

UT2. Parte 2: Trabajando con JDBC y SQLite

Cuando desarrollamos pequeñas aplicaciones en las que no vamos a almacenar grandes cantidades de información, no es necesario que utilicemos un Sistema Gestor de Bases de Datos como *Oracle*, *MySQL* o *Postgres*. En su lugar podemos utilizar bases de datos embebidas, que tienen la particularidad de que el motor está incrustado en la aplicación y es exclusivo para ella. La base de datos se inicia cuando se ejecuta la aplicación y termina cuando se cierra la aplicación. Algunos ejemplos de bases de datos embebidas son: *SQLite*, *Apache Derby*, *HSQLDB*, *H2* y *Db4o*.

Para este apartado vamos a utilizar SQLite, que es un proyecto de dominio público cuyo motor, contenido en una relativamente pequeña biblioteca escrita en C y compatible con **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad), nos permite trabajar con bases de datos relacionales de una forma sencilla.

1. Instalación de SQLite 3 en Linux

En sistemas Linux podemos instalar SQLite ejecutando el comando:

```
$ sudo apt install sqlite3
```

En sistemas Mac, SQLite viene preinstalado en las últimas versiones.

En cualquier caso podemos descargar desde la **página oficial** (<https://github.com/xerial/sqlite-jdbc/releases>) un archivo comprimido, cuyo contenido se muestra a continuación:

- **sqlite3**: el programa gestor de la base de datos.
- **sqldiff**: muestra diferencias entre dos bases de datos.
- **sqlite3_analyzer**: muestra un análisis del espacio consumido por la base de datos en aras de mejorar la eficiencia.

El contenido de esta carpeta debe almacenarse en algún directorio accesible desde todo el Sistema o añadirlo a la variable PATH.

2. Creación de una base de datos y de una tabla

1. Las bases de datos SQLite habitualmente se guardan en un archivo con extensión `.sqlite` o `.db`

Ejecutamos:

```
$ sqlite3 nuevaBD.sqlite # Otra extensión habitual es .db
```

```
alumno@ubuntu-alumno:~$ sqlite3 nuevaBD.sqlite
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
sqlite> 
```

Con este comando creamos la BD `nuevaBD.sqlite`. Pasamos a crear una tabla 'persona' con este otro comando:

```
sqlite> create table persona(
...> dni varchar(9) primary key,
...> nombre varchar(50),
...> edad int);
sqlite>
```

La sentencia SQL para crear una tabla es igual que la que se utilizaría en un SGBD.

Para ejecutar comandos internos en SQLite, tienen que ir precedidos de un punto (.). Con `.help`, por ejemplo, podemos acceder a la ayuda. Para ver si hemos creado correctamente la tabla, ejecutaremos el comando `.tables`:

```
sqlite> .tables
persona
```

Para ver la estructura de una tabla (más concretamente, su script de creación), utilizamos `.schema nombreTabla`.

```
sqlite> .schema persona
CREATE TABLE persona(
dni varchar(9) primary key,
nombre varchar(50),
edad int);
```

Insertamos unas cuantas filas y consultamos:

```
sqlite> insert into persona values("12345678A","Marco Antonio",34);
sqlite> insert into persona values("4567891A","Cleopatra", 41);
sqlite> insert into persona values("78912345A","Godofredo", 39);
sqlite> select * from persona;
12345678A|Marco Antonio|34
4567891A|Cleopatra|41
78912345A|Godofredo|39
```

El comando `.mode` permite cambiar el modo de salida, en texto, columnas e incluso html:

```
sqlite> .mode column
sqlite> select * from persona;
dni          nombre          edad
-----
12345678A    Marco Antonio    34
4567891A     Cleopatra        41
78912345A    Godofredo        39
```

Podemos salir con `.quit`:

```
sqlite> .quit

alumno@ubuntu-alumno:~$
```

Observamos que, en el contenido de nuestra carpeta, aparece un nuevo archivo denominado `nuevaBD.sqlite`:

```
alumno@ubuntu-alumno:~$ ls -lh
total 2,8M
drwxr-xr-x 2 alumno alumno 4,0K oct 14 11:14 Descargas
drwxr-xr-x 3 alumno alumno 4,0K sep 27 10:36 Documentos
drwxr-xr-x 2 alumno alumno 4,0K sep  8 19:49 Escritorio
drwxr-xr-x 2 alumno alumno 4,0K sep  8 19:49 Imágenes
drwxr-xr-x 2 alumno alumno 4,0K sep 21 16:33 Música
drwxr-xr-x 7 alumno alumno 4,0K oct 14 10:32 NetBeansProjects
-rw-r--r-- 1 alumno alumno 12K oct 15 09:47 nuevaBD.sqlite
drwxr-xr-x 2 alumno alumno 4,0K sep  8 19:49 Plantillas
drwxr-xr-x 2 alumno alumno 4,0K sep  8 19:49 Público
-rw-rw-r-- 1 alumno alumno 2,7M jul 24 08:36 setup-lightshot.exe
drwx----- 5 alumno alumno 4,0K sep 19 17:46 snap
drwxr-xr-x 2 alumno alumno 4,0K sep  8 19:49 Videos
```

3. Ejecución de un script desde línea de comandos

Nota: movemos nuevaBD.sqlite a una carpeta en Documentos que llamaremos bbdd.

Ejemplo: users.sql

Contenido del fichero:

```
create table users (
    username varchar(20),
    password varchar(40) not null,
    primary key (username)
);
```

Lo creamos (con gedit, por ejemplo) y guardamos en /home/alumno/Documentos/bbdd:



Accedemos a nuestra base de datos sqlite:

```
alumno@ubuntu-alumno:~/Documentos/bbdd$ sqlite3
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open nuevaBD.sqlite
sqlite>
```

Podemos acceder directamente a una base de datos especificando su nombre a continuación del comando sqlite:

```
sqlite> .quit
alumno@ubuntu-alumno:~/Documentos/bbdd$ sqlite3 nuevaBD.sqlite
SQLite version 3.37.2 2022-01-06 13:25:41
Enter ".help" for usage hints.
sqlite> .tables
persona
sqlite>
```

Ejecutamos el script users.sql:

```
sqlite> .shell ls      <-- Listamos (ls) lo que tenemos
```

```

                                (ejecuta en la Shell del sistema el comando ls)

nuevaBD.sqlite    users.sql

sqlite> .read users.sql      <-- Carga el script y lo ejecuta

sqlite> .tables            <-- vemos qué tablas tenemos

persona  users

sqlite> .schema users      <-- vemos script de creación tabla users

CREATE TABLE users (
    username varchar(20),
    password varchar(40) not null,
    primary key (username)
);

sqlite>

```

Insertamos algunos registros en la tabla users:

```

sqlite> insert into users values
("Juanin", "@ju4n123"),
("Homer", "S1mpS@n"),
("Lorenzo", "L0r3nz@");

```

Comprobamos:

```

sqlite> .mode column
sqlite> select * from users;
username  password
-----  -
Juanin    @ju4n123
Homer     S1mpS@n
Lorenzo   L0r3nz@
sqlite>

```

4. Instalación de SQLite Browser en Linux

Aparte de trabajar desde la línea de comandos, también podemos instalar una herramienta gráfica llamada SQLite, para trabajar con bases de datos SQLite.

Actualizamos los paquetes de Ubuntu:

```

$ sudo apt update
$ sudo apt upgrade

```

Instalamos SQLite browser:

```

$ sudo apt install sqlitebrowser -y

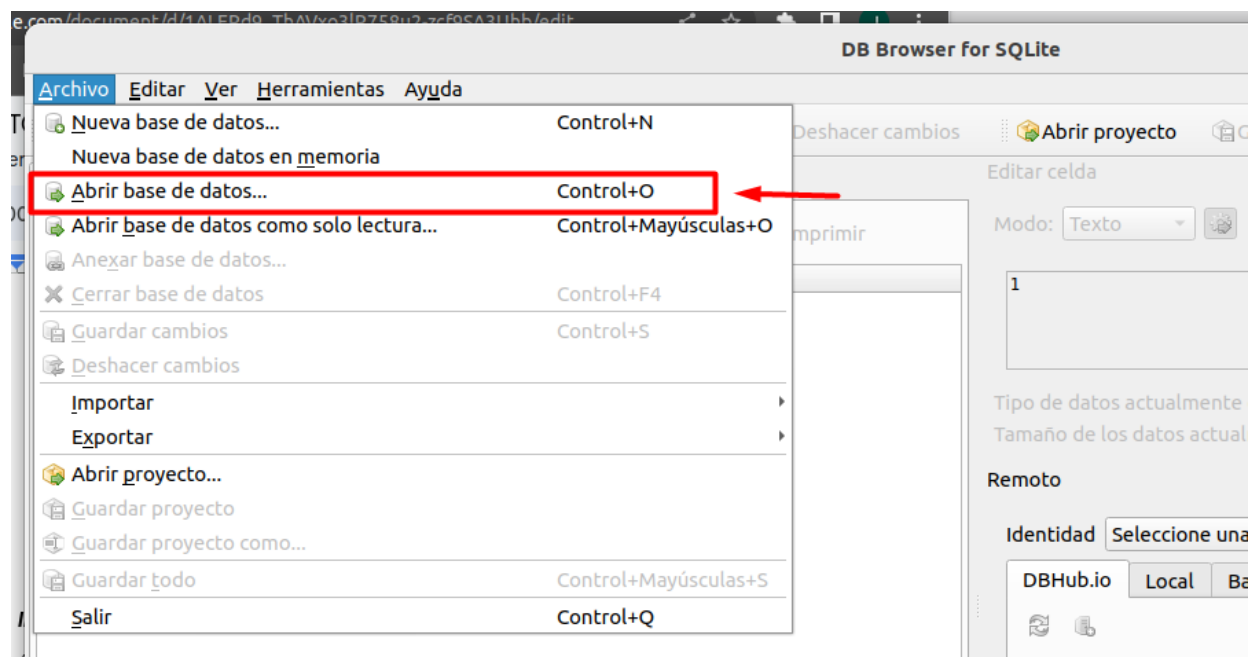
```

La opción -y responde automáticamente "yes" a cualquier pregunta que surja durante la instalación.

Abrimos la aplicación:



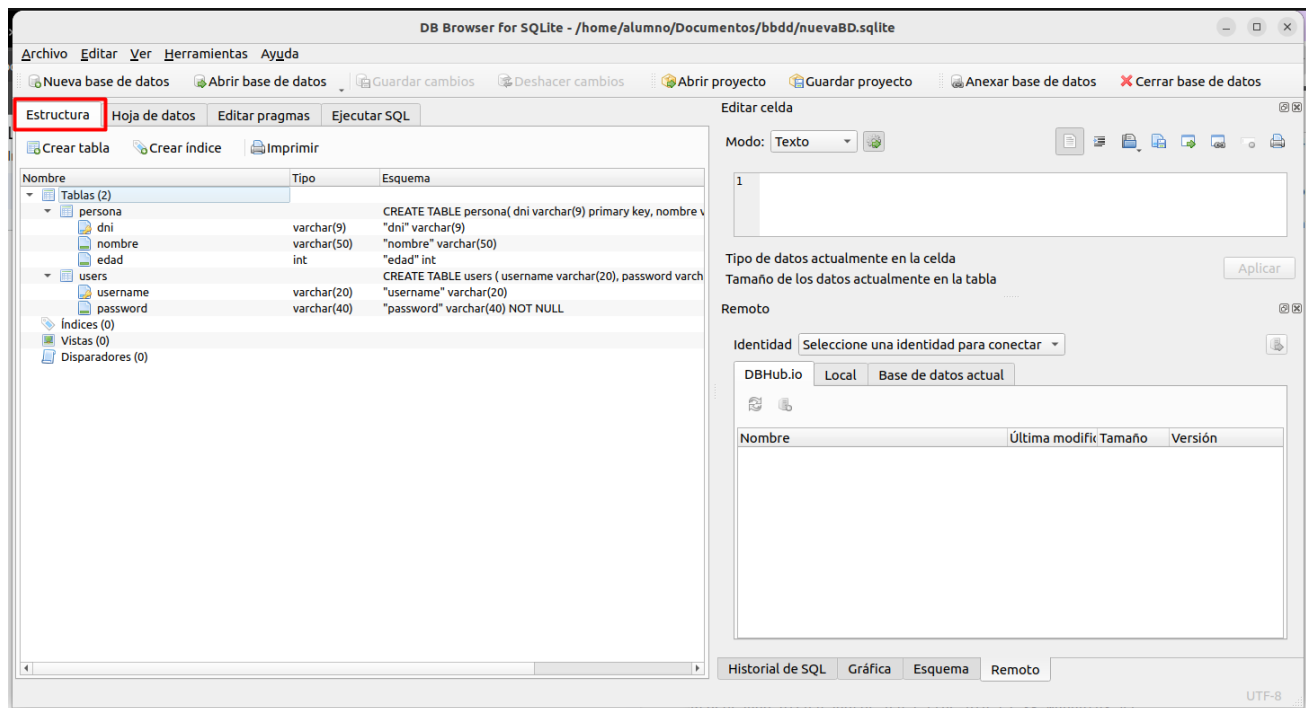
Abrimos nuestra base de datos nuevaBD.sqlite:



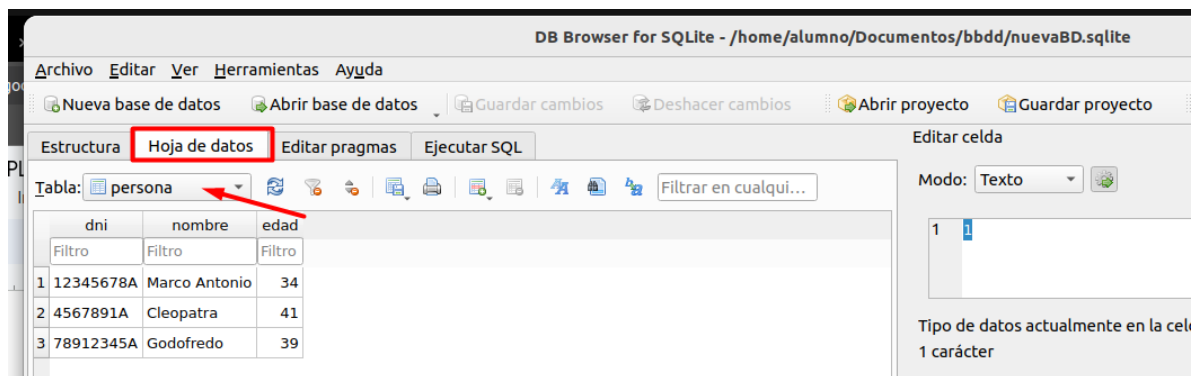
Seleccionamos el archivo nuevaBD.sqlite:



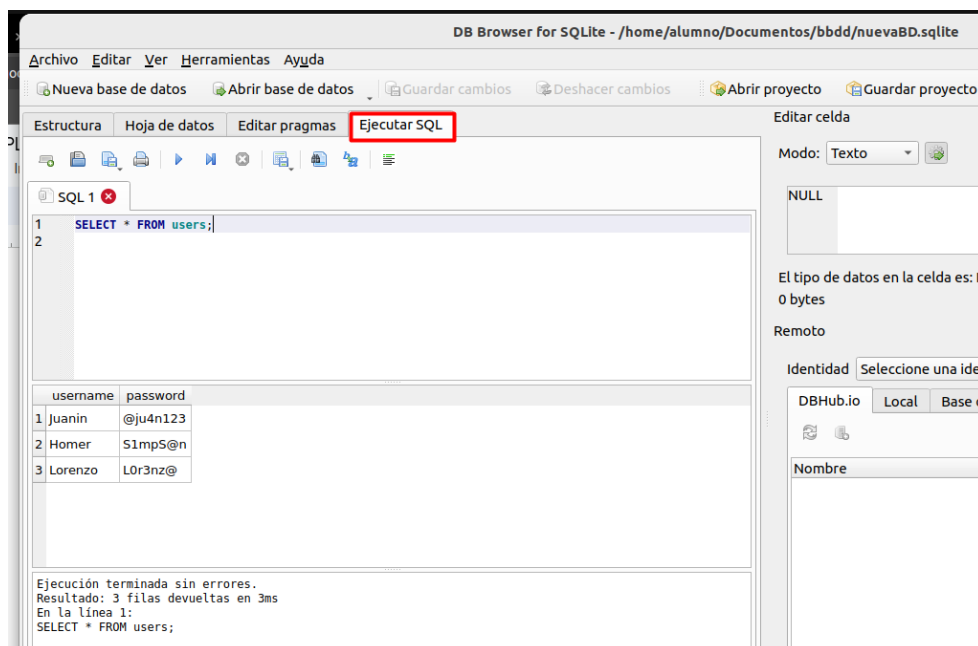
Aparecerá la estructura de nuestra base de datos con las tablas persona y users:



Pestaña 'Hoja de datos' (ver registros de una tabla):



Pestaña 'Ejecutar SQL' (ejecutar sentencias SQL):



5. Creación de un proyecto para trabajar con una BD SQLite

- Creamos un proyecto nuevo en NetBeans, de nombre **UT2_DEMO001_SQLITE** y en él un paquete **app**. Creamos en dicho paquete una Java Main Class, llamada **DemoSQLite**.
- Creamos una base de datos **SQLite** de nombre **ejemplo.db** en la siguiente ruta dentro del directorio del proyecto: `./src/db/ejemplo.db`

Conexión a la base de datos

- Vamos a crear una conexión desde nuestro proyecto Java a la base de datos SQLite método `getConnection()` de la clase `DriverManager` en su versión más básica, en la que sólo le pasamos al método la cadena de conexión:

```
static Connection getConnection(String url)
```

- Recordamos el formato genérico de una cadena de conexión:

```
jdbc:driver://servidor:puerto/nombre_base_datos
```

- En este caso, en el que vamos a utilizar una base de datos embebida (SQLite), no es necesario especificar host y puerto en la cadena de conexión (no tenemos un servidor).
- La **cadena de conexión** que tenemos que utilizar será:

```
jdbc:sqlite:./src/db/ejemplo.db
```



IMPORTANTE:

Para el correcto funcionamiento del ejemplo, tenemos que crear un paquete denominado **db** en nuestro proyecto (`./src/db/ejemplo.db`).

- Ejemplo de código para establecer la conexión a la BD (clase **DemoSQLite.java**). Se abre y se cierra la conexión en el main (en el ejemplo del main al final del documento se usa `try-with-resources`).

```

public static void main(String[] args) {

    // 1. Creamos cadena de conexión
    String url = "jdbc:sqlite:./src/db/ejemplo.db";
    // 2.1
    Connection conexion = null;

    try { //También se puede usar try-with-resources (ver ejemplo del main al final)
        // 3
        // Class.forName("org.sqlite.JDBC"); // <-- Desde Java 6 no hace falta
        // Class.forName("com.mysql.cj.jdbc.Driver"); // <-- Para MySQL
        conexion = DriverManager.getConnection(url);
        // ^ Si no existe la bdd, se crea
        System.out.println("Conexión con BD establecida");
        //} catch (ClassNotFoundException ex) { //Solo si ejecuta Class.forName
        // System.out.println("ClassNotFoundException generada: " + ex.getMessage());
    } catch (SQLException ex) {
        System.out.println("SQLException generada: " + ex.getMessage());
    } finally {
        // Cerramos la conexión con la base de datos
        try {
            if (conexion != null) {
                conexion.close();
            }
        } catch (SQLException ex) {
            System.out.println("Error al cerrar la conexión " + ex.getMessage());
        }
    }
}

```


- Si **ejecutamos** el código veremos que el resultado no es el esperado. No se crea la base de datos ni se muestra el mensaje "Conexión con BD establecida". Lo que ocurre es que nos falta añadir el archivo .jar (conector/driver) de SQLite al proyecto.

Lo descargamos en la máquina virtual Linux desde la dirección <https://github.com/xerial/sqlite-jdbc/releases/tag/3.36.0.3>.

Releases / 3.36.0.3




3.36.0.3





Compare

 xerial released this Aug 30, 2021 · 347 commits to master since this release · 3.36.0.3 · 5bc86e1

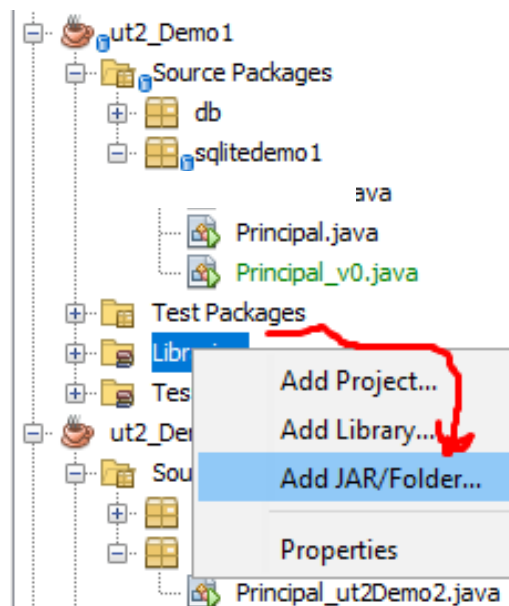
- Fixes for GraalVM
- Internal update: Migrate to JUnit5. Add CI with GraalVM

▼ Assets 3

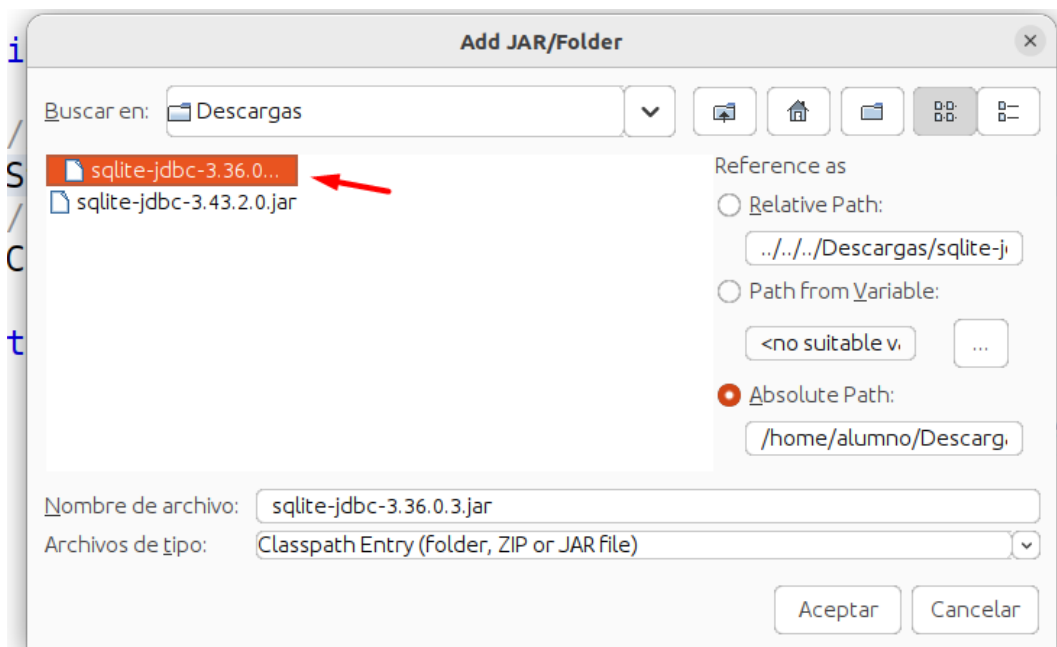
 sqlite-jdbc-3.36.0.3.jar	9.28 MB	Aug 30, 2021
 Source code (zip)		Aug 30, 2021
 Source code (tar.gz)		Aug 30, 2021

 10
  9
  1
  1
 15 people reacted

A continuación, hacemos clic en el botón derecho sobre Libraries, Add jar/folder...



...seleccionamos sqlite-jdbc-3.36.0.3.jar donde lo hayamos descargado...



Pinchamos Aceptar y, ahora sí, debería verse por consola el mensaje indicando la conexión exitosa con la base de datos:

```
Conexión con BD establecida
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

Pasos para ejecutar sentencias desde el programa Java con JDBC

El funcionamiento de un programa con JDBC requiere los siguientes pasos:

- 0 Importar las clases necesarias del paquete `java.sql`

1	Cargar driver	<code>Class.forName(nombre del driver); // No necesario desde JDBC 4.0 (Java SE 6)</code>	
2	Crear conexión	<code>Connection c = DriverManager.getConnection(datos de conexión);</code>	
3	Crear sentencia	<code>Statement s = c.createStatement();</code>	
4	Ejecutar sentencia Si consulta, obtener resultados fila a fila y cerrar lista de resultados	<code>ResultSet rs = s.executeQuery(consulta); while(rs.next()) { ... } rs.close();</code>	<code>int res = s.executeUpdate (sent. DML); (o bien) boolean res = s.execute (sent. DDL);</code>
5	Cerrar sentencia	<code>s.close();</code>	
6	Cerrar conexión	<code>c.close();</code>	

Vamos a crear en los siguientes apartados métodos para la realización de las operaciones clásicas con bases de datos SQLite.

Creación de una tabla

- Vamos a añadir a nuestro programa el **método crearTabla**, que nos permitirá crear una **tabla alumnos en la base de datos ejemplo.db**. El método recibe como parámetro el objeto `Connection`. Utilizará el método **execute()** de la interfaz **Statement** para ejecutar la sentencia SQL de creación de la tabla, controlando las posibles excepciones.
- Debemos tener en cuenta los **tipos de datos básicos** para los valores que podemos almacenar en **SQLite**, que son:
 - **INTEGER**: Este tipo de dato se utiliza para almacenar números enteros, ya sean con signo o sin signo. Puede representar valores enteros como 1, -42, 1000, etc.
 - **REAL**: Se utiliza para almacenar valores de punto flotante, es decir, números con decimales. Podemos usarlo para representar números como 3.14, -0.001, 2.71828, etc.
 - **TEXT**: El tipo **TEXT** se utiliza para almacenar texto, como cadenas de caracteres, palabras, frases o cualquier otro contenido de texto. Es adecuado para almacenar nombres, descripciones, direcciones, etc.
 - **BLOB**: Este tipo se utiliza para almacenar datos binarios, como imágenes, archivos, objetos serializados u otros datos no interpretados. Podemos usarlo para guardar información en formato binario.

- NULL: El tipo de dato NULL se utiliza para representar valores nulos o ausentes. Puede ser útil cuando no se tiene un valor válido para una columna en un registro.

Además, SQLite es bastante flexible en la conversión automática entre tipos de datos, lo que significa que, en muchos casos, podemos insertar datos de un tipo en una columna de otro tipo, y SQLite se encargará de la conversión según sea necesario.

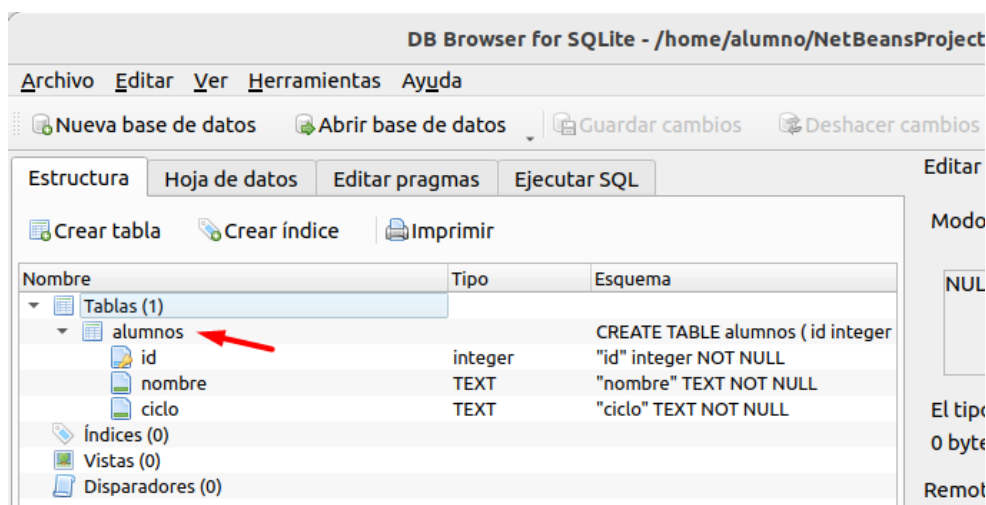
- **Nuestra tabla alumnos tendrá los siguientes campos:**

- *id*: identificador del alumno, de tipo entero, clave principal.
- *nombre*: nombre del alumno, de tipo TEXT, no puede ser NULL.
- *ciclo*: ciclo formativo en el que está matriculado, de tipo TEXT, no puede ser NULL.

Ejemplo de crear tabla:

```
private static void crearTabla(Connection conn) {
    // Creamos la sentencia SQL
    String sql = "CREATE TABLE IF NOT EXISTS alumnos (\n"
        + " id integer NOT NULL PRIMARY KEY,\n"
        + " nombre TEXT NOT NULL,\n"
        + " ciclo TEXT NOT NULL\n"
        + ");";
    // Creamos un objeto Statement a partir de la conexión que nos llega como parámetro
    try ( Statement s = conn.createStatement() ) {
        // Ejecutamos la consulta. Para DDL se suele utilizar execute.
        s.execute(sql);
        System.out.println("Tabla alumnos creada correctamente.");
    } catch (SQLException ex) {
        System.out.println("Error en crearTabla: " + ex.getMessage());
    }
} //Fin crearTabla
```

Si llamamos a este método en el método main, al ejecutar el programa se creará la tabla alumnos. Podemos comprobarlo mediante la línea de comandos o utilizando el programa instalado DBBrowser for SQLite:



Operaciones CRUD en SQLite con JDBC

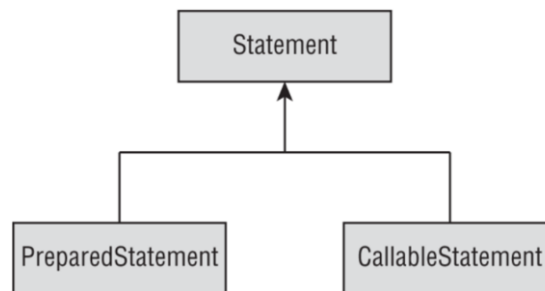
En los siguientes apartados vamos a realizar ejemplos de los 4 tipos de operaciones CRUD (Create, Read, Update, Delete).

Insertar registros en una tabla (INSERT)

- Vamos a crear el método `insertarAlumnos`, que nos va a permitir insertar registros en la base de datos `ejemplo.db`. Se insertarán los siguientes 4 registros en la tabla `alumnos`:

id	nombre	ciclo
1	Estudiante1	DAM
2	Estudiante2	DAW
3	Estudiante3	ASIR
4	Estudiante4	SMR

- El método recibe como parámetro el objeto `Connection`. Utilizará el método `executeUpdate()` de la interfaz **PreparedStatement**, y sus métodos `setXXX`, para ejecutar la sentencia `INSERT`.
- Podríamos utilizar tanto `Statement` como `PreparedStatement`, pero generalmente se considera más seguro y eficiente utilizar `PreparedStatement` en este tipo de escenario. A continuación veremos la diferencia entre ambas opciones



- Statement:** normalmente se utiliza para ejecutar instrucciones SQL que no contienen parámetros. Si bien podemos concatenar valores directamente en la cadena SQL, esto hace que nuestro código sea vulnerable a ataques de inyección SQL si los valores provienen de fuentes externas. Además, cada vez que ejecutamos una consulta, SQLite la compila nuevamente, lo que puede tener un impacto en el rendimiento si estamos realizando muchas inserciones.
- PreparedStatement:** se utiliza para ejecutar consultas parametrizadas. Podemos vincular valores a los parámetros de la consulta de manera segura, evitando la inyección SQL. Además, `PreparedStatement` precompila la consulta, lo que puede mejorar el rendimiento si vamos a realizar múltiples inserciones con la misma estructura de consulta, ya que la consulta se compila una vez y se puede reutilizar con diferentes valores.

Dado que en el ejemplo vamos a realizar una serie de inserciones con valores variables (id's, nombres y ciclos de formación profesional), es más apropiado utilizar `PreparedStatement` para garantizar la seguridad y el rendimiento de la aplicación. Además, el uso de `PreparedStatement` también permite que el código sea

más legible y fácil de mantener a medida que aumenta la complejidad de las consultas y operaciones en la base de datos.

Por tanto, aunque técnicamente podemos usar Statement, es más recomendable y seguro utilizar PreparedStatement para las inserciones de datos parametrizados en una base de datos SQLite.

Ejemplo de insertar alumnos:

```
private static void insertarAlumnos(Connection conn) {
    // Datos de ejemplo de los alumnos
    String[] nombres = {"Estudiante1", "Estudiante2", "Estudiante3", "Estudiante4"};
    String[] ciclos = {"DAM", "DAW", "ASIR", "SMR"};

    // Sentencia SQL
    String sqlInsert = "INSERT INTO alumnos VALUES (?, ?, ?)";

    try (PreparedStatement ps = conn.prepareStatement(sqlInsert)) {
        // Insertar los 4 alumnos
        for (int i = 0; i < 4; i++) {
            ps.setInt(1, i+1);
            ps.setString(2, nombres[i]);
            ps.setString(3, ciclos[i]);
            System.out.println("Nº de filas insertadas: " + ps.executeUpdate());
        }
    } catch (SQLException ex) {
        System.out.println("Error en insertar alumnos: " + ex.getMessage());
    }
} // Fin insertarAlumnos
```

Ejecutamos y comprobamos en DB Browser que se han insertado correctamente los alumnos:

```
run-single:
Conexión con BD establecida
Nº de filas insertadas: 1
Nº de filas insertadas: 1
Nº de filas insertadas: 1
Nº de filas insertadas: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

DB Browser for SQLite - /home

Archivo Editar Ver Herramientas Ayuda

Nueva base de datos Abrir base de datos Guardar cambios

Estructura Hoja de datos Editar pragmas Ejecutar SQL

Tabla **alumnos**

id ▼	nombre	ciclo
Filtro	Filtro	Filtro
1	1 Estudiante1	DAM
2	2 Estudiante2	DAW
3	3 Estudiante3	ASIR
4	4 Estudiante4	SMR

Consulta de los datos de una tabla (SELECT)

- Vamos a crear ahora el método `consultarTodosLosAlumnos`, que muestra por consola el contenido de la tabla alumnos, para no tener que comprobar los cambios mediante línea de comandos o aplicación externa (DB Browser). Utilizaremos para ello la clase Statement (en este caso no vamos a tener ningún parámetro), su método `executeQuery` y recorreremos el objeto resultante de la sentencia SELECT (ResultSet) mostrando los resultados por consola.

Ejemplo del método `consultarTodosLosAlumnos`:

```
private static void consultarTodosLosAlumnos(Connection conexion) {  
  
    System.out.println("\nContenido de la tabla alumnos...");  
    String sqlSelect = "SELECT * FROM alumnos";  
  
    try (Statement stmt = conexion.createStatement();  
         ResultSet rs = stmt.executeQuery(sqlSelect)) {  
  
        System.out.printf("%-4s %-15s %-5s\n", "ID", "NOMBRE", "CICLO");  
        System.out.println("-----");  
  
        // Recorremos el ResultSet resultante de ejecutar la sentencia SELECT  
        while (rs.next()) {  
            // Mostramos el resultado por consola  
  
            // Con printf  
            System.out.printf("%-4s %-15s %-5s\n",  
                              rs.getInt("id"),  
                              rs.getString("nombre"),  
                              rs.getString("ciclo"));  
        }  
    } catch (SQLException e) {  
        System.out.println("Error en consultar alumnos: " + e.getMessage());  
    }  
} //Fin consultarTodosLosAlumnos
```



IMPORTANTE: Si en la sentencia SELECT tuviéramos algún parámetro, por ejemplo, mostrar sólo los alumnos de un ciclo determinado utilizando una cláusula WHERE, sería recomendable utilizar PreparedStatement en lugar de Statement.

Eliminar registros de una tabla (DELETE)

- Creamos el método eliminarAlumnoPorId que elimina un alumno con un id que se pide por teclado.

Ejemplo de eliminar alumno:

```
private static void eliminarAlumnoPorId(Connection conexion) {

    String sqlDelete = "DELETE FROM alumnos WHERE id = ?"; //El ; no es obligatorio en este caso
    try (PreparedStatement ps = conexion.prepareStatement(sqlDelete)) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Introduzca el ID del alumno a eliminar: ");

        ps.setInt(1, sc.nextInt());
        int filasEliminadas = ps.executeUpdate();
        System.out.println( (filasEliminadas == 0) ? "No existe ese ID en la base de datos"
            : "Filas eliminadas " + filasEliminadas);
    } //fin try
    catch (SQLException ex) {
        System.out.println("Error en eliminar alumno: " + ex.getMessage());
    } // fin catch
} // Fin eliminarAlumnoPorId
```

Actualizar registros de una tabla (UPDATE)

- Creamos el método modificarAlumno, para eliminar un alumno con un id que se pide por teclado.

Ejemplo de modificar alumno:

```
private static void modificarAlumno(Connection conexion) {

    String sqlUpdate = "UPDATE alumnos SET nombre = ?, ciclo = ? WHERE id = ?";
    try (PreparedStatement ps = conexion.prepareStatement(sqlUpdate)) {

        //Scanner sc = new Scanner(System.in);
        Scanner sc = new Scanner(System.in).useDelimiter("\n");

        //Pedimos el id del alumno que queremos modificar
        System.out.print("Id alumno a modificar: ");
        int id = sc.nextInt();

        System.out.print("Introduzca el nuevo nombre del alumno: ");
        //String nuevoNombre = sc.nextLine();
        String nuevoNombre = sc.next();
        System.out.print("Introduzca el nuevo ciclo: ");
        //String nuevoCiclo = sc.nextLine();
        String nuevoCiclo = sc.next();

        ps.setString(1, nuevoNombre);
        ps.setString(2, nuevoCiclo);
        ps.setInt(3, id);

        int filasActualizadas = ps.executeUpdate();
        System.out.println( (filasActualizadas == 0) ? "No existe ese ID en la base de datos"
            : "Filas actualizadas " + filasActualizadas);
    } // Fin try
    catch (SQLException ex) {
        System.out.println("Error en modificar alumno: " + ex.getMessage());
    } // Fin catch
} // Fin modificarAlumno
```


Ejemplo de método main

```
public class DemoSqlite {
    public static void main(String[] args) {

        // 1. Creamos cadena de conexión
        String url = "jdbc:sqlite:./src/db/ejemplo.db";
        // 2. Abrimos conexión con la BD
        //try-with-resources, no tenemos que cerrar manualmente la conexión
        try (Connection conexion = DriverManager.getConnection(url)) {
            // Si no existe la bdd, se crea
            // 3 Class.forName("org.sqlite.JDBC"); // <-- Desde Java 6 no hace falta
            // Class.forName("com.mysql.cj.jdbc.Driver"); // <-- Para MySQL
            System.out.println("Conexión con BD establecida");

            //4 Creamos método para crear una tabla en la base de datos ejemplo.db
            crearTabla(conexion);
            //5 Método para añadir algunos alumnos
            insertarAlumnos(conexion);
            //6 Creamos un método para consultar
            consultarTodosLosAlumnos(conexion);
            //7 Eliminamos un alumno solicitando su id
            eliminarAlumnoPorId(conexion);
            consultarTodosLosAlumnos(conexion);
            //8 Modificamos el nombre y el ciclo de un alumno
            modificarAlumno(conexion);
            consultarTodosLosAlumnos(conexion);

            //} catch (ClassNotFoundException ex) { //sólo si se ejecuta Class.forName
            //    System.out.println("ClassNotFoundException generada: " + ex.getMessage());
            } catch (SQLException ex) {
                System.out.println("SQLException generada: " + ex.getMessage());
            }
        }
    }
} // Fin método main
```