Unidad 2: Manejo de conectores.

Índice de contenidos

1 Acceso a datos.

El acceso a datos **implica la capacidad de recuperar, modificar, agregar o eliminar información dentro de una *BD* o sistema de almacenamiento** (origen de datos)**, según los permisos otorgados al usuario o aplicación que intenta acceder a los datos**.

Los **orígenes de datos** se refieren a las **fuentes de donde se obtienen los datos que pueden ser utilizados** en las aplicaciones. Algunos **ejemplos de orígenes de datos** son:

- ✓ **Sistemas de archivos**: los archivos pueden servir como fuentes de datos (en sistemas de archivos locales o en la nube). Ejemplos: documentos de texto, hojas de cálculo, imágenes, videos, etc.
- ✓ ***BD* relacionales**: como *MariaDB/MySQL*, *PostgreSQL*, *Oracle* o *SQL Server*. Almacenan **datos estructurados en tablas** y son muy utilizadas en todo tipo de aplicaciones.
- ✓ ***BD NoSQL***: como *MongoDB*, *Cassandra* o *Redis*. Se utilizan para gestionar **datos no estructurados o semiestructurados** (documentos, gráficos, datos de clave-valor, etc.).
- ✓ ***API*** (*Interfaces de Programación de Aplicaciones*): permiten que las aplicaciones obtengan datos de servicios web o sistemas externos (datos de redes sociales, información meteorológica, datos financieros, etc.).
- ✓ **Dispositivos *IoT***: los dispositivos *IoT* (*Internet de las cosas*) generan una gran cantidad de datos que pueden ser utilizados para el monitoreo y control en tiempo real.

La **elección de la fuente de datos dependerá de los requisitos específicos del proyecto, así como de la naturaleza de los datos que se necesitan** para lograr los objetivos establecidos.

2 Desfase objeto-relacional.

El desfase objeto-relacional **se refiere a la diferencia que existe entre la forma en que los lenguajes de *POO* representan los datos y los manipulan, y la forma en que las *BD* relacionales almacenan y gestionan los datos**. Esta **discrepancia puede dar lugar a varios desafíos**:

- ✓ **Modelo de datos diferente** (principal del desfase): en los lenguajes de *POO* (*Java*, *C#*, *Python*, etc.) los **datos se representan como objetos con atributos y métodos**. En cambio, las ***BD* relacionales utilizan tablas con filas y columnas para representar los datos**.
- ✓ **Herencia y polimorfismo**: los lenguajes de *POO* admiten conceptos de herencia y polimorfismo, que **pueden no tener un equivalente directo en las *BD* relacionales**. Mapear jerarquías de clases a tablas puede ser complicado y puede requerir estrategias especiales.
- ✓ **Relaciones complejas**: en las *BD* relacionales, las relaciones entre tablas se establecen mediante claves primarias y externas. La **navegación de relaciones complejas en un lenguaje de *POO* puede requerir un código adicional** para traducir entre objetos y tablas.
- ✓ **Lenguaje de consulta**: las *BD* relacionales utilizan *SQL* para recuperar y manipular datos, mientras que los lenguajes de programación utilizan lenguajes de programación más generales. La **diferencia en los lenguajes** puede dificultar la escritura de consultas *SQL* eficientes y comprensibles desde el código de programación.
- ✓ **Dificultades de mapeo**: el proceso de mapear objetos de un lenguaje de programación a tablas de una *BD* y viceversa puede ser complicado y propenso a errores. Esto a menudo se aborda mediante herramientas de mapeo objeto-relacional (*ORM*) que automatizan gran parte de este proceso.
- ✓ **Rendimiento ineficiente**: la **conversión entre objetos y tablas puede ser costosa**, especialmente cuando se manejan grandes volúmenes de datos. Las consultas *SQL* generadas automáticamente por los *ORM* a veces pueden no ser óptimas.
- ✓ **Actualización de esquemas**: cambiar la estructura de una *BD* relacional puede ser complicado, lo que puede afectar la integridad de los datos y la funcionalidad de la aplicación.

Para abordar estos desafíos, se utilizan herramientas de mapeo objeto-relacional (*ORM*) **que proporcionan una capa de abstracción entre la aplicación y la *BD***. Estas herramientas facilitan el mapeo de objetos a tablas y simplifican muchas de las tareas asociadas con el desfase objeto-relacional. Sin embargo, **es importante comprender los fundamentos de la relación entre objetos y *BD* relacionales para diseñar aplicaciones eficientes y efectivas**. Este tema se tratará más ampliamente en la unidad 3.

3 *SGBD* embebidos e independientes.

3.1 Embebidos.

Los *SGBD* embebidos **se incorporan directamente en la aplicación**, lo que significa que **la *BD* se ejecuta en el mismo proceso**. Los *SGBD* embebidos **son adecuados para aplicaciones que necesitan gestionar datos de manera local y no requieren acceso concurrente** (por múltiples usuarios o aplicaciones).

Características de los *SGBD* embebidos:

- ✓ La ***BD* está completamente integrada en la aplicación**.
- ✓ La ***BD* se almacena en un archivo local o en memoria**.
- ✓ **Se utilizan usualmente en aplicaciones móviles, aplicaciones de escritorio y sistemas embebidos**.
- ✓ Ofrecen un muy **buen rendimiento**.
- ✓ **No son adecuados para aplicaciones que requieren acceso concurrente**.

Ejemplos de *SGBD* embebidos: ***SQLite***, ***HSQLDB*** (*Hyperthreaded Structured Query Language Database*), ***Apache Derby***, *H2*, *Db4o*, *Firebird*, *Oracle Embedded* y *MS SQL Server Compact*.

3.1.1 SQLite.

SQLite es un ***SGBD* relacional embebido de código abierto**, ampliamente utilizado por su simplicidad, ligereza y facilidad de integración en una gran variedad de aplicaciones. Algunas **características y detalles importantes** de *SQLite* son:

- ✓ **Autónomo y embebido**: es una **biblioteca *C* que proporciona una *BD* completamente autónoma y embebida** en la aplicación. No requiere un *SGBD* independiente ni configuración externa. La ***BD* se almacena en un archivo local**, lo que facilita su portabilidad.
- ✓ **Soporte completo de *SQL***: es compatible con *SQL* y admite la mayoría de sus operaciones.
- ✓ **Transacciones *ACID***: garantiza la integridad de los datos y la consistencia a través de transacciones *ACID* (*Atomicidad*, *Consistencia*, *Aislamiento*, *Durabilidad*).
- ✓ **Ligero y eficiente**: está diseñado para ser eficiente en cuanto a recursos y tener un **uso de memoria bajo**.
- ✓ **Ampliamente utilizado**: se utiliza en una gran variedad de aplicaciones, navegadores web (para el almacenamiento de cookies y datos de caché), sistemas de gestión de versiones (como *Git* y *Subversion*), aplicaciones móviles, …
- ✓ **Multiplataforma**: es compatible con múltiples plataformas (*Windows*, *macOS*, *Linux*, *Android* e *iOS*).
- ✓ **Gratuito**: es de **código abierto y se distribuye bajo una licencia de dominio público** (es gratuito para su uso en proyectos comerciales y no comerciales sin la necesidad de pagar licencias).
- ✓ **Extensible**: admite extensiones y módulos escritos en *C* o mediante la *API* de carga de extensiones.

SQLite **es una elección popular para aplicaciones que requieren almacenamiento local de datos**.

Descargar *SQLite*:

1. Visitar el **sitio web oficial de *SQLite*** en https://www.sqlite.org/.
2. En la página de inicio, buscar la **sección *Download* y seleccionar la versión que mejor se adapte al *SO***.
3. **Descargar el archivo binario**.

En ***Windows***:

1. **Descomprimir el archivo *ZIP* descargado** (`sqlite-tools-win-x64-XXXXXXX.zip`).
2. **Agregar la ubicación de la carpeta descomprimida a la variable de entorno PATH** para que se puedan ejecutar comandos *SQLite* desde cualquier ubicación en la línea de comandos.

Para ***Linux*** (*Ubuntu*/*Debian*):

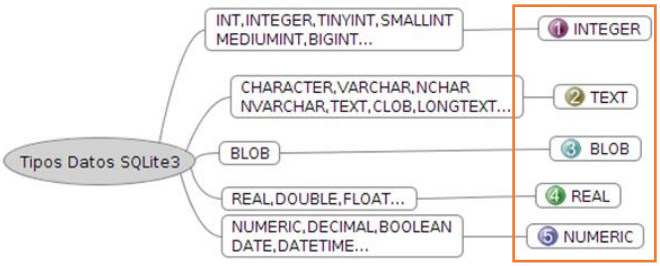
1. **Descomprimir el archivo *ZIP* descargado** (`sqlite-tools-linux-x64-XXXXXXX.zip`).
2. **Abrir una terminal**.
3. **Navegar a la ubicación donde se descomprimió el archivo**.
4. **Copiar el archivo binario de *SQLite* en una ubicación accesible**: `sudo cp sqlite3 /usr/local/bin/`

Otra opción en ***Ubuntu*** es ejecutar desde la **terminal**: **`sudo apt update && sudo apt install sqlite3`**

Para **verificar que *SQLite* se ha instalado correctamente**, abrir una terminal (en *Windows*, abrir el símbolo del sistema) y **ejecutar el siguiente comando**:

```
sqlite3 --version
```

Tipos de datos en *SQLite*:



Para ejecutar *SQLite* y trabajar con una *BD* "my_db.db" ejecutar las sentencias que se muestran en la imagen. Si la *BD* indicada no existe, se crea, si existe se carga.

```
C:\java\sqlite>sqlite3 c:\java\my_db.db
SQLite version 3.43.1 2023-09-11 12:01:27
Enter ".help" for usage hints.
sqlite> CREATE TABLE departamento (
(x1...> id_departamento INTEGER PRIMARY KEY,
(x1...> nombre TEXT,
(x1...> localidad TEXT);
sqlite> .tables
departamento
sqlite> INSERT INTO departamento VALUES (1, 'Contabilidad', 'Sevilla');
sqlite> INSERT INTO departamento VALUES (2, 'Investigación', 'Madrid');
sqlite> INSERT INTO departamento VALUES (3, 'Ventas', 'Barcelona');
sqlite> INSERT INTO departamento VALUES (4, 'Producción', 'Granada');
sqlite> SELECT * FROM departamento;
1|Contabilidad|Sevilla
2|Investigación|Madrid
3|Ventas|Barcelona
4|Producción|Granada
sqlite> .quit
```

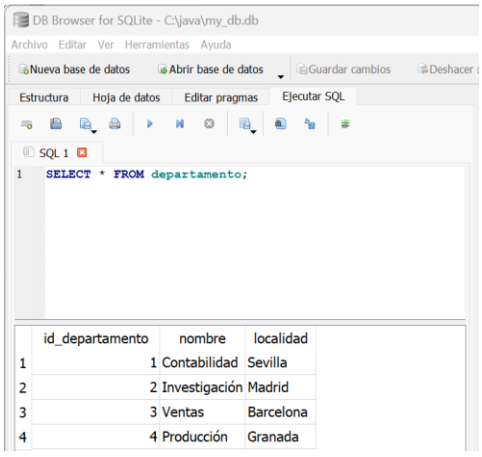
Órdenes básicas:

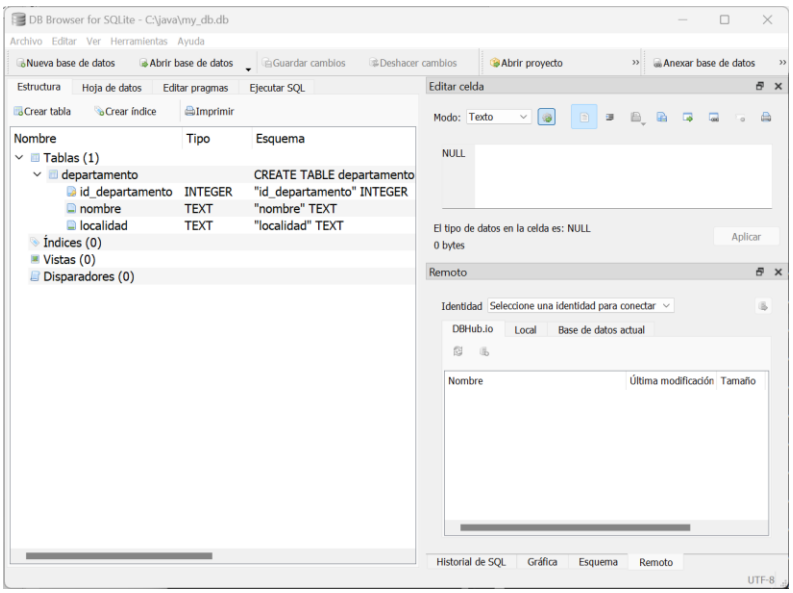
- ✓ **`.help`**: obtener ayuda.
- ✓ **`.quit`**: salir de *SQLite*.
- ✓ **`.open <nombre_archivo.db>`**: abrir una *BD* desde dentro de *SQLite*.
- ✓ **`.show`**: mostrar la configuración actual de *SQLite*.
- ✓ **`.database`**: mostrar información de la *BD* seleccionada.
- ✓ **`.tables`**: obtener un listado con las tablas de la *BD* abierta.
- ✓ **`.schema <nombre_tabla>`**: ver el esquema utilizado en una tabla.
- ✓ **`.mode columna`**: mostrar el resultado de la sentencia `SELECT` en modo columna.
- ✓ **`.output <nombre_fichero>`**: enviar el resultado de las sentencia ejecutadas a un fichero.
- ✓ **`.once <nombre_fichero>`**: enviar a un fichero únicamente el resultado de la próxima consulta.

Existen diferentes **herramientas gráficas que facilitan el trabajo con *SQLite***:

- ✓ ***SQLiteStudio***: de código abierto y multiplataforma. Proporciona una interfaz gráfica para crear y gestionar *BD*, escribir consultas *SQL*, importar/exportar datos, … Sitio web: https://sqlitestudio.pl/
- ✓ ***DBeaver***: es de código abierto y admite múltiples *SGBD*, incluido *SQLite*. Ofrece una interfaz gráfica potente y fácil de usar para trabajar con *BD SQLite*, así como la capacidad de ejecutar consultas *SQL* y gestionar esquemas. Sitio web: https://dbeaver.io/
- ✓ ***DB Browser for SQLite***: es una **herramienta de código abierto específicamente diseñada para trabajar con *SQLite***. Es ligera y fácil de usar, lo que la hace ideal para tareas de administración de *BD SQLite*. Se pueden crear, editar, eliminar y consultar datos de forma sencilla. Sitio web: https://sqlitebrowser.org/

BD creada anteriormente desde la terminal, abierta con la herramienta gráfica *DB Browser for SQLite*.





3.2 Independientes.

Los *SGBD* independientes **son *SGBD* que se ejecutan como procesos separados y gestionan las *BD* de manera independiente de las aplicaciones que las utilizan**. Estos gestores **son adecuados para aplicaciones que requieren acceso concurrente desde múltiples usuarios o aplicaciones**.

Características de los *SGBD* independientes:

- ✓ **Se ejecutan como servicios o procesos independientes**.
- ✓ **Permiten el acceso simultáneo a las *BD*** desde múltiples ubicaciones o usuarios.
- ✓ Son adecuados **para aplicaciones web, sistemas de gestión empresarial (*ERP*), aplicaciones de servidor**, …
- ✓ **Proporcionan capacidades de seguridad y control de acceso avanzadas**.
- ✓ Suelen ser **más escalables y robustos** en entornos de producción.

Ejemplos de *SGBD* independientes: ***MariaDB/MySQL***, ***PostgreSQL***, ***Microsoft SQL Server*** y ***Oracle***.

Los SGBD embebidos son útiles cuando se desea una solución ligera y local para el almacenamiento de datos, mientras que los SGBD independientes son esenciales para aplicaciones empresariales o sistemas que requieren un acceso concurrente y robustez en un entorno de red.

En esta unidad se va a trabajar principalmente con *MariaDB* y *SQLite*.

4 Protocolos de acceso a *BD*. Conectores.

Los protocolos de acceso a *BD* y los conectores **son componentes esenciales para la comunicación entre las aplicaciones y los *SGBD*, permitiendo que las aplicaciones envíen solicitudes y reciban respuestas desde el *SGBD***.

Protocolos (reglas y convenciones que establecen cómo se puede acceder y manipular la información almacenada en una *BD*) **de acceso a *BD***:

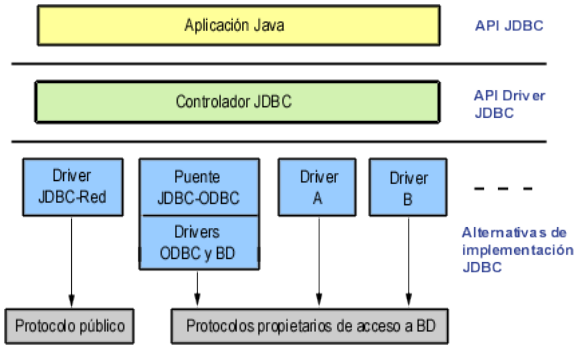
- ✓ ***ODBC*** (*Open Database Connectivity*): **estándar que proporciona una *API* de acceso a *BD* independiente del sistema y del lenguaje**. Los controladores (drivers) *ODBC* están disponibles para muchas plataformas y *SGBD*.
- ✓ ***JDBC*** (*Java Database Connectivity*): **estándar de acceso a *BD* para el lenguaje de programación *Java***. Proporciona una *API* para interactuar con *SGBD* relacionales desde las aplicaciones.
- ✓ ***ADO.NET***: **tecnología de acceso a *BD* desarrollada por *Microsoft* para aplicaciones en *.NET***. Proporciona una biblioteca de clases que permite a las aplicaciones *.NET* interactuar con *SQL Server* y otros *SGBD*.
- ✓ ***OLE DB*** (*Object Linking and Embedding Database*): **tecnología de acceso a *BD* desarrollada por *Microsoft* que permite a las aplicaciones de *Windows* acceder a una variedad de fuentes de datos** (*BD* relacionales, hojas de cálculo, archivos, etc.).
- ✓ ***ADO*** (*ActiveX Data Objects*): **tecnología de acceso a datos de *Microsoft* que se utiliza en entornos de desarrollo de *Windows***.

Conectores (controladores o drivers, facilitan la comunicación entre la aplicación y la *BD*):

- ✓ ***ODBC Driver***: conector específico para *ODBC* que **permite a una aplicación conectarse a una *BD* a través de *ODBC***. Los controladores *ODBC* están disponibles para muchos *SGBD*.
- ✓ ***JDBC Driver***: **conector específico para *Java*** que permite que una aplicación *Java* se comunique con un *SGBD*. **Cada *SGBD* suele proporcionar su propio controlador *JDBC***.
- ✓ ***Entity Framework***: *ORM* (*Object-Relational Mapping*) de *Microsoft* que proporciona una capa de abstracción sobre *BD* relacionales en entornos *.NET*. Utiliza controladores específicos para conectarse a diferentes *SGBD*.
- ✓ ***Data Providers de ADO.NET***: para **acceder a *BD* desde aplicaciones *.NET***.
- ✓ ***Driver OLE DB***: es un **componente que permite a las aplicaciones de *Windows* acceder a diversas fuentes de datos** mediante la tecnología *OLE DB*. Cada *SGBD* suele proporcionar su propio driver *OLE DB*.

Estos protocolos y conectores son esenciales para que las aplicaciones puedan interactuar de manera efectiva con los *SGBD*, independientemente del lenguaje de programación o el *SGBD* utilizado. **La elección del protocolo y el conector adecuado dependerá de la tecnología y el entorno de desarrollo específico que se utilice**.

5 Acceso a datos mediante *JDBC*.



***JDBC* es una especificación de un conjunto de clases y métodos que permiten a cualquier programa *Java* acceder a cualquier *SGBD* de forma homogénea**. La aplicación *Java* debe tener acceso a un driver *JDBC* adecuado, este driver es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el *API JDBC* y el *SGBD* real.

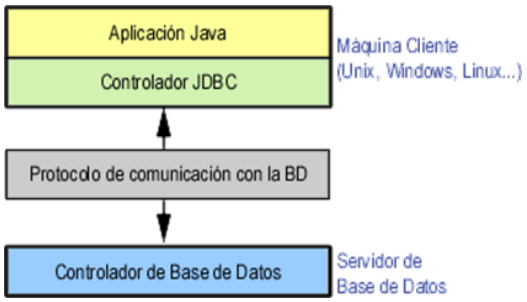
La necesidad de *JDBC*, a pesar de la existencia de *ODBC*, viene dada porque *ODBC* es una interfaz escrita en lenguaje *C*, que, al no ser un lenguaje portable, haría que las aplicaciones *Java* también perdiesen la portabilidad. Además, *ODBC* tiene el inconveniente de que se ha de instalar manualmente en cada equipo; al contrario que los **drivers *JDBC***, que **al estar escritos en *Java* son automáticamente instalables, portables y seguros**.

***JDBC* define una arquitectura estándar** para que los fabricantes puedan crear los drivers que permitan a las aplicaciones *Java* el acceso a los datos.

5.1 Modelos de acceso a *BD*.

JDBC* es una tecnología que permite a las aplicaciones *Java* interactuar con *BD. Se puede implementar el acceso a *BD* con *JDBC* utilizando **diferentes modelos de arquitectura**, como el modelo de dos capas (*2-Tier*) y el modelo de tres capas (*3-Tier*).

5.1.1 Modelo de dos capas (2-Tier).



El modelo de dos capas, también conocido como el **modelo cliente-servidor**, es uno de los enfoques más simples para el acceso a *BD*. En este modelo, **la aplicación se divide en dos capas principales**:

- ✓ **Capa de presentación** (cliente): **se encarga de la interfaz de usuario y la lógica de la aplicación**. Se encuentra la interfaz de usuario que envía solicitudes de acceso a la *BD*.
- ✓ **Capa de datos** (servidor): **se encuentra la lógica para acceder a la *BD***. Se utiliza *JDBC* para establecer la conexión con el *SGBD*, enviar consultas *SQL*, recuperar resultados y realizar operaciones sobre la *BD*.

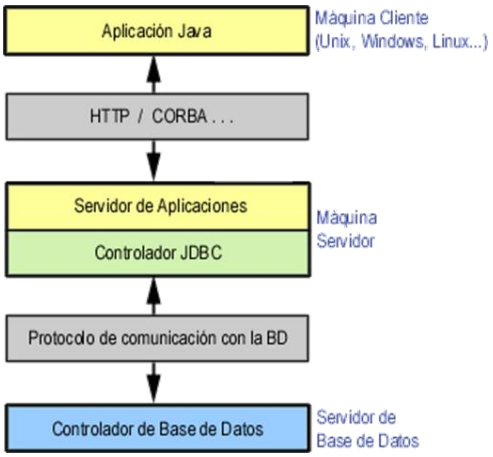
Ventajas:

- ✓ **Simplicidad**: este modelo es fácil de implementar y adecuado para aplicaciones pequeñas y simples.
- ✓ **Menor latencia**: las solicitudes de acceso a la *BD* son manejadas directamente por la aplicación.

Desventajas:

- ✓ **Falta de separación de preocupaciones**: la lógica de acceso a la *BD* se mezcla con la lógica de la aplicación en la capa de presentación.
- ✓ **Escalabilidad limitada**: no es ideal para aplicaciones que requieren escalabilidad, ya que la lógica de la *BD* se encuentra en la misma máquina que la aplicación.
- ✓ El **driver *JDBC* específico** para conectarse con el *SGBD*, **debe residir en el sistema local**.

Este modelo se basa en que la aplicación Java se conecta directamente al SGBD.



5.1.2 Modelo de tres capas (3-Tier).

El modelo de tres capas es un **enfoque más estructurado que separa la aplicación en tres capas**:

- ✓ **Capa de presentación** (interfaz de usuario): **se encarga de la presentación de la información al usuario y de capturar sus interacciones**. Contiene la lógica de la interfaz de usuario, como la

presentación de datos y la respuesta a eventos del usuario.

- ✓ **Capa de lógica de negocio** (capa intermedia): esta capa **procesa y gestiona los datos según las reglas y la lógica específica de la aplicación**. Es responsable de aplicar las reglas de negocio y garantizar la coherencia y la integridad de los datos. Esta capa actúa como intermediaria entre la capa de presentación y la capa de datos.
- ✓ **Capa de datos**: se encarga de **gestionar el acceso y la manipulación de los datos**. Aquí se realizan operaciones directas sobre la *BD*, como consultas, inserciones, actualizaciones y eliminaciones. La capa de datos proporciona una interfaz para que la capa de lógica de negocio interactúe con la *BD*.

Ventajas:

- ✓ **Separación de preocupaciones**: la **lógica de la aplicación se divide en capas**, lo que facilita la gestión y el mantenimiento del código.
- ✓ **Escalabilidad**: más adecuado para aplicaciones escalables, ya que **las capas pueden distribuirse en servidores separados** si es necesario.
- ✓ Los **drivers *JDBC* no tienen que residir en la máquina cliente**, lo cual libera al usuario de la instalación de cualquier tipo de driver.

Desventajas:

- ✓ **Mayor complejidad**: la implementación y el mantenimiento de tres capas puede ser más complejo que en el modelo de dos capas, especialmente para aplicaciones pequeñas.

Las instrucciones son enviadas a una capa intermedia entre Cliente y Servidor, que es la que se encarga de enviar las sentencias SQL al SGBD y recoger el resultado. El usuario no tiene contacto directo, ni a través de la red, con el SGBD.

5.2 Interfaces y clases involucradas en el acceso al *SGBD* con *JDBC*.

El **paquete `java.sql`** (contiene la *API* de *JDBC*) **contiene clases e interfaces fundamentales para trabajar con *BD* relacionales y realizar operaciones de acceso a datos**. **Clases e interfaces más importantes** de este paquete:

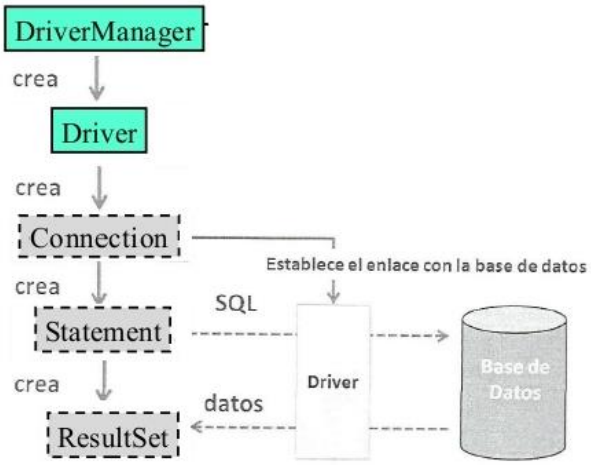
- ✓ **`DriverManager`**: se utiliza **para gestionar los distintos controladores/drivers** (cada driver permite acceder a un *SGBD* concreto). Ayuda en la **creación y gestión de conexiones al *SGBD***.
- ✓ **`Driver`**: es **específico de cada *SGBD***. Cuando un fabricante desarrolla un nuevo *SGBD* debe desarrollar el driver que **permite su uso con *JDBC***. Se utiliza **para registrar un controlador de *BD* en el `DriverManager`**.

Cada fabricante desarrolla su driver JDBC que conoce las peculiaridades del SGBD asociado, haciendo de puente/traductor entre el resto de las clases de la API de JDBC y el SGBD.

- ✓ **`DriverPropertyInfo`**: se utiliza **para obtener información sobre las propiedades y opciones de configuración admitidas por un controlador de *BD* específico**.
- ✓ **`Connection`**: **representa una conexión entre la aplicación y el *SGBD***, permitiendo que las sentencias *SQL* viajen desde la aplicación al *SGBD* y los resultados de las consultas se muevan en sentido contrario. Puede utilizarse para crear objetos `Statement`, gestionar transacciones, etc.
- ✓ **`DatabaseMetaData`**: **proporciona información sobre la *BD* a la que se está conectado**. Puede utilizarse para obtener detalles sobre la estructura de la *BD* (tablas, procedimientos almacenados, capacidades del *SGBD*...).
- ✓ **`Statement`**: se utiliza **para ejecutar sentencias *SQL* sin parámetros** (representa una sentencia *SQL*).
- ✓ **`PreparedStatement`**: **se utiliza para ejecutar sentencias *SQL* precompiladas con parámetros**. Es **útil para prevenir ataques de *SQL Injection* y mejorar la eficiencia al ejecutar la misma sentencia varias veces**.
- ✓ **`CallableStatement`**: se utiliza **para ejecutar procedimientos almacenados**.
- ✓ **`ResultSet`**: **representa el conjunto de resultados de una consulta *SQL*** (tabla con el resultado). Permite iterar a través de los registros devueltos por la consulta y acceder a los datos de cada registro.
- ✓ **`ResultSetMetaData`**: se utiliza **para obtener información sobre los metadatos de un conjunto de resultados** (`ResultSet`). Proporcionar detalles como nombres de columnas, tipos de datos, tamaños, etc.
- ✓ **`SQLException`**: **representa una excepción relacionada con el acceso a la *BD***. Se lanza cuando ocurren errores durante la ejecución de las sentencias *SQL*.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/package-summary.html

En la imagen se pueden ver las interfaces (gris con trazo discontinuo) y las clases (verde con trazo continuo) involucradas en un acceso a *BD* y el orden en el que deben ser utilizadas.



El **paquete `java.sql`** (está incorporado en el *JDK*) **proporciona las interfaces y clases fundamentales para establecer conexiones a *BD* relacionales, ejecutar consultas y procesar resultados**. Algunas de las **clases más importantes son `Connection`, `Statement`, `PreparedStatement` y `ResultSet`. Ofrece una capa de abstracción que permite a los desarrolladores interactuar con *BD* de manera uniforme, independientemente del *SGBD* utilizado**.

Cada *SGBD* tiene su propio protocolo de comunicaciones y formato de datos. Los drivers son necesarios para traducir las solicitudes realizadas a través de las interfaces de `java.sql` en comandos que el *SGBD* puede entender (los drivers implementan las interfaces de `java.sql`). Estos **drivers no vienen incluidos en el *JDK* y deben ser agregados al proyecto**.

Las **aplicaciones interactúan con el paquete `java.sql` y este paquete interactúa con el driver**.

Ejemplo: mostrar el contenido de una tabla alojada en un *SGBD MariaDB* haciendo uso de *JDBC*.

Asegurarse de tener el controlador *JDBC* de *MariaDB* en el proyecto. Descargar desde el sitio web oficial de *MariaDB* o a través de herramientas de gestión de dependencias como *Maven* o *Gradle*.

- *Maven*: agregar las siguientes dependencias en el archivo `pom.xml` dentro de la sección `<dependencies>`.

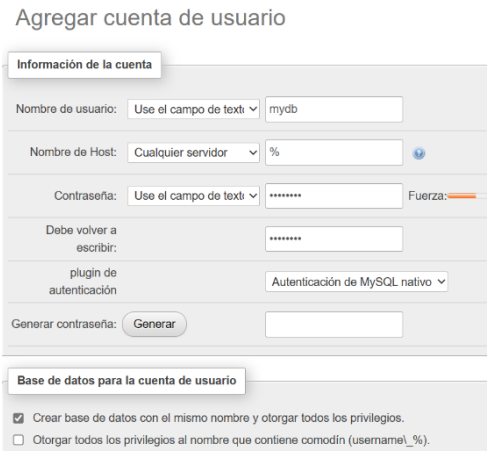
```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
    <version>3.4.1</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.16</version>
</dependency>
```

Aunque no es estrictamente necesario tener un *logger* para usar los drivers *JDBC* de *MariaDB*, es altamente recomendado para mejorar la visibilidad y el control sobre la interacción con la *BD* (los *loggers* pueden generar registros de actividad y errores que son útiles para el desarrollo y la depuración).

- *Gradle*: agregar las siguientes dependencias en el archivo `build.gradle` en la sección `dependencies`.

```
implementation 'org.mariadb.jdbc:mariadb-java-client:3.4.1'
implementation 'org.slf4j:slf4j-simple:2.0.16'
```

Una vez se haya descargado o configurado el controlador *JDBC* de *MariaDB*, se podrá utilizar en el proyecto para conectarte y trabajar con *BD MariaDB* desde *Java*.

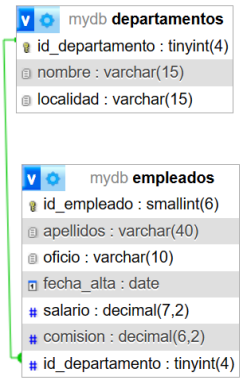


Desde *phpMyAdmin* crear un usuario *mydb* con clave *password* que tenga todos los permisos sobre la *BD mydb* (*Cuentas de usuarios* → *Agregar cuenta de usuario*).

Ejecutar las siguientes sentencias *SQL* para crear las tablas necesarias e insertar datos en ellas:

```
CREATE TABLE departamentos (
    id_departamento TINYINT PRIMARY KEY,
    nombre VARCHAR(15),
    localidad VARCHAR(15)
);
INSERT INTO departamentos VALUES (1,'Contabilidad','Sevilla'), (2,'Investigación','Madrid'),
                                 (3,'Ventas','Barcelona'), (4,'Producción','Granada');
CREATE TABLE empleados (
    id_empleado SMALLINT PRIMARY KEY,
    nombre VARCHAR(40),
    oficio VARCHAR(10),
    fecha_alta DATE,
    salario DECIMAL(7, 2),
    comision DECIMAL(6, 2),
    id_departamento TINYINT NOT NULL,
    FOREIGN KEY(id_departamento) REFERENCES departamentos(id_departamento)
);
INSERT INTO empleados VALUES (7369, 'Luis Sánchez', 'EMPLEADO', '2010/12/17', 1040, NULL, 2),
              (7499, 'Carlos Arroyo', 'VENDEDOR', '2010/02/20', 1500, 390, 3),
              (7521, 'María Sala', 'VENDEDOR', '2011/02/22', 1625, 650, 3),
```

```
(7566, 'Luis Jiménez', 'DIRECTOR', '2011/04/02', 2900, NULL, 2),
(7654, 'José Martín', 'VENDEDOR', '2011/09/29', 1600, 1020, 3),
(7698, 'Enrique Negro', 'DIRECTOR', '2011/05/01', 3005, NULL, 3),
(7782, 'Javier Cerezo', 'DIRECTOR', '2011/06/09', 2885, NULL, 1),
(7788, 'Álvaro Gil', 'ANALISTA', '2011/11/09', 3000, NULL, 2),
(7839, 'Barbara Rey', 'PRESIDENTE', '2011/11/17', 4100, NULL, 1),
(7844, 'Luis Tovar', 'VENDEDOR', '2011/09/08', 1350, 0, 3),
(7876, 'Pepe Alonso', 'EMPLEADO', '2011/09/23', 1430, NULL, 2),
(7900, 'Francisco Jimeno', 'EMPLEADO', '2011/12/03', 1335, NULL, 3),
(7902, 'Luisa Fernández', 'ANALISTA', '2011/12/03', 3000, NULL, 2),
(7934, 'Nicolás Muñoz', 'EMPLEADO', '2012/01/23', 1690, NULL,1);
```



Código aplicación *Java* para acceder al *SGBD MariaDB*:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class MariaDBEjemplo {
    public static void main(String[] args) {
        // Datos de conexión a la BD mydb alojada en un servidor MariaDB
        String jdbcURL = "jdbc:mariadb://localhost:3306/mydb"; // Puerto 3306 por defecto, puede no ponerse
        // String jdbcURL = "jdbc:mysql://localhost:3306/mydb"; --> Para MySQL
        String usuario = "mydb", clave = "password";
        // Declaraciones JDBC
        Connection conexion = null;
        Statement statement = null;
        ResultSet resultSet = null;
        try {
            Class.forName("org.mariadb.jdbc.Driver"); // Paso 1: Cargar el driver JDBC de MariaDB – NO NECESARIO
            // Class.forName("com.mysql.cj.jdbc.Driver"); --> Para MySQL
            // Paso 2: Establecer la conexión
            conexion = DriverManager.getConnection(jdbcURL, usuario, clave);
            statement = conexion.createStatement(); // Paso 3: Crear una declaración
            // Paso 4: Preparar y ejecutar una consulta SQL
            String sqlQuery = "SELECT * FROM departamentos";
            resultSet = statement.executeQuery(sqlQuery);
            // Paso 5: Procesar los resultados. Se repite el bucle mientras haya registros en el ResultSet
            while (resultSet.next())
                System.out.println("ID: " + resultSet.getInt("id_departamento") + ", Nombre: " +
                        resultSet.getString("nombre") + ", Ciudad: " + resultSet.getString("localidad"));
        } catch (ClassNotFoundException e) { // Puede causarla Class.forName(...)
            System.out.println("Error al cargar el controlador JDBC: " + e.getMessage());
        } catch (SQLException e) { // Puede causarla el resto de métodos SQL
            System.out.println("Error de SQL: " + e.getMessage());
        } finally {
           try { // Paso 6: Cerrar la conexión y liberar recursos, en orden inverso al que se crean
                if (resultSet != null)
                    resultSet.close(); // Cerrar ResultSet
                if (statement != null)
                    statement.close(); // Cerrar Statement
                if (conexion != null)
                    conexion.close(); // Cerrar conexión
            } catch (SQLException e) { // Puede causarla el cierre y liberación de los recursos
                System.out.println("Error al cerrar la conexión y liberar recursos: " + e.getMessage());
            }
        }
    }
}
```

DriveManager.drivers() devuelve un Stream<Driver> que permite usar forEach para iterar sobre los controladores registrados en el DriveManager:

```
DriverManager.drivers().forEach(driver -> System.out.println(driver.toString()));
```

Se puede **utilizar un try-with-resources para manejar automáticamente el cierre de recursos** como conexiones, declaraciones y resultados relacionados con *JDBC*.

SQLException es una clase en *Java* que forma parte del paquete `java.sql`. Se utiliza para **manejar errores que ocurren durante las operaciones relacionadas con *BD***.

Ejemplo: mostrar el contenido de una tabla *SQLite* haciendo uso de *JDBC*.

Agregar el controlador *JDBC* de *SQLite* al proyecto:

- *Maven*: agregar la siguiente dependencia en el archivo `pom.xml` dentro de la sección `<dependencies>`.

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.46.1.3</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.16</version>
</dependency>
```

- *Gradle*: agregar la siguiente dependencia en el archivo `build.gradle` dentro de la sección `dependencies`.

```
implementation 'org.xerial:sqlite-jdbc:3.46.1.3'
implementation 'org.slf4j:slf4j-simple:2.0.16'
```

Haciendo uso de *DB Browser for SQLite* ejecutar las sentencias *SQL* del ejemplo anterior para crear las tablas necesarias e insertar datos en ellas. Tener en cuenta que los tipos de datos indicados, aunque no se tengan en *SQLite*, se traducirán al tipo correspondiente internamente. Dar el nombre `mydb.db` a la *BD*.

Código aplicación *Java* para acceder a la *BD SQLite* `mydb.db`:

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

public class SQLiteEjemplo {
    public static void main(String[] args) {
        // Ruta de la BD SQLite (puede ser una ruta absoluta o relativa)
        String dbURL = "jdbc:sqlite:C:\\java\\mydb.db"; // En lugar de utilizar \\ se puede utilizar /

        // Declaraciones JDBC
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            Class.forName("org.sqlite.JDBC"); // Paso 1: Cargar el controlador JDBC de SQLite – NO NECESARIO

            connection = DriverManager.getConnection(dbURL); // Paso 2: Establecer la conexión con la BD

            statement = connection.createStatement(); // Paso 3: Crear una declaración SQL

            String sqlQuery = "SELECT * FROM departamentos"; // Paso 4: Ejecutar una consulta SQL
            resultSet = statement.executeQuery(sqlQuery);

            while (resultSet.next()) // Paso 5: Procesar los resultados
                System.out.println("ID: " + resultSet.getInt("id_departamento") + ", Nombre: " +
                        resultSet.getString("nombre") + ", Ciudad: " + resultSet.getString("localidad"));
        } catch (ClassNotFoundException e) {
            System.out.println("Error al cargar el controlador JDBC: " + e.getMessage());
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        } finally {
            try { // Paso 6: Cerrar la conexión y liberar recursos
                if (resultSet != null)
                    resultSet.close();
                if (statement != null)
                    statement.close();
                if (connection != null)
                    connection.close();
            } catch (SQLException e) {
                System.out.println("Error al cerrar la conexión y liberar recursos: " + e.getMessage());
            }
        }
    }
}
```

Los dos ejemplos propuestos, tienen similitudes en su estructura general y en cómo interactúan con la *BD* mediante *JDBC*. Sin embargo, también existen diferencias debido a las particularidades de cada *SGBD*:

- ✓ Controlador *JDBC*: para *MariaDB* es "`org.mariadb.jdbc.Driver`" y para *SQLite* es "`org.sqlite.JDBC`". Esto se debe a que cada *BD* requiere su propio controlador *JDBC*.
- ✓ *URL* de conexión: para *MariaDB* es "`jdbc:mariadb://localhost:3306/mydb`" e incluye el nombre del servidor, el puerto y la *BD*. En cambio, para *SQLite* es "`jdbc:sqlite:c:\\java\\mydb.db`", que simplemente especifica la ubicación del archivo de *BD SQLite* en el sistema de archivos local.
- ✓ Establecimiento de la conexión: cambia entre los dos programas debido a las diferencias en la *URL* de conexión y los detalles de autenticación. En *MariaDB* hay que indicar el usuario y clave.

5.3 Método estático `Class.forName()`.

El método `Class.forName()` es un método en *Java* que se utiliza para cargar dinámicamente una clase en tiempo de ejecución. En *JDBC* se utiliza para **cargar el controlador *JDBC*** necesario para establecer la conexión con la *BD*.

Su **sintaxis** es: **`Class.forName("nombre_de_la_clase");`**, donde **"nombre_de_la_clase"** es una **cadena que contiene el nombre completo de la clase que se desea cargar**. Por ejemplo, si se está trabajando con *MySQL*, se

cargaría el controlador *JDBC* de *MySQL* de la siguiente forma:

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

En versiones de *Java* recientes (> *Java 6*), **no es necesario utilizar `Class.forName()` para cargar los controladores *JDBC***, ya que ***JDBC 4.0* y posteriores incluyen un mecanismo de autodetección de controladores**. En estos casos, simplemente incluir el controlador en el proyecto o configuración de la aplicación es suficiente para que *JDBC* lo detecte automáticamente. Sin embargo, **aún es común ver `Class.forName()` en código *JDBC* más antiguo o en situaciones donde se requiere una carga explícita del controlador**.

Tabla con **ejemplos** de usos de `Class.forName()` para **cargar controladores *JDBC* de diferentes *SGBD***:

SGBD	Ejemplo de `Class.forName()`
MariaDB	`Class.forName("org.mariadb.jdbc.Driver");`
SQLite	`Class.forName("org.sqlite.JDBC");`
MySQL	`Class.forName("com.mysql.cj.jdbc.Driver");`
PostgreSQL	`Class.forName("org.postgresql.Driver");`
Oracle Database	`Class.forName("oracle.jdbc.driver.OracleDriver");`
Microsoft SQL Server	`Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");`
H2 Database	`Class.forName("org.h2.Driver");`
IBM DB2	`Class.forName("com.ibm.db2.jcc.DB2Driver");`
Apache Derby	`Class.forName("org.apache.derby.jdbc.EmbeddedDriver");`
Sybase	`Class.forName("com.sybase.jdbc4.jdbc.SybDriver");`
HSQLDB	`Class.forName("org.hsqldb.JDBCDriver");`
Microsoft Access 2007 o posterior	`Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");`

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Class.html

5.4 Método estático `DriverManager.getConnection()`.

`getConnection()` **se utiliza para establecer una conexión a una *BD***. Este método se encuentra en la **clase `DriverManager`** (paquete `java.sql.DriverManager`) que **es responsable de administrar la lista de controladores de *BD* disponibles y permite a los programas de *Java* abrir conexiones a los distintos *SGBD* utilizando estos controladores**.

Sintaxis:
```
Connection getConnection(String url[, String user, String password])
                                    throws SQLException | SQLTimeoutException;
```

El método `getConnection()` puede requerir tres **parámetros**:

- ✓ ***URL* de conexión** (`url`): **especifica la dirección y otros detalles necesarios para conectarse a la *BD*. Depende del *SGBD* que se esté utilizando**.
- ✓ **Nombre de usuario** (`user`): es el **nombre de usuario que se utilizará para autenticarse en el *SGBD***.
- ✓ **Contraseña** (`password`): es la **contraseña asociada al nombre de usuario para la autenticación**.

Este método **devuelve un objeto de tipo `Connection`**, que **representa la conexión establecida** entre la aplicación y el *SGBD*.

Si **se produce algún error al crear la conexión**, lanzará alguna de las siguientes **excepciones**:

- ✓ **`SQLException`**: si ocurre algún error en el acceso a la *BD* o la *URL* es `null`.
- ✓ **`SQLTimeoutException`**: cuando el tiempo que ha transcurrido sin llegar a conectar a la *BD* es excesivo.

En la mayoría de las llamadas al método `getConnection()`, se deberá proporcionar tanto el nombre de usuario como la contraseña, pero **hay situaciones específicas en las que las credenciales pueden no ser obligatorias** (algunos *SGBD* permiten configurar un acceso público o sin contraseña).

Nota: el uso de una contraseña sin cifrar en *JDBC* no es muy seguro, ya que esta viajará por la red desde la aplicación hasta el *SGBD* sin ningún tipo de seguridad. Para evitar esto es necesario enviar la contraseña cifrada en lugar de como texto en claro. Los mecanismos para cifrar las contraseñas son ajenos a *JDBC*, por lo que, para no complicar esta unidad

se utilizarán contraseñas sin cifrar. *MariaDB*: https://mariadb.com/kb/en/using-tls-ssl-with-mariadb-java-connector/

Nota: existen *SO* y *SGBD* que no son sensibles a mayúsculas y minúsculas, sin embargo, los *SO* tipo *GNU/Linux* sí lo son. Se recomienda escribir siempre los nombres de *BD*, tablas, campos, usuarios, etc. de forma idéntica a como se crearon.

La **conexión con la *BD* es la única parte que depende directamente del *SGBD* que se vaya a utilizar**. No es un cambio muy grande puesto que simplemente hay que seleccionar el driver adecuado (que depende de cada *SGBD*) y la cadena de conexión (que también dependerá del *SGBD*). Simplemente **dependiendo del driver que se haya seleccionado la aplicación deberá cargar el driver y cadena de conexión adecuada**.

A la hora de desconectar, basta con **cerrar la conexión**, que será la **misma operación independientemente del driver utilizado**. **Sintaxis**:

```
void close()
```

Ejemplos de ***URLs* de conexión *JDBC* para diferentes *SGBD***:

SGBD	URL de Conexión JDBC	Puerto por defecto
MariaDB	jdbc:mariadb://<host>[:<port>]/<nombre_de_base_de_datos>	3306
MySQL	jdbc:mysql://<host>[:<port>]/<nombre_de_base_de_datos>	3306
PostgreSQL	jdbc:postgresql://<host>[:<port>]/<nombre_de_base_de_datos>	5432
Oracle	jdbc:oracle:thin:@<host>[:<port>]:<SID>	1521
SQL Server	jdbc:sqlserver://<host>[:<port>];databaseName=<nombre_de_base_de_datos>	1433
IBM DB2	jdbc:db2://<host>[:<port>]/<nombre_de_base_de_datos>	50000
SQLite	jdbc:sqlite:<ruta_al_archivo> | jdbc:sqlite:memory: (en memoria)	---
H2 Database	jdbc:h2:<ruta_al_archivo>	---
Apache Derby	jdbc:derby:<ruta_al_archivo>	---
HSQLDB	jdbc:hsqldb:file:<ruta_al_archivo>	---
Access 2007+	jdbc:ucanaccess://<ruta_al_archivo>	---

Reemplazar <host>, <port>, <nombre_de_base_de_datos>, <SID> (identificador de servicio) **y <ruta_al_archivo> con los valores específicos de la configuración de cada *SGBD***. Cada *URL* de conexión está formateada de manera diferente según el *SGBD* que se esté utilizando.

Método	Descripción
`Connection DriverManager.getConnection(String url, String user, String password)`	Constructor para crear una nueva conexión a la BD utilizando una URL de conexión, nombre de usuario y contraseña.
`Connection DriverManager.getConnection(String url)`	Constructor para crear una conexión a la BD utilizando una URL.
`Connection DriverManager.getConnection(String url, Properties info)`	Constructor para crear una conexión a la BD utilizando un objeto Properties.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/DriverManager.html

Ejemplo: uso del método `getConnection()` para conectarse a una *BD MariaDB*.

El **método `getConnection()`** de la clase `DriverManager` se utiliza para establecer una conexión con una *BD*. Existen varias sobrecargas de este método, una de las cuales **acepta un objeto de tipo `Properties`**. Este objeto `Properties` es **útil para proporcionar parámetros adicionales** (como el nombre de usuario, la contraseña u otras propiedades de configuración) al momento de crear la conexión.

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

public class EjemploConexionJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost:3306/mydb";
        String usuario = "mydb", clave = "password";

        try (Connection conexion = DriverManager.getConnection(url, usuario, clave)) {
            // Realizar operaciones con la conexión a la BD
        } catch (SQLException e) {
            System.out.println("Error al conectar con la BD: " + e.getMessage());
        }
    }
}
```

Ejemplo: clase estática que permite obtener la conexión a una *BD* y cerrarla.

```
import java.sql.Connection;
```

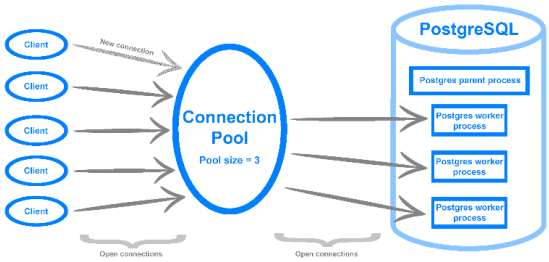
```java
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {
    private static final String URL = "jdbc:mariadb://localhost:3306/mydb";
    private static final String USER = "mydb";
    private static final String PASSWORD = "password";

    // Constructor privado para evitar instancias
    private ConnectionManager() { }

    public static Connection getConnection() throws SQLException { // Obtener conexión a la BD
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
    public static void closeConnection(Connection connection) throws SQLException { // Cerrar conexión
        if (connection != null)
            connection.close();
    }
}
```

Nota: **no implementar un patrón *Singleton* para manejar conexiones a la *BD* en una aplicación que gestiona transacciones** (se tratarán en el punto 10 de la unidad). Dado que el patrón *Singleton* proporciona una única instancia de conexión compartida, múltiples transacciones concurrentes pueden mezclarse, llevando a inconsistencias y datos corruptos, ya que una transacción podría no completarse antes de que otra comience a usar la misma conexión. Además, si una transacción falla y se realiza un `rollback`, todos los cambios realizados en la conexión se revertirían, afectando a otras transacciones que podrían estar en proceso.

Ejemplo: programa que hace uso de la clase estática del ejemplo anterior para obtener una conexión y cerrarla.

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

public class EjemploConexionJDBC2 {
    public static void main(String[] args) {
        Connection conexion = null;

        try {
            conexion = ConnectionManager.getConnection(); // Obtener la conexión

            // Realizar operaciones con la conexión a la BD
        } catch (SQLException e) {
            System.out.println("Error al conectar con la BD: " + e.getMessage());
        } finally {
            try {
                ConnectionManager.closeConnection(conexion); // Cerrar la conexión
            } catch (SQLException e) {
                System.out.println("Error al desconectar de la BD: " + e.getMessage());
            }
        }
    }
}
```

5.5 `DataSource`. Pooling de conexiones.



En *Java*, la **interfaz `DataSource`** (definida en el paquete `javax.sql`) es una **alternativa al uso directo de `DriverManager` para obtener conexiones a *BD***. Es más flexible y ofrece características adicionales, como el **manejo de conexiones en entornos de producción, donde se necesita un pool de conexiones** (*Connection Pooling*).

El *pooling* de conexiones es una **técnica que ayuda a gestionar conexiones a la *BD* de manera eficiente**. En lugar de abrir y cerrar una conexión cada vez que se necesita interactuar con la *BD*, el ***pooling* mantiene un grupo** (*pool*) **de conexiones disponibles para ser reutilizadas**.

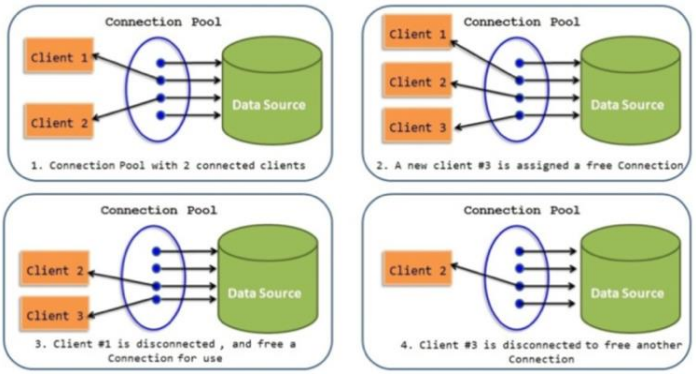
Se necesita una implementación de `DataSource`, que es una interfaz proporcionada por *JDBC*. Muchas bibliotecas y servidores de aplicaciones ofrecen **implementaciones** de `DataSource` que manejan el *pooling* de conexiones. Ejemplos: ***Apache Commons DBCP*, *HikariCP*, *AgroalDatasource* y *C3P0***.

Descripción del **proceso** de trabajo con el *pool* de conexiones:

- ✓ **Inicialización del *pool***: al iniciar la aplicación, se crean varias conexiones y se agregan al *pool*.
- ✓ **Solicitud de conexión**: cuando la aplicación necesita realizar una operación en la *BD*, solicita una conexión al *pool*.

- ✓ **Uso de la conexión**: la aplicación utiliza la conexión para realizar las operaciones necesarias en la *BD*.
- ✓ **Liberación de la conexión**: en lugar de cerrar la conexión, la aplicación la devuelve al *pool* para reutilizarla.

Un ***pool* de conexiones no genera problemas con las transacciones** (se tratarán en el punto 10 de la unidad) **siempre que las conexiones se gestionen adecuadamente**. Esto significa que, al finalizar una transacción, las **conexiones deben ser devueltas al pool en un estado limpio y restaurado**. Es esencial **restablecer configuraciones como el modo de `autocommit` antes de devolver la conexión**, lo que asegura que no haya efectos secundarios en futuras transacciones realizadas con la misma conexión.



5.5.1 Apache Commons DBCP.

La **biblioteca *Apache Commons DBCP*** (*Database Connection Pooling*) **es una implementación de *pooling* de conexiones para *Java***. Se puede descargar desde https://commons.apache.org/proper/commons-dbcp/download_dbcp.cgi.

Con ***Maven***, agregar la siguiente **dependencia** en el archivo `pom.xml` dentro de la sección `<dependencies>`:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.11.0</version>
</dependency>
```

Con ***Gradle***, agregar la siguiente **dependencia** en el archivo `build.gradle` dentro de la sección `dependencies`:

```
implementation 'org.apache.commons:commons-dbcp2:2.11.0'
```

Se puede tener una **clase estática para el *pool* de conexiones**, lo cual puede ser útil si se desea acceder al *pool* de conexiones desde varias clases sin necesidad de instanciarla cada vez. Ejemplo de cómo se podría implementar:

```
import java.sql.Connection;
import java.sql.SQLException;
import org.apache.commons.dbcp2.BasicDataSource;
public class ConnectionPoolManager {
    private static final BasicDataSource dataSource;
    static {
        dataSource = new BasicDataSource(); // Configuración del pool de conexiones
        dataSource.setDriverClassName("org.mariadb.jdbc.Driver");
        dataSource.setUrl("jdbc:mariadb://localhost/mydb");
        dataSource.setUsername("mydb");
        dataSource.setPassword("password");

        // Configuración adicional del pool
        dataSource.setInitialSize(5); // Número inicial de conexiones en el pool
        dataSource.setMaxTotal(10); // Número máximo de conexiones en el pool
    }
    // Constructor privado para evitar instancias
    private ConnectionPoolManager() { }
    // Obtener una conexión del pool
    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
    // Devolver conexión al pool
    public static void closeConnection(Connection connection) throws SQLException {
        if (connection != null)
            connection.close();
    }
    // Proporciona información sobre el estado actual del pool de conexiones.
    public static String getPoolStatus() {
        return String.format("Conexiones activas: %d, Inactivas: %d, Totales: %d ",
                dataSource.getNumActive(),      // Número de conexiones activas
                dataSource.getNumIdle(),        // Número de conexiones inactivas
                dataSource.getMaxTotal());      // Número total de conexiones
    }
}
```

`ConnectionPoolManager` es una clase estática que encapsula la lógica del *pool* de conexiones. El bloque `static`

se ejecuta cuando la clase se carga, inicializando el *pool* de conexiones. Luego, las **otras clases pueden obtener conexiones llamando a `ConnectionPoolManager.getConnection()` y cerrar conexiones llamando a `ConnectionPoolManager.closeConnection(connection)`**. El método `closeConnection()` cierra la conexión individual y la devuelve al *pool* para que pueda ser reutilizada, no disminuye el número total de conexiones disponibles.

5.5.2 HikariCP.

HikariCP es una **biblioteca popular de *Java* que proporciona una implementación de pooling de conexiones altamente eficiente y rápida**. Al igual que *commons-dbcp2*, ayuda a gestionar conexiones a *BD* de manera eficiente, mejorando el rendimiento de aplicaciones que hacen un uso intensivo de estas conexiones.

Con ***Maven***, agregar la siguiente **dependencia** en el archivo `pom.xml` dentro de la sección `<dependencies>`:

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>6.0.0</version>
</dependency>
```

Con ***Gradle***, agregar la siguiente **dependencia** en el archivo `build.gradle` dentro de la sección `dependencies`:

```
implementation 'com.zaxxer:HikariCP:6.0.0'
```

Ejemplo de **implementación de un pool de conexiones utilizando *HikariCP* con una clase estática**, similar al ejemplo proporcionado anteriormente con *commons-dbcp2*:

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
import java.sql.Connection;
import java.sql.SQLException;

public class HikariCPConnectionPool {
    private static final HikariDataSource dataSource;

    static {
        HikariConfig config = new HikariConfig(); // Configuración del pool de conexiones
        config.setJdbcUrl("jdbc:mariadb://localhost/mydb");
        config.setUsername("mydb");
        config.setPassword("password");
        config.setDriverClassName("org.mariadb.jdbc.Driver");
        // Configuración adicional del pool
        config.setMinimumIdle(5);  // Número mínimo de conexiones inactivas
        config.setMaximumPoolSize(10); // Número máximo de conexiones en el pool
        config.setIdleTimeout(30000); // Tiempo máximo de inactividad de una conexión antes de ser eliminada
        dataSource = new HikariDataSource(config);
    }

    // Constructor privado para evitar instancias
    private HikariCPConnectionPool() { }

    // Obtener una conexión del pool
    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }

    // Devolver conexión al pool
    public static void closeConnection(Connection connection) throws SQLException {
        if (connection != null)
            connection.close();  // Las conexiones se devuelven automáticamente al pool
    }

    // Proporciona el estado actual del pool
    public static String getPoolStatus() {
        return String.format("Conexiones activas: %d, Inactivas: %d, Totales: %d",
                dataSource.getHikariPoolMXBean().getActiveConnections(),
                dataSource.getHikariPoolMXBean().getIdleConnections(),
                dataSource.getHikariPoolMXBean().getTotalConnections());
    }
}
```

HikariCP es una excelente opción cuando se necesita una implementación de pooling de conexiones rápida y eficiente. Su simplicidad de configuración, junto con su rendimiento superior, lo ha convertido en una de las bibliotecas más utilizadas para manejo de conexiones en Java.

Ejemplo: múltiples conexiones y visualización del estado del pool utilizando la clase `HikariCPConnectionPool`.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class EjemploUsoHikariCPConnectionPool {
    public static void main(String[] args) {
        Connection connection1 = null, connection2 = null, connection3 = null;

        try {
            // Mostrar el estado del pool antes de obtener varias conexiones
            System.out.println("Estado del pool antes de establecer tres conexiones:");
            System.out.println(HikariCPConnectionPool.getPoolStatus());

            // Establecer varias conexiones del pool
            connection1 = HikariCPConnectionPool.getConnection();
            connection2 = HikariCPConnectionPool.getConnection();
            connection3 = HikariCPConnectionPool.getConnection();

            // Mostrar el estado del pool después de obtener varias conexiones
            System.out.println("Estado del pool tras establecer tres conexiones:");
            System.out.println(HikariCPConnectionPool.getPoolStatus());

            // Ejecutar una consulta simple con la primera conexión
            ejecutarConsulta(connection1);

            // Cerrar la conexión 1 y devolverla al pool
            HikariCPConnectionPool.closeConnection(connection1);

            // Mostrar el estado del pool tras ejecutar una consulta
            System.out.println("Estado del pool tras ejecutar la consulta en la primera conexión:");
            System.out.println(HikariCPConnectionPool.getPoolStatus());
        } catch (SQLException e) {
            System.err.println("Error al interactuar con la base de datos: " + e.getMessage());
        } finally {
            try { // Cerrar las conexiones 2 y 3, y devolverlas al pool
                HikariCPConnectionPool.closeConnection(connection2);
                HikariCPConnectionPool.closeConnection(connection3);

                // Mostrar el estado del pool después de cerrar todas las conexiones
                System.out.println("Estado del pool tras cerrar las conexiones:");
                System.out.println(HikariCPConnectionPool.getPoolStatus());
            } catch (SQLException e) {
                System.err.println("Error al cerrar las conexiones: " + e.getMessage());
            }
        }
    }
    // Método auxiliar para ejecutar una consulta con una conexión
    private static void ejecutarConsulta(Connection connection) throws SQLException {
        String query = """
            SELECT e.id_empleado, e.nombre AS empleado, e.oficio, e.salario, d.nombre AS departamento, d.localidad
            FROM empleados e
            JOIN departamentos d ON e.id_departamento = d.id_departamento
        """;
        try (PreparedStatement stmt = connection.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) { // Procesar los resultados
                int idEmpleado = rs.getInt("id_empleado");
                String nombreEmpleado = rs.getString("empleado");
                String oficio = rs.getString("oficio");
                double salario = rs.getDouble("salario");
                String nombreDepartamento = rs.getString("departamento");
                String localidad = rs.getString("localidad");

                System.out.printf("Empleado: %d, Nombre: %s, Oficio: %s, Salario: %.2f, Departamento: %s,
Localidad: %s%n",
                        idEmpleado, nombreEmpleado, oficio, salario, nombreDepartamento, localidad);
            }
        }
    }
}
```

5.6 Interfaz **Connection**.

La interfaz `Connection` **define un conjunto de métodos que deben ser implementados por las clases concretas que representan una conexión a una *BD***. Las clases concretas proporcionan la implementación real de estos métodos para trabajar con un *SGBD* específico, como *MySQL*, *Oracle*, *PostgreSQL*, etc.

La interfaz `Connection` se encuentra en el **paquete `java.sql`** y define **métodos para establecer, administrar y cerrar conexiones a *BD*, así como para crear objetos relacionados con la ejecución de consultas y transacciones**.

Tabla con algunos de los **métodos más utilizados** de la **clase `Connection`**:

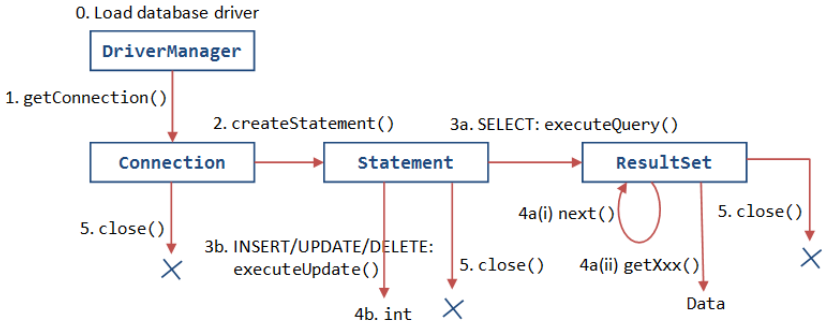
Método	Descripción
`Statement createStatement()`	Crea un objeto `Statement` para ejecutar consultas SQL simples.
`PreparedStatement prepareStatement(String sql)`	Crea un objeto `PreparedStatement` para ejecutar consultas parametrizadas.
`CallableStatement prepareCall(String sql)`	Crea un objeto `CallableStatement` para ejecutar procedimientos almacenados.
`void setAutoCommit(boolean autoCommit)`	Establece si la conexión debe utilizar el modo de autocommit o no.
`void commit()`	Confirma la transacción actual en la BD.
`void rollback()`	Revierte la transacción actual en la BD.
`Savepoint setSavepoint()`	Crea un punto de guardado para usar en transacciones.
`void releaseSavepoint(Savepoint savepoint)`	Libera un punto de guardado previamente creado.
`void setReadOnly(boolean readOnly)`	Establece si la conexión debe ser de solo lectura o no.
`boolean isReadOnly()`	Verifica si la conexión es de solo lectura.
`DatabaseMetaData getMetaData()`	Obtiene información sobre la BD a la que está conectada la conexión.
`void close()`	Cierra la conexión a la BD.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Connection.html

Ejemplo: uso de la interfaz `Connection` para establecer una conexión a una *BD MariaDB* e insertar un registro.

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

public class EjemploConexionJDBC3 {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb", usuario = "mydb", clave = "password";

        try (Connection conexion = DriverManager.getConnection(url, usuario, clave);
             Statement statement = conexion.createStatement()) {
            // Sentencia SQL para insertar un nuevo departamento
            String consulta = "INSERT INTO departamentos (id_departamento, nombre, localidad)"
                    + " VALUES (5, 'Compras', 'Granada')";

            statement.executeUpdate(consulta); // Ejecutar la sentencia

            System.out.println("La inserción se realizó correctamente.");
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

5.7 Interfaz `Statement`.



La interfaz `Statement` **se utiliza para ejecutar sentencias estáticas *SQL*** (sin parámetros), principalmente **consultas *SQL* simples** y es una de las **formas más básicas de interactuar con un *SGBD*** desde una aplicación *Java*.

La interfaz `Statement` se encuentra en el **paquete `java.sql`** y **no se pueden crear objetos directamente** de ella, **se hace usando un objeto de la interfaz `Connection`**:

```
Statement stm = conexion.createStatement();
```

Los **métodos más usados** son:

- ✓ **`ResultSet executeQuery(String)`**: ejecuta una consulta y **devuelve** el resultado de esta mediante un **objeto de tipo `ResultSet`** (sentencias **`SELECT`**).
- ✓ **`int executeUpdate(String)`**: sentencias *SQL* que **no devuelven un `ResultSet`** (*DML*: **`INSERT`**, **`UPDATE`** y

`DELETE`; *DDL*: **`CREATE`**, **`DROP`** y **`ALTER`**) y **generan cambios en la *BD***. Con **sentencias del *DML* devuelve un `int` que indica el número de filas afectadas y con sentencias del *DDL* devuelve 0** (esto se debe a que las sentencias *DDL* no afectan directamente a las filas de las tablas, sino que cambian la estructura de la *BD*).

- ✓ **`boolean execute(String)`**: para **cualquier sentencia *SQL***. **Devuelve `true` si la sentencia devuelve un `ResultSet`** (para obtener el `ResultSet` usar `stm.getResultSet()`) **y `false` si no devuelve resultados** (para obtener el número de filas afectadas usar `stm.getUpdateCount()`).

La ejecución de **estos métodos puede lanzar excepciones** de tipo **`SQLException` o `SQLTimeoutException`**.

Nota: tener cuidado al elegir la clase que se va a importar, ya que **existen varias clases con idéntico nombre, pero en distintos paquetes**. Se han de **elegir siempre las clases del paquete `java.sql`**.

La interfaz `Statement` **es propensa a ataques de inyección *SQL***, se debe **utilizar `PreparedStatement` o `CallableStatement` cuando se trabaja con sentencias parametrizadas para evitar la inyección *SQL***.

Tabla con algunos de los **métodos más utilizados** de la **clase `Statement`**:

Método	Descripción
`ResultSet executeQuery(String sql)`	Ejecuta una consulta SQL que devuelve un conjunto de resultados. Devuelve un objeto ResultSet.
`int executeUpdate(String sql)`	Ejecuta una sentencia SQL que realiza una actualización, eliminación o inserción en la BD y devuelve el número de filas afectadas.
`boolean execute(String sql)`	Ejecuta cualquier tipo de sentencia SQL y devuelve true si devuelve un conjunto de resultados (SELECT) o false si no lo devuelve.
`ResultSet getResultSet()`	Devuelve el conjunto de resultados (ResultSet) de la última consulta que devolvió un conjunto de resultados (SELECT).
`int getResultSetType()`	Devuelve el tipo de conjunto de resultados de la última sentencia.
`int getResultSetConcurrency()`	Devuelve el nivel de concurrencia del conjunto de resultados de la última sentencia.
`int getUpdateCount()`	Devuelve el número de filas afectadas por la última sentencia de actualización (INSERT, UPDATE, DELETE).
`void addBatch(String sql)`	Agrega una sentencia SQL a una lista para su posterior ejecución en conjunto.
`void clearBatch()`	Limpia todas las sentencias SQL agregadas a la lista.
`int[] executeBatch()`	Ejecuta todas las sentencias SQL en la lista y devuelve un arreglo de enteros que representa el número de filas afectadas por cada sentencia.
`void setFetchSize(int rows)`	Controla la cantidad de filas recuperadas en cada llamada de red a la BD.
`void setMaxRows(int max)`	Establece un límite total en la cantidad de filas que se devolverán en el conjunto de resultados (limita el número total de filas recuperadas).
`void setEscapeProcessing(boolean enable)`	Habilita o deshabilita el procesamiento de caracteres de escape en sentencias SQL.
`void close()`	Cierra el objeto `Statement` y libera los recursos asociados.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/Statement.html

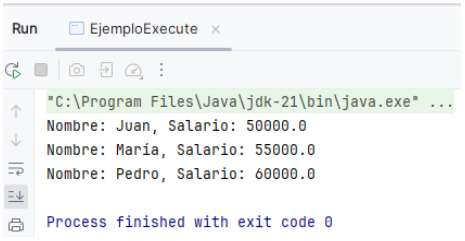
Ejemplo: uso de varios de los métodos de la interfaz `Statement` en diferentes situaciones.

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

public class EjemploStatementJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb";
        String usuario = "mydb", clave = "password";

        try (Connection con = DriverManager.getConnection(url, usuario, clave);
             Statement stm = con.createStatement()) {
            // Eliminar la tabla si ya existe para evitar errores
            String eliminarTabla = "DROP TABLE IF EXISTS empleados2";
            stm.executeUpdate(eliminarTabla);

            // Crear una tabla
            String tabla = "CREATE TABLE empleados2 (id INT PRIMARY KEY, nombre VARCHAR(255), salario DOUBLE)";
```

```
            stm.executeUpdate(tabla);

            // Insertar datos
            String insertar = "INSERT INTO empleados2(id, nombre, salario) VALUES (1, 'Juan', 50000)";
            stm.executeUpdate(insertar);

            // Consulta SELECT
            String consulta = "SELECT nombre, salario FROM empleados2 WHERE salario > 40000";
            ResultSet res = stm.executeQuery(consulta);

            // Procesar el resultado de la consulta
            while (res.next())
                System.out.println("Nombre: " + res.getString("nombre") + ", Salario: " + res.getDouble("salario"));

            // Cerrar ResultSet explícitamente para evitar posibles fugas de memoria
            res.close();

            // Ejecutar varias consultas en lote
            stm.addBatch("INSERT INTO empleados2 (id, nombre, salario) VALUES (2, 'María', 55000)");
            stm.addBatch("INSERT INTO empleados2 (id, nombre, salario) VALUES (3, 'Pedro', 60000)");
            int[] filasAfectadas = stm.executeBatch();

            // Informar cuántas filas se afectaron en total con el batch
            int totalFilasAfectadas = 0;
            for (int filas : filasAfectadas)
                totalFilasAfectadas += filas;
            System.out.println("Número total de filas afectadas por el batch: " + totalFilasAfectadas);
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

Nota: los **literales cadena en *Java* utilizan comillas dobles "**, por tanto, se dejan las **comillas simples ' para los literales usados en *SQL***. También es posible usar las comillas dobles mediante la secuencia de escape \".

Ejemplo: uso de `boolean execute(String)`.




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

public class EjemploExecute {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb";

        try (Connection conexion = DriverManager.getConnection(url, "mydb", "password");
             Statement statement = conexion.createStatement()) {
            String consultaSQL = "SELECT * FROM empleados2";

            // Ejecutar la sentencia SQL y verificar si devuelve un conjunto de resultados
            boolean tieneResultados = statement.execute(consultaSQL);
            if (tieneResultados) { // Si tiene resultados, procesarlos
                try (ResultSet res = statement.getResultSet()) {
                    while (res.next()) {
                        String nombre = res.getString("nombre");
                        double salario = res.getDouble("salario");
                        System.out.println("Nombre: " + nombre + ", Salario: " + salario);
                    }
                }
            } else {
                System.out.println("La sentencia no devuelve resultados.");
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

5.8 Clase/interfaz `ResultSet`.

Nota: en *JDBC*, **`ResultSet` es tanto una interfaz como una clase**. La interfaz `ResultSet` define un conjunto de métodos que representan diversas operaciones para trabajar con conjuntos de resultados de *BD*. Por otro lado, hay implementaciones específicas de esta interfaz proporcionadas por los controladores *JDBC*, y estas implementaciones son clases concretas que se pueden usar para interactuar con los resultados de una consulta *SQL*.

La clase `ResultSet` en *JDBC* **se utiliza para representar los resultados de una consulta *SQL* a una *BD* relacional**.

Un **objeto `ResultSet` actúa como una tabla virtual que contiene filas y columnas de datos, lo que permite a las aplicaciones *Java* recuperar y manipular los datos devueltos**.

Cuando se ejecuta una consulta (`SELECT`) se obtendrán los resultados encapsulados en un objeto de tipo `ResultSet`, que representa una tabla con los datos que genera la consulta.

La clase `ResultSet` se encuentra en el **paquete `java.sql`** y la importación se realiza de la siguiente forma:

```
import java.sql.ResultSet;
```

La clase `ResultSet` tiene una **forma de trabajar muy parecida a un iterador**, dispone de un **cursor que apunta en cada momento a una única fila** (fila activa). Se debe tener en cuenta que **únicamente se permite acceder a los datos de la fila activa**, por lo que **se tendrá que ir moviendo el cursor** de fila en fila.

En el momento en el que se crea el objeto `ResultSet` el cursor se encuentra en la primera fila. El **paso de una fila a la siguiente** se lleva a cabo mediante el **método `next()`**, que **devuelve `true` o `false` si ha sido posible realizar el movimiento**.

Para **extraer los datos de la fila activa se dispone de distintos métodos `get`** (`getString`, `getInt`, `getDouble`, `getDate`, …). Todos los **métodos para extraer los datos de una tabla están sobrecargados** para que, **además de poder especificar el nombre del campo, se pueda indicar su posición en la consulta** (`ResultSet` numera los **campos de la consulta comenzando en 1**).

La **estructura habitual para trabajar** con un `ResultSet` es:

```
ResultSet rs = sentencia.executeQuery(sql);
while (rs.next()) { // Extraer datos fila activa
    variable1 = rs.getString("nombre");
    variable2 = rs.getInt("num");
    ...
}
```

Al **salir del bucle `while` se tendrá la certeza de que se han recorrido todas las filas** obtenidas en la consulta.

Tabla con algunos de los **métodos más utilizados** de la **clase `ResultSet`**:

Tipo de dato SQL	Tipo de dato JDBC	Método ResultSet
CHAR(n)	String	getString(column)
VARCHAR(n)	String	getString(column)
TEXT	String	getString(column)
INT, INTEGER	int	getInt(column)
BIGINT	long	getLong(column)
SMALLINT	short	getShort(column)
TINYINT	byte	getByte(column)
FLOAT	float	getFloat(column)
DOUBLE, REAL	double	getDouble(column)
DECIMAL(p,s), NUMERIC	java.math.BigDecimal	getBigDecimal(column)
BOOLEAN, BIT	boolean	getBoolean(column)
DATE	java.sql.Date	getDate(column)
TIME	java.sql.Time	getTime(column)
TIMESTAMP	java.sql.Timestamp	getTimestamp(column)
BLOB	java.sql.Blob	getBlob(column)
CLOB	java.sql.Clob	getClob(column)
BINARY	byte[]	getBytes(column)
VARBINARY(n)	byte[]	getBytes(column)
ARRAY	java.sql.Array	getArray(column)
UUID	java.util.UUID	getObject(column, UUID.class)

Método	Descripción
`boolean next()`	Mueve el cursor al siguiente registro en el conjunto de resultados y devuelve `true` si hay más registros, o `false` si se ha llegado al final.
`boolean previous()`	Mueve el cursor al registro anterior en el conjunto de resultados.
`boolean first()`	Mueve el cursor al primer registro en el conjunto de resultados.
`boolean last()`	Mueve el cursor al último registro en el conjunto de resultados.
`boolean absolute(int row)`	Mueve el cursor a una fila específica en el conjunto de resultados, donde `row` es el número de fila.
`boolean getBoolean(String columnName)`	Devuelve el valor de una columna como un valor booleano.
`byte getByte(String columnName)`	Devuelve el valor de una columna como un byte.
`short getShort(String columnName)`	Devuelve el valor de una columna como un entero corto.
`int getInt(String columnName)`	Devuelve el valor de una columna como un entero.
`long getLong(String columnName)`	Devuelve el valor de una columna como un entero largo.
`Sting getString(String columnName)`	Devuelve el valor de una columna como una cadena.
`float getFloat(String columnName)`	Devuelve el valor de una columna como un número de punto flotante.
`double getDouble(String columnName)`	Devuelve el valor de una columna como un número de punto flotante de doble precisión.
`Date getDate(String columnName)`	Devuelve el valor de una columna como una fecha.
`Time getTime(String columnName)`	Devuelve el valor de una columna como una hora.
`Timestamp getTimestamp(String columnName)`	Devuelve el valor de una columna como Timestamp (java.sql.Timestamp).
`ResultSetMetaData getMetaData()`	Devuelve un objeto `ResultSetMetaData` que proporciona información sobre las columnas del conjunto de resultados, como nombres y tipos de datos.

`void updateInt(String columnName, int x)`	Actualiza el valor de una columna de tipo entero en el conjunto de resultados. Se debe llamar a updateRow() para actualizar la fila.
`void updateString(String columnName, String x)`	Actualiza el valor de una columna de tipo cadena en el conjunto de resultados. Se debe llamar a updateRow() para actualizar la fila.
`void updateRow()`	Actualiza el registro actual en el conjunto de resultados después de realizar cambios.
`void deleteRow()`	Elimina el registro actual en el conjunto de resultados.
`void moveToInsertRow()`	Mueve el cursor a una fila especial donde se puede insertar nuevos datos.
`void insertRow()`	Inserta la nueva fila con los valores asignados directamente en la BD.
`void close()`	Cierra el `ResultSet` y libera los recursos asociados.

Recordar que todos los **métodos get también disponen del mismo método en los que** en lugar de indicar el nombre de la columna **se puede indicar el número de la columna**. Ej.: `String getString(int numeroColumna)`

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSet.html

5.8.1 Tipos de `ResultSet`.

El objeto `ResultSet` que se ha utilizado hasta el momento puede denominarse por defecto, ya que es de solo lectura y su cursor siempre avanza hacia adelante. Es posible utilizar otros tipos de `ResultSet` que permiten la modificación de sus datos y poder mover el cursor hacia delante o atrás, así como posicionarlo en cualquier fila.

Para **obtener otros tipos de `ResultSet`** es necesario **crear los objetos `Statement`** o **`PreparedStatement`** (se verá en el punto 8) **mediante el método sobrecargado de `Connection`**:

```
Statement createStatement(int tipoResultSet, int concurrencia)
PreparedStatement prepareStatement(String sql, int tipoResultSet, int concurrencia)
```

En *JDBC*, **para especificar el tipo y la concurrencia de un `ResultSet`, se pueden utilizar constantes predefinidas** en la interfaz `ResultSet`. Algunas de las constantes **más comunes** que se utilizan al crear un `Statement` son:

- ✓ **Tipo de Resultset**:
  - **`ResultSet.TYPE_FORWARD_ONLY`**: `ResultSet` **de solo avance**, el cursor solo podrá moverse hacia delante.
  - **`ResultSet.TYPE_SCROLL_INSENSITIVE`**: `ResultSet` con **capacidad de desplazamiento hacia delante o atrás**, **y no sensible a cambios externos** (datos contenidos en el `ResultSet` son una copia de los datos almacenados en la *BD*).
  - **`ResultSet.TYPE_SCROLL_SENSITIVE`**: `ResultSet` con **capacidad de desplazamiento hacia delante o atrás, y sensible a cambios externos** (cualquier modificación de los datos que contiene el `ResultSet` en la *BD* produce una modificación en los datos del `ResultSet`).
- ✓ **Concurrencia** (indica la posibilidad de modificar los datos contenidos en el objeto `ResultSet`):
  - **`ResultSet.CONCUR_READ_ONLY`**: `ResultSet` de **solo lectura**.
  - **`ResultSet.CONCUR_UPDATABLE`**: `ResultSet` con **capacidad de actualización**.

No todos los drivers de *BD* y todas las implementaciones de *JDBC* admiten `ResultSet` actualizables. Además, tener en cuenta que **realizar actualizaciones directas en la *BD* a través de un `ResultSet` puede tener implicaciones de rendimiento y de seguridad**, por lo que se debe utilizar con precaución.

Al crear un `Statement`, se pueden combinar estas constantes según las necesidades. Ejemplo:

```
Statement stmt = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  ResultSet.CONCUR_UPDATABLE);
ResultSet resultSet = stmt.executeQuery("SELECT * FROM nombre_tabla");
```

En el ejemplo, se crea un `Statement` con un `ResultSet` que es de tipo `TYPE_SCROLL_INSENSITIVE` (puede desplazarse hacia adelante y hacia atrás, pero no es sensible a cambios externos) y con concurrencia `CONCUR_UPDATABLE` (se puede actualizar la *BD* a través del `ResultSet`).

Cuando se usa el **método `createStatement()` sin parámetros**, los objetos `ResultSet` obtenidos por defecto serán de **tipo `TYPE_FORWARD_ONLY` y `CONCUR_READ_ONLY`**.

5.8.2 Métodos para mover el cursor.

En *JDBC*, se pueden utilizar varios métodos para desplazar el cursor dentro de un `ResultSet` y moverse a través de

los resultados obtenidos de una consulta *SQL*. Algunos de los métodos **más comunes** son:

- ✓ **`boolean next()`**: mueve el cursor al **siguiente registro** en el conjunto de resultados.
- ✓ **`boolean previous()`**: mueve el cursor al **registro anterior** en el conjunto de resultados.
- ✓ **`boolean first()`**: mueve el cursor al **primer registro** en el conjunto de resultados.
- ✓ **`boolean last()`**: mueve el cursor al **último registro** en el conjunto de resultados.
- ✓ **`boolean absolute(int row)`**: mueve el cursor a una **posición específica** en el conjunto de resultados, donde **`row` es el número de fila** (positivo o negativo). Las **filas se numeran comenzando en 1**, pero existe otra forma de numerar las filas que consiste en utilizar números negativos, en este caso, la cuenta comienza por la última fila (la fila -1 es la última, la -2 la penúltima, etc.).
- ✓ **`boolean relative(int rows)`**: mueve el cursor una **cantidad relativa de filas hacia adelante o hacia atrás** (si `rows` es negativo) **desde la posición actual**. Si el parámetro `rows` es 0 no se moverá.
- ✓ **`void beforeFirst()`**: mueve el cursor **antes del primer registro** en el conjunto de resultados. No hace nada si el `ResultSet` está vacío.
- ✓ **`void afterLast()`**: mueve el cursor **después del último registro** en el conjunto de resultados. No hace nada si el `ResultSet` está vacío.

Los **métodos `previous()`, `first()`, `last()`, `absolute(int row)`, `relative(int rows)`, `beforeFirst()` y `afterLast()`** solo están **disponibles si se ha creado un `ResultSet` de tipo `TYPE_SCROLL_SENSITIVE` o `TYPE_SCROLL_INSENSITIVE`**.

Todos los **métodos que mueven el cursor, salvo `beforeFirst()` y `afterLast()`, devuelven un booleano que indica si ha sido posible desplazar el cursor** (`true`) o, por el contrario, el movimiento del cursor no puede realizarse (`false`).

5.8.3 Ubicación del cursor.

La ubicación del cursor en un `ResultSet` se refiere a la **posición actual del cursor dentro del conjunto de resultados** obtenidos de una consulta *SQL*. La **interfaz `ResultSet` proporciona métodos para obtener información sobre su ubicación**:

- ✓ **`boolean isBeforeFirst()`**: devuelve **`true`** si el **cursor está antes del primer registro** (posición inicial, cuando se crea el `ResultSet`).
- ✓ **`boolean isAfterLast()`**: devuelve **`true`** si el **cursor está después del último registro** (posición que se alcanza tras recorrer el `ResultSet`).
- ✓ **`boolean isFirst()`**: devuelve **`true`** si el **cursor está en la primera fila**.
- ✓ **`boolean isLast()`**: devuelve **`true`** si el **cursor está en la última fila**.
- ✓ **`int getRow()`**: devuelve el **número de la fila actual** en el conjunto de resultados.

Estos métodos permiten comprobar la ubicación del cursor en relación con el conjunto de resultados. **Se pueden usar estas comprobaciones para controlar el flujo de la aplicación según la posición del cursor**.

Ejemplo:

```
// Moverse a la fila especial para inserción
rs.moveToInsertRow();
// Asignar valores a las columnas
rs.updateInt("id", 5);  // Asumiendo que la columna 'id' es un entero
rs.updateString("nombre", "Nuevo Empleado");
rs.updateInt("edad", 30);
// Insertar la fila en la base de datos
rs.insertRow();
```

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

public class EjemploResultSetJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb", usuario = "mydb", clave = "password";
        try (Connection con = DriverManager.getConnection(url, usuario, clave);
             Statement stm = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
            String consulta = "SELECT id, nombre, salario FROM empleados2";
            ResultSet rs = stm.executeQuery(consulta);

            while (rs.next()) { // Moverse a través de los resultados
                int id = rs.getInt("id"); // Acceso por nombre o índice
                String nombre = rs.getString("nombre");
                double salario = rs.getDouble("salario");
                System.out.println("ID: " + id + ", Nombre: " + nombre + ", Salario: " + salario);
            }
            // Verificar si hay al menos 2 filas antes de intentar moverse a la segunda
            rs.last(); // Mover al final para contar las filas
```

```
            int numeroFilas = rs.getRow();
            if (numeroFilas >= 2) {
                rs.absolute(2); // Mover al segundo registro
                rs.updateDouble("salario", 75000.0); // Actualizar el salario de la segunda fila
                rs.updateRow(); // Aplicar la actualización
                System.out.println("Salario del segundo empleado actualizado a 75000.0.");
            } else
                System.out.println("No hay suficientes filas en la tabla para realizar la actualización.");
            rs.close(); // Cerrar el ResultSet
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

En este ejemplo, se ha configurado el `ResultSet` para ser actualizable utilizando `ResultSet.TYPE_SCROLL_SENSITIVE` y `ResultSet.CONCUR_UPDATABLE`.

```
rs.beforeFirst(); // Mover el cursor al inicio para recorrer el ResultSet
while (rs.next()) {
    if (rs.getInt("id") == 6) { // Si el ID coincide con 6
        rs.deleteRow();  // Eliminar la fila
        System.out.println("Fila eliminada.");
    }
}
```

TAREA

1. **¿Cómo se puede diseñar una aplicación que permita al usuario seleccionar qué *SGBD* utilizar? Describir los pasos clave para ofrecer al usuario la opción de elegir entre diferentes *SGBD* (por ejemplo, *MySQL*, *PostgreSQL*, o *MariaDB*), y explicar cómo configurar las conexiones de forma dinámica según la selección.**
2. **Crear un programa que se conecte a una *BD* y realice lo siguiente (utilizar tablas y datos ejemplo punto 5.2):**
   a) **Mostrar en pantalla todos los registros de la tabla "empleados".**
   b) **Mostrar la información de un empleado específico, dado su *ID*.**
   c) **Mostrar la información de los empleados cuyo nombre coincida total o parcialmente con el valor introducido.**
   d) **Insertar nuevos empleados, solicitando al usuario los datos necesarios por teclado.**
   e) **Actualizar el salario de un empleado, dado su *ID*, solicitando el nuevo salario por teclado.**
   f) **Eliminar un empleado, solicitando el *ID* por teclado.**
   g) **Mostrar la información de todos los empleados, incluyendo el departamento al que pertenecen.**

Notas:

- ✓ **Utilizar una clase estática que permita obtener y cerrar la conexión con la *BD*.**
- ✓ **Crear un menú que permita de forma fácil el acceso a cada una de las opciones.**

6 Sentencias de descripción de datos (*DDL*).

6.1 Obtener información sobre las *BD*. Interfaz `DatabaseMetaData`.

`DatabaseMetaData` es una **interfaz que proporciona métodos para obtener información sobre la *BD* a la que se está conectado** (sobre tablas, procedimientos almacenados, tipos de datos, etc.).

Es una interfaz que **no tiene constructores propios**, ya que **se obtiene a través de un objeto `Connection` utilizando el método `getMetaData()`**.

Tabla con algunos de los **métodos más utilizados** de la **interfaz `DatabaseMetaData`**:

Método	Descripción
`String getDatabaseProductName()`	Devuelve el nombre del producto de la BD, como "MySQL" u "Oracle".
`String getDatabaseProductVersion()`	Devuelve la versión del producto de la BD.
`String getDriverName()`	Devuelve el nombre del controlador JDBC que se está utilizando.
`String getDriverVersion()`	Devuelve la versión del controlador JDBC.
`String getURL()`	Devuelve la URL de conexión utilizada para conectarse a la BD.
`String getUserName()`	Devuelve el nombre de usuario utilizado para la conexión.

`ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)`	Devuelve información sobre las tablas de la BD, como nombres de tablas y tipos de tabla ("TABLE" o "VIEW").
`ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)`	Devuelve información sobre las columnas de una tabla específica, incluyendo nombres de columna y tipos de datos.
`ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern)`	Devuelve información sobre procedimientos almacenados en la BD.
`ResultSet getPrimaryKeys(String catalog, String schema, String table)`	Devuelve información sobre las claves primarias de una tabla.
`ResultSet getImportedKeys(String catalog, String schema, String table)`	Devuelve información sobre las claves foráneas importadas por una tabla (sobre las claves externas que apuntan a la tabla especificada).
`ResultSet getExportedKeys(String catalog, String schema, String table)`	Devuelve información sobre las claves foráneas exportadas por una tabla (sobre las claves externas definidas en la tabla especificada que apuntan a otras tablas).

6.1.1 Tablas.

El método `getTables()` **devuelve un objeto `ResultSet` con información sobre las tablas y vistas de la *BD***.

Sintaxis:

`ResultSet getTables(String catálogo, String esquema, String patrónDeTabla, String tipos[])`

, donde:

- ✓ **`catálogo`**: nombre de la *BD* (`null` indica todas las *BD*).
- ✓ **`esquema`**: patrón para el nombre del esquema (*BD* en *MariaDB*/*MySQL*). Se pueden utilizar caracteres comodines como, *%* para representar cualquier secuencia de caracteres. Si se desea obtener información de todas las *BD*, se puede establecer este valor en `null`.
- ✓ **`patrónDeTabla`**: patrón para el nombre de la tabla que se desea buscar. Se pueden utilizar caracteres comodines para buscar tablas específicas o patrones de nombres de tablas. Si se desea obtener información de todas las tablas, se puede establecer este valor en `null`. Ej. `De%` → todas las que empiecen por *De*.
- ✓ **`tipos[]`**: array de tipos de tablas que se desean buscar (`TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, `SYNONYM`, etc.). Se pueden proporcionar varios tipos en el arreglo para buscar múltiples tipos de tablas. Si se desea obtener información de todos los tipos de tablas, se puede proporcionar `null`. Ej: `String[] tipos = {"TABLE","SYNONYM"}; result = dbmd.getTables(null, null, null, tipos);`

Tener en cuenta que la disponibilidad y el comportamiento exacto de estos parámetros puede variar según el *SGBD* que se esté utilizando, ya que no todos los *SGBD* utilizan los catálogos, esquemas y tipos de tablas de la misma manera. Por lo tanto, es importante consultar la documentación específica del *SGBD* que se esté utilizando para comprender cómo se comportan estos parámetros en ese contexto particular.

Las **columnas que suelen estar disponibles** en el conjunto de resultados (`ResultSet`) devuelto son:

1. **`TABLE_CAT`**: nombre del catálogo o *BD* al que pertenece la tabla.
2. **`TABLE_SCHEM`**: nombre del esquema al que pertenece la tabla (o *BD* en *MariaDB*/*MySQL*).
3. **`TABLE_NAME`**: nombre de la tabla en la *BD*.
4. **`TABLE_TYPE`**: tipo de la tabla (`TABLE`, `VIEW`, `SYSTEM TABLE`, etc.).
5. **`REMARKS`**: comentarios o descripción de la tabla.
6. **`TYPE_CAT`**: catálogo que contiene el tipo de la tabla.
7. **`TYPE_SCHEM`**: esquema que contiene el tipo de la tabla.
8. **`TYPE_NAME`**: nombre del tipo de la tabla.
9. **`SELF_REFERENCING_COL_NAME`**: el nombre de la columna que se utiliza como clave principal para la autoreferencia en una tabla.
10. **`REF_GENERATION`**: indica cómo se generan las claves de referencia, como "`SYSTEM`" o "`USER`".

Se debe tener en cuenta que la disponibilidad exacta de estas columnas puede variar según el *SGBD* que se esté utilizando, ya que no todas los *SGBD* admiten los mismos atributos.

```
String catalogo = resul.getString(1);//columna 1
String esquema = resul.getString(2); //columna 2
String tabla = resul.getString(3);   //columna 3
String tipo = resul.getString(4);    //columna 4
```

```
String catalogo  = resul.getString("TABLE_CAT");   //columna 1
String esquema   = resul.getString("TABLE_SCHEM"); //columna 2
String tabla     = resul.getString("TABLE_NAME");  //columna 3
String tipo      = resul.getString("TABLE_TYPE");  //columna 4
```

Ejemplo: obtener información sobre la *BD* "*mydb*" y sus tablas.

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
public class EjemploDatabaseMetadata {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb", "mydb", "password")) {
            DatabaseMetaData dbmd = con.getMetaData(); // Crear objeto DatabaseMetaData
            // Obtener y mostrar información básica de la BD
            String nombre = dbmd.getDatabaseProductName();
            String driver = dbmd.getDriverName();
            String url = dbmd.getURL();
            String usuario = dbmd.getUserName();
            System.out.println("=======================================");
            System.out.println("=  INFORMACIÓN SOBRE LA BASE DE DATOS  =");
            System.out.println("=======================================");
            System.out.printf("Nombre del producto : %s %n", nombre);
            System.out.printf("Driver utilizado    : %s %n", driver);
            System.out.printf("URL de conexión     : %s %n", url);
            System.out.printf("Usuario conectado   : %s %n", usuario);
            // Obtener información de las tablas
            System.out.println("=======================================");
            System.out.println("=  TABLAS Y VISTAS EN LA BASE DE DATOS =");
            System.out.println("=======================================");
            // Obtener solo las tablas y vistas
            String[] tipos = {"TABLE", "VIEW"};
            try (ResultSet resul = dbmd.getTables("mydb", null, null, tipos)) {
                while (resul.next()) {
                    String catalogo = resul.getString("TABLE_CAT"); // getString(1)
                    String esquema = resul.getString("TABLE_SCHEM"); // getString(2)
                    String tabla = resul.getString("TABLE_NAME"); // getString(3)
                    String tipo = resul.getString("TABLE_TYPE"); // getString(4)
                    System.out.printf("Tipo: %s - Catálogo: %s, Esquema: %s, Tabla: %s %n",
                                                                    tipo, catalogo, esquema, tabla);
                }
            } catch (SQLException e) {
                System.out.println("Error al obtener las tablas: " + e.getMessage());
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.1.2 Columnas.

El método `getColumns()` **devuelve un objeto `ResultSet` con información sobre las columnas de una tabla**.

Sintaxis:

```
ResultSet getColumns(String catálogo, String esquema, String patrónDeTabla,
                                                        String patrónNombreDeColumna)
```

, donde:

- ✓ **`catálogo`**: nombre del catálogo o *BD* sobre el cual se desea obtener información de las columnas de una tabla.
- ✓ **`esquema`**: patrón para el nombre del esquema (*BD* en *MariaDB*/*MySQL*) que contiene la tabla.
- ✓ **`patrónDeTabla`**: patrón para el nombre de la tabla que contiene las columnas.
- ✓ **`patrónNombreDeColumna`**: patrón para el nombre de la columna que se desea buscar. Se pueden utilizar caracteres comodines para buscar columnas específicas o patrones de nombres de columnas. Ej. `d%` → nombres de las columnas que empiezan por `d`.

La **disponibilidad y el uso de estos parámetros pueden variar según el *SGBD* que se esté utilizando**. `null` en todos los parámetros → obtener información de todas las columnas y tablas de las *BD* a las que tiene acceso el usuario.

El conjunto de resultados (`ResultSet`) devuelto contendrá varias columnas que proporcionan detalles sobre cada columna de la tabla:

1. **`TABLE_CAT`**: nombre del catálogo (*BD*) al que pertenece la tabla que contiene la columna.
2. **`TABLE_SCHEM`**: nombre del esquema o *BD* al que pertenece la tabla que contiene la columna.
3. **`TABLE_NAME`**: nombre de la tabla que contiene la columna.

4. **COLUMN_NAME**: nombre de la columna en la tabla.
5. **DATA_TYPE**: código numérico que representa el tipo de dato de la columna, como un valor entero que corresponde a un tipo de dato específico (por ejemplo, 4 para `INTEGER`).
6. **TYPE_NAME**: nombre del tipo de dato de la columna, como una cadena descriptiva.
7. **COLUMN_SIZE**: tamaño máximo de la columna en caracteres o bytes, dependiendo del tipo de dato.
8. **BUFFER_LENGTH**: la longitud del búfer utilizado para almacenar el valor de la columna.
9. **DECIMAL_DIGITS**: número de dígitos decimales para columnas de tipo numérico, de lo contrario, 0.
10. **NUM_PREC_RADIX**: la base utilizada para representar la precisión numérica de la columna.
11. **NULLABLE**: indica si la columna puede contener valores nulos (0 para no, 1 para sí, 2 para desconocido).
12. **REMARKS**: comentarios o descripciones adicionales de la columna.
13. **COLUMN_DEF**: valor predeterminado de la columna, si se especifica.
14. **SQL_DATA_TYPE**: tipo de dato *SQL* de la columna.
15. **SQL_DATETIME_SUB**: subtipo de fecha y hora *SQL* para columnas de fecha y hora.
16. **CHAR_OCTET_LENGTH**: longitud en octetos de caracteres para columnas de tipo `CHARACTER`.
17. **ORDINAL_POSITION**: la posición ordinal de la columna en la tabla.
18. **IS_NULLABLE**: indica si la columna puede contener valores nulos (`YES` o `NO`).
19. **SCOPE_CATALOG**: el catálogo que contiene la tabla de alcance para columnas de alcance.
20. **SCOPE_SCHEMA**: el esquema que contiene la tabla de alcance para columnas de alcance.
21. **SCOPE_TABLE**: el nombre de la tabla de alcance para columnas de alcance.
22. **SOURCE_DATA_TYPE**: tipo de dato fuente de una columna `REF` o de tipo `STRUCT`.
23. **IS_AUTOINCREMENT**: indica si la columna es auto incrementable o no. El valor puede ser `YES`, `NO` o `NO INFORMATION`.
24. **IS_GENERATEDCOLUMN**: indica si la columna es una columna generada o calculada, es decir, si su valor se genera automáticamente por una expresión o una función en lugar de ser ingresado manualmente. El valor puede ser `YES`, `NO` o `NO INFORMATION`.

Ejemplo: obtener información sobre las columnas de la tabla "*departamentos*" de la *BD* "*mydb*".

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
public class EjemplogetColumns {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb","mydb","password")) {
            DatabaseMetaData dbmd = con.getMetaData(); // Crear objeto DatabaseMetaData
            System.out.println("==================================");
            System.out.println("=   COLUMNAS TABLA DEPARTAMENTOS  =");
            System.out.println("==================================");
            try (ResultSet columnas = dbmd.getColumns("mydb", null, "departamentos", null)) {
                while (columnas.next()) {
                    String nombCol = columnas.getString("COLUMN_NAME");
                    String tipoCol = columnas.getString("TYPE_NAME");
                    int tamCol = columnas.getInt("COLUMN_SIZE"); // Convertido a int para mejor manejo
                    String nula = columnas.getString("IS_NULLABLE");
                    System.out.printf("Columna: %s, Tipo: %s, Tamaño: %d, ¿Puede ser nula?: %s %n",
                                                                    nombCol, tipoCol, tamCol, nula);
                }
            } catch (SQLException e) {
                System.out.println("Error al obtener las columnas de la tabla 'departamentos': " + e.getMessage());
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.1.3 Claves primarias.

El método **`getPrimaryKeys()` se utiliza para obtener información sobre las columnas que forman parte de la clave primaria de una tabla en una *BD***. **Sintaxis**:

```
ResultSet getPrimaryKeys(String catálogo, String esquema, String tabla)
```

, donde:

- ✓ **`catálogo`**: nombre del catálogo (*BD*) al que pertenece la tabla. Se puede establecer este valor en `null` para

obtener información de todas las *BD*.

- ✓ **esquema**: nombre del esquema (o *BD* en *MariaDB*/*MySQL*) al que pertenece la tabla. Se puede establecer este valor en `null` para obtener información de todos los esquemas.
- ✓ **tabla**: nombre de la tabla para la cual se desea obtener las columnas de clave primaria.

La descripción de cada columna de la clave primaria tiene las **columnas**:

1. **TABLE_CAT**: nombre del catálogo (*BD*) al que pertenece la tabla que contiene la columna de clave primaria.
2. **TABLE_SCHEM**: nombre del esquema o *BD* al que pertenece la tabla que contiene la columna de clave primaria.
3. **TABLE_NAME**: nombre de la tabla que contiene la columna de clave primaria.
4. **COLUMN_NAME**: nombre de la columna que forma parte de la clave primaria.
5. **KEY_SEQ**: secuencia de la columna dentro de la clave primaria. Esto indica el orden de las columnas en la clave primaria si hay varias columnas en la clave.
6. **PK_NAME**: nombre de la clave primaria a la que pertenece la columna, si se ha asignado un nombre. Esto puede ser `null` si no se ha especificado un nombre.

Ejemplo: obtener la clave primaria de la tabla "*departamentos*" de la *BD* "*mydb*".

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

public class EjemplogetPrimaryKeys {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb", "mydb", "password");
             ResultSet pk = con.getMetaData().getPrimaryKeys("mydb", null, "departamentos")) {
            System.out.println("=====================================");
            System.out.println("= CLAVE PRIMARIA TABLA DEPARTAMENTOS =");
            System.out.println("=====================================");

            StringBuilder pkDep = new StringBuilder(); // Usar StringBuilder para mejorar la eficiencia
            String separador = "";

            while (pk.next()) {
                pkDep.append(separador).append(pk.getString("COLUMN_NAME")); // Concatenar nombres de columnas
                separador = " + "; // Separador para los nombres de columnas
            }
            System.out.println("Clave primaria: " + pkDep.toString());
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.1.4 Claves foráneas.

El método **`getExportedKeys()` se utiliza para obtener información sobre las claves externas (claves foráneas)** que hacen referencia a una tabla específica en una *BD*. Las claves foráneas son restricciones que mantienen la integridad referencial entre tablas, y estas restricciones especifican cómo las columnas de una tabla (claves foráneas) están relacionadas con las columnas de otra tabla (claves primarias). **Sintaxis**:

```
ResultSet getExportedKeys(String catálogo, String esquema, String tabla)
```

, donde:

- ✓ **catálogo**: el nombre del catálogo (*BD*) al que pertenece la tabla de referencia. Se puede establecer este valor en `null` para obtener información de todas las *BD*.
- ✓ **esquema**: el nombre del esquema al que pertenece la tabla de referencia. Se puede establecer este valor en `null` para obtener información de todos los esquemas.
- ✓ **tabla**: el nombre de la tabla de referencia para la cual se desea obtener las claves foráneas.

La descripción de cada columna de la clave foránea tiene las **columnas**:

1. **PKTABLE_CAT**: nombre del catálogo (*BD*) al que pertenece la tabla primaria (tabla referenciada).
2. **PKTABLE_SCHEM**: nombre del esquema al que pertenece la tabla primaria (tabla referenciada).
3. **PKTABLE_NAME**: nombre de la tabla primaria (tabla referenciada).
4. **PKCOLUMN_NAME**: nombre de la columna primaria (columna referenciada) en la tabla primaria.
5. **FKTABLE_CAT**: nombre del catálogo al que pertenece la tabla foránea (tabla de referencia).
6. **FKTABLE_SCHEM**: nombre del esquema al que pertenece la tabla foránea (tabla de referencia).
7. **FKTABLE_NAME**: nombre de la tabla foránea (tabla de referencia).

8. **FKCOLUMN_NAME**: nombre de la columna foránea (columna de referencia) en la tabla foránea.
9. **KEY_SEQ**: secuencia de la columna de clave foránea dentro de la restricción de clave foránea. Esto indica el orden de las columnas en la restricción de clave foránea si hay múltiples columnas en la clave.
10. **UPDATE_RULE**: regla de actualización para la clave foránea.
11. **DELETE_RULE**: regla de eliminación para la clave foránea.
12. **FK_NAME**: nombre de la restricción de clave foránea. Puede ser `null` si no se ha especificado un nombre.
13. **PK_NAME**: nombre de la restricción de clave primaria correspondiente. Puede ser `null` si no se ha especificado un nombre.
14. **DEFERRABILITY**: indica la capacidad de diferir (posponer) las restricciones de clave foránea en una *BD*.

Ejemplo: obtener las claves foráneas que referencian a la tabla "*departamentos*".

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
public class EjemplogetExportedKeys {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb", usuario = "mydb", clave = "password";
        try (Connection conexion = DriverManager.getConnection(url, usuario, clave)) {
            DatabaseMetaData dbmd = conexion.getMetaData(); // Crear objeto DatabaseMetaData
            System.out.println("================================================");
            System.out.println("= CLAVES AJENAS QUE REFERENCIAN A DEPARTAMENTOS =");
            System.out.println("================================================");
            try (ResultSet fk = dbmd.getExportedKeys("mydb", null, "departamentos")) {
                while (fk.next()) {
                    String fkName = fk.getString("FKCOLUMN_NAME");
                    String pkName = fk.getString("PKCOLUMN_NAME");
                    String pkTableName = fk.getString("PKTABLE_NAME");
                    String fkTableName = fk.getString("FKTABLE_NAME");
                    System.out.printf("Tabla PK: %s, Clave Primaria: %s %n", pkTableName, pkName);
                    System.out.printf("Tabla FK: %s, Clave Ajena: %s %n", fkTableName, fkName);
                }
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.1.5 Procedimientos almacenados.

El método **`getProcedures()` se utiliza para obtener información sobre los procedimientos almacenados en una *BD***. Los procedimientos almacenados son secuencias de instrucciones *SQL* que se almacenan en la *BD* y se pueden ejecutar de manera repetida a través de una llamada a ese procedimiento. **Sintaxis**:

```
ResultSet getProcedures(String catálogo, String esquema, String procedure)
```

, donde:

- ✓ **catálogo**: el nombre del catálogo (*BD*) al que pertenecen los procedimientos. Se puede establecer este valor en `null` para obtener información de todos los catálogos.
- ✓ **esquema**: un patrón para el nombre del esquema que contiene los procedimientos. Se puede establecer este valor en `null` para obtener información de todos los esquemas.
- ✓ **procedure**: un patrón para el nombre de los procedimientos. Se puede establecer este valor en `null` para obtener información de todos los procedimientos.

Cada descripción de procedimiento tiene las siguientes **columnas**:

1. **PROCEDURE_CAT**: nombre del catálogo (*BD*) al que pertenece el procedimiento.
2. **PROCEDURE_SCHEM**: nombre del esquema al que pertenece el procedimiento.
3. **PROCEDURE_NAME**: nombre del procedimiento almacenado.
4. **NUM_INPUT_PARAMS**: el número de parámetros de entrada (`IN`) del procedimiento.
5. **NUM_OUTPUT_PARAMS**: el número de parámetros de salida (`OUT`) del procedimiento.
6. **NUM_RESULT_SETS**: el número de conjuntos de resultados devueltos por el procedimiento.
7. **REMARKS**: comentarios o descripciones adicionales del procedimiento.
8. **PROCEDURE_TYPE**: indica el tipo del procedimiento. Puede ser uno de los siguientes valores:
   - **`DatabaseMetaData.procedureNoResult`**: el procedimiento no devuelve un conjunto de resultados.

- **`DatabaseMetaData.procedureReturnsResult`**: el procedimiento devuelve uno o más conjuntos de resultados.
- **`DatabaseMetaData.procedureResultUnknown`**: no se sabe si el procedimiento devuelve conjuntos de resultados.

9. **`SPECIFIC_NAME`**: nombre específico del procedimiento en la *BD*. Este es un nombre único que identifica de manera única el procedimiento en la *BD*.

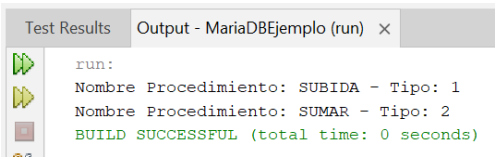
Ejemplo: obtener los procedimientos almacenados de la *BD* "*mydb*".

Nota: antes de poder probar este programa hay que crear varios procedimientos almacenados en la *BD* "*mydb*". Para ello ejecutar desde *phpMyAdmin* o *HeidiSQL* el siguiente código *SQL* (ver apuntes de primero de *Bases de Datos* para recordar):

```
DELIMITER //
CREATE FUNCTION sumar(n1 INT, n2 INT) RETURNS INT
BEGIN
	RETURN n1 + n2;
END;
//
CREATE PROCEDURE subida()
BEGIN
	UPDATE empleados SET salario = salario + 100 WHERE id_departamento = 3;
END;
//
DELIMITER ;
-- Para probarlos: SELECT sumar(2,22); CALL subida();
```

Programa *Java*:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
public class VerProcedimientosYFunciones {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb","mydb","password")) {
            DatabaseMetaData dbmd = con.getMetaData(); // Crear objeto DatabaseMetaData
            try (ResultSet proc = dbmd.getProcedures("mydb", null, null)) {
                boolean hayProcedimientos = false; // Variable para controlar si hay procedimientos
                while (proc.next()) {
                    hayProcedimientos = true; // Cambiar a verdadero si hay al menos un procedimiento
                    String proc_name = proc.getString("PROCEDURE_NAME");
                    String proc_type = proc.getString("PROCEDURE_TYPE");
                    System.out.printf("Nombre Procedimiento: %s - Tipo: %s %n", proc_name, proc_type);
                }
                if (!hayProcedimientos) // Mensaje si no se encontraron procedimientos
                    System.out.println("No se encontraron procedimientos en la base de datos.");
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```



Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/DatabaseMetaData.html

Ejemplo: información general sobre todas las *BD* que se encuentran en el servidor, listando las tablas y las columnas de cada tabla, y también se obtienen detalles sobre las claves primarias y foráneas.

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
public class EjemploDatabaseMetaData2 {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/", usuario = "root", clave = "root";
        try (Connection conexion = DriverManager.getConnection(url, usuario, clave)) {
            DatabaseMetaData metaData = conexion.getMetaData();
            // Información general sobre el SGBD
            System.out.println("Nombre del producto.....: " + metaData.getDatabaseProductName());
```

```
            System.out.println("Versión del producto.....: " + metaData.getDatabaseProductVersion());
            System.out.println("Nombre del controlador JDBC....: " + metaData.getDriverName());
            System.out.println("Versión del controlador JDBC...: " + metaData.getDriverVersion());
            System.out.println("Usuario.....: " + metaData.getUserName());

            // Listar tablas en la BD
            try (ResultSet tablas = metaData.getTables(null, null, null, new String[] { "TABLE" })) {
                System.out.println("\nTablas en las BD del servidor:");
                if (!tablas.next()) {
                    System.out.println("No se encontraron tablas en la base de datos.");
                } else {
                    do {
                        String nombreTabla = tablas.getString("TABLE_NAME");
                        System.out.println("\nTabla: " + nombreTabla);
                        System.out.println("*********************************************************************");

                        // Listar columnas de la tabla
                        System.out.println("Columnas de la tabla " + nombreTabla + ":");
                        try (ResultSet columnas = metaData.getColumns(null, null, nombreTabla, null)) {
                            while (columnas.next()) {
                                String nombreColumna = columnas.getString("COLUMN_NAME");
                                String tipoDato = columnas.getString("TYPE_NAME");
                                System.out.println("  " + nombreColumna + " : " + tipoDato);
                            }
                        }

                        // Obtener claves primarias
                        System.out.println("Columnas que conforman la clave primaria de la tabla " + nombreTabla + ":");
                        try (ResultSet clavesPrimarias = metaData.getPrimaryKeys(null, null, nombreTabla)) {
                            while (clavesPrimarias.next()) {
                                String nombreClavePrimaria = clavesPrimarias.getString("COLUMN_NAME");
                                System.out.println("  --> " + nombreClavePrimaria);
                            }
                        }

                        // Obtener claves foráneas
                        System.out.println("Claves foráneas referenciando la tabla " + nombreTabla + ":");
                        try (ResultSet clavesForaneas = metaData.getImportedKeys(null, null, nombreTabla)) {
                            while (clavesForaneas.next()) {
                                String nomTabFor = clavesForaneas.getString("PKTABLE_NAME");
                                String nomColFor = clavesForaneas.getString("FKCOLUMN_NAME");
                                System.out.println("  Desde la tabla " + nomTabFor + ", columna: " + nomColFor);
                            }
                        }
                    } while (tablas.next());
                }
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.2 Obtener información sobre los metadatos de un `ResultSet`.

La **clase `ResultSetMetaData` en *JDBC* se utiliza para obtener información sobre los metadatos de un conjunto de resultados (`ResultSet`)**. Proporciona **detalles sobre las columnas del conjunto de resultados, como nombres, tipos de datos y propiedades de cada columna**. Esto es útil cuando se necesita conocer la estructura de los datos recuperados de una consulta *SQL* sin necesidad de acceder a los datos en sí.

La clase `ResultSetMetaData` está contenida en el **paquete `java.sql`**. La importación se realiza de la siguiente manera:

```
import java.sql.ResultSetMetaData;
```

Tabla con algunos de los **constructores y métodos más utilizados** de la **clase `ResultSetMetaData`**:

Constructor/Método	Descripción
`ResultSetMetaData()`	Constructor predeterminado para crear un objeto `ResultSetMetaData`. No se utiliza directamente en aplicaciones, ya que se obtiene a través de un objeto `ResultSet`.
`int getColumnCount()`	Devuelve el número de columnas en el conjunto de resultados.
`String getColumnName(int column)`	Devuelve el nombre de la columna en la posición especificada.
`String getColumnLabel(int column)`	Devuelve la etiqueta de la columna en la posición especificada.
`String getColumnTypeName(int column)`	Devuelve el nombre del tipo de datos de la columna en la posición especificada.
`int getColumnType(int column)`	Devuelve el código del tipo de datos de la columna en la posición especificada.

`int getColumnDisplaySize(int column)`	Devuelve el tamaño de visualización de la columna en la posición especificada.
`int isNullable(int column)`	Devuelve un valor que indica si la columna en la posición especificada permite valores nulos.
`int getPrecision(int column)`	Devuelve la precisión de la columna en la posición especificada.
`int getScale(int column)`	Devuelve la escala de la columna en la posición especificada.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/ResultSetMetaData.html

Ejemplo:

```
Número de columnas: 3
Información de las columnas:
Columna 1:
   Nombre: id
   Etiqueta: id
   Tipo: INTEGER
   Código de Tipo: 4
   Tamaño de Visualización: 11
   Permite Nulos: false
   Precisión: 10
   Escala: 0

Columna 2:
   Nombre: nombre
   Etiqueta: nombre
   Tipo: VARCHAR
   Código de Tipo: 12
   Tamaño de Visualización: 255
   Permite Nulos: true
   Precisión: 255
   Escala: 0

Columna 3:
   Nombre: salario
   Etiqueta: salario
   Tipo: DOUBLE
   Código de Tipo: 8
   Tamaño de Visualización: 22
   Permite Nulos: true
   Precisión: 22
   Escala: 31
```

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

public class EjemploResultSetMetaData {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb";

        try (Connection conexion = DriverManager.getConnection(url, "mydb", "password");
                Statement statement = conexion.createStatement()) {
            String consulta = "SELECT id, nombre, salario FROM empleados2";

            try (ResultSet resultSet = statement.executeQuery(consulta)) {
                if (!resultSet.isBeforeFirst()) { // Comprobar si hay resultados
                    System.out.println("No se encontraron registros en la consulta.");
                } else {
                    ResultSetMetaData metaData = resultSet.getMetaData();
                    int columnCount = metaData.getColumnCount();
                    System.out.println("Número de columnas: " + columnCount);
                    System.out.println("Información de las columnas:");
                    for (int i = 1; i <= columnCount; i++) {
                        String nombreColumna = metaData.getColumnName(i);
                        String etiquetaColumna = metaData.getColumnLabel(i);
                        String tipoColumna = metaData.getColumnTypeName(i);
                        int tipoCodigo = metaData.getColumnType(i);
                        int tamanoVisualizacion = metaData.getColumnDisplaySize(i);
                        boolean permiteNulos = (metaData.isNullable(i) == ResultSetMetaData.columnNullable);
                        int precision = metaData.getPrecision(i);
                        int escala = metaData.getScale(i);

                        System.out.println("Columna " + i + ":");
                        System.out.println("   Nombre: " + nombreColumna);
                        System.out.println("   Etiqueta: " + etiquetaColumna);
                        System.out.println("   Tipo: " + tipoColumna);
                        System.out.println("   Código de Tipo: " + tipoCodigo);
                        System.out.println("   Tamaño de Visualización: " + tamanoVisualizacion);
                        System.out.println("   Permite Nulos: " + permiteNulos);
                        System.out.println("   Precisión: " + precision);
                        System.out.println("   Escala: " + escala);
                        System.out.println();  // Línea en blanco para mayor claridad
                    }
                }
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

6.3 Ejecución de sentencias de descripción de datos.

La ejecución de sentencias de descripción de datos (*DDL*) como la creación de *BD*, tablas, índices, etc., a través de *JDBC* se puede realizar **utilizando objetos `Statement` o `PreparedStatement`** (se verá en un próximo punto).

Nota: tener en cuenta que la creación de *BD* y tablas puede variar según el *SGBD* que se esté utilizando.

Ejemplo: ejecución de sentencias del *DDL* haciendo uso de `Statement`.

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

```
import java.sql.Statement;
public class OperacionesDDLEnJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/", usuario = "root", clave = "root", nomBD = "nueva_bd";
        try (Connection conexion = DriverManager.getConnection(url, usuario, clave)) {
            // Crear una nueva BD
            try (Statement crearBaseDeDatosStmt = conexion.createStatement()) {
                String crearBaseDeDatosSQL = "CREATE DATABASE " + nomBD;
                crearBaseDeDatosStmt.executeUpdate(crearBaseDeDatosSQL);
                System.out.println("BD creada con éxito.");
            } catch (SQLException e) {
                System.out.println("Error al crear la BD '" + nomBD + "': " + e.getMessage());
                return; // Salir si hay error al crear la BD
            }
            // Usar la nueva BD
            String urlNuevaBD = url + nomBD;
            try (Connection nuevaConexion = DriverManager.getConnection(urlNuevaBD, usuario, clave);
                 Statement crearTablaStmt = nuevaConexion.createStatement()) {
                // Crear una tabla en la nueva BD
                String crearTablaSQL = "CREATE TABLE ejemplo_tabla ("
                        + "id INT AUTO_INCREMENT PRIMARY KEY,"
                        + "nombre VARCHAR(255) NOT NULL,"
                        + "edad INT)";
                crearTablaStmt.executeUpdate(crearTablaSQL);
                System.out.println("Tabla creada con éxito en la nueva BD.");
                // Modificar la tabla (agregar una columna)
                String modificarTablaSQL = "ALTER TABLE ejemplo_tabla ADD COLUMN nuevo_campo VARCHAR(255)";
                crearTablaStmt.executeUpdate(modificarTablaSQL);
                System.out.println("Tabla modificada con éxito.");
                // Eliminar la tabla
                String eliminarTablaSQL = "DROP TABLE ejemplo_tabla";
                crearTablaStmt.executeUpdate(eliminarTablaSQL);
                System.out.println("Tabla eliminada con éxito.");
            } catch (SQLException e) {
                System.out.println("Error al realizar operaciones en la BD '" + nomBD + "': " + e.getMessage());
            }
            // Eliminar la nueva BD
            try (Statement eliminarBaseDeDatosStmt = conexion.createStatement()) {
                String eliminarBaseDeDatosSQL = "DROP DATABASE " + nomBD;
                eliminarBaseDeDatosStmt.executeUpdate(eliminarBaseDeDatosSQL);
                System.out.println("BD eliminada con éxito.");
            } catch (SQLException e) {
                System.out.println("Error al eliminar la BD '" + nomBD + "': " + e.getMessage());
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

TAREA

3. Crear un programa que se conecte a una *BD* y realice las siguientes tareas:

- **Muestre una lista de todas las tablas disponibles en la *BD*.**
- **Permita al usuario seleccionar una tabla por su nombre y muestre información sobre sus columnas (nombres, tipos de datos y si permiten valores nulos).**
- **Pida al usuario el nombre de una nueva *BD* y la cree. Además, debe pedir el nombre de una nueva tabla y los campos que tendrá junto a sus tipos de datos y crearla dentro de la nueva *BD*.**

7 Ejecución de múltiples sentencias en una llamada.

La **propiedad `allowMultiQueries=true` se utiliza en la cadena de conexión *JDBC*** (*URL* de conexión) **para habilitar la ejecución de múltiples sentencias o scripts *SQL* en una sola llamada al *SGBD***.

Las **sentencias *SQL* deben ir separadas por punto y coma** (;) **en una sola cadena de texto**. El *SGBD* ejecutará cada sentencia en orden y devolverá los resultados correspondientes.

Tener en cuenta que el **uso de `allowMultiQueries=true` puede tener implicaciones de seguridad si se utilizan consultas generadas dinámicamente con datos no confiables**, ya que podría dar lugar a inyección de *SQL*.

No todos los *SGBD* admiten la propiedad `allowMultiQueries` para ejecutar múltiples sentencias en una sola llamada a la *BD*. Por tanto, es importante comprobar la documentación del *SGBD* para conocer si esta característica está disponible y cómo se debe utilizar. Compatibilidad:

- ✓ Admiten la propiedad: ***MySQL/MariaDB***.
- ✓ No admiten la propiedad: *SQLite*, *Oracle*, *SQL Server*, *PostgreSQL*.

Ejemplo: ejecución de varias sentencias `INSERT` en una única llamada.

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

public class EjemploMultiQueriesJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb?allowMultiQueries=true";

        try (Connection conexion = DriverManager.getConnection(url, "mydb", "password");
             Statement statement = conexion.createStatement()) {
            // Ejemplo de ejecución de múltiples consultas en una sola llamada
            String consultaSQL = "INSERT INTO departamentos VALUES (6, 'Marketing', 'Granada');" +
                    "INSERT INTO departamentos VALUES (7, 'Publicidad', 'Granada');";

            boolean resultado = statement.execute(consultaSQL);

            if (!resultado)
                System.out.println("Las sentencias se ejecutaron con éxito.");
            else
                System.out.println("Las sentencias devolvieron un conjunto de resultados.");
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        }
    }
}
```

Ejemplo: ejecución de un script *SQL*.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class EjecutarScriptSQL {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb?allowMultiQueries=true";
        String archivoScript = "ruta_del_script.sql";

        try (Connection conexion = DriverManager.getConnection(url, "mydb", "password");
             Statement statement = conexion.createStatement();
             BufferedReader br = new BufferedReader(new FileReader(archivoScript))) {
            // Leer el archivo de script SQL línea por línea
            String linea;
            StringBuilder scriptCompleto = new StringBuilder();
            while ((linea = br.readLine()) != null)
                scriptCompleto.append(linea).append("\n");

            // Ejecutar el script SQL que contiene múltiples sentencias
            boolean tieneResultados = statement.execute(scriptCompleto.toString());

            if (!tieneResultados)
                System.out.println("El script se ejecutó con éxito.");
            else
                System.out.println("El script devolvió un conjunto de resultados.");
        } catch (SQLException e) {
            System.out.println("Error de SQL: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Error al leer el fichero: " + e.getMessage());
        }
    }
}
```

El programa se conecta a una *BD MariaDB* y ejecuta un script *SQL* ubicado en un archivo. Utiliza un `BufferedReader` para leer el contenido del archivo línea por línea y construir una cadena que representa el script completo. Luego, se ejecuta este script a través de un `Statement`.

8 Sentencias preparadas.

8.1 *SQL injection*.

SQL injection es una **técnica de hacking** muy conocida, sencilla y fácil de utilizar. Consiste en **inyectar código *SQL* en una consulta mediante la entrada de datos**. El atacante inserta o manipula maliciosamente sentencias *SQL* que un programa de aplicación envía al *SGBD*. Esto **se hace aprovechando la falta de validación o escape de datos de entrada por parte de la aplicación**.

Un ejemplo sencillo de *SQL injection* sería si una aplicación utiliza una consulta *SQL* como la siguiente para autenticar a un usuario:

```
SELECT * FROM usuarios WHERE usuario = 'nombreUsuario' AND clave = 'contraseña';
```

Si la aplicación no valida o escapa adecuadamente las entradas del usuario y un atacante ingresa ' OR '1'='1 como contraseña, la consulta modificada se vería así:

```
SELECT * FROM usuarios WHERE usuario = 'nombreUsuario' AND clave = '' OR '1'='1';
```

En este caso, la condición '1'='1' siempre es verdadera, lo que significa que el atacante podría obtener acceso no autorizado, ya que la consulta devolverá al menos una fila.

Para prevenir *SQL injection*, es esencial utilizar sentencias preparadas o parametrizadas, que permiten a la *BD* distinguir entre las sentencias *SQL* y los datos proporcionados por el usuario. También **es muy importante validar** (haciendo uso de expresiones regulares) **y sanitizar** (eliminar caracteres o secuencias) **todas las entradas del usuario antes de incluirlas en sentencias *SQL*** (validar todos y cada uno de los datos que se recogen en la aplicación, evitando que incluyan comillas simples, puntos y comas y todos aquellos caracteres que faciliten la inyección de código *SQL*). Además, **limitar los privilegios de la cuenta de acceso a la *BD* utilizada por la aplicación a solo aquellos necesarios** para realizar las operaciones requeridas también ayuda a mitigar el riesgo de *SQL injection*.

Ejemplo: vulnerabilidad de la inyección *SQL*.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class VulnerableApp {
    public static void main(String[] args) {
        String userInput = "'; DROP TABLE usuarios; --"; // Entrada maliciosa del usuario
        // Construcción de la consulta sin validar la entrada del usuario
        String query = "SELECT * FROM usuarios WHERE nombre = '" + userInput + "'";
        try (Connection connection = DriverManager.getConnection("jdbc:mariadb://localhost/base_de_datos",
            "usuario", "contraseña"); Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(query)) {
            // Procesamiento de los resultados
            while (resultSet.next()) {
                System.out.println("Nombre: " + resultSet.getString("nombre"));
                // Otros campos...
            }
        } catch (SQLException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Este código es vulnerable a la inyección *SQL* porque construye la consulta mediante la concatenación de cadenas, sin validar ni escapar la entrada del usuario. Un atacante podría introducir un valor malicioso en `userInput`, como el que se muestra, para manipular la consulta y realizar acciones no deseadas, como eliminar la tabla de `usuarios`.

La forma correcta de hacer esto es utilizando sentencias preparadas. Ejemplo corregido para que no se pueda dar la inyección *SQL*:

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

```
public class SeguraApp {
    public static void main(String[] args) {
        String userInput = "'; DROP TABLE usuarios; --"; // Input malicioso del usuario
        // Uso de una sentencia preparada para evitar la inyección SQL
        try (Connection connection = DriverManager.getConnection("jdbc:mariadb://localhost/base_de_datos",
                                                                 "usuario", "contraseña")) {
            // Uso de una sentencia preparada para evitar la inyección SQL
            String query = "SELECT * FROM usuarios WHERE nombre = ?";
            try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {
                preparedStatement.setString(1, userInput); // Parámetro 1 (el primero), no empieza en 0
                try (ResultSet resultSet = preparedStatement.executeQuery()) {
                    // Procesamiento de los resultados
                    while (resultSet.next()) {
                        System.out.println("Nombre: " + resultSet.getString("nombre"));
                        // Otros campos...
                    }
                }
            }
        } catch (SQLException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

En este segundo ejemplo, el uso de `PreparedStatement` y la asignación de valores mediante `setString` previenen eficazmente la inyección *SQL*.

Es crucial proteger las aplicaciones contra la inyección SQL mediante prácticas seguras de programación, como el uso de consultas parametrizadas o el uso de funciones que escapen y validen correctamente las entradas del usuario.

8.2 Interfaz `PreparedStatement`.

Como se ha visto en los ejemplos del punto anterior, para evitar *SQL Injection* no es buena idea construir sentencias como concatenación de cadenas a partir de datos introducidos por el usuario. En lugar de esto, se usarán **sentencias parametrizadas**, que son aquellas que **incluyen unos marcadores o parámetros que se sustituyen por valores**. Este mecanismo permite adaptar y reutilizar la misma consulta varias veces. En ***JDBC***, la interfaz **`PreparedStatement` representa una consulta parametrizada**.

La **interfaz `PreparedStatement`** es una **extensión de la interfaz `Statement`** y **se utiliza para ejecutar sentencias *SQL* precompiladas que pueden contener parámetros** (sentencias parametrizadas). Esta interfaz es **especialmente útil para evitar problemas de seguridad como la inyección de *SQL* y para mejorar el rendimiento en ejecuciones repetidas de la misma sentencia con diferentes valores de parámetros**.

Un `PreparedStatement` se precompila una vez y luego puede reutilizarse con diferentes valores de parámetros sin tener que volver a compilar la sentencia *SQL*, lo que puede ser más eficiente en términos de rendimiento.

Los **objetos `PreparedStatement` no se pueden utilizar para representar nombres de columnas o tablas**, ya que estos nombres son estáticos y se definen en la propia consulta.

El **símbolo ?** (*placeholder*) se utiliza como **marcador de posición para representar valores que serán proporcionados más tarde como parámetros en la sentencia *SQL***. Estos **marcadores de posición permiten crear consultas parametrizadas en las que los valores reales se asignan de manera dinámica antes de ejecutar la sentencia**, lo que mejora la seguridad y el rendimiento.

El **uso de ?** en las sentencias preparadas tiene varios **propósitos** importantes:

- ✓ **Prevención de inyección de *SQL***: los valores de los parámetros se tratan como datos y no como parte de la consulta *SQL*, lo que hace que sea extremadamente difícil para un atacante modificar la consulta.
- ✓ **Reutilización de sentencias**: las sentencias preparadas **se precompilan una vez y luego se pueden reutilizar con diferentes valores de parámetros** sin tener que volver a compilarla.
- ✓ **Legibilidad del código**: la estructura de la **sentencia se mantiene separada de los valores de los parámetros**.

Una ventaja de este tipo de consultas es que **no es necesario prestar atención a las comillas, solo hay que asignar valores a los parámetros** y es el propio *JDBC* el que determina que campos se entrecomillan.

En el caso de tener que **reutilizar una sentencia**, es tan simple como **asignar nuevos valores a los parámetros**.

Para **asignar los valores a los parámetros** se dispone de **distintos métodos setter** (tantos como tipos de datos), donde el **primer parámetro indica el índice del parámetro** de la sentencia (los parámetros se comienzan a contar desde 1), **y el segundo es el valor que se quiere asignar**.

En ocasiones, puede ser interesante que el **valor de un parámetro sea nulo**, para ello se dispone del método:

```
void setNull(int parameterIndex, int sqlType)
```

que pone a nulo el parámetro indicado, que se trata como si fuera del tipo `sqlType`, y que será uno de los **tipos definidos en `java.sql.Types`**: INTEGER, BOOLEAN, VARCHAR, DECIMAL, etc.

Tener en cuenta, que en las **consultas el objeto `ResultSet` es el cursor que contiene el resultado de la consulta** y, al recorrerlo, **se puede acceder a cualquier columna de dicho resultado indicando el número de ésta**, siempre teniendo en cuenta que **la posición 1 se utiliza para acceder a la primera columna**.

En el **caso** de consultas que hagan uso de **funciones agregadas**, tener en cuenta que **sólo van a devolver un valor, por lo que no será necesario preparar el código para recorrer el cursor**. Se podrá acceder directamente a la primera fila del `ResultSet` y mostrar el resultado (usar `resultado.next();` sin incluirla en un bucle).

Tabla con algunos de los **métodos más utilizados** de la **interfaz `PreparedStatement`**:

Método	Descripción
`PreparedStatement pstmt = connection.prepareStatement(String sql)`	Crea un objeto PreparedStatement a partir de una consulta SQL parametrizada.
`void setBoolean(int parameterIndex, boolean x)`	Establece un valor booleano en el parámetro especificado.
`void setByte(int parameterIndex, byte x)`	Establece un valor byte en el parámetro especificado.
`void setShort(int parameterIndex, short x)`	Establece un valor entero corto en el parámetro especificado.
`void setInt(int parameterIndex, int x)`	Establece un valor entero en el parámetro especificado.
`void setLong(int parameterIndex, long x)`	Establece un valor entero largo en el parámetro especificado.
`void setString(int parameterIndex, String x)`	Establece una cadena de caracteres en el parámetro especificado.
`void setFloat(int parameterIndex, float x)`	Establece un valor de punto flotante en el parámetro especificado.
`void setDouble(int parameterIndex, double x)`	Establece un valor de doble precisión en el parámetro especificado.
`void setDate(int parameterIndex, Date x)`	Establece un objeto `java.sql.Date` en el parámetro especificado.
`void setTime(int parameterIndex, Time x)`	Establece un objeto `java.sql.Time` en el parámetro especificado.
`void setNull(int parameterIndex, int sqlType)`	Establece un parámetro como nulo con un tipo SQL específico.
`ResultSet executeQuery()`	Ejecuta la consulta SQL y devuelve un conjunto de resultados.
`int executeUpdate()`	Ejecuta una sentencia de actualización (INSERT, UPDATE o DELETE) y devuelve el número de filas afectadas.
`boolean execute()`	Ejecuta la consulta SQL y determina si devuelve un conjunto de resultados (true) o no (false).
`void clearParameters()`	Borra todos los valores de parámetros previamente establecidos.
`void close()`	Cierra el PreparedStatement y libera recursos.
`ResultSetMetaData getMetaData()`	Obtiene información sobre los metadatos de los resultados de la consulta.
`void addBatch()`	Agrega la consulta actual al lote de consultas para ejecución en lote.
`int[] executeBatch()`	Ejecuta el lote de consultas y devuelve un arreglo de enteros con el número de filas afectadas por cada consulta.
`void setBlob(int parameterIndex, InputStream inputStream)`	Establece un parámetro BLOB (Binary Large Object) utilizando un flujo de entrada de datos.
`void setClob(int parameterIndex, Reader reader)`	Establece un parámetro CLOB (Character Large Object) utilizando un lector de caracteres.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html

Ejemplo: obtener un listado con el nombre y salario de los empleados cuyo id de departamento sea mayor que 2.

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

```
import java.sql.ResultSet;
import java.sql.SQLException;

public class EjemploPreparedStatement {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb";
        int idDepartamento = 2;
        String consultaSQL = "SELECT nombre, salario FROM empleados WHERE id_departamento > ?";

        try (Connection con = DriverManager.getConnection(url, "mydb", "password");
             PreparedStatement ps = con.prepareStatement(consultaSQL)) {
            ps.setInt(1, idDepartamento); // Establecer un valor de parámetro

            // Ejecutar la consulta
            try (ResultSet resultado = ps.executeQuery()) {
                while (resultado.next()) {
                    String nombre = resultado.getString("nombre");
                    double salario = resultado.getDouble("salario");
                    System.out.println("Nombre: " + nombre + ", Salario: " + salario);
                }
            }
        } catch (SQLException e) {
            System.out.println("Error de SQL en la consulta: " + e.getMessage());
        }
    }
}
```

Ejemplo: insertar un nuevo departamento haciendo uso de los valores pasados como parámetros al programa.

Método	Función
int getErrorCode()	Devuelve un entero que proporciona el código de error del fabricante. Normalmente será el código de error real devuelto por la BD
String getSQLState()	Devuelve una cadena que contiene un estado definido por el estándar X/OPEN SQL
String getMessage()	Devuelve una cadena que describe el error. Es un método heredado de la clase java.Lang.Throwable

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

public class InsertaDepPreparedStatement {
    public static void main(String[] args) {
        if (args.length == 3) {
            String idDep = args[0], nombre = args[1], localidad = args[2];

            try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb", "mydb", "password")) {
                int idDepartamento;
                try {
                    idDepartamento = Integer.parseInt(idDep); // Validar que el ID del departamento sea un entero
                } catch (NumberFormatException e) {
                    System.out.println("Error: el ID del departamento debe ser un número entero.");
                    return;
                }

                String sql = "INSERT INTO departamentos VALUES (?, ?, ?)";
                try (PreparedStatement sentencia = con.prepareStatement(sql)) {
                    sentencia.setInt(1, idDepartamento);
                    sentencia.setString(2, nombre);
                    sentencia.setString(3, localidad);
                    int filas = sentencia.executeUpdate();
                    System.out.println("Filas afectadas: " + filas);
                } catch (SQLException e) {
                    System.out.println("Error al insertar el departamento:");
                    System.out.println("Mensaje:    " + e.getMessage());
                    System.out.println("SQL estado: " + e.getSQLState());
                    System.out.println("Cód. error: " + e.getErrorCode());
                }
            } catch (SQLException e) {
                System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
                System.out.println("Mensaje:    " + e.getMessage());
                System.out.println("SQL estado: " + e.getSQLState());
                System.out.println("Cód. error: " + e.getErrorCode());
            }
        } else {
            System.out.println("Error: se deben pasar tres parámetros al llamar al programa.");
        }
    }
}
```

```
$> SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'test.no_such_table' doesn't exist
```

ErrorCode SQLState Message

TAREA

4. **Crear un programa que muestre los datos de los empleados de un departamento, cuyo valor se recibirá como argumento desde la línea de comandos. El programa también debe mostrar el nombre del departamento, el salario medio y el número total de empleados. Si el departamento no existe, se debe mostrar un mensaje de**

error.

Nota: utilizar la clase `DecimalFormat` para dar formato al sueldo (`DecimalFormat formato = new DecimalFormat("##,##0,00"); String valor_formateado = formato.format(resul.getFloat(1));`).

5. **Dada la siguiente estructura de *BD*:**

Tabla usuarios:
- **ID (Clave primaria)**
- **Nombre**
- **Correo electrónico**
- **Contraseña**

Tabla productos:
- **ID (Clave primaria)**
- **Nombre del producto**
- **Descripción del producto**
- **Precio**
- **Cantidad en stock**

Tabla pedidos:
- **ID (Clave primaria)**
- **ID del usuario (Clave foránea que se relaciona con la tabla usuarios)**
- **ID del producto (Clave foránea que se relaciona con la tabla productos)**
- **Cantidad solicitada**
- **Fecha del pedido**

Desarrollar un programa que cree la *BD* y las tablas indicadas si no existen, y que muestre un menú con las siguientes opciones (usando sentencias preparadas):

a) **Ver el historial de pedidos de un usuario específico.**
b) **Actualizar la información de perfil de los usuarios (nombre, correo electrónico y contraseña).**
c) **Eliminar usuarios.**
d) **Actualizar la cantidad en stock de un producto.**
e) **Registrar nuevos pedidos, teniendo en cuenta la relación entre usuarios y productos.**

9 Procedimientos y funciones almacenadas.

9.1 Procedimientos y funciones almacenadas en *MariaDB*.

Un **procedimiento almacenado** (también conocidos como "stored procedure" en inglés) es un **conjunto de instrucciones/sentencias *SQL* predefinidas y almacenadas en el *SGBD* para su posterior ejecución**. Estos procedimientos almacenados pueden tomar parámetros de entrada, realizar operaciones en la *BD* y devolver resultados.

A los procedimientos almacenados, **si devuelven un valor, se les llama función**.

Ejemplos: funciones y procedimientos almacenados en *MariaDB* sobre la *BD mydb*.

```
DELIMITER //
CREATE OR REPLACE PROCEDURE subir_salario(id_p TINYINT, subida_p DECIMAL(7,2))
BEGIN
    UPDATE empleados SET salario = salario + subida_p WHERE id_departamento = id_p;
END//
CREATE OR REPLACE PROCEDURE obtener_nombre_localidad_departamento(id_p TINYINT, OUT nombre_p
    VARCHAR(15), OUT localidad_p VARCHAR(15))
BEGIN
    SET localidad_p = 'INEXISTENTE';
    SET nombre_p = 'INEXISTENTE';
    SELECT nombre, localidad INTO nombre_p, localidad_p FROM departamentos WHERE id_departamento =
    id_p;
END//
CREATE OR REPLACE FUNCTION obtener_nombre_departamento(id_p TINYINT) RETURNS VARCHAR(15)
BEGIN
    DECLARE nombre_d VARCHAR(15);
    SET nombre_d = 'INEXISTENTE';
    SELECT nombre INTO nombre_d FROM departamentos WHERE id_departamento = id_p;
    RETURN nombre_d;
END//
DELIMITER ;
CALL subir_salario(2, 250);
CALL obtener_nombre_localidad_departamento(2, @nombre, @localidad);
SELECT @nombre, @localidad;
SELECT obtener_nombre_departamento(2);
```

En *MariaDB*, **se pueden conceder permisos para ejecutar procedimientos y funciones almacenadas utilizando la sentencia `GRANT`**. La **sintaxis** general para conceder estos permisos es la siguiente:

```
GRANT EXECUTE ON PROCEDURE database_name.procedure_name TO 'username'@'hostname';
GRANT EXECUTE ON FUNCTION database_name.function_name TO 'username'@'hostname';
```

, donde:

- ✓ **`database_name`** es el nombre de la *BD* donde se encuentra el procedimiento o función almacenada.
- ✓ **`procedure_name/function_name`** es el nombre del procedimiento/función al que se desea conceder permisos de ejecución.
- ✓ **`'username'@'hostname'`** es el usuario y el host al que se desea otorgar los permisos de ejecución.

Recordar que **se deben tener privilegios de administrador o ser el propietario del procedimiento o función almacenada para conceder estos permisos**.

Para **dar permisos a un usuario** en *MariaDB* **para crear procedimientos y funciones almacenados**, se puede utilizar igualmente la sentencia GRANT. **Sintaxis**:

```
GRANT CREATE ROUTINE ON database_name.* TO 'nombre_usuario'@'localhost';
```

Ejemplo:

```
GRANT EXECUTE ON PROCEDURE mydb.obtener_nombre_localidad_departamento TO 'mydb'@'%';
GRANT EXECUTE ON FUNCTION mydb.obtener_nombre_departamento TO 'mydb'@'%';
```

9.2 Ejecución de procedimientos y funciones almacenadas.

La ejecución de procedimientos y funciones almacenadas **sigue la misma estructura que cualquiera de las sentencias *SQL* de los ejemplos anteriores**, **con la excepción de que se usará la interfaz `CallableStatement` para representar al procedimiento y el método `execute()` de la misma para ejecutarlo**.

La interfaz `CallableStatement` **extiende la interfaz `PreparedStatement`**, que se utiliza para ejecutar sentencias *SQL* parametrizadas. `CallableStatement` **agrega funcionalidad adicional para trabajar con procedimientos almacenados**, lo que incluye:

- ✓ **Registro de parámetros de entrada y salida**: se pueden registrar los parámetros de entrada y salida del procedimiento almacenado utilizando métodos como `registerOutParameter`. Esto permite pasar valores al procedimiento almacenado y recibir valores devueltos después de su ejecución.
- ✓ **Ejecución de procedimientos almacenados**: se puede ejecutar el procedimiento almacenado utilizando el método `execute` o sus variantes. Los parámetros se pueden establecer mediante métodos como `setString`, `setInt`, etc., antes de la ejecución.
- ✓ **Recuperación de resultados**: después de ejecutar el procedimiento almacenado, se pueden recuperar los valores de los parámetros de salida utilizando métodos como `getInt`, `getString`, etc.

Según si devuelve algo o no, **la llamada se realizaría de una de las siguientes formas**:

- **Función**: **`{?=call <nombre_func>[(arg1, arg2...)]}`**
- **Procedimiento**: **`{call <nombre_proc>[(arg1, arg2...)]}`**

Las **llaves {} en la llamada no son necesarias para la mayoría de los *SGBD*** cuando se ejecutan procedimientos almacenados a través de *JDBC*. Las llaves se utilizan en algunos *SGBD* específicos (como *Oracle*) para definir bloques anónimos *PL/SQL*, pero no son necesarias para la mayoría de las operaciones de ejecución de procedimientos almacenados. En **caso de no recibir parámetros el procedimiento o la función se puede prescindir de los ()**.

En la mayoría de los casos, el **usuario necesita tener permisos adecuados para ejecutar procedimientos almacenados a través de `CallableStatement`**. Estos permisos varían según la *BD* y la configuración de seguridad, y generalmente **se gestionan a nivel de la *BD* y se otorgan por separado del código *Java* que llama al procedimiento almacenado**.

Tabla con algunos de los **métodos más utilizados** de la **interfaz `CallableStatement`**:

Método	Descripción
`void registerOutParameter(int parameterIndex, int sqlType)`	Registra un parámetro de salida para un procedimiento almacenado en una posición específica. El sqlType indica el tipo de datos que se espera como resultado.
`void setInt(int parameterIndex, int x)`	Establece un valor entero como un parámetro de entrada para el procedimiento almacenado en la posición especificada.

`void setString(int parameterIndex, String x)`	Establece una cadena como un parámetro de entrada para el procedimiento almacenado en la posición especificada.
`void setBoolean(int parameterIndex, boolean x)`	Establece un valor booleano como un parámetro de entrada para el procedimiento almacenado en la posición especificada.
`void execute()`	Ejecuta el procedimiento almacenado con los parámetros proporcionados. Puede devolver resultados o no, dependiendo del procedimiento.
`int getInt(int parameterIndex)`	Obtiene un valor entero del resultado de un parámetro de salida del procedimiento almacenado en la posición especificada.
`String getString(int parameterIndex)`	Obtiene una cadena del resultado de un parámetro de salida del procedimiento almacenado en la posición especificada.
`boolean getBoolean(int parameterIndex)`	Obtiene un valor booleano del resultado de un parámetro de salida del procedimiento almacenado en la posición especificada.
`void close()`	Cierra el objeto `CallableStatement` después de su uso, liberando los recursos asociados.

Tipos de datos comunes que se pueden utilizar en `registerOutParameter`, junto con sus valores de código de tipo (`java.sql.Types`) y una descripción general:

Tipo de Dato	Código de Tipo (`java.sql.Types`)	Descripción
CHAR	CHAR	Cadena de caracteres de longitud fija.
VARCHAR	VARCHAR	Cadena de caracteres de longitud variable.
INTEGER	INTEGER	Número entero de 32 bits.
SMALLINT	SMALLINT	Número entero corto de 16 bits.
BIGINT	BIGINT	Número entero largo de 64 bits.
REAL	REAL	Número de punto flotante de precisión simple.
DOUBLE	DOUBLE	Número de punto flotante de precisión doble.
DATE	DATE	Fecha (sin hora).
TIME	TIME	Hora (sin fecha).
TIMESTAMP	TIMESTAMP	Fecha y hora.
BOOLEAN	BOOLEAN	Valor booleano (verdadero/falso).
BLOB	BLOB	Objeto binario grande.
CLOB	CLOB	Objeto de caracteres grandes.
ARRAY	ARRAY	Matriz (Array) de valores.
STRUCT	STRUCT	Estructura personalizada.
CURSOR	CURSOR (o REF_CURSOR)	Cursor o conjunto de resultados.

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/CallableStatement.html

Ejemplo: llamar a un procedimiento almacenado en una *BD*.

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

public class CallableStatementEjemplo {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection("jdbc:mariadb://localhost/mydb", "mydb", "password");
             CallableStatement cs = con.prepareCall("{call obtener_nombre_localidad_departamento(?, ?, ?)}")) {
            // Registrar el parámetro de entrada
            cs.setInt(1, 2); // Parámetro de entrada: ID del departamento
            // Registrar los parámetros de salida
            cs.registerOutParameter(2, Types.VARCHAR); // Par. salida: nombre del departamento
            cs.registerOutParameter(3, Types.VARCHAR); // Par. salida: localidad

            // Ejecutar el procedimiento almacenado
            cs.execute();

            // Recuperar y procesar los valores de los parámetros de salida
            String nombre = cs.getString(2);
            String localidad = cs.getString(3);

            // Validar los resultados de salida
            if (nombre != null && localidad != null)
```

```
                System.out.println("El resultado es: " + nombre + " - " + localidad);
            else
                System.out.println("No se pudo obtener el nombre o la localidad.");
        } catch (SQLException e) {
            System.out.println("Error al ejecutar el procedimiento almacenado: " + e.getMessage());
        }
    }
}
```

Ejemplo: llamar a una función almacenada en una *BD*.

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

public class LlamadaFuncionEjemplo {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb", usuario = "mydb", clave = "password";

        try (Connection con = DriverManager.getConnection(url, usuario, clave);
             CallableStatement cs = con.prepareCall("{? = call obtener_nombre_departamento( ? )}")) {
            cs.registerOutParameter(1, Types.VARCHAR); // Registrar el parámetro de salida
            cs.setInt(2, 2); // Establecer el parámetro de entrada

            cs.execute(); // Ejecutar la función

            String nombre = cs.getString(1); // Obtener el resultado
            if (nombre != null)
                System.out.println("El nombre del departamento es: " + nombre);
            else
                System.out.println("No se encontró un nombre para el departamento.");
        } catch (SQLException e) {
            System.out.println("Error: " + e.getMessage());
            System.out.println("Código SQL: " + e.getErrorCode());
            System.out.println("Estado: " + e.getSQLState());
        }
    }
}
```

10 Gestión de transacciones.

En el ámbito de las *BD*, una **transacción** es cualquier **conjunto de sentencias *SQL* que se ejecutan como si de una sola se tratara**. La idea principal es poder ejecutar varias sentencias, que están relacionadas de alguna manera, de forma que, **si cualquiera de ellas falla o produce un error, no se ejecute ninguna más y se deshagan todos los cambios que hubieran podido efectuarse** (las que ya se habían ejecutado dentro de la misma transacción).

Para ello, se dispone de **tres métodos**: un método para indicar que comienza una transacción (**`conexion.setAutoCommit(false)`**), otro para indicar cuando termina de forma satisfactoria (**`conexion.commit()`**) y otra para indicar que la transacción actual debe abortarse y los cambios deben ser restaurados al estado anterior (**`conexion.rollback()`**).

Los **pasos** a seguir para trabajar con transacciones utilizando *JDBC* son:

1. **Establecer la conexión y deshabilitar la confirmación automática**. Cuando se crea una conexión a la *BD*, por defecto, *JDBC* utiliza el modo de confirmación automática (*auto-commit*), por lo que cada sentencia *SQL* se confirma automáticamente después de ejecutarse. Para gestionar las transacciones manualmente, se debe deshabilitar esta función:

   ```
   Connection conexion = DriverManager.getConnection(jdbcUrl, usuario, contraseña);
   conexion.setAutoCommit(false);
   ```

   Al desactivar la confirmación automática, se pueden agrupar múltiples operaciones en una transacción.
2. **Realizar operaciones del *DML* dentro de la transacción**.
3. **Confirmar o revertir la transacción**. Después de ejecutar todas las sentencias, se puede decidir si confirmar o revertir la transacción:
   - Para confirmar la transacción: **`conexion.commit();`**
   - Para revertir la transacción y deshacer todas las operaciones: **`conexion.rollback();`**
4. **Cerrar la conexión**. Una vez finalizada la transacción, cerrar la conexión: **`conexion.close();`**

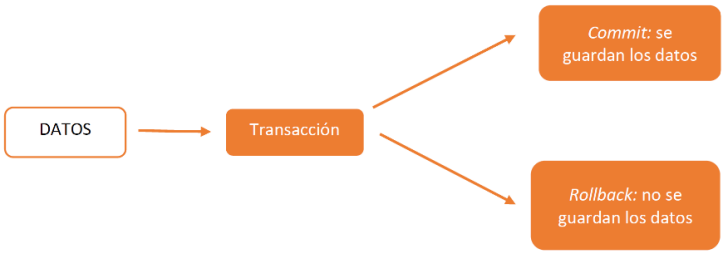
Puntos de guardado (***savepoints***):

✓ Un punto de guardado es un **marcador dentro de una transacción que permite volver a ese punto específico**

si es necesario, sin tener que deshacer la transacción completa.

- ✓ Se puede **crear un punto de guardado** utilizando el **método `Savepoint setSavepoint(String nombre)`** del objeto `Connection` (devuelve un objeto `Savepoint`).
- ✓ Se puede **borrar un punto de guardado** haciendo uso de **`void releaseSavepont(Savepoint nombre)`**.
- ✓ Para **volver a un punto de guardado**, se puede utilizar **`void rollback(Savepoint savepoint)`** y pasar el punto de guardado como argumento (no devuelve nada).

Ejemplo: se crea una tabla en una *BD* y se realizan varias operaciones (insertar, modificar y borrar registros) sobre ella haciendo uso de transacciones.




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

public class EjemploGestionTransacciones {
    public static void main(String[] args) {
        Connection conexion = null;
        String jdbcUrl = "jdbc:mariadb://localhost/mydb";

        try {
            conexion = DriverManager.getConnection(jdbcUrl, "mydb", "password");
            crearTabla(conexion); // Crear una tabla
            conexion.setAutoCommit(false); // Deshabilitar la confirmación automática
            // Insertar un registro
            insertarRegistro(conexion, "Juan", "Pérez");
            // Modificar un registro
            modificarRegistro(conexion, 1, "Pedro", "Gómez");
            eliminarRegistro(conexion, 1); // Eliminar un registro
            // Confirmar todas las sentencias una vez ejecutadas sin que se produzca un error
            conexion.commit();
            conexion.setAutoCommit(true); // Se vuelve a poner el autocommit, para el resto de la aplicación
            System.out.println("Transacción ejecutada con éxito.");
        } catch (SQLException e) {
            System.out.println();
            mostrarMensajeError(e);
            try { // Revertir todas las transacciones en caso de error
                if (conexion != null)
                    conexion.rollback(); // Si algo falla se hace "ROLLBACK"
            } catch (SQLException ex) {
                mostrarMensajeError(e);
            }
        } finally {
            try { // Cerrar la conexión
                if (conexion != null)
                    conexion.close();
            } catch (SQLException e) {
                mostrarMensajeError(e);
            }
        }
    }
    private static void crearTabla(Connection conexion) throws SQLException {
        String crearTablaSQL = "CREATE TABLE IF NOT EXISTS usuarios (id INT AUTO_INCREMENT, nombre VARCHAR(50),
apellido VARCHAR(50), PRIMARY KEY (id))";
        try (Statement statement = conexion.createStatement()) {
            statement.execute(crearTablaSQL);
        }
    }
    private static void insertarRegistro(Connection conexion, String nombre, String apellido) throws SQLException {
        String insertarSQL = "INSERT INTO usuarios (nombre, apellido) VALUES (?, ?)";
        try (PreparedStatement pstmt = conexion.prepareStatement(insertarSQL)) {
            pstmt.setString(1, nombre);
            pstmt.setString(2, apellido);
            pstmt.executeUpdate();
        }
    }
    private static void modificarRegistro(Connection conexion, int id, String nuevoNombre, String nuevoApellido)
            throws SQLException {
        String modificarSQL = "UPDATE usuarios SET nombre = ?, apellido = ? WHERE id = ?";
        try (PreparedStatement pstmt = conexion.prepareStatement(modificarSQL)) {
            pstmt.setString(1, nuevoNombre);
            pstmt.setString(2, nuevoApellido);
            pstmt.setInt(3, id);
```

Recordar que **para trabajar con transacciones en *MariaDB/MySQL* se debe utilizar el motor de almacenamiento *InnoDB***.

Incluir el `rollback()` en el bloque `catch` es esencial para garantizar que cualquier error durante el proceso de la transacción no deje la *BD* en un estado inconsistente.

```
            pstmt.executeUpdate();
        }
    }
    private static void eliminarRegistro(Connection conexion, int id) throws SQLException {
        String eliminarSQL = "DELETE FROM usuarios WHERE id = ?";
        try (PreparedStatement pstmt = conexion.prepareStatement(eliminarSQL)) {
            pstmt.setInt(1, id);
            pstmt.executeUpdate();
        }
    }
    private static void mostrarMensajeError(SQLException e) {
        System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
        System.out.println("Mensaje:    " + e.getMessage());
        System.out.println("SQL estado: " + e.getSQLState());
        System.out.println("Cód. error: " + e.getErrorCode());
    }
}
```

Ejemplo: uso de transacciones que hacen uso de los puntos de guardado (*savepoints*).

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Savepoint;
import java.sql.Statement;

public class EjemploTransaccionesJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mariadb://localhost/mydb";
        Connection conexion = null;
        Savepoint savepoint = null;

        try {
            conexion = DriverManager.getConnection(url, "mydb", "password");
            createTable(conexion); // Crear una tabla en la BD
            conexion.setAutoCommit(false); // Desactivar el autocommit para controlar las transacciones

            insertData(conexion, "Juan", 30); // Insertar datos en la tabla

            savepoint = conexion.setSavepoint("puntoGuardado"); // Crear un punto de guardado

            insertData(conexion, null, 25); // Intentar insertar datos con un error (nombre nulo)

            conexion.commit(); // Confirmar la transacción
            System.out.println("Transacción completada con éxito.");
        } catch (SQLException e) {
            System.err.println("Error SQL: " + e.getMessage());
            try {
                if (conexion != null && savepoint != null) {
                    conexion.rollback(savepoint); // Volver al punto de guardado en caso de error
                    System.err.println("Rollback al punto de guardado.");
                }
            } catch (SQLException e2) {
                System.err.println("Error al hacer rollback al punto de salvaguarda.");
            }
        } finally {
            try {
                if (conexion != null)
                    conexion.close();
            } catch (SQLException e) {
                System.err.println("Error al cerrar la conexión.");
            }
        }
    }
    private static void createTable(Connection conexion) throws SQLException {
        String createTableSQL = "CREATE TABLE IF NOT EXISTS personas ("
                + "id INT AUTO_INCREMENT PRIMARY KEY,"
                + "nombre VARCHAR(255) NOT NULL,"
                + "edad INT NOT NULL)";
        try (Statement statement = conexion.createStatement()) {
            statement.executeUpdate(createTableSQL);
        }
    }
    private static void insertData(Connection conexion, String nombre, int edad) throws SQLException {
        String insertSQL = "INSERT INTO personas (nombre, edad) VALUES (?, ?)";
        try (PreparedStatement preparedStatement = conexion.prepareStatement(insertSQL)) {
            preparedStatement.setString(1, nombre);
            preparedStatement.setInt(2, edad);
```

```
                preparedStatement.executeUpdate();
            }
        }
    }
```

TAREA

6. **Diseñar un procedimiento almacenado en *MariaDB* que reciba un id. de departamento y devuelva el salario medio de los empleados de ese departamento, así como el número de empleados del departamento. Si el departamento no existe, debe devolver como salario medio -1 y como empleados 0.**
   Crear un programa que cree y llame al procedimiento diseñado para recorrer la tabla departamentos y mostrar los datos del departamento, incluyendo el número de empleados y el salario medio.
7. **Crear un programa que conecte con una *BD* sobre *Pokémon* en *MariaDB* de forma totalmente transparente para el usuario e implementar un menú que permita realizar las siguientes acciones:**
   a) **Opción de listar el contenido de una tabla *pokemon*, mostrando solamente su *nombre* y el *ps*, ordenado por este último campo.**
   b) **Opción de registrar nuevas filas en la tabla *pokemon*. Se deberá permitir registrar un *Pokémon* sólo indicando el nombre por parte del usuario, aunque los demás campos sean obligatorios.**
   c) **Opción de subir el *ps* de los *Pokémon* (incrementar su nivel en una unidad).**
   d) **Opción de buscar *Pokémon* introduciendo su nombre o descripción.**
   e) **Opción de eliminar filas de la tabla *pokemon* indicando el *id_pokemon* que se quiere eliminar.**
   f) **Opción de listar el contenido de la tabla *pokemon*.**
   g) **Opción de salir del programa.**

#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
1	id_pokemon	SMALLINT	5	☑	☐	☐	AUTO_INCREME...
2	nombre	VARCHAR	15	☐	☐	☐	Sin valor predeter...
3	peso	DECIMAL	6,2	☐	☐	☐	Sin valor predeter...
4	altura	DECIMAL	6,2	☐	☐	☐	Sin valor predeter...
5	ps	TINYINT	3	☑	☐	☐	Sin valor predeter...
6	ataque	TINYINT	3	☑	☐	☐	Sin valor predeter...
7	defensa	TINYINT	3	☑	☐	☐	Sin valor predeter...
8	especial	TINYINT	3	☑	☐	☐	Sin valor predeter...
9	velocidad	TINYINT	3	☑	☐	☐	Sin valor predeter...
10	sexo	SET	'M','H'	☐	☐	☐	Sin valor predeter...
11	descripcion	VARCHAR	300	☐	☑	☐	NULL
12	url_imagen	VARCHAR	250	☐	☑	☐	NULL

   Nota: la estructura de la tabla se muestra en la imagen.
8. **Crear una aplicación para gestionar una flota de taxis. La aplicación tiene que almacenar los siguientes datos:**
   - **De cada taxista: nombre, *DNI* y fecha de nacimiento.**
   - **De cada taxi: precio, matrícula y número de plazas.**

   Cuando empieza el turno de un taxista se le asigna uno de los taxis que no está siendo utilizado por nadie. Cada taxista, durante su jornada laboral, y hasta que esta concluya, será responsable del taxi asignado. Cuando finaliza el trabajo de un taxista, devuelve el taxi utilizado, que estará libre para asignarlo a otro trabajador.
   La aplicación tendrá el siguiente menú:

   ```
   1. Alta de nuevo taxista.
   2. Alta de nuevo taxi.
   3. Comienzo de la jornada taxista.
   4. Fin de la jornada taxista.
   5. Información de un taxista y su taxi.
   6. Mostrar taxistas trabajando.
   7. Mostrar taxistas fuera de servicio.
   8. Salir.
   ```

   Para facilitar el trabajo de los usuarios de la aplicación, cada vez que se solicite un *DNI* o una matrícula, se mostrará un listado con los datos disponibles. Por ejemplo, cuando se quiere finalizar una jornada de un taxista, antes de pedir el *DNI*, se puede mostrar todos los taxistas que están trabajando.

11 Validación de entradas.

Una **vía importante de errores de seguridad y de inconsistencia de datos dentro de una aplicación se produce a través de los datos que introducen los usuarios**. Un fallo de seguridad muy frecuente, consiste en los errores basados en *buffer overflow*, se producen cuando se desborda el tamaño de una determinada variable, array, etc. y se consigue

acceder a zonas de memoria reservadas. Por ejemplo, si se reserva memoria para el nombre de usuario que ocupa 20 caracteres y, de alguna forma, el usuario consigue que la aplicación almacene más datos, se está produciendo un *buffer overflow* ya que los primeros 20 valores se almacenan en un lugar correcto, pero los restantes valores se almacenan en zonas de memoria destinadas a otros fines.

La **validación de datos permite**:

- ✓ **Mantener la consistencia de los datos**. Por ejemplo, si a un usuario se le indica que debe introducir su *DNI* éste debe tener el mismo formato siempre.
- ✓ **Evitar desbordamientos de memoria *buffer overflow***. Al comprobar el formato y la longitud del campo se evita que se produzcan los desbordamientos de memoria.

Para llevar un riguroso control, sobre los datos de entrada de los usuarios **hay que tener en cuenta la validación del formato y la validación del tamaño de la entrada**.

***Java* incorpora una potente y útil librería (`java.util.regex`) para utilizar la clase `Pattern` que permite definir expresiones regulares**. Las expresiones regulares **permiten definir exactamente el formato de la entrada de datos**.

Elementos para la validación entradas

Elemento	Comentario.
x	El carácter x.
[abc]	Los caracteres a, b o c.
[a-z]	Una letra en minúscula.
[A-Z]	Una letra en mayúscula.
[a-zA-Z]	Una letra en minúscula o mayúscula.
[0-9]	Un número comprendido entre el 0 y el 9.
[a-zA-Z0-9]	Una letra en minúscula, mayúscula o un número.

Para utilizar las expresiones regulares se deben realizar los siguientes pasos:

1. **Importar la librería**.
   ```
   import java.util.regex.*;
   ```
2. **Definir *Pattern* y *Matcher***.
   ```
   Pattern pat = null;
   Matcher mat = null;
   ```
3. **Compilar el patrón a utilizar**.
   ```
   pat = Pattern.compile(patron);
   ```
   , donde el patrón a comprobar es la parte más importante ya que es donde se tiene que indicar el formato que va a tener la entrada.

Operadores para la validación entradas

Operador	Comentario
[a-z]{2}	Hay que introducir 2 letras en minúsculas.
[a-z]{2,5}	Hay que introducir de 2 a 5 letras en minúsculas.
[a-z]{2,}	Hay que introducir más de 2 letras en minúsculas.
hola|adios	Es la operación OR lógica y permite indicar que se introduzca el texto "hola" o "adios".
XY	Es la operación AND lógica y permite indicar que se deben introducir dos expresiones X seguida de Y.
e(n|l)campo	Los delimitadores ( ) permite hacer expresiones más complejas. En el ejemplo, el usuario debe introducir el texto "en campo" o "el campo".

A modo de ejemplo, a continuación, se muestran varios ejemplos de expresiones:

- Teléfono (formato 000-000000): `pat = Pattern.compile("[0-9]{3}-[0-9]{6}");`
- DNI: `pat = Pattern.compile("[0-9]{8}-[a-zA-Z]");`
- Provincias andaluzas:
  ```
  pat = Pattern.compile("Almería", "Granada", "Jaén", "Málaga",
                        "Sevilla", "Cádiz", "Córdoba", "Huelva");
  ```

4. **Pasar al evaluador de expresiones el texto a comprobar.**
   ```
   mat = pat.matcher(texto_a_comprobar);
   ```
5. **Se comprueba si hay alguna coincidencia**.
   ```
   if (mat.find()) {
       // Coincide con el patrón
   } else {
       // NO coincide con el patrón
   }
   ```

Ejemplo: validación de un texto de entrada para que cumpla el formato de *DNI*.

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class ValidarDNI {
    public static void validarDNI() {
        String dni;
        Pattern pat;
        Matcher mat;
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduce tu DNI (Formato 00000000-A): ");
        dni = sc.nextLine();
```

```
            pat = Pattern.compile("[0-9]{8}-[a-zA-Z]");
            mat = pat.matcher(dni);
            if (mat.find())
                System.out.println("Correcto!! " + dni);
            else
                System.out.println("El DNI está mal " + dni);
        }
        public static void main(String[] arg) {
            validarDNI();
        }
    }
```

Ejemplo: clase que se puede incorporar a los proyectos basados en la terminal para validar la entrada por teclado.

```
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
/**
 * Clase que contine métodos para leer de forma segura por teclado. Cada método
 * realiza la validación del tipo de dato.
 *
 * @author jl
 * @version 1.0
 */
public class LecturaValidadaPorTeclado {
    Scanner s; // Para leer por teclado
    /**
     * Constructor vacío para la clase. Crea el objeto para leer por teclado.
     */
    public LecturaValidadaPorTeclado() {
        s = new Scanner(System.in);
    }
    /**
     * Método que lee y valida un valor de tipo byte por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo byte
     */
    public byte leerByte(String mensaje) {
        byte x = 0;
        boolean flag = true;
        do {
            try {
                x = Byte.parseByte((String) validar(1, mensaje, "Ingresa un valor de tipo byte."));
                flag = false;
            } catch (NumberFormatException e) {
                System.out.println("Valor byte fuera del rango.");
            }
        } while (flag);
        return x;
    }
    /**
     * Método que lee y valida un valor de tipo short por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo short
     */
    public short leerShort(String mensaje) {
        short x = 0;
        boolean flag = true;
        do {
            try {
                x = Short.parseShort((String) validar(2, mensaje, "Ingresa un valor de tipo short."));
                flag = false;
            } catch (NumberFormatException e) {
                System.out.println("Valor short fuera del rango.");
            }
        } while (flag);
        return x;
    }
    /**
     * Método que lee y valida un valor de tipo int por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo int
```

```
     */
    public int leerInt(String mensaje) {
        int x = 0;
        boolean flag = true;
        do {
            try {
                x = Integer.parseInt((String) validar(3, mensaje, "Ingresa un valor int."));
                flag = false;
            } catch (NumberFormatException e) {
                System.out.println("Valor int fuera del rango.");
            }
        } while (flag);
        return x;
    }
    /**
     * Método que lee y valida un valor de tipo long por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo long
     */
    public long leerLong(String mensaje) {
        long x = 0;
        boolean flag = true;
        do {
            try {
                x = Long.parseLong((String) validar(4, mensaje, "Ingresa un valor long."));
                flag = false;
            } catch (NumberFormatException e) {
                System.out.println("Valor long fuera del rango.");
            }
        } while (flag);
        return x;
    }
    /**
     * Método que lee y valida un valor de tipo char por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo char
     */
    public String leerChar(String mensaje) {
        return (String) validar(5, mensaje, "Ingresa un solo caracter.");
    }
    /**
     * Método que lee y valida un valor de tipo cadena por teclado. Únicamente
     * permite la lectura de letras, números, _ y espacios.
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Cadena de texto que contiene letras, números, _ y espacios
     */
    public String leerString(String mensaje) {
        return (String) validar(6, mensaje,
                "Ingresa una cadena de caracteres que contenga letras, números, _ o espacios.");
    }
    /**
     * Método que lee y valida un valor de tipo booleano por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor booleano (true o false)
     */
    public String leerBoolean(String mensaje) {
        return (String) validar(7, mensaje, "Ingresa un valor booleano: true o false.");
    }
    /**
     * Método que lee y valida un valor de tipo float por teclado
     *
     * @param mensaje String con mensaje a mostrar al usuario
     * @return Valor numérico de tipo float
     */
    public float leerFloat(String mensaje) {
        float x = 0;
        boolean flag = true;
        do {
            try {
                x = Float.parseFloat((String) validar(8, mensaje, "Ingresa un valor float."));
```

```
            flag = false;
        } catch (NumberFormatException e) {
            System.out.println("Valor float fuera del rango.");
        }
    } while (flag);
    return x;
}
/**
 * Método que lee y valida un valor de tipo double por teclado
 *
 * @param mensaje String con mensaje a mostrar al usuario
 * @return Valor numérico de tipo double
 */
public double leerDouble(String mensaje) {
    double x = 0;
    boolean flag = true;
    do {
        try {
            x = Double.parseDouble((String) validar(9, mensaje, "Ingresa un valor double."));
            flag = false;
        } catch (NumberFormatException e) {
            System.out.println("Valor double fuera del rango.");
        }
    } while (flag);
    return x;
}
/**
 * Método que lee y valida un teléfono por teclado
 *
 * @param mensaje String con mensaje a mostrar al usuario
 * @return Cadena de texto con formato teléfono 000 00 00 00
 */
public String leerTelefono(String mensaje) {
    return (String) validar(10, mensaje, "Ingresa un teléfono válido con formato 000 00 00 00.");
}
/**
 * Método que lee y valida una fecha por teclado
 *
 * @param mensaje String con mensaje a mostrar al usuario
 * @return Cadena de texto con formato fecha dd-mm-YYYY
 */
public String leerFecha(String mensaje) {
    return (String) validar(11, mensaje, "Ingresa una fecha válida con formato dd-mm-yyyy.");
}
/**
 * Método que lee y valida un NIF por teclado
 *
 * @param mensaje String con mensaje a mostrar al usuario
 * @return Cadena de texto con formato NIF 00000000-A
 */
public String leerNIF(String mensaje) {
    return (String) validar(12, mensaje, "Ingresa un NIF válido con formato 00000000-A.");
}
/**
 * Método que lee y valida un e-mail por teclado
 *
 * @param mensaje String con mensaje a mostrar al usuario
 * @return Cadena de texto con formato e-mail aaaa@bbbbb.ccc
 */
public String leerEMail(String mensaje) {
    return (String) validar(13, mensaje, "Ingresa un e-mail válido con formato aaaaa@bbbbb.ccc");
}
/**
 * Método que permite validad una cadena de caracteres según el tipo indicado
 *
 * @param v          Tipo del dato a validar: 1:byte, 2: short, 3:int, 4:long, 5:char, 6:cadena de texto,
 *                   7:booleano, 8:float, 9:double, 10:teléfono, 11:fecha, 12:nif y 13:e-mail
 * @param mensaje    String con mensaje a mostrar al usuario
 * @param validacion String con mensaje sobre formato del campo a validar
 * @return Object Objeto validado
 */
private Object validar(int v, String mensaje, String validacion) {
    boolean flag = true;
```

```
        CharSequence ob;
        Pattern pat = null;
        Matcher mat = null;
        switch (v) {
            case 1, 2, 3, 4 -> // byte, short, int y long
                pat = Pattern.compile("^[+-]?\\d+$");
            case 5 -> // char
                pat = Pattern.compile("^.$");
            case 6 -> // Cadena con letras, números, _ y espacios
                pat = Pattern.compile("^[a-zA-Z0-9_ ]+$");
            case 7 -> // boolean
                pat = Pattern.compile("^true|false$");
            case 8, 9 -> // float y double
                pat = Pattern.compile("[+-]?(\\d+|\\d+\\.\\d+|\\.\\d+|\\d+\\.)([eE]\\d+)?");
            case 10 -> // Teléfono
                pat = Pattern.compile("^[0-9]{3} [0-9]{2} [0-9]{2} [0-9]{2}$");
            case 11 -> // Fecha
                pat = Pattern.compile("^(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])/[0-9]{4}$");
            case 12 -> // NIF
                pat = Pattern.compile("^[0-9]{8}-[a-zA-Z]$");
            case 13 -> // E-mail
                pat = Pattern.compile("^[\\w!#$%&'*+/=?`{|}~^-]+(?:\\.[\\w!#$%&'*+/=?`{|}~^-]+)*@(?:[a-zA-Z0-9-
]+\\.)+[a-zA-Z]{2,6}$");
        }
        do {
            System.out.print(mensaje + ": ");
            ob = s.nextLine();
            if (pat != null)
                mat = pat.matcher(ob);
            if (mat != null && mat.matches())
                flag = false;
        } while (flag);
        return ob;
    }
}
```

TAREA

Crear una *BD* que contenga al menos una tabla con entre 6 y 7 campos de diferentes tipos de datos, como enteros, fechas, cadenas de caracteres, números decimales, correos electrónicos y valores booleanos. La elección de la tabla y los campos es libre, siempre y cuando se cumplan los requerimientos dados. Ejemplos: trabajadores, facturas, etc.

9. **Implementar una aplicación que se conecte con la *BD* anterior (utilizando *MariaDB*) y cumpla con los siguientes requisitos:**
   - a) **La conexión con la *BD* debe realizarse de manera transparente para el usuario. Los parámetros de conexión deben configurarse a través de un fichero de configuración (por ejemplo, un fichero `.properties`).**
   - b) **La aplicación debe permitir al usuario registrar nuevos datos, modificar los existentes y eliminar registros.**
   - c) **Todos los datos ingresados por el usuario deben ser validados utilizando una clase de validación (ya sea la propuesta en este punto o una clase nueva que se desarrolle específicamente para este propósito).**
   - d) **La aplicación debe incluir una opción que muestre todos los registros almacenados en la *BD*.**
   - e) **Implementa un sistema de autenticación de usuarios, que deberá estar respaldado por una tabla de usuarios en la *BD*.**
   - f) **Agregar una funcionalidad de búsqueda que permita al usuario localizar registros específicos.**
   - g) **Ejecutar al menos dos funciones almacenadas que realicen tareas útiles para la aplicación.**
   - h) **Ejecutar dos procedimientos almacenados que ejecuten tareas relevantes dentro de la aplicación.**
   - i) **Usar transacciones para realizar alguna operación compleja (de alta, baja o modificación).**
   - j) **La aplicación debe ser capaz de conectarse también a *BD* en *PostgreSQL* o *SQLite*.**
   - k) **Añadir soporte para multiusuario, implementando lo necesario para que varios usuarios simultáneos puedan trabajar con la aplicación sin que se produzcan problemas (por ejemplo, que dos usuarios estén modificando el mismo elemento).**
   - l) **Añadir una opción que permita recuperar el último elemento borrado en la *BD*.**
   - m) **Incluir una opción que permita eliminar todos los datos almacenados en la *BD* desde la aplicación.**
   - n) **Implementar una opción que permita al usuario cerrar la aplicación de forma segura.**

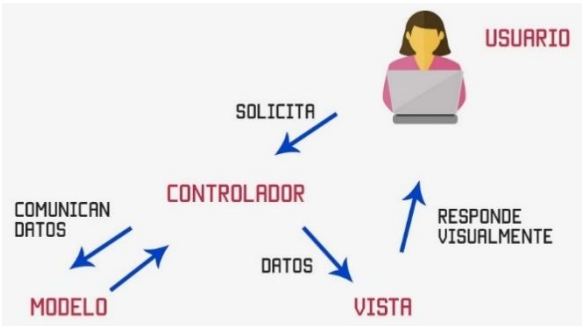
12 Patrón arquitectónico *MVC*.

El patrón *MVC* (*Modelo* – *Vista* – *Controlador*), es un patrón de arquitectura de software que **separa los datos de una aplicación y la interfaz que ve el usuario de la lógica del negocio**. Es más frecuente en aplicaciones *Web* que en aplicaciones de escritorio, sin embargo, es aplicable también a este, sin ningún problema, *Java* ya contaba con *Observer* y *Observable*, herramientas que ayudan a la interacción entre la interfaz y el modelo.

La descripción del patrón *MVC* es:

- ✓ **Vista** (*View*): representa la **interfaz de usuario** y todas las herramientas con las cuales el usuario hace uso del programa.
- ✓ **Modelo** (*Model*): es donde esta toda la **lógica del negocio**, la representación de todo el sistema incluido la interacción con una *BD*, si es que el programa así lo requiere.
- ✓ **Controlador** (*Controller*): este componente es el que **responde a la interacción (eventos) que hace el usuario en la interfaz y realiza las peticiones al modelo para pasar estos a la vista**.

Flujo de control:



1. El usuario realiza una acción en la interfaz.
2. El controlador trata el evento de entrada.
   - Previamente se ha registrado.
3. El controlador notifica al modelo la acción del usuario, lo que puede implicar un cambio del estado del modelo (si no es una mera consulta).
4. Se genera una nueva vista. La vista toma los datos del modelo.
   - El modelo no tiene conocimiento directo de la vista.
5. La interfaz de usuario espera otra interacción del usuario, que comenzará otro nuevo ciclo.

El patrón de diseño *MVC*, **permite separar los componentes de la aplicación dependiendo de la responsabilidad que tienen**, esto significa que cuando se hace un cambio en alguna parte del código, esto no afecta a otra parte del mismo. Por ejemplo, si se modifica la *BD*, sólo se debería modificar el modelo que es quién se encarga de los datos y el resto de la aplicación debería permanecer intacta. Esto **respeta el principio de la responsabilidad única**. Es decir, una parte del código no debe de saber qué es lo que hace toda la aplicación, sólo debe de tener una responsabilidad.

Este patrón es uno de los más usados, en la actualidad se puede encontrar tanto en pequeños como en grandes sistemas, en el mundo laboral es indispensable llevarlo a la práctica.

12.1 *CRUD* con validación usando patrones *MVC* y *DAO* (*Java Swing*).

En *Moodle Centros* se dispone de un archivo comprimido que contiene todo el código del proyecto de ejemplo (`CrudEmpleadosMVC.zip`).

Para que funcione correctamente se deben ejecutar las siguientes sentencias *SQL* en *MariaDB* (crea la *BD*, usuario y tabla necesaria).

```
DROP USER IF EXISTS empleados_user;
CREATE USER empleados_user IDENTIFIED BY "empleados_password";
CREATE OR REPLACE DATABASE empleados COLLATE utf8mb4_spanish_ci;
GRANT ALL PRIVILEGES ON empleados.* TO empleados_user;
USE empleados;
CREATE TABLE empleado (
     nif VARCHAR(9) PRIMARY KEY,
     nombre VARCHAR(20) NOT NULL,
     apellidos VARCHAR(20) NOT NULL,
     direccion VARCHAR(50),
     telefono VARCHAR(9),
     e_mail VARCHAR(50),
     sueldo DECIMAL(8,2),
     fecha_alta DATE DEFAULT CURRENT_DATE
);
```

#	Nombre	Tipo de datos	Longitud/Con...	Sin signo	Permitir NULL	Rellen...	Predeterminado
1	**nif**	**VARCHAR**	**10**	☐	☐	☐	**Sin valor predet...**
2	nombre	VARCHAR	20	☐	☐	☐	Sin valor predeter...
3	apellidos	VARCHAR	20	☐	☐	☐	Sin valor predeter...
4	direccion	VARCHAR	50	☐	☑	☐	NULL
5	telefono	VARCHAR	15	☐	☑	☐	NULL
6	e_mail	VARCHAR	50	☐	☑	☐	NULL
7	sueldo	DECIMAL	8,2	☐	☑	☐	NULL
8	fecha_alta	DATE		☐	☑	☐	curdate()

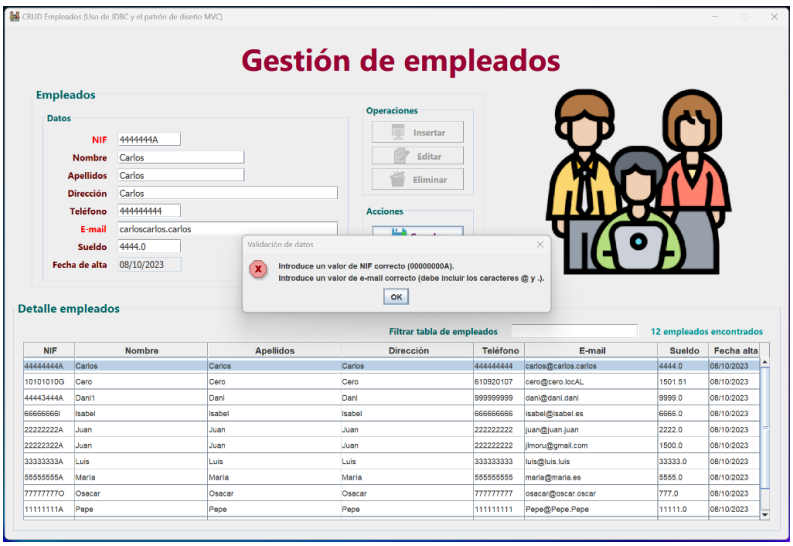
La aplicación maneja un archivo de configuración (`config.properties`) que permite definir varios parámetros de configuración (en caso de no existir se utilizarán los definidos por defecto en el código):

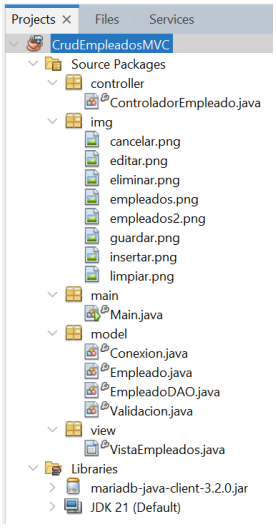
```
url=jdbc:mariadb://localhost:3306/empleados
user=empleados_user
password=empleados_password
```

Este proyecto está realizado utilizando *NetBeans* y *Ant*.

Para la ejecución de las sentencias *SQL* se ha hecho uso de `PreparedStatement` para evitar la inyección *SQL*. Para la validación de datos se utilizan las clases `Matcher` y `Pattern`.

La interfaz gráfica que presenta la aplicación se muestra en las imágenes.





PROYECTO : Gestión de productos con *JavaFX* y *Maven/Gradle*.

Desarrollar una aplicación de escritorio para la gestión de productos, implementada en *JavaFX* y utilizando *Maven* o *Gradle* como herramientas de construcción. La aplicación debe seguir el patrón de diseño *DAO* o *Repository* para la gestión de datos y permitir el mantenimiento de una tabla de productos con las siguientes características:

- ✓ **ID del producto: identificador único para cada producto.**
- ✓ **Nombre del producto: descripción breve y precisa del producto.**
- ✓ **Descripción: detalle adicional sobre las características del producto.**
- ✓ **Precio: valor monetario del producto.**
- ✓ **Stock disponible: cantidad de unidades disponibles en inventario.**
- ✓ **Fecha de alta: fecha en la que el producto fue añadido al inventario.**
- ✓ **Descuento: cualquier oferta o reducción de precio aplicable al producto.**
- ✓ **Peso: peso del producto en Kilogramos.**
- ✓ **Dimensiones: medidas físicas del producto (largo, ancho, alto).**
- ✓ **Imagen: *URL* o referencia a la imagen del producto.**

Requisitos adicionales:

- **Implementar un sistema de validación de datos para asegurar la integridad de la información ingresada.**
- **La interfaz debe permitir *CRUD* (crear, leer, actualizar, eliminar) de los productos en un formato intuitivo y visualmente atractivo.**
- **Se valorará positivamente la inclusión de mejoras opcionales, tales como un formulario de *login* para controlar el acceso a la aplicación.**

El proyecto debe estar estructurado de manera clara, siguiendo las mejores prácticas de desarrollo y diseño de software.