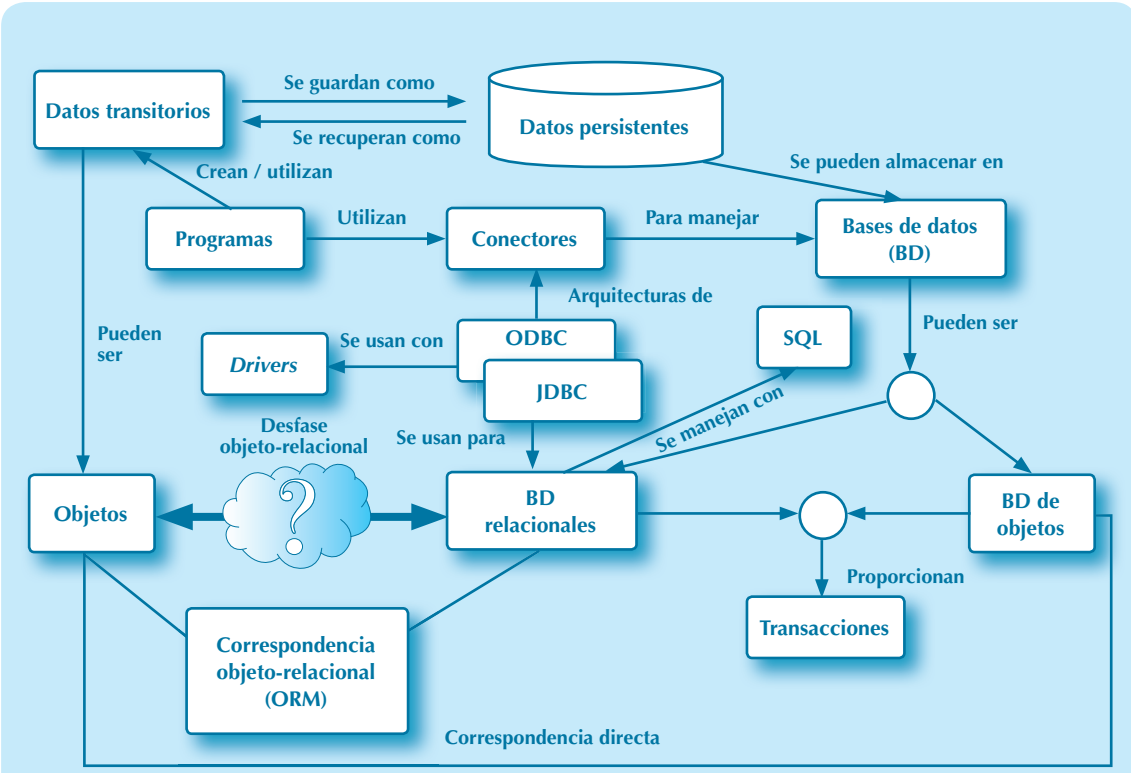
3

Bases de datos relacionales

Objetivos

- ✓ Conocer las características fundamentales de las API *(application programming interface)* que proporcionan conectores a bases de datos y, en particular, de JDBC (Java Database Connectivity) para bases de datos relacionales.
- ✓ Comprender el desfase objeto-relacional y, en particular, las dificultades para el almacenamiento de objetos complejos en estructuras de almacenamiento basadas en tablas.
- ✓ Abrir una conexión a una base de datos con JDBC utilizando un *driver* de JDBC para ella.
- ✓ Escribir programas en Java utilizando JDBC para ejecutar todo tipo de sentencias de SQL: de definición de datos (DDL) y de modificación y consulta de datos (DML).
- ✓ Utilizar sentencias preparadas para ejecutar sentencias de SQL de manera segura y eficiente.
- ✓ Trabajar con transacciones para ejecutar atómicamente un grupo de sentencias de SQL.

Mapa conceptual



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.

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

Desfase objeto-relacional. Conjunto de dificultades que plantea el almacenamiento de objetos complejos en bases de datos relacionales, con estructuras de almacenamiento basadas en tablas.

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.

Iterador. Mecanismo que permite acceder a los resultados de una consulta. Tiene operaciones para navegar por el conjunto de resultados y para recuperar los resultados uno a uno.

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.

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.

3.1. Conectores

Los sistemas gestores de bases de datos (SGBD) de distintos tipos (relacionales, de XML, de objetos o de otros tipos) 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 y se denominan *conectores*.

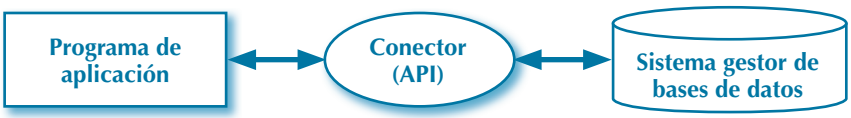


Figura 3.1
Interacción con bases de datos utilizando conectores

3.2. 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 tanto para las aplicaciones como para las distintas bases de datos, y los *drivers* se ocupan de las particularidades de las distintas bases de datos.

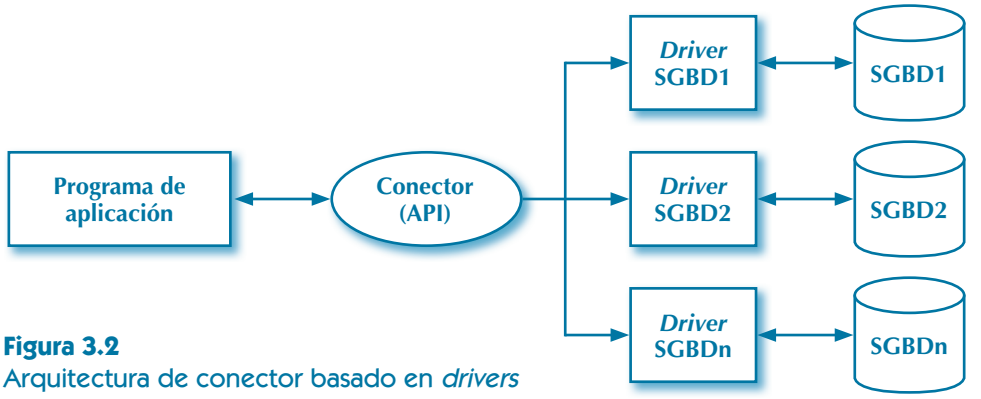


Figura 3.2
Arquitectura de conector basado en *drivers*

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). Existen *drivers* para ellas, no solo para bases de datos relacionales, sino también para formatos de ficheros tabulares como hojas de cálculo y ficheros CSV, e incluso para fuentes de datos no relacionales como documentos XML. Estos sucesores de ODBC son API nativas para Windows, y en entornos Windows se utilizan hoy como alternativa o evolución de ODBC. Pero en entornos Linux o Unix, ODBC sigue siendo la principal solución para acceso a bases de datos relacionales desde C.

PARA SABER MÁS

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.

Todas las bases de datos importantes proporcionan hoy en día *drivers* de ODBC y de JDBC. 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. Existen también API que proporcionan acceso directo a determinadas bases de datos, y están ganando importancia con las aplicaciones web, cada vez más frecuentes, que se ejecutan en un entorno de servidor, que tienen una interfaz de usuario basada en HTML, y a las que se accede a través de un navegador web. Un ejemplo es el lenguaje PHP, que se ejecuta en un módulo de un navegador web Apache, y que tiene API para acceder directamente a bases de datos MySQL, todo funcionando en Linux. Este entorno se denomina con las siglas LAMP (Linux, Apache, MySQL, PHP). Para el acceso a bases de datos MySQL desde PHP se dispone de la extensión MySQLi, que proporciona una API con dos interfaces distintas: una procedural y otra orientada a objetos. También existe para PHP una API de conectores con una interfaz orientada a objetos y basada en *drivers,* llamada PDO (Portable Data Objects), con *drivers* para MySQL, Oracle y PostgreSQL, entre otras.

Recursos web

www

Más información acerca de las API MySQLi y PDO para PHP en la web PHP.net:

http://php.net/manual/es/book.mysqli.php
http://php.net/manual/es/book.pdo.php

En cualquier caso, el uso de un conector para una base de datos relacional es siempre igual en lo esencial, y conforme al estándar X/Open SQL CLI.

3.3. 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. En el caso de 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.

```
Connection c = getConnection(datos de conexión)
Statement s = c.createStatement();
ResultSet rs = s.executeQuery("SELECT... ");
while(rs.next()) { //En rs están disponibles más resultados de la consulta
    String dato1 = rs.getString(1); // obtener String de primera columna
    int dato2 = rs.getInt(2); // obtener int de segunda columna
}
s.close();
c.close();
```

Figura 3.3
Obtención de resultados de una consulta mediante iteradores

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.

3.4. 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 1.9).

Para la persistencia de objetos complejos hay dos posibilidades. Una es utilizar bases de datos de objetos, que permiten almacenar directamente objetos. La otra, utilizar técnicas o herramientas de correspondencia objeto-relacional (ORM). Ambas se verán en temas posteriores.

3.5. 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

El siguiente enlace proporciona acceso a un sitio web de Oracle con información general, una breve introducción y tutoriales de JDBC:
http://docs.oracle.com/javase/tutorial/jdbc/index.html

3.6. Operaciones básicas con JDBC

En este apartado se verá en detalle cómo se pueden realizar distintos tipos de operaciones sobre bases de datos relacionales con JDBC. Se trata de ejecutar los principales tipos de sentencias de SQL. 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 siguiente esquema se resume todo lo necesario para ejecutar cualquier sentencia de SQL con JDBC.

<table>
<tr><td colspan="2"><code>Class.forName(nombre del driver); // No necesario desde JDBC 4.0 (Java SE 6)</code></td><td>Cargar driver</td></tr>
<tr><td colspan="2"><code>Connection c = DriverManager.getConnection(datos de conexión);</code></td><td>Crear 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 (sent. DML);</code>
(o bien)
<code>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>

3.6.1. Apertura y cierre de conexiones

Los *drivers* de JDBC están disponibles en ficheros de tipo `jar`. Se puede establecer una conexión mediante la clase `DriverManager`, con el método `getConnection(String URL_conexión)`. La URL de conexión contiene un identificador del tipo de base de datos y los datos

necesarios para establecer una conexión con ella. Para MySQL, por ejemplo, tiene la forma `jdbc:mysql:host:puerto/basedatos`, donde host es localhost si está en el mismo host y puerto suele ser 3306. Este método carga automáticamente en memoria las clases para los *drivers* de JDBC de versión 4.0 (incluida en Java SE 6) o posteriores disponibles. Una vez cargados, selecciona el apropiado para la base de datos indicada en la URL de conexión y le pasa esta para que establezca la conexión. Si el *driver* lo consigue, devuelve como resultado una `Connection` y desde entonces se encargará de todas las operaciones realizadas con ella.

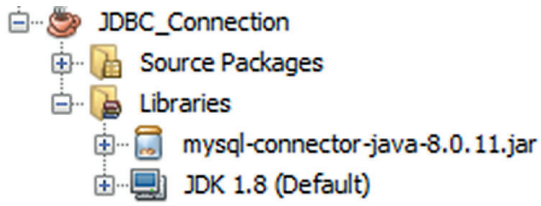


Figura 3.4
Driver de MySQL en proyecto de NetBeans

El *driver* para MySQL 8.0, por ejemplo, lo proporciona la clase `com.mysql.cj.jdbc.Driver`.

Para añadir un *driver* de JDBC en un IDE como NetBeans o Eclipse, se puede añadir el fichero `jar` al proyecto. Con el *driver* para MySQL 8.0 quedaría como en la figura 3.4.

Si el *driver* no se carga automáticamente en memoria, debe cargarlo el programa a la manera antigua:

```
Class.forName(nombre de la clase);
```

La clase que implementa el *driver* estará en el fichero `jar`. En caso de duda, hay que consultar la documentación del *driver,* que explicará también el formato de la cadena de conexión.

El siguiente programa abre una conexión a una base de datos de MySQL 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). Se pueden proporcionar en la URL parámetros de conexión adicionales. En el siguiente programa se utilizan variables para todos los datos de conexión, y con ellos se compone la URL de conexión. Para algunas no se indica un valor, o se indica el habitual. Los valores exactos dependen del entorno donde se ejecute el programa.

Para abrir y cerrar los recursos necesarios y poder realizar operaciones con JDBC se utiliza un bloque `try` con recursos. Esto evita tener que cerrar explícitamente recursos con `close()`, a la vez que asegura que se cierran de manera apropiada aunque ocurra alguna excepción.

Las excepciones que se puedan producir en las operaciones con bases de datos utilizando JDBC serán generalmente de la clase `SQLException`. El método `muestraErrorSQL` muestra toda la información relativa a una `SQLException`, y se usará sin cambios en los siguientes programas de ejemplo, por lo que su código se omitirá en ellos.

```
// Apertura y cierre de conexión con JDBC
package jdbc_connection;

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

```
public class JDBC_Connection {

public static void muestraErrorSQL(SQLException e) {
    System.err.println("SQL ERROR mensaje: " + e.getMessage());
    System.err.println("SQL Estado: " + e.getSQLState());
    System.err.println("SQL código específico: " + e.getErrorCode());
}

public static void main(String[] args) {

    String basedatos = "...";
    String host = "localhost";
    String port = "3306";
    String parAdic = "...";
    String urlConnection = "jdbc:mysql://"+host+":"+port+"/"+basedatos
    +parAdic;
    String user = "...";
    String pwd = "...";

//Class.forName("com.mysql.jdbc.Driver"); // No necesario desde SE 6.0
//Class.forName("com.mysql.cj.jdbc.Driver"); // para MySQL 8.0, no necesario

    try (Connection c=DriverManager.getConnection(urlConnection, user,
        pwd)) {
        System.out.println("Conexión realizada.");
    } catch (SQLException e) {
        muestraErrorSQL(e);
    } catch (Exception e) {
        e.printStackTrace(System.err);
    }
}
}
```

Recurso digital

En el anexo web 3.1, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás algunas recomendaciones.

3.6.2. 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`. 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 3.1
Métodos de `Statement`

Método	Funcionalidad
`ResultSet executeQuery(String sql)`	Ejecuta una consulta (sentencia SELECT de SQL), y devuelve un `ResultSet` que permite acceder a sus resultados.

[.../...]

CUADRO 3.1 (CONT.)

`ResultSet getResultSet()`	Obtiene el conjunto de resultados de una consulta 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 consulta. Es el método que hay que utilizar preferentemente para sentencias de DDL (sublenguaje de SQL para definición de datos), tales como CREATE, ALTER, DROP. El valor devuelto depende de la sentencia ejecutada y de sus resultados. Si se trata de una sentencia de DDL, devuelve `false`. Para ejecutar sentencias de DDL se podría utilizar también el método executeUpdate, y devolvería cero.
`void close()`	Cierra el Statement.

3.6.3. 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()`.

Como ejemplo, el siguiente programa crea, utilizando SQL, una tabla para almacenar datos de clientes. Por supuesto, si se ejecuta una segunda vez, se producirá una excepción. No es necesario llamar al método `close()` ni de `Statement` ni de `Collection`, porque se crean en la parte de inicialización de recursos del bloque `try`.

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

package JDBC_create_table;

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

public class JDBC_create_table {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los 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 e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

3.6.4. 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()
package JDBC_insert;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;
public class JDBC_insert {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión
        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            Statement s = c.createStatement()) {
            int nFil = s.executeUpdate(
                "INSERT INTO CLIENTES (DNI,APELLIDOS,CP) VALUES "
                + "('78901234X','NADALES','44126'),"
                + "('89012345E','HOJAS', null),"
                + "('56789012B','SAMPER','29730'),"
                + "('09876543K','LAMIQUIZ', null);");
            System.out.println(nFil + " Filas insertadas.");
        } catch (SQLException e) {
            muestraErrorSQL(e);
            System.err.println("SQL código específico: " + e.getErrorCode());
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.1

Haz un programa que haga los cambios necesarios para que los contenidos de la tabla CLIENTES sean los siguientes: ('78901234X', 'NADALES', '44126'), ('89012345E', 'ROJAS', null), ('56789012B', 'SAMPER', '29730'), partiendo de los contenidos de la tabla resultantes de la ejecución del programa anterior. El programa debe utilizar sentencias UPDATE y DELETE.

3.6.5. Ejecución de consultas y manejo de `ResultSet`

Las consultas son sentencias SELECT. Se pueden ejecutar con `executeQuery()`, que devolverá un `ResultSet` con sus resultados. Un `ResultSet` contiene los resultados de una consulta como un conjunto de filas, y mantiene internamente un cursor o puntero a la fila actual. Hay métodos de `ResultSet` para obtener los datos de las distintas columnas de la fila actual, tanto por posición (por ejemplo, tercera columna) como por nombre de la columna.

`Sentence` tiene un constructor sin parámetros. Los `ResultSet` que se obtienen de las `Sentence` creadas con este constructor solo se pueden recorrer empezando por el primer resultado y avanzando al siguiente con el método `next()`. El siguiente cuadro muestra los métodos disponibles para `ResultSet` de este tipo:

CUADRO 3.2
Métodos de `ResultSet` para consultar contenidos

Método	Funcionalidad
`boolean next()`	El cursor del `ResultSet` puede apuntar a cualquier fila suya. Puede, además, apuntar también a una posición especial justo antes de la primera fila y a otra posición especial justo después de la última fila. El cursor apunta inicialmente a la posición especial de antes de la primera fila. Este método mueve el cursor a la siguiente posición y devuelve `true`, a menos que el cursor esté en la última fila o en la posición especial tras ella, en cuyo caso devuelve `false`.
`getXXX(int)` `getXXX(String)`	Obtiene el contenido de la columna especificada de la fila actual. Se puede especificar una columna por su posición o por su nombre. Hay distintas funciones para distintos tipos: `getInt()`, `getString()`, `getDate()`, etc.
`close()`	Cierra el `ResultSet`. Debería hacerse siempre.

Como ejemplo, el siguiente programa muestra los datos de todos los clientes en la tabla. No hace falta utilizar el método `close()` ni del `Statement` ni del `ResultSet`, porque se han creado en la parte de inicialización de recursos del bloque `try`.

```
// Ejecución de una consulta con executeQuery()
package JDBC_select;

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

public class JDBC_select {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            Statement s = c.createStatement();
```

```
            ResultSet rs = s.executeQuery("SELECT * FROM CLIENTES")) {
            int i=1;
            while (rs.next()) {
                System.out.println("[" + (i++) + "]");
                System.out.println("DNI: " + rs.getString("DNI"));
                System.out.println("Apellidos: " + rs.getString("APELLIDOS"));
                System.out.println("CP: " + rs.getString("CP"));
            }
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.2

El código postal (columna CP) está definido con tipo CHAR(5) en la tabla CLIENTES, pero es siempre un número entero. ¿Se podría utilizar `getInt()` en lugar de `getString()` para recuperar su valor? Cambia el programa y verifica tu hipótesis, o justifica los resultados si no son los que esperabas.

Es posible obtener `ResultSet` que permitan una mayor libertad para navegar por sus contenidos. Este tipo de `ResultSet` se llama *scrollable* (que en inglés significa *enrollable* o *desplazable*). Para ello hay que utilizar constructores para `Statement` con parámetros adicionales. Todo lo que se explica a continuación es aplicable igualmente a `PreparedStatement`, que se verá en el siguiente apartado.

CUADRO 3.3
Métodos de `Connection` para obtener `Statement` de tipo *scrollable*

```
Statement createStatement(int tipo, int concurrencia)
PreparedStatement prepareStatement(String sql, int tipo, int
concurrencia).
```

Para el parámetro `tipo` se pueden especificar los siguientes valores:

- `ResultSet.TYPE_FORWARD_ONLY`: el tipo por defecto, utilizado hasta ahora.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: crea un `ResultSet` de tipo *scrollable* o desplazable, y en el que no se reflejan los cambios realizados por otros procesos en la base de datos.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: crea un `ResultSet` de tipo *scrollable* o desplazable, y sensible a cambios realizados por otros procesos en la base de datos. Quiere esto decir que si algún otro proceso modifica en la base de datos algunos de los datos recuperados en el `ResultSet`, las modificaciones se reflejarán automáticamente en los contenidos del `ResultSet`.

La cuestión es que, si se quiere un `ResultSet` de tipo *scrollable*, hay que decidir si se necesita que sea sensible a cambios realizados por otros procesos en la base de datos. Si no se necesita, lo mejor es seleccionar `ResultSet.TYPE_SCROLL_INSENSITIVE`. Pero, además, hay que decidir un valor para el parámetro concurrencia. Más adelante se verán sus posibles valores. Entre tanto, se puede usar el valor `ResultSet.CONCUR_READ_ONLY`.

CUADRO 3.4
Métodos disponibles adicionalmente para consultar contenidos de `ResultSet` *scrollable*

Método	Funcionalidad

`boolean previous()`
`boolean first()`
`boolean last()`
`void beforeFirst()`
`void afterLast()`
`boolean absolute(int pos)`
`boolean isFirst()`
`boolean isLast()`
`boolean isBeforeFirst()`
`boolean isAfterLast()`
`int getRow()`
	Movimiento del cursor a distintas posiciones y funciones que proporcionan información acerca de la posición del cursor. `previous()` es la recíproca de `next()`, es decir, mueve el cursor a la anterior posición y devuelve `true`, a menos que esté en la primera fila o en la posición especial anterior a ella. `first()` y `last()` mueven el cursor a la primera y última fila, y devuelven `true` a menos que no haya ninguna fila en el `ResultSet`. Los dos métodos siguientes mueven el cursor a las posiciones especiales antes de la primera fila y después de la última. `absolute()` mueve el cursor a la posición indicada. Las funciones cuyo nombre empieza por `is` indican si el cursor está en determinadas posiciones. `getRow()` devuelve la posición actual del cursor.

`getXXX(int)` `getXXX(String)`	Obtiene el contenido de la columna especificada de la fila actual, especificada por posición (`int`) o por nombre (`String`). Hay distintas funciones para distintos tipos: `getInt()`, `getString()`, `getDate()`, etc.
`close()`	Cierra el `ResultSet`. Debería hacerse siempre.

Actividades propuestas

3.3. Haz un programa que muestre los resultados de la misma consulta de SQL del programa anterior pero en orden inverso, del último al primero. La consulta de SQL debe ser la misma, sin ningún cambio.

3.4. ¿Cómo se podría averiguar el número de filas obtenidas por una consulta utilizando los métodos de `ResultSet`, pero sin un recorrer sus contenidos para contarlas? Escribe un programa que lo haga. Puedes emplear cualquier consulta.

Con lo visto hasta ahora se está en condiciones de ejecutar prácticamente cualquier sentencia de SQL y, en el caso de consultas, obtener todos sus resultados. Pero falta alguna cosa, como, por ejemplo, las funciones y procedimientos almacenados, que se verán más adelante.

En otro orden de cosas, faltan funcionalidades fundamentales de JDBC tales como sentencias preparadas, transacciones y alguna cosa más, que se verán a continuación. Por último, se verán algunas características algo más avanzadas de JDBC.

3.7. Sentencias preparadas

En los ejemplos vistos hasta ahora las sentencias de SQL eran fijas, estaban en cadenas de caracteres constantes. En una aplicación real suele ser necesario ejecutar sentencias de SQL en las que intervengan variables, bien porque dependan de valores introducidos desde la interfaz de usuario de una aplicación (por ejemplo, un formulario de consulta para clientes con diversos criterios de búsqueda, tales como DNI, nombre, etc.), bien porque dependan de un dato obtenido desde una fuente de datos (como, por ejemplo, un fichero o una consulta en SQL).

Se podría crear la sentencia en un `String`, y asignarle una expresión en la que intervengan variables. Por ejemplo: `String cons= "SELECT * FROM CLIENTES WHERE DNI= '"+dni+"'"`. Pero este planteamiento plantea graves problemas que obligan a descartarlo:

1. *Seguridad*. Una sentencia de SQL construida en tiempo de ejecución utilizando variables está expuesta a ataques mediante técnicas de inyección de SQL. Estas son técnicas para lograr que código SQL hábilmente introducido como parte del contenido de estas variables (`dni` en el ejemplo anterior) se ejecute. Esto permite al atacante consultar datos e incluso modificarlos y borrarlos. El riesgo es especialmente grande cuando los valores se introducen desde una interfaz de usuario, y mayor cuanto más amplia sea la posible base de usuarios. El peor escenario posible sería el acceso universal desde la web. Estos ataques se podrían evitar verificando el contenido de las variables, pero esto resulta engorroso y complejo, y no elimina completamente el riesgo.
2. *Rendimiento.* Una consulta que se envía al SGBD se compila antes de ejecutarse, es decir, se analiza y se crea un plan de ejecución para ella. Si entre una consulta y otra solo cambia el valor de algunas variables, se compilará cada consulta, para obtener siempre el mismo plan de ejecución.

Las sentencias preparadas permiten evitan estos problemas. Una sentencia preparada permite incluir marcadores (*placeholders* en inglés) en determinados lugares de la sentencia donde van valores que se proporcionarán en el momento de ejecutar la sentencia. Una sentencia preparada se proporciona al servidor de base de datos solo una vez, y este la prepara o precompila. A partir de ahí, solo hay que pasar un valor para cada marcador cada vez que se ejecuta la consulta.

Para hacer consultas con sentencias preparadas se utiliza `PreparedStatement`. Se obtiene un `PreparedStatement` con el método `getPrepraredStatement` de `Connection`. A este método se le pasa como parámetro la sentencia SQL, y se utiliza el carácter "?" como marcador *(placeholder)*.

En el cuadro 3.5 se incluyen los métodos importantes de `PreparedStatement` que no están en `Statement` o que tienen distintos parámetros.

CUADRO 3.5
Métodos de `PreparedStatement`

Método	Funcionalidad

`ResultSet executeQuery()`
`int executeUpdate()`
`boolean execute()`
	Estos tres métodos no tienen como parámetro la consulta, esta se le pasó al constructor.

[.../...]

CUADRO 3.5 (CONT.)

`setXXX(int pos, YYYY valor)`	Se utilizan para asignar un valor a un placeholder determinado, dado por su posición, siendo 1 la posición del primero. Los hay con distintos nombres, dependiendo del tipo.
`setNull(int pos, int tipoSQL)`	Asigna valor NULL a una columna. Debe indicarse el tipo. Los posibles tipos están definidos en la clase `java.sql.Types`.

El siguiente programa de ejemplo inserta en una tabla CLIENTES1 los datos de tres clientes utilizando una sentencia preparada. Para la inserción de los últimos registros, se ha utilizado un planteamiento que puede ser útil para hacer frente al problema que supone, para la mantenibilidad del código, el identificar por posición los parámetros cuando hay muchos. Si más adelante hay que añadir uno, esto obliga a cambiar sentencias de Java (y podrían ser muchas) solo para incrementar un número. Se evita esto utilizando una variable que se va incrementando. La operación de posincremento i++ en la última línea no es necesaria, pero evita el riesgo de que se añada un marcador después y se olvide añadir esta operación. Esto podría pasar, e incluso el programa podría funcionar aparentemente bien, y este error podría pasar desapercibido por un tiempo, durante el cual se introducirían datos incorrectos.

La tabla CLIENTES1 se puede crear en MySQL con: `CREATE TABLE CLIENTES1 (DNI CHAR(9) NOT NULL,APELLIDOS VARCHAR(32) NOT NULL, CP INTEGER, PRIMARY KEY(DNI));`

```
// Ejecución de varias sentencias INSERT de SQL con una sentencia preparada

package JDBC_prepared_statement;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Types;

public class JDBC_prepared_statement {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            PreparedStatement sInsert = c.prepareStatement("INSERT INTO
                CLIENTES1(DNI,APELLIDOS,CP) VALUES (?,?,?)")) {
            sInsert.setString(1, "78901234X");
            sInsert.setString(2, "NADALES");
            sInsert.setInt(3, 44126);
            sInsert.executeUpdate();
            int i = 1;
            sInsert.setString(i++, "89012345E");
            sInsert.setString(i++, "ROJAS");
            sInsert.setNull(i++, Types.INTEGER);
            sInsert.executeUpdate();
```

```
            i = 1; sInsert.setString(i++, "56789012B");
            sInsert.setString(i++, "SAMPER");
            sInsert.setInt(i++, 29730);
            sInsert.executeUpdate();
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.5

Escribe un programa que muestre los datos de varios clientes, o todos, de la tabla CLIENTES1. El programa debe utilizar una sentencia preparada para la consulta `SELECT * FROM CLIENTES1 WHERE DNI=?` Debe realizarse una consulta para cada cliente, especificando su DNI, y obtener los datos del `ResultSet` resultante, que solo tendrá una fila, al ser el acceso por clave primaria.

3.8. Transacciones

En este apartado se explica cómo utilizar JDBC para ejecutar varias sentencias de SQL como una transacción y gestionar los errores que se puedan producir durante su ejecución.

Una transacción es un conjunto de operaciones que se ejecutan conjuntamente como un todo, y de manera aislada de otras operaciones que pudiera realizar en paralelo otro proceso sobre los mismos datos. Las operaciones que componen una transacción se ejecutan todas completamente, con lo que todos sus cambios se confirman en la base de datos, o bien, si por cualquier motivo no se puede completar, la base de datos queda como si nunca se hubiera empezado a realizar la transacción. Las características de una transacción se resumen en inglés con el acrónimo ACID *(atomic, consistent, isolated, durable)*. En el primer capítulo se explicaron en detalle todas estas características, porque se pueden aplicar a cualquier tipo de base de datos, no solo relacionales, y de hecho existen para otros tipos de bases de datos.

Las transacciones son muy importantes y se utilizan muy frecuentemente. Todos los SGBD relacionales proporcionan soporte para transacciones, pero cada uno tiene sus particularidades. Algunos, como Oracle, las tienen habilitadas por defecto, de manera que ningún cambio se confirma en la base de datos hasta que no se ejecuta una sentencia que marca el fin de una transacción (normalmente COMMIT). Otros, como MySQL, las tienen deshabilitadas por defecto, de manera que los cambios realizados por cualquier sentencia se confirman automáticamente aunque no se ejecute una sentencia COMMIT. Además, cada una utiliza distintas sentencias, o con distinta sintaxis, para la gestión de transacciones. JDBC proporciona una interfaz común para todas las bases de datos. Por supuesto, también es posible ejecutar directamente las sentencias de SQL que controlan las transacciones, pero esto da como resultado una aplicación que solo funciona para una base de datos en particular.

Una transacción se realiza en SQL como sigue (se usa una sintaxis que no tiene por qué corresponder con la del SQL de ninguna base de datos en particular):

```
START TRANSACTION
Operación 1
Operación 2
...
COMMIT
```

Se puede abortar una transacción con la sentencia `ROLLBACK`. Con ello, se descartan todos los cambios, y todo queda como si la transacción nunca se hubiera iniciado.

La propia sentencia `COMMIT` podría fallar, aunque esto es muy inusual. Entonces se descartarían también todos los cambios, como si nunca se hubiera iniciado la transacción. También podría fallar la sentencia `ROLLBACK`.

La interfaz `Connection` tiene varios métodos para realizar estas operaciones y algunos más relacionados con transacciones. Aquí se verá lo necesario para una gestión básica de transacciones que, por otra parte, es más que suficiente en la inmensa mayoría de los casos.

CUADRO 3.6
Métodos de `Connection` relacionados con transacciones

Método	Funcionalidad
`void setAutoCommit(boolean autoCommit)`	Con `autoCommit=false` inicia una transacción. En este sentido, equivale a `START TRANSACTION`.
`void commit()`	Equivalente a una sentencia COMMIT de SQL. Si después de utilizar este método se quiere ejecutar más sentencias de SQL pero no en una transacción, se puede hacer `setAutoCommit(true)`.
`void rollback()`	Descarta todos los cambios realizados por la transacción actual. Se hará normalmente en respuesta a cualquier excepción de tipo `SQLException`. Esto significa que alguna sentencia de SQL no se ha ejecutado correctamente y entonces, normalmente, se querrá estar seguro de que se aborta la transacción, descartando todos los cambios.

Un fallo durante la ejecución de una sentencia de SQL provoca una excepción de tipo `SQLException`. En ese caso, normalmente se querrá abortar la transacción con `rollback()`.

Como ejemplo se muestra un programa que ejecuta varias sentencias de SQL como una transacción. No se puede agrupar, como en ejemplos anteriores, la creación de la conexión con la creación de la sentencia preparada en el mismo bloque de inicialización de recursos de un bloque `try`, porque estos no están disponibles en el bloque `catch` correspondiente, y entonces la conexión no estaría disponible para hacer `c.rollback()`. Además, hay que hacer `c.rollback()` en un bloque `try … catch`, porque puede lanzar una `SQLException`.

```
// Ejecución de varias sentencias en una transacción
package JDBC_transacciones;

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

```
import java.sql.SQLException;

public class JDBC_transacciones {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd)) {
            try (
                PreparedStatement sInsert = c.prepareStatement("INSERT INTO
                    CLIENTES1(DNI,APELLIDOS,CP) VALUES (?,?,?);")) {

                c.setAutoCommit(false);

                int i = 0;
                sInsert.setString(++i, "54320198V");
                sInsert.setString(++i, "CARVAJAL");
                sInsert.setString(++i, "10109");
                sInsert.executeUpdate();

                sInsert.setString(i = 1, "76543210S");
                sInsert.setString(++i, "MARQUEZ");
                sInsert.setString(++i, "46987");
                sInsert.executeUpdate();

                sInsert.setString(i = 1, "90123456A");
                sInsert.setString(++i, "MOLINA");
                sInsert.setString(++i, "35153");
                sInsert.executeUpdate();

                c.commit();

            } catch (SQLException e) {
                muestraErrorSQL(e);
                try {
                    c.rollback();
                    System.err.println("Se hace ROLLBACK");
                } catch (SQLException er) {
                    System.err.println("ERROR haciendo ROLLBACK");
                    muestraErrorSQL(er);
                }
            }
        } catch (Exception e) {
            System.err.println("ERROR de conexión");
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.6

Comprueba las diferencias entre el programa anterior, en el que se agrupan todas las inserciones de registros con transacciones, y otro programa igual pero sin transacciones. Primero hay que hacer una copia del programa y eliminar el código que gestiona las transacciones (`setAutoCommit`, `commit` y `rollback`). Después se trata de probar ambos programas con un mismo conjunto de datos inicial en la tabla CLIENTES1, para comprobar la manera distinta en que se comportan.

El conjunto de datos inicial y el programa podrían ser tales que las dos primeras inserciones se realizaran sin problemas, pero la última no, por haber ya en la tabla un cliente con ese DNI. El programa con transacciones no insertaría ningún registro. El programa sin transacciones insertaría registros hasta que se produjera un error. La creación del conjunto de datos inicial debe automatizarse. Se puede hacer mediante una secuencia de sentencias SQL, que se pueden guardar en un fichero de texto, y ejecutar con el intérprete de SQL, o mediante un programa. En esas secuencias podrían borrarse todos los contenidos de la tabla con una sentencia DELETE y añadirse varias filas con sentencias INSERT.

3.9. Valores de claves autogeneradas

Es una práctica muy habitual crear una tabla de manera que la clave primaria sea una columna numérica y se deje al sistema gestor de base de datos que proporcione un nuevo valor para cada nueva fila que se inserta. Un buen ejemplo es una aplicación que trabaja con facturas. Cada factura debe tener un identificador único, que normalmente es un simple número. El número asignado a una factura no tiene ninguna significación especial. Lo importante es que cada factura tenga un número distinto, y se deja al sistema que asigne un nuevo número consecutivo cada vez que crea una nueva factura. Las claves de este tipo se suelen llamar claves autoincrementales o, en JDBC, claves autogeneradas.

El mecanismo es esencialmente igual en casi todas las bases de datos, con pequeños cambios en la sintaxis del SQL utilizado. Oracle ha tenido históricamente una particularidad al respecto, y es el uso de secuencias para la generación de este tipo de identificadores. Las secuencias son objetos especiales que se crean en la base de datos para este fin. Pero en la versión 12 Oracle introdujo un mecanismo similar al habitual en otras bases de datos. A continuación, se muestra la definición de una tabla FACTURAS con clave autogenerada en MySQL y Oracle. Se define, además, una clave foránea sobre la columna DNI_CLIENTE para que solo se puedan crear facturas para clientes que existen en la tabla CLIENTES.

CUADRO 3.7
Definición de tabla FACTURAS con claves autogeneradas en MySQL y Oracle

MySQL

```
CREATE TABLE FACTURAS(
    NUM_FACTURA INTEGER AUTO_INCREMENT
        NOT NULL,
    DNI_CLIENTE CHAR(9) NOT NULL,
    PRIMARY KEY(NUM_FACTURA),
    FOREIGN KEY FK_FACT_DNI_CLIENTES
        (DNI_CLIENTE) REFERENCES
        CLIENTES(DNI)
);
```

Oracle 12 o posterior con IDENTITY

```
CREATE TABLE FACTURAS(
    NUM_FACTURA INTEGER NOT NULL
        GENERATED BY DEFAULT AS
        IDENTITY,
    DNI_CLIENTE CHAR(9) NOT NULL,
    CONSTRAINT PK_FACTURAS PRIMARY
        KEY(NUM_FACTURA),
    CONSTRAINT FK_FACT_DNI_CLIENTES
        FOREIGN KEY(DNI_CLIENTE)
        REFERENCES CLIENTES(DNI)
);
```

Aparte de la tabla FACTURAS, se utilizará en el ejemplo otra tabla LINEAS_FACTURA.

CUADRO 3.8
Definición de tabla LINEAS_FACTURA en MySQL y Oracle

MySQL

```
CREATE TABLE LINEAS_FACTURA(
    NUM_FACTURA INTEGER NOT NULL,
    LINEA_FACTURA SMALLINT NOT NULL,
    CONCEPTO VARCHAR(32) NOT NULL,
    CANTIDAD SMALLINT NOT NULL,
    PRIMARY KEY(NUM_FACTURA,
       LINEA_FACTURA),
    FOREIGN KEY FK_LINEAFACT_
       NUM_FACTURA(NUM_FACTURA)
       REFERENCES FACTURAS
       (NUM_FACTURA)
);
```

Oracle 12

```
CREATE TABLE LINEAS_FACTURA(
    NUM_FACTURA INTEGER NOT NULL,
    LINEA_FACTURA SHORTINTEGER NOT NULL,
    CONCEPTO VARCHAR2(32) NOT NULL,
    CANTIDAD SHORTINTEGER NOT NULL,
    CONSTRAINT PK_LINEAS_FACTURA
       PRIMARY KEY(NUM_FACTURA,
       LINEA_FACTURA),
    CONSTRAINT FK_LINEAFACT_NUM_FACTURA
       FOREIGN KEY(NUM_FACTURA)
       REFERENCES FACTURAS(NUM_FACTURA)
);
```

Ahora falta saber cómo se puede recuperar, con JDBC, el valor asignado a la clave autogenerada una vez que se ha insertado una nueva fila en una tabla que tiene una clave de este tipo. Esto se puede hacer utilizando algunos métodos de `Statement` y de `Connection` (para obtener `PreparedStatement`).

CUADRO 3.9
Métodos de `Statement` (y de `PreparedStatement`) para claves autogeneradas

Método	Funcionalidad

`int executeUpdate(`
`  String sql,`
`  int autoGeneratedKeys`
`)`
	`autoGeneratedKeys` puede tomar valores:
	• `Statement.RETURN_GENERATED_KEYS`
	• `Statement.NO_GENERATED_KEYS`
	Con el primero se pueden recuperar las claves autogeneradas como se explica a continuación. Normalmente será solo una. Este método no se puede utilizar con `PreparedStatement`. Una sentencia preparada debe crearse utilizando un método de Connection, como se explica más abajo

`ResultSet` `  getGeneratedKeys()`	Devuelve un `ResultSet` con los valores de las claves autogeneradas. Normalmente será solo uno. Se puede utilizar el mismo método con `PreparedStatement`.

CUADRO 3.10
Métodos de `Connection` relacionados con claves autogeneradas

Método	Funcionalidad

`PreparedStatement prepareStatement(`
`  String sql,`
`  int autoGeneratedKeys`
`)`
	Prepara una sentencia. Para poder recuperar valores para claves autogeneradas, hay que indicar el valor `PreparedStatement.RETURN_GENERATED_KEYS` para `autoGeneratedKeys`.

El siguiente programa de ejemplo crea una factura con varias líneas. Este ejemplo funciona sin cambios en MySQL, en Oracle 12 y en otras bases de datos que dispongan de un mecanismo similar para claves autogeneradas (todas en general). Si hay alguna diferencia, está en la creación de las tablas, y eso es otra cuestión. Una vez insertada una nueva factura en FACTURAS, inserta sus líneas en LINEAS_FACTURAS, e indica como valor para el número de factura el valor para la clave autogenerada que se acaba de crear para el número de factura. La creación de la factura y de todas sus líneas se realiza conjuntamente como una transacción.

```
// Recuperación de valores para claves autogeneradas. Creación de factura.

package JDBC_claves_autogeneradas;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBC_claves_autogeneradas {

public static void main(String[] args) {
    (...) // Se omite declaración de variables para los datos de conexión

    try (
        Connection c=DriverManager.getConnection(urlConnection, user, pwd)) {
        try (
            PreparedStatement sInsertFact = c.prepareStatement("INSERT INTO
                FACTURAS(DNI_CLIENTE) VALUES (?)",PreparedStatement.RETURN_
                GENERATED_KEYS);
            PreparedStatement sInsertLineaFact = c.prepareStatement("INSERT
                INTO LINEAS_FACTURA(NUM_FACTURA,LINEA_FACTURA,CONCEPTO,CANTIDAD)
                VALUES (?,?,?,?);");) {

            c.setAutoCommit(false);

            int i = 1;
            sInsertFact.setString(i++, "78901234X");
            sInsertFact.executeUpdate();
            ResultSet rs=sInsertFact.getGeneratedKeys();
            rs.next();
            int numFact=rs.getInt(1);

            int lineaFact = 1;
            i = 1;
            sInsertLineaFact.setInt(i++, numFact);
            sInsertLineaFact.setInt(i++, lineaFact++);
            sInsertLineaFact.setString(i++, "TUERCAS");
            sInsertLineaFact.setInt(i++, 25);
            sInsertLineaFact.executeUpdate();

            i = 1;
            sInsertLineaFact.setInt(i++, numFact);
            sInsertLineaFact.setInt(i++, lineaFact++);
            sInsertLineaFact.setString(i++, "TORNILLOS");
            sInsertLineaFact.setInt(i++, 250);
            sInsertLineaFact.executeUpdate();

            c.commit();
```

```
        } catch (SQLException e) {
            muestraErrorSQL(e);
            try {
                c.rollback();
                System.err.println("Se hace ROLLBACK");
            } catch (Exception er) {
                System.err.println("ERROR haciendo ROLLBACK");
                er.printStackTrace(System.err);
            }
                }
            } catch (Exception e) {
            System.err.println("ERROR de conexión");
            e.printStackTrace(System.err);
        }
    }
}
```

3.10. Llamadas a procedimientos y funciones almacenados

El estándar SQL incluye extensiones procedurales del lenguaje SQL, es decir, lenguajes de programación estructurados basados en SQL, pero que incluyen sentencias condicionales (IF), de asignación e iterativas (bucles). Los procedimientos y funciones almacenados son bloques de código escrito en un lenguaje de este tipo que pueden tener parámetros de entrada, de salida y de entrada-salida, y que, en el caso de las funciones, pueden devolver un valor. Son análogos a los procedimientos y funciones de cualquier lenguaje de programación estructurado (como Java). Se pueden invocar desde un intérprete de SQL y, como no, desde JDBC. Cada SGBD tiene su propio lenguaje de este tipo y su propia sintaxis para definir procedimientos y funciones almacenados, pero son muy similares. JDBC proporciona una interfaz única e independiente de la base de datos para utilizarlos: `CallableStatement`. En este apartado se explicará cómo utilizar esta interfaz, y se desarrollará un ejemplo completo para llamar a un procedimiento almacenado de MySQL.

El procedimiento que se va a utilizar se puede crear de la siguiente manera desde el intérprete de SQL. No es necesario entender los detalles de su implementación. Se trata de entender qué hace, cómo se le pasan valores para sus parámetros y cómo se obtienen los resultados.

```
DELIMITER //
CREATE PROCEDURE listado_parcial_clientes
(IN in_dni CHAR(9), INOUT inout_long INT)
BEGIN
    DECLARE apell VARCHAR(32) DEFAULT NULL;
    SELECT APELLIDOS FROM CLIENTES WHERE DNI=in_dni INTO apell;
    SET inout_long = inout_long + LENGTH(apell);
    SELECT DNI, APELLIDOS FROM CLIENTES
        WHERE APELLIDOS<=apell ORDER BY APELLIDOS;
END //
DELIMITER;
```

A este procedimiento se le pasa un DNI en el parámetro de entrada `in_dni`, y devuelve un conjunto de filas con DNI y APELLIDOS de los clientes por orden alfabético de APELLIDOS, hasta el valor de APELLIDOS para el cliente con el DNI proporcionado. Al valor pasado en el parámetro de entrada y salida `inout_long` se le suma la longitud del valor de APELLIDOS para el cliente con el

DNI pasado en `in_dni`. Este ejemplo tiene todo lo que puede tener un procedimiento almacenado: parámetros de entrada y de entrada-salida, y que devuelva una lista de filas. Lo único que podría faltarle sería un parámetro de solo salida, pero esto no supondría ninguna complicación adicional. Se puede utilizar este procedimiento desde el intérprete de SQL de la siguiente manera:

```
SET @long=0;
CALL listado_parcial_clientes('78901234X', @long);
SELECT @long;
```

Primero, se asigna un valor a la variable `@long`, que se pasa como parámetro de entrada y salida `inout_long`. La llamada al procedimiento con `CALL` muestra las filas devueltas por el procedimiento, y la sentencia `SELECT` del final muestra el valor devuelto en `@long`.

La secuencia de pasos para utilizar el procedimiento con JDBC es la misma, como se verá en breve en un programa de ejemplo. Lo primero es obtener un `CallableStatement` con el método `prepareCall(String patron_llamada)` de `Connection`. *`patron_llamada`* incluye el nombre del procedimiento o de la función almacenada. Su sintaxis tiene variantes según se trate de un procedimiento o de una función, y según tenga o no parámetros.

CUADRO 3.11
Sintaxis para *`patron_llamada`* en método `prepareCall (String patron_llamada)` de `CallableStatement`

	Con parámetros	Sin parámetros
Procedimiento	`{ call procedimiento(?, ?,... ) }`	`{ call procedimiento }`
Función	`{ ? = call función(?, ?,... ) }`	`{ call función }`

Los métodos de `CallableStatement` que permiten realizar la llamada y obtener los resultados se detallan en el cuadro 3.12.

CUADRO 3.12
Métodos de `CallableStatement` para uso de procedimientos y funciones almacenados

Método	Funcionalidad
`setXXX(int pos, YYYY valor)`	De manera similar a `PreparedStatement`, se utilizan para asignar un valor de diversos tipos (las hay con distintos nombres dependiendo del tipo) a un parámetro determinado, identificado por su posición, siendo 1 el primero.
`void registerOutParameter( int pos, int sqlType)`	Registra un parámetro de salida para poder obtener el valor devuelto en él.
`getXXX(int pos)`	Obtiene el valor de un parámetro de salida.
`ResultSet getResultSet()`	Obtiene el `ResultSet` resultado del procedimiento o función.

En este programa de ejemplo se llama al procedimiento almacenado anterior.

```
// Llamada a procedimiento almacenado en base de datos de MySQL

package JDBC_callable_statement;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.CallableStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class callable_statement {

    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd);
            CallableStatement s =
                c.prepareCall("{call listado_parcial_clientes(?,?)}")) {

            s.setString(1, "78901234X");
            s.setInt(2, 0);
            s.registerOutParameter(2,java.sql.Types.INTEGER);

            s.execute();

            ResultSet rs = s.getResultSet();

            int inout_long = s.getInt(2);
            System.out.println("=> inout_long: "+inout_long);
            int nCli=0;
            while (rs.next()) {
                System.out.println("[" + (++nCli) + "]");
                System.out.println("DNI: " + rs.getString("DNI"));
                System.out.println("Apellidos: " + rs.getString("APELLIDOS"));
            }
        } catch (SQLException e) {
            muestraErrorSQL(e);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.7

Crea una función almacenada con MySQL (o con Oracle o con otra base de datos) a la que se le pase el DNI de un cliente y que devuelva sus apellidos. Crea un programa en Java que realice una llamada a ella utilizando JDBC y escriba el valor devuelto por la función. Será necesario consultar documentación o investigar cómo crear funciones almacenadas en la base de datos utilizada, pero se hará de manera muy similar a un procedimiento almacenado.

3.11. Actualizaciones sobre los resultados de una consulta

La interfaz `ResultSet` no solo tiene métodos para consultar los contenidos de una base de datos, sino también para modificar esos contenidos. Con ellos se pueden eliminar, insertar y modificar filas en una tabla. Pero para ello es necesario crear `ResultSet` actualizables que permitan estas operaciones, y especificar algunos parámetros adicionales al crearlos.

Los `ResultSet` actualizables no se utilizan con mucha frecuencia. Normalmente, para hacer cambios en las bases de datos, se emplean sentencias de SQL `UPDATE`, `DELETE` e `INSERT`, y con `UPDATE` y `DELETE` las filas se seleccionan con una cláusula `WHERE`. Pueden ser útiles cuando las condiciones para seleccionar las filas para actualizar o borrar, o los cálculos para los nuevos valores que asignar a los atributos son muy complejos, o si no es posible, o es muy difícil, seleccionar las filas con SQL. Pero casi siempre es posible –y recomendable– utilizar sentencias de SQL separadas para modificaciones en bases de datos relacionales.

CUADRO 3.13
Métodos de `Connection` para obtener `Statement` y `PreparedStatement` actualizables

```
Statement createStatement(int tipo, int concurrencia)
PreparedStatement prepareStatement(String sql, int tipo, int concurrencia)
```

El parámetro tipo permite obtener un `ResultSet` de tipo *scrollable,* como ya se ha visto en un apartado previo, en el que se dejaron para más adelante las explicaciones sobre el parámetro concurrencia. Pues bien, este es el momento. Sus posibles valores son:

- `ResultSet.CONCUR_READ_ONLY`: el `ResultSet` no es actualizable, es decir, no permite que los cambios realizados en él se graben en la base de datos.
- `ResultSet.CONCUR_UPDATABLE`: el `ResultSet` es actualizable, es decir, los cambios realizados en él se pueden grabar en la base de datos.

En el cuadro 3.14 se muestran los métodos disponibles para los `ResultSet` actualizables.

CUADRO 3.14
Métodos disponibles para actualizaciones con `ResultSet` actualizables

Método	Funcionalidad
`void updateXXX(int pos, YYY valor)` `void updateXXX(String nombre, YYY valor)`	Asigna el valor indicado a la columna indicada, que se puede identificar, bien por posición, bien por nombre. XXX es un tipo de JDBC, YYY es un tipo de Java.
`void updateNull(int pos)` `void updateNull(String nombre)`	Asigna valor NULL a la columna indicada.
`void updateRow()`	Graba en la base de datos los contenidos de la fila actual del `ResultSet`.
`deleteRow()`	Borra de la base de datos la fila actual del `ResultSet`.

[.../...]

CUADRO 3.14 (CONT.)

`moveToInsertRow()`	Mueve el cursor a una posición especial, la fila de inserción *(insert row)*, que sirve para guardar los contenidos de una nueva fila que insertar en la base de datos. Una vez ejecutado este método, se asignan valores a las columnas con `updateXXX()` y finalmente se inserta la fila en la base de datos con `insertRow()`. Una vez hecho esto, se puede volver a la fila anterior con `moveToCurrentRow()`.
`void insertRow()`	Inserta los contenidos de la fila de inserción en el `ResultSet` y en la base de datos.

El siguiente programa de ejemplo utiliza las funciones anteriores para modificar el código postal del último cliente recuperado por una consulta, borrar el penúltimo cliente e insertar un nuevo cliente, todo dentro de una transacción.

```
// Modificación de contenidos de una tabla con un ResultSet actualizable

package JDBC_ResultSet_actualizable;

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

public class JDBC_ResultSet_actualizable {

public static void main(String[] args) {
    (...) // Se omite declaración de variables para los datos de conexión

        try (
        Connection c=DriverManager.getConnection(urlConnection, user, pwd)) {
            try (
                Statement sConsulta = c.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_UPDATABLE)) {

                ResultSet rs = sConsulta.executeQuery(
                    "SELECT * FROM CLIENTES WHERE CP IS NOT NULL");

                c.setAutoCommit(false);

                rs.last(); // Modifica último cliente
                rs.updateString("CP", "02568");
                rs.updateRow();

                rs.previous(); // Borra penúltimo cliente
                rs.deleteRow();

                rs.moveToInsertRow(); // Inserta nuevo cliente
                rs.updateString("DNI", "24862486S");
                rs.updateString("APELLIDOS", "ZURITA");
                rs.updateString("CP", "33983");
                rs.insertRow();

                c.commit();
```

```
            } catch (SQLException e) {
            muestraErrorSQL(e);
            try {
                c.rollback();
                System.err.println("Se hace ROLLBACK");
                } catch (Exception er) {
                    System.err.println("ERROR haciendo ROLLBACK");
                    er.printStackTrace(System.err);
                }
            }
        } catch (Exception e) {
            System.err.println("ERROR de conexión");
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.8

¿Existe alguna diferencia entre `updateXXX(1, null)` y `updateNull(1)`?

3.12. Ejecución de *scripts*

Un *script* de SQL es una secuencia de sentencias de SQL separadas por el carácter ";". Para poder ejecutar *scripts* con MySQL debe establecerse la conexión indicando en la URL de conexión la opción `allowMultiQueries=true`. Con ello se puede ejecutar un *script,* utilizando un `Statement`, exactamente igual que una sentencia de SQL individual. Los *scripts* de SQL son muy utilizados para la instalación de aplicaciones, para crear y poner a punto la base de datos con la que trabajan. Normalmente se prepara en un fichero de texto una secuencia de sentencias de SQL que crean una base de datos, y dentro de ella tablas, vistas y en general todos los objetos necesarios, y añaden algunos datos básicos en determinadas tablas.

3.13. Ejecución de sentencias por lotes

Para ejecutar un número muy grande de sentencias de SQL, puede ser conveniente hacerlo por lotes. Un lote es un conjunto de sentencias de SQL que se envían todas juntas al servidor de bases de datos y se ejecutan todas juntas, en lugar de enviar y ejecutar una a una.

No hay métodos para crear un lote, se crean automáticamente al ejecutar sentencias con el método `addBatch(String sql)` de `Statement` y `addBatch()` de `PreparedStatement`. En un lote creado para Statement las sentencias pueden ser de distinto tipo. En un lote creado para `PreparedStatement` son del mismo, porque el lote se crea con una sentencia preparada, pero cada ejecución se hace con un conjunto de parámetros distinto. El lote se ejecuta con `executeBatch()`, que devuelve un *array* de tipo `int[]` con el número de filas afectadas por cada sentencia del lote. Existe también un método `executeLargeBatch()` que devuelve un *array* de tipo `long[]`. También `CallableStatement` admite ejecución por lotes.

El siguiente programa crea un lote para una `PreparedStatement` de tipo `INSERT`, añade varias sentencias y lo ejecuta, todo dentro de una transacción. Se obtienen los datos de un *array* para ilustrar el hecho de que, normalmente, los datos se obtendrán de una fuente de datos mediante un bucle, dado que la ejecución por lotes tiene sentido cuando se agrupa un número grande de sentencias en el lote.

```
// Ejecución de un lote de sentencias preparadas
package JDBC_prepared_statement_en_lote;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class JDBC_prepared_statement_en_lote {
    public static void main(String[] args) {
        (...) // Se omite declaración de variables para los datos de conexión

        String[][] datosClientes = {
            {“13579135G”, “MOYA”, null},
            {“24680246G”, “SILVA”, “25865”},
            {“96307418R”, “TORRES”, “19273”}
            }
        try (
            Connection c = DriverManager.getConnection(urlConnection, user,
                pwd)) {
        try (
            PreparedStatement sInsert = c.prepareStatement(“INSERT INTO
                CLIENTES1(DNI,APELLIDOS,CP) VALUES (?,?,?)”)) {

            c.setAutoCommit(false);
            for (int nCli = 0; nCli < datosClientes.length; nCli++) {
                for (int i = 0; i < datosClientes[nCli].length; i++) {
                    sInsert.setString(i + 1, datosClientes[nCli][i]);
                }
                sInsert.addBatch();
            }
            sInsert.executeBatch();
            c.commit();
            } catch (SQLException e) {
                muestraErrorSQL(e);
            try {
                c.rollback();
            } catch (Exception er) {
                System.err.println(“ERROR haciendo ROLLBACK”);
                er.printStackTrace(System.err);
                }
            }
        } catch (Exception e) {
            System.err.println(“ERROR de conexión”);
            e.printStackTrace(System.err);
        }
    }
}
```

Actividad propuesta 3.9

No todas las bases de datos proporcionan soporte para lotes. Se puede saber con `soportaLotes = c.getMetaData().supportsBatchUpdates()`, siendo `c` una `Connection`. Crea un programa a partir de una copia del programa anterior que compruebe si la base de datos proporciona soporte para lotes y escriba esta información en pantalla. Si es el caso, agrupará los cambios en un lote, como hace el programa anterior. Si no, ejecutará directamente las sentencias preparadas. Si tu base de datos tiene soporte para lotes, prueba que funciona bien en ambos casos, por ejemplo, añadiendo solo para una prueba: `soportaLotes = false`.

Recurso digital

En el anexo web 3.2, disponible en www.sintesis.com y accesible con el código indicado en la primera página del libro, encontrarás información sobre *pool* de conexiones.

Resumen

- Un conector es una API que permite acceder, desde programas de aplicación, a los datos almacenados en SGBD.
- Cada tipo de base de datos tiene su propio mecanismo para almacenamiento de los datos y su propio lenguaje para trabajar con ella. Los conectores suelen ser para un tipo particular de bases de datos (relacional, de objetos, de XML). Pero tienen características comunes. Para empezar, proporcionan métodos para abrir una conexión a una base de datos. Una vez hecho esto, permiten ejecutar sentencias en el lenguaje propio de la base de datos (SQL en el caso de las bases de datos relacionales). Proporcionan iteradores para poder acceder a los resultados de una consulta. En concreto, para navegar por el conjunto de resultados y para obtener los datos de cada resultado individual. En el caso concreto de las bases de datos relacionales, las consultas devuelven un conjunto de filas, y los iteradores permiten navegar por el conjunto de filas y obtener los datos de cada fila particular. Si la base de datos proporciona soporte para transacciones, también proporcionan métodos para gestionarlas.
- La API JDBC para Java permite trabajar con bases de datos relacionales. Es más que una API, es una arquitectura, dado que proporciona una interfaz común para los programas de aplicaciones y para el acceso a bases de datos, y requiere *drivers* específicos para el acceso a las bases de datos particulares. Un *driver* hace posible la conexión a una base de datos particular utilizando sus propios protocolos de red e interfaces de bajo nivel, y proporciona la implementación para esa base de datos de todas las interfaces recogidas en la especificación JDBC.

- JDBC permite ejecutar cualquier sentencia de SQL mediante `execute`, `executeUpdate` y `executeQuery`. En el caso de consultas, permite obtener los resultados en un `ResultSet`.
- En el caso de que en la consulta intervengan variables, deben utilizarse siempre sentencias preparadas o `PreparedStatement` por seguridad (para evitar ataques de inyección de SQL). Deben utilizarse también, por eficiencia, siempre que se ejecuten muchas sentencias que son iguales salvo que cambien determinados valores de una a otra.
- JDBC permite ejecutar un conjunto de sentencias de SQL en una transacción de manera sencilla e independiente de la base de datos, y utiliza métodos estándares de JDBC en lugar de sentencias de SQL distintas según la base de datos.
- JDBC permite invocar procedimientos y funciones almacenados de manera independiente de la base de datos, utiliza métodos estándares, y hace abstracción de las diferencias entre distintas bases de datos en lo referente a procedimientos y funciones almacenados.
- JDBC permite ejecutar grupos de sentencias como *scripts,* la base de datos lo permite.
- JDBC permite ejecutar por lotes sentencias de SQL y sentencias de SQL preparadas, si la base de datos lo permite.

Ejercicios propuestos

Hay que utilizar transacciones para las modificaciones de datos siempre que sea apropiado. Para la realización de estos ejercicios puede ser necesario, como en los capítulos anteriores, consultar la documentación de Java SE 8 (https://docs.oracle.com/javase/8/docs/api/).

1. Haz un programa que permita navegar de forma interactiva por los contenidos de la tabla CLIENTES creada con un programa de ejemplo anterior. Primero, el programa debe realizar una consulta para obtener los contenidos de la tabla, y debe mostrar el mensaje "fila 1" y el contenido de la fila, indicando para cada columna el nombre de la columna y su valor. Después, se deben ejecutar los comandos que se vayan introduciendo por teclado. Si el comando es ".", debe terminar, por supuesto liberando todos los recursos. Si es "k", debe ir a la siguiente fila, indicar el número de la fila y mostrar sus contenidos, como al principio para la primera fila. El comando para ir a la fila anterior será "d". Si se introduce un número, se debe mostrar la fila en la posición indicada por el número. El programa debe mostrar mensajes apropiados en caso de que el comando que se ha introducido no se pueda realizar (por ejemplo, estando en la última fila se pide ir a la siguiente, o se introduce el número de una fila que no existe). La clase `Integer` tiene métodos que permiten determinar si un `String` representa un número entero.

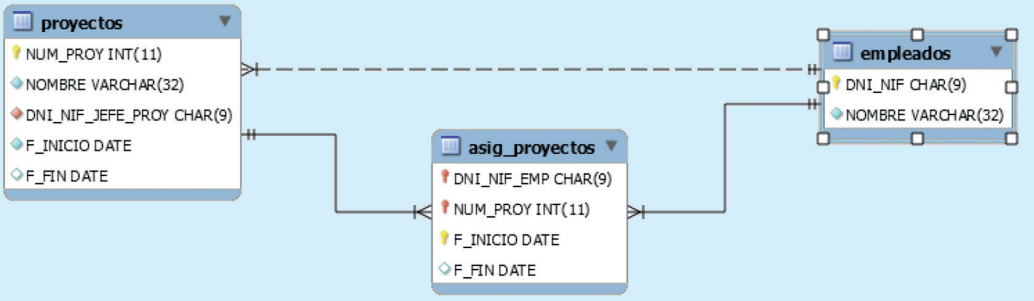
    Nota: se puede leer una cadena de caracteres desde el teclado de la siguiente forma: `BufferedReader br=new BufferedReader(new InputStreamReader(System.in)); String comando=br.readLine();`

2. Mejora el programa anterior para que se pueda utilizar con cualquier tabla, cuyo nombre se pedirá al iniciarse el programa. Será necesario obtener información acerca de la tabla mediante el método `getMetaData()` de `ResultSet`.

Para algunos ejercicios que siguen, debe utilizarse un conjunto de tablas para representar los proyectos que desarrolla una empresa, sus empleados y las asignaciones de empleados a proyectos. Las tablas se pueden crear en MySQL con las siguientes sentencias (una posibilidad es ejecutarlas con un programa Java con opción de conexión `allowMultiQueries=true`).

```
CREATE TABLE EMPLEADOS(DNI_NIF CHAR(9) NOT NULL, NOMBRE VARCHAR(32) NOT
    NULL, PRIMARY KEY(DNI_NIF));
CREATE TABLE PROYECTOS(NUM_PROY INTEGER AUTO_INCREMENT NOT NULL, NOMBRE
    VARCHAR(32) NOT NULL, DNI_NIF_JEFE_PROY CHAR(9) NOT NULL, F_INICIO
    DATE NOT NULL, F_FIN DATE, PRIMARY KEY(NUM_PROY), FOREIGN KEY
    FK_PROY_JEFE(DNI_NIF_JEFE_PROY) REFERENCES EMPLEADOS(DNI_NIF));
CREATE TABLE ASIG_PROYECTOS(DNI_NIF_EMP CHAR(9), NUM_PROY INTEGER NOT
    NULL, F_INICIO DATE NOT NULL, F_FIN DATE, PRIMARY KEY(DNI_NIF_EMP,
    NUM_PROY, F_INICIO), FOREIGN KEY F_ASIG_EMP(DNI_NIF_EMP) REFERENCES
    EMPLEADOS(DNI_NIF), FOREIGN KEY F_ASIG_PROY(NUM_PROY) REFERENCES
    PROYECTOS(NUM_PROY));
```

Para mayor claridad, el siguiente diagrama, obtenido con MySQL Workbench, muestra los esquemas de las tablas, incluyendo campos y claves primarias, y las relaciones entre tablas (claves foráneas).



3. Crea una clase `GestorProyectos` que contenga métodos para almacenar datos de empleados, proyectos y asignaciones de empleados a proyectos. La clase debe tener métodos para:

a) Crear un nuevo empleado (`nuevoEmpleado`). Debe devolver true si el empleado se creó correctamente. A este método se le pueden pasar todos los datos del empleado. Un valor `null` para alguno significa que no se especifica valor. No debe validar los datos que se le pasan. Si el valor indicado para alguno provoca que se lance alguna excepción, este método la propagará. Es decir, su definición debe incluir la opción `throws` con la clase de excepción correspondiente (al menos `SQLException`).

b) Crear un nuevo proyecto (`nuevoProyecto`). Debe devolver el número de proyecto (`NUM_PROY`). Como el método anterior, tiene parámetros para todos los datos del proyecto (un valor `null` significa que no se especifica valor), no valida los valores proporcionados para ellos, y propaga excepciones. Si se especifica `null` para `F_INICIO`, debe asignarse la fecha actual como fecha de inicio del proyecto, que en MySQL se puede obtener con la función `now()`. Un valor `null` para `F_FIN` significa que no está informada, y debe asignarse un valor `NULL` en la base de datos.

c) Asignar un empleado a un proyecto (`asignaEmpAProyecto`). Seguir para ello las mismas directrices que para los métodos anteriores, incluyendo las referentes a `F_INICIO` y `F_FIN`.

Debe probarse esta clase mediante un programa de prueba en el método `main()` que cree varios empleados y proyectos, y realice la asignación de algunos empleados a proyectos. No es necesario realizar ninguna verificación de fechas. Por ejemplo: que la fecha de inicio de una asignación de un empleado a un proyecto no es anterior a la fecha de inicio del proyecto, y otras similares. Pero por supuesto se pueden incluir como mejora. Una posibilidad es realizar estas verificaciones en la clase de Java. Otra es realizarlas mediante restricciones de integridad definidas en la propia base de datos o mediante *triggers*.

4. Se trata de hacer algo similar a lo hecho en el anterior ejercicio, pero con un planteamiento algo distinto. Hay que crear clases `Empleado`, `Proyecto` y `AsignacionEmpAProyecto`. Cada una de ellas debe tener un constructor sin parámetros y campos correspondientes a los campos de las correspondientes tablas en la base de datos. Deben tener métodos `getXXX()` y `setXXX()` para cada propiedad. Por ejemplo, para "Empleado" serían `String getDNINIF()`, `void setDNINIF(String DNINIF)`, `String getNombre()` y `void setNombre(String nombre)`. Aparte del constructor sin parámetros, deben tener uno con los parámetros correspondientes a los campos de la clave primaria. Por ejemplo: Empleado(`String DNINIF`), que lanzará una excepción `SQLException` si no existe en la base de datos una fila para los valores de atributos de la clave primaria proporcionados. Deben tener un método `save()` que guarde el objeto en la base de datos, con una sentencia `INSERT`, si no existe (en el caso de clientes, si no existe ninguno con el DNI), o que modifique los datos, con una sentencia `UPDATE`, si existe. Puedes considerar el uso de `INSERT... ON DUPLICATE KEY UPDATE` en MySQL, o sentencias similares en otras bases de datos.

5. Completa la clase `Proyecto` desarrollada en el ejercicio anterior con un método `getListAsigEmpleados()` que devuelva una lista con los empleados asignados actualmente al proyecto. Un empleado está asignado actualmente a un proyecto si existe para él una fila en `ASIG_PROYECTOS` con `F_INICIO` anterior a la fecha actual (que se puede recuperar en MySQL con `now()`) y con `F_FIN` no informada (es decir, con valor `NULL`) o posterior a la fecha actual.

6. Haz un programa para insertar en una tabla de clientes los datos leídos de un fichero en formato CSV, que debes preparar tú mismo. Los datos de cada cliente deben estar en una línea distinta, y como separador de campos puedes utilizar ";" o "|". Debe crearse una clase `LectorDatosClientes`, con un método `insertaDatosClientes (String nombreFichero, String nombreTabla, String separadorCampos)`.

La tabla debe tener la misma estructura que la tabla de ejemplo CLIENTES, y debe estar creada previamente a la ejecución del programa. El parámetro `separadorCampos` indica el separador de campos. Se puede leer el fichero línea a línea utilizando un `BufferedReader`. Se pueden obtener los campos de una línea utilizando la clase `StringTokenizer`, o bien con el método `split()` de `String`. Se debe utilizar una sentencia preparada (`PreparedStatement`) para las operaciones INSERT, y deben agruparse todas en una única transacción. Si el carácter separador aparece dos veces seguidas, eso significa que para un campo no se ha especificado un valor, y entonces se le debe asignar el valor `null`. Haz alguna prueba con ficheros cuyos contenidos puedan provocar errores, para asegurarte de que se da el mensaje de error apropiado y no se realiza ningún cambio, al estar todas las operaciones dentro de una transacción. Por ejemplo, especifica un valor nulo para DNI o valores incorrectos para algunos campos.

7. Haz una lista lo más completa posible con condiciones de prueba para el programa anterior. Para cada condición de prueba, indica el resultado esperado. Un ejemplo de condición de prueba sería: "Se especifica valor nulo para tal columna (que no admite valor nulo)", y resultado esperado: "Se muestra un mensaje de error apropiado, se termina la ejecución del programa y no se realiza ningún cambio en la base de datos". Otra sería: "Se especifica un valor alfabético para tal columna (que tiene valores numéricos)", y resultado esperado podría ser el mismo que para la anterior condición de prueba. Verifica cada condición de prueba con un fichero apropiado. No es necesario que el programa proporcione mensajes de error específicos, ni que en las condiciones de prueba se indique un error específico. Normalmente bastará con que el programa gestione las excepciones de tipo `SQLException` y muestre la información que proporciona esta clase de excepciones. Se recomienda redactar las condiciones de prueba sin tener el programa en mente, como lo haría alguien que no sabe cómo está hecho el programa, pero sí qué debe hacer.
8. Si para el ejercicio 6 no has utilizado lotes, modifica el programa para que los utilice. El lote se creará con el método `createBatch()` de `PreparedStatement`. Después se le añadirán todas las sentencias con `addBatch()` y por último se ejecutará el lote con `executeBatch()`.

ACTIVIDADES DE AUTOEVALUACIÓN

1. Un conector es una API que permite a las aplicaciones utilizar bases de datos, pero:
   - ☐ *a)* Solo utilizando el lenguaje Java.
   - ☐ *b)* Solo bases de datos relacionales.
   - ☐ *c)* Solo mediante *un driver* específico para cada base de datos.
   - ☐ *d)* Ninguna de las respuestas anteriores es correcta.

2. Parte de la problemática del desfase objeto-relacional consiste en que:
   - ☐ *a)* No siempre es fácil obtener los contenidos de una tabla desde un lenguaje orientado a objetos.

- ☐ *b)* No siempre es sencillo almacenar los datos contenidos en objetos complejos en un conjunto de tablas y, a la inversa, recuperar esos datos como objetos.
- ☐ *c)* Cuando se modifican objetos nunca se utilizan transacciones, pero cuando se modifican los contenidos de una base de datos relacional sí.
- ☐ *d)* Todas las respuestas anteriores son correctas.

3. Los *drivers* de JDBC:
   - ☐ *a)* Proporcionan acceso a una base de datos relacional particular, pero a cambio de utilizar SQL estándar, es decir, sin ninguna característica propia de la base de datos.
   - ☐ *b)* Solo están disponibles para bases de datos relacionales.
   - ☐ *c)* Proporcionan clases que implementan las interfaces de la especificación JDBC para una base de datos particular.
   - ☐ *d)* Se pueden comunicar directamente *con drivers* de ODBC para complementar sus funcionalidades.

4. La carga de *un driver* JDBC con `Class.forName(...)`:
   - ☐ *a)* No es necesaria a partir de Java SE 6.
   - ☐ *b)* Es imprescindible si se quiere que el programa funcione con versiones muy antiguas de Java.
   - ☐ *c)* Requiere que se indique el nombre de la clase que implementa el *driver*.
   - ☐ *d)* Todas las respuestas anteriores son correctas.

5. Para ejecutar una sentencia `INSERT` de SQL, lo más aconsejable es utilizar el método:
   - ☐ *a)* `executeQuery()`, porque siempre hay que utilizar este método para cualquier sentencia de SQL que afecte a un conjunto de filas.
   - ☐ *b)* `execute()`, porque una sentencia `INSERT` solo afecta a una fila, y entonces basta con un boolean para saber si la sentencia insertó una fila o no.
   - ☐ *c)* `executeUpdate`, que devuelve el número de filas insertadas.
   - ☐ *d)* `executeQuery`, que devuelve un `ResultSet` con la fila recién insertada.

6. Un `ResultSet`:
   - ☐ *a)* Permite obtener datos de la base de datos, pero no modificarlos.
   - ☐ *b)* Permite obtener datos de la base de datos, pero no modificarlos, a menos que sea de tipo *scrollable,* en cuyo caso solo se pueden recorrer secuencialmente empezando por la primera fila y avanzando una a una.
   - ☐ *c)* Permite obtener filas de tablas de la base de datos, modificar y borrar esas filas en la base de datos, y añadir nuevas filas en la base de datos.
   - ☐ *d)* Proporciona una imagen consistente de los contenidos de la base de datos en el momento de realizar la consulta. Esto significa que los cambios realizados por otros procesos en la base de datos nunca se reflejarán en sus contenidos.

7. A los valores para los campos de la fila actual de un `ResultSet`:
   - ☐ *a)* Se puede acceder tanto por posición como por nombre.
   - ☐ *b)* Se puede acceder por posición nada más.
   - ☐ *c)* Se puede acceder por nombre nada más.
   - ☐ *d)* Se puede acceder por posición, por nombre o tanto por posición como por nombre, según se especifique en los parámetros del método `getResultSet()`.

8. Si una sentencia de SQL solo se va a ejecutar una vez:
   - ☐ *a)* Da igual ejecutarla con un `Statement` o con un `PreparedStatement`, porque solo se ejecutará una vez y el rendimiento es igual.
   - ☐ *b)* Debe ejecutarse con `PreparedStatement` si para construir la sentencia con `Statement` se necesita utilizar alguna variable de programa. Ejecutar con `Statement` sentencias construidas utilizando variables de programa hace al programa vulnerable ante técnicas de inyección de SQL.
   - ☐ *c)* Debe ejecutarse siempre utilizando un `PreparedStatement` para evitar ataques por inyección de SQL.
   - ☐ *d)* Debe ejecutarse siempre utilizando un `PreparedStatement` si se trata de una sentencia que modifique los datos. Las consultas solo leen datos, por lo que no son vulnerables a ataques por inyección de SQL.

9. Un lote creado para un `PreparedStatement`:
   - ☐ *a)* Permite ejecutar sentencias de SQL que sería imposible ejecutar con un lote creado para un `Statement`.
   - ☐ *b)* Puede tener sentencias SQL de cualquier tipo, siempre que solo tenga sentencias de un tipo (SELECT, INSERT, UPDATE o DELETE).
   - ☐ *c)* Es para una única sentencia preparada.
   - ☐ *d)* Puede tener sentencias SQL de cualquier tipo, siempre que sean del mismo tipo y tengan el mismo número de marcadores *(placeholders)*.

10. Las claves autogeneradas:
    - ☐ *a)* Pueden ser de tipo tanto alfabético como numérico, pero para que puedan ser de tipo alfabético hay que utilizar secuencias.
    - ☐ *b)* Solo pueden ser de tipo numérico.
    - ☐ *c)* No existen en Oracle, en su lugar se usan secuencias.
    - ☐ *d)* Son valores que se generan para un campo clave y que se pueden recuperar con un método de `Connection`.

SOLUCIONES:

1.	a b c **d**	**5.**	a b **c** d
2.	a **b** c d	**6.**	a b **c** d
3.	a b **c** d	**7.**	**a** b c d
4.	a b c **d**	**8.**	a **b** c d
9.	a b **c** d		
10.	a **b** c d		