Tema 2

Persistencia en bases de datos relacionales

Acceso a Datos (Desarrollo de aplicaciones multiplataforma)
Carlos Alberto Cortijo Bon

Índice

1. JDBC

JDBC (*Java Database Connectivity*) es una API que permite a los programas de aplicación interactuar con sistemas gestores de bases de datos relacionales (SGBDR).

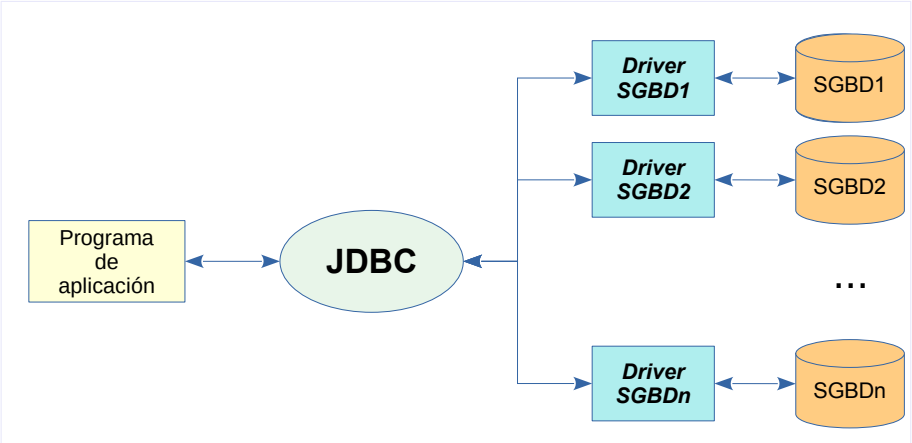


Ilustración 2.1: Interacción con bases de datos relacionales con JDBC

JDBC es parte de la biblioteca estándar de clases de Java, y sus clases e interfaces están en el paquete `java.sql`.

Incluye muchas interfaces cuyos métodos deben implementarse para realizar operaciones determinadas con sistemas gestores de bases de datos relacionales (SGBDR).

Un **driver de JDBC** para un SGBDR incluye una colección de clases que implementan las interfaces de JDBC para ese SGBD. Normalmente se proporciona en forma de fichero JAR.

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.

2. Operaciones con JDBC

Para realizar operaciones en una base de datos con JDBC, primero se debe establecer una conexión (**`Connection`**).

Una vez establecida la conexión, se debe crear una sentencia (**`Statement`**) o una sentencia preparada (**`PreparedStatement`**). Las sentencias de SQL se pueden ejecutar con métodos de estas clases.

Se pueden distinguir varios sublenguajes del lenguaje SQL. JDBC permite ejecutar sentencias de todos estos tipos, a saber:

- DDL (*Data Definition Language*, o lenguaje de definición de datos). Se pueden ejecutar con el método `execute()`.
- DML (*Data Manipulation Language* o lenguaje de manipulación de datos). Dentro del cual se pueden distinguir:
  - Sentencias para consulta. Las sentencias `SELECT` se pueden ejecutar con el método `executeQuery()`, que devuelve una lista de filas en un **`ResultSet`**, sobre el que se puede iterar para obtener los resultados uno a uno.

- Sentencias para creación, borrado y modificación de datos. Las sentencias `UPDATE`, `DELETE` e `INSERT` se pueden ejecutar con el método `executeUpdate()`, que devuelve el número de filas afectadas por la operación.

2.1. Prerrequisitos

Antes de poder utilizar programas en Java que se conecten y realicen operaciones con una base de datos, son necesarios los siguientes prerrequisitos:

- Instalación de un servidor de bases de datos MySQL.
- Creación de una base de datos en el servidor de MySQL.
- Descarga del driver de JDBC para el servidor de bases de datos MySQL.

A continuación se explican las operaciones que hay que realizar para disponer de estos prerrequisitos.

Instalación de servidor MySQL y creación de base de datos para pruebas

Se necesita una base de datos para los siguientes programas. Todos ellos, en general, establecerán una conexión con un servidor de base de datos para el acceso a una base de datos existente en él, realizarán determinadas operaciones, y cerrarán la conexión.

La interacción con diferentes sistemas gestores bases de datos (SGBD) es igual cuando se utiliza JDBC. Esa es la idea, precisamente. Pero la instalación del SGBD y la creación de una base de datos es diferente para cada una, y además varían según el sistema operativo.

En lo sucesivo, se asume que se trabaja con un servidor de bases de datos MySQL instalado en la máquina local. Si el servidor está en otra máquina, habrá que cambiar `localhost` por el nombre de la máquina o su dirección IP en lo sucesivo.

En Linux se puede instalar el servidor de bases de datos MySQL en la máquina local con el siguiente comando.

```
$ sudo apt install mysql-server
```

Ahora se procederá a la creación de una nueva base de datos en el servidor para ejecutar los programas de prueba, y se creará un nuevo usuario para acceder a ella.

Pero antes de nada, se cambiarán algunos aspectos de la configuración de seguridad de MySQL. Estos tienen como objetivo disponer de un entorno más cómodo de utilizar para desarrollo. Por supuesto, para un entorno real de producción sería conveniente una configuración distinta.

En particular, se permitirán contraseñas más sencillas y, por lo tanto, menos seguras. Y se desligará la autenticación de los usuarios de la base de datos de la de los usuarios del sistema operativo.

Para ello, se ejecuta `mysql_secure_installation`, y se procede como se muestra a continuación. Se muestra en negrita lo que debe escribirse como usuario.

```
$ sudo mysql_secure_installation

Securing the MySQL server deployment.

Connecting to MySQL using a blank password.

VALIDATE PASSWORD COMPONENT can be used to test passwords
and improve security. It checks the strength of password
and allows the users to set only those passwords which are
secure enough. Would you like to setup VALIDATE PASSWORD component?
```

```
Press y|Y for Yes, any other key for No: n
```

A continuación, se pide la contraseña del usuario `root` de la base de datos. Pero se produce un error al intentar cambiarla, y se explica que no tiene sentido cambiarla en este contexto.

```
Please set the password for root here.

New password:

Re-enter new password:
... Failed! Error: SET PASSWORD has no significance for user
'root'@'localhost' as the authentication method used doesn't store
authentication data in the MySQL server. Please consider using ALTER USER
instead if you want to change authentication parameters.


New password:
```

Es un tanto extraño, porque se entra en un bucle en el que se pregunta una contraseña y se responde que no se puede establecer la contraseña. Este bucle no se puede interrumpir. Pero ya se ha conseguido lo que se quería, deshabilitar el componente de validación de contraseñas, para poder utilizar contraseñas sencillas en este entorno de desarrollo. Por tanto, se puede terminar la sesión, aunque sea abruptamente.

Ahora se procede a cambiar la contraseña para el usuario `root`. Para ello se ejecuta el cliente `mysql` como superusuario, con `sudo`. Se puede poner como contraseña la que se quiera.

```
$ sudo mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.33-0ubuntu0.20.04.2 (Ubuntu)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> alter user 'root'@'localhost' identified with mysql_native_password by
'(contraseña)';
Query OK, 0 rows affected (0,08 sec))

mysql> quit
Bye
```

Nota: es importante fijarse en la versión del servidor de bases de datos de MySQL. El driver de JDBC que se descargará más adelante debe ser de la misma versión. En este caso, es la versión 8.

Ahora se accede al servidor de MySQL con el usuario `root`, con la contraseña que se le acaba de asignar, para crear una nueva base de datos de pruebas que se utilizará durante el resto del tema, y un usuario con permisos sobre esta base de datos.

```
$ mysql -u root -p
```

Con esto se accede a un intérprete de línea de comandos. Ahora se creará una base de datos `pruebasprog`, y un usuario `usupruebasprog` con contraseña `usupruebasprog`. Se puede hacer con los siguientes comandos.

Nota: es importante que el usuario que se va a crear, que es para pruebas y para realizar prácticas y

también ejercicios de exámenes, tenga ese nombre y esa contraseña, para facilitar su revisión y corrección. De nuevo, se trata de un entorno para pruebas, aprendizaje y evaluación, y la prioridad es la sencillez y la facilidad de uso, no la seguridad.

```
mysql> create database pruebasprog;
```

```
mysql> create user 'usupruebasprog'@'localhost' identified with
mysql_native_password by 'usupruebasprog';
```

```
mysql> grant all privileges on pruebasprog.* TO 'usupruebasprog'@'localhost';
```

Ahora se puede terminar la sesión con el usuario root con el siguiente comando.

```
mysql> quit;
```

Se puede verificar que se puede acceder a la base de datos pruebasprog con el usuario usupruebasprog con el siguiente comando.

```
$ mysql -u usupruebasprog -p
```

Una vez en el intérprete de línea de comandos, se puede verificar que se tiene acceso a la base de datos con los siguientes comandos.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| performance_schema |
| pruebasprog        |
+--------------------+
3 rows in set (0,03 sec)
```

```
mysql> use pruebasprog
Database changed
```

Para terminar, se cierra la sesión con el servidor de bases de datos.

```
mysql> quit;
```

Descarga del driver de JDBC para MySQL

El driver de JDBC para MySQL está disponible en https://dev.mysql.com/downloads/connector/j/

Nota: es importante fijarse en la versión del software que se va a descargar, que tiene que coincidir con la del servidor de MySQL. En este caso se puede ver que es la versión 8 (Connector/J 8.1.0), que coincide con la del servidor de MySQL previamente instalada. En caso de duda, se puede verificar la versión del servidor de bases de datos accediendo a él desde línea de comandos.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.34-0ubuntu0.22.04.1 (Ubuntu)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select version();
+-------------------------+
| version()               |
+-------------------------+
| 8.0.34-0ubuntu0.22.04.1 |
+-------------------------+
```

Hay que seleccionar la opción “Platform Independent”. El driver se puede descargar en un fichero comprimido de tipo tar o zip, el que se prefiera.

En la siguiente pantalla, se pueden ignorar los dos botones grandes que salen y descargar directamente el driver con el enlace “No thanks, just start my download”.

De todos los contenidos del fichero zip descargado, solo interesa un fichero de tipo jar que es el que contiene el driver de JDBC. Su nombre también incluye la versión.

3. Apertura y cierre de conexiones

Como ya se ha visto, los drivers de JDBC están disponibles en ficheros de tipo `jar`. Se puede establecer una conexión con el método `static Connection getConnection(String urlCon)` de la clase

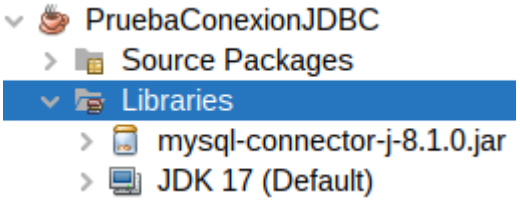
`DriverManager`. El URL de conexión `urlCon` contiene un identificador del tipo de base de datos y los datos necesarios para establecer una conexión con un servidor para ese tipo. Para MySQL tiene el formato

`jdbc:mysql:`*`host`*`:`*`puerto`*`/`*`basedatos`*

donde:

- *`host`* es el servidor, que suele ser `localhost` si está en el mismo *host*.
- *`puerto`* suele ser 3306.

El método `getConnection` carga, de entre los *drivers* de JDBC disponibles, el apropiado para la base de datos indicada en el URL de conexión, y se lo pasa 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.



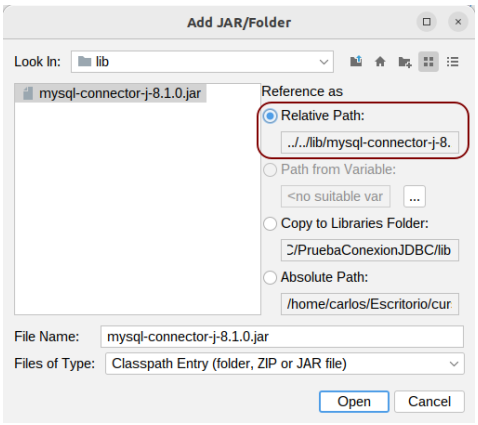
Para utilizar un driver de JDBC en un proyecto, debe añadirse el fichero jar al proyecto.

Aquí se muestra un proyecto de NetBeans para el primer programa de prueba que, simplemente, abrirá y cerrará una conexión con la base de datos para pruebas.

IMPORTANTE

El *path* con el que se añade el fichero jar del driver de JDBC al proyecto debe ser relativo, para que el proyecto pueda funcionar en cualquier ordenador.

Para que el proyecto pueda funcionar en cualquier ordenador, debe suministrarse en un fichero comprimido en el que se incluya tanto el proyecto de NetBeans como un directorio llamado `lib` que contiene el fichero jar del driver.



A continuación se proporciona el código fuente de este primer programa de ejemplo, que simplemente abre una conexión con la base de datos de pruebas del servidor de bases de datos MySQL, muestra información acerca del servidor y de la base de datos, y luego cierra la conexión. El método `muestraErrorSQL` muestra información detallada de error en caso de que se produzca alguno. Esta información se obtiene con métodos de la clase `SQLException`. El significado de esta información depende del servidor de bases de datos en particular. Para conocer el significado de algunos mensajes o códigos de error puede ser necesario consultar su documentación.

Para que este programa funcione, se asignan los valores apropiados a `basedatos` (nombre de la base de datos), `user` (nombre del usuario con el que se accede a la base de datos) y `pwd` (contraseña del usuario).

La conexión se abre en un bloque de inicialización de recursos. Esto garantiza que se invoca siempre el método `close()` para cerrarla, incluso aunque se produzca una excepción.

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

public class PruebaConexionJDBC {

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

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

4. La interfaz `Statement`

La interfaz `Statement` tiene métodos que permiten la ejecución de cualquier tipo de sentencia de SQL. El método `createStatement()` de `Connection` devuelve un `Statement`. La manera de ejecutar la sentencia y, en su caso, de obtener los resultados, será distinta según el tipo de sentencia.

Cuadro 2.1: Métodos de la interfaz `Statement` preferibles para distintos tipos de sentencias de SQL

Sentencias de DDL: **CREATE**, **ALTER**, **DROP**	`boolean execute(String sql)`
	Devuelve `false` para este tipo de sentencias.
Sentencias que modifican los contenidos de la base de datos: **INSERT**, **UPDATE** y **DELETE**.	`int executeUpdate(String sql)`
	Devuelve el número de filas afectadas (insertadas, cambiadas o borradas, respectivamente).
Consultas: **SELECT**.	`ResultSet executeQuery(String sql)`
	Devuelve un `ResultSet` que permite acceder a los resultados.

5. Ejecución de sentencias de DDL

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

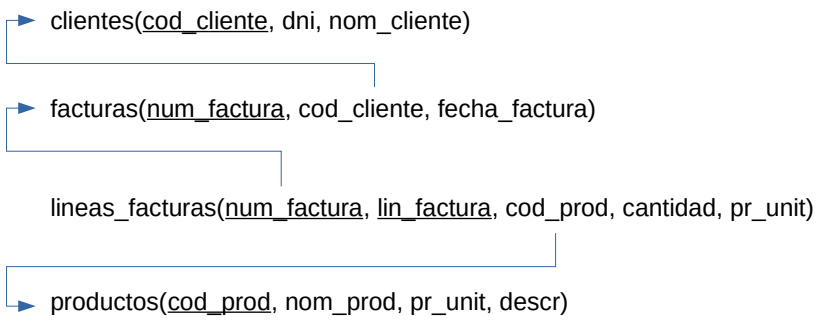
El siguiente programa de ejemplo crea una tabla para almacenar datos de clientes. Por supuesto, si se ejecuta una segunda vez, se producirá una `SQLException`. No es necesario llamar al método `close()` ni de `Statement` ni de `Connection`, porque se crean en la parte de inicialización de recursos del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

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

public class EjecucionSentenciaDDL {

    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(cod_cliente integer not null, dni char(9)
not null, nom_cliente varchar(40) not null, primary key(cod_cliente))");
        } catch (SQLException e) {
            muestraErrorSQL(e);
        }
    }
}
```

El método `muestraErrorSQL` es el mismo utilizado en el ejemplo anterior, y se utilizará en los ejemplos siguientes.

Actividad 2.1

El grafo relacional para la base de datos que se va a crear es el siguiente. En él se muestran las tablas, sus claves primarias, y las claves foráneas.



A continuación se proporcionan las sentencias de SQL con las que se pueden crear estas tablas con sus restricciones de integridad, incluyendo claves primarias y claves foráneas.

```
create table clientes(cod_cliente integer not null, dni char(9) not null, nom_cliente
varchar(40) not null, primary key(cod_cliente));
create unique index i_clientes_dni on clientes(dni);

create table productos(cod_prod integer not null, nom_prod varchar(40) not null,
```

```
pr_unit decimal(7,2) not null, descr varchar(120), primary key(cod_prod));


create table facturas(num_factura int not null auto_increment, cod_cliente integer not
null, fecha_factura date not null, primary key(num_factura), foreign key
fk_fact_cli(cod_cliente) references clientes(cod_cliente));

create table lineas_facturas(num_factura integer not null, lin_factura integer not
null, cod_prod integer not null, cantidad decimal(10,3) not null, pr_unit decimal(9,3)
not null, primary key(num_factura, lin_factura), foreign key
fk_linfact_numfact(num_factura) references facturas(num_factura), foreign key
fk_linfact_codprod(cod_prod) references productos(cod_prod));
```

Crear un programa que cree este esquema relacional. Las sentencias de SQL deben estar en un *array*: `String[] sentencias = { ... }`. Si una tabla ya existe (como será el caso de la tabla `clientes`, creada en una actividad anterior), fallará la creación de la tabla y se mostrará información al respecto, pero se seguirá con las sentencias siguientes.

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

Las sentencias que modifican los contenidos de la base de datos (`INSERT`, `UPDATE` o `DELETE`) se pueden ejecutar con el método `executeUpdate()`, que devuelve el número de filas afectadas por la operación (insertadas para `INSERT`, modificadas para `UPDATE`, o borradas para `DELETE`).

Como ejemplo, el siguiente programa añade varias filas a una tabla con una sentencia `INSERT`. La llamada a `executeStatement` devuelve el número de filas insertadas.

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

public class EjecucionSentenciaModifDatos {

    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 (cod_cliente, dni,nom_cliente) values"
               + "(1, '12345678A','astorga'),"
               + "(2, '23456789B','benítez'),"
               + "(3, '34567890C','cervera')");
            System.out.printf("%d filas insertadas.\n", nFil);
        } catch (SQLException e) {
            muestraErrorSQL(e);
        }
    }
```

```
}
```

Actividad 2.2

Crear un programa que cree al menos 3 productos en la tabla `productos`. Los valores para el campo `cod_prod` deben ser consecutivos empezando con 1.

Actividad 2.3

Crea un programa que pase a mayúsculas todos los nombres de clientes, utilizando una sentencia `update` de SQL. Al final, debe escribir el número de filas que han cambiado al ejecutar la sentencia. Si no sabes cuál es la función de MySQL que pasa una cadena de caracteres a mayúsculas, consulta la documentación.

7. Ejecución de consultas y recuperación de los resultados

Las sentencias `SELECT` se pueden ejecutar con `executeQuery()`, que devuelve los resultados de la consulta en un `ResultSet`. La interfaz `ResultSet` tiene métodos que permiten iterar sobre los resultados de la consulta, que son una lista de filas. Mantiene un cursor que apunta a la fila actual, y tiene métodos que permiten obtener el valor de cada una de las columnas de la fila actual. En el siguiente cuadro se muestran algunos de los métodos más importantes de `ResultSet`.

Cuadro 2.2: Métodos de la interfaz ResultSet

`boolean next()`	El cursor interno del `ResultSet` puede estar situado entre dos filas cualesquiera, o bien en una de las siguientes dos posiciones especiales: antes de la primera fila o después de la última fila. Inicialmente, el cursor interno del `ResultSet` está situado en la posición especial antes de la primera fila. Con el método **`next()`**, se obtiene una fila, se avanza el cursor a la siguiente posición, y se devuelve el valor `false`, a menos que esté en la posición especial de después de la última fila, en cuyo caso el cursor se queda en esa posición y se devuelve `true`. → Fila 1 → Fila 2 ... → Fila n-1 → Fila n →
`getXXX(int columnIndex)` `getXXX(String columnLabel)`	Obtienen el contenido de la columna especificada de la última fila obtenida del `ResultSet`. Se puede especificar la columna de una de dos maneras: • Con un número, que indica la posición de la columna dentro de la lista de columnas recuperadas en el `ResultSet`. • Con un `String`, que indica el nombre de la columna.

El siguiente programa de ejemplo muestra los datos de todas las filas de una tabla.

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

public class EjecucionSentenciaModifDatos {
```

```
    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 cod_cliente,dni,nom_cliente from clientes")) {
            int i = 1;
            System.out.printf("  (%s, %s, %s)\n", "cod_cliente", "dni", "nom_cliente" );
            while (rs.next()) {
                System.out.printf("%2d: [%s, %s, %s]\n", i++,
                  rs.getInt("cod_cliente"),
                  rs.getString("dni"),x
                  rs.getString("nom_cliente"));
            }
        } catch (SQLException e) {
            muestraErrorSQL(e);
        }
    }
}
```

Actividad 2.4

Crear una clase `Cliente` cuyos atributos se correspondan con las columnas de la tabla `clientes` de la base de datos. Tanto el nombre de la clase como sus atributos, por supuesto, deben ser conformes con los convenios de notación habituales en Java. Añade a esta clase un método estático **`Cliente getDB(String dni, Connection c)`** que, para un dni dado, y utilizando la conexión que se le pasa, obtenga los datos del cliente de la base de datos, y devuelva un objeto con todos los datos del cliente. Si no existe un cliente para el dni dado, devolverá `null`.

Para probar esta clase, se podría utilizar el siguiente código.

```
try ( Connection c = DriverManager.getConnection(urlConnection, user, pwd))) {

    String dnis = { "12345678G", "87654321E" };
    for(String unDni: dnis) {
        Cliente cl = Cliente.getDB(unDni , c);
        if(cl != null) {
            System.out.printf("Datos de cliente con DNI %s: %s", unDni, cl);
        } else {
            System.out.printf("ERROR: No existe cliente con DNI %s\n", unDni);
        }
    }
} catch (SQLException e) {
    muestraErrorSQL(e);
}
```

Actividad 2.5

Añade a la clase creada para la anterior actividad un nuevo método estático `int getNumClientesDB(Connection c)` que devuelva el número de clientes que existen en la tabla `clientes`. Lo mejor para ello es obtener el resultado de la consulta `select count(*) from clientes`.

8. Sentencias preparadas

Todas las sentencias de SQL que se han utilizado hasta ahora son constantes. Pero en la práctica es muy habitual el caso en que se quiere ejecutar una sentencia que tiene una parte variable que cambia entre una ejecución y otra. Por ejemplo: un programa podría obtener un DNI y asignarlo a una variable, y después obtener las facturas que existan para el cliente con ese DNI.

Para ello, se deben utilizar sentencias preparadas, que son objetos de la clase `PreparedStatement`.

Una sentencia preparada es una sentencia que tiene determinados marcadores o *placeholders*, que representan valores que se proporcionan en el momento de ejecutar la sentencia.

A continuación se compara la forma en que se ejecuta una sentencia `insert` con `Statement` y con `PreparedStatement`. La primera forma (con `Statement`) debe evitarse, como se explica en breve.

Statement	PreparedStatement

`int codProd = 1;`
`String nomProd = "Pan";`
`double prUnit = 0.80;`	

`String sent =`
`"insert into productos values("`
`+ codProd + ",'" + nomProd + "'," + prUnit`
`+ "')";`
	`PreparedStatement ps =`
	`c.prepareStatement("insert into productos(cod_prod, nom_prod, pr_unit) values (?, ?, ?)");`
	
	`int i = 1;`
	`ps.setInt(i++, codProd);`
	`ps.setString(i++, nomProd);`
	`ps.setDouble(i++, prUnit);`

`s.executeUpdate(sent);`	`ps.executeUpdate();`

La interfaz `PreparedStatement` incluye:

- Métodos `setXXX`, donde *XXX* representa distintos tipos de datos, para asignar valores a distintos *placeholders* o marcadores, que en la sentencia se representan con el carácter `'?'`. El marcador para el que se asigna valor viene designado por un número entero que indica su posición.
- Un método `setNull`, que asigna un valor `null` en SQL. Como atributo tiene también un número entero que indica la posición del marcador para el que se asigna un valor `null`.

Ventajas del uso de sentencias preparadas

Cuando una sentencia de SQL contiene valores que varían entre diferentes ejecuciones, deben utilizarse sentencias preparadas (`PreparedStatement`) en lugar de sentencias (`Statement`), por los siguientes motivos.

- Seguridad. Una sentencia de SQL que se compone mediante concatenación de cadenas de caracteres en las que algunas están en variables es un vector para ataques por **inyección de SQL**. Este es un grave riesgo de seguridad, porque mediante inyección de SQL se puede conseguir el

acceso no autorizado a datos y la modificación y el borrado de datos.

- Rendimiento para las consultas. Cuando una consulta debe ejecutarse, el servidor de bases de datos la analiza y crea un plan de ejecución para ella, con el objetivo de optimizar su tiempo de ejecución y uso de recursos. Si se usa una sentencia preparada, esto solo se hace cuando se prepara la sentencia. Si no, se hace cada vez que se ejecuta la sentencia.

Actividad 2.6

Crea una clase **`Producto`** cuyos atributos se correspondan con las columnas de la tabla `productos`. Los nombres de sus atributos, por supuesto, deben seguir los convenios de nomenclatura habituales para la programación en Java. Crea un programa que cree un *array* con varios objetos de esta clase. En alguno de ellos, el valor del atributo correspondiente al campo `descr` de la tabla productos debe ser `null`. Crea un método estático **`boolean insertDB(Connection c)`**. Este método debe crear una nueva fila en la base de datos con los datos del producto. Debe utilizar `PreparedStatement`, y no `Statement`. Si el valor de algún atributo tiene valor `null`, debe asignarse un valor `null` al correspondiente atributo de la nueva fila que se inserta en la base de datos. Si se ha podido crear, debe devolver `true`. En otro caso, debe devolver `false`. Este método debe propagar cualquier excepción de tipo `SQLException`.

Crea un método estático **`getDB`** para crear un `Producto` con los datos existentes en la base de datos para un código de producto dado. Para verificar ambos métodos se puede utilizar el siguiente código.

```
try (Connection c = DriverManager.getConnection(urlConnection, user, pwd)) {
    Producto pr = new Producto(5, "SACACORCHOS", 6.50, "SACACORCHOS ACERO");
    pr.insertDB(c);

    Producto prVerif = Producto.getDB(pr.getCodProd());
    if(!pr.equals(prVerif)) {
        System.out.printf(
          "ERROR: producto en memoria (%s) distinto del de base de datos (%s)\n",
          pr, prVerif);
    }
} catch (SQLException e) {
    muestraErrorSQL(e);
}
```

Actividad 2.7

Crea un programa que introduzca datos de productos utilizando `PreparedStatement`. Los datos para los productos debe obtenerlos de un fichero de texto que debes preparar a mano, con datos para varios productos. El fichero debe leerse línea a línea, utilizando `BufferedReader` Dentro de una línea debe utilizarse el carácter `'|'` como separador para los valores de los diferentes atributos. Si no se especifica valor para un atributo, debe interpretarse como un valor nulo.

Con el siguiente programa de ejemplo se obtienen los datos de un producto, dado su código de producto. Como la consulta se hace por el código de producto, que es la clave primaria, solo hay dos posibilidades. O bien se obtiene una fila, o bien no se obtiene ninguna.

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

```java
public class ObtenDatosDeProducto {

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

      int codProd = 2;
      try (
         Connection c = DriverManager.getConnection(urlConnection, user, pwd);
         PreparedStatement ps = c.prepareStatement(
               "select * from productos where cod_prod=?"
         )
      ) {
         ps.setInt(1, codProd);
         try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
               System.out.printf("%d %s %f\n", rs.getInt("cod_prod"),
                  rs.getString("nom_prod"), rs.getDouble("pr_unit")
               );
            } else {
               System.out.printf("No existe ningún producto con código %d\n", codProd);
            }
         }
      } catch (SQLException e) {
         muestraErrorSQL(e);
      }
   }
}
```

Actividad 2.8

Cambiar las clases **`Cliente`** y **`Producto`** creadas para actividades anteriores, para que usen sentencias preparadas (`PreparedStatement`).

9. Obtención de valores de claves autogeneradas

Una clave autogenerada se define en MySQL con la opción `auto_increment`. No es necesario especificar un valor para un atributo que se define como clave autogenerada, porque MySQL le asignará automáticamente un nuevo valor disponible. La tabla `facturas` tiene una clave autogenerada, el campo `num_factura`. En el siguiente ejemplo se crea una nueva factura desde el intérprete de línea de comandos de SQL. Cuando se inserta una nueva fila en la tabla `facturas` , no se especifica un valor para el atributo `num_factura`.

```
mysql> insert into facturas (cod_cliente, fecha_factura) values(2, now());
Query OK, 1 row affected, 1 warning (0,04 sec)
```

Una vez creada la factura, se puede ver que el valor asignado a `num_factura` es 1. Con MySQL se puede obtener el último valor asignado a una clave autogenerada con la función `last_insert_id()`. Y con este valor para la clave autogenerada, se puede obtener la fila recién insertada.

```
mysql> select last_insert_id();
+------------------+
| last_insert_id() |
+------------------+
|                1 |
+------------------+
1 row in set (0,00 sec)

mysql> select * from facturas where num_factura=last_insert_id();
+-------------+-------------+---------------+
| num_factura | cod_cliente | fecha_factura |
+-------------+-------------+---------------+
|           1 |           2 | 2023-04-29    |
+-------------+-------------+---------------+
1 row in set (0,00 sec)
mysql> select last_insert_id();
```

La función `last_insert_id()` es propia de MySQL, y no existe en general para otros sistemas gestores de bases de datos.

Hay que recalcar que el valor recién generado no tiene por qué ser necesariamente el mayor valor existente en la tabla. Por tanto, no son fiables consultas del tipo `select max(num_factura) from facturas` o `select * from facturas where num_factura>=all(select num_factura from facturas)`.

Para crear dos líneas para esta factura, hay que especificar este valor para el número de factura.

```
mysql> insert into lineas_facturas(num_factura, lin_factura, cod_prod,
cantidad, pr_unit) values(1, 1, 2, 10, 9.50);
Query OK, 1 row affected (0,04 sec)

mysql> insert into lineas_facturas(num_factura, lin_factura, cod_prod,
cantidad, pr_unit) values(1, 2, 3, 24, 11.50);
Query OK, 1 row affected (0,04 sec)
```

La cuestión que se plantea, entonces, cuando se quiere crear una factura y dos líneas suyas con Java y JDBC, es cómo recuperar el valor que se ha generado para la clave autogenerada, es decir, para el número de factura. De manera fiable y sin utilizar mecanismos específicos de SGBD particulares.

Tanto `PreparedStatement` como `Statement` tienen un método `getGeneratedKeys` que devuelve un `ResultSet` con el que se puede obtener el valor generado para la clave autogenerada. Para poder recuperlo, debe crearse la sentencia preparada (`PreparedStatement`) con la opción `PreparedStatement.RETURN_GENERATED_KEYS`.

En el siguiente ejemplo, se muestra la creación de una factura y de dos líneas para ella.

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

public class CreaFactura {

    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 sInsertFact = c.prepareStatement(
                "insert into facturas(cod_cliente, fecha_factura) VALUES (?, ?)",
                PreparedStatement.RETURN_GENERATED_KEYS);
            PreparedStatement sInsertLinFact = c.prepareStatement("insert into
lineas_facturas(num_factura, lin_factura, cod_prod, cantidad, pr_unit) VALUES
(?,?,?,?,?);")) {

            int i = 1;
            sInsertFact.setInt(i++, 3);  // Código de cliente
            sInsertFact.setDate(i++, new java.sql.Date(new java.util.Date().getTime()));
            sInsertFact.executeUpdate();
            ResultSet rs = sInsertFact.getGeneratedKeys();
            rs.next();
            int numFact = rs.getInt(1);

            System.out.printf("Creada nueva factura con número fact.: %d.\n", numFact);

            int lineaFact = 1;
            i = 1;
            sInsertLinFact.setInt(i++, numFact);
            sInsertLinFact.setInt(i++, lineaFact++);
            sInsertLinFact.setInt(i++, 3);  // Código de producto
            sInsertLinFact.setDouble(i++, 24);  // Cantidad
            sInsertLinFact.setDouble(i++, 5.84);  // precio unitario
            sInsertLinFact.executeUpdate();

            i = 1;
            sInsertLinFact.setInt(i++, numFact);
            sInsertLinFact.setInt(i++, lineaFact++);
            sInsertLinFact.setInt(i++, 2);  // Código de producto
            sInsertLinFact.setDouble(i++, 120);  // Cantidad
            sInsertLinFact.setDouble(i++, 3.15);  // precio unitario
            sInsertLinFact.executeUpdate();

            System.out.printf("Creadas %d líneas para factura con núm. %d.\n",
                lineaFact - 1, numFact);

        } catch (SQLException e) {
            muestraErrorSQL(e);
        }
    }
}
```

Para este programa se utilizan varias prácticas que, aunque pueda argumentarse que complican el programa, en realidad lo simplifican y, sobre todo, lo hacen más mantenible para el futuro.

- Se especifica el nombre de todos los atributos de la tabla con la sentencia `insert`. De esa forma, las sentencias seguirán funcionando aunque se cambie la estructura de la taba para añadir o eliminar atributos o para cambiar su orden.
- Se utiliza un contador `i` para el atributo al que se le asigna un valor. De esta forma, para todos los atributos se tiene `i++` en lugar de números diferentes, que podría ser necesario cambiar más adelante si se eliminan o añaden atributos en la sentencia `insert`.

- También se utiliza un contador para las líneas de factura. De esta forma, se tiene también siempre lo mismo para todas las líneas (`lineaFact++`), en lugar de números distintos.

Este código es más homogéneo, consistente y, sobre todo, mantenible. Porque sigue funcionando aunque cambie la estructura de las tablas, es decir, el conjunto de atributos y su orden. También son más fáciles las modificaciones y se modifica el riesgo de errores, incluso de errores que pueden pasar inadvertidos durante un tiempo, si se cambian las sentencias para asignar valor a un conjunto distinto de atributos de la tabla.

Gestión de fechas con JDBC

La gestión de fechas es muy problemática en Java, porque la clase `java.util.Date` tiene graves errores de diseño. La cosa se complica con JDBC, porque utiliza una clase `java.sql.Date`. Se puede asignar la fecha actual con `new java.sql.Date(new java.util.Date().getTime())`. Pero hay que tener presente que, con esta solución, la fecha que se inserta en la base de datos es la de la máquina virtual de Java en la que se ejecuta el programa. Si se está ejecutando el mismo programa en distintas máquinas virtuales, cada una puede tener una fecha distinta, y además distinta de la fecha del sistema en la que se ejecuta el servidor de bases de datos.

10. Transacciones

Una transacción es una secuencia de sentencias de SQL que se agrupan, de manera que se ejecutan en conjunto, una detrás de otra. Se ejecutan de manera atómica. Esto significa que si sucede cualquier error durante la ejecución de cualquiera de ellas, que impide que se ejecute correctamente, se deshacen todos los cambios hechos desde que se comenzó la transacción, de manera que los contenidos de la base de datos quedan exactamente como estaban antes de iniciarse.

El siguiente programa de ejemplo hace lo mismo que el anterior, pero los cambios que se hacen el la base de datos se agrupan en una transacción. No se quiere crear la factura si no se pueden crear también todas sus líneas. O se crea una factura con todas sus líneas, o la base de datos se queda igual que como estaba antes de empezar a crear el conjunto de la factura con todas sus líneas.

Las sentencias incluidas en la transacción se ejecutan entre `c.setAutoCommit(false)` y `c.commit()`. Si se produce una excepción de tipo `SQLException`, se deshacen los cambios realizados hasta el momento con `c.rollback()`.

Ello obliga a sacar la creación de la conexión a un bloque `try` con recursos previo, para que en el bloque `catch` se tenga acceso a la conexión `c`.

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

public class CreaFactura {

    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(cod_cliente, fecha_factura) VALUES (?, ?)",
                    PreparedStatement.RETURN_GENERATED_KEYS);
                PreparedStatement sInsertLinFact = c.prepareStatement(
"insert into lineas_facturas(num_factura, lin_factura, cod_prod, cantidad, pr_unit)
VALUES (?,?,?,?,?);")) {

                c.setAutoCommit(false);

                int i = 1;
                sInsertFact.setInt(i++, 3);  // Código de cliente
                sInsertFact.setDate(i++,
                    new java.sql.Date(new java.util.Date().getTime()));
                sInsertFact.executeUpdate();
                ResultSet rs = sInsertFact.getGeneratedKeys();
                rs.next();
                int numFact = rs.getInt(1);

                System.out.printf("Creada nueva factura con número: %d.\n", numFact);

                int lineaFact = 1;
                i = 1;
                sInsertLinFact.setInt(i++, numFact);
                sInsertLinFact.setInt(i++, lineaFact++);
                sInsertLinFact.setInt(i++, 3);  // Código de producto
                sInsertLinFact.setDouble(i++, 24);  // Cantidad
                sInsertLinFact.setDouble(i++, 5.84);  // precio unitario
                sInsertLinFact.executeUpdate();

                i = 1;
                sInsertLinFact.setInt(i++, numFact);
                sInsertLinFact.setInt(i++, lineaFact++);
                sInsertLinFact.setInt(i++, 2);  // Código de producto
                sInsertLinFact.setDouble(i++, 120);  // Cantidad
                sInsertLinFact.setDouble(i++, 3.15);  // precio unitario
                sInsertLinFact.executeUpdate();

                c.commit();

                System.out.printf(
                    "Creadas %d líneas de factura para factura con número %d.\n",
                    lineaFact - 1, numFact);

            } catch (Exception ex) {
                if (ex instanceof SQLException) {
                  muestraErrorSQL((SQLException) ex);
                } else {
                  System.out.printf("ERROR: %s\n", ex.getMessage());
                }
                try {
                    c.rollback();
                    System.out.println("Se hace ROLLBACK");
                } catch(SQLException exr) {
```

```
                    System.out.printf("ERROR en rollback: %s.\n", exr.getMessage());
                }
            }
        } catch(SQLException ex) {
            muestraErrorSQL(ex);
        }
    }
}
```

Actividad 2.9

Verificar que, si se produjera algún error durante la ejecución de la transacción, se desharían todos los cambios realizados. Para forzar un fallo, se puede cambiar la sentencia que crea la última línea para que la cree para un producto que no existe (es decir, con un valor de `cod_prod` no existente en la tabla `productos`). Esto provocará un error, porque se viola la restricción de integridad referencial con origen en el campo `cod_prod` de `facturas` y destino en la tabla `productos`. Una vez ejecutado el programa, verificar los mensajes que escribe el programa, para hacerse una idea de cuál ha sido su curso de ejecución, y comprobar con sentencias de SQL que no existe ni la factura ni, por supuesto, ninguna línea suya en la base de datos. Esto último se puede hacer con un intérprete de línea de comandos de SQL, como `mysql`.

Actividad 2.10

Crear una tabla `monedas` que contenga datos de monedas de curso legal. Debe tener como campos `cod_moneda` y `nom_moneda` para el código y el nombre de la moneda. Deben insertarse los datos de las monedas presentes en la tabla disponible en el artículo https://es.wikipedia.org/wiki/ISO_4217 relativo al estándar ISO 4217). A partir de esta tabla, se debe crear un fichero en formato CSV, que debe leerse línea a línea. Con los datos de cada línea se insertará una línea en la tabla `monedas`. Debe utilizarse una sentencia preparada (`PreparedStatement`) para una sentencia `insert into` monedas … Todas las sentencias deben ir en una misma transacción, de manera que, si se produce un error que impide que se ejecuten correctamente todas las sentencias `insert`, se deshagan todos los cambios.

Actividad 2.11

Crear una tabla `cuentas` que contenga datos de cuentas bancarias. Al menos un número de cuenta (24 caracteres), que es la clave primaria, una moneda (tres caracteres, según el estándar ISO 4217), para la que se define una clave foránea que apunta hacia la tabla `monedas` creada con la actividad anterior, y un saldo inicial.

Crear un programa que cree al menos cinco cuentas, con distintas monedas. Los datos para crear cada cuenta se leerán de un *array* de dos dimensiones, `Object[][] datosCuentas`. En cada fila estarán los datos de una cuenta. Se da como ejemplo un *array* con una fila, y un programa que escribe sus contenidos.

```
Object[][] datosCuentas = {
    { "ES9812345678901234567890", "EUR", 2354.32  },
    { "GB92BARC20005275849855200052758498 55", "GBP", 5634.32  }
};

for(Object[] datosCuenta: datosCuentas) {
    System.out.printf("Cuenta %s, moneda %s, saldo inicial %f.\n",
        datosCuenta[0], datosCuenta[1], datosCuenta[2]);
}
```

Para crear los datos de las cuentas en la base de datos, se creará una clase `Cuenta` con los siguientes métodos.

- Un constructor al que se le pasan todos los datos para crear una cuenta.
- Un método `boolean insertDB(Connection c)` que crea una cuenta en la base de datos. Si se ha

podido crear, debe devolver `true`. En otro caso, debe devolver `false`. Este método debe propagar cualquier excepción de tipo `SQLException`.

Todas las operaciones para crear las cuentas en la base de datos se deben realizar en una única transacción. Si se produce cualquier error, se deben deshacer todos los cambios realizados hasta el momento.

Actividad 2.12

Añadir a la clase `Cuenta` los siguientes métodos:

- Un método estático `Cuenta getDB(String numCuenta, Connection c)` que crea un objeto a partir de los datos de la base de datos. Si en la base de datos no hay ninguna cuenta con el número de cuenta dado, debe devover `null`. Este método debe propagar cualquier excepción de tipo `SQLException`.
- Métodos `boolean ingreso(double cant, Connection c)` y `boolean retirada(double cant, Connection c)`, que ingresan y retiran, respectivamente, la cantidad dada. Devuelven `true` si se ha podido realizar la operación, y `false` en otro caso. Propagan cualquier excepción de tipo `SQLException` que se pueda producir.
- Métodos `boolean transferenciaHacia(Cuenta cuentaDest, double cant, Connection c)` y `boolean transferenciaDesde(Cuenta cuentaOrig, double cant, Connection c)`. Ambos realizan una retirada de la cantidad dada en la cuenta de origen, y un ingreso de la misma cantidad en la cuenta destino. Estas dos operaciones deben hacerse dentro de una transacción, de manera que, si se produce un error cuando se hace la transferencia, todo quede como estaba al principio.

Probar todos estos métodos con un programa que lea varias cuentas de la base de datos y realice 5000 transferencias de una cantidad al azar entre 1 y 1000€ (con dos decimales) entre una cuenta origen y otra destino, ambas diferentes, elegidas al azar. No elegir las cuentas al azar entre todas las que existan en la tabla `cuentas`, sino entre un conjunto más pequeño de cuentas, identificadas por sus números de cuenta, que deben estar en un *array*, y que deben existir en la base de datos. Para cada número de cuenta del *array*, se debe obtener un objeto de clase `Cuenta` con el método `Cuenta getDB(String numCuenta, Connection c)`.

Deben realizarse 5000 transferencias. Cuando se elige una cuenta de origen y otra de destino y una cantidad al azar, podría ser posible que no se pueda realizar la transferencia porque la cuenta de origen no tenga bastante saldo, o porque las cuentas sean de distinta moneda. En ese caso, no se cuenta la transferencia.

Al final del programa, se debe indicar el número de transferencias intentadas (que deben ser como mínimo 5000), el número de transferencias realizadas (que deben ser 5000), y el saldo final de cada cuenta (indicar para cada una número de cuenta, moneda, y saldo). El saldo conjunto de todas las cuentas debe ser igual al final que al principio.

11. Procedimientos y funciones almacenados

Para crear procedimientos y funciones almacenados se necesitan extensiones procedurales del lenguaje SQL. Estas incluyen el tipo de sentencias que existen en todos los lenguajes imperativos estructurados, a saber:

- Sentencias de asignación. Con las cuales se asigna un valor a una variable. Las extensiones procedurales de SQL incluyen una sentencia `select … into variable`, que permite asignar a una variable el resultado de una consulta de SQL. Por ejemplo: `select count(*) clientes into num_clientes`.
- Sentencias condicionales de los tipos `if … end if`, `if … else … end if`, `if … else if … else … end if`.
- Sentencias iterativas o bucles, entre ellos bucles `while`.

Los procedimientos y funciones almacenados son muy similares. Ambos consisten en bloques de código que tienen un nombre y pueden tener parámetros. Las funciones almacenadas pueden, además, devolver un valor.

Se muestra a continuación un ejemplo de procedimientos almacenado. Se puede invocar con `call`.

Declaración de procedimiento almacenado: crear un cliente, dados sus datos (código de cliente, dni y nombre)	`DELIMITER //`
	`CREATE PROCEDURE crea_cliente`
	`(cod_cliente INTEGER, dni CHAR(9), nom_cliente`
	`VARCHAR(40))`
	`BEGIN`
	`  INSERT INTO clientes(cod_cliente, dni, nom_cliente)`
	`  VALUES (cod_cliente, dni, nom_cliente);`
	`END //`
	`DELIMITER ;`

Invocación con SQL	`call crea_cliente(20, '87654321G', 'LUPIÁNEZ');`

Se muestra a continuación un ejemplo de función. Se puede utilizar en una sentencia `select`.

Función almacenada para obtener el número de facturas para un cliente, dado su DNI	`DELIMITER //`
	`CREATE FUNCTION num_fact_cliente (in_cod_cliente INTEGER)`
	`RETURNS INTEGER DETERMINISTIC`
	`BEGIN`
	`  DECLARE num_fact INTEGER;`
	`  SELECT COUNT(*) INTO num_fact FROM facturas`
	`  WHERE cod_cliente=in_cod_cliente;`
	`  RETURN(num_fact);`
	`END //`
	`DELIMITER ;`

Invocación con SQL	`select num_fact_cliente(3);`

Los anteriores procedimientos y funciones almacenados son muy sencillos. Se podría hacer lo mismo con sencillas sentencias de SQL. Pero lo que interesa aquí es ilustrar la forma en que se pueden llamar desde un programa de Java con JDBC.

Los procedimientos y funciones almacenados se invocarán de manera distinta.

- **Procedimientos almacenados**. Se utilizará el método `prepareCall` de la clase `CallableStatement`. A este se le pasará como parámetro un patrón de llamada. Se muestra un ejemplo a continuación para un procedimiento almacenado que borra un producto de la base de datos, dado su código de producto.

```
try (
    Connection c = DriverManager.getConnection(urlConnection, user, pwd);
    CallableStatement s = c.prepareCall("{call crea_cliente(?, ?, ?)}")
    ) {
  int i = 1;
  s.setInt(i++, 21);
  s.setString(i++, "79863542W");
  s.setString(i++, "CORTÉS");
  s.execute();
} catch(SQLException ex) {
  muestraErrorSQL(ex);
}
```

- **Funciones almacenadas**. Se puede usar el mismo planteamiento que para una sentencia de SQL del tipo `select count(*)`, ya que devuelven un único valor. Por ejemplo, para llamar a una función que devuelve el número de facturas para un cliente con un DNI dado, se podría utilizar el siguiente código de programa.

```
try (
    Connection c = DriverManager.getConnection(urlConnection, user, pwd);
    PreparedStatement ps = c.prepareStatement("select num_fact_cliente(?)")
    ) {
  int codCliente = 3;
  ps.setInt(1, codCliente);
  try (ResultSet rs = ps.executeQuery()) {
    rs.next();
    System.out.printf("%d facturas para cliente con código %d.\n",
      rs.getInt(1), codCliente);
  }
} catch(SQLException ex) {
  muestraErrorSQL(ex);
}
```

Actividad 2.13

Crea un procedimiento almacenado que borre un cliente de la base de datos, dado su DNI. Crea un programa en Java que utilice este procedimiento almacenado.

Actividad 2.14

Crea una función almacenada que devuelva el número de facturas existentes para un cliente, dado su DNI. Crea un programa en Java que utilice esta función.

Actividad 2.15

Crea un procedimiento almacenado que realice una transferencia de una cantidad dada de una cuenta de origen dada a una cuenta de destino dada. Esta solo se realizará si el saldo de la cuenta de origen es mayor o igual que la cantidad dada. Y en ese caso, se hará dentro de una transacción en el procedimiento almacenado.

Crea un programa en Java que utilice este procedimiento almacenado para realizar una transferencia desde una cuenta de origen a una cuenta de destino. No hay que utilizar una transacción en este programa, porque la transacción ya se hace en el procedimiento almacenado.

Los procedimientos y funciones almacenados tienen muchas más posibilidades que las vistas hasta ahora. Por ejemplo, podrían devolver un conjunto de filas que se podrían obtener en un `ResultSet`. Pero lo único más que se verá, para terminar, es la posibilidad de que un procedimiento almacenado devuelva un valor.

El siguiente procedimiento almacenado crea un nuevo producto dados todos los datos menos el código de producto. El procedimiento asigna como número de producto el siguiente al más alto existente, o 0 si no existe ningún producto. Este valor se devuelve en un parámetro de salida (**`OUT out_cod_prod INTEGER`**).

Declaración de procedimiento almacenado: crear un producto, dados todos sus datos menos el código de producto (nombre de producto, precio unitario y descripción). Devolver el nuevo número de producto asignado.	`DELIMITER //`
	`CREATE PROCEDURE crea_producto`
	`(IN in_nom_prod VARCHAR(40), IN in_pr_unit DECIMAL(7,2), IN in_descr VARCHAR(120), OUT out_cod_prod INTEGER)`
	`BEGIN`
	`  SELECT ifnull(max(cod_prod)+1,0) INTO out_cod_prod FROM productos;`
	`  INSERT INTO productos(cod_prod, nom_prod, pr_unit, descr)`
	`  VALUES (out_cod_prod, in_nom_prod, in_pr_unit, in_descr);`
	`END //`
	`DELIMITER ;`

Invocación con SQL	`call crea_producto('ESCUADRA', 7.50, 'ESCUADRA DE PLASTICO', @cod_prod);` `SELECT @cod_prod;`

Con el siguiente código de programa se llama a este procedimiento almacenado para crear un nuevo producto, y se obtiene el nuevo código de producto que el procedimiento almacenado crea y devuelve en un parámetro de salida.

```
try (
    Connection c = DriverManager.getConnection(urlConnection, user, pwd);
    CallableStatement s = c.prepareCall("{call crea_producto(?, ?, ?, ?)}")) {
  int i = 1;
  s.setString(i++, "CARTABÓN");
  s.setDouble(i++, 8.60);
  s.setString(i++, "CARTABÓN DE PLÁSTICO");
  s.registerOutParameter(i, java.sql.Types.INTEGER);
  s.execute();
  int cod_nuevo_prod = s.getInt(i);
  System.out.printf(
    "Código asignado al nuevo producto: %s.\n", cod_nuevo_prod);
} catch (SQLException ex) {
  muestraErrorSQL(ex);
}
```

Actividad 2.16

Mejora el procedimiento almacenado creado en una actividad anterior para realizar una transferencia entre dos cuentas utilizando una transacción.

El procedimiento debe devolver el valor 1 si la transferencia se realizó correctamente, y el valor 0 si no se realizó porque el saldo en la cuenta de origen es menor que el saldo en la cuenta de destino.

Crea un programa en Java que utilice este procedimiento almacenado para realizar una transferencia entre dos cuentas. Pruébalo para el caso en que el saldo en la cuenta de origen es suficiente para realizar la transferencia y para el caso en que no lo es.

Pruébalo también para el caso en que la cuenta de origen no existe y para el caso en que la cuenta de destino no existe. Si piensas que aún habría que hacer algún cambio al procedimiento almacenado, hazlo. Ten en cuenta que no solo debe hacer lo que tiene que hacer en el caso en que la transferencia se puede hacer, sino que también debe proporcionar información precisa acerca de cualquier tipo de error que se pueda producir.