UT2. Parte 1: Conectores para bases de datos relacionales. JDBC.

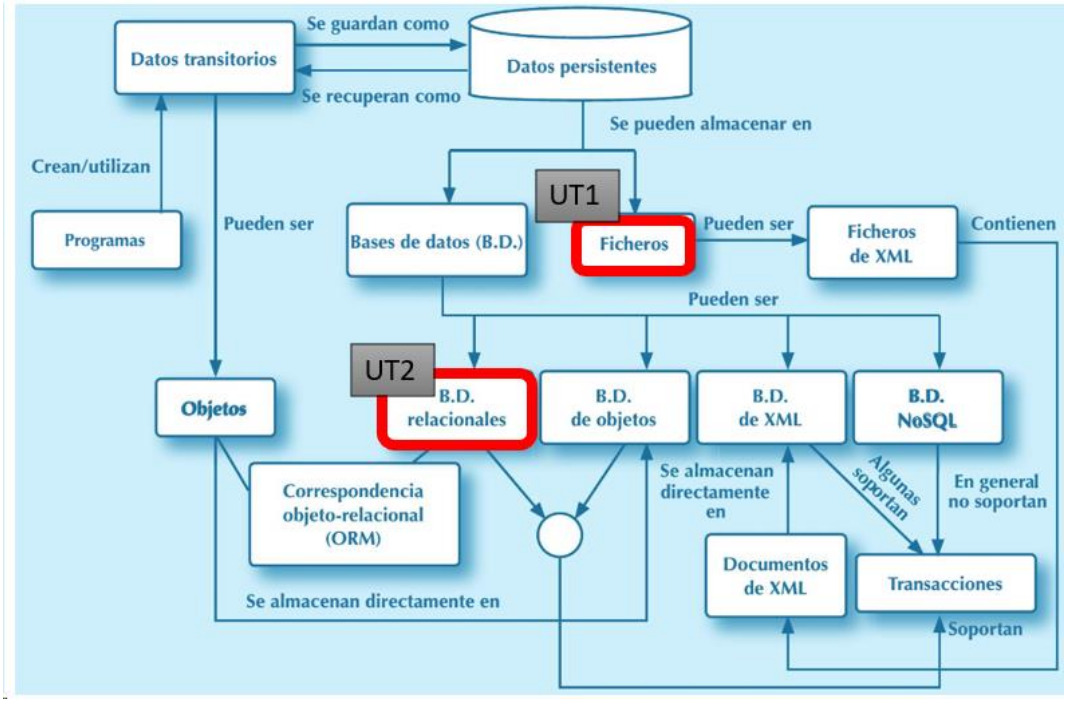
Tabla de contenido

1. Introducción

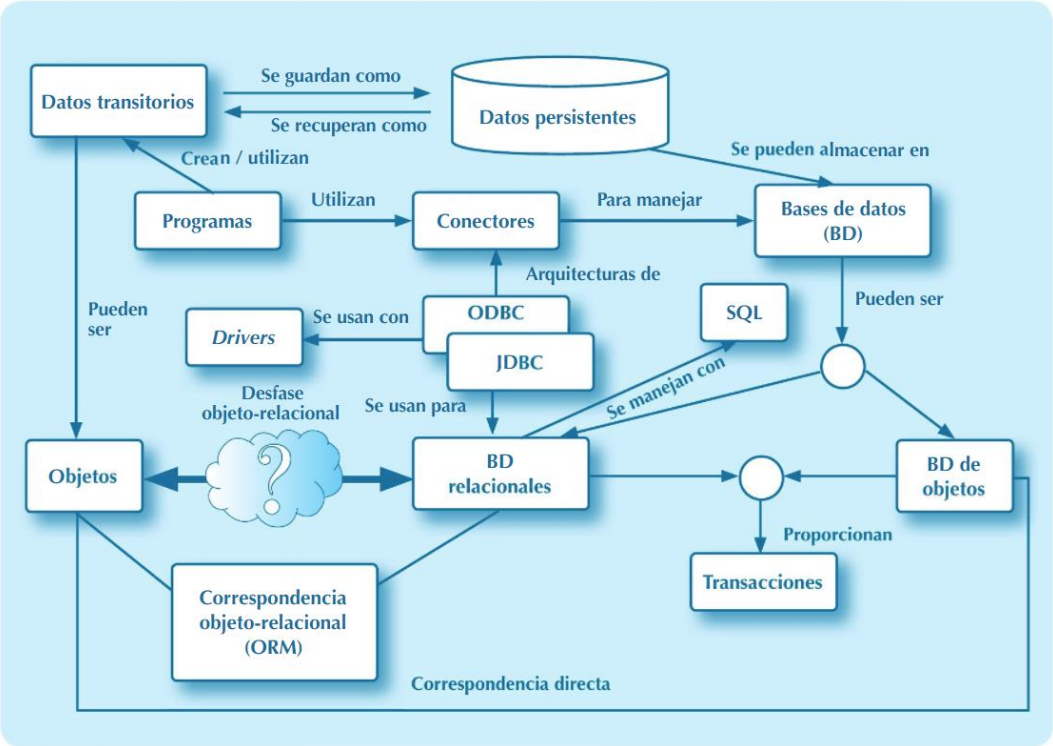
En esta unidad vamos a ver:

- Cómo conectarnos a bases de datos desde nuestros programas Java.
- Cómo realizar operaciones básicas (***CRUD***, Create, Read, Update, Delete) sobre bases de datos utilizando Java y conectores JDBC (para SQLite y MySQL).

Recordatorio esquema general del módulo



Esquema de la Unidad 2



Recordatorio: desventajas de los ficheros

- Si hay que realizar **consultas complejas** o que requieren relacionar mucha información diversa, será difícil escribir un programa para realizarlas.
- Si el **volumen de datos** para manejar es muy grande, o si es necesario realizar con mucha frecuencia **operaciones de borrado o modificación** de datos, el rendimiento será muy pobre.
- **Concurrencia**: Permitir que varios programas realicen a la vez operaciones de consulta / actualización puede introducir inconsistencias en los datos e incluso dañar los ficheros.
- Es difícil establecer y hacer que se cumplan **restricciones de integridad** sobre los datos.
- También es difícil evitar redundancias en los datos:
    - Se desperdicia espacio de almacenamiento.
    - Pueden surgir inconsistencias cuando se añaden o modifican datos : si la misma información está en más de un lugar, puede acabar teniendo un valor distinto en cada uno.

Persistencia de datos en Bases de Datos

- Ventajas de las BBDD:
    - Control sobre la redundancia de los datos
    - Consistencia de los datos
    - Información compartida (concurrencia)
    - Mantenimiento de estándares
    - Operaciones de borrado, inserción, modificación “sencillas”
    - Admiten volúmenes de datos grandes

2. Bases de datos relacionales y POO: desfase objeto-relacional

El desfase objeto-relacional (también conocido como impedancia objeto-relacional o Object-Relational Impedance Mismatch en inglés) se refiere a la desalineación conceptual y estructural entre los modelos de datos utilizados en la programación orientada a objetos (POO) y los modelos relacionales empleados en las bases de datos.

Los sistemas orientados a objetos y las bases de datos relacionales gestionan los datos de maneras fundamentalmente diferentes, lo que puede crear dificultades cuando intentamos hacer que ambos trabajen juntos. Algunas de las principales diferencias que contribuyen a este desfase son:

- **Representación de los datos**:
    - En la POO, los datos se modelan como **objetos**, que pueden tener atributos (campos) y comportamientos (métodos).
    - En las bases de datos relacionales, los datos se modelan en **tablas** de filas y columnas, sin comportamientos, solo con valores (aunque pueden estar normalizadas o estructuradas en varias tablas).
- **Herencia**:
    - En los sistemas orientados a objetos, los objetos pueden heredar propiedades y comportamientos de otras clases (herencia).
    - En las bases de datos relacionales, no existe un concepto directo de herencia; la información de una jerarquía de clases debe ser modelada en varias tablas, lo que puede hacer que la relación entre los datos no sea tan sencilla.
- **Relaciones**:
    - En un modelo orientado a objetos, las relaciones entre objetos son generalmente directas y pueden representarse como referencias entre instancias de objetos (por ejemplo, a través de punteros o referencias).
    - En una base de datos relacional, las relaciones entre los datos se gestionan a través de claves ajenas y deben ser manejadas de forma explícita con SQL.

Acceso a BBDD relacionales desde aplicaciones en lenguajes orientados a objetos

Las dos opciones principales para acceder a bases de datos relacionales desde aplicaciones en lenguajes orientados a objetos (como Java) son conectores (o drivers) y herramientas ORM.

- **Conectores**
    - Los conectores son componentes de software que permiten a una aplicación conectar directamente con una base de datos y ejecutar sentencias SQL. Con los conectores, el desarrollador trabaja con consultas SQL directamente en el código. Esto implica un control directo sobre las consultas y la estructura de la base de datos, pero por sí mismos no ofrecen un mapeo directo entre objetos y las tablas de una base de datos, por lo que el desarrollador tiene que escribir el código para gestionar el desfase objeto-relacional.
    - En el caso de Java, esto se realiza mediante **JDBC** (Java Database Connectivity), que proporciona una API estándar para conectar y ejecutar consultas en bases de datos relacionales.
- **Frameworks ORM (Mapeo Objeto-Relacional)**
    - El proceso de convertir un objeto en una fila de una tabla y viceversa se llama mapeo objeto-relacional (ORM, por sus siglas en inglés). Este mapeo puede ser complejo, especialmente cuando

se manejan relaciones más complejas entre objetos o tablas, como relaciones uno a muchos o muchos a muchos.

- Para facilitar este proceso, existen frameworks ORM, herramientas que simplifican el acceso y gestión de bases de datos relacionales al permitir trabajar con objetos en lugar de directamente con tablas y SQL.

- En lugar de escribir SQL, el desarrollador mapea las clases de su aplicación con las tablas de la base de datos y trabaja con estas clases para realizar operaciones CRUD. El framework ORM traduce automáticamente las operaciones en las consultas SQL adecuadas y gestiona las relaciones entre objetos.

- Un ejemplo de framework ORM para Java es **Hibernate**, que implementa la especificación de **JPA** (Java Persistence API).

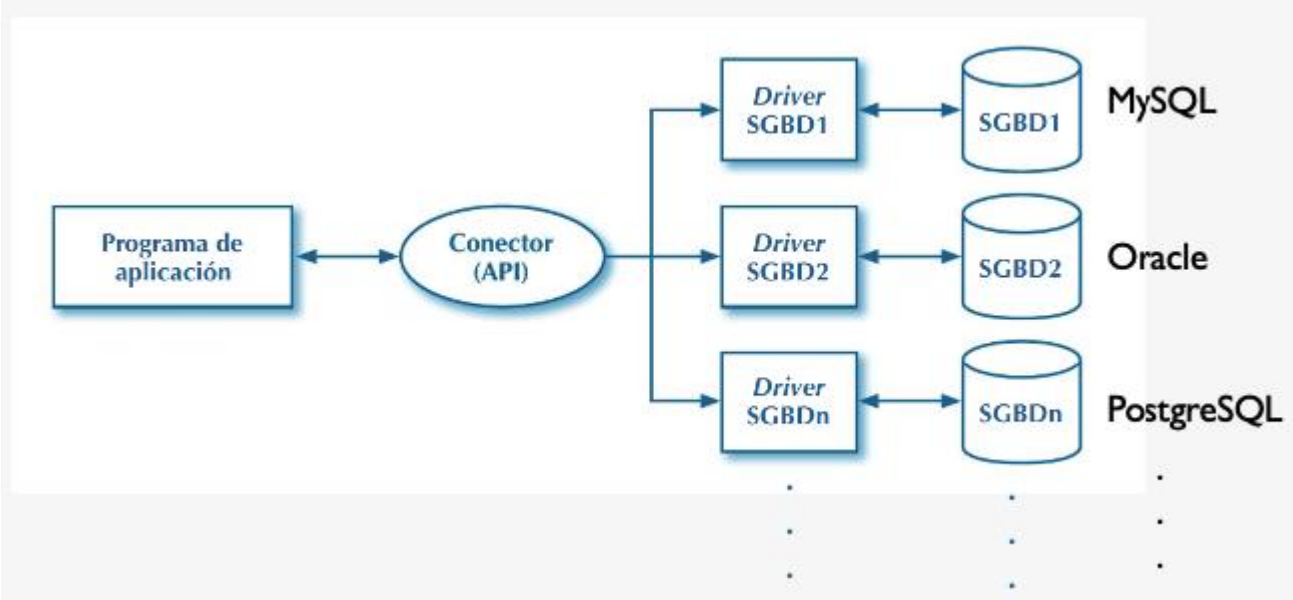
3. Conectores

- Los **conectores** son componentes o herramientas que permiten establecer una conexión entre una aplicación y una base de datos. Actúan como un intermediario para facilitar la comunicación y el intercambio de datos entre ambos, implementando una interfaz que estandariza la forma en que la aplicación puede interactuar con la base de datos.
- No sólo sirven para bases de datos relacionales, sino también para sistemas basados en ficheros que almacenan los datos en formatos tabulares, como CSV y hojas de cálculo, o en estructuras jerárquicas, como XML.

Programas de aplicación	Conectores (APIs)	Sistemas gestores de bases de datos (SGBD)
Se escriben con lenguajes de programación de propósito general, como por ejemplo Java.	Para que los programas de aplicación puedan interactuar con los SGBD, se necesitan mecanismos que permitan esta interacción. Estos mecanismos se implementan en APIs y se denominan **conectores**.	Relacionales, XML, de objetos o de otros tipos: tienen sus propios lenguajes especializados para operar con los datos que almacenan

Conectores para Bases de Datos relacionales

- Los sistemas de bases de datos relacionales son actualmente los más utilizados.
- Para trabajar con ellos se utiliza el lenguaje SQL.
- SQL es un estándar: American National Standards Institute (ANSI), 1986 / ISO (International Organization for Standardization, 1987).
- Pero existen multitud de SGBD distintos, y cada uno usa su propia versión de SQL. Además, cada base de datos tiene sus propias interfaces de bajo nivel, por lo que se requiere el uso de **drivers** específicos para cada SGBD.
- Con los conectores, el uso de una API y de drivers específicos permite desarrollar una arquitectura genérica: la API proporciona una interfaz común tanto para las aplicaciones como para las distintas bases de datos, mientras que los drivers, proporcionados por los fabricantes de los SGBD, se encargan de gestionar las particularidades de cada base de datos.



- A través de los conectores es posible realizar las operaciones principales en una base de datos:
  - **Conexión**: Iniciar una sesión con la base de datos para habilitar la comunicación y el envío de comandos.
  - **Escritura** (**C**reate): Añadir nuevos datos a la base de datos usando comandos como INSERT.
  - **Lectura** (**R**ead): Realizar consultas para obtener datos, generalmente mediante sentencias SELECT.
  - **Actualización** (**U**pdate): Modificar datos existentes en la base de datos con sentencias UPDATE.
  - **Eliminación** (**D**elete): Borrar datos de la base de datos mediante comandos DELETE.
  - **Desconexión**: Finalizar la sesión y liberar los recursos asignados a la conexión.

  Estas operaciones se gestionan mediante el uso de instrucciones SQL que el conector interpreta y ejecuta en la base de datos, garantizando la consistencia y seguridad en el manejo de los datos.

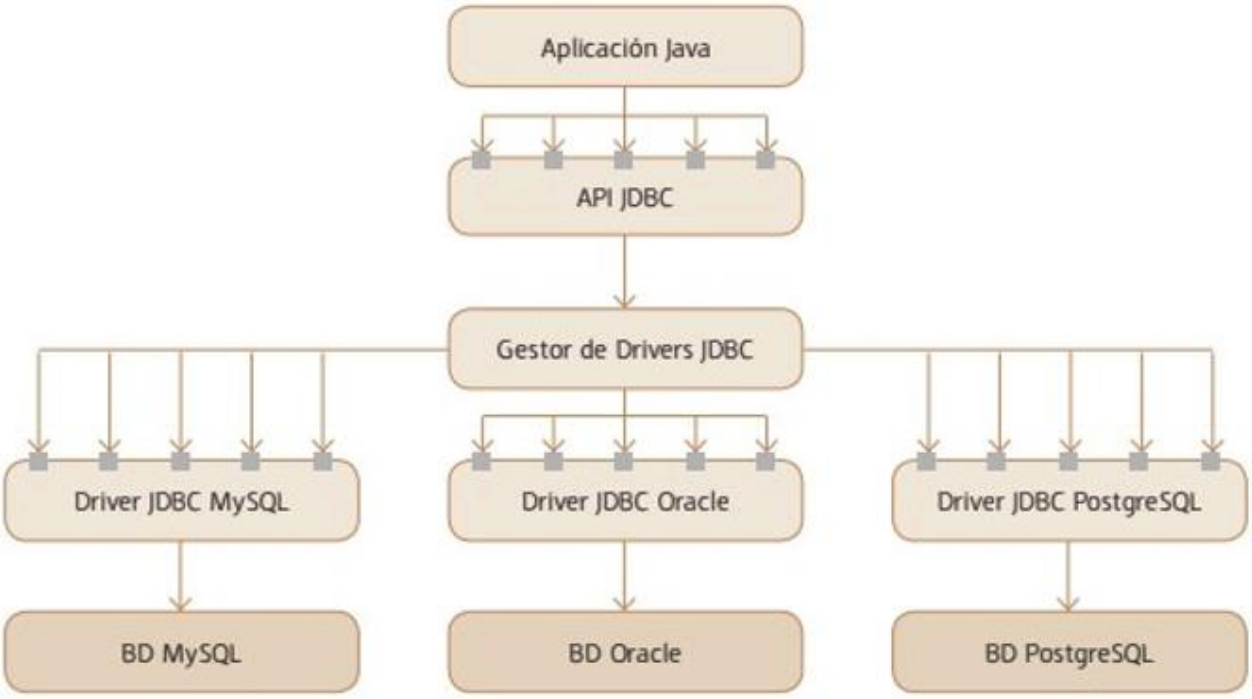
- Ejemplos de arquitecturas de conectores:
    - **ODBC** (Open DataBase Connectivity): desarrollada por Microsoft a principios de los años 90. API para el lenguaje C. Para entornos Windows y Linux/Unix. Sucesores de ODBC:
        - OLE-DB (sólo Windows)
        - ADO (ActiveX Data Objects) (sólo Windows)
    - **JDBC** (Java DataBase Connectivity): desarrollada por Sun Microsystems (ahora Oracle) a finales de los años 90. API para lenguaje Java.

- Existen también drivers de JDBC no para conectar a una base de datos específica, sino a otro conector. En particular, existe un driver de JDBC para ODBC: permite acceder a una base de datos que no soporte JDBC pero sí ODBC.

4. Conector JDBC. Clases e interfaces

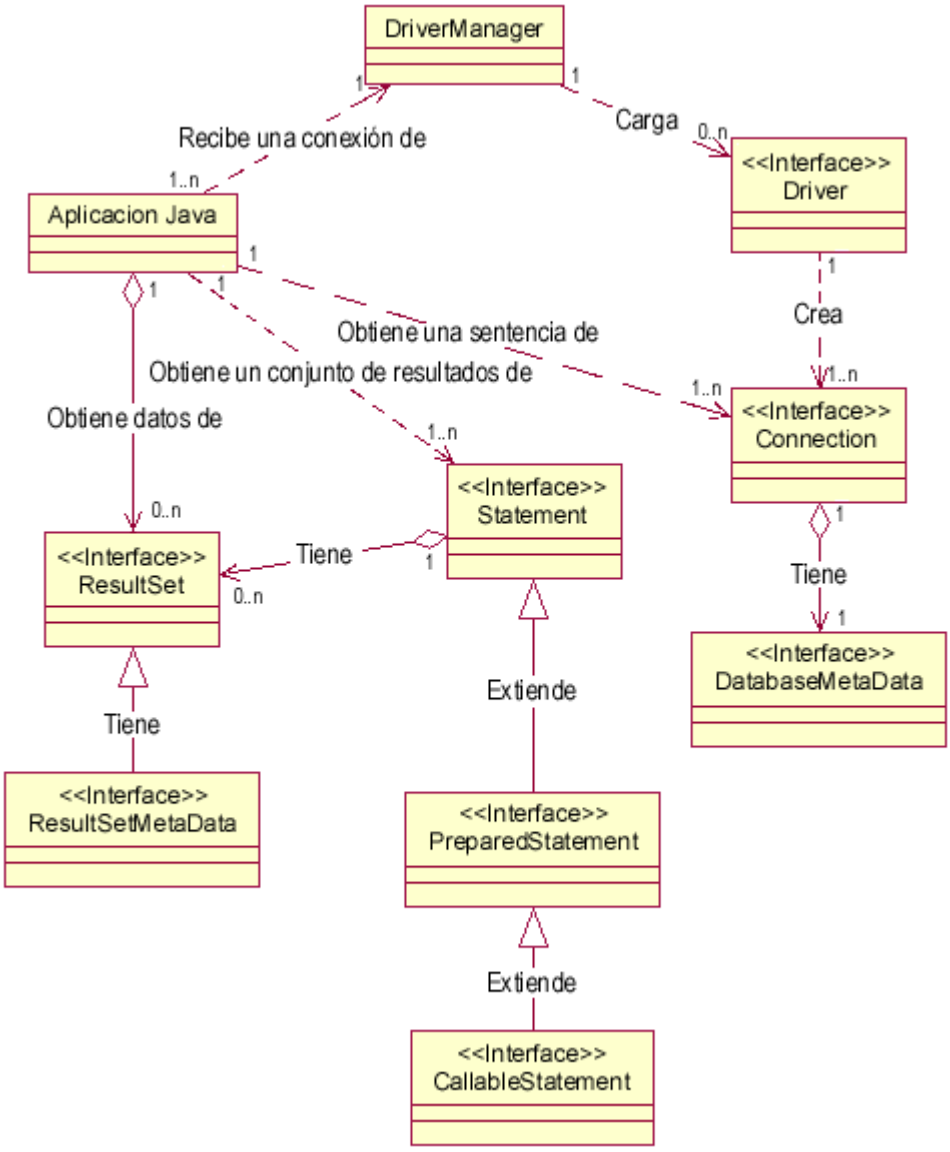
- **JDBC** (Java DataBase Connectivity) es una API que proporciona acceso uniforme a bases de datos relacionales mediante el lenguaje de programación Java.

- Arquitectura genérica JDBC:



- Para poder hacer uso de JDBC en nuestros proyectos Java, lo primero que tendremos que hacer es añadir el conector/driver (en ocasiones se utilizan como sinónimos, aunque estrictamente no lo sean) correspondiente para la base de datos particular con la que queramos trabajar, ya sea utilizando un fichero con extensión **.jar** o especificando los datos necesarios en una herramienta de automatización de construcción, como Maven, Gradle o Ant.

- Una vez añadido el conector, podremos hacer uso de la jerarquía de clases e interfaces que proporciona el paquete **java.sql**, del que podemos ver un esquema simplificado en la siguiente imagen:



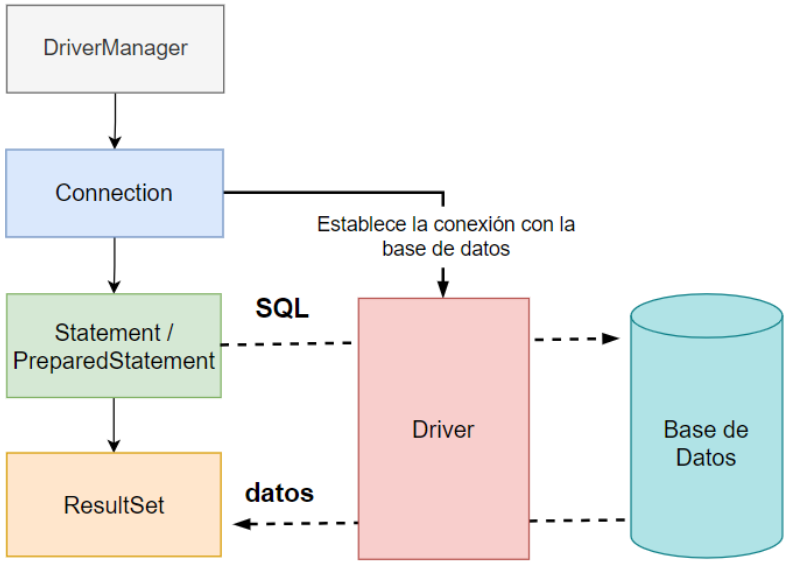
Clases e Interfaces Principales de JDBC en el paquete java.sql

- **DriverManager:** esta clase administra un conjunto de controladores de base de datos y permite establecer conexiones con bases de datos a través de los métodos getConnection().
- **Connection**: representa una conexión a una base de datos específica. A través de una instancia de Connection, se pueden crear objetos Statement y manejar transacciones.
- **Statement**: se utiliza para ejecutar consultas SQL estáticas y devolver resultados a partir de la base de datos. Subclases:
    - **PreparedStatement**: extiende Statement y se utiliza para ejecutar consultas parametrizadas (declaraciones SQL precompiladas con parámetros).
    - **CallableStatement**: extiende PreparedStatement y se utiliza para ejecutar procedimientos almacenados en la base de datos.
- **ResultSet**: interfaz que representa el conjunto de resultados de una consulta SQL y permite recorrer los datos recuperados. Mantiene un cursor que permite iterar sobre los resultados de una consulta SQL.

- **ResultSetMetaData**: interfaz que proporciona información (metadatos) sobre los tipos y propiedades de las columnas en un ResultSet, permitiendo obtener información sobre el tipo y el número de columnas.
- **DatabaseMetaData**: interfaz que proporciona métodos para obtener información (metadatos) sobre la base de datos en sí, como su versión, características, tablas, procedimientos almacenados, etc.
- **Driver**: interfaz que define los métodos que deben implementar los controladores JDBC para conectar Java con distintos sistemas de bases de datos. Cada proveedor de base de datos (como MySQL, PostgreSQL, Oracle, etc.) ofrece su propia implementación de Driver para que las aplicaciones Java puedan conectarse a sus sistemas.

Funcionamiento de JDBC

La siguiente figura muestra las 4 clases principales que usa un programa Java para comunicarse con una base de datos mediante JDBC para le ejecución de sentencias SQL:



Funcionamiento de JDBC

Clase DriverManager e interfaz Connection

- **DriverManager**: esta clase se utiliza para registrar en nuestro programa el controlador (driver) para un tipo de base de datos específico (SQLite, MySQL, Oracle,...) y para establecer una conexión con el servidor de base de datos a través de su método **`getConnection()`**.
- **Connection:** representa una conexión con una base de datos. Una aplicación puede tener más de una conexión y con más de una base de datos. Permite crear sentencias para ejecutar consultas, recuperar resultados, obtener metadatos sobre la base de datos, etc.

  Algunos métodos de Connection son:
  - **`String getCatalog()`**: devuelve el catálogo (base de datos) del objeto Connection, es decir, el nombre de la base de datos a la que estamos conectados, o null si no estamos conectados a ninguna.
  - **`Statement createStatement:`** crea un objeto Statement, que permitirá enviar consultas a la base de datos.
  - **`PreparedStatement prepareStatement(String sql):`** crea un objeto PreparedStatement, que permitirá enviar consultas parametrizadas a la base de datos. El String **sql** es una sentencia SQL que puede contener uno o más placeholder o marcadores de posición de parámetros (representados con el símbolo **?**)

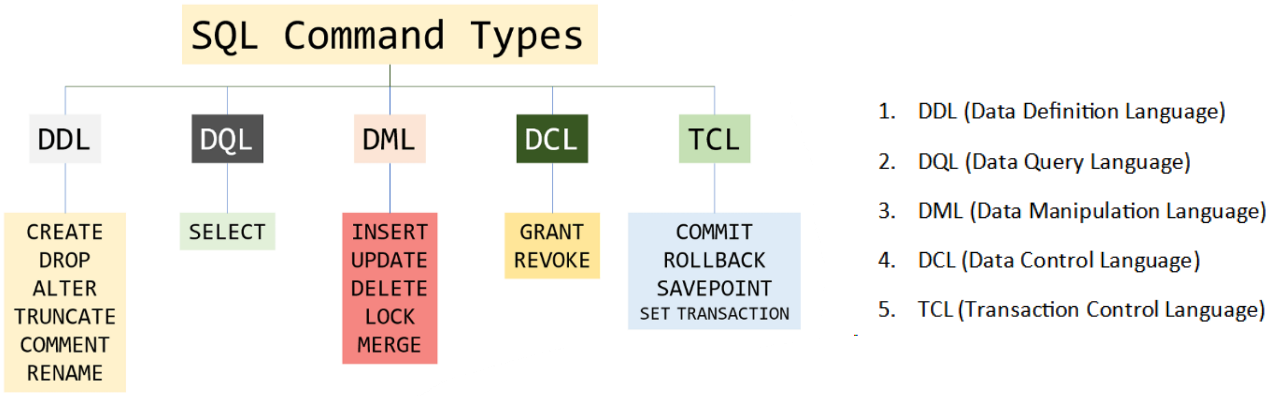
Interfaz Statement

La interfaz `Statement` de `java.sql` se utiliza para ejecutar instrucciones SQL en una base de datos a través de una conexión JDBC (Java Database Connectivity). Proporciona un mecanismo para enviar consultas SQL a la base de datos y recuperar, si corresponde, resultados. Es uno de los elementos fundamentales para interactuar con bases de datos desde una aplicación Java.

Algunas de las ***tareas y operaciones principales*** para las que se utiliza la interfaz `Statement` son:

- *Ejecución de consultas SQL*: Podemos utilizar `Statement` para ejecutar consultas SQL, como `SELECT`, `INSERT`, `UPDATE` o `DELETE`, enviando la instrucción SQL a la base de datos para su ejecución.
- *Creación de tablas y otros objetos*: También podemos utilizar `Statement` para crear tablas, vistas, índices y otros objetos en la base de datos, utilizando comandos SQL de creación.
- *Gestión de transacciones*: Podemos iniciar y finalizar transacciones utilizando métodos de `Statement`. Esto nos permite controlar cómo se aplican las operaciones en la base de datos y cómo se confirman (*commit*) o revierten (*rollback*) los cambios.
- *Obtención de resultados*: `Statement` permite recuperar resultados de las consultas `SELECT` en forma de conjuntos de resultados (result sets). Podemos usar métodos como `executeQuery` para obtener datos de la base de datos y trabajar con ellos en nuestra aplicación.

  Recordatorio clasificación de sentencias SQL:



Estos son algunos de los métodos de la interfaz Statement. Para más información: https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html

Métodos de Statement	**Descripción**
`ResultSet` **`executeQuery`**`(String sql)`	Ejecuta una consulta SQL (SELECT), que devuelve un único ResultSet.
`int` **`executeUpdate`**`(String sql)`	Se utiliza para sentencias que no devuelven un ResultSet, como son las sentencias de modificación o DML (INSERT, UPDATE, DELETE), y las sentencias de definición de datos o DDL(CREATE, DROP, ALTER). Devuelve el número de filas afectadas o en el caso de sentencias DDL devuelve 0.

Métodos de Statement	Descripción
`boolean` **`execute`**`(String sql)`	Se puede utilizar para ejecutar cualquier sentencia SQL, tanto las que devuelven un ResultSet (como SELECT), como las que devuelven un número de filas afectadas (por ejemplo INSERT, UPDATE, DELETE) y para las de definición de datos (como CREATE, DROP, ALTER). El método devuelve true si obtiene un ResultSet (para recuperar las filas será necesario llamar al método `getResultSet()` a continuación), y false si se trata de un recuento de actualizaciones o no hay resultados (en cuyo caso se usará el método `getUpdateCount()` para recuperar el valor devuelto).
`ResultSet` **`getResultSet`**`()`	Devuelve el resultado actual como un ResultSet, o null si el resultado actual no es un ResultSet
`int` **`getUpdateCount`**`()`	Devuelve el número de filas afectadas por un update, o -1 si el resultado actual es un ResultSet.

Ejemplo: uso de executeUpdate() para insertar una fila en una tabla

```
String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
String user = "mi_usuario";
String password = "mi_contraseña";

Connection conn = null;
Statement stmt = null;

try {
    // Establecer conexión
    conn = DriverManager.getConnection(url, user, password);

    // Crear Statement
    stmt = conn.createStatement();

    // Sentencia SQL de inserción
    String sql = "INSERT INTO empleados (id, nombre, puesto) " +
                 "VALUES (1, 'Ana Pérez', 'Desarrolladora')";

    // Ejecutar la sentencia
    int filasAfectadas = stmt.executeUpdate(sql);

    // Mostrar resultado
    System.out.println("Filas insertadas: " + filasAfectadas);

} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (stmt != null) stmt.close();
        if (conn != null) conn.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Interfaz PreparedStatement

- La interfaz PreparedStatement se utiliza para ejecutar consultas SQL parametrizadas.
- PreparedStatement precompila la consulta, lo que puede mejorar el rendimiento si vamos a realizar múltiples inserciones con la misma estructura de consulta, ya que la consulta se compila una vez y se puede reutilizar con diferentes valores.
- Se construye una consulta SQL que incluye placeholders o marcadores de posición, representados con el carácter '**?**', que representan los parámetros que serán asignados más adelante. Cada placeholder tiene un índice que se corresponde con su posición en la sentencia, empezando por 1
- Ejemplo: sentencia con 3 parámetros, de índices 1, 2 y 3

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

- Los métodos para ejecutar consultas son los mismos que los de Statement, pero no se les pasa ningún String con la orden SQL, ya que esto lo hace el método prepareStatement(String) de la interfaz Connection, cuando creamos el objeto PreparedStatement.
- Antes de ejecutar un PreparedStatement es necesario asignar los datos a los parámetros. Esto se hace con métodos set de la clase PreparedStament, que tienen la forma:

  `void setXXX(int indiceParametro, tipoJAva valor)`

  Por ejemplo:

```
PreparedStatement sentencia = conexion.prepareStatement(sql);
sentencia.setInt(1, Integer.parseInt(dep)); //num departamento
sentencia.setString(2, dnombre);           // nombre
sentencia.setString(3, loc);               //localidad
//Ahora ejecutamos la consulta
int filas = sentencia.executeUpdate();
```

Estos son algunos de los métodos de la interfaz PreparedStatement. Para más información:
https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html

Métodos de PreparedStatement	**Descripción**
`ResultSet` **`executeQuery`**`()`	Ejecuta la consulta SQL en este objeto PreparedStatement y devuelve el objeto ResultSet generado por la consulta.
`int` **`executeUpdate`**`()`	Ejecuta la sentencia SQL en este objeto PreparedStatement, que debe ser una sentencia DML, como INSERT, UPDATE o DELETE; o una sentencia SQL que no devuelve nada, como una sentencia DDL. Devuelve el número de filas afectadas o en el caso de sentencias DDL devuelve 0.
`boolean` **`execute`**`()`	Ejecuta la sentencia SQL en este objeto PreparedStatement, que puede ser cualquier tipo de sentencia SQL (como en la interfaz Statement). El método devuelve true si obtiene un ResultSet (para recuperar las filas será necesario llamar al método `getResultSet()` a

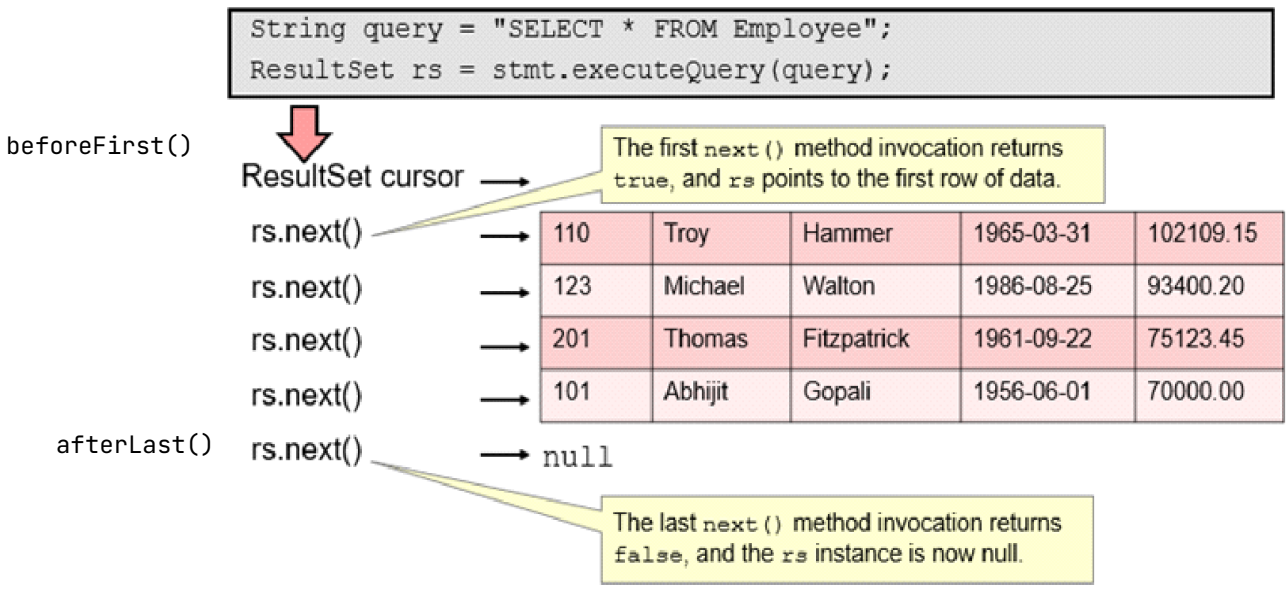
Métodos de PreparedStatement	Descripción
	continuación), y false si se trata de un recuento de actualizaciones o no hay resultados (en cuyo caso se usará el método `getUpdateCount()` para recuperar el valor devuelto).
`void setXXX(int indiceParametro, tipoJAva valor)`	Asigna al parámetro de índice "indiceParametro" el valor del tipo "tipoJava".

Interfaz ResultSet

- **ResultSet**: permite almacenar los datos de una tabla devueltos por una sentencia SQL (generalmente SELECT).
  - Se itera sobre los resultados (filas) mediante el método `next()`
  - Se obtiene el valor de una columna de la fila actual mediante los métodos `getXXX()` (por ejemplo, `getString()`, `getInt()`, `getFloat()`, etc.).
  - El valor de la columna se puede recuperar por número de índice (empezando en 1) o por nombre de columna.

Formato de los resultados obtenidos mediante un ResultSet

El formato de ResultSet tiene tipo tabla:



Tipos de ResultSet según el tipo de desplazamiento

Existen tres tipos de ResultSet según el tipo de desplazamiento que se pueda realizar.

- **TYPE_FORWARD_ONLY**: es el tipo predeterminado para un ResultSet.
  - Solo permite avanzar en el ResultSet hacia delante, de fila en fila. No se puede volver hacia atrás ni moverse a una posición específica.
  - No refleja los posibles cambios realizados en la base de datos mientras el ResultSet está en uso.

- **TYPE_SCROLL_INSENSITIVE:**
  - Permite desplazarse hacia adelante, hacia atrás y moverse a posiciones específicas dentro del ResultSet.
  - No refleja los cambios realizados en la base de datos mientras el ResultSet está abierto.

- **TYPE_SCROLL_SENSITIVE:**
  - Permite moverse hacia adelante, hacia atrás y a posiciones específicas, igual que TYPE_SCROLL_INSENSITIVE.
  - refleja los posibles cambios realizados en la base de datos mientras el ResultSet está en uso.

- Se puede especificar el tipo de ResultSet al crear el objeto Statement/PreparedStatement/Calalble que genera el ResultSet, utilizando la siguiente versión del método createStatment (o los equivalentes para PreparedStatemen/CallableStatement):

  ```
  Statement createStatement(int resultSetType, int resultSetConcurrency)
  ```

  Donde:

  - **`resultSetType`** puede ser tomar el valor TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE o TYPE_SCROLL_SENSITIVE
  - **`resultSetConcurrency`** puede tomar el valor **`CONCUR_READ_ONLY`** (el ResultSet es de solo lectura) o **`CONCUR_UPDATABLE`** (permite modificar los datos del ResultSet y reflejar esos cambios directamente en la base de datos).

Nota: el comportamiento depende del controlador JDBC. No todos los controladores JDBC soportan TYPE_SCROLL_INSENSITIVE o TYPE_SCROLL_SENSITIVE. Si no son compatibles, el controlador degradará el tipo de desplazamiento a TYPE_FORWARD_ONLY sin lanzar excepciones.

Métodos de la interfaz ResultSet

Estos son algunos de los métodos de la interfaz ResultSet. Para más información: https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html

Métodos de posicionamiento del cursor y para obtener información sobre la fila	
Métodos de ResultSet	**Descripción**
`public boolean next()`	Mueve el cursor a la fila siguiente desde su posición actual. Inicialmente el cursor del ResultSet está colocado antes de la primera fila; la primera llamada al método next convierte la primera fila en la fila actual; la

Métodos de posicionamiento del cursor y para obtener información sobre la fila	
Métodos de ResultSet	**Descripción**
	segunda llamada convierte la segunda fila en la fila actual, y así sucesivamente. Devuelve true cuando la nueva posición del cursor está en una fila válida, y false cuando no hay más filas.
`public boolean previous()`	Mueve el cursor a la fila anterior desde su posición actual. Devuelve true cuando la nueva posición del cursor está en una fila válida, y false si no hay filas en el ResultSet. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`public boolean last()`	Mueve el cursor a la última fila. Devuelve true cuando la nueva posición del cursor está en una fila válida, y false si está posicionado antes de la primera fila. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`public boolean first()`	Mueve el cursor a la primera fila. Devuelve true cuando la nueva posición del cursor está en una fila válida, y false si está posicionado antes de la primera fila. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`void afterLast()`	Mueve el cursor al final del ResultSet, después de la última fila. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`void beforeFirst()`	Mueve el cursor al inicio del ResultSet, antes de la primera fila. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`public boolean absolute(int fila)`	Pasa a la fila número `fila`. No válido si el ResultSet es TYPE_FORWARD_ONLY.
`public boolean relative(int n)`	Mueve el cursor `n` filas desde la actual (valor positivo hacia delante, negativo hacia atrás). No válido si el ResultSet es TYPE_FORWARD_ONLY. Una llamada `relative(1)` conseguiría el mismo efecto que llamar a `next()`. Una llamada `relative(-1)` conseguiría el mismo efecto que llamar a `previous()`.
`public int getRow()`	Devuelve el número de fila actual, teniendo en cuenta que la primera es la fila 1 (no la cero). No válido si el ResultSet es TYPE_FORWARD_ONLY.
`public boolean isLast()`	Devuelve true si la fila actual es la última.
`public boolean isFirst()`	Devuelve true si la fila actual es la primera.

Métodos para obtener los valores de cada columna: métodos getter

Devuelven el valor de la columna especificada de la fila actual del ResultSet. Hay que utilizar el método get que corresponda según el tipo de datos almacenado en la columna. El índice de una columna es un entero que vale 1 en la primera columna, 2 en la segunda, etc. Estos son solo algunos ejemplos.

Método de ResultSet	**Descripción**
`public int getInt(int columnIndex)`	Devuelve el valor de la columna de índice columnIndex como entero (int). El valor null se devolverá como 0 (cero). Si el valor en esa columna no es convertible a int se lanza una SQLException.
`public int getInt(String columnLabel)`	Devuelve el valor de la columna de nombre columnLabel como un entero (int). El valor null se devolverá como 0 (cero). Si el valor en esa columna no es convertible a int se lanza una SQLException.
`public String getString(int columnIndex)`	Devuelve el valor de la columna indicada con el índice columnIndex como un String (convierte el valor a un String, independientemente de su tipo de datos en la base de datos.
`public String getString(String columnLabel)`	Devuelve el valor de la columna indicada con el nombre columnLabel como un String (convierte el valor a un String, independientemente de su tipo de datos en la base de datos.
`public Object getObject(int columnIndex)`	Devuelve el valor de la columna del índice 'columnIndex' como un objeto Java.
`public Object getObject(String columnLabel)`	Devuelve el valor de la columna de nombre 'columnLabel' como un objeto Java.

Otros métodos

Método de ResultSet	**Descripción**
`ResultSetMetaData getMetaData()`	Devuelve información de las columnas del ResultSet (número, tipos y propiedades) en un objeto ResultSetMedatada.
`boolean wasNull()`	Devuelve true si la última columna leída tenía un valor SQL NULL. Antes de ejecutar este método hay que llamar a uno de los métodos getter
`int getType()`	Devuelve el tipo de ResultSet: 1003 (TYPE_FORWARD_ONLY), 1004 (TYPE_SCROLL_INSENSITIVE), o 1005 (TYPE_SCROLL_SENSITIVE).

Ejemplo: ejecuta una consulta con executeQuery y muestra los resultados

```
try {
  // Crea el objeto Statement
  Statement instruccion = conexion.createStatement();

  // Crea un String con el texto de la consulta SQL que se va a ejecutar
  String query = "SELECT * FROM clientes WHERE nombre LIKE \"Empresa%\"";

  // Ejecuta la consulta y guarda el resultado en un ResultSet "resultados"
  ResultSet resultados = instruccion.executeQuery(query);

  System.out.println("Listado de clientes: ");

  // Bucle while que recorre cada fila del ResultSet desde la primera a la última,
  // y muestra los valores de cada columna (nif, nombre y telefono)
  while (resultados.next()) {
    System.out.println("Cliente "+resultados.getString("nif")
    +", Nombre: "+resultados.getString("nombre")
    +", Teléfono: " +resultados.getString("telefono") );
  }
} catch (Exception e) {
    e.printStackTrace();
}
```

Ejemplo: uso de execute()

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

//Sentencia de tipo DQL
String sql1="SELECT * FROM departamentos";
boolean valor = sentencia.execute(sql1);

//Sentencia de tipo DML
//String sql2="UPDATE departamentos SET dnombre=LOWER(dnombre)"
//boolean valor = sentencia.execute(sql2);

if(valor){ //Si ejecutamos la sentencia tipo DQL, el método execute() devuelve true;
          //hay que obtener el ResultSet
  ResultSet rs = sentencia.getResultSet(); //Se obtiene el ResultSet

  // Bucle while que recorre cada fila del ResultSet y muestra los valores de las
  // columnas de índice 1, 2 3
  while (rs.next())
    System.out.printf("%d, %s, %s %n",
            rs.getInt(1), rs.getString(2), rs.getString(3));
    rs.close(); //Se cierrra el ResultSet

} else { //si ejecutamos la sentencia de tipo DML, execute() devuelve false, se ejecuta
        //getUpdateCount y se muestra el valor de filas afectadas
    int f = sentencia.getUpdateCount();
    System.out.printf("Filas afectadas:%d %n", f);
}

 sentencia.close(); //se cierra el Statement
```

Excepción SQLException

- **SQLException:** esta excepción puede ser lanzada por métodos de cualquiera de las clases e interfaces al ocurrir errores en la conexión a la base de datos o en la ejecución de consultas y procedimientos almacenados, por lo que se debe controlar obligatoriamente `(try-catch)`.

Interfaz DatabaseMetaData

- Para consultar la metainformación de una base de datos, Java dispone de la interfaz `DatabaseMetaData`, que obtendremos a partir de la conexión que tengamos establecida, de la siguiente manera:

```
Connection conexion = DriverManager.getConnection(connectionUrl);
DatabaseMetaData dbmd = conexión.getMetaData();
```

- Proporciona gran cantidad de información (más de 100 métodos) acerca de una base de datos, como las tablas/vistas que contiene, el nombre y la versión de la base de datos, el nombre del driver, etc.
- Además de los métodos de la interfaz DatabaseMetaData, hay otro método la interfaz **Connection** que permite obtener el nombre del catálogon (base de datos) a la que estamos conectados:

Método de Connection	
`String` **`getCatalog`**`()`	Devuelve el nombre del catálogo (base de datos) del objeto Connection o, dicho de otra forma, el nombre de la base de datos a la que estamos conectados o null si no estamos conectados a ninguna.

Estos son algunos de los métodos de la interfaz DatabaseMetaData. Más información:
https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html

Métodos de DatabaseMetadata	
`String` **`getDatabaseProductName`**`()`	Devuelve el nombre del SGBD.
`String` **`getDatabaseProductVersion`**`()`	Devuelve el número de versión del SGBD
`String` **`getDriverName`**`()`	Devuelve el nombre del driver JDBC.
`String` **`getDriverVersion`**`()`	Devuelve el número de versión del JDBC como un String.
`StringString` **`getURL`**`()`	Devuelve la URL de conexión.
`String` **`getUserName`**`()`	Devuelve el nombre de usuario.
`Connection` **`getConnection`**`()`	Devuelve un objeto Connection, que representa la conexión activa con la base de datos.

Métodos de DatabaseMetadata	
`ResultSet` **`getTables`**`(String catalog, String schemaPattern, String tableNamePattern, String[] types)`	Devuelve un ResultSet con información sobre las tablas y vistas de la base de datos. Cada fila del ResultSet tiene información sobre una tabla. Cada columna contiene un campo de información de la tabla: TABLE_CAT, TABLE_SCHEM, TABLE_NAME, TABLE_TYPE , REMARKS, etc. • catalog: el catálogo de la BD. Un valor null indica que el nombre del catálogo no debe utilizarse para limitar la búsqueda. • schemaPattern: esquema de la base de datos (en MySQL el nombre de usuario).Un valor null indica que el nombre del esquema no debe utilizarse para limitar la búsqueda. • tableNamePattern: patrón que indica el nombre de las tablas que queremos que obtenga el método. Se pueden buscar columnas que coincidan parcialmente con nombres, utilizando "_" o " %". Ejemplo: "de%" obtendría todas las tablas cuyo nombre empieza por "de" • types: sirve para indicar qué tipos de objetos queremos obtener, por ejemplo TABLE, VIEW, SYNONYM, etc.
`ResultSet` **`getCatalogs`**`()`	Devuelve un ResultSet con los nombres de todos los catálogos (bases de datos) disponibles en el servidor al que estemos conectados.
`ResultSet` **`getPrimaryKeys`**`(String catalog,String schema,String table)`	Devuelve la lista de columnas que forman parte de la clave primaria de la tabla especificada. Cada fila contiene información de una de las columnas que forman la clave primaria: COLUMN_NAME, PK_NAME, KEY_SEQ (Posición de la columna en la clave primaria (para claves compuestas)
`ResultSet` **`getImportedKeys`**`(String catalog, String schema, String table)`	Devuelve una lista de las claves ajenas existentes en la tabla especificada.
`ResultSet` **`getExportedKeys`**`(String catalog, String schema, String table)`	Devuelve una lista de todas las claves ajenas que utilizan la clave primaria de la tabla especificada.
`ResultSet` **`getProcedures`**`(String catalog, String schemaPattern, String procedureNamePattern)`	Devuelve la lista de procedimientos almacenados.

Métodos de DatabaseMetadata	
`ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)`	Devuelve un ResultSet con información sobre las columnas de una tabla "tableNamePattern" de la base de datos "catalog". Cada fila del ResultSet contiene información de una columna de la tabla. Cada columna del ResultSet contiene un campo de información sobre la columna de la tabla (hasta 24) , como por ejemplo COLUMN_NAME, DATA_TYPE, TYPE_NAME, COLUMN_SIZE, etc. • catalog: el catálogo de la BD. Un valor null indica que el nombre del catálogo no debe utilizarse para limitar la búsqueda. • schemaPattern: esquema de la base de datos (en MySQL el nombre de usuario). Un valor null indica que el nombre del esquema no debe utilizarse para limitar la búsqueda. • tableNamePattern: patrón que indica el nombre de las tablas que queremos que obtenga el método. Se pueden buscar columnas que coincidan parcialmente con nombres, utilizando "_" o " %". Ejemplo: "de%" obtendría todas las tablas cuyo nombre empieza por "de". • columnNamePattern: patrón que indica el nombre de las columnas que queremos que obtenga el método.

Ejemplo: obtener algunos metadatos de la BD

```
try {

  DatabaseMetaData infoBD = conexion.getMetaData();//Obtiene los metadatos de la BD

  //Muestra el nombre del SGBD
  System.out.println("Base de datos: " + infoBD.getDatabaseProductName());

  //Muestra el número de versión del SGBD
  System.out.println("Version: " + infoBD.getDatabaseProductVersion());

} catch (Exception ex) {
    // Tratar la excepción
}
```

Métodos getImportedKeys y getExportedKeys de la interfaz DatabaseMetaData

Ambos métodos son útiles para analizar dependencias y relaciones entre tablas en una base de datos relacional.

- **ResultSet getImportedKeys(String catalog, String schema, String table)**

Este método obtiene información sobre las claves ajenas que apuntan hacia la tabla especificada (es decir, claves que importa esta tabla desde otras tablas).

Uso típico: Para identificar qué tablas referencian la tabla especificada como origen de sus claves foráneas.

Parámetros:

- **catalog:** El nombre del catálogo donde se encuentra la tabla (puede ser null para ignorarlo).
- **schema:** El esquema donde se encuentra la tabla (puede ser null para ignorarlo).
- **table:** El nombre de la tabla cuyos importadores de claves foráneas quieres consultar.

Ejemplo: Si la tabla pedidos tiene una clave ajena apuntando a clientes, este método devuelve información sobre esa clave en pedidos.

- **ResultSet getExportedKeys(String catalog, String schema, String table)**

Este método obtiene información sobre las claves ajenas que apuntan desde la tabla especificada hacia otras tablas (es decir, claves que exporta esta tabla para ser referenciadas por otras tablas).

Uso típico: Para identificar qué otras tablas utilizan la tabla especificada como destino de sus claves foráneas.

Parámetros:

- **catalog**: El nombre del catálogo donde se encuentra la tabla (puede ser null para ignorarlo).
- **schema:** El esquema donde se encuentra la tabla (puede ser null para ignorarlo).
- **table**: El nombre de la tabla cuyos exportadores de claves foráneas quieres consultar.

Ejemplo: Si la tabla clientes es referenciada por una clave foránea en pedidos, este método devolverá información sobre la clave en clientes.

Ambos métodos devuelven un `ResultSet` con las mismas columnas. Algunas de las más relevantes son:

COLUMNA	DESCRIPCIÓN
PKTABLE_NAME	Nombre de la tabla primaria.
PKCOLUMN_NAME	Nombre de una columna específica que forma parte de la clave primaria. Ejemplo: Si una clave primaria está compuesta por las columnas id_cliente e id_pedido, el ResultSet incluirá dos filas, una para PKCOLUMN_NAME = id_cliente y otra para PKCOLUMN_NAME = id_pedido
FKTABLE_NAME	Nombre de la tabla foránea.
FKCOLUMN_NAME	Nombre de una columna específica que forma parte de la clave ajena. Ejemplo: Si una clave ajena está compuesta por las columnas id_cliente e id_pedido en una tabla, el ResultSet incluirá dos filas, una para cada FKCOLUMN_NAME = id_cliente y otra para FKCOLUMN_NAME = id_pedido.

FK_NAME	Si la FK está compuesta por más de una columna, se refiere al identificador de la clave ajena en su conjunto (no a las columnas individuales que la componen). Ejemplo: Si la clave foránea en una tabla se define como FK_CLIENTE_PEDIDO, este valor aparecerá en FK_NAME para todas las columnas que forman parte de esa clave.
PK_NAME	Si la PK está compuesta por más de una columna, se refiere al identificador de la clave primaria en su conjunto (no a las columnas individuales que la componen). Ejemplo: `PRIMARY KEY (id_cliente, id_pedido) CONSTRAINT PK_CLIENTE_PEDIDO` Si la clave primaria definida en una tabla se llama PK_CLIENTE_PEDIDO, este valor aparecerá en PK_NAME para todas las columnas que forman parte de esa clave.

Ejemplo: obtener dependencias entre dos tabla relacionadas

Supongamos una base de datos con dos tablas relacionadas:

- **`clientes`** (clave primaria: `id_cliente`).
- **`pedidos`** (clave foránea: `id_cliente` que referencia `clientes.id_cliente`).

Ejemplo de código para obtener las dependencias entre las dos tablas:

```
public class ForeignKeyExample {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/mi_base_datos";
        String user = "root";
        String password = "admin";

        try (Connection conn = DriverManager.getConnection(url, user, password)) {
            DatabaseMetaData metaData = conn.getMetaData();

            // Obtener claves foráneas que apuntan hacia la tabla 'clientes'
            ResultSet importedKeys = metaData.getImportedKeys(null, null, "pedidos");
            System.out.println("Claves foráneas importadas en 'pedidos':");
            while (importedKeys.next()) {
                System.out.println("Columna: " + importedKeys.getString("FKCOLUMN_NAME") +
                                   " referencia a " + importedKeys.getString("PKTABLE_NAME") +
                                   "." + importedKeys.getString("PKCOLUMN_NAME"));
            }

            // Obtener claves foráneas que apuntan desde la tabla 'clientes'
            ResultSet exportedKeys = metaData.getExportedKeys(null, null, "clientes");
            System.out.println("\nClaves foráneas exportadas por 'clientes':");
            while (exportedKeys.next()) {
                System.out.println("Columna: " + exportedKeys.getString("PKCOLUMN_NAME") +
                                   " es referenciada por " + exportedKeys.getString("FKTABLE_NAME") +
                                   "." + exportedKeys.getString("FKCOLUMN_NAME"));
            }
        }
    }
}
```

Ejemplo de ejecución

Claves foráneas importadas en "pedidos"':
Columna: id_cliente referencia a clientes.id_cliente

Claves foráneas exportadas por "clientes":
Columna: id_cliente es referenciada por pedidos.id_cliente

Interfaz ResultSetMetaData

Podemos obtener metainformación de un ResultSet con el siguiente método de la interfaz `ResultSet:`

```
ResultSetMetaData getMetaData()
```

Este método devuelve un objeto de tipo **ResultSetMetaData**, que proporciona métodos para obtener información del ResultSet, como el número de columnas, sus nombres, etc.

Estos son algunos de los métodos de la interfaz ResultSet. Más información:
https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html

Métodos de ResultSetMetaData	
`int getColumnCount()`	Devuelve el número de columnas del ResultSet
`String getColumnName(int columnIndex)`	Devuelve el nombre de la columna cuya posiciones e indica en columnIndex.
`String getColumnTypeName(int columnIndex)`	Devuelve el nombre del tipo de dato específico del SGBD que contiene la columna indicada en columnIndex.
`int getColumnType(int columnIndex)`	Devuelve el tipo SQL de la columna indicada en columnIndex, tal y como está definido en la clase `java.sql.Types`. Cada tipo de datos se identifica por una constante de tipo int. Ejemplos: `CHAR` (valor 1), `VARCHAR` (valor 12), `INTEGER` (valor 4), `FLOAT` ((valor 7), etc.

Ejemplo 1: obtener metadatos del ResultSet

```
// Ejecuta una consulta y guarda los resultados en el objeto ResultSet rs
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");

//Guarda los metadatos de rs en el objeto ResultSetMedaData rsmd
ResultSetMetaData rsmd = rs.getMetaData(); //Guarda los metadatos de rs en rsmd

// Obtiene el número de columnas del RestultSet a partir del objeto rsmd
int numberOfColumns = rsmd.getColumnCount();

// Obtiene si la columna de índice 1 (primera columna) puede usarse en una cláusula WHERE
boolean b = rsmd.isSearchable(1);
```

Ejemplo 2: realiza una consulta SQL que selecciona todos los registros de una tabla clientes, obteniendo y mostrando la información de cada columna en el resultado, sin conocer los nombres de las columnas.

```
try {
    // Crea el objeto Statement
    Statement instruccion = conexion.createStatement();

    // Crea un String con el texto de la consulta SQL que se va a ejecutar
    String query = "SELECT * FROM clientes";

    // Ejecuta la consulta y guarda el resultado en un ResultSet "resultados"
    ResultSet resultados = instruccion.executeQuery(query);

    // Obtiene los metadatos del ResultSet y los guarda en un objeto ResultSetMetadata
    ResultSetMetaData infoResultados = resultados.getMetaData();

    //Obtiene el número total de columnas del ResultSet
    int col = infoResultados.getColumnCount();

    // Bucle while que recorre cada fila del ResultSet desde la primera a la última
    while (resultados.next()) {
        // Bucle for que recorre las columnas de cada fila e imprime sus nombres
        for (int i = 1; i <= col; i++)System.out.print(resultados.getString(i) + "\t");

        //Al terminar cada fila se imprime un salto de línea
        System.out.println("");
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Ejemplo 3: obtener los tipos de las columnas de un ResultSet

```
public class ColumnTypeExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_datos";
        String user = "root";
        String password = "admin";

        try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT * FROM mi_tabla")) {

            ResultSetMetaData metaData = rs.getMetaData();
            int columnCount = metaData.getColumnCount();

            for (int i = 1; i <= columnCount; i++) {
                String columnName = metaData.getColumnName(i);
                int columnType = metaData.getColumnType(i);
                String columnTypeName = metaData.getColumnTypeName(i);

                System.out.println("Columna: " + columnName);
                System.out.println("Tipo SQL: " + columnType + " (" + columnTypeName + ")");
                System.out.println("--------------------------------");
            }

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

Ejemplo de ejecución

Supongamos una tabla con las siguientes columnas:

- `id` (INTEGER)
- `nombre` (VARCHAR)
- `fecha_nacimiento` (DATE)

El código produciría una salida como esta:

```
Columna: id
Tipo SQL: 4 (INTEGER)
--------------------------------
Columna: nombre
Tipo SQL: 12 (VARCHAR)
--------------------------------
Columna: fecha_nacimiento
Tipo SQL: 91 (DATE)
--------------------------------
```

5. Conexión a la base de datos

- Para establecer una conexión a la base de datos debemos crear un objeto Connection mediante el método **getConnection** de la clase **java.sql.DriverManager**, en cualquiera de sus versiones:
    - **static Connection getConnection(String url) throws SQLException**: devuelve una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL.
    - **static Connection getConnection(String url, Properties info) throws SQLException**: devuelve una conexión, si es posible, a la base de datos; algunos parámetros están especificados en la URL y otros en un objeto de propiedades (Properties).
    - **static Connection getConnection(String url, String user, String password) throws SQLException**: devuelve una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL. Los datos de usuario y contraseña se suministran en dos parámetros adicionales.

  El parámetro **url** representa la cadena de conexión a una base de datos.
- Una **cadena de conexión** es una secuencia de texto (String) que proporciona la información necesaria para que una aplicación se conecte a una base de datos.
- Los componentes de una cadena de conexión pueden variar según el sistema de base de datos, pero el formato genérico suele incluir los siguientes campos:
    - **Protocolo y driver JDBC**: Especifica el tipo de base de datos y el controlador que se va a usar. En JDBC, comienza con jdbc: seguido del nombre del controlador, como mysql, postgresql, sqlite, oracle, etc.
    - **Ubicación del servidor (Host)**: identifica el servidor donde se está ejecutando la base de datos. Puede ser mediante su dirección IP o su nombre de dominio, o, puede ser localhost si la base de datos está en el mismo equipo que la aplicación.
    - **Puerto**: Especifica el número de puerto en el que la base de datos escucha las conexiones. Cada sistema de base de datos tiene un puerto predeterminado (por ejemplo, 3306 para MySQL, 5432 para PostgreSQL), aunque este puede modificarse, por ejemplo, por decisiones de seguridad o por tener varios servidores ejecutándose a la vez (no pueden usar el mismo puerto).

- **Nombre de la base de datos**: Es el nombre de la base de datos específica a la que se desea conectar dentro del sistema de gestión de bases de datos.

- El formato genérico de una cadena de conexión en JDBC sería:

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

  Vemos que el formato es similar a una URL HTTP o FTP.

- Ejemplos de cadenas de conexión a una base de datos llamada "mi_base_datos"
  - Para mysql: jdbc:mysql://localhost:3306/mi_base_datos"
  - Para SQLite: jdbc:sqlite:mi_base_datos

- Para acceder a la base de datos es necesario cargar el **driver JDBC**. La clase **java.sql.DriverManager** gestiona los drivers de bases de datos y permite obtener una conexión a la base de datos. Esta clase intenta cargar los drivers que se encuentran en una propiedad del sistema llamada **jdbc.drivers**, siempre que estos se encuentren en el classpath. Para ello, tendremos que añadir el driver adecuado (archivo .jar) al proyecto.

  Nota: Antes de la versión Java 1.6 había que cargar el driver explícitamente usando la instrucción:

  `Class.forName("org.sqlite.JDBC");` // Esto no es necesario desde Java 1.6

Ejemplo: conexión a una base de datos SQLite de nombre "mi_base_de_datos.sb"

```
Connection conn = null;
try {
    // Carga explícita del driver (ya no es necesaria desde Java 1.6 en adelante)
    Class.forName("org.sqlite.JDBC");

    // Obtención de la conexión a la base de datos
    conn = DriverManager.getConnection("jdbc:sqlite:mi_base_de_datos.db");

    System.out.println("Conexión exitosa a la base de datos.");
    // Resto del código para trabajar con la base de datos...

} catch (ClassNotFoundException e) {
    System.out.println("No se encontró el driver JDBC.");
    e.printStackTrace();
} catch (SQLException e) {
    System.out.println("Error al conectar con la base de datos.");
    e.printStackTrace();
} finally {
    // Cerrar la conexión si no es nula
    if (conn != null) {
        try {
            conn.close();
            System.out.println("Conexión cerrada.");
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

6. Valores de claves autogeneradas

Es una práctica muy habitual crear una tabla de manera que la clave primaria sea una columna numérica y se deje al sistema gestor de base de datos que proporcione un nuevo valor para cada nueva fila que se inserta. Un ejemplo podría ser una aplicación que trabaja con facturas. Cada factura debe tener un identificador único, que normalmente es un simple número. El número asignado a una factura no tiene ninguna significación especial. Lo importante es que cada factura tenga un número distinto, y se deja al sistema que asigne un nuevo número consecutivo cada vez que crea una nueva factura. Las claves de este tipo se suelen llamar claves **autoincrementales** o, en JDBC, claves **autogeneradas**. El mecanismo es esencialmente igual en casi todas las bases de datos, con pequeños cambios en la sintaxis de la sentencia SQL utilizada.

Recuperación de claves autogeneradas

Para obtener las claves primarias autogeneradas de una o varias filas recién insertadas se utiliza el método getGeneratedKeys() de la interfaz Statement o PreparedStatement. Si se insertan varias filas en una sola operación, este método devuelve todas las claves generadas para esas filas.

El procedimiento para hacer esto es el siguiente:

1. Configurar el Statement o PreparedStatement que inserta las filas para devolver las claves generadas.

Tenemos que crear el Statement o PreparedStatement indicando que las claves generadas serán accesibles después de ejecutar la sentencia INSERT. Esto se logra utilizando la constante **Statement.RETURN_GENERATED_KEYS**.

Ejemplo:

```
PreparedStatement ps=conexion.prepareStatement(sql,Statement.RETURN_GENERATED_KEYS);
```

2. Ejecutar la sentencia SQL que inserta una o varias filas (se les asignará claves autogeneradas)

Utilizamos el método **executeUpdate()** para ejecutar la inserción:

```
pstmt.executeUpdate();
```

3. Recuperar las claves autogeneradas.

Después de ejecutar la sentencia, se llama al método **getGeneratedKeys()** para obtener un ResultSet que contiene las claves generadas automáticamente por la base de datos.

```
ResultSet generatedKeys = pstmt.getGeneratedKeys();
```

4. Leer el valor de las claves autogeneradas.

El ResultSet devuelto por getGeneratedKeys() contiene las claves generadas, que estarán en la primera columna si la clave primaria no es compuesta (índice 1). Podemos acceder a ellas usando los métodos habituales de ResultSet como **getInt()**, **getLong()**, etc.

Ejemplo:

```
if (generatedKeys.next()) {
    int generatedId = generatedKeys.getInt(1); // O `getLong(1)` si la clave
                                               //es de tipo BIGINT
    System.out.println("Clave generada: " + generatedId);
}
```

Ejemplo Supongamos que tenemos la siguiente tabla:

```
CREATE TABLE MiTabla (
  id INT AUTO_INCREMENT PRIMARY KEY, -- Clave primaria autoincremental
  columna1 VARCHAR(255) NOT NULL,   -- Columna para almacenar texto
  columna2 VARCHAR(255) NOT NULL    -- Otra columna para texto
);
```

El siguiente código inserta una fila en la tabla, asignando valores a columna1 y columna2, mientras que mientras que el campo id será generado automáticamente. Después muestra las claves autogeneradas y el contenido de la tabla.

```
package Ejemplos_MySQL;

import java.sql*;

public class UT2_24_25_EjemploClavesAutogeneradas {
        public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/bd_pruebas";
        String user = "root";
        String password = "admin";

        // SQL para insertar datos
        String sqlInsert = "INSERT INTO MiTabla (columna1, columna2) VALUES (?, ?)";

        try (Connection con = DriverManager.getConnection(url, user, password);
             PreparedStatement pstmt = con.prepareStatement(sqlInsert,
                                       Statement.RETURN_GENERATED_KEYS)) {

            // Configurar parámetros de inserción
            pstmt.setString(1, "valor1");
            pstmt.setString(2, "valor2");

            // Ejecutar inserción
            int filasAfectadas = pstmt.executeUpdate();
            System.out.println("Filas insertadas: " + filasAfectadas);

            // Recuperar claves generadas
            try (ResultSet generatedKeys = pstmt.getGeneratedKeys()) {
                if (generatedKeys.next()) {
                    int idGenerado = generatedKeys.getInt(1);
                    System.out.println("Clave autogenerada: " + idGenerado);
                }
            }
            // Mostrar contenido actual de la tabla
            String sqlSelect = "SELECT * FROM MiTabla";
            try (Statement stmt = con.createStatement();
                 ResultSet rs = stmt.executeQuery(sqlSelect)) {

                System.out.println("Contenido actual de la tabla MiTabla:");
                while (rs.next()) {
                    int id = rs.getInt("id");
                    String columna1 = rs.getString("columna1");
                    String columna2 = rs.getString("columna2");
                    System.out.printf("id: %d, columna1: %s, columna2: %s%n", id,
                                      columna1, columna2);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```