métodos son útiles cuando se implementan aplicaciones genéricas que pueden acceder a diversas bases de datos.
- Los objetos de la clase *ResultSetMetaData* son el *ResultSet* que se devuelve al hacer un *executeQuery()* de un objeto *DatabaseMetaData*. Los métodos de *ResultSetMetaData* permiten determinar las características de un objeto *ResultSet*. Por ejemplo, con un objeto de la clase *ResultSetMetaData* se puede determinar el número de columnas; información sobre una columna, tal como el tipo de datos o la longitud; la precisión y la posibilidad de contener nulos, e información sobre si una columna es de solo lectura, etc.

## ACTIVIDADES 2.3



- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que conecte con la base de datos.



### PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado. Sin embargo, no tiene asociada la base de datos, que se creó con XAMPP<sup>32</sup> en local.

32 <http://www.apachefriends.org/es/xampp.html>

## 2.3 EJECUCIÓN DE SENTENCIAS DE DEFINICIÓN DE DATOS

El lenguaje de definición de datos (*Data Definition Language* o *Data Description Language* [DDL] según autores) es la parte de SQL dedicada a la definición de una base de datos. Dicho lenguaje consta de sentencias para definir la estructura de la base de datos y permite definir gran parte del nivel interno de la misma. Por este motivo, estas sentencias serán utilizadas normalmente por el administrador de la base de datos.

Las principales sentencias asociadas con el lenguaje DDL son CREATE, ALTER y DROP. Siempre se usan estas sentencias junto con el tipo de objeto y el nombre del objeto. Dichas sentencias permiten:

- CREATE sirve para crear una base de datos o un objeto.
- ALTER sirve para modificar la estructura de una base de datos o de un objeto.
- DROP permite eliminar una base de datos o un objeto.

Para enviar comandos SQL a la base de datos con JDBC se usa un objeto de la clase *Statement*. Este objeto se obtiene a partir de una conexión a base de datos, de esta forma:

```
Statement st = conexion.createStatement();
```

*Statement* tiene muchos métodos, pero hay dos especialmente interesantes: *executeUpdate()* y *executeQuery()*.

- *executeUpdate()*: se usa para sentencias SQL que impliquen modificaciones en la base de datos (INSERT, UPDATE, DELETE, etc.).
- *executeQuery()*: se usa para consultas (SELECT y similares).

El siguiente ejemplo muestra cómo se modifica una tabla desde una aplicación Java:

```
// Añadir una nueva columna a una tabla ya existente
Statement sta = con.createStatement();
int count = sta.executeUpdate("ALTER TABLE contacto ADD edad");
```

Por último, el siguiente código elimina la tabla llamada *contacto* de una base de datos previamente abierta:

```
st.executeUpdate("DROP TABLE contacto");
```

## 2.4 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

Las sentencias de manipulación de datos (*Data Manipulation Language* [DML]) son las utilizadas para insertar, borrar, modificar y consultar los datos que hay en una base de datos. Las sentencias DML son las siguientes:

- La sentencia SELECT sirve para recuperar información de una base de datos y permite la selección de una o más filas y columnas de una o muchas tablas.
- La sentencia INSERT se utiliza para agregar registros a una tabla.
- La sentencia UPDATE permite modificar la información de las tablas.
- La sentencia DELETE permite eliminar una o más filas de una tabla.

El siguiente ejemplo muestra un método para insertar valores a la tabla *álbum* creada en la Actividad 2.2.

```
public void Insertar() {  
    try {  
        // Crea un statement  
        Statement sta = conn1.createStatement();  
        // Ejecuta la inserción  
        sta.executeUpdate("INSERT INTO album " + "VALUES (3, 'Black Album', 'Metallica')");  
        // Cierra el statement  
        sta.close();  
    } catch (SQLException ex) {  
        System.out.println("ERROR: al hacer un Insert");  
        ex.printStackTrace();  
    }  
}
```

En el ejemplo, partiendo de una conexión previa (*conn1*) se crea un objeto *Statement* llamado *sta*. Sobre ese objeto se ejecuta una consulta *Insert into.SQLException*, que se encarga de capturar los errores que se cometan en la sentencia. Por último, al terminar de usar el *Statement*, este debe cerrarse. (*close()*) para evitar errores inesperados.

### ACTIVIDADES 2.4



- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita modificar la tabla *álbum* para incluir un nuevo campo que contenga las imágenes de las carátulas de cada álbum.

## 2.5 EJECUCIÓN DE CONSULTAS

La ejecución de consultas sobre bases de datos desde aplicaciones Java descansa en dos tipos de clases disponibles en el API JDBC y en dos métodos. Las clases son *Statement* y *ResultSet*, y los métodos, *executeQuery* y *executeUpdate*.

### 2.5.1 CLASE STATEMENT

Como se ha comentado en varias ocasiones, las sentencias *Statement* son las encargadas de ejecutar las sentencias SQL estáticas con *Connection.createStatement()*.

El método *executeQuery()* de *Statement* está diseñado para sentencias que devuelven un único resultado (*ResultSet*),<sup>33</sup> como es el caso de las sentencias SELECT.

```
ResultSet res = sta.executeQuery();
```

Los objetos de la clase *ResultSet* son los utilizados para representar la respuesta a las peticiones que se hacen a una base de datos. Esta clase no es más que un conjunto ordenado de filas de una tabla. Asociados a la clase *ResultSet* existen métodos como *next()* y *getXXX()* para iterar por las filas y obtener los valores de los campos deseados. Después de invocar al método *next()*, el resultado recién traído está disponible en el *ResultSet*. La forma de recoger los campos es pedirlos con algún método *getXXX()*. Si se sabe de qué tipo es el dato, se puede pedir con *getInt()*, *getString()*, etc. Si no se sabe o da igual el tipo (como en el ejemplo), bastará con un *getObject()*, que es capaz de traer cualquier tipo de dato.

El siguiente código muestra un ejemplo de acceso a la base de datos con una consulta SELECT que obtiene todos los álbumes cuyo título empieza por "B"

```
public void Consulta_Statement() {
    try {
        Statement stmt = conn1.createStatement();
        String query = "SELECT * FROM album WHERE titulo like 'B%'";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            System.out.println("ID - " + rs.getInt("id") +
                ", Título " + rs.getString("titulo") +
                ", Autor " + rs.getString("autor"));
        }
        rs.close();
        stmt.close();
    } catch (SQLException ex) {
        System.out.println("ERROR: al hacer un Insert");
        ex.printStackTrace();
    }
}
```

<sup>33</sup> Más información sobre esta clase se puede encontrar en: <http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html>

El código ejecuta la consulta deseada con `executeQuery()` y el resultado lo devuelve en un `ResultSet` (llamado `rs`). El método `next()` de `ResultSet` permite recorrer todas las filas para ir sacando cada uno de los valores devueltos. Como el atributo `id` es de tipo `Int` se usa un método `getInt()` para recuperarlo. Sin embargo, como `título` y `autor` es de tipo `VARCHAR()` se usa un `getString()`.

Por último, al terminar de usar `ResultSet` y `Statement`, estos deben cerrarse, `close()` para evitar errores inesperados.

### 2.5.2 CLASE PREPAREDSTATEMENT

Una primera variante de la sentencia `Statement` es la sentencia `PreparedStatement`. Se utiliza para ejecutar las sentencias SQL precompiladas. Permite que los parámetros de entrada sean establecidos de forma dinámica, ganando eficiencia.

El siguiente ejemplo muestra la ejecución de la consulta de la sección anterior, pero parametrizando el criterio de búsqueda. La diferencia principal con respecto a los `Statement` es que las consultas pueden tener valores indefinidos que se establecen con el símbolo interrogación (?). En las consultas se ponen tantas interrogaciones como parámetros se quieran usar. En el siguiente ejemplo solo hay un parámetro.

```
public void Consulta_preparedStatement() {
    try {
        String query = "SELECT * FROM album WHERE titulo like ?";
        PreparedStatement pst = conn1.prepareStatement(query);
        pst.setString(1, "B%");

        ResultSet rs = pst.executeQuery();
        while (rs.next()) {
            System.out.println("ID - " + rs.getInt("id") +
                ", Título " + rs.getString("titulo") +
                ", Autor " + rs.getString("autor"));
        }
        rs.close();
        pst.close();
    } catch (SQLException ex) {
        System.out.println("ERROR:al consultar");
        ex.printStackTrace();
    }
}
```

Al crear el `prepareStatement` se precompila la consulta para dejarla preparada para recibir los valores de los parámetros. Si el parámetro que se quiere colocar es de tipo `Int` se usa `setInt()`. Sin embargo, en el ejemplo anterior se quiere dar un valor de texto por lo que se usa `setString(1, "B%")`. El primer valor (1) indica que la "B%" se asocia con la primera interrogación que se encuentra. Si hubiese más parámetros (?) entonces habría que indicar la posición que ocupa para que el `prepareStatement` sepa a cuál asociarle el valor (`setString(2,"")`, `setString(3, "")`, etc.).<sup>34</sup>

<sup>34</sup> Más información sobre métodos para asignar valores a los parámetros de `preparedStatement` son mostrados en <http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html>

### 2.5.3 CLASE CALLABLESTATEMENT

Otro tipo de sentencias son las asociadas a los objetos de la clase *CallableStatement*, que son sentencias *preparedStatement* que llaman a un procedimiento almacenado, es decir, métodos incluidos en la propia base de datos. No todos los gestores de bases de datos admiten este tipo de procedimientos. La manera de proceder es la misma que la mostrada en el ejemplo *prepareStatement* aunque lo que se le da como parámetro es el nombre del procedimiento almacenado junto con los valores de los parámetros del procedimiento puestos como parámetros del *Statement*.

En el siguiente código se muestra un ejemplo de llamada para un supuesto procedimiento almacenado llamado *DameAlbumes(titulo, autor)*, que devuelve todos los álbumes cuyo título y autor coincida con los valores dados.

```
CallableStatement cs =
conn1.prepareCall("CALL DameAlbumes(?,?)");

// Se proporcionan valores de entrada al procedimiento
cs.setString(1, "Black%");
cs.setString(2, "Metallica");
```

### ACTIVIDADES 2.5



- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita consultar con consultas SELECT las tablas *álbum* y *canciones*. Las consultas deben ser parametrizadas (*Statement*) y no parametrizadas (*PreparedStatement*). El resultado debe mostrarse en forma de lista de resultados.

## 2.6 GESTIÓN DE TRANSACCIONES

Una transacción en un sistema de gestión de bases de datos (SGBD) es un conjunto de órdenes que se ejecutan como una unidad de trabajo, es decir, de forma indivisible o atómica. Una transacción se inicia cuando se encuentra una primera sentencia DML y finaliza cuando se ejecuta alguna de las siguientes sentencias:

- Un COMMIT o un ROLLBACK.
- Una sentencia DDL, por ejemplo CREATE.
- Una sentencia DCL (lenguaje de control de datos), dentro de las que se incluyen las sentencias que permiten al administrador controlar el acceso a los datos contenidos en una base de datos (GRANT o REVOKE).

Las transacciones consisten en la ejecución de bloques de consultas manteniendo las propiedades ACID (*Atomicity-Consistency-Isolation-Durability*), es decir, permiten garantizar integridad ante fallos y concurrencia de transacciones.

Después de que una transacción finaliza, la siguiente sentencia ejecutada automáticamente inicia la siguiente transacción. Una sentencia DDL o DCL es automáticamente completada y por consiguiente implícitamente finaliza una transacción.

Una transacción que termina con éxito se puede confirmar con una sentencia COMMIT, en caso contrario puede abortarse utilizando la sentencia ROLLBACK. En JDBC por omisión cada sentencia SQL se confirma tan pronto se ejecuta, es decir, una conexión funciona por defecto en modo *auto-commit*. Para ejecutar varias sentencias en una misma transacción es preciso deshabilitar el modo *auto-commit*, después se podrán ejecutar las instrucciones, y terminar con un COMMIT si todo va bien o un ROLLBACK en otro caso.

El siguiente código muestra el uso de transacciones con el ejemplo de insertar valores en la tabla *álbum*.

```
public void Insertar_con_commit(){
    try {
        conn1.setAutoCommit(false);
        Statement sta = conn1.createStatement();
        sta.executeUpdate("INSERT INTO album " + "VALUES (5, 'Black Album', 'Metallica')");
        sta.executeUpdate("INSERT INTO album " + "VALUES (6, 'A kind of magic', 'Queen')");
        conn1.commit();
    } catch (SQLException ex) {
        System.out.println("ERROR: al hacer un Insert");
        try{
            if(conn1!=null) conn1.rollback();
        }catch(SQLException se2){
            se2.printStackTrace();
        }//end try
        ex.printStackTrace();
    }
}
```

Como se puede ver en el ejemplo, si las dos operaciones *Insert into* se ejecutan correctamente, entonces se aplica un *commit* antes de salir. Sin embargo, si no es así, y salta una excepción, hay que hacer un *rollback()*. El *rollback()* es obligado ponerlo dentro de un *try/catch*.

## ACTIVIDADES 2.6



- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita realizar varias inserciones de datos en las tablas *canciones* y *álbum* de manera atómica. Si falla la inserción en una de las tablas entonces todo el proceso se debe anular.

## 2.7

### CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se ha mostrado JDBC como alternativa para conectar aplicaciones Java con bases de datos. El uso de conectores es muy habitual en el desarrollo de aplicaciones ya que facilita enormemente el acceso a datos y la ejecución de persistencias, al independizar el código del sistema gestor de bases de datos subyacentes. Un programador de aplicaciones multiplataforma debe tener unos conocimientos avanzados sobre el uso de conectores.

El lector interesado en profundizar en las tecnologías expuestas en el capítulo puede hacerlo en las líneas siguientes:

- ✓ Optimización de conexiones (*pool* de conexiones en aplicaciones multihilo) y de acceso a datos relacionales con SQL.
- ✓ Otras alternativas para el acceso con conectores, por ejemplo Java Blend y SQLJ.

Para profundizar en estas líneas de trabajo, el título *SQL y Java: Guía para SQLJ, JDBC y tecnologías relacionadas*, de Jim Melton y Andrew Eisenberg, de la editorial RA-MA, es una buena referencia.



### RESUMEN DEL CAPÍTULO



En este capítulo se ha abordado el acceso a datos con conectores. En concreto, siendo coherentes con el resto de contenidos de este libro, se ha trabajado con JDBC, la alternativa más extendida para el acceso a datos almacenados en sistemas gestores relacionales desde Java.

La primera parte se ha centrado en una introducción a JDBC y sus posibilidades.

La segunda parte muestra con ejemplos de código el uso de las principales interfaces Java que permiten ejecutar operaciones de modificación y consulta con SQL.