

Unidad 15. Programación de base de datos en Java.

Contenido

1. Introducción.	2
1.1 El desfase objeto-relacional	3
1.2 JDBC.	4
1.3 Conectores o Drivers.	4
1.1.1 Cómo funciona JDBC.....	5
1.4 Instalación de la base de datos. Tarea guiada	8
1.1.2 SQLite.....	8
2. Creación de las tablas en una base de datos.	13
2.1 Lenguaje SQL (I)	14
2.2 Lenguaje SQL (II). Tarea guiada creación de base de datos	15
3. Establecimiento de conexiones. Tarea guiada conexión jdbc	15
3.1 Instalar el conector de la base de datos.	17
3.2 Registrar el controlador JDBC.	18
4. Ejecución de consultas sobre la base de datos.	20
4.1 Recuperación de información (I).	21
1.1.3 Recuperación de información (II).	22
4.2 Actualización de información.....	27
4.3 Adición de información.	28
4.4 Borrado de información.	32
4.5 Sentencias preparadas.	33
4.6 Cierre de conexiones.....	36
4.7 Excepciones.	36
5. Referencias.	48

Caso Práctico

Ada ha asignado un proyecto a María y a Juan. Se trata de un proyecto importante, y puede suponer muchas ventas, y por tanto una gran expansión para la empresa.

En concreto, un notario de renombre en el panorama nacional, se dirigió a BK programación para pedirles que les desarrolle un programa para su notaría, de modo que toda la gestión de la misma, incluyendo la emisión de las escrituras, se informatizaran. Además, si el programa es satisfactorio, se encargará de promocionar la aplicación ante el resto de sus conocidos notarios, pudiendo por tanto suponer muchas ventas y por ello, dinero.

Una cuestión vital en la aplicación es el almacenamiento de los datos. Los datos de los clientes, y de las escrituras deberán guardarse en bases de datos, para su tratamiento y recuperación las veces que haga falta.

Como en BK programación trabajan sobre todo con Java, desde el primer momento Juan y María tienen claro que van a tener que utilizar bases de datos relacionales y JDBC y así lo comentan con Ada.

1. Introducción.

Hoy en día, la mayoría de aplicaciones informáticas necesitan almacenar y gestionar gran cantidad de datos. Esos datos, se suelen guardar en bases de datos relacionales, ya que éstas son las más extendidas actualmente.

Las bases de datos relacionales permiten organizar los datos en tablas y esas tablas y datos se relacionan mediante campos clave. Además, se trabaja con el lenguaje estándar conocido como SQL, para poder realizar las consultas que deseemos a la base de datos.

Una base de datos relacional se puede definir de una manera simple como aquella que presenta la información en tablas con filas y columnas.

Una tabla es una serie de filas y columnas, en la que cada fila es un registro y cada columna es un campo. Un campo representa un dato de los elementos almacenados en la tabla (NSS, nombre, etc.) Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

No se permite que pueda aparecer dos o más veces el mismo registro, por lo que uno o más campos de la tabla forman lo que se conoce como clave primaria.

El sistema gestor de bases de datos, en inglés conocido como: **Database Management System (DBMS)**, gestiona el modo en que los datos se almacenan, mantienen y recuperan. En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: **Relational Database Management System (RDBMS)**.

Tradicionalmente, la programación de bases de datos ha sido como una Torre de Babel: gran cantidad de productos de bases de datos en el mercado, y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC (Java Database Connectivity)**, permite simplificar el acceso a base de datos, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java,
- Aprovechar la experiencia de los APIs de bases de datos existentes, □ Ser sencillo.

1.1 *El desfase objeto-relacional*

El **desfase objeto-relacional**, también conocido como **impedancia objeto-relacional** (o desajuste de la impedancia), consiste en la **diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos**. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación.** El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos:** en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.
- **Paradigma de programación.** En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo EntidadRelación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

El **modelo relacional trata con relaciones y conjuntos** debido a su naturaleza matemática. Sin embargo, el **modelo de Programación Orientada a Objetos trata con objetos y las asociaciones** entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio

La escritura (y de manera similar la lectura) **mediante JDBC** implica: abrir una conexión, crear una sentencia en SQL y copiar **todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto**. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. **Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.**

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en con u lenguajes de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador

tiene un atributo "teléfono". Al transformar éste caso a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

Debes conocer

Si no has estudiado nunca bases de datos, ni tienes idea de qué es SQL o el modelo relacional, sería conveniente que te familiarizaras con él. A continuación te indicamos un tutorial bastante ameno sobre SQL y en donde describe brevemente el modelo relacional.

<https://www.aulalic.es/sql/>

1.2 JDBC.

JDBC es un API Java que hace posible ejecutar sentencias SQL.

De JDBC podemos decir que:

- Consta de un conjunto de clases e interfaces escritas en Java.
- Proporciona un API estándar para desarrollar aplicaciones de bases de datos con un API Java pura.

Con JDBC, no hay que escribir un programa para acceder a una base de datos Access, otro programa distinto para acceder a una base de datos Oracle, etc., sino que podemos escribir un único programa con el API JDBC y el programa se encargará de enviar las sentencias SQL a la base de datos apropiada. Además, y como ya sabemos, una aplicación en Java puede ejecutarse en plataformas distintas.

En el desarrollo de JDBC, y debido a la confusión que hubo por la proliferación de API's propietarios de acceso a datos, Sun buscó los aspectos de éxito de un API de este tipo, ODBC (Open Database Connectivity).

ODBC se desarrolló con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.

Aunque la industria ha aceptado ODBC como medio principal para acceso a bases de datos en Windows, ODBC no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

El nivel de abstracción al que trabaja JDBC es alto en comparación con ODBC, la intención de Sun fue que supusiera la base de partida para crear librerías de más alto nivel.

JDBC intenta ser tan simple como sea posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

1.3 Conectores o Drivers.

El API JDBC viene distribuido en dos paquetes:

- `java.sql`, dentro de J2SE
- `javax.sql`, extensión dentro de J2EE

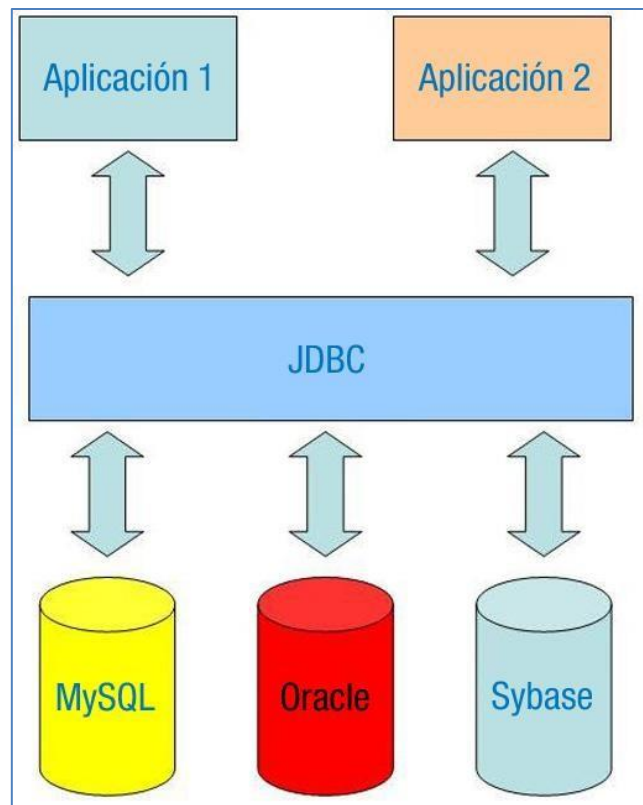
Un **conector** o **driver** es un conjunto de clases encargadas de implementar las interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un conector adecuado. Un conector suele ser un fichero **.jar** que contiene una implementación de todas las interfaces del API JDBC.

Cuando se construye una aplicación de base de datos, **JDBC** oculta los detalles específicos de cada base de datos, de modo que el programador se ocupe sólo de su aplicación.

El conector lo proporciona el fabricante de la base de datos o bien un tercero.

El código de nuestra aplicación no depende del driver, puesto que trabajamos contra los paquetes **java.sql** y **javax.sql**.



Cada BD necesita su conector.

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.

1.1.1 Cómo funciona JDBC

JDBC define varias interfaces que permite realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes. Estas están definidas en el paquete **java.sql**. La siguiente tabla muestra las clases e interfaces más importantes:

CLASE E INTERFACE	DESCRIPCIÓN
-------------------	-------------

Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
DriverManager	Permite gestionar todos los drivers instalados en el sistema.
DriverPropertyInfo	Proporciona diversa información acerca de un driver.
Connection	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión.
DatabaseMetadata	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros.
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada.
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
ResultSet	Contiene las filas resultantes de ejecutar una orden SELECT.
ResultSetMetadata	Permite obtener información sobre un ResultSet , como el número de columnas, sus nombres, etc.
http://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html	

La siguiente figura muestra las 4 clases principales que usa cualquier programa Java con JDBC:



Figura. Funcionamiento de JDBC.

El trabajo con JDBC comienza con la clase **DriverManager** que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

1. Importar las clases necesarias.
2. Cargar el driver JDBC.
3. Identificar el origen de datos.
4. Crear un objeto **Connection**.
5. Crear un objeto **Statement**.

6. Ejecutar una consulta con el objeto **Statement**.
7. Recuperar los datos del objeto **ResultSet**.
8. Liberar el objeto **ResultSet**.
9. Liberar el objeto **Statement**.
10. Liberar el objeto **Connection**.

El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a una base de datos **MySQL** de nombre **ejemplo**, que utiliza como nombre de usuario y clave **ejemplo**; y muestra el contenido de la tabla *departamentos*:

```
import java.sql.*; public class Main {
public static void main(String[] args) {
try {

    //Cargar el driver
    Class.forName("com.mysql.jdbc.Driver");

    //Establecemos la conexion con la BD
    Connection conexion = DriverManager.getConnection
        ("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");

    // Preparamos la consulta
    Statement sentencia = conexion.createStatement();
String sql = "SELECT * FROM departamentos";

    // Ejecutamos la consulta
    ResultSet resul = sentencia.executeQuery(sql);
    //Recorremos el resultset para visualizar cada fila
    //Se hace un bucle mientras haya filas y se van mostrando
while (resul.next()) {
    System.out.printf("%d, %s, %s %n",
        resul.getInt(1), //NUMERO DEPARTAMENTO
        resul.getString(2), //DNOMBRE
        resul.getString(3));
    //LOCALIDAD
    }
    resul.close(); // Cerrar ResultSet
    sentencia.close(); // Cerrar Statement
    conexion.close(); // Cerrar conexión

    } catch (ClassNotFoundException cn) {
        cn.printStackTrace();
    } catch (SQLException e) {
e.printStackTrace();
    }

    } // fin de main

} // fin de la clase
```

En principio, todos los conectores deben ser compatibles con **ANSI SQL-2 Entry Level** (ANSI SQL-2 se refiere a los estándares adoptados por el *American National Standards Institute* (ANSI) en 1992. *Entry Level* se refiere a una lista específica de capacidades de SQL). Los desarrolladores de conectores pueden establecer que sus conectores conocen estos estándares.

1.4 Instalación de la base de datos. **Tarea guiada**

Lo primero que tenemos que hacer, para poder realizar consultas en una base de datos, es obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas.

Cuando desarrollamos pequeñas aplicaciones en las que no vamos a almacenar grandes cantidades de información no es necesario que utilicemos un sistema gestor de base de datos como Oracle o MySQL. En su lugar podemos utilizar una **base de datos embebida** donde el motor esté incrustado en la aplicación y sea exclusivo para ella. La base de datos se inicia cuando se ejecuta la aplicación, y termina cuando se cierra la aplicación.

Por lo general, este tipo de bases de datos vienen del movimiento *Open Source*, aunque también hay algunas de origen propietario.

En este capítulo trabajaremos con **SQLite**

1.1.2 SQLite.

SQLite es un **sistema gestor de base de datos multiplataforma escrito en C** que proporciona un **motor muy ligero**. Las bases de datos se guardan en forma de ficheros por lo que es fácil trasladar la base de datos con la aplicación que la usa. Cuenta con una utilidad que nos permitirá ejecutar comandos SQL en modo consola. Es un proyecto de dominio público. La biblioteca implementa la mayor parte del estándar **SQL-92**, incluyendo transacciones de base de datos atómicas, consistencia de base de datos, aislamiento y durabilidad, triggers (o disparadores) y la mayor parte de las consultas complejas. Los programas que utilizan la funcionalidad de **SQLite** lo hacen a través de llamadas simples a subrutinas y funciones. **SQLite** se puede utilizar desde programas en C/C++, PHP, Visual Basic, Perl, Delphi, Java, etc.

Su instalación es sencilla. Desde la página <http://www.sqlite.org/download.html> se puede descargar. Para sistemas Windows podemos descargar el fichero ZIP **sqlite-tools-win32-x863300100.zip**.

Precompiled Binaries for Windows

[sqlite-dll-win32-x86-3300100.zip](#) 32-bit DLL (x86) for SQLite version 3.30.1.
(sha1: d63326562a5b9b46da202b6af2f14cc646a50d81)
(478.32 KiB)

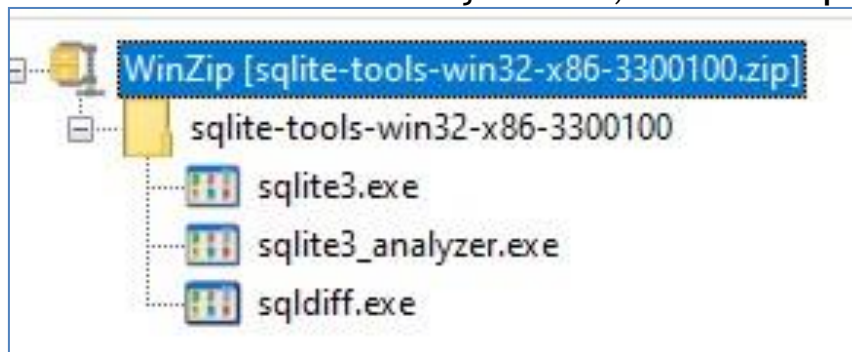
[sqlite-dll-win64-x64-3300100.zip](#) 64-bit DLL (x64) for SQLite version 3.30.1.
(sha1: 725eb6588492b5728993815a6e5f0f8f08ddfc4)
(788.50 KiB)

[sqlite-tools-win32-x86-3300100.zip](#) A bundle of command-line tools for managing SQLite database files, including the [command-line shell](#), the [sqlite3_analyzer.exe](#) program, and the [sqlite3_diff.exe](#) program.
(1.72 MiB) (sha1: 11d5d92b21a50678d73aba61389b6a568c39de2a)

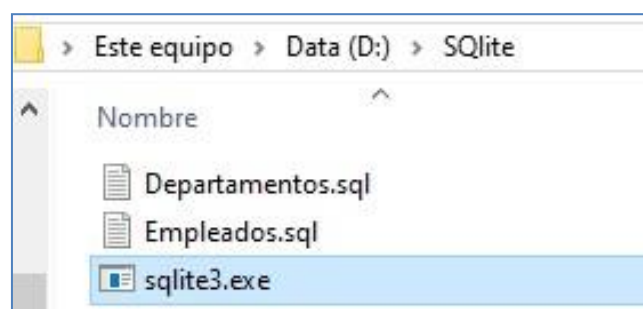
Universal Windows Platform

[sqlite-uwp-](#) VSIX package for Universal Windows Platform development using Visual Studio 2015.

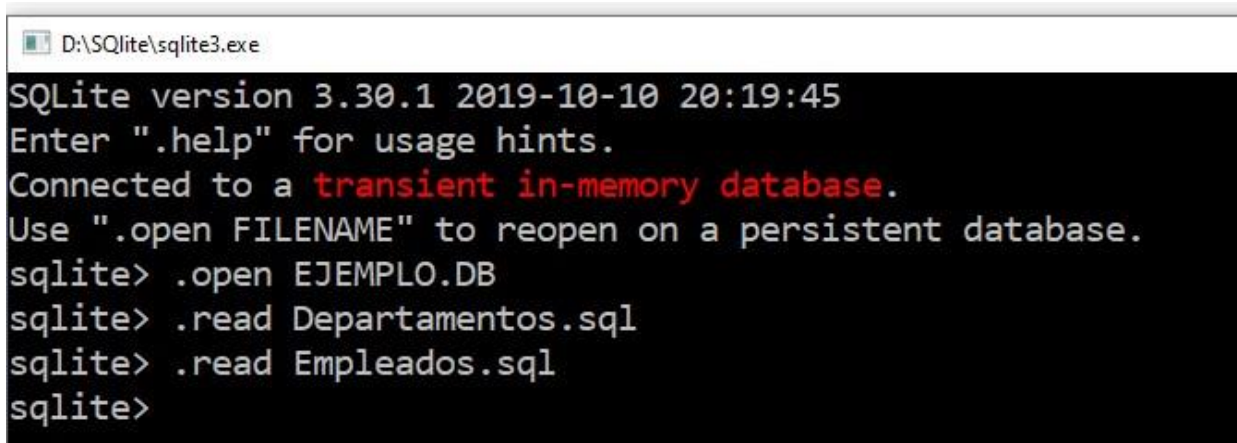
Al descomprimirlo obtenemos varios ficheros ejecutables, entre ellos **sqlite3.exe**.



Para probar esta base de datos podemos crear la carpeta **SQLite** en la unidad D (si no tenemos lo haremos en la unidad C), y almacenamos en ella el fichero **sqlite3.exe**, y los ficheros **.sql** que contienen las órdenes SQL para crear y almacenar datos en las tablas de departamentos y empleados (*Departamentos.sql* y *Empleados.sql*), estos ficheros los puedes encontrar en los recursos del capítulo:



A continuación vamos a crear una base de datos de nombre **EJEMPLO.DB** y crearemos las tablas empleados y departamentos. Para ello hacemos doble click sobre el fichero **sqlite3.exe** y creamos la base de base de datos escribiendo a la derecha del prompt de SQLite, **sqlite>**, la orden: **.open EJEMPLO.DB**. A continuación usando el comando **.read** cargamos los ficheros sql de departamentos y empleados que contienen las órdenes para crear las tablas de esta base de datos:

A screenshot of a SQLite command-line interface window. The title bar shows the file path 'D:\SQLite\sqlite3.exe'. The window has a black background with white text. The text inside the window shows the SQLite version (3.30.1), the date and time (2019-10-10 20:19:45), and instructions on how to use the interface. It also shows the user entering commands to open a database, read SQL files, and read more SQL files.

```
D:\SQLite\sqlite3.exe
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open EJEMPLO.DB
sqlite> .read Departamentos.sql
sqlite> .read Empleados.sql
sqlite>
```

También podemos copiar los datos del fichero sql y pegarlo a la derecha del prompt de SQLite. Una vez que tenemos las tablas creadas ya podemos hacer consultas sobre ellas. El siguiente ejemplo crea la tabla de departamentos a partir del fichero sql y la tabla empleados la crearía copiando las órdenes del fichero sql y pegándolas a la derecha del prompt de SQLite, después se consultan las tablas de departamentos y de empleados:

```

sqlite> .open EJEMPLO.DB
sqlite> .read Departamentos.sql
sqlite> CREATE TABLE empleados
(
    ...> emp_no      INT NOT NULL PRIMARY KEY,
    ...> apellido    VARCHAR(10),
    ...> oficio      VARCHAR(10),
    ...> dir         INT,
    ...> fecha_alt   DATE,
    ...> salario     FLOAT,
    ...> comision    FLOAT,
    ...> dept_no     INT NOT NULL REFERENCES departamentos(dept_no)
    ...> );
sqlite> INSERT INTO empleados VALUES (7369,'SANCHEZ','EMPLEADO',7902,'1990-12-17',
    ...> 1040,NULL,20);
sqlite> INSERT INTO empleados VALUES (7499,'ARROYO','VENDEDOR',7698,'1990-02-20',
    ...> 1500,390,30);
sqlite> INSERT INTO empleados VALUES (7521,'SALA','VENDEDOR',7698,'1991-02-22',
    ...> 1625,650,30);
sqlite> INSERT INTO empleados VALUES (7566,'JIMENEZ','DIRECTOR',7839,'1991-04-02',
    ...> 2900,NULL,20);
sqlite> INSERT INTO empleados VALUES (7654,'MARTIN','VENDEDOR',7698,'1991-09-29',
    ...> 1600,1020,30);
sqlite> INSERT INTO empleados VALUES (7698,'NEGRO','DIRECTOR',7839,'1991-05-01',
    ...> 3005,NULL,30);
sqlite> INSERT INTO empleados VALUES (7782,'CEREZO','DIRECTOR',7839,'1991-06-09',
    ...> 2885,NULL,10);
sqlite> INSERT INTO empleados VALUES (7788,'GIL','ANALISTA',7566,'1991-11-09',
    ...> 3000,NULL,20);
sqlite> INSERT INTO empleados VALUES (7839,'REY','PRESIDENTE',NULL,'1991-11-17',
    ...> 4100,NULL,10);
sqlite> INSERT INTO empleados VALUES (7844,'TOVAR','VENDEDOR',7698,'1991-09-08',
    ...> 1350,0,30);
sqlite> INSERT INTO empleados VALUES (7876,'ALONSO','EMPLEADO',7788,'1991-09-23',
    ...> 1430,NULL,20);
sqlite> INSERT INTO empleados VALUES (7900,'JIMENO','EMPLEADO',7698,'1991-12-03',
    ...> 1335,NULL,30);
sqlite> INSERT INTO empleados VALUES (7902,'FERNANDEZ','ANALISTA',7566,'1991-12-03',
    ...> 3000,NULL,20);
sqlite> INSERT INTO empleados VALUES (7934,'MUÑOZ','EMPLEADO',7782,'1992-01-23',
    ...> 1690,NULL,10); sqlite>
select * from departamentos;
10|CONTABILIDAD|SEVILLA
20|INVESTIGACION|MADRID
30|VENTAS|BARCELONA
40|PRODUCCION|BILBAO sqlite>
select * from empleados;
7369|SANCHEZ|EMPLEADO|7902|1990-12-17|1040.0||20
7499|ARROYO|VENDEDOR|7698|1990-02-20|1500.0|390.0|30
7521|SALA|VENDEDOR|7698|1991-02-22|1625.0|650.0|30
7566|JIMENEZ|DIRECTOR|7839|1991-04-02|2900.0||20
7654|MARTIN|VENDEDOR|7698|1991-09-29|1600.0|1020.0|30
7698|NEGRO|DIRECTOR|7839|1991-05-01|3005.0||30
7782|CEREZO|DIRECTOR|7839|1991-06-09|2885.0||10
7788|GIL|ANALISTA|7566|1991-11-09|3000.0||20
7839|REY|PRESIDENTE||1991-11-17|4100.0||10
7844|TOVAR|VENDEDOR|7698|1991-09-08|1350.0|0.0|30
7876|ALONSO|EMPLEADO|7788|1991-09-23|1430.0||20
7900|JIMENO|EMPLEADO|7698|1991-12-03|1335.0||30
7902|FERNANDEZ|ANALISTA|7566|1991-12-03|3000.0||20
7934|MUÑOZ|EMPLEADO|7782|1992-01-23|1690.0||10 sqlite>

```

Problemas con acentos y otros caracteres como la ñ. Cuando hacemos la inserción de datos en las tablas **a partir de los ficheros sql, si en estos hay acentos, eñes, y otro tipo**

de caracteres; las órdenes se ejecutarán pero estos caracteres no se almacenarán correctamente.

Para finalizar la sesión se escribe el comando `.quit`.

El comando `.tables` muestra las tablas creadas.

Para activar en SQLite el **uso de claves ajenas** debemos ejecutar esta orden:

```
sqlite> PRAGMA foreign_keys = ON;
```

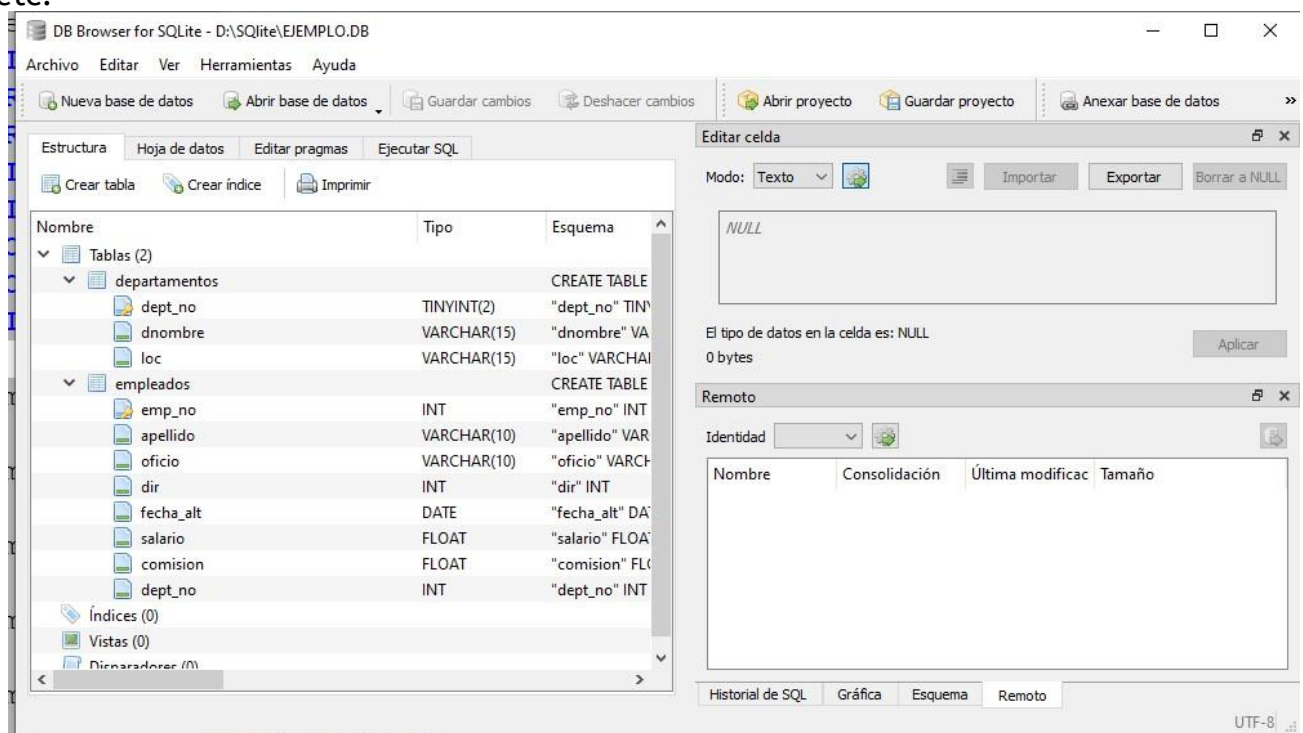
Para instalar SQLite en Linux escribimos desde la línea de comandos:

```
$sudo apt-get install sqlite3
```

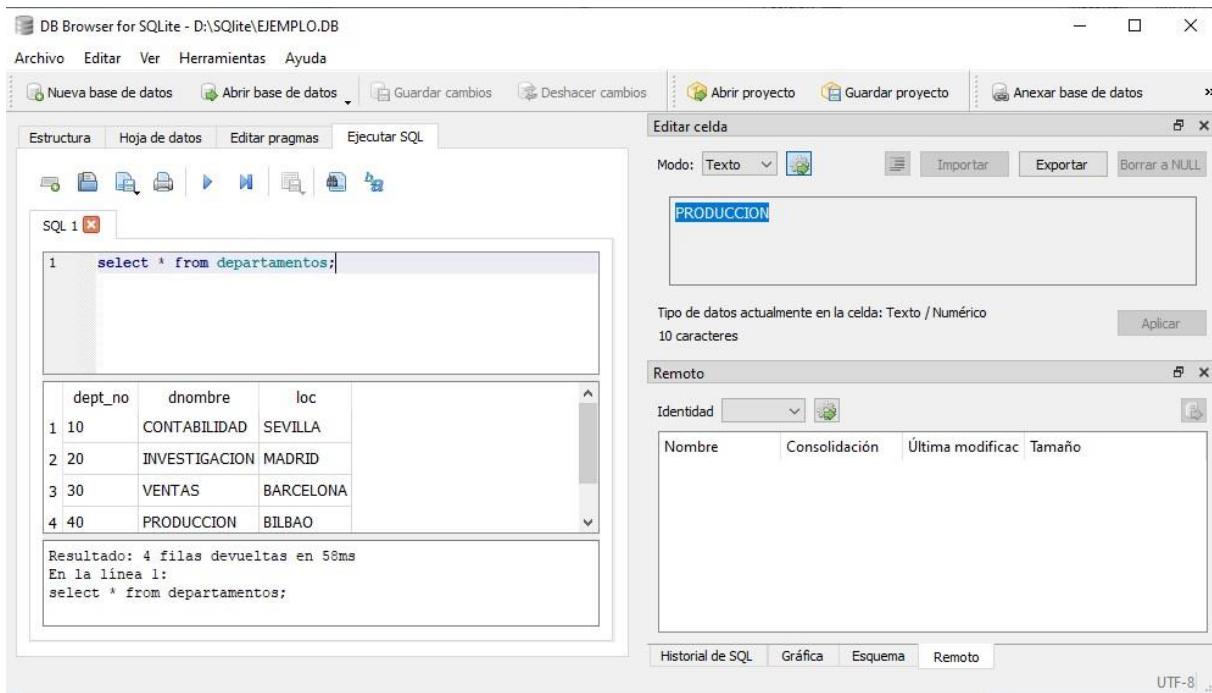
Y para ejecutar SQLite escribimos lo siguiente para crear la base de datos en la carpeta `/home/usuario/DB/SQLITE` (que tiene que existir):

```
$ sqlite3 /home/usuario/DB/SQLITE/ejemplo.db
```

Podemos **descargarnos un visor y editor ligero para ficheros de bases de datos SQLite llamado DB Browser for SQLite** desde la URL <https://sqlitebrowser.org/>. Desde este editor se puede abrir la BD, hacer consultas, crear nuevas tablas, una nueva base de datos, etc:



Si descargamos la versión ZIP tendríamos que ejecutar el fichero **DB Browser for SQLite.exe** que se encuentra en la carpeta **DB Browser for SQLite**.



2. Creación de las tablas en una base de datos.

Caso práctico

María, Ada y Juan han realizado concienzudamente el diseño de las tablas necesarias para la base de datos de la aplicación de notaría.

También se han decantado por el sistema gestor de bases de datos a utilizar. Emplearán un sistema gestor de bases de datos relacional. Una vez instalado el sistema gestor, tendrán que programar los accesos a la base de datos para guardar los datos, recuperarlos, realizar las consultas para los informes y documentos que sean necesarios, etc.

En **Java podemos conectarnos y manipular bases de datos utilizando JDBC**. Pero la creación en sí de la base de datos debemos hacerla con la herramienta específica para ello.

Normalmente será el administrador de la base de datos, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos. **No todos los drivers JDBC soportan la creación de la base de datos mediante el lenguaje** de definición de datos (DDL).

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos con sus tablas, claves, y todo lo necesario.

También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.

Vamos utilizar el sistema gestor de base de datos SQLite, por ser un producto gratuito y de fácil uso, además de muy potente. Ya hemos visto como acceder desde la consola de SQLite o usando una herramienta gráfica como **DB Browser for SQLite**.

2.1 Lenguaje SQL (I)

¿Cómo le pedimos al **Sistema Gestor de Bases de Datos Relacional** (SGBDR), en concreto en este caso, al de SQLite, que nos proporcione la información que nos interesa de la base de datos?

Se utiliza el lenguaje **SQL (Structured Query Language)** para interactuar con el SGBDR. SQL es un lenguaje no procedimental en el cual se le indica al SGBDR qué queremos obtener y no cómo hacerlo. El SGBDR analiza nuestra orden y si es correcta sintácticamente la ejecuta.

El estudio de SQL nos llevaría mucho más que una unidad, y es objeto de estudio en otros módulos de este ciclo formativo. Pero como resulta imprescindible para poder continuar, haremos una mínima introducción sobre él.

Los comandos SQL se pueden dividir en dos grandes grupos:

- Los que se utilizan para definir las estructuras de datos, llamados comandos **DDL (Data Definition Language). Lenguaje de Definición de Datos.**
- Los que se utilizan para operar con los datos almacenados en las estructuras, llamados **DML (Data Manipulation Language). Lenguaje de Manipulación de Datos.**

En la siguiente tabla se muestran algunos de los comandos SQL más utilizados:

<i>Sentencia DDL</i>	<i>Objetivo</i>
Alter Table	Añadir o redefinir una columna, modificar la asignación de almacenamiento.
Create Table	Crear una tabla.
Create Index	Crear un índice.
Drop Table	Eliminar una tabla.
Drop Index	Eliminar un índice.
Grant	Conceder privilegios o roles, a un usuario o a otro rol.
Revoke	Retirar los privilegios de un usuario o rol de la base de datos.
<i>Sentencia DML</i>	<i>Objetivo</i>
Insert	Añadir filas de datos a una tabla.
Delete	Eliminar filas de datos de una tabla.
Update	Modificar los datos de las filas de una tabla.
Select	Recuperar datos de las filas de una o varias tablas.
Commit	Confirmar como permanentes las modificaciones realizadas.

Rollback	Deshacer todas las modificaciones realizadas desde la última confirmación.
----------	--

2.2 Lenguaje SQL (II). **Tarea guiada creación de base de datos**

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL, puesto que antes de poder almacenar y recuperar información debemos definir las estructuras donde agrupar la información. Las estructuras básicas con las que trabaja SQL son las tablas.

Como hemos visto antes, una tabla es un conjunto de celdas agrupadas en filas y columnas donde se almacenan elementos de información.

Antes de llevar a cabo la creación de una tabla conviene planificar: nombre de la tabla, nombre de cada columna, tipo y tamaño de los datos almacenados en cada columna, información adicional, restricciones, etc.

Hay que tener en cuenta también ciertas restricciones en la formación de los nombres de las tablas: longitud. Normalmente, aunque dependen del sistema gestor, suele tener una longitud máxima de 30 caracteres, no puede haber nombres de tabla duplicados, deben comenzar con un carácter alfabético, permitir caracteres alfanuméricos y el guión bajo '_', y normalmente no se distingue entre mayúsculas y minúsculas.

Por ejemplo para crear una tabla de departamentos podríamos hacer:

```
CREATE TABLE departamentos (  
  dept_no TINYINT(2) NOT NULL PRIMARY KEY,  dnombre VARCHAR(15),  loc  
  VARCHAR(15)  
);
```

Donde se crea la tabla con clave primaria en la columna dept_no.

Y una tabla de empleados, teniendo en cuenta el departamento en el que trabajen:

```
CREATE TABLE empleados (  
  emp_no INT NOT NULL PRIMARY  
  KEY,  apellido VARCHAR(10),  
  oficio VARCHAR(10),  dir  
  INT,  fecha_alt DATE,  salario  
  FLOAT,  comision FLOAT,  dept_no  
  INT NOT NULL,  
  FOREIGN KEY(dept_no) REFERENCES departamentos(dept_no)  
);
```

La columna emp_no es la clave primaria, y la columna dept_no es clave ajena que referencia a la tabla de departamentos.

3. Establecimiento de conexiones. **Tarea guiada conexión jdbc**

Caso práctico

Tanto Juan como María saben que trabajar con bases de datos relacionales en Java es tremendamente sencillo, por lo que establecer una conexión desde un programa en Java, a una base de datos, es muy fácil.

Juan le comenta a María: -Empleando la tecnología sólo necesitamos dos simples sentencias Java para conectar la aplicación a la base de datos. María, prepárate que en un periquete tengo lista la conexión con la base de datos y salimos a tomar un café.

Cuando queremos **acceder a una base de datos para operar con ella**, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método **getConnection()** de la clase **DriverManager**. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión, también se puede indicar el nombre y la clave del usuario que accede a la base de datos. En el ejemplo inicial, **Main.java**, para conectar con MySQL escribimos lo siguiente:

```
Connection conexion = DriverManager.getConnection  
("jdbc:mysql://localhost/ejemplo", "ejemplo", "ejemplo");
```

El **primer parámetro** del método **getConnection()** representa la **URL de conexión a la base de datos**. Tiene el siguiente formato para conectarse a MySQL:

```
jdbc:mysql://nombre_host:puerto/nombre_basedatos
```

Donde

- **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
- **nombre_host** indica el nombre del servidor donde está la base de datos. Aquí puede ponerse una IP o un nombre de máquina que esté en la red. Si especificamos **localhost** como nombre de servidor, estamos indicando que el servidor de base de datos se encuentra en la misma máquina en la que se ejecuta el programa Java.
- **puerto** es el puerto predeterminado para las bases de datos MySQL, por defecto es **3306**. Si no se pone se asume este valor.
- **nombre_basedatos** es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL. En este caso el nombre es *ejemplo*.

El **segundo parámetro** es el nombre de usuario que accede a la base de datos, en este caso se llama *ejemplo*.

El **tercer parámetro** es la clave del usuario, que en este caso también es *ejemplo*.

La ejecución de este método devuelve un objeto **Connection** que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, **DriverManager** itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún driver adecuado, se lanza una **SQLException**.

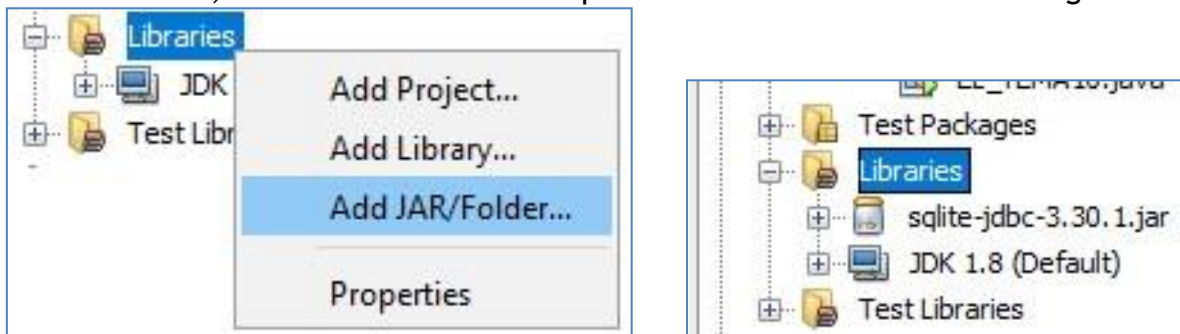
Para establecer la conexión con la base de datos de SQLite creada anteriormente después indicar que estamos usando el driver JDBC para para sqlite (`jdbc:sqlite`), indicamos el fichero de base de datos almacenado en disco que contiene nuestras tablas, no se indica nombre de usuario ni clave ya que no se necesitan en SQLite:

```
Connection conexion = DriverManager.getConnection  
    ("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
```

3.1 Instalar el conector de la base de datos.

Para conectarnos a SQLite necesitamos la librería **sqlite-jdbc-3.30.1.jar** que se puede descargar desde la URL <https://bitbucket.org/xerial/sqlite-jdbc/downloads>.

En el entorno gráfico que usemos incluimos el fichero JAR. Desde Netbeans y después de crear el proyecto Java pulsamos con el botón derecho del ratón sobre **Libraries** -> **Add JAR Folder** y buscamos el fichero jar descargado para incluirlo en el proyecto. Una vez añadido el fichero, el nodo **Libraries** debe quedar como se muestra en la imagen:



Partimos del programa Java inicial que recorre la tabla *departamentos* (su nombre es **Main.java**). Cambiamos dos cosas: la carga del driver, en este caso se llama **org.sqlite.JDBC** y la conexión a la base de datos, SQLite no se necesita nombre de usuario ni clave:

```
Class.forName("org.sqlite.JDBC");
```

```
Connection conexion = DriverManager.getConnection  
    ("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
```

El ejemplo **Main.java** completo quedaría así:

```
import java.sql.*;
```

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

    //Cargar el driver
    Class.forName("org.sqlite.JDBC");

    //Establecemos la conexion con la BD
    Connection conexion = DriverManager.getConnection
("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");

    // Preparamos la consulta
    Statement sentencia = conexion.createStatement();
String sql = "SELECT * FROM departamentos";

    // Ejecutamos la consulta
    ResultSet resul = sentencia.executeQuery(sql);
    //Recorremos el resultset para visualizar cada fila
    //Se hace un bucle mientras haya filas y se van mostrando
while (resul.next()) {
    System.out.printf("%d, %s, %s %n",
        resul.getInt(1),    //NUMERO DEPARTAMENTO
resul.getString(2),    //DNOMBRE    resul.getString(3));
//LOCALIDAD
    }
    resul.close();    // Cerrar ResultSet
    sentencia.close(); // Cerrar Statement
    conexion.close(); // Cerrar conexión

    } catch (ClassNotFoundException cn) {
        cn.printStackTrace();
    } catch (SQLException e) {
e.printStackTrace();
    }

    }// fin de main

} // fin de la clase
```

La ejecución mostraría la siguiente salida:

```
run:
10, CONTABILIDAD, SEVILLA
20, INVESTIGACION, MADRID
30, VENTAS, BARCELONA
40, PRODUCCION, BILBAO
BUILD SUCCESSFUL (total time: 3 seconds)
```

3.2 Registrar el controlador JDBC.

Al fin y al cabo ya lo hemos visto en el ejemplo de código que poníamos antes, pero incidimos de nuevo. Registrar el controlador que queremos utilizar es tan fácil como escribir una línea de código.

Hay que consultar la documentación del controlador que vamos a utilizar para conocer el nombre de la clase que hay que emplear. En el caso del controlador para SQLite es "org.sqlite.JDBC", o sea, que se trata de la clase JDBC que está en el paquete org.sqlite del conector que hemos descargado, y que has observado y que no es más que una librería empaquetada en un fichero .jar.

La línea de código necesaria en este caso, en la aplicación Java que estemos construyendo es:

```
//Cargar el driver      Class.forName("org.sqlite.JDBC");
```

Conexión a MySQL.




Para conectarnos a MySQL necesitamos la librería **mysql-connector-java-5.1.48.jar**, podemos descargarla desde la URL <http://dev.mysql.com/downloads/connector/j/>. Podemos descargar la versión ZIP, dentro del paquete se encuentra el JAR. El driver se llama **com.mysql.jdbc.Driver** y la conexión es la siguiente:

```
Class.forName("com.mysql.jdbc.Driver");
```

```
Connection conexion = DriverManager.getConnection
    ("jdbc:mysql://localhost/basedatos",
     "nombreusuario", "claveusuario");
```

Conexión a Oracle.

Para conectarnos mediante JDBC usamos el driver **JDBC Thin**. Se puede descargar desde la URL de Oracle <https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html>. Es necesario comprobar antes la versión de la base de datos instalada y la versión de nuestro entorno Java. Por ejemplo para funcionar con JDK8 y Oracle18 descargaríamos el driver **ojdbc8.jar**.

OR	
The Unzipped JDBC Driver and Companion JARs	
The JARs included in the ojdbc10-full.tar.gz and ojdbc8-full.tar.gz are also available as individual downloads in this section.	
Download	Release Notes
 ojdbc10.jar	Certified with JDK10; Oracle JDBC driver except classes for NLS support in Oracle Object and Collection types. (4,243,140 bytes - SHA1: bba59347e68c9416d14fcc9a9209e869f842e48d)
 ojdbc8.jar	Certified with JDK8; Oracle JDBC driver except classes for NLS support in Oracle Object and Collection types. (4,210,517 bytes - SHA1: 967c0b1a2d5b1435324de34a9b8018d294f8f47b)
 ucp.jar	the Universal Connection Pool (1,680,074 bytes - SHA1: 796b661b0bb1818b7c04171837356acddcea504c)

Necesitamos saber el nombre de servicio que usa la base de datos para incluirlo en la URL de la conexión. Normalmente para la versión *Express Edition* el nombre es XE. También se necesita el nombre de usuario y la clave. El driver se llama **oracle.jdbc.driver.OracleDriver**, y la conexión a la base de datos es la siguiente:

```
Class.forName ("oracle.jdbc.driver.OracleDriver");
```

```
Connection conexion = DriverManager.getConnection  
    ("jdbc:oracle:thin:@localhost:1521:XE",  
     "nombreusuario", "claveusuario");
```

Una vez cargado el controlador, es posible hacer una conexión al SGBD. Hay que asegurarse de que si no utilizáramos NetBeans u otro IDE, para añadir el .jar, entonces el archivo jar que contiene el controlador JDBC para el SGBD habría que incluirlo en el CLASSPATH que emplea nuestra máquina virtual, o bien en el directorio ext del JRE de nuestra instalación del JDK.

Hay una excepción en la que no hace falta ni hacer eso: en caso de utilizar un acceso mediante el puente JDBC-ODBC, ya que ese driver está incorporado dentro de la distribución de Java, por lo que no es necesario incorporarlo explícitamente en el CLASSPATH de una aplicación Java. Por ejemplo, sería el caso de acceder a una base de datos Microsoft Access.

4. Ejecución de consultas sobre la base de datos. Tarea guiada consultas BD

Caso práctico

Ada está echando una mano a Juan y María en la creación de consultas, para los informes que la aplicación de notaría debe aportar a los usuarios de la misma.

Hacer consultas es una de las facetas de la programación que más entretiene a Ada, le resulta muy ameno y fácil. Además, y dada la importancia del proyecto, cuanto antes avancen en él, mucho mejor.

Por suerte, los tres: Ada, María y Juan tienen experiencia en consultas SQL y saben que, cuando se hace una consulta a una base de datos, hay que afinar y hacerla lo más eficiente posible, pues si se descuidan el sistema gestor puede tardar mucho en devolver los resultados. Además, algunas consultas pueden devolver un conjunto de registros bastante grande, que puede resultar difícil de manejar desde el programa, ya que por norma general tendremos que manejar esos datos registro a registro.

Para operar con una base de datos ejecutando las consultas necesarias, nuestra aplicación deberá hacer las operaciones siguientes:

- Cargar el conector necesario para comprender el protocolo que usa la base de datos en cuestión.
- Establecer una conexión con la base de datos.
- Enviar consultas SQL y procesar el resultado.
- Liberar los recursos al terminar.
- Gestionar los errores que se puedan producir.

Podemos utilizar los siguientes tipos de sentencias:

- **Statement**: para sentencias sencillas en SQL.

- **PreparedStatement:** para consultas preparadas, como por ejemplo las que tienen parámetros.
- **CallableStatement:** para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- **Consultas:** SELECT. Para las sentencias de consulta que obtienen datos de la base de datos, se emplea el método **ResultSet executeQuery(String sql)**. El método de ejecución del comando SQL devuelve un objeto de tipo **ResultSet** que sirve para contener el resultado del comando SELECT, y que nos permitirá su procesamiento.
- **Actualizaciones:** INSERT, UPDATE, DELETE, sentencias DDL. Para estas sentencias se utiliza el método **executeUpdate(String sql)**

4.1 Recuperación de información (I).

Las consultas a la base de datos se realizan con sentencias SQL que van "embebidas" en otras sentencias especiales que son propias de Java. Por tanto, podemos decir que las consultas SQL las escribimos como parámetros de algunos métodos Java que reciben el String con el texto de la consulta SQL.

Las consultas devuelven un **ResultSet**, que es una clase java parecida a una lista en la que se aloja el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos que cumple con los requisitos de la consulta.

El **ResultSet no contiene todos los datos**, sino que los va **obteniendo de la base de datos según se van pidiendo**. La razón de esto es evitar que una consulta que devuelva una cantidad muy elevada de registros, tarde mucho tiempo en obtenerse y sature la memoria del programa.

Con el **ResultSet** hay disponibles una serie de métodos que permiten movernos hacia delante y hacia atrás en las filas, y obtener la información de cada fila.

Por ejemplo, para obtener: el número de empleado, apellido oficio y salario de los empleados que están almacenados en la tabla del mismo nombre, de la base de datos EJEMPLO que se creó anteriormente, haríamos la siguiente consulta:

```
// Preparamos la consulta
Statement sentencia = conexion.createStatement();
String sql = "SELECT EMP_NO, APELLIDO, OFICIO, SALARIO "
            + "FROM EMPLEADOS";

// Ejecutamos la consulta
ResultSet resul = sentencia.executeQuery(sql);
```

El método **next()** del **ResultSet** hace que dicho puntero avance al siguiente registro. Si lo consigue, el método **next()** devuelve true. Si no lo consigue, porque no haya más registros que leer, entonces devuelve false.

1.1.3 Recuperación de información (II).

El método `executeQuery()` devuelve un objeto `ResultSet` para poder recorrer el resultado de la consulta utilizando un cursor.

Para obtener una columna del registro utilizamos los métodos `get`. Hay un método `get...` para cada tipo básico Java y para las cadenas.

Un método interesante es `wasNull()` que nos informa si el último valor leído con un método `get` es nulo.

Cuando trabajamos con el `ResultSet`, en cada registro, los métodos `getInt()`, `getString()`, `getDate()`, etc., nos devuelve los valores de los campos de dicho registro.

Podemos pasar a estos métodos un índice (que comienza en 1) para indicar qué columna de la tabla de base de datos deseamos, o bien, podemos usar un String con el nombre de la columna (tal cual está en la tabla de base de datos).

El ejemplo anterior completo mostrando en consola los datos de los empleados de la base de datos SQLite EJEMPLO se muestra a continuación:

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

public class ListaEmpleados {

    public static void main(String[] args) {

        try {
            //Cargar el driver
            Class.forName("org.sqlite.JDBC");
            //Establecemos la conexion con la BD
            Connection conexion =
                DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
            // Preparamos la consulta
            Statement sentencia = conexion.createStatement();
            String sql = "SELECT EMP_NO, APELLIDO, OFICIO, SALARIO
"                + "FROM EMPLEADOS";
            // Ejecutamos la consulta
            ResultSet resul = sentencia.executeQuery(sql);
            //Recorremos el Resultset para visualizar cada fila
            //Se hace un bucle mientras haya filas, next() devuelve true,
            //y se van mostrando
            while (resul.next()) {
                System.out.printf("%d, %s, %s, %.2f%n",
resul.getInt(1),    //EMP_NO
resul.getString(2), //APELLIDO
resul.getString(3), //OFICIO
resul.getFloat(4)); //SALARIO            }
```

```

        resul.close();        // Cerrar
ResultSet      sentencia.close(); //
Cerrar Statement      conexion.close();
// Cerrar conexión

    } catch (ClassNotFoundException cn) {        cn.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}

    }

} //main
} //

```

También podemos visualizar las columnas de la consulta usando el nombre indicado en la SELECT (mayúscula o minúscula):

```

    System.out.printf("%d, %s, %s, %.2f\n",
resul.getInt("EMP_NO"),        //EMP_NO
resul.getString("APELLIDO"),   //APELLIDO
resul.getString("OFICIO"),     //OFICIO
resul.getFloat("salario") );  //SALARIO

```

Algunos de los métodos **getXXX()** para la obtención de valores son los siguientes:

Método	Tipo Java devuelto:
getString(int númerodecolumna) getString(String nombredocolumna)	String
getBoolean(int númerodecolumna) getBoolean(String nombredocolumna)	boolean
getByte(int númerodecolumna) getByte(String nombredocolumna)	byte
getShort(int númerodecolumna) getShort(String columna)	short
getInt(int númerodecolumna) getInt(String nombredocolumna)	int
getLong(int númerodecolumna) getLong(String nombredocolumna)	long
getFloat(int númerodecolumna) getFloat(String nombredocolumna)	float
getDouble(int númerodecolumna) getDouble(String nombredocolumna)	double
getBytes(int númerodecolumna) getBytes(String nombredocolumna)	byte[]
getDate(int númerodecolumna) getDate(String nombredocolumna)	Date
getTime(int númerodecolumna) getTime(String nombredocolumna)	Time
getTimestamp(int númerodecolumna) getTimestamp(String nombredocolumna)	Timestamp
Más información sobre métodos de ResultSet: https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html	

ACTIVIDADES RESUELTAS 1.

Ejemplo1: A partir de la BD SQLITE creada anteriormente, realizar un programa Java para obtener el APELLIDO, OFICIO y SALARIO de los empleados de un departamento leído por teclado. Si el departamento no tiene empleados se mostrará un mensaje indicándolo. El departamento se leerá en un proceso repetitivo que finalizará cuando sea <=0.


```

package Ejemplos; import
java.sql.*;

import java.util.Scanner;

public class Ejemplo1 {

    static Connection conexion;
    static Scanner sc;

    public static void main(String[] args) {
sc = new Scanner(System.in);
        try {
            Class.forName("org.sqlite.JDBC");
            conexion = DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");

            listadoEmpleados();

            conexion.close();
        } catch (ClassNotFoundException cn)
        {
            cn.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    static void listadoEmpleados()
    {
        int dep = 0;          int
        cuenta = 0;

        while (true) {
            System.out.printf("\nNº de departamento: ");
            dep = sc.nextInt();          if (dep <= 0) {
                break;
            }
            sc.nextLine();
            cuenta = 0;
            try {
                Statement sentencia = conexion.createStatement();
                System.out.println("=====");
                System.out.println("APELLIDO          OFICIO          SALARIO");
                System.out.println("=====");
                String consulta = "SELECT apellido, oficio, salario
from"                + " empleados where dept_no = " + dep;
                ResultSet resul = sentencia.executeQuery(consulta);

                while (resul.next()) {
                    System.out.printf("%-15s %-15s %.2f %n",
                        resul.getString(1), resul.getString(2), resul.getFloat(3));
                    cuenta++;
                }
            }
            if (cuenta == 0) {
                System.out.println("\t\tNO HAY EMPLEADOS...");
            }
        }
    }
}

```

```

    }
    System.out.println("=====");
    resul.close();// Cerrar ResultSet
sentencia.close();// Cerrar Statement

    } catch (SQLException e) {
        e.printStackTrace();
    }

    } // while
} // listado

} // fin

```

Ejemplo2: Realiza otro programa Java que visualice el APELLIDO del empleado con máximo salario, visualiza también su SALARIO y el NOMBRE DE SU DEPARTAMENTO.

```

package Ejemplos;

import java.sql.*;

public class Ejemplo2 {

    static Connection conexion;

    public static void main(String[] args) {
try {
        Class.forName("org.sqlite.JDBC");
        conexion = DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");

        EmpleadomasSalario();

        conexion.close();
    } catch (ClassNotFoundException cn)
    {
        cn.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
} // main

    private static void EmpleadomasSalario() {

        try {
            Statement sentencia = conexion.createStatement();
ResultSet resul;
            String sql = "SELECT apellido, salario, dnombre "
                + " FROM empleados e, departamentos d "
                + " WHERE salario= (select max(salario) from empleados)"
+ " AND d.dept_no=e.dept_no ";

            // Preparamos la consulta
            sentencia = conexion.createStatement();
resul = sentencia.executeQuery(sql);
            System.out.println("=====");
            System.out.print("EMPLEADO CON MAS SALARIO: ");

```

```
        if (resul.next()) {  
            System.out.printf("%s, Dep: %s, Salario: %.2f %n",  
                resul.getString("apellido"), resul.getString("dnombre"),  
                resul.getFloat("salario"));  
        }  
        System.out.println("=====");  
        resul.close();// Cerrar ResultSet  
        sentencia.close();// Cerrar Statement  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
} //
```

//fin

FIN ACTIVIDADES RESUELTAS 1.

4.2 Actualización de información. Tarea guiadas CRUD

Respecto a las consultas de actualización, **executeUpdate()**, retornan el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Al crearse un objeto **Statement** se crea un espacio de trabajo para crear consultas SQL, ejecutarlas y para recibir los resultados de las consultas. Una vez creado el objeto se pueden usar los siguientes métodos:

- **ResultSet executeQuery(String)**: lo hemos visto anteriormente, se utiliza para sentencias SQL que recuperan datos de un único objeto **ResultSet**, se utiliza para las sentencias **SELECT**.
- **int executeUpdate(String)**: se utiliza para sentencias que no devuelven un **ResultSet** como son las sentencias de manipulación de datos (DML): **INSERT**, **UPDATE** y **DELETE**; y las sentencias de definición de datos (DDL): **CREATE**, **DROP** y **ALTER**. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de las sentencias DDL devuelve el valor 0.
- **boolean execute(String)**: se puede utilizar para ejecutar cualquier sentencia SQL. Tanto para las que devuelven un **ResultSet** (por ejemplo **SELECT**), como para las que devuelven el número de filas afectadas (por ejemplo **INSERT**, **UPDATE**, **DELETE**) y para las de definición de datos como por ejemplo **CREATE**. El método devuelve **true** si devuelve un **ResultSet** (para recuperar las filas será necesario llamar al método **getResultSet()**) y **false** si se trata de un recuento de actualizaciones o no hay resultados; en este caso se usará el método **getUpdateCount()** para recuperar el valor devuelto. En este ejemplo **execute()** ejecuta una sentencia **SELECT**, devuelve **true**; por tanto es necesario recuperar las filas devueltas usando el método **getResultSet()**:

```
import java.sql.*;

public class EjemploExecute {

    public static void main(String[] args) throws
        ClassNotFoundException, SQLException {

        Class.forName("org.sqlite.JDBC");
        Connection conexion = DriverManager.getConnection
            ("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
        String sql = "SELECT * FROM departamentos";
        Statement sentencia = conexion.createStatement();

        boolean valor = sentencia.execute(sql);

        if (valor) {

            ResultSet rs = sentencia.getResultSet();
            while (rs.next()) {
                System.out.printf("%d, %s, %s %n",
                    rs.getInt(1), rs.getString(2), rs.getString(3));
            }
            rs.close();

        } else {

            int f = sentencia.getUpdateCount();
            System.out.printf("Filas afectadas:%d %n", f);
        }
        sentencia.close();
        conexion.close();
    } //main
} //
```

Si cambiamos la orden SQL por esta otra: *String sql= "UPDATE departamentos SET dnombre = LOWER(dnombre)";* entonces la variable *valor* será *false* y la salida del programa será diferente.

4.3 Adición de información.

Si queremos añadir un registro a una tabla de la base de datos con la que estamos trabajando tendremos que utilizar la sentencia **INSERT INTO** de SQL. Utilizaremos **executeUpdate()** pasándole como parámetro la consulta de inserción en este caso.

El siguiente ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se asignarán en el programa. Antes de ejecutar la orden **INSERT** construimos la sentencia SQL en un *String*, las cadenas de caracteres (en este caso el nombre del departamento y la localidad) deben ir encerradas entre comillas simples:

```
import java.sql.*;

public class InsertarDep {
```

```

public static void main(String[] args) {
    try {
        Class.forName("org.sqlite.JDBC");
        Connection conexion =
            DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");

        int dep = 11;           // num. departamento
        String dnombre = "Dep once"; // nombre
        String loc = "Guadalajara"; // localidad

        // construir orden INSERT
        String sql = String.format("INSERT INTO departamentos VALUES "
+ "(%d, '%s', '%s')", dep, dnombre, loc);
        //Tambien se puede hacer así
        //String sql = "INSERT INTO departamentos VALUES
        //( " + dep + " , '" + dnombre + "', '" + loc + "' ) ";
        Statement sentencia = conexion.createStatement();
        int filas = sentencia.executeUpdate(sql);
        System.out.printf("Filas afectadas: %d %n", filas);
        sentencia.close(); // Cerrar
        conexion.close(); // Cerrar
    } catch (ClassNotFoundException cn) {

        cn.printStackTrace();

    } catch (SQLException e) {

        System.out.println("ERROR: " + e.getMessage());
        System.out.println("cod ERROR: " + e.getErrorCode());
    }

    } // fin de main
} // fin de la clase

```

Si se ejecuta más de una vez el ejercicio insertando los mismos valores en los campos de departamento se producirá un error, ya que la clave primaria existe en la tabla, por ello se ha incluido la sección de excepciones para que muestre el mensaje de error y el código de error. Si se produce error se mostrará un mensaje similar a:

```

ERROR: [SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY constraint failed
(UNIQUE constraint failed: departamentos.dept_no) cod
ERROR: 19

```

El siguiente ejemplo inserta un empleado en la tabla de empleados. Recuerda que para poder dar soporte a las claves ajenas necesitamos ejecutar la orden: **PRAGMA foreign_keys = ON**. Por tanto antes de realizar la inserción del empleado ejecutamos dicha orden con el método **execute()**:

```

import java.sql.*;

public class InsertarEmple {

    public static void main(String[] args) {

```

```
try {  
    Class.forName("org.sqlite.JDBC");  
    Connection conexion =
```

```

        DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
        Statement sentencia1 =
conexion.createStatement();                String activarfk = "
PRAGMA foreign_keys = ON ";
sentencia1.execute(activarfk);
sentencia1.close();

        // construir orden INSERT
        String sql = "INSERT INTO empleados "
            + "(emp_no, apellido, oficio,salario, dept_no) "
            + " VALUES (1006, 'nuevo2', 'EMPLEADO', 2000, 10)";
        Statement sentencia2 =
conexion.createStatement();                int filas = 0;
try {
            filas = sentencia2.executeUpdate(sql.toString());
System.out.println("Filas afectadas: " + filas);
        } catch (SQLException e) {
            System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:%n");
            System.out.printf("Mensaje   : %s %n", e.getMessage());
            System.out.printf("SQL estado: %s %n", e.getSQLState());
            System.out.printf("Cod error : %s %n", e.getErrorCode());
        }

        sentencia2.close(); // Cerrar
Statement          conexion.close(); // Cerrar
conexion

        } catch (ClassNotFoundException cn) {
cn.printStackTrace();        } catch
(SQLException e) {
        e.printStackTrace();
    }

    } // fin de main
} // fin de la clase

```

Los errores que se pueden dar es que el empleado ya exista, El control de excepciones muestra los siguientes mensajes:

```

HA OCURRIDO UNA EXCEPCIÓN:
Mensaje   : [SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY
constraint failed (UNIQUE constraint failed: empleados.emp_no)
SQL estado: null
Cod error : 19

```

O que el departamento no exista:

```

HA OCURRIDO UNA EXCEPCIÓN:
Mensaje   : [SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint failed
(FOREIGN KEY constraint failed)
SQL estado: null Cod
error : 19

```

El siguiente ejemplo **sube el salario a los empleados de un departamento**. El número de departamento y la subida los indicamos directamente en dos variables:

```
import java.sql.*;

public class ModificarSalario {

    public static void main(String[] args) {
        int dep = 10;
        int subida = 100;

        try {
            Class.forName("org.sqlite.JDBC");
            Connection conexion =
                DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
            String sql = "UPDATE empleados SET salario = salario + " +
subida + " WHERE dept_no = " + dep;
            Statement sentencia = conexion.createStatement();
            int filas = sentencia.executeUpdate(sql);
            System.out.printf("Empleados modificados: %d %n", filas);
            sentencia.close(); // Cerrar
Statement          conexion.close(); // Cerrar
conexión

        } catch (ClassNotFoundException cn)
        {
            cn.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    } // fin de main
} // fin de la clase
```

4.4 Borrado de información.

Cuando nos interese eliminar registros de una tabla de una base de datos, emplearemos la sentencia SQL: DELETE. Así, por ejemplo, si queremos eliminar el empleado de la tabla empleados cuyo número de empleado es 1002 utilizo el código siguiente:

```
int emp_no = 1002;
String sql = "DELETE FROM EMPLEADOS WHERE emp_no=" + emp_no;
Statement sentencia = conexion.createStatement();

int filas = sentencia.executeUpdate(sql);
System.out.printf("Empleados ELIMINADOS: %d %n", filas);
```


Si intento eliminar un departamento y este tiene empleados, ocurrirá un error siempre y cuando se haya definido la activación del uso de claves ajenas.

4.5 Sentencias preparadas.

En los ejemplos anteriores hemos creado sentencias SQL a partir de cadenas de caracteres en las que íbamos concatenando los datos necesarios para construir la sentencia completa. La interfaz **PreparedStatement** nos va a permitir construir una cadena de caracteres SQL con *placeholder* o marcadores de posición, que representarán los datos que serán asignados más tarde, el *placeholder* se representa mediante el símbolo interrogación (?). Por ejemplo la orden INSERT para insertar un departamento se representa así:

```
String sql= "INSERT INTO departamentos VALUES (?, ?, ?)";  
                                                    //1 2 3 valor del  
índice
```

La orden INSERT para insertar un empleado y asignar los valores a cada campo se representa así:

```
String sql = "INSERT INTO empleados VALUES(?, ?, ?, ?, ?, ?, ?, ? )"  
;
```

```
PreparedStatement sentencia = conexion.prepareStatement(sql);
```

```
sentencia.setInt(1, emp_no));           // num empleado  
sentencia.setString(2, apellido);       // apellido  
sentencia.setString(3, oficio);         // oficio  
sentencia.setInt(4, dir);               // director  
sentencia.setDate(5, fecha);            // fecha de alta  
sentencia.setDouble(6, salario);        // salario  
sentencia.setDouble(7, comision);       // comisión  
sentencia.setInt(8, dept_no);           // departamento
```

```
int filas = sentencia.executeUpdate(); // filas afectadas
```

Cada *placeholder* tiene un índice, el 1 correspondería al primero que se encuentre en la cadena, el 2 al segundo y así sucesivamente. Solo se pueden utilizar para ocupar el sitio de los datos en la cadena SQL, no se pueden usar para representar una columna o un nombre de una tabla, por ejemplo *FROM ?* sería incorrecto.

Antes de ejecutar un **PreparedStatement** es necesario asignar los datos para que cuando se ejecute la base de datos asigne variables de unión con estos datos y ejecute la orden SQL. Los objetos **PreparedStatement** se pueden preparar o precompilar una sola vez y ejecutar las veces que queramos asignando diferentes valores a los marcadores de posición, en cambio en los objetos **Statement**, la sentencia SQL se suministra en el momento de ejecutar

la sentencia.

Los métodos de **PreparedStatement** tienen los mismos nombres que en **Statement**: **executeQuery()**, **executeUpdate()** y **execute()** pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método **prepareStatement(String)**. El ejemplo anterior en el que se inserta una fila en la tabla *departamentos* quedaría así:

```
//construir orden INSERT
String sql= "INSERT INTO departamentos VALUES(?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(1, Integer.parseInt(dep)); // num departamento
sentencia.setString(2, dnombre);           // nombre
sentencia.setString(3, loc);               // localidad
int filas = sentencia.executeUpdate(); // filas
afectadas
```

Usando **String.format()** la orden quedaría (en el ejemplo se asume que los datos del departamento se proporcionan en forma de String, incluido el nº de departamento):

```
String sql =
    String.format("INSERT INTO departamentos VALUES (%s,
'%s','%s')",
                dep, dnombre, loc);
```

Para asignar valor a cada uno de los marcadores de posición se utilizan los métodos **setXXX()**. La sintaxis es la siguiente:

```
public abstract void setXXX
    (int indexedelParametro, tipoJava valor) throws
SQLException
```

Donde, se asigna el valor indicado en *tipoJava* al parámetro cuyo índice coincide con *indexedelParametro*, que es transformado por el controlador JDBC en un tipo SQL correspondiente para pasarlo a la base de datos. Los métodos **setXXX()** son los siguientes:

Método	Tipo SQL
void setString(int índice, String valor)	VARCHAR
void setBoolean(int índice, boolean valor)	BIT
void setByte(int índice, byte valor)	TINYINT
void setShort(int índice, short valor)	SMALLINT
void setInt(int índice, int valor)	INTEGER
void setLong(int índice, long valor)	BIGINT
void setFloat(int índice, float valor)	FLOAT
void setDouble(int índice, double valor)	DOUBLE
void setBytes(int índice, byte[] valor)	VARBINARY
void setDate(int índice, Date valor)	DATE
void setTime(int índice, Time valor)	TIME

Más información sobre métodos de PreparedStatement:
<https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html>

El siguiente ejemplo muestra el uso de sentencias preparadas para obtener el apellido y salario de los empleados del departamento 10 cuyo oficio sea DIRECTOR:

```
import java.sql.*;

public class VerEmpleado {

    public static void main(String[] args) {
    try {
        Class.forName("org.sqlite.JDBC");
        Connection conexion =
            DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
        String dep = "10";           //departamento
        String oficio = "DIRECTOR"; //oficio

        //construimos la orden SELECT
        String sql = "SELECT apellido, salario FROM empleados
                     WHERE dept_no = ? AND oficio = ? ORDER BY 1";
        // Preparamos la sentencia
        PreparedStatement sentencia = conexion.prepareStatement(sql);

        sentencia.setInt(1, Integer.parseInt(dep));
        sentencia.setString(2, oficio);

        ResultSet rs = sentencia.executeQuery();

        while (rs.next()) {
            System.out.printf("%s => %.2f %n",
                rs.getString("apellido"), rs.getFloat("salario"));
        }

        rs.close(); // liberar recursos
        sentencia.close();
        conexion.close();

    } catch (Exception e) {
        e.printStackTrace();
    }

    } //fin de main
} //fin de la clase
```

4.6 Cierre de conexiones.

Las conexiones a una base de datos consumen muchos recursos en el sistema gestor por ende en el sistema informático en general. Por ello, conviene cerrarlas con el **método close()** siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (**Statement** y **PreparedStatement**) y los resultados (**ResultSet**) para liberar los recursos. Ejemplo:

```
        resul.close();        // Cerrar ResultSet
sentencia.close(); // Cerrar Statement
conexion.close(); // Cerrar conexión
```

4.7 Excepciones.

En todas las aplicaciones en general, y por tanto en las que acceden a bases de datos en particular, nos puede ocurrir con frecuencia que la aplicación no funciona, no muestra los datos de la base de datos que deseábamos, etc.

Es importante capturar las excepciones que puedan ocurrir para que el programa no aborte de manera abrupta. Además, es conveniente tratarlas para que nos den información sobre si el problema es que se está intentando acceder a una base de datos que no existe, o que el servicio MySQL no está arrancado, o que se ha intentado hacer alguna operación no permitida sobre la base de datos, como acceder con un usuario y contraseña no registrados, ...

Por tanto es conveniente emplear el método **getMessage()** de la clase **SQLException** para recoger y mostrar el mensaje de error que ha generado MySQL, lo que seguramente nos proporcionará una información más ajustada sobre lo que está fallando.

Cuando se produce un error se lanza una excepción del tipo **java.sql.SQLException**. Es importante que las operaciones de acceso a base de datos estén dentro de un bloque **try-catch** que gestione las excepciones.

Los objetos del tipo **SQLException** tienen dos métodos muy útiles para obtener el código del error producido y el mensaje descriptivo del mismo, **getErrorCode()** y **getMessage()** respectivamente.

El método **getMessage()** imprime el mensaje de error asociado a la excepción que se ha producido, que aunque esté en inglés, nos ayuda a saber qué ha generado el error que causó la excepción.

El método **getErrorCode()**, devuelve un número entero que representa el código de error asociado. Habrá que consultar en la documentación para averiguar su significado.

Hasta ahora en casi todos los ejemplos cuando se producía un error se visualizaba con el método **printStackTrace()** la secuencia de llamadas al método que ha producido la excepción y la línea de código donde se produce el error. Veamos un ejemplo de cómo se utilizan esos métodos para visualizar los mensajes de error:

```
try
```

```
{
```

```
//Código que puede producir error
}
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e)
{
    System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:%n");
    System.out.printf("Mensaje    : %s %n", e.getMessage());
    System.out.printf("SQL estado: %s %n", e.getSQLState());
    System.out.printf("Cód error  : %s %n",
e.getErrorCode());
}
```

El siguiente ejemplo muestra la salida que se produce cuando se intenta hacer **SELECT** de una tabla que no existe (en MySQL):

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje    : Table 'ejemplo.departamento' doesn't exist
SQL estado: 42S02
Cód error  : 1146
```

En Oracle se visualizaría información diferente:

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje    : ORA-00942: la tabla o vista no existe

SQL estado: 42000  Cód error : 942
```

Cuando se intenta insertar una fila en una tabla cuya clave primaria ya existe en **MYSQL** se muestra la siguiente información:

```
Mensaje    : Duplicate entry '10' for key 'PRIMARY'
SQL estado: 23000
Cód error  : 1062
```

Y en Oracle:

```
Mensaje    : ORA-00001: restricción única (EJEMPLO.PK_DEP)
violada
SQL estado: 23000
Cód error  : 1
```

Cuando se inserta una clave ajena en una tabla y no existe su correspondiente clave primaria en la otra tabla, en **MYSQL** se muestra la siguiente información:

```
Mensaje    : Cannot add or update a child row: a foreign key
constraint fails (`ejemplo`.`empleados`, CONSTRAINT
`FK_DEP`
FOREIGN KEY (`dept_no`) REFERENCES `departamentos`
(`dept_no`))
SQL estado: 23000
```

```
Cód error : 1452
```

En ORACLE:

```
Mensaje      : ORA-02291: restricción de integridad  
(EJEMPLO.FK_EMP)  
violada - clave principal no encontrada  
SQL estado: 23000  
Cód error : 2291
```

ACTIVIDADES RESUELTAS 2

Ejemplo3: Realiza un programa Java que introduzca por teclado un número de departamento y lo elimine de la tabla de departamentos. El departamento no se eliminará si tiene empleados, realizaremos un método para comprobarlo. Controla las excepciones que se puedan producir y muestra mensajes de si se elimina o no el departamento. Para la entrada se realizará un proceso repetitivo hasta que el número de departamento sea 0.

```
package Ejemplos;  
  
import java.sql.*;  
import  
java.util.InputMismatchException;  
import java.util.Scanner;  
  
public class Ejemplo3 {
```

```
static Scanner sc = new Scanner(System.in);
static Connection conexion;

    public static void main(String[] args) {

        try {

            Class.forName("org.sqlite.JDBC");
            conexion = DriverManager.getConnection
("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");

            int dep = 1;
            PreparedStatement sentencia = null;

            while (dep >= 0) {

                dep = LecturaDepartamento();

                if (dep == 0) {
                    break;
                }

                // comprobar si el departamento tiene empleados
                if (NumeroEmpleados(dep) > 0) {
                    System.out.println("\tEl
DEPARTAMENTO TIENE
EMPLEADOS.");
                    System.out.println("\tNO SE
ELIMINARÁ.");
                    continue;
                }

                // construir orden DELETE
                String sql = "DELETE FROM
departamentos where dept_no = ?";

                // Preparamos la
sentencia

                sentencia =
conexion.prepareStatement(sql);

                sentencia.setInt(1, dep);

                try {
                    int filas = sentencia.executeUpdate();
                    if (filas == 0) {

                        System.out.println("\tEl departamento no
                        existe.");
                    } else {

                        System.out.println("\tDepartamento eliminado.");
                    }

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

```
        } // while
        sentencia.close();
        conexion.close(); // Cerrar conexión

    } catch (ClassNotFoundException cn) {
cn.printStackTrace();
        } catch (SQLException e) {
MensajeException(e);
        }

    } // main
```

//MÉTODO QUE RECIBE LA EXCEPCIÓN Y MUESTRA

INFORMACIÓN SOBRE ELLA

```
private static void MensajeException(SQLException e) {
System.out.printf("HA OCURRIDO UNA EXCEPCION:%n");
```



```

        System.out.printf("Mensaje   : %s %n", e.getMessage());
        System.out.printf("SQL estado: %s %n", e.getSQLState());
        System.out.printf("Cod error  : %s %n", e.getErrorCode());
    }

    //MÉTODO QUE LEE UN NUMERO DE DEPARTAMENTO POR TECLADO
    static int lecturaDepartamento() {
        int dep = 0;
        int sw = 0;
        do {
            try {
                System.out.print("\nIntroduce el DEPARTAMENTO
                                (0 para finalizar): ");
                dep = sc.nextInt();
                sc.nextLine();
                sw = 1;

            } catch (InputMismatchException i) {
                System.out.println("\tNo valido");
                sc.nextLine();
            }

        } while (sw == 0);

        return dep;
    }

    // MÉTODO QUE DEVUELVE EL NÚMERO DE EMPLEADOS DE UN DEPARTAMENTO
    private static int NumeroEmpleados(int dep) throws SQLException {
        int cuenta = 0;
        String sql = "SELECT count(*) cuenta FROM empleados WHERE dept_no = ? ";
        PreparedStatement sentencia = conexion.prepareStatement(sql);
        sentencia.setInt(1, dep);
        ResultSet rs = sentencia.executeQuery();

        if (rs.next()) {
            cuenta = rs.getInt("cuenta");
        }
        return cuenta;
    }

}

}

```

Ejemplo4: Crea un programa Java que inserte un empleado en la tabla empleados. Se debe leer por teclado los datos a insertar en un proceso repetitivo hasta que el número de empleado sea 0. Cuando sea 0 no se pedirán más datos.

Se leerá por teclado: EMP_NO, APELLIDO, OFICIO, DIR, SALARIO, COMISIÓN, DEPT_NO.

Se deben validar los datos de entrada:

- DEPT_NO entre 1 Y 99.
- EMP_NO y DIR entre 1 Y 9999.

- COMISION entre 0 Y 999.
- SALARIO entre 1 Y 99999.
- Longitud de APELLIDO Y OFICIO > 0 (máximo 10 CARACTERES).

Antes de insertar se deben realizar las siguientes comprobaciones:

- Que el departamento exista en la tabla departamentos, si no existe no se inserta.
- Que el número del empleado no exista, si existe no se inserta.
- Que el salario sea > que 0, si es <= 0 no se inserta. (controlado en entrada)
- Que el director (DIR, es el número de empleado de su director) exista en la tabla empleados, si no existe no se inserta.
- El APELLIDO y el OFICIO no pueden ser nulos. □ La fecha de alta del empleado es la fecha actual.

Cuando se inserte la fila visualizar mensaje y si no se inserta visualizar el motivo (departamento inexistente, número de empleado duplicado, director inexistente, etc.).

```
package Ejemplos;

import java.sql.*; import
java.util.Scanner;

public class Ejemplo4 {

    static Scanner sc = new Scanner(System.in);
    static Connection conexion;

    public static void main(String[] args) throws ClassNotFoundException, SQLException
    {

        String emp_noS; // numero empleado 1 - 9999
        String apellido; // apellido
        String oficio; // oficio          int
        dir; // dir 1 - 9999
        Float salario; // salario 1 - 99999
        int comision; // comision 0-999      int
        dept_no; // 1--99

        Class.forName("org.sqlite.JDBC");
        conexion = DriverManager.getConnection("jdbc:sqlite:D:/SQLITE/EJEMPLO.DB");
        System.out.println("-----");
        System.out.println("Introduce datos de empleado (0 para salir): ");
        int emp_no = LeerEntero("\tIntroduce nº empleado: ", 0, 9999);

        while (emp_no > 0) {
            apellido = LeerCadena("\tIntroduce Apellido: ", 10);
            oficio = LeerCadena("\tIntroduce Oficio: ", 10);
            LeerEntero("\tIntroduce director: ", 1, 9999);
            LeerFloat("\tIntroduce salario: ", 1, 99999);
            LeerEntero("\tIntroduce comision: ", 0, 999);
            LeerEntero("\tIntroduce departamento: ", 1, 99);
            IntroducirDatos(emp_no, apellido, oficio, dir, salario, comision, dept_no);
            System.out.println("-----");
            System.out.println("Introduce datos de empleado: ");
            LeerEntero("\tIntroduce nº empleado: ", 0, 9999);
            System.out.println("-----");
            System.out.println("Fin de proceso....");
            conexion.close();
        }
    }
}
```

```

    }// MAIN

    //INSERTA LOS DATOS EN LA TABLA SI TODO VA BIEN
    private static void IntroducirDatos(int emp_no, String apellido, String oficio,
    int dir, Float salario,
        int comision, int dept_no) throws SQLException {

        boolean insertar = true;

        //COMPRUEBA SI EL DIRECTOR EXISTE
        if (!existeEmpleado(dir)) {
            System.out.println("\t<<DIRECTOR NO EXISTE, NO SE INSERTAR EL EMPLEADO>>");
            insertar = false;
        }

        //COMPRUEBA SI EL DEPARTAMENTO EXISTE
        if (!existeDepartamento(dept_no)) {
            System.out.println("\t<<EL DEP NOEXISTE, NO SE INSERTAR EL EMPLEADO>>");
            insertar = false;
        }

        //OBTIENE LA FECHA DE ALTA, QUE ES LA FECHA ACTUAL
        java.util.Date hoy = new java.util.Date();
        java.sql.Date fecha = new java.sql.Date(hoy.getTime());

        if (insertar) {
            String sql = "INSERT INTO empleados VALUES(?, ?, ?, ?, ?, ?, ?, ?)";
            int filas;
            try {
                PreparedStatement sentencia = conexion.prepareStatement(sql);
                sentencia.setInt(1, emp_no); // num emleado
                sentencia.setString(2, apellido); // apellido
                sentencia.setString(3, oficio); // oficio
                sentencia.setInt(4, dir); // director
                sentencia.setDate(5, fecha); // fecha
                sentencia.setFloat(6, salario); // salario
                sentencia.setFloat(7, comision); // comision
                sentencia.setInt(8, dept_no); // departamento
                filas = sentencia.executeUpdate(); // filas
                System.out.println("\tFilas insertadas: " +
                filas);
            } catch (SQLException e) {
                if (e.getErrorCode() == 19) {
                    System.out.println("\t<<EL NUMERO DE EMPLEADO YA EXISTE>>");
                } else {
                    MensajeException(e);
                }
                insertar = false;
            }
        }
    }// IntroducirDatos

    //DEVUELVE LOS MENSAJES DE LA EXCEPCION QUE RECIBE
    private static void MensajeException(SQLException e) {
        System.out.printf("HA OCURRIDO UNA EXCEPCION:%n");
        System.out.printf("Mensaje : %s %n", e.getMessage());
        System.out.printf("SQL estado: %s %n", e.getSQLState());
        System.out.printf("Cod error : %s %n", e.getErrorCode());
    }

```

```
}
```

```
//LEE UNA CADENA DE TECLADO HASTA UNA LONGITUD MAXIMA
```

```
private static String LeerCadena(String mensaje, int i) {
    String cadena;
    do {
        System.out.print(mensaje);
cadena = sc.nextLine();

        } while (cadena.trim().length() == 0 || cadena.trim().length() > i);
return cadena.trim();
} // LeerCadena

//LEE UN NÚMERO ENTERO COMPRENDIDO ENTRE UNOS LIMITES
private static int LeerEntero(String mensaje, int desde, int hasta) {
String cadena;        int numero = 0;
    do {
        System.out.print(mensaje);
cadena = sc.nextLine();
    try {
        numero = Integer.parseInt(cadena);
    } catch (NumberFormatException e) {
numero = 0;
        }
        } while (numero < desde || numero > hasta);
    return
numero;
} // LeerEntero

//LEE UN NUMERO DOUBLE COMPRENDIDO ENTRE UNOS LIMITES
private static double LeerDouble(String mensaje, int desde, int hasta) {
String cadena;        double numero = 0;
    do {
        System.out.print(mensaje);
cadena = sc.nextLine();
    try {
        numero = Double.parseDouble(cadena);
    } catch (NumberFormatException e) {
numero = 0;
        }
        } while (numero < desde || numero > hasta);
    return
numero;
} // LeerDouble

//LEE UN NUMERO FLOAT COMPRENDIDO ENTRE UNOS LIMITES
private static float LeerFloat(String mensaje, int desde, int hasta) {
String cadena;        float numero = 0;        do {
        System.out.print(mensaje);
cadena = sc.nextLine();
    try {
        numero = Float.parseFloat(cadena);
    } catch (NumberFormatException e) {
numero = 0;
        }
        } while (numero < desde || numero > hasta);
    return
numero;
} // LeerDouble
```

```
//DEVUELVE TRUE SI EL DEPARTAMENTO EXISTE
private static boolean existeDepartamento(int dept_no) throws SQLException {
    Statement sentencia = conexion.createStatement();
    String sql = "select * from departamentos where dept_no = " + dept_no;
    boolean existe = false;
    ResultSet resul = sentencia.executeQuery(sql);
    if (resul.next()) {
        existe = true;
    }
    sentencia.close();
    resul.close();
    return
    existe;
}

//DEVUELVE TRUE SI EL EMPLEADO EXISTE
private static boolean existeEmpleado(int emp_no) throws SQLException {
    Statement sentencia = conexion.createStatement();
    String sql = "select * from empleados where emp_no = " + emp_no;
    boolean existe = false;
    ResultSet resul = sentencia.executeQuery(sql);
    if (resul.next()) {
        existe = true;
    }
    sentencia.close();
    resul.close();

    return existe;
}
}
} // fin
```

FIN ACTIVIDADES RESUELTAS 2

5. Referencias.

Documentación SQLite: <https://www.sqlite.org/docs.html>

Tutorial de SQLite: <https://www.sqlitetutorial.net/>

Tutorial de SQLite: <https://www.techonthenet.com/sqlite/>

Tutorial JDBC: <https://www.tutorialspoint.com/jdbc/index.htm>

Tutorial JDBC:

<https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>

Libro Acceso a Datos. 2ª Edición

Aut: María Jesús Ramos /
Alicia Ramos Ed: Editorial
Garceta.

Fecha: 2016

ISBN: 978-84-1622-860-7