3. Transacciones

Los datos almacenados pueden compartirse entre múltiples aplicaciones, y estas pueden realizar lecturas y modificaciones concurrentes (simultáneas) sobre ellos. Por ello, debe controlarse el acceso a los datos para evitar problemas que puedan surgir cuando distintos programas realizan consultas y modificaciones simultáneas sobre los mismos datos. Es además muy habitual que un grupo de operaciones sobre datos relacionados formen un todo que debe llevarse a cabo conjuntamente y de manera aislada con respecto a otras operaciones simultáneas sobre esos datos. Aquí es donde entran en juego las **transacciones**.

Por ejemplo, imaginemos que hacemos una transferencia bancaria de una cuenta a otra. El importe se debe restar del saldo de la cuenta de origen y sumarse en la cuenta de destino. Si, por la razón que sea, no se pudiera sumar el saldo a la cuenta de destino, no debe hacerse efectiva la resta del saldo a la cuenta de origen, sino que debe deshacerse para que todo quede como al principio. Estas dos operaciones que de las que consta la transferencia constituyen una transacción. Además, hay que tener en cuenta que no se pueden realizar simultáneamente dos transferencias que involucren a la misma cuenta. En todo caso, una de ellas debe esperar a que termine la otra. Si un programa consulta el saldo de cualquiera de las dos cuentas bancarias involucradas en una transferencia que se está llevando a cabo, no obtendrá ese saldo hasta que la transferencia haya concluido. Si se completa con éxito, obtendrá el nuevo importe, y si no, obtendrá el importe antiguo, como si la transferencia nunca se hubiera intentado. Si solicita modificar el importe de cualquiera de las dos cuentas, esta modificación no se hará antes de que la transferencia haya concluido.

La sincronización de operaciones de lectura y escritura y las transacciones aseguran que los programas de aplicación siempre tengan una vista consistente, actualizada y correcta de los datos almacenados.

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** (atómica): una transacción debe ejecutarse completamente y sin errores. Si ocurre algún error, los cambios que se hayan hecho deben deshacerse, de modo que todo quede como si nunca se hubiese iniciado la transacción.
- **Consistent** (consistente): las restricciones de integridad definidas para los datos deben cumplirse tras cada operación incluida en la transacción y también al finalizar ésta.

- **Isolated** (aislada): una transacción está aislada de otras transacciones que se ejecutan simultáneamente. Las distintas transacciones tienen una visión consistente del conjunto de los datos y no ven las modificaciones hechas por otras transacciones que están en curso pero no concluidas.
- **Durable** (duradera): una vez completadas todas las operaciones que forman la transacción (y no antes), se confirman los cambios, que quedan grabados de forma permanente.

Una transacción se realiza en SQL de la siguiente forma:

```
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.

Las transacciones son importantes y se utilizan frecuentemente. Todos los SGBD relacionales proporcionan soporte para transacciones, pero cada uno tiene sus particularidades. En MySQL, las transacciones están deshabilitadas por defecto, de manera que los cambios realizados por cualquier sentencia se confirman automáticamente aunque no se ejecute una sentencia **COMMIT**.

JDBC proporciona una interfaz común para todas las bases de datos. También sería 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.

La interfaz Connection tiene varios métodos para realizar estas operaciones, siendo los principales:

Método	Funcionalidad
void setAutoCommit(boolean autoCommit)	Con autoCommit = false inicia una transacción. Es equivalente a START TRANSACTION.
void commit()	Equivalente a una sentencia COMMIT de SQL. Si después de utilizar este método se quieren 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 del tipo SQLException. Esto significa que alguna sentencia de SQL no se ha ejecutado correctamente y, entonces, se querrá estar seguro de que se aborta la transacción, descartando todos los cambios.

Como ejemplo, se muestra un programa que ejecuta varias sentencias de SQL como una transacción:

```
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);
        }
    }
}
```

No se pueden agrupar la creación de la conexión y la creación de la sentencia preparada en el mismo bloque de inicialización de recursos de un bloque try, porque 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.

4. Valores de claves autogeneradas

Es muy habitual crear una tabla de manera que la clave primaria sea una columna numérica y se deje al SGBD que proporcione un nuevo valor para cada nueva fila que se inserta. Las claves de este tipo se suelen llamar claves autoincrementales y en MySQL se definen como AUTO_INCREMENT. En JDBC, también se las conoce como claves autogeneradas.

Cuando insertamos una nueva fila en una tabla que tiene claves autogeneradas, puede que nos interese recuperar la clave que se acaba de generar para esa fila. Por ejemplo, si tenemos una aplicación que trabaja con facturas, cada factura debe tener un identificador único, que normalmente es un simple número. El número de cada factura es único y lo podemos definir como AUTO_INCREMENT para que sea el sistema el que le asigna un número nuevo a cada factura. Entonces, seguramente nos interesará recuperar ese número de factura en el momento de crearla.

Para poder hacer esto utilizando la interfaz *Statement*, debemos añadir el parámetro **`Statement.RETURN_GENERATED_KEYS`** cuando ejecutamos la sentencia:

```
sentencia.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
```

Si utilizamos la interfaz *PreparedStatement*, hacemos lo mismo al cargar la sentencia:

```
conexion.prepareStatement(sql, PreparedStatement.RETURN_GENERATED_KEYS);
```

Después, para obtener las claves autogeneradas, utilizamos el método **`getGeneratedKeys()`**, perteneciente a cualquiera de las dos interfaces. Este método devuelve un *ResultSet* al que deberemos acceder para obtener el valor de la clave. Un ejemplo de código es el siguiente:

```
Connection conexion = DriverManager.getConnection("url", "usuario", "contraseña");
PreparedStatement sentencia = conexion
    .prepareStatement("INSERT INTO factura VALUES (?)",
                      PreparedStatement.RETURN_GENERATED_KEYS);
sentencia.setDouble(1, 250.60);
sentencia.executeUpdate();
ResultSet resultado = sentencia.getGeneratedKeys();
resultado.next();
System.out.println("El numero de factura generado es: " + resultado.getInt(1));
```

EJERCICIOS PROPUESTOS

1. Implementa un programa en una clase llamada **TransaccionEmpleado** que inserte en la base de datos ***empleados*** (creada al principio de esta unidad) tres nuevos contables, pertenecientes al departamento 1, que se llamen Pedro, Lucía y Daniel. ¿Qué pasa si hay un error al insertar alguno de los empleados?

   Utiliza transacciones, tal y como se hace en el código de ejemplo, para controlar que se inserten los 3 empleados a la vez. Si hay algún error, no se insertará ninguno.

2. Crea una base de datos con el script **bd-facturas.sql**. Implementa un programa en una clase **TransaccionFactura** que cree una nueva factura para el cliente David Pérez López, en la que conste que compró 25 tuercas y 60 tornillos.
   Ten en cuenta el número de factura generado para luego insertar las líneas de factura. También debes controlar con transacciones que se inserta la factura completa.