

**IES**

**Francisco de Goya**

**1º DAW. Programación**

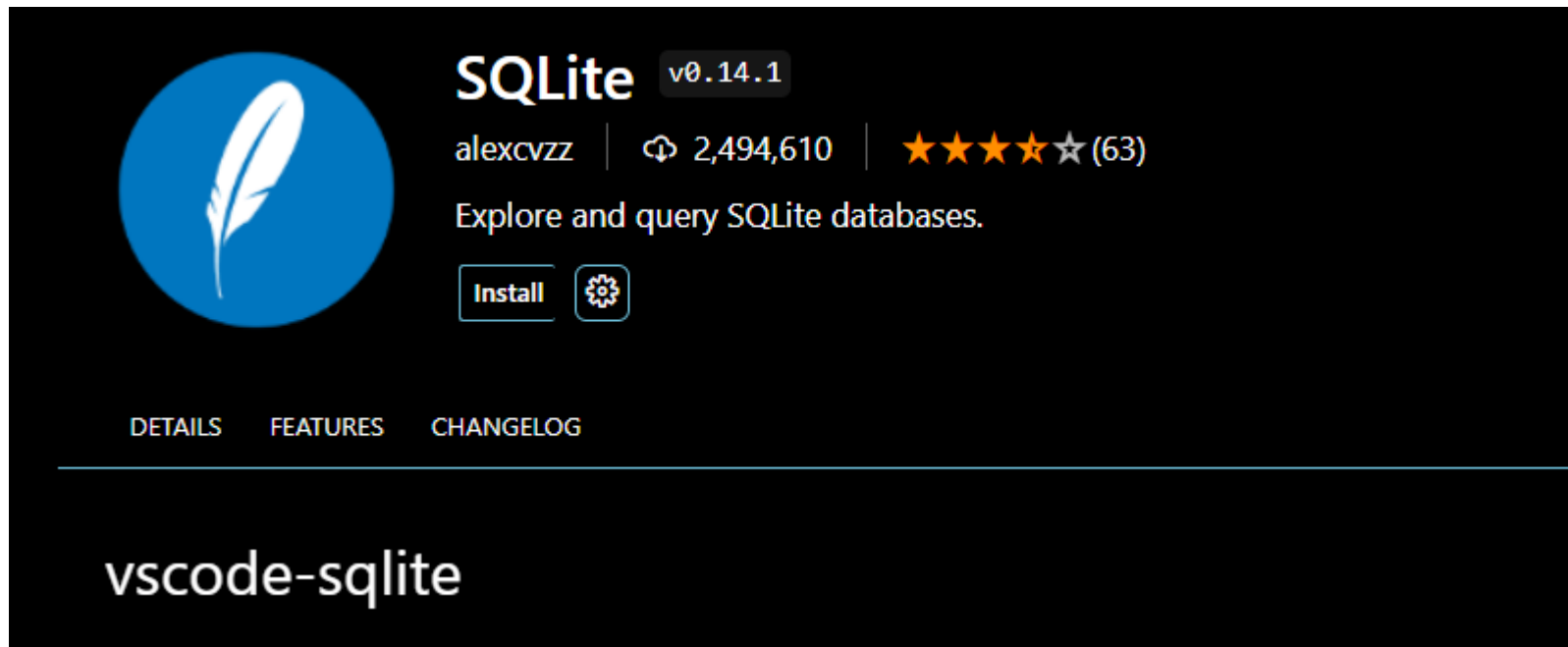


***Unidad de Trabajo 11: Manejo de Bases de Datos Relacionales.***

## *Introducción*

- ✗ En esta Unidad de Trabajo vamos a ver cómo consultar o manipular información en una Base de Datos.
- ✗ Puedes utilizar cualquier tipo de Base de Datos:
  - Oracle
  - MySQL
  - SQLite...
- ✗ Podemos conectarnos a la base de datos y ejecutar las diferentes sentencias selects, insert, update, delete, DML (create, alter...), DB functions...
- ✗ Usaremos **JDBC**: Java Database Connectivity que es una API con diferentes objetos para realizar estas tareas y con diferentes controladores para conectarse a los diferentes tipos de bases de datos.

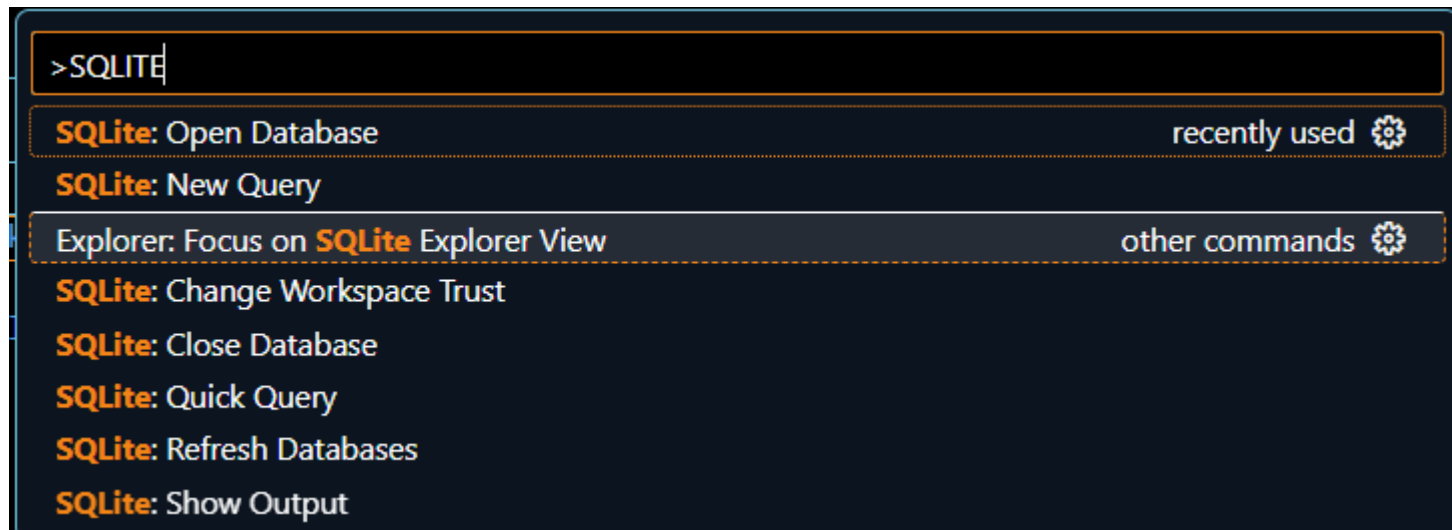
También vamos a instalar una extensión que nos permitirá de forma gráfica, ejecutar sentencias DDL, DML y crear Bases de Datos.



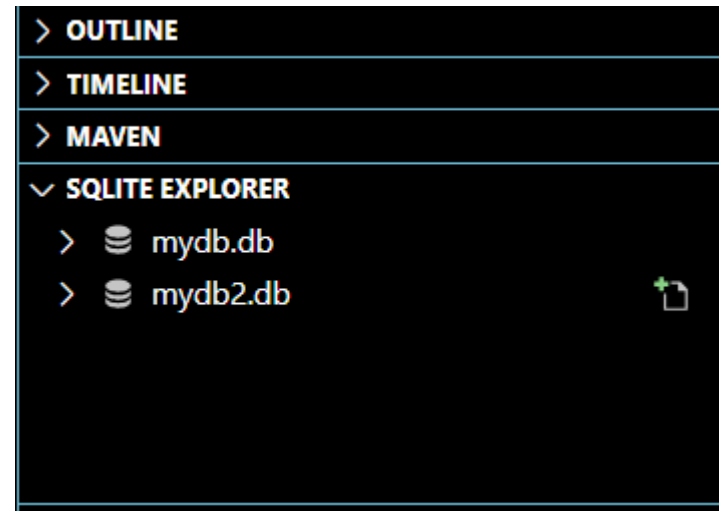
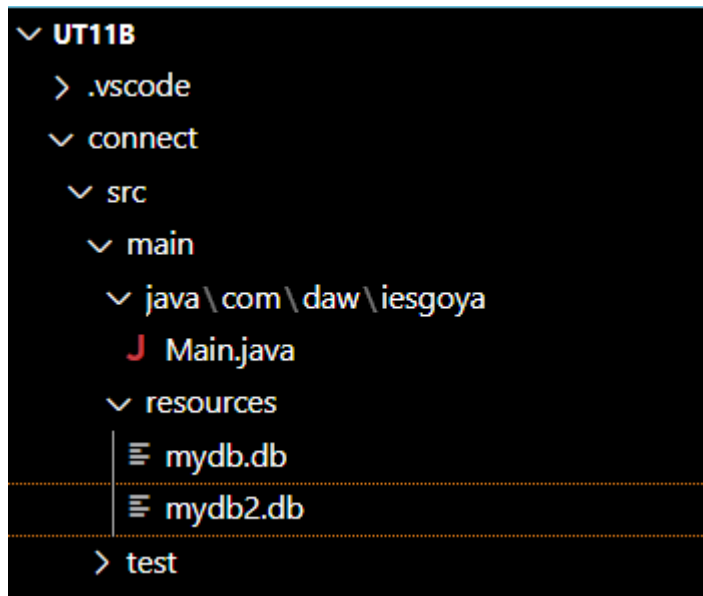
Recargamos la extensión: desde la paleta de Comandos buscamos reload:

>developer: reload Windows

Una vez instalada utilizando la paleta de comandos de VSCODE podemos acceder a los comandos de esta extensión:

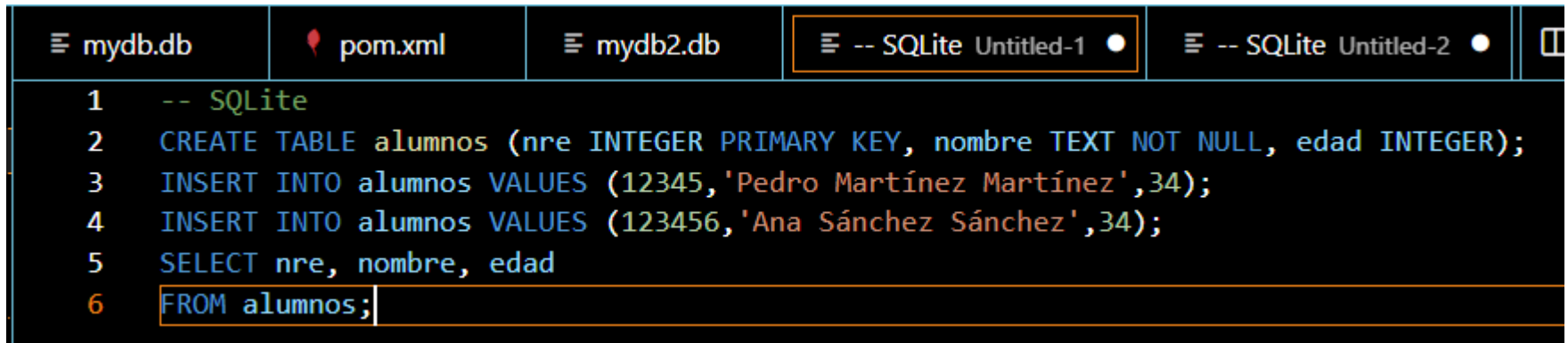


Nos creamos un nuevo fichero con **extensión.db** que albergará nuestra base de datos.



Al hacer click en el explorador de SQLITE EXPLORER sobre el fichero que contendrá la base de datos nos crea un nuevo fichero para ejecutar sentencias DDL o DML

Podemos escribir SQL en el mismo y ejecutarlo (botón derecho para abrir menú contextual).



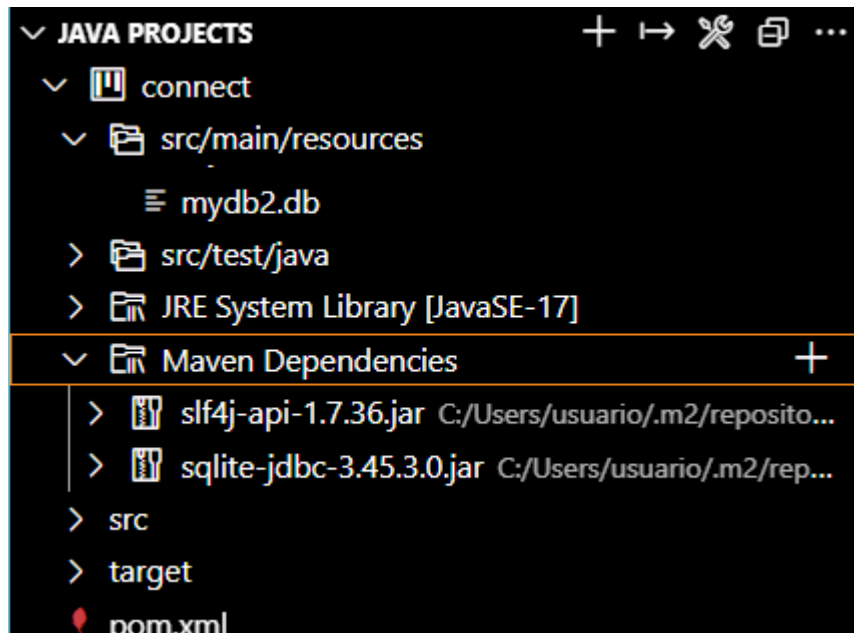
```
1  -- SQLite
2  CREATE TABLE alumnos (nre INTEGER PRIMARY KEY, nombre TEXT NOT NULL, edad INTEGER);
3  INSERT INTO alumnos VALUES (12345, 'Pedro Martínez Martínez', 34);
4  INSERT INTO alumnos VALUES (123456, 'Ana Sánchez Sánchez', 34);
5  SELECT nre, nombre, edad
6  FROM alumnos;
```

nre	nombre	edad
12345	Pedro Martínez Martínez	34
123456	Ana Sánchez Sánchez	34

Para acceder desde Java a nuestra base de datos necesitamos el driver JDBC de SQLite, buscamos la dependencia SQLite jdbc en nuestro Proyecto/Maven:



- Le estamos diciendo a Maven que se encargue de localizar el driver JDBC de SQLite y agregarlo a nuestro proyecto



## *Conexiones SQLite: código para conectarse*

```
String driver = "org.sqlite.JDBC";
String url = "jdbc:sqlite:E:/MiBD/daw.db";
Connection conn = null;
try {
    Class.forName(driver);
    conn = DriverManager.getConnection(url);
    System.out.println("Connection established");
}
catch (ClassNotFoundException | SQLException ex) {
    ex.printStackTrace();
}
```

✖ Lanza el siguiente error (en caso de no localizar el el driver):

```
java.lang.ClassNotFoundException: org.sqlite.JDBC
...
    at java.base/java.lang.Class.forName(Class.java:376)
    at com.daw.programming.JDBCExample.main(JDBCExample.java:21)
```



## *Conexiones SQLite: dependencia controlador maven*

- ✗ Esto se debe a que no encuentra el controlador “org.sqlite.JDBC”.
- ✗ Para agregarlo a nuestro proyecto, dado que nuestro proyecto es un proyecto “Maven”, podemos incorporar el controlador como una “dependencia maven”.
- ✗ Para ello, buscamos en Google: “dependencia de sqlite maven”.
- ✗ Aparece la siguiente página: <https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc>
- ✗ Si hacemos clic en él y hacemos clic en la última versión, es decir:

```
<dependencia>  
  <groupId>org.xerial</groupId>  
  <artifactId>sqlite-jdbc</artifactId>  
  <versión>3.41.2.1</versión>  
</dependencia>
```

- ✗ En el archivo “**pom.xml**” en la sección “archivos de proyecto” del proyecto, en una sección <dependencias> (créela si no existe).

## **Conexiones SQLite: la ruta no existe**

✗ Ahora, si ejecutamos el código nuevamente, arroja el siguiente error:

```
java.sql.SQLException:path  
to'E:/MiCarpeta/daw.db': 'E:\MiBD' no existe
```

✗ En este caso, el error es muy claro. Quizás, si creamos la carpeta en la unidad “E:\”, se ejecute.

```
Conexión establecida
```

✗ En el sistema de archivos, si vamos a la ruta “E:\MiBD” podremos ver que se ha creado un nuevo archivo “daw.db”.

✗ Entonces ya tenemos nuestra conexión SQLite.

**Desafío11\_1,11\_2**

## *Creando una tabla*

✗ Podemos crear una tabla con:

```
String sql = "CREATE TABLE users (" +  
            "    username TEXT PRIMARY KEY," +  
            "    password TEXT" +  
            ");";  
  
PreparedStatement ps = conn.prepareStatement(sql);  
ps.executeUpdate();  
conn.close();
```

✗ Podemos ver que la tabla se ha creado con “DB Browser for SQLite en Netbeans o con la extensión de VSCode.

✗ Método `executeUpdate()` // Consultas de creación o modificación de datos no para consultas `SELECT` o que devuelvan datos

### **Reto11\_3**

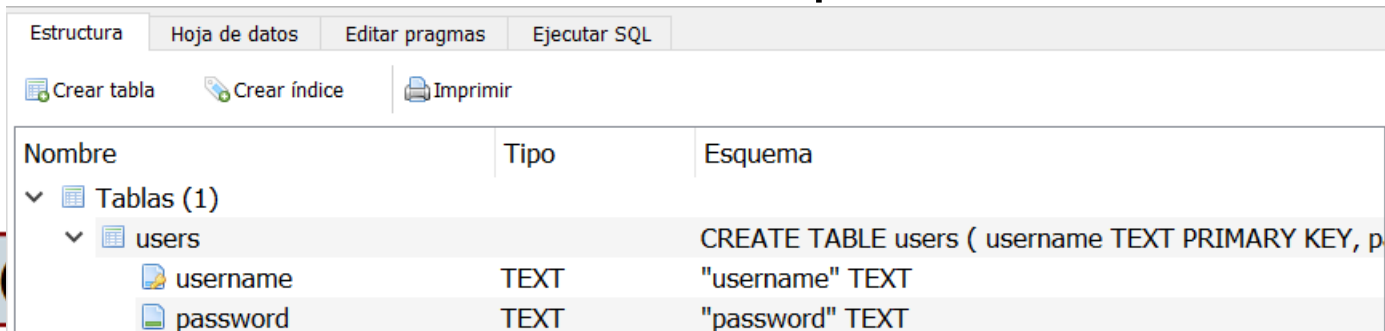
## ***Instalar DB Browser para SQLite y abrir DB.***

- ✗ Se puede descargar desde: <https://sqlitebrowser.org/>
- ✗ Haga clic en "Descargar" > "DB Browser para SQLite - Aplicación portátil".
- ✗ Vaya a "Descargas" y haga clic en "SQLiteDatabaseBrowserPortable-3.12.2\_English.paf.exe"
- ✗ "Siguiente" > En "Carpeta de destino" elige tu "disco duro externo" o "pendrive" > "Instalar" > "Finalizar".

✗ Haga clic en "Abrir base de datos" 

✗ Seleccione su base de datos, es decir: "E:/MiBD/daw.db".

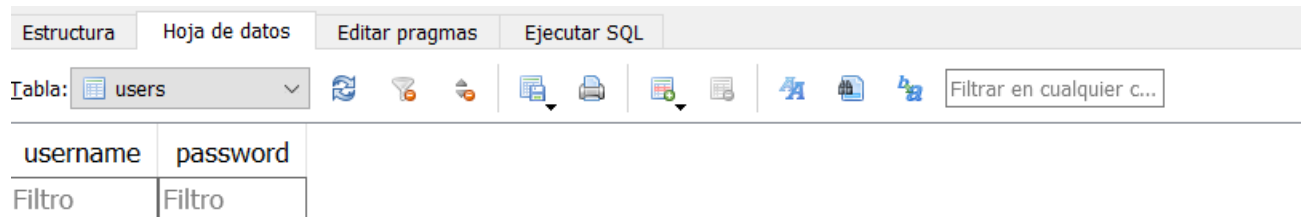
✗ Puedes ver la tabla "USUARIOS" que hemos creado:



Estructura		
Hoja de datos   Editar pragmas   Ejecutar SQL		
Crear tabla   Crear índice   Imprimir		
Nombre	Tipo	Esquema
▼ Tablas (1)		
▼ users		CREATE TABLE users ( username TEXT PRIMARY KEY, password TEXT )
username	TEXT	"username" TEXT
password	TEXT	"password" TEXT

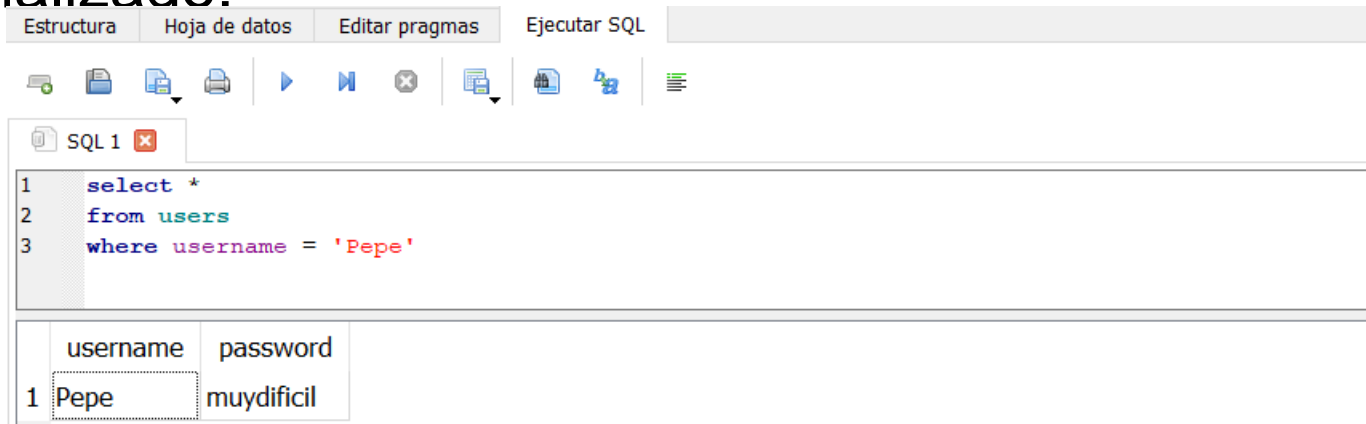
## *Navegación y manipulación de datos.*

✗ Si haces clic en “Hoja de Datos” podrás ver los datos de la tabla seleccionada.

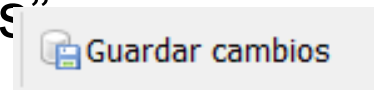


✗ Allí podrás insertar, actualizar o eliminar, filtrar información...

✗ En la pestaña “Ejecutar SQL” puedes ejecutar SQL personalizado:



✗ Puedes guardar cambios con “Guardar Cambios”



## *Insertar datos en la tabla*

```
try {
    String sql = "INSERT INTO users (username, password) values (?, ?);";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, "Pepe");
    ps.setString(2, "muydifícil");
    int numRows = ps.executeUpdate();
    System.out.println(numRows + " row(s) inserted.");
}
catch (SQLException ex) {
    ex.printStackTrace();
}
finally {
    try {
        if (conn!=null) {
            conn.close();
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

**Reto11\_4**

## *Cerrando la conexión*

- ✗ En la diapositiva anterior podemos ver que “`conn.close()`” está dentro del bloque “`finally`”.
- ✗ El bloque “`finally`” asegura que la conexión se va a cerrar sin importar si hay una excepción en alguna de las líneas anteriores.
- ✗ Por ejemplo, si intentamos insertar un registro y la clave principal para este registro ya existe, se generará una excepción porque no podemos insertar 2 filas con la misma clave principal.
- ✗ En este caso, “`executeUpdate`” generará la excepción y las siguientes líneas no se ejecutarán.
- ✗ Pero, debido a que nuestro “`conn.close()`” está dentro del bloque “`finally`”, esto asegura que la conexión se cerrará aunque ocurra una excepción.

## *Consultar información de la tabla.*

```
try {
    String sql = "select * from users;";
    PreparedStatement ps = conn.prepareStatement(sql);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        System.out.print(rs.getString("username"));
        System.out.print(", ");
        System.out.println(rs.getString("password"));
    }
    rs.close();
}
catch (SQLException ex) {
    ex.printStackTrace();
}
finally {
    try {
        if (conn!=null)
            conn.close();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

**Reto11\_5**



## ***Autocommit y transacciones (1/2)***

- ✗ En JDBC, autocommit está habilitado de forma predeterminada. Es por eso que no necesitamos realizar un commit explícito. Después de cada sentencia, se ejecuta un commit automáticamente.
  
- ✗ Pero, si queremos crear una transacción en la que necesitamos ejecutar varias oraciones como una transacción atómica, debemos deshabilitar la confirmación automática y ejecutar la confirmación explícitamente.
  
- ✗ Por ejemplo, si queremos insertar todas las notas de un estudiante en una sola transacción, de modo que, si una inserción falla, todas las oraciones se “reviertan”, tenemos que:
  - ✗ Deshabilitar “autocommit”.
  - ✗ Hacer rollback si falla una sentencia(se produce una excepción).
  - ✗ Hacer commit si todas las sentencias se ejecutan correctamente.

## Confirmación automática y transacciones (2/2)

```
try {
    conn.setAutoCommit(false);
    String sql = "INSERT INTO marks (username, module, mark) values (?, ?, ?);";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, "Juan");
    ps.setString(2, "Base de Datos");
    ps.setInt(3, 4);
    int numRows = ps.executeUpdate();
    ps.setString(1, "Juan");
    ps.setString(2, "Programación");
    ps.setInt(3, 6);
    numRows = ps.executeUpdate();
    conn.commit(); → Si todo va bien, confirmamos toda la transacción.
}
catch (SQLException ex) {
    try {
        if (conn!=null)
            conn.rollback(); → Si algo sale mal, revertimos (rollback) la transacción "completa"
    } catch (SQLException ex2) {
        ex2.printStackTrace();
    }
    ex.printStackTrace();
}
finally {
    // Close the connection ...
}
```

Primer inserto: Juan, Base de Datos, 4

Segundo inserto: Juan, Programación, 6  
*No tenemos que volver a "preparar" la frase ya que es el mismo insert.*

## *Autocommit y velocidad (1/2)*

✗ Imagínate, tienes que insertar 100 registros:

```
long initialMillis = System.currentTimeMillis();
String sql = "INSERT INTO marks (username, module, mark) values (?, ?, ?);";
PreparedStatement ps = conn.prepareStatement(sql);
for (int i=0; i<100; i++) {
    ps.setString(1, "Froilan");
    ps.setString(2, "Modulo"+i);
    ps.setInt(3, 4);
    int numRows = ps.executeUpdate();
}
long finalMillis = System.currentTimeMillis();
System.out.println("Time: "+(finalMillis-initialMillis)/1000+" secs");
```

✗ La ejecución tardará unos 15 segundos:

Tiempo: 15 segundos

✗ Vamos a acelerar el proceso. Primero, elimine los registros:

```
delete from marks where username = 'Froilan';
```

## *Autocommit y velocidad (2/2)*

✗ Ahora, deshabilite autocommit e inserte los mismos 100 registros y confirme al final:

```
conn.setAutoCommit(false);
long initialMillis = System.currentTimeMillis();
String sql = "INSERT INTO marks (username, module, mark) values (?, ?, ?);";
PreparedStatement ps = conn.prepareStatement(sql);
for (int i=0; i<100; i++) {
    ps.setString(1, "Froilan");
    ps.setString(2, "Modulo"+i);
    ps.setInt(3, 4);
    int numRows = ps.executeUpdate();
}
conn.commit();
long finalMillis = System.currentTimeMillis();
System.out.println("Time: "+(finalMillis-initialMillis)/1000+" secs");
```

✗ Reducirá el tiempo a 0 segundos:

Tiempo: 0 segundos

## *Evitar el ciberataque de SQL injection (1/4)*

✗ Imagine el siguiente código, si el usuario ingresa **Pepe**:

```
Scanner keyboard = new Scanner(System.in);
System.out.println("Introduzca el usuario deseado: ");
String user = keyboard.nextLine();
String sql = "select * from marks where username = '"+user+"'";
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.print(rs.getString("username"));
    System.out.print(", ");
    System.out.print(rs.getString("module"));
    System.out.print(", ");
    System.out.println(rs.getString("mark"));
}
rs.close();
rs.cerrar();
```

✗ Tendremos las marcas de Pepe:

```
Pepe, Base de Datos, 8
Pepe, Programación, 7
```

## *Evitar el ciberataque de SQL injection (2/4)*

✗ Pero, si el usuario inserta ' OR 1=1 --

```
' union select username, password from users --
```

✗ El atacante obtendrá la información de todos los registros de la tabla.

```
Introduce un nombre de usuario
' OR 1=1 --
select * from marks where username=' ' OR 1=1 --';
Pedro Perez, Bases de Datos,10.0
Noelia Perez, Programacion,10.0
Pedro, Bases de Datos,6.0
Pedro, Programacion,6.0
Pedro, Sistemas Informaticos,7.0
```

✗ Podemos evitar esto usando el símbolo "?" para agregar parámetros a la oración.(1). Mira otro possible ataque:

```
select * from marks where username = ' ' union select username, password,
5 from users -- '
```

**Desafío11\_6...**

## *Evitar el ciberataque de SQL injection (3/4)*

Introduce un nombre de usuario

```
' UNION SELECT username, password, 5 from users --  
select * from marks where username='' UNION SELECT username, password, 5 from users --';  
USER1, PASSWORD1,5  
USER2, PASSWORD2,5  
USER3, PASSWORD3,5  
USER4, PASSWORD4,5
```

(1) Utilizando ? y los métodos “setString”, “setInt”...el driver transforma cualquier carácter problemático como la comilla simple ( ' ) y otros, escapándolos, haciendo que ese carácter no transforme nuestra sentencia.

## *Evitar el ciberataque de SQL injection (4/4)*

[https://www.w3schools.com/sql/sql\\_injection.asp](https://www.w3schools.com/sql/sql_injection.asp)