FP Informática y Comunicaciones
G.S. Desarrollo de Aplicaciones Multiplataforma

Carlos Alberto Cortijo Bon

ACCESO A DATOS

Segunda edición

Complementos digitales

EDITORIAL SINTESIS

Acceso a datos

Carlos Alberto Cortijo Bon

(segunda edición)

EDITORIAL SINTESIS

Vallehermoso, 34. 28015 Madrid
Teléfono 91 593 20 98
www.sintesis.com

ISBN: 978-84-1357-486-8
Depósito Legal: M-8.992-2026

Impreso en España - Printed in Spain

ÍNDICE

2. Bases de datos relacionales RA2

3. Correspondencia objeto-relacional RA3

5. Bases de datos documentales RA5

6. Componentes para el acceso a datos RA6

Índice de complementos digitales

2 Bases de datos relacionales

RESULTADO DE APRENDIZAJE Y CRITERIOS DE EVALUACIÓN

RA 2. Desarrolla aplicaciones que gestionan información almacenada en bases de datos relacionales identificando y utilizando mecanismos de conexión.

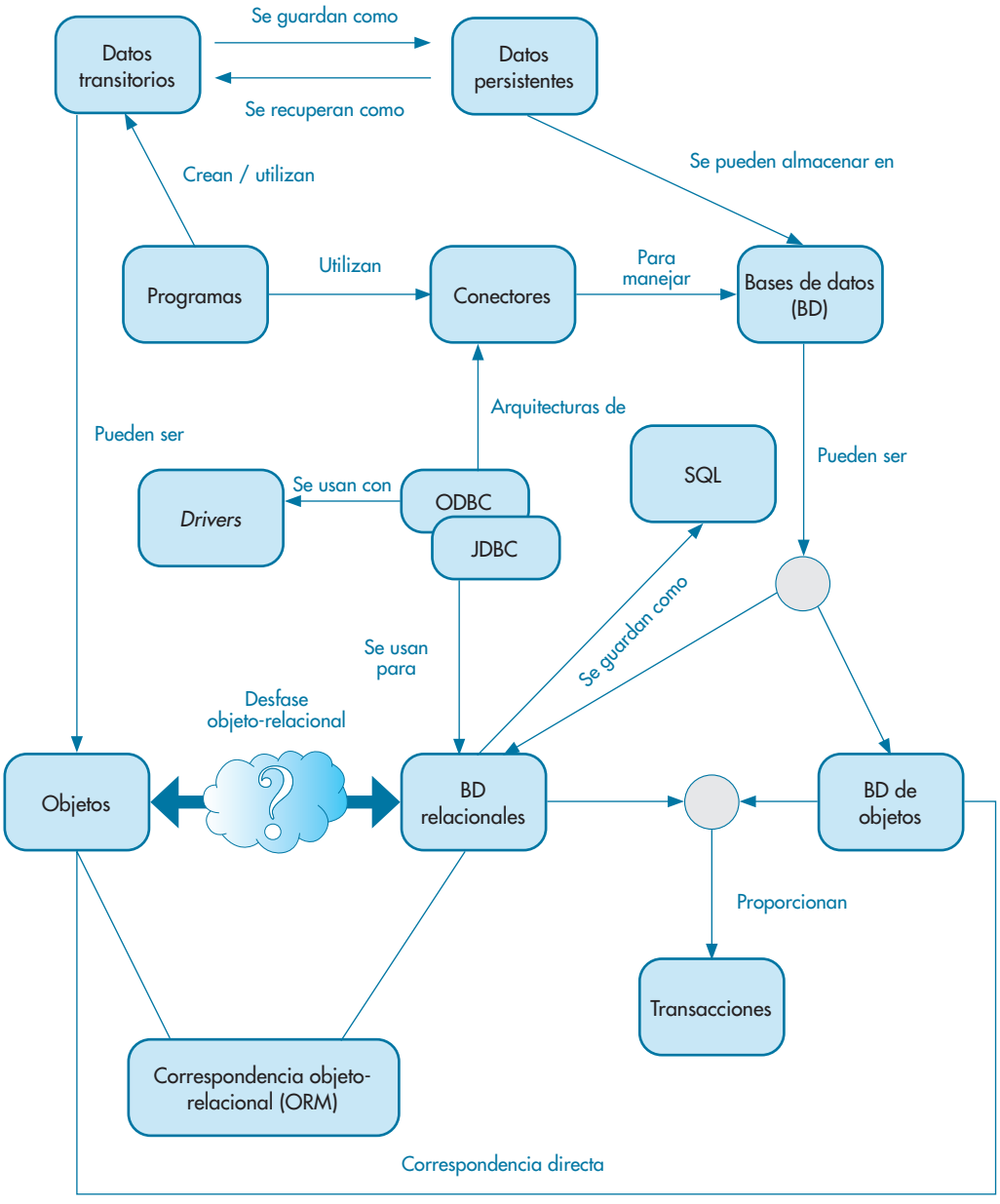
a) Se han valorado las ventajas e inconvenientes de utilizar conectores.
b) Se han utilizado gestores de bases de datos embebidos e independientes.
c) Se ha utilizado el conector idóneo en la aplicación.
d) Se ha establecido la conexión.
e) Se ha definido la estructura de la base de datos.
f) Se han desarrollado aplicaciones que modifican el contenido de la base de datos.
g) Se han definido los objetos destinados a almacenar el resultado de las consultas.
h) Se han desarrollado aplicaciones que efectúan consultas.
i) Se han eliminado los objetos una vez finalizada su función.
j) Se han gestionado las transacciones.
k) Se han ejecutado procedimientos almacenados en la base de datos.

Objetivos de Desarrollo Sostenible

En este capítulo se va a trabajar el ODS 9.

MAPA CONCEPTUAL

BASES DE DATOS RELACIONALES



GLOSARIO

Clave autogenerada. Columna numérica de una tabla que se define como clave primaria y para la que no se especifica valor cuando se inserta una nueva fila, de manera que el propio sistema gestor de bases de datos (SGBD) le asigna automáticamente un valor.

Commit. Operación que confirma de forma definitiva todos los cambios realizados en una transacción, haciendo que pasen a ser visibles para otros usuarios y procesos.

Conector. API que permite a los programas de aplicación trabajar con bases de datos.

Cursor. Mecanismo que permite recorrer secuencialmente las filas de un conjunto de resultados devuelto por una consulta SQL.

Driver de JDBC. Biblioteca de *software* que proporciona una implementación para una base de datos particular de las interfaces definidas en la especificación JDBC, de manera que permite a JDBC interactuar con esa base de datos.

Excepción de SQL. Error que se produce durante la comunicación o ejecución de operaciones con una base de datos y que proporciona información sobre su causa.

JDBC *(Java Database Connectivity)*. Conector a bases de datos relacionales para Java.

Pool de conexiones. Conjunto de conexiones a una base de datos que se mantienen abiertas y a disposición de los procesos que puedan necesitarlas, lo que evita el retraso y la sobrecarga que supone la apertura de una nueva conexión cada vez que un proceso necesita una.

Procedimientos y funciones almacenados. Procedimientos y funciones escritos en un lenguaje procedural que es una extensión de SQL, y que se ejecutan en el propio SGBD.

***Rollback*.** Operación que deshace todos los cambios realizados en una transacción desde su inicio, devolviendo la base de datos a su estado anterior.

Sentencia preparada. Sentencia de SQL parametrizada que, una vez precompilada en el SGBD, permite su ejecución de manera segura y eficiente.

Transacción. Conjunto de sentencias de SQL que forma una unidad lógica y que se ejecuta de manera atómica y aislada de otras transacciones u operaciones con la base de datos.

PUNTO DE PARTIDA

La empresa SoftManage S.L., dedicada al desarrollo de soluciones informáticas para pequeñas y medianas empresas, ha detectado la necesidad de mejorar su sistema de gestión interna. Actualmente, la información sobre empleados y departamentos se guarda en distintos ficheros y hojas de cálculo, lo que provoca errores de actualización y pérdida de datos. Para resolverlo, la dirección ha decidido centralizar la información en una base de datos relacional, que garantice la integridad y facilite el acceso simultáneo por parte de varios usuarios.

Como nuevo integrante del equipo de desarrollo, se te encarga crear un módulo de acceso a datos en Java utilizando el conector JDBC. Este módulo deberá ser capaz de:

- Crear las tablas `departamentos` y `empleados` mediante sentencias DDL.
- Insertar registros de ejemplo utilizando sentencias preparadas *(PreparedStatement)*.

- Consultar los datos de los empleados, y del departamento al que pertenecen.
- Gestionar las excepciones SQL que se puedan producir, de manera que se proporcione información adecuada acerca de ellas y no se aborte la ejecución del programa.
- Implementar una transacción que asegure en todo momento la consistencia e integridad de los datos.

El resultado será un programa funcional que sirva de base para futuras ampliaciones, como la gestión de proyectos o el control horario, y que demuestre la correcta comunicación entre la aplicación y la base de datos relacional.

2.1. Conectores

Los sistemas gestores de bases de datos (SGBD) de distintos tipos (relacionales, de XML, de objetos, documentales, o del tipo que sea) tienen sus propios lenguajes especializados para operar con los datos que almacenan. En cambio, los programas de aplicación se escriben con lenguajes de programación de propósito general, como por ejemplo Java. Para que los programas de aplicación puedan interactuar con los SGBD, se necesitan mecanismos que permitan a los programas de aplicación comunicarse con las bases de datos en estos lenguajes. Estos se implementan en API que se denominan *conectores* (véase figura 2.1).

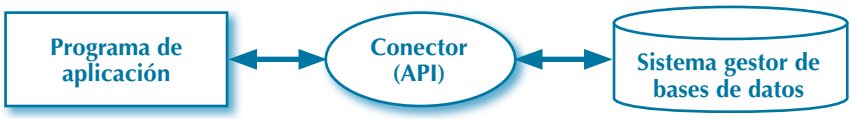


Figura 2.1
Interacción con bases de datos utilizando conectores

2.2. Conectores para bases de datos relacionales

Los sistemas de bases de datos más utilizados hoy en día, con mucha diferencia, son los relacionales. Para trabajar con ellos se utiliza SQL. SQL es un lenguaje estándar. Pero existen multitud de bases de datos relacionales distintas, y cada una tiene su propia versión de SQL con sus propias particularidades. Aparte de eso, cada base de datos tiene sus propias interfaces de bajo nivel.

El uso de *drivers* permite desarrollar una arquitectura genérica en la que el conector tiene una interfaz común para las aplicaciones, y los *drivers* se ocupan de las particularidades de las distintas bases de datos (figura 2.2).

De esta manera, el conector no es solo una simple API, sino una arquitectura, porque especifica unas interfaces que los distintos *drivers* tienen que implementar para acceder a las bases de datos particulares. La primera arquitectura de conectores que surgió fue ODBC (Open DataBase Connectivity), desarrollada por Microsoft para Windows a principios de los noventa. ODBC es una API para el lenguaje C, y se usa ampliamente hoy en día tanto en entornos Windows como en Linux y Unix. Con el tiempo surgieron otras arquitecturas como sucesoras

de ODBC, tales como OLE-DB y ADO (ActiveX Data Objects). Estos sucesores de ODBC son API nativas para Windows, y en entornos Windows se utilizan hoy como alternativa o evolución de ODBC. En entornos Linux o Unix, ODBC sigue siendo la principal solución para acceso a bases de datos relacionales desde C.

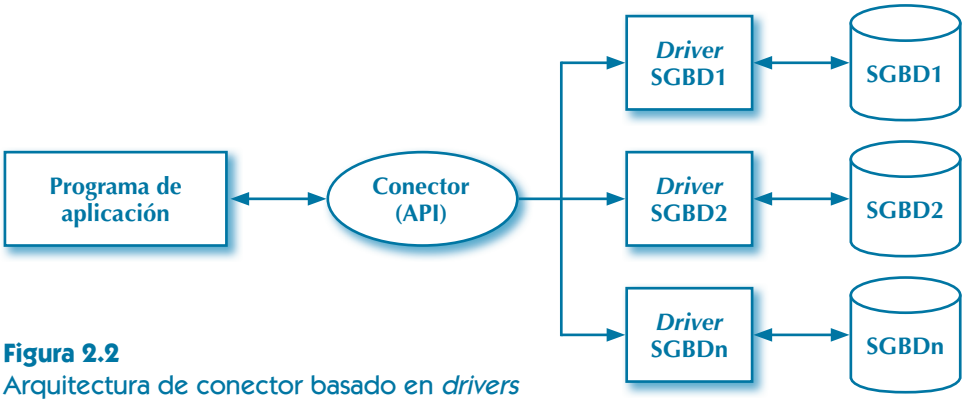


Figura 2.2
Arquitectura de conector basado en *drivers*

TOMA NOTA

A finales de los años noventa surgió JDBC, como el equivalente a ODBC para Java, y es similar en muchos aspectos. De hecho, según se indica en la propia especificación de JDBC, ambos están basados en el estándar X/Open SQL CLI que especifica la manera en que un programa debe enviar sentencias de SQL a un SGBD y operar con los *recordsets* (conjuntos de registros o filas) obtenidos. Este estándar se definió a principios de los noventa y solo para los lenguajes C y COBOL.

Existen *drivers* de ODBC y de JDBC para todas las bases de datos importantes hoy en día. Existe un *driver* de JDBC para ODBC que se puede utilizar cuando no se dispone de un *driver* de JDBC específico para una base de datos, pero sí de uno de ODBC.

Existen *drivers* de JDBC no solo para bases de datos relacionales, sino para sistemas de almacenamiento basados en ficheros que almacenan los datos en forma más o menos tabular, como por ejemplo ficheros CSV (ya vistos en capítulos anteriores) y hojas de cálculo, e incluso para XML, que almacena los datos no de forma tabular, sino jerárquica.

Los beneficios que proporcionan los conectores basados en *drivers* –principalmente, independencia de la base de datos– se consiguen a cambio de una mayor complejidad y, en algunos casos, de un menor rendimiento.

2.3. Acceso a resultados de consultas sobre bases de datos relacionales mediante conectores

Los conectores permiten realizar todo tipo de operaciones sobre una base de datos relacional. Pero este apartado se centrará en las operaciones de consulta.

La cuestión fundamental que se plantea con los conectores es la correspondencia entre las estructuras de datos utilizadas para el almacenamiento en la base de datos y las estructuras de datos de las que dispone el lenguaje de programación. Con las bases de datos relacionales, la estructura de datos fundamental para el almacenamiento de la información es la tabla. Cada tabla tiene un conjunto fijo de columnas, cada una con un tipo de datos determinado. Una consulta de SQL devuelve un conjunto de filas o *recordset*. Los conectores permiten recuperar estos resultados fila a fila, mediante un objeto que actúa como *iterador* o *cursor*. A continuación, se muestra la manera de realizar una consulta y obtener sus resultados utilizando conectores. Se ha utilizado la sintaxis del lenguaje Java y las clases de JDBC, pero en esencia es igual para cualquier base de datos relacional y para cualquier lenguaje de programación.

```
try(
        Connection c = DriverManager.getConnection(datos de conexión);
        Statement s = c.createStatement();
        ResultSet rs = s.executeQuery("SELECT... ")
) {
    while(rs.next()) { // Quedan resultados de la consulta
        String dato1 = rs.getString(1); // String de la columna
        int dato2 = rs.getInt(2); // int de segunda columna
    }
```

El objeto de tipo `ResultSet` actúa como iterador sobre los resultados de la consulta, que son un conjunto de filas. Una vez recuperada una fila, se puede acceder a cada dato indicanado su posición (como en el ejemplo anterior) o su nombre (como se verá más adelante).

Este modelo de acceso es válido, con algunas diferencias, para otros tipos de bases de datos, tales como bases de datos de objetos y de XML. Se trata siempre de abrir una conexión, realizar una consulta y utilizar un iterador o cursor para obtener uno a uno los resultados. Según el tipo de base de datos, habrá diferentes operaciones para hacer avanzar el cursor, y se utilizarán diferentes estructuras de datos para recuperar los resultados individuales.

2.4. Desfase objeto-relacional

Hacer a la inversa que en el apartado anterior, es decir, almacenar los resultados de variables de memoria en una base de datos relacional, puede ser más complicado. Especialmente cuando se trabaja con un lenguaje orientado a objetos y con objetos complejos, es decir, con objetos que contienen referencias a otros objetos, y referencias a colecciones de objetos relacionados. Una colección de objetos complejos tiene estructura de grafo, y no es sencillo almacenar esta información en tablas con filas y columnas. Al conjunto de dificultades que eso plantea se le conoce como *desfase objeto-relacional,* del inglés *object-relational impedance mismatch* o desajuste de impedancia objeto-relacional (véase figura 2.3).

ACTIVIDAD GRUPAL 2.1

En grupos de cuatro, pensad en un modelo de clases (un conjunto de clases relacionadas) que describa algún aspecto de la vida cotidiana con la que se pueda explicar

el desfase objeto-relacional. Cada grupo pondrá en común su ejemplo y explicará al resto de la clase los motivos por los que lo han escogido. Al terminar todas las exposiciones, se debatirá cuál representa mejor las dificultades de transformar estructuras de objetos complejos en tablas relacionales.

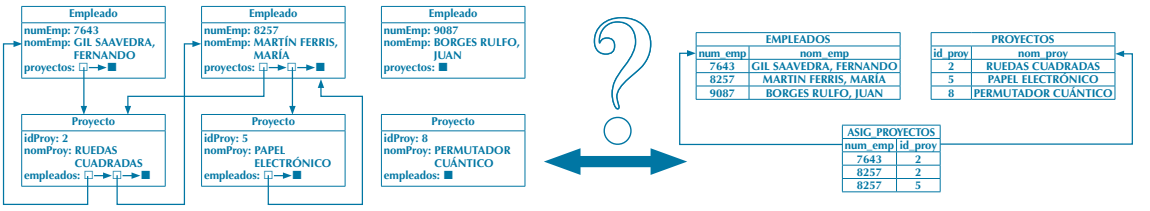


Figura 2.3
Desfase objeto relacional

2.5. JDBC (Java Database Connectivity)

La arquitectura de conectores JDBC para Java está basada en *drivers,* como ya se ha explicado, y su API está disponible en el paquete `java.sql`. Los *drivers* de JDBC proporcionan clases que implementan las interfaces de la API JDBC para una base de datos particular. Como ya se ha comentado también, puede haber *drivers* de JDBC no para una base de datos, sino para otro conector. En particular, existe un *driver* de JDBC para ODBC que se puede utilizar si, para una base de datos, no existe un *driver* de JDBC, pero sí de ODBC.

RECURSO WEB

Este código QR proporciona acceso a un sitio web de Oracle con información general, una breve introducción y tutoriales de JDBC.

2.6. Operaciones básicas con JDBC

En este apartado se presupone un conocimiento básico del lenguaje SQL.

En SQL se pueden diferenciar varios sublenguajes, y a cada uno de ellos pertenecen varios tipos de sentencias. Se puede diferenciar entre DML (*data manipulation language* o lenguaje de manipulación de datos) y DDL (*data definition language* o lenguaje de definición de datos). Dentro de DML se pueden diferenciar operaciones de consulta y de modificación de datos. Las sentencias de consulta (`SELECT`) se ejecutan con `executeQuery`, que devuelve una lista de filas en un `ResultSet`, sobre el que se puede iterar para obtener los resultados uno a uno. El resto de las sentencias de DML (`UPDATE`, `DELETE`, `INSERT`) se ejecutan con `executeUpdate`, que devuelve el número de filas afectadas por la operación. Las sentencias de DDL se ejecutan

con `execute`. En el cuadro 2.1 se resume lo necesario para ejecutar cualquier sentencia de SQL con JDBC.

Las interfaces `Connection`, `Statement` y `ResultSet` implementan la interfaz `AutoCloseable`, y por tanto deberían gestionarse en un bloque `try` con recursos, y cuando así se hace no es necesario ejecutar explícitamente el método `close` con ellas.

CUADRO 2.1. Ejecución de sentencias SQL con JDBC

<table>
<tr><td colspan="2"><code>Connection c = DriverManager.getConnection(url y datos de conexión);</code></td><td>Crear conexión utilizando el driver apropiado según el URL de conexión.</td></tr>
<tr><td colspan="2"><code>Statement s = c.createStatement();</code></td><td>Crear sentencia</td></tr>
<tr><td><code>ResultSet rs = s.executeQuery(consulta);
while(rs.next()) {
...
}
rs.close();</code></td><td><code>int res = s.executeUpdate
(o bien) sent. DML);
boolean res = s.execute
(sent. DDL);</code></td><td>Ejecutar sentencia
Si consulta, obtener resultados fila a fila y cerrar lista de resultados</td></tr>
<tr><td colspan="2"><code>s.close();</code></td><td>Cerrar sentencia</td></tr>
<tr><td colspan="2"><code>c.close();</code></td><td>Cerrar conexión</td></tr>
</table>

RECUERDA

Deben cerrarse siempre los objetos `Connection`, `Statement`, `PreparedStatement` y `ResultSet`. Todos ellos implementan la interfaz `AutoCloseable`, por lo que se pueden gestionar en bloques `try` con asignación de recursos. Estos se han explicado en el tema anterior dedicado a ficheros. Su uso automatiza el cierre de recursos que ya no se necesitan, sin necesidad de llamar explícitamente al método `close`, y garantiza que se cierran correctamente siempre, incluso cuando se producen excepciones.

No cerrar estos objetos hace que se desperdicie memoria y en general recursos limitados del sistema gestor de bases de datos. Puede incluso provocar que algunas operaciones se bloqueen o no se puedan completar con éxito. Especialmente en entornos empresariales complejos en los que diversos usuarios realizan simultáneamente diferentes operaciones con la base de datos, utilizando diversas aplicaciones.

2.6.1. Apertura y cierre de conexiones

Los *drivers* de JDBC contienen clases que implementan las interfaces de JDBC y están disponibles en ficheros de tipo `jar` que se pueden incluir como dependencias de Maven.

Se puede establecer una conexión mediante la clase `DriverManager`, con el método `getConnection`. A este se le pasa un URL de conexión, que contiene un identificador del tipo de base de datos, y los datos adicionales necesarios para acceder a ella, como por ejemplo usuario y contraseña.

Un URL de conexión tiene la siguiente forma para MariaDB, por ejemplo:

```
jdbc:mariadb://host:puerto/basedatos
```

Donde `host` es `localhost` si está en el mismo ordenador, y `puerto` suele ser 3306.

El método `getConnection` de `DriverManager` carga los drivers de JDBC disponibles y determina cuál es el apropiado para la base de datos indicada en el URL. Entonces, le pasa el URL y datos adicionales para que establezca una conexión. Si lo consigue, devuelve una `Connection` que se usará en adelante para trabajar con la base de datos.

El *driver* de JDBC para MariaDB se puede incluir con una dependencia de Maven, que se puede encontrar en el repositorio de Maven buscando por el texto "MariaDB Java Client" (figura 2.4). Por cierto, que en la descripción se dice que sirve para MariaDB y también para MySQL. MariaDB se creó para poder reemplazar a MySQL. Allá donde se pueda utilizar MySQL, debería poderse utilizar MariaDB en su lugar, y a la inversa. Una vez seleccionada una versión (relativamente reciente, y mejor con un número relativamente elevado de usos), se muestra el texto que hay que añadir en `pom.xml` para la dependencia (figura 2.5).

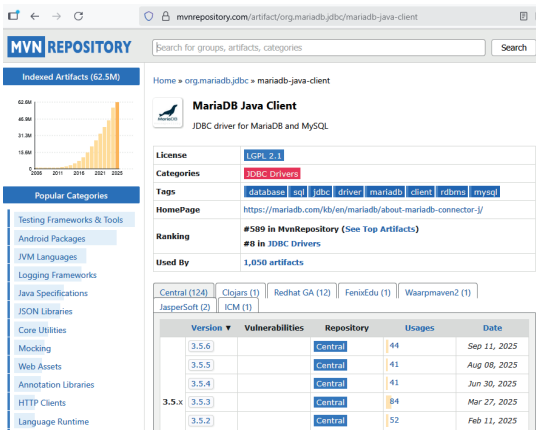


Figura 2.4
Repositorio de Maven con la información del *driver* JDBC para MariaDB

Figura 2.5
Dependencia para driver JDBC para MariaDB en el repositorio de Maven

Para los programas de ejemplo y actividades, a partir de ahora, se utilizará una base de datos vacía, `libro_ad`, a la que se accederá con un usuario `libro_ad`, con permisos para realizar cualquier operación en dicha base de datos.

Se accede al intérprete de línea de comandos de MariaDB con el ususario `root`.

```
$ mariadb -u root –p
```

Se crea la nueva base de datos `libro_ad`.

```
MariaDB [(none)]> create database libro_ad;
Query OK, 1 row affected (0,001 sec)
```

Se crea un usuario `libro_ad` con contraseña `libro_ad`.

```
MariaDB [(none)]> create user 'libro_ad'@'localhost' identified by 'li-
    bro_ad';
Query OK, 0 rows affected (0,013 sec)
```

Se otorgan todos los permisos a este usuario sobre la base de datos antes creada.

```
MariaDB [(none)]> grant all privileges on libro_ad.* to 'libro_ad'@'lo-
    calhost';
Query OK, 0 rows affected (0,005 sec)
```

Se termina la sesión como `root` con el comando `quit`.

```
MariaDB [(none)]> quit;
Bye
$
```

Se verifica que se puede acceder a la base de datos `libro_ad` con el usuario `libro_ad`.

```
$ mariadb -u libro_ad -p;
```

Y que se puede utilizar dicha base de datos.

```
MariaDB [(none)]> use libro_ad;
Database changed
```

Ahora se puede terminar la sesión con como usuario `libro_ad` con el comando `quit`.

```
MariaDB [libro_ad]> quit;
Bye
$
```

El siguiente programa abre una conexión a la base de datos creada anteriormente con el usuario creado anteriormente con permisos sobre dicha base de datos, y luego la cierra. El servidor de base de datos es un proceso que escucha en un puerto TCP de un *host*. Para la conexión hace falta indicar el servidor (*host* y puerto), el nombre de la base de datos y los datos de autenticación (usuario y contraseña).

Para abrir y cerrar los recursos (en este caso, la conexión con la base de datos) se utiliza un bloque `try` con recursos.

Las excepciones que se puedan producir en las operaciones con bases de datos utilizando JDBC serán de la clase `SQLException`. El método `muestraErrorSQL` muestra toda la infor-

mación relativa a una `SQLException`, y se usará sin cambios en los siguientes programas de ejemplo, por lo que su código se omitirá en ellos.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCConnection {

    public static void muestraErrorSQL(SQLException e) {
        System.out.printf("SQL ERROR mensaje: %s\n", e.getMessage());
        System.out.printf("SQL Estado: %s\n", e.getSQLState());
        System.out.printf("SQL código específico: %s\n", e.getErrorCode());
    }

    public static void main(String[] args) {
        String basedatos = "libro_ad";
        String host = "localhost";
        String port = "3306";
        String parAdic = "";
        String urlConnection = "jdbc:mariadb://" + host + ":" + port + "/" +
            basedatos + parAdic;
        String user = "libro_ad";
        String pwd = "libro_ad";

        try (Connection c = DriverManager.getConnection(urlConnection, user,
            pwd)) {
            System.out.printf("Conexión establecida con %s, BD %s, servidor %s
                versión %d.\n",
                    urlConnection,
                    c.getCatalog(), c.getMetaData().getDatabaseProductName(),
                    c.getMetaData().getDatabaseMajorVersion());
        } catch (SQLException ex) {
            muestraErrorSQL(ex);
        }
    }
}
```

```
Conexión establecida con jdbc:mariadb://localhost:3306/libro_ad, BD libro_ad,
    servidor MariaDB versión 10.
```

COMPLEMENTO DIGITAL 2.1

Recomendaciones

En la programación de bases de datos, existen un conjunto de buenas prácticas relacionadas con la seguridad, la mantenibilidad y la escalabilidad. En este recurso, disponible en la web de Síntesis (www.sintesis.com) y accesible con el código indicado en la página 9 del libro, podrás leer sobre ellas a partir del ejemplo anterior.

2.6.2. La interfaz `Statement`

La interfaz `Statement` se utiliza para ejecutar cualquier tipo de sentencia SQL. Se puede obtener un `Statement` mediante el método `getStatement()` de `Connection` (cuadro 2.2). En los siguientes apartados se explica cómo ejecutar distintos tipos de sentencias de SQL utilizando esta interfaz y, si se trata de una consulta, cómo recuperar los resultados en un `ResultSet`.

CUADRO 2.2. Métodos de `Statement`

Método	Funcionalidad
`ResultSet getResultSet()`	Ejecuta una consulta (sentencia `SELECT` de SQL) y devuelve un `ResultSet` que permite acceder a sus resultados.
`ResultSet executeQuery(String sql)`	Obtiene el conjunto de resultados de una sentencia `SELECT` y de otros tipos de sentencias que se verán más adelante, como procedimientos almacenados.
`int executeUpdate(String sql)`	Se utiliza para realizar operaciones que modifican los contenidos de la base de datos. A saber, sentencias `INSERT`, `UPDATE` y `DELETE`. Devuelve el número de filas afectadas.
`boolean execute(String sql)`	Se puede utilizar para ejecutar cualquier tipo de sentencia. Es el método que hay que utilizar preferentemente para sentencias de DDL (sublenguaje de SQL para definición de datos), tales como `CREATE`, `ALTER` y `DROP`. El valor devuelto depende de la sentencia ejecutada y de sus resultados.

2.6.3. Ejecución de sentencias de DDL

DDL es el lenguaje de definición de datos. Incluye sentencias para crear, modificar y borrar tablas, vistas y el resto de los objetos que pueden existir en una base de datos relacional.

Las sentencias de DDL se pueden ejecutar con el método `execute`.

El siguiente programa de ejemplo crea, utilizando SQL, una tabla para almacenar datos de clientes. Por supuesto, si se ejecuta una segunda vez, se producirá una excepción. Se utiliza un bloque de inicialización de recursos para `Connection` y para `Statement`, y se hará en general en adelante para todos los recursos de JDBC.

```
// Ejecución de sentencias de DDL con execute()

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBC_create_table {

    public static void main(String[] args) {
```

```
        (...) // Se omite declaración de variables para datos de conexión
        try (
            Connection c = DriverManager.getConnection(
                urlConnection, user, pwd);
            Statement s = c.createStatement()) {
            s.execute("create table clientes(dni char(9) not null, apellidos
                varchar(32) not null, cp char(5), primary key(dni))");
        } catch (SQLException ex) {
            muestraErrorSQL(ex);
        }
    }
}
```

Se puede verificar que se ha creado la tabla con el intérprete de SQL de MariaDB.

```
$ mariadb -u libro_ad –p

MariaDB [(none)]> use libro_ad
Database changed

MariaDB [libro_ad]> show tables
+--------------------+
| Tables_in_libro_ad |
+--------------------+
| clientes           |
+--------------------+
1 row in set (0,000 sec)

MariaDB [libro_ad]> desc clientes
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| dni       | char(9)     | NO   | PRI | NULL    |       |
| apellidos | varchar(32) | NO   |     | NULL    |       |
| cp        | char(5)     | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
3 rows in set (0,001 sec)
```

2.6.4. Ejecución de sentencias para modificar contenidos de la base de datos

Estas sentencias se pueden ejecutar con el método `executeUpdate`, que devolverá el número de filas afectadas por la operación, ya se trate de una sentencia `INSERT`, `UPDATE` o `DELETE`. Como ejemplo, el siguiente programa añade varias filas con datos de clientes con una sentencia `INSERT`.

```
// Ejecución de sentencias de modificación de datos con executeUpdate

import java.sql.Connection;
```