UD1-Acceso a RDBMS

CFGS DAM - MP Acceso a Datos

1. INTRODUCCIÓN

El uso de una **base de datos** supone una mejora importante respecto a la gestión de los datos a través de **ficheros**, especialmente en los casos en que tengamos una cantidad grande de datos, con una estructura interna más o menos compleja, y con accesos concurrentes.

El **DBMS (DataBase Management System)** se encarga de resolver las consultas, asegurar la integridad de los datos, y permitir el acceso concurrente a múltiples usuarios o instancias de programas.

Como programadores, necesitaremos que nuestros programas puedan comunicarse con el DBMS: establecer una conexión, ejecutar una serie de sentencias de consulta y/o modificación de los datos almacenados, y finalizar la conexión.

En esta unidad, trabajaremos con **RDBMS (DBMS relacionales)** y utilizaremos el lenguaje SQL (Structured Query Language) para acceder a dichos datos.

2. ODBC Y JDBC

2.1. ODBC (Open DataBase Connectivity)

ODBC es un estándar de conexión a bases de datos relacionales, desarrollado por Microsoft a principios de los 90, y convertido en un estándar posteriormente.

ODBC utiliza un modelo de tres capas para acceder a la BD desde una aplicación:

1. **Para cada DBMS compatible con ODBC, existe un driver ODBC**. El driver ofrece una serie de funcionalidades de acceso al DBMS: traduce las funciones que utilizamos en el lenguaje de programación a las órdenes concretas que se tienen que enviar al SGBD concreto. Diferentes drivers harán esta traducción a órdenes diferentes del SGBD, pero mantendrán una interfaz casi idéntica de cara al lenguaje de programación.
2. **Un driver manager se encargará de mirar qué drivers tenemos disponibles y de configurarlos para poder utilizarlos**. También, el driver manager se encarga de uniformar el proceso de conexión a instancias diferentes del SGBD (por ejemplo, nos permite conectar a un servidor local, o en un servidor remoto).
3. **La aplicación se comunica con el driver manager para obtener el driver**

adecuado y establecer una conexión con la BD deseada. Después, utiliza el driver para realizar sus consultas contra la BD.

2.2. API Java JDBC (Java DataBase Connectivity)

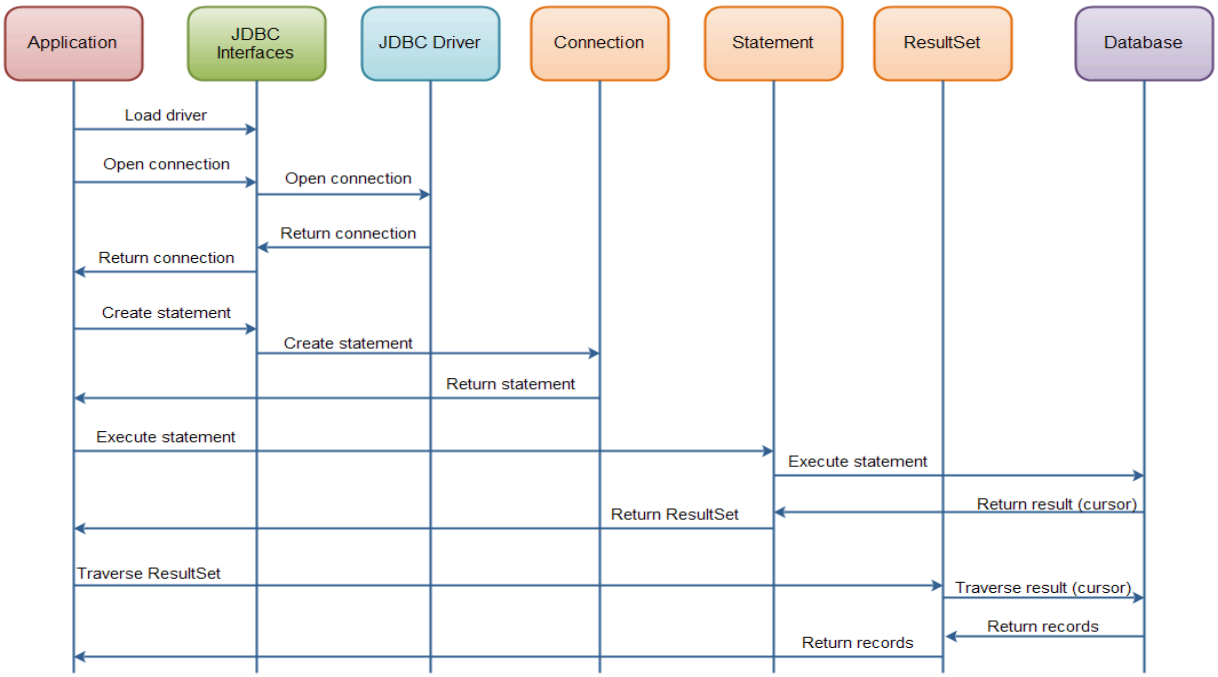
Una de las tareas que realiza ODBC es traducir los tipos de datos SQL a los tipos de datos del lenguaje de programación C. Dado que los tipos de C y Java son diferentes, ODBC no es un sistema adecuado para utilizar directamente desde Java. Además, el ODBC utiliza una interfaz de programación C y es relativamente difícil de aprender.

Para acceder a bases de datos relacionales, Java proporciona su propia API Java JDBC como parte de la edición estándar, lo que permite tener una solución puramente Java. La API Java JDBC es parte del SDK básica de Java SE.

JDBC funciona de forma similar a ODBC: existen drivers JDBC para cada uno de los SGBD habituales (MySQL, PostgreSQL, MS SQL Server, Oracle…) y los utilizamos desde nuestra aplicación para acceder a los datos.

Esencialmente, utilizaremos JDBC para siguientes tareas:

- **Cargar un controlador JDBC (opcional desde Java 6)**
- **Establecer una conexión al DBMS y BD.**
- **Crear un objeto Statement.**
- **Enviar sentencias SQL para que se procesen en DBMS.**
- **Obtener los resultados de nuestras sentencias.**
- **Cerrar las conexiones (innecesario si la conexión se ha abierto con try-with-resources)**



Problema: el uso de JDBC supone que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.

2.3. Driver JDBC

Los drivers JDBC son bibliotecas que permiten comunicar una aplicación Java con una base de datos específica e implementan las interfaces JDBC.

No todos los drivers tienen por qué tener toda la funcionalidad especificada por JDBC implementada, pero siempre tendrán la básica. **Las interfaces principales que declara JDBC son Statement, Connection, ResultSet, PreparedStatement y CallableStatement**.

Habitualmente, utilizaremos los drivers implementados y proporcionados por el creador del DBMS que utilicemos.

3. DBMS

Para practicar con JDBC necesitamos un DBMS relacional. Para ello, haremos una instalación local de MySQL o MariaBD en nuestro equipo.

MySQL es uno de los DBMS más utilizados, y es especialmente adecuado para aplicaciones web que necesitan bastante rapidez y no trabajan con un volumen muy grande de datos.

MariaDB es un fork de MySQL con un modelo de desarrollo más abierto que este último, y que proporciona algunas características adicionales.

Tanto uno como otro son software libre y son adecuados para seguir los ejemplos de estos apuntes.

También necesitaremos el driver JDBC de conexión a MySQL/MariaDB.

3.1. Instalación

Podemos utilizar el instalador de MySQL o el instalador de MariaDB.

3.2. Creación de usuarios

Para crear un usuario de MySQL/MariaDB ante todo debemos conectarnos como root. Aquí lo haremos desde línea de comandos, aunque también se puede hacer desde entorno gráfico utilizando MySQL Workbench o PHPMyAdmin.

Para conectarnos como root haremos:

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

Después de introducir la contraseña deberíamos ver el prompt del DBMS:

```
mysql>
```

A continuación podemos crear un usuario con la sentencia SQL CREATE USER y asignarle permisos con GRANT:

```
CREATE USER 'nombre_usuario'@'localhost' IDENTIFIED BY 'contraseña';
GRANT ALL PRIVILEGES ON nombre_BD.* TO 'nombre_usuario'@'localhost';
```

Esto creará un usuario que sólo podrá conectarse desde el equipo local y que tendrá todos los permisos sobre todas las tablas de la base de datos especificada.

Es necesario ejecutar el GRANT para cada una de las bases de datos que utilizamos.

En entornos de producción se pueden crear más usuarios con permisos distintos, para conseguir que nuestro programa tenga siempre los permisos mínimos necesarios. Esto nos ayudaría a proteger los datos en caso de que nuestro programa tenga una vulnerabilidad o se haya cometido un error.

También puede resultar una buena idea crear un usuario administrador distinto de root, que podamos utilizar directamente:

```
sudo /usr/bin/mysql -e "GRANT ALL ON *.* TO 'nombre_usuario'@'localhost'
IDENTIFIED BY 'contraseña' WITH GRANT OPTION"
```

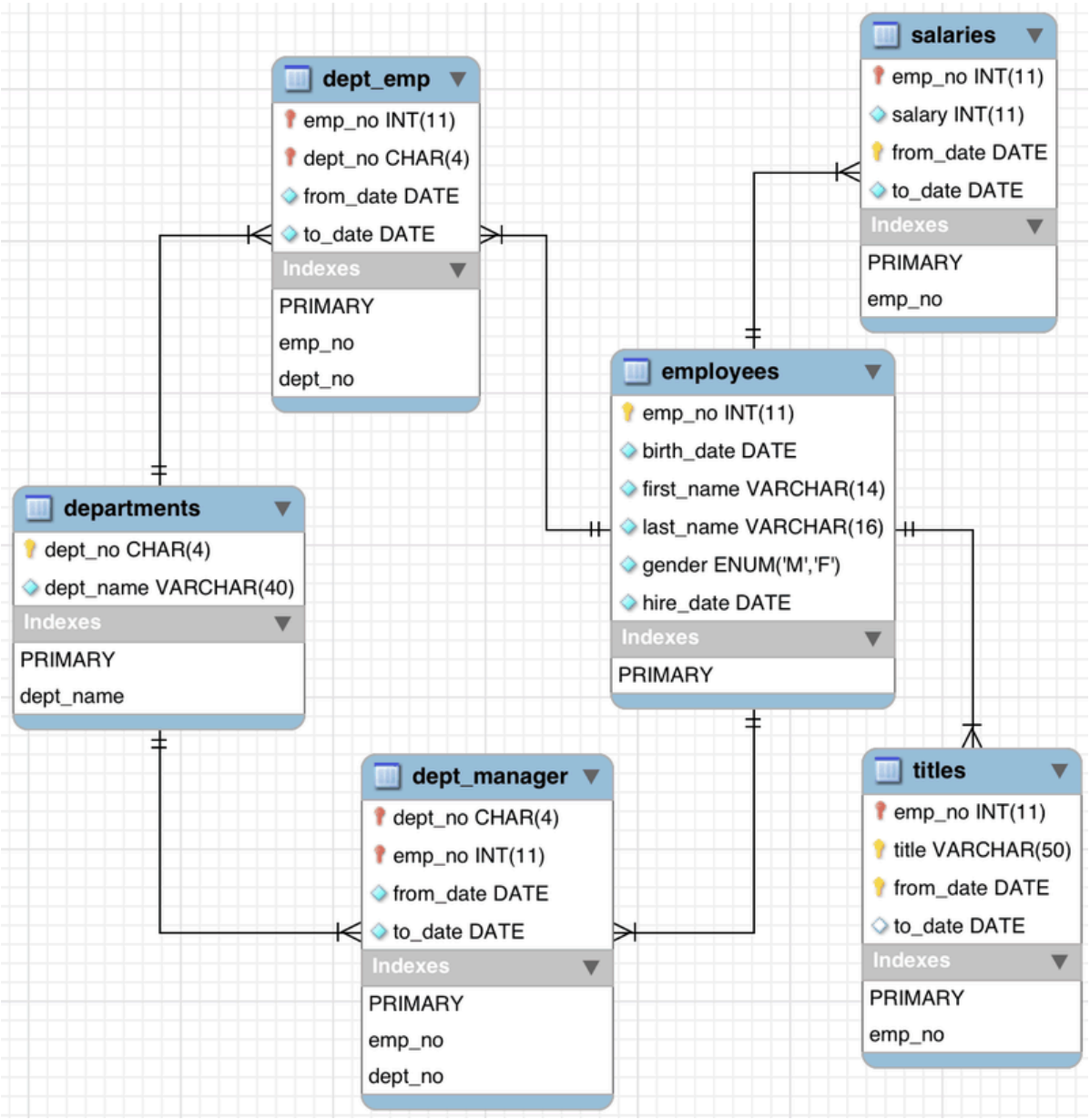
Este último usuario también nos permitirá administrar MySQL/MariaDB desde entornos gráficos, como Workbench o PHPMyAdmin.

3.4. Bases de datos de ejemplo

En la web oficial de MySQL disponemos de algunas BD de ejemplo con licencias libres. También os he publicado en el moodle algunas bases de datos de temática freak que quizás os resultan interesantes (starwars, lord of the rings, jurassic park y howarts).

Concretamente, para los ejemplos de estos materiales, se ha utilizado employees del repositorio oficial de MySQL, que es una base de datos con pocas tablas, pero muchos registros.

A continuación os muestro su diagrama CrowsFoot:



Para instalarla se debe descargar del repositorio correspondiente de github. Lo más fácil es descargarlo haciendo clic en Download zip.

Después de descomprimir, podemos pasarle el script que genera la base de datos en mysql:

```
$ cd <directorio en el que hemos descomprimido>

$ mysql -u root -p < employees.sql
```

O desde dentro del cliente mysql:

$ cd <directorio en el que hemos descomprimido>

$ sudo mysql

mysql> source employees.sql

4. Establecimiento de conexión

4.1. Incorporación del driver en el proyecto

Para poder establecer una conexión a MySQL/MariaDB desde Java es necesario incorporar el driver a nuestro proyecto.

Esto se puede automatizar utilizando un sistema como Maven o Gradle o también podemos importar directamente el archivo jar del driver en el proyecto.

Los conectores para MySQL y MariaDB son, en principio, intercambiables, pero es más seguro si utilizamos el conector adecuado al sistema que estemos utilizando.

Para añadir el uso de Maven en nuestro proyecto, entonces haremos botón derecho sobre el proyecto “Add Framework support” y seleccionaremos Maven.

Si abrimos el fichero pom.xml de configuración, añadiremos la dependencia del conector:

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>9.2.0</version>
</dependency>
```

4.2. Prueba de conexión

Vamos a intentar ahora establecer una conexión con el DBMS.

Las clases que aparecen en los siguientes ejemplos deben importarse del paquete **java.sql**. En este paquete se definen las interfaces que deben cumplir los drivers y cada driver proporciona sus implementaciones propias.

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

public class Conectar {
   public static void main(String[] args) {
       try (Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/employees?user=alu
mno&password=XXX");) {
           System.out.println("Conexión exitosa en la base de datos!");
       } catch (SQLException e) {
           System.err.println("Error de establecimiento de conexión: " +
e.getMessage());
       }
   }
}
```

Merece especial atención la línea de conexión:

jdbc:mysql://localhost:3306/employees?user=usuario&password=contraseña

- En primer lugar aparece el **protocolo**. Aquí debemos poner **jdbc:mariadb** si nos conectamos a un servidor MariaDB, o **jdbc:mysql** si nos conectamos a MySQL.
- Después tenemos la **dirección a la que nos conectamos**, en nuestro caso se trata del mismo equipo (**localhost**) en el puerto **3306**, que es el puerto por defecto de MySQL/MariaDB.
- A continuación tenemos la base de datos que queremos utilizar, en el ejemplo **employees**.
- Después pasamos el **usuario** y la **contraseña** que utilizamos para establecer la conexión.

Nótese que el formato de la línea de conexión es el mismo que el que se utiliza para indicar una URL en el navegador web.

3.3. Forma alternativa y recomendable de establecer la conexión

El método DriverManager.getConnection() está sobrecargado y admite varios formatos para indicar los parámetros de conexión.

En este ejemplo pasamos los parámetros a través de un object **Properties**. Como ya hemos visto, la clase Properties es un Map que se utiliza para guardar parejas de String. Se utiliza sobre todo para almacenar las opciones de configuración de un programa en un fichero.

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

public class Conectar2 {
   public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost:3306/employees";
        Properties propiedadesConexion = new Properties();
        propiedadesConexion.setProperty("user", "alumno");
        propiedadesConexion.setProperty("password", "XXX");
        try (Connection con =
                    DriverManager.getConnection(url,
propiedadesConexion);){
            System.out.println("Conexión exitosa en la base de datos!");
        } catch (SQLException e) {
            System.err.println("Error de establecimiento de conexión: " +
e.getMessage());
        }
   }
}
```

5. Sentencias DML

La interfaz Java JDBC Statement se utiliza para ejecutar sentencias SQL en una base de datos relacional.

Obtenemos un JDBC Statement de una conexión JDBC. Una vez que tengamos una instancia de Java Statement se puede ejecutar una consulta de base de datos o una actualización de base de datos con ella.

5.1. Consultas

Las consultas a una base de datos usando JDBC se hacen enviando sentencias SQL a la base de datos con un objeto Statement que se crea a partir de una conexión abierta y nos devuelve un objeto ResultSet con la consulta.

El objeto ResultSet devuelto está compuesto por filas (registros) con columnas de datos (campos).

El método next() mueve el cursor a la siguiente fila y devuelve true si existe, false en caso contrario.

Debe llamarse al menos una vez, ya que antes de la primera llamada el cursor se coloca antes de la primera fila.

Los datos de columna para la fila actual se obtienen llamando a algunos de los métodos getXXX(), donde XXX es un tipo de datos primitivo. O también con el método getString(X) donde X es el número de posición del campo en la consulta SELECT.

```
import java.sql.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Properties;

public class CrearConsulta {
   public static void main(String[] args) {
       DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy");
       String url = "jdbc:mysql://localhost:3306/employees";
       Properties connectionProperties = new Properties();
       connectionProperties.setProperty("user", "alumno");
       connectionProperties.setProperty("password", "XXX");
       try (Connection con =
DriverManager.getConnection(url,connectionProperties);
            Statement st = con.createStatement();) {
           System.out.println("Base de datos conectada!");
           try (ResultSet rs = st.executeQuery("SELECT * FROM employees
ORDER BY last_name, first_name LIMIT 1000");) {
               System.out.println("Datos de la tabla employees:");
               while (rs.next()) {
```

```
                    int empNo = rs.getInt("emp_no");
                    LocalDate birthDate = rs.getObject("birth_date",
LocalDate.class);
                   String firstName = rs.getString("first_name");
                    String lastName = rs.getString("last_name");
                    Gender gender = Gender.valueOf(rs.getString("gender"));
                    LocalDate hireDate =
rs.getObject("hire_date",LocalDate.class);
                    System.out.printf("%8d %14s %15s %15s %6s
%14s\n",empNo,birthDate.format(formatter),
firstName,lastName,gender,hireDate.format(formatter));
                }
            }
        } catch (SQLException e) {
            System.err.println("Error SQL: " + e.getMessage());
        }
    }

    public enum Gender {
        M, F;
    }
}
```

Para acceder a cada campo del ResultSet lo haremos con el método **get** del tipo de dato en cuestión (getString, getInt, getDouble…) pasándole como parámetro el nombre de la columna o el número de posición en la consulta SELECT.

Para los tipos enumeración, podemos crear un enum con los mismos valores y utilizar el método valueOf del enum, que traduce una cadena de texto a su valor correspondiente del enum.

El siguiente ejemplo muestra una situación algo más compleja. En la BD de empleados, cada empleado puede tener uno o varios títulos. Queremos un programa que nos muestre todos los títulos que hay en la base de datos, que permita al usuario elegir uno, y que finalmente muestre todos los empleados que tienen el título elegido por el usuario.

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class CrearConsulta2 {
```

```
    private Connection connection = null;
    private Statement statement = null;
    private Scanner scanner= new Scanner(System.in);

    private void connect() throws SQLException {
        String url = "jdbc:mysql://localhost:3306/employees";
        String user = "alumno";
        String passwd = "XXX";
        if (connection==null)
            connection = DriverManager.getConnection(url, user, passwd);
        if (statement==null)
            statement = connection.createStatement();
    }

    private void disconnect() {
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {
                ;
            } finally {
                statement=null;
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                ;
            } finally {
                connection=null;
            }
        }
    }

    public List<String> getTitles() throws SQLException {
        List<String> titles = new ArrayList<String>();
        try (ResultSet rs = statement.executeQuery("SELECT DISTINCT title
FROM titles;");) {
            //Añadir todos los títulos a la lista
            while (rs.next()) {
                titles.add(rs.getString(1));
```

```
            }
        }
        return titles;
    }

    public String chooseTitle(List<String> titles) {
        String title = "";
        for(String t : titles) {
            System.out.println(t);
        }
        System.out.println("\n¿Qué título quieres? ");
        while (!titles.contains(title)) {
            title = scanner.nextLine();
            if (!titles.contains(title))
                System.out.println("Este título no existe.");
        }
        return title;
    }

    public void employeesByTitle(String title) throws SQLException {
        try (ResultSet rs = statement.executeQuery("SELECT first_name,
last_name "
                + "FROM employees JOIN titles "
                + "ON employees.emp_no=titles.emp_no "
                + "WHERE title LIKE '" + title + "'");
        ) {
            while(rs.next()) {
                System.out.println(rs.getString("first_name")+" "+
                        rs.getString("last_name"));
            }
        }
    }

    public void run() {
        String title;
        try {
            //1- Conecta a la bbdd
            connect();
            //2- Muestra los títulos y pide al usuario que elija uno
            title = chooseTitle(getTitles());
            //3- Muestra a los empleados que tienen el título seleccionado
            System.out.println("Título seleccionado: "+title);
```

```
            employeesByTitle(title);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        } finally {
            disconnect();
        }
    }

    public static void main(String[] args) {
        CrearConsulta2 ej = new CrearConsulta2();
        ej.run();
    }
}
```

El problema de este código es que permitiría realizar SQLInjections (imaginemos que en lugar de introducir un title de empleado ponemos Manager' or '1'=1)

La solución pasaría porque sea el propio JDBC quién filtre las entradas inseguras. Esto lo realizaremos utilizando un PreparedStatement en lugar de un Statement. Lo veremos a continuación.

5.2. Sentencias preparadas

Un PreparedStatement es un tipo especial de Statement con algunas ventajas adicionales:

- Permite el uso de parámetros en la instrucción SQL.
- Filtra dichos parámetros y evita vulnerabilidades.
- Simplifica la escritura de sentencias ya que no se tienen que ir abriendo y cerrando las comillas y concatenando las cadenas.
- Permite reutilizar la sentencia con nuevos valores de parámetros.
- Puede aumentar el rendimiento de las sentencias ejecutadas si se repiten reiteradamente.
- Permite actualizaciones por lotes más fáciles.

```
import java.sql.*;
import java.util.Scanner;
public class SentenciaPreparada {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String lastName=" ";
        try (Connection conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/employees", "alumno", "XXX");
```

```
            PreparedStatement st = conn.prepareStatement(
                    "SELECT emp_no, first_name, last_name "
                            + "FROM employees WHERE last_name LIKE ?");) {
            while (!lastName.equals("")) {
                System.out.println("Apellido a buscar (en blanco para
salir): ");
                lastName = sc.nextLine();
                if (!lastName.equals("")) {
                    st.setString(1, lastName);
                    try (ResultSet rs = st.executeQuery();) {
                        while (rs.next()) {
                            int empNo = rs.getInt("emp_no");
                            String firstName = rs.getString("first_name");
                            lastName = rs.getString("last_name");
                            System.out.println(empNo+"\t"+firstName+"
"+lastName);
                        }
                    }
                }
            }
        } catch (SQLException e) {
            System.err.println("Error SQL: "+e.getMessage());
        }
        sc.close();
    }
}
```

5.3. Actualizaciones

Para actualizar la base de datos hay que utilizar un objeto Statement como en la consulta, pero en lugar de llamar al método executeQuery() se llama al executeUpdate().

El método executeUpdate() permite enviar cualquier sentencia de modificación de datos, así como de definición de datos (CREATE, DROP, ALTER); devuelve un int que es la cantidad de filas afectadas por el cambio.

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

```
import java.sql.Statement;
import java.util.Properties;

public class CrearInsert {
   public static void main(String[] args) {
       String url = "jdbc:mysql://localhost:3306/employees";
       Properties propiedadesConexion = new Properties();
       propiedadesConexion.put("user", "alumno");
       propiedadesConexion.put("password", "XXX");

       try (Connection con =
DriverManager.getConnection(url,propiedadesConexion);
            Statement st = con.createStatement();) {
           int numFiles= st.executeUpdate("INSERT INTO "
                   + "employees(emp_no, birth_date, first_name, last_name,
gender, hire_date) "
                   +
"VALUES(800001,'1953-03-16','Richard','Stallman','M','1975-01-21')");
           System.out.println("Inserción creada! Filas afectadas: " +
numFiles);
       } catch (SQLException e) {
           System.err.println("Error SQL: " + e.getMessage());
       }
   }
}
```

También podemos utilizar sentencias preparadas para las modificaciones de datos.

6. Transacciones

Una transacción es un conjunto de acciones que se llevarán a cabo como una única acción atómica: o se realizan todas las acciones, o no se realiza ninguna.

6.1. Autocommit

Si no decimos lo contrario, cuando trabajamos con JDBC contra MySQL tenemos la funcionalidad de **autocommit activada**. Esto significa que cada una de las operaciones que realizamos contra la base de datos se considera una transacción que es confirmada automáticamente cuando se ha ejecutado con éxito.

La ventaja de esto es que podemos olvidarnos de las transacciones y sabemos que

cualquier cosa que podamos expresar cómo una única sentencia SQL se ejecutará entera o fallará sin realizar cambios en la base de datos.

Pero el problema viene cuando necesitamos que varias sentencias se ejecuten como una única transacción. Entonces debemos deshabilitar el autocommit y coger nosotros el control para indicar cuándo comienza y termina cada una de las transacciones.

Con el **autocommit desactivado**, la totalidad de consultas ejecutadas no son definitivas hasta que se llama el método **commit()** explícitamente.

En cambio, si queremos anular todos los cambios que se han realizado dentro de la transacción, podemos hacerlo con el método **rollback()**.

6.2. Pasos

- Poner autocommit a false: connection.setAutoCommit(false)
- Ejecutar todas las actualizaciones.
- Si algo falla, deshacemos todo con connection.rollback();
- Si hay todo bien, confirmamos con connection.commit();
- Poner autocommit a true para que no afecte al resto del programa

```
import java.sql.*;

public class Transaccion {

   public static void main(String[] args) {
       int empNo = 0;
       int currentSalary = 0;

       // Sentencias SQL que necesitaremos
       String findEmpNoSQL = "select emp_no from employees " +
               "where first_name=? and last_name=? and hire_date=?";
       String findCurrentSalarySQL = "select salary from salaries " +
               "where emp_no=? and to_date='9999-01-01'";
       String closeSalarySQL = "update salaries set to_date=curdate() " +
               "where emp_no=? and to_date='9999-01-01'";
       String openSalarySQL = "insert into salaries " +
               "values (?, ?, curdate(), '9999-01-01')";

       // Establecemos conexión
       try (Connection con = DriverManager.getConnection(
```

```
                "jdbc:mysql://localhost:3306/employees", "alumno", "XXX");)
{
        // Creamos los PreparedStatement
            try (PreparedStatement findEmpNoSt =
con.prepareStatement(findEmpNoSQL);
                 PreparedStatement findCurrentSalarySt =
con.prepareStatement(findCurrentSalarySQL);
                 PreparedStatement closeSalarySt =
con.prepareStatement(closeSalarySQL);
                 PreparedStatement openSalarySt =
con.prepareStatement(openSalarySQL);) {
                // Desactivamos autocommit para iniciar transacción
                con.setAutoCommit(false);

                // Buscamos el número del empleado
                findEmpNoSt.setString(1, "Georgi");
                findEmpNoSt.setString(2, "Facello");
                findEmpNoSt.setString(3, "1986-06-26");
                try (ResultSet rs = findEmpNoSt.executeQuery();) {
                    if (rs.next()) {
                        empNo = rs.getInt(1);
                        if (rs.next()) {
                            throw new SQLException("Hay más de un trabajador
con estos datos.");
                        }
                    } else {
                        throw new SQLException("No hay ningún trabajador con
estos datos.");
                    }
                }

                // Buscamos el sueldo actual del empleado
                findCurrentSalarySt.setInt(1, empNo);
                try (ResultSet rs = findCurrentSalarySt.executeQuery();) {
                    rs.next();
                    currentSalary = rs.getInt(1);
                }

                // Finalizamos el sueldo actual
                closeSalarySt.setInt(1, empNo);
                closeSalarySt.executeUpdate();
                // Asignamos el nuevo sueldo
```

```
                openSalarySt.setInt(1, empNo);
                openSalarySt.setInt(2, (int) Math.round(currentSalary *
1.05));
                openSalarySt.executeUpdate();

                // Confirmamos los cambios para finalizar la transacción
                con.commit();
                System.out.println("Transacción realizada!");
            } catch (SQLException e) {
                // En caso de error, devolvemos la base de datos a su estado
inicial
                System.err.print("Se ha producido un error en la
transacción, deshacemos los cambios");
                con.rollback();
            } finally {
                // Finalmente reactivamos el autocommit
                con.setAutoCommit(true);
            }
        } catch (SQLException e) {
            System.err.println("Error estableciendo conexión: " +
e.getMessage());
        }
    }
}
```

EXTRA 1. Llamada a procedimientos almacenados

Un CallableStatement se usa para llamar a procedimientos almacenados en una base de datos.

Las principales ventajas del uso de procedimientos almacenados son:

- Reutilización de los procedimientos en diversas aplicaciones y diferentes lenguajes que accedan a la BD.
- Simplificación de las modificaciones de las consultas en el caso de que cambie el esquema de la BD.
- Optimización de algunas consultas si se ejecutan dentro del mismo espacio de memoria que el servidor de la base de datos, como un procedimiento almacenado.

A un procedimiento almacenado podemos pasarle parámetros y recoger los valores devueltos.

Para poder ejecutar procedimientos almacenados, el usuario tiene que tener permisos y en MySQL/MariaDB esto se consigue dando el permiso de select sobre la taula proc de mysql (ya que es allí dónde se guardan estos procedimientos y funciones).

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}");

callableStatement.setString(1, "param1");
callableStatement.setInt   (2, 123);

callableStatement.registerOutParameter(1, java.sql.Types.VARCHAR);
callableStatement.registerOutParameter(2, java.sql.Types.INTEGER);

ResultSet result = callableStatement.executeQuery();
while(result.next()) { ... }

String out1 = callableStatement.getString(1);
int    out2 = callableStatement.getInt   (2);
```

EXTRA 2. Acceso a metadatos de la BD

A través de la interfaz DatabaseMetaData se pueden obtener metadatos sobre la base de datos a la que nos hemos conectado.

Por ejemplo, podemos ver las tablas definidas en la base de datos, las columnas de cada tabla, si las características dadas son compatibles, etc.

Más información sobre metadatos.

```
import java.sql.*;

public class Metadatos {
   public static void muestraInfoBD(DatabaseMetaData dbmd) throws
SQLException {
       String nombre = dbmd.getDatabaseProductName();
       String driver = dbmd.getDriverName();
       String url = dbmd.getURL();
       String usuario = dbmd.getUserName();

       System.out.println("Información de la BD:");
       System.out.println("Nombre: "+nombre);
       System.out.println("Driver: "+driver);
       System.out.println("URL: "+url);
       System.out.println("Usuario: "+usuario);
   }
   public static void muestraTablas(DatabaseMetaData dbmd) throws
SQLException {
       try (ResultSet rs = dbmd.getTables("employees", null, null, null);)
{
           // Obtener sólo las tablas:
           // String[] types = {"TABLE"};
           // rs = dbmd.getTables(null, null, null, types);
           while (rs.next()) {
               String catalogo = rs.getString(1);
               String esquema = rs.getString(2);
               String tabla = rs.getString(3);
               String tipo = rs.getString(4);
               System.out.println(tipo + " - Catálogo: " + catalogo +
                       ", Esquema: "+esquema+", Nombre: "+tabla);
           }
       }
   }

   public static void muestraColumnas(DatabaseMetaData dbmd, String tabla)
throws SQLException {
       System.out.println("Columnas de la tabla: "+tabla);
```

```
        try (ResultSet columnas = dbmd.getColumns(null, "employees", tabla,
null);) {
            while (columnas.next()) {
                String nombre = columnas.getString("COLUMN_NAME");
                String tipo = columnas.getString("TYPE_NAME");
                String tamaño = columnas.getString("COLUMN_SIZE");
                String nula = columnas.getString("IS_NULLABLE");
                System.out.println("Columna: "+nombre+", Tipo: "+tipo+",
Tamaño: "+tamaño+", ¿Puede ser nula? "+nula);
            }
        }
    }




    public static void main(String[] args) {
        DatabaseMetaData dbmd = null;

        try (Connection connection = DriverManager.getConnection
                ("jdbc:mysql://localhost/employees","root","1234");) {
            dbmd = connection.getMetaData();
            muestraInfoBD(dbmd);
            muestraTablas(dbmd);
            muestraColumnas(dbmd, "departments");
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

EXTRA 3. Actualizaciones en lote

JDBC Batch Updates es un conjunto de actualizaciones agrupadas y enviadas a la base de datos en un lote. Es más rápido que enviarlas una a una, esperando que termine cada una:

- Menos tráfico de red en el envío de actualizaciones (solo un viaje de ida y vuelta).
- Se podrían ejecutar actualizaciones en paralelo.

Podemos usarlo con Statement y PreparedStatement.

```
Statement statement = null;

try{
    statement = connection.createStatement();

    statement.addBatch("update people set firstname='John' where id=123");
    statement.addBatch("update people set firstname='Eric' where id=456");
    statement.addBatch("update people set firstname='May'  where id=789");

    int[] recordsAffected = statement.executeBatch();
} finally {
    if(statement != null) statement.close();
}
```