



1. Introducción

Hasta ahora, ya se sabe cómo abrir un archivo y utilizarlo como “almacén” para los datos que maneja tu aplicación. Utilizar un archivo para almacenar datos es la forma más sencilla de persistencia, porque en definitiva, la persistencia es hacer que los datos perduren en el tiempo.

Sin embargo, cuando los datos de la aplicación solo están disponibles mientras la aplicación se está ejecutando, tenemos un **nivel de persistencia muy bajo**.

Afortunadamente, hay otras formas de hacer los datos de una aplicación persistentes, y niveles de persistencia más altos. Una de las formas de lograrlo es almacenando los datos en una base de datos relacional.

2. Bases de datos relacionales

Actualmente, las bases de datos relacionales constituyen el sistema de almacenamiento probablemente más extendido, aunque otros sistemas de almacenamiento de la información se estén abriendo paso poco a poco.

Una base de datos relacional se puede definir, de una manera simple, como aquella que presenta la información en tablas con filas y columnas, siendo una tabla un colección de objetos del mismo tipo (**filas o tuplas**).

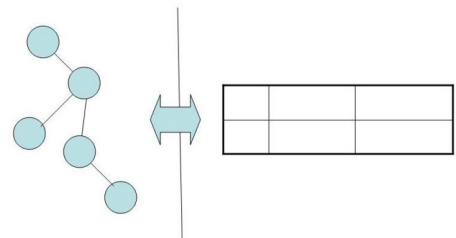
Cada fila de la misma se identifica de manera unívoca mediante una **clave primaria**.

El modo en que los datos se almacenan, mantienen y recuperan es gestionado por un sistema gestor de base de datos, que en el caso concreto de las bases de datos relacionales se denomina: **Relational Database Management System (RDBMS)** o en español “**Sistema Gestor de bases de Datos Relacional**” (**SGBDR**).

2.1. Desfase objeto-relacional

Las bases de datos relacionales no están diseñadas para almacenar objetos. Trata con relaciones o tablas debido a su naturaleza matemática.

En cambio, el modelo de POO trata con objetos y las asociaciones entre ellos. Por se tiene un problema entre estos dos modelos: como persistir los objetos de la aplicación.



¿Cómo se puede solventar este problema?.



Acceso a Datos

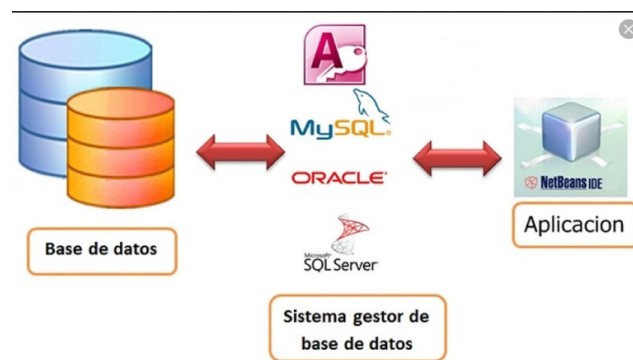
U2: Manejo de conectores

Cada vez que los objetos deben extraerse o almacenarse de la base de datos relacional se requiere **un mapeo desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación.**

Esto es sencillo para un caso simple, pero complicado si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

2.2. Protocolos de acceso a bases de datos

Cada empresa desarrolladora de un **SGBD** implementó soluciones propietarias específicas para su sistema, es decir, cada **SGBD** tenía su propia conexión y su propio API.

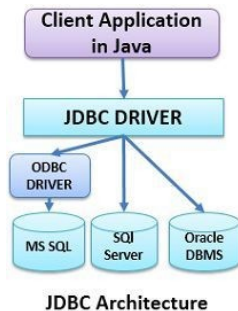


Sin embargo, los propietarios de los **SGBD** pronto se dieron cuenta de que colaborando conjuntamente podían sacar mayor rendimiento y avanzar mucho más rápidamente. Esto hizo que cada uno de ellos proporcionara diferentes drivers específicos (**ODBC, JDBC, ADO.NET, PDO, ...**) para que sus bases de datos pudiesen ser accedidas por las aplicaciones de un modo más sencillo.

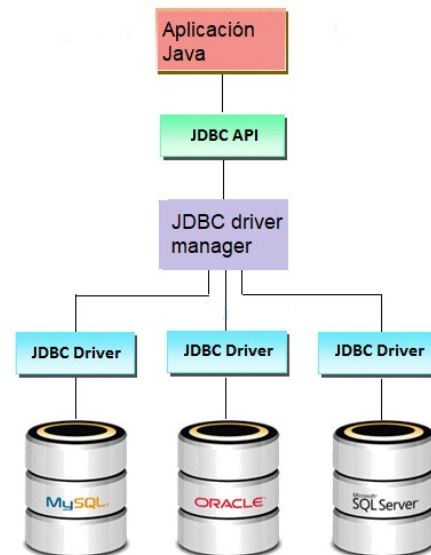
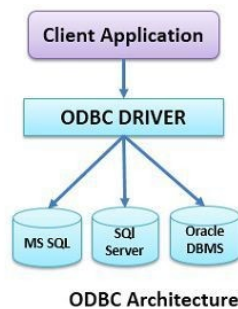
Java, mediante **JDBC (Java Database Connectivity)**, permite simplificar el acceso a bases de datos relacionales, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos.

Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java.
- Aprovechar la experiencia de los API's de bases de datos existentes.
- Ser lo más sencillo posible.



Vs



3. Acceso a datos mediante JDBC

Como se aprecia en las imágenes anteriores, una aplicación Java que utiliza **JDBC** puede comunicarse con diferentes bases de datos. Al igual que **ODBC**, **JDBC** proporciona una manera consistente de conectarse a una base de datos, ejecutar comandos y recuperar los resultados.

JDBC no impone un lenguaje de comando común: puede usar la sintaxis específica de Oracle cuando está conectado a un servidor Oracle y la sintaxis específica de **MySQL** cuando está conectado a un servidor **MySQL**.

La clase **JDBC DriverManager** es responsable de localizar un controlador **JDBC** que necesita la aplicación. Cuando una aplicación cliente solicita una conexión de base de datos, la solicitud se expresa en forma de una **URL** (*Uniform Resource Locator*). Una **URL** de **JDBC** es similar a las **URL** que utiliza con un navegador web.

Para conectarnos a la base de datos Oracle, por ejemplo:

```
jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE
```

Cuando una aplicación solicita una conexión, el **DriverManager** pregunta a cada controlador si puede conectarse a la base de datos especificada con la **URL** dada.

Tan pronto como encuentra un controlador adecuado, la búsqueda se detiene y el controlador intenta establecer una conexión con la base de datos. Si el intento de conexión falla, el controlador lanzará una **SQLException** a la aplicación. Si la conexión se completa con éxito, el controlador crea un objeto de conexión y lo devuelve a la aplicación.

3.1. Modelos de acceso a bases de datos con JDBC

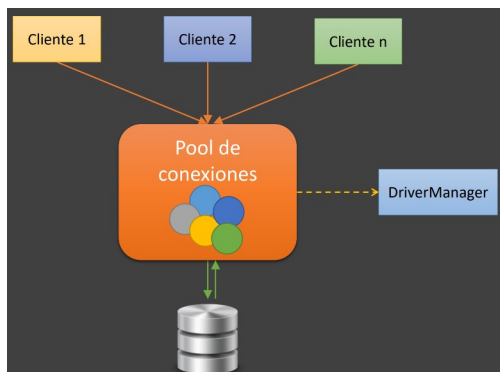
JDBC proporciona diversos modelos para acceder a bases de datos desde aplicaciones Java:

- **Modelo Standalone o de conexión directa**

En este caso la aplicación Java se conecta directamente a la base de datos utilizando el driver específico. Es utilizado principalmente, para aplicaciones sencillas.

- **Modelo de Pool de Conexiones (Connection Pooling)**

Se usa este modelo para mejorar el rendimiento evitando abrir y cerrar conexiones repetidamente. Es muy útil en sistemas concurrentes.



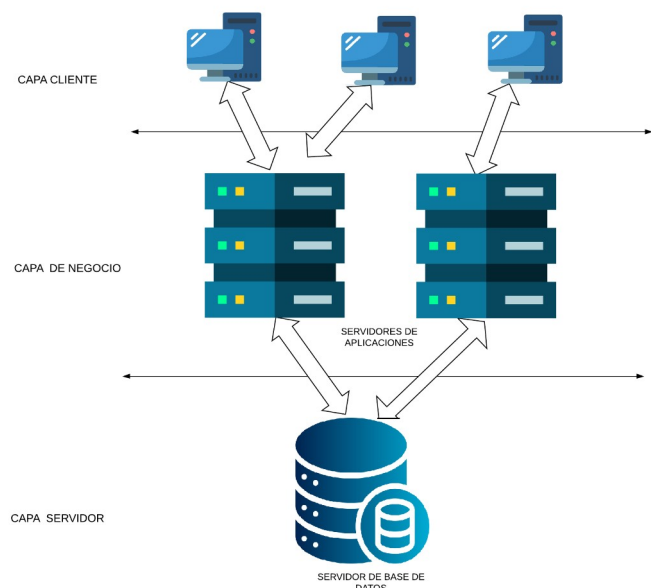
Un pool de conexiones es un grupo de conexiones abiertas a una base de datos, de forma que cuando una aplicación necesita conectarse, obtiene una conexión del pool en lugar de crear una nueva. Una vez que la aplicación finaliza la conexión, esta no se cierra, sino que se devuelve al pool para que sea reutilizada por otro proceso.

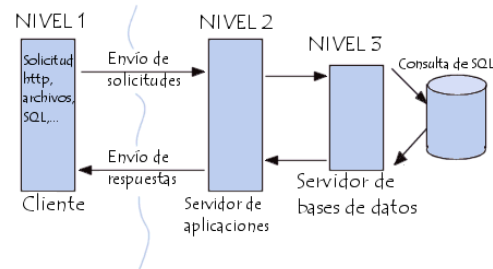
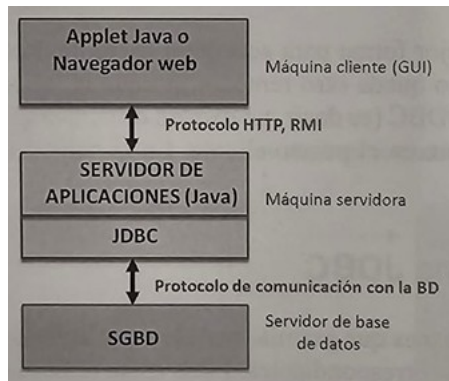
- **Modelo Cliente-Servidor (MultiCapa)**

Arquitectura en capas, donde la aplicación Java se conecta a una base de datos a través de un servidor de aplicaciones (Tomcat, Jboss, GlassFish,...) que gestiona las conexiones.

Este modelo permite separar a lógica de negocio de la lógica de acceso a datos, siendo compatible con patrones como MVC.

Se usa mucho en el desarrollo de aplicaciones web.





Por tanto, el flujo de la información en el modelo de Capas en JDBC es:

- **Capa de Presentación:** el usuario realiza una acción (por ejemplo, registrar una compra).
- **Capa Lógica de Negocio:** el controlador o la clase de servicio valida la solicitud, aplica las reglas de negocio, y llama a la capa de acceso a datos para interactuar con la base de datos.
- **Capa de Acceso a Datos:** la capa de acceso a datos establece una conexión a la base de datos usando **JDBC**, ejecuta las consultas **SQL** necesarias y devuelve los resultados a la capa de lógica de negocio.
- **Capa de Lógica de Negocio:** procesa los resultados y envía la respuesta a la capa de presentación.
- **Capa de Presentación:** Muestra los resultados al usuario (por ejemplo, una confirmación de compra).
- **Modelo Mapeo Objeto-Relacional (ORM)**

Las tablas se mapean para convertirse en objetos java. Un ejemplo es **Hibernate** o **JPA**. Es más simple de manejar ya que se trabaja con objetos en lugar de con sentencias SQL, pero es más lento puesto que hay que traducir a SQL todas las operaciones que se realicen.



Acceso a Datos

U2: Manejo de conectores

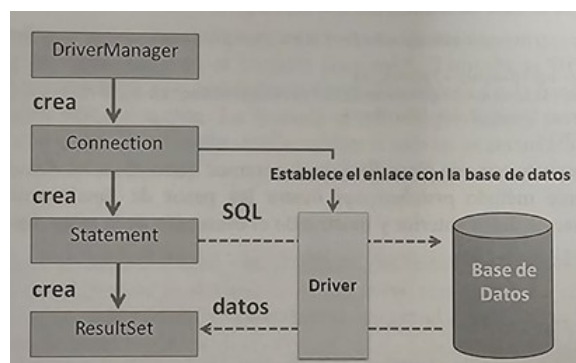
3.2. API JDBC

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

Clase - Interface	Descripción
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
DriverManager	Permite gestionar todos los drivers instalados en el sistema.
DriverPropertyInfo	Proporciona diversa información acerca de un driver.
Connection	Representa una conexión con una base de datos.
DatabaseMetadata	Proporciona información acerca de una base de datos.
Statement	Permite ejecutar sentencias SQL sin parámetros.
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada.
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
ResultSet	Contiene las filas resultantes de ejecutar una SELECT.
ResultSetMetadata	Permite obtener información sobre ResultSet , como número de columnas, sus nombres, etc...

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

1. Importar las clases necesarias
2. Cargar el driver **JDBC**
3. Identificar el origen de datos
4. Crear un objeto **Connection**
5. Crear un objeto **Statement**
6. Ejecutar una consulta con un objeto **Statement** o **PreparedStatement**
7. Recuperar los datos del objeto **ResultSet**
8. Liberar el objeto **ResultSet**
9. Liberar el objeto **Statement**
10. Liberar el objeto **Connection**





Acceso a Datos

U2: Manejo de conectores

4. Conexión a una base de datos.

Para acceder a una base de datos y así poder operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método `getConnection()` de la clase `DriverManager`. Este método recibe como parámetro la **URL** de **JDBC** que identifica a la base de datos con la que queremos realizar la conexión.

Tal y como se ha indicado en un apartado anterior, cuando se presenta con una **URL** específica, `DriverManager` itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún conector adecuado, se lanza una **SQLException**

Si la conexión se establece, la ejecución del método `getConnection()` devuelve un objeto `Connection` que representa la conexión con la base de datos.

```
try
{
    Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost/tienda","pruebas", "pruebas");
}
catch (SQLException ex) {
    // Tratar el error
}
```

La **URL** pasada como primer parámetro tiene el siguiente formato:

```
jdbc:mysql://nombre_host:puerto/nombre_basededatos
```

donde:

- `jdbc:mysql` - indica que vamos a usar un driver jdbc para mysql.
- `nombre_host` – indica el nombre del servidor donde se aloja la base de datos. Aquí se puede poner una IP o un nombre de la máquina que está en la red. Si se especifica localhost como nombre, estamos indicando que el servidor de la base de datos está alojado en la misma máquina en la que se ejecuta el programa Java.
- `Puerto` – Puerto de la base de datos. Si no se pone se asume que la base de datos está en su puerto por defecto. Por ejemplo, para mysql, 3306.
- `nombre_basededatos` – Nombre de la base de datos a la que se va a conectar la aplicación.

El segundo parámetro es el nombre del usuario con el que se accede y el tercer parámetro es la clave del usuario.

El método está sobrecargado pudiendo usarse como otra opción la siguiente:

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




Acceso a Datos

U2: Manejo de conectores

En este caso, solo se le pasa la **URL** explicada anteriormente.

```
try
{
    Connection conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost/tienda?user=pruebas&password=pruebas");
}
catch (SQLException ex) {
    // Tratar el error
}
```

5. Ejecución de sentencias de manipulación de datos.

Como ya se ha comentado anteriormente para la manipulación de datos la API JDBC ofrece las siguientes clases:

- **Statement:** para la ejecución de consultas sin parámetros.
- **PreparedStatement:** para la ejecución de consultas preparadas (consultas con parámetros).
- **CallableStatement:** para ejecutar procedimientos almacenados en la base de datos.

Además, el API JDBC distingue dos tipos de consultas:

1. Consultas: **SELECT**
2. Actualizaciones: **INSERT, UPDATE, DELETE**, sentencias **DDL**

5.1. Clase Statement

Statement es un interfaz que proporciona métodos para ejecutar sentencias *SQL* y obtener resultados. Al ser un interfaz no se pueden crear objetos directamente. En su lugar los objetos se obtienen con una llamada al método `createStatement()` de un objeto **Connection** válido.

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

Al crearse un objeto **Statement** se crea un espacio de trabajo para crear consultas *SQL*, ejecutarlas y recibir los resultados.



Acceso a Datos

U2: Manejo de conectores

Una vez creado el objeto se pueden usar los siguientes métodos:

- **ResultSet executeQuery (String)**: Se utiliza para sentencias *SELECT* que recuperan datos a partir de un objeto *ResultSet*.
- **int executeUpdate (String)**: Se utiliza para sentencias que no devuelven un *ResultSet*, como son:
 - Las sentencias de manipulación de datos (DML): *INSERT*, *UPDATE* y *DELETE*. En este caso el método devuelve un entero indicando el número de filas que se vieron afectadas.
 - Las sentencias de definición de datos (DDL): *CREATE*, *DROP* y *ALTER*. En este caso el método devuelve siempre el valor 0.
- **int getUpdateCount()**: Devuelve el número de filas afectadas por la última consulta *INSERT*, *UPDATE* o *DELETE*. Devuelve -1 si no hay filas afectadas.
- **boolean execute (String)**: Se puede usar para cualquier sentencia SQL, tanto para las que devuelven un *ResultSet* (SELECT), como para que devuelven el número de filas afectadas (DML) y para las de definición de datos (DDL). Este método devuelve:
 - **true**, si la consulta devuelve un *ResultSet*. En este caso será necesario llamar al método *getResultSet()* de la clase *Statement* para recuperar las filas devueltas por la consulta.
 - **false**, si la consulta no devuelve un *ResultSet*. En este caso será necesario llamar al método *getUpdateCount()* para obtener el número de filas que se vieron afectadas por la consulta DML.

5.2. Clase ResultSet

A través de un objeto *ResultSet* se puede acceder al valor de cualquier columna de la fila actual por nombre:

```
tipoDato valor = resultSet.getXXX("nombre_columna");
```

o por posición:

```
tipoDato valor = resultSet.getXXX(indice_columna);
```

Hay que destacar que el índice de la columna siempre empieza por 1.



Acceso a Datos

U2: Manejo de conectores

Algunos de los métodos getter para la obtención de valores son los siguientes:

Tipo SQL (en BD)	Método getter	Tipo Java devuelto	Ejemplo
CHAR, VARCHAR, TEXT	getString(String columnLabel) getString(int columnIndex)	String	rs.getString("nombre") rs.getString(1)
INTEGER, INT, SMALLINT	getInt(String columnLabel) getInt(int columnIndex)	int	rs.getInt("edad") rs.getInt(2)
BIGINT	getLong(String columnLabel) getLong(int columnIndex)	long	rs.getLong("poblacion") rs.getLong(3)
FLOAT, REAL	getFloat(String columnLabel) getFloat(int columnIndex)	float	rs.getFloat("altura") rs.getFloat(4)
DOUBLE, DECIMAL, NUMERIC	getDouble(String columnLabel) / getBigDecimal() getDouble(int columnIndex) / getBigDecimal(int columnIndex)	double / BigDecimal	rs.getDouble("salario") rs.getDouble(5)
BOOLEAN, BIT	getBoolean(String columnLabel) getBoolean(int columnIndex)	boolean	rs.getBoolean("activo") rs.getBoolean(6)
DATE	getDate(String columnLabel) getDate(int columnIndex)	java.sql.Date	rs.getDate("fecha_nacimiento") rs.getDate(7)
TIME	getTime(String columnLabel) getTime(int columnIndex)	java.sql.Time	rs.getTime("hora") rs.getTime(8)
TIMESTAMP, DATETIME	getTimestamp(String columnLabel) getTimestamp(int columnIndex)	java.sql.Timestamp	rs.getTimestamp("ultima_actualizacion") rs.getTimestamp(9)
BLOB	getBlob(String columnLabel) getBlob(int columnIndex)	java.sql.Blob	rs.getBlob("imagen") rs.getBlob(10)
CLOB	getClob(String columnLabel) getClob(int columnIndex)	java.sql.Clob	rs.getClob("descripcion_larga") rs.getClob(11)
ARRAY	getArray(String columnLabel) getArray(int columnIndex)	java.sql.Array	rs.getArray("valores") rs.getArray(12)
NULL	(Usar) wasNull()	boolean (indica si fue NULL)	if (rs.wasNull()) {...}
OBJECT (genérico)	getObject(String columnLabel) getObject(int columnIndex)	Object	rs.getObject("dato") rs.getObject(13)



Acceso a Datos

U2: Manejo de conectores

Ejemplo de uso de getters usando el nombre del campo:

```
ResultSet rs = stmt.executeQuery("SELECT id, nombre, edad, salario, activo FROM empleados");

while (rs.next()) {
    int id = rs.getInt("id");
    String nombre = rs.getString("nombre");
    int edad = rs.getInt("edad");
    double salario = rs.getDouble("salario");
    boolean activo = rs.getBoolean("activo");

    if (rs.wasNull()) {
        System.out.println("Uno de los campos era NULL");
    }

    System.out.printf("%d - %s - %d años - %.2f€ - Activo: %b\n",
        id, nombre, edad, salario, activo);
}
```

Ejemplo de uso de getters usando la posición del campo:

```
ResultSet rs = stmt.executeQuery("SELECT id, nombre, edad, salario, activo FROM empleados");

while (rs.next()) {
    int id = rs.getInt(1);           // Columna 1 → id
    String nombre = rs.getString(2); // Columna 2 → nombre
    int edad = rs.getInt(3);         // Columna 3 → edad
    double salario = rs.getDouble(4); // Columna 4 → salario

    System.out.printf("%d - %s - %d años - %.2f€\n", id, nombre, edad, salario);
}
```

Algunas indicaciones a tener muy presentes:

1. Si el valor de una columna en la base de datos es **NULL**, el **getter** devolverá el valor por defecto del tipo (por ejemplo, 0 para **int**, **false** para **boolean**, **null** para **String**), y puedes verificar si era realmente **NULL** usando **rs.wasNull()**.
2. Se puede usar **getObject()** para obtener un valor sin importar su tipo, pero luego deberás hacer un *cast* al tipo apropiado.
3. Los nombres de las columnas **no son sensibles a mayúsculas/minúsculas**, pero deben coincidir con los del **ResultSet**.

La clase **ResultSet** también dispone de unas constantes que definen diferentes tipos de desplazamientos dentro del **ResultSet**. Estas constantes se usan al crear los objetos **Statement** o **PreparedStatement** para configurar cómo se accede y se **mueve el cursor dentro de un ResultSet**.

Para eso, el API JDBC define dos grupos de constantes muy importantes:

1. Tipos de desplazamiento (scroll type)
2. Tipos de concurrencia (concurrency type)



Acceso a Datos

U2: Manejo de conectores

1. Tipos de desplazamientos

Las constantes de este tipo determinan cómo se puede mover el cursor dentro del `ResultSet`. Por defecto, un `ResultSet` solo se puede recorrer hacia adelante, pero *JDBC* permite otros modos más flexibles.

Constante	Descripción	Características principales
<code>ResultSet.TYPE_FORWARD_ONLY</code>	Solo hacia adelante	<ul style="list-style-type: none">• Es el tipo por defecto.• El cursor se mueve únicamente con <code>next()</code>.• No se puede volver atrás ni reposicionarse.• Es el más rápido y eficiente.
<code>ResultSet.TYPE_SCROLL_INSENSITIVE</code>	Desplazamiento libre (insensible a cambios)	<ul style="list-style-type: none">• Permite moverse hacia adelante y atrás:<ul style="list-style-type: none">◦ <code>next()</code>◦ <code>previous()</code>◦ <code>first()</code>◦ <code>last()</code>◦ <code>absolute(int)</code>,◦ <code>beforeFirst()</code>,◦ <code>afterLast()</code>, etc.• El <code>ResultSet</code> no refleja cambios en la base de datos hechos después de ejecutar la consulta.
<code>ResultSet.TYPE_SCROLL_SENSITIVE</code>	Desplazamiento libre (sensible a cambios)	<ul style="list-style-type: none">• Igual que el anterior, pero el <code>ResultSet</code> refleja cambios hechos en la BD después de la consulta (si el driver lo soporta).• Es más costoso en rendimiento y no todos los drivers lo implementan.

Ejemplo de uso de constante asociada al tipo de desplazamiento:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);
ResultSet rs = stmt.executeQuery("SELECT * FROM empleados");
// Puedes moverte libremente:
rs.last();
```



Acceso a Datos

U2: Manejo de conectores

```
System.out.println("ID del empleado del último registro: " + rs.getInt("id"));

rs.first();
System.out.println("Nombre del empleado del primer registro: " + rs.getString("nombre"));

rs.absolute(3);
System.out.println("Nombre del empleado del tercer registro: " + rs.getString("nombre"));
```

2. Tipos de concurrencia

Estas constantes indican si el conjunto de resultados es modificable o solo de lectura. Se refieren a la posibilidad de usar métodos como `updateString()`, `updateRow()`, `insertRow()`, o `deleteRow()` sobre el `ResultSet`.

Constante	Descripción	Características principales
<code>ResultSet.CONCUR_READ_ONLY</code>	Solo lectura	<ul style="list-style-type: none">No se pueden modificar los datos a través del <code>ResultSet</code>.Es el tipo por defecto.Más eficiente y común.
<code>ResultSet.CONCUR_UPDATABLE</code>	Modificable	<ul style="list-style-type: none">Permite actualizar, insertar o borrar filas directamente desde el <code>ResultSet</code> usando los métodos:<ul style="list-style-type: none"><code>updateXXX()</code>,<code>updateRow()</code>,<code>insertRow()</code>,<code>deleteRow()</code>, etc.El driver JDBC y la base de datos deben soportarlo.Suele ser más lento y con más restricciones (por ejemplo, no puede provenir de un <code>JOIN</code> complejo).

Ejemplo de uso de constante asociada al tipo de desplazamiento y al tipo de concurrencia:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE
);

ResultSet rs = stmt.executeQuery("SELECT id, nombre, salario FROM empleados");
// Actualizar datos directamente
if (rs.next()) {
    double salarioActual = rs.getDouble("salario");
    rs.updateDouble("salario", salarioActual + 500);
    rs.updateRow(); // Aplica el cambio en la BD
}
```



Acceso a Datos

U2: Manejo de conectores

No todos los drivers JDBC soportan todos los tipos de desplazamiento o concurrencia. Puedes verificar si lo soporta con:

```
DatabaseMetaData tiposSoporte = conexion.getMetaData();

boolean soporta = tiposSoporte.supportsResultSetConcurrency(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Si el tipo solicitado no se soporta, el driver puede degradarlo automáticamente al tipo más básico (`TYPE_FORWARD_ONLY`, `CONCUR_READ_ONLY`).

```
import java.sql.*;

public class EjemploResultSetEditable {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/empresa";
        String user = "root";
        String password = "1234";

        try (Connection conn = DriverManager.getConnection(url, user, password)) {

            // Crear un Statement desplazable y actualizable
            Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE, // desplazamiento libre
                ResultSet.CONCUR_UPDATABLE      // permite editar datos
            );

            // Ejecutar la consulta
            ResultSet rs = stmt.executeQuery("SELECT id, nombre, salario FROM empleados");

            // Leer los datos existentes
            System.out.println("=== Empleados actuales ===");
            while (rs.next()) {
                System.out.printf("%d - %s - %.2f€\n",
                    rs.getInt("id"),
                    rs.getString("nombre"),
                    rs.getDouble("salario"));
            }

            // Moverse al principio y actualizar un salario
            rs.beforeFirst(); // volver al inicio
            if (rs.next()) {
                double nuevoSalario = rs.getDouble("salario") + 500;
                rs.updateDouble("salario", nuevoSalario);
                rs.updateRow(); // aplica el cambio
                System.out.println("\nSalario actualizado para el primer empleado.");
            }

            // Insertar un nuevo registro
            rs.moveToInsertRow(); // crea una nueva fila vacía
            rs.updateInt("id", 100);
            rs.updateString("nombre", "Nuevo empleado");
            rs.updateDouble("salario", 2500.0);
            rs.insertRow(); // inserta en la BD
            System.out.println("Nuevo empleado insertado.");

            // Eliminar un registro específico
            rs.beforeFirst();
```



Acceso a Datos

U2: Manejo de conectores

```
while (rs.next()) {
    if (rs.getString("nombre").equals("Juan")) {
        rs.deleteRow(); // elimina la fila actual
        System.out.println("Empleado 'Juan' eliminado.");
    }
}

//Volver a mostrar los datos actualizados
rs = stmt.executeQuery("SELECT id, nombre, salario FROM empleados");
System.out.println("\n=== Datos actualizados ===");
while (rs.next()) {
    System.out.printf("%d - %s - %.2f€\n",
        rs.getInt(1),
        rs.getString(2),
        rs.getDouble(3));
}

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

Consideraciones

- **No todas las consultas** son actualizables.
- Debe cumplir condiciones como:
 - Provenir de **una sola tabla**.
 - Contener una **clave primaria** o columna que identifique unívocamente cada fila.
 - No incluir **JOIN**, **GROUP BY**, ni funciones agregadas (**SUM**, **AVG**, etc.).

Métodos de modificación de **ResultSet**

Método	Descripción	Uso típico / Ejemplo
<code>updateXXX(String columnLabel, XXX value)</code>	Actualiza el valor de una columna en la fila actual en memoria.	<code>rs.updateString("nombre", "Juan");</code>
<code>updateXXX(int columnIndex, XXX value)</code>	Igual que el anterior, pero usando índice de columna.	<code>rs.updateDouble(3, 2500.0);</code>
<code>updateRow()</code>	Aplica los cambios hechos con <code>updateXXX()</code> en la fila actual a la base de datos.	Después de <code>updateXXX()</code> : <code>rs.updateRow();</code>
<code>insertRow()</code>	Inserta en la base de datos la fila actual previamente preparada con <code>moveToInsertRow()</code> y <code>updateXXX()</code> .	<code>rs.moveToInsertRow();</code> <code>rs.updateInt("id", 101);</code> <code>rs.updateString("nombre", "Ana");</code> <code>rs.insertRow();</code>
<code>moveToInsertRow()</code>	Mueve el cursor a una fila especial para preparar un nuevo registro .	Ver ejemplo anterior.
<code>deleteRow()</code>	Elimina la fila actual de la base de datos.	<code>if(rs.getInt("id")==5)</code> <code>rs.deleteRow();</code>
<code>refreshRow()</code>	Refresca los valores de la fila actual	<code>rs.refreshRow();</code>



Acceso a Datos

U2: Manejo de conectores

Método	Descripción	Uso típico / Ejemplo
	desde la base de datos, descartando cambios locales no aplicados.	
<code>cancelRowUpdates()</code>	Cancela los cambios realizados en la fila actual antes de llamar a <code>updateRow()</code> .	<code>rs.updateDouble("salario", 3000.0); rs.cancelRowUpdates();</code>
<code>rowInserted()</code>	Devuelve <code>true</code> si la fila actual fue insertada por este <code>ResultSet</code> .	<code>if(rs.rowInserted()) {...}</code>
<code>rowUpdated()</code>	Devuelve <code>true</code> si la fila actual fue modificada por este <code>ResultSet</code> .	<code>if(rs.rowUpdated()) {...}</code>
<code>rowDeleted()</code>	Devuelve <code>true</code> si la fila actual fue eliminada por este <code>ResultSet</code> .	<code>if(rs.rowDeleted()) {...}</code>

5.3. PreparedStatement

En los apartados anteriores se han creado sentencias *SQL* a partir de cadenas de caracteres en las que se iban concatenando los datos necesarios para construir la sentencia completa. Sin embargo, para prevenir las **inyecciones SQL** y mejorar el rendimiento, se utilizan sentencias preparadas. Para ello se usará la interfaz `PreparedStatement`.

Las sentencias preparadas de JDBC permiten la “precompilación” del código SQL antes de ser ejecutado, permitiendo consultas o actualizaciones más eficientes. En el momento de compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados. Este proceso, obviamente, consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.

Con ellas, en lugar de concatenar cadenas para formar la sentencia SQL, se utilizan parámetros usando marcadores de posición (*placeholder*) que representarán los datos que serán asignados más tarde. El marcador de posición será representado en la sentencia SQL mediante el símbolo (?).

Una sentencia *INSERT* sobre “departamentos” podría representarse así:

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

Cada marcador de posición tiene un índice, la primera ? tiene el índice 1, la segunda el índice 2, ...

Pero antes de ejecutar un `PreparedStatement` es necesario asignar los datos a cada una de las interrogaciones para que cuando se ejecute la sentencia, la base de datos asigne variables de unión con estos datos y ejecute la orden SQL.



Acceso a Datos

U2: Manejo de conectores

Los pasos a seguir cuando se utilizan consultas preparadas son:

1. Asignar a una sentencia preparada la consulta SQL. Consulta de los empleados del departamento 10

```
PreparedStatement pstmt = (PreparedStatement)conexion.prepareStatement("SELECT * FROM empleados WHERE dept_no = ? ");
```

2. Establecer los valores de los parámetros mediante métodos `set()`.

```
pstmt.setInt(1, 10);
```

3. Ejecutar la consulta utilizando el método apropiado `executeQuery()`, `executeUpdate()` o `execute()` todos ellos sin parámetros.

```
ResultSet result = pstmt.executeQuery();
```

El ejemplo de inserción de una fila en la tabla Departamentos quedaría así:

```
//construir una orden INSERT indicando los marcadores
String sql= "INSERT INTO departamentos VALUES (?, ?, ?)";
PreparedStatement pstmt = (PreparedStatement)conexion.prepareStatement(sql);

//establecemos los parámetros mediante métodos set
pstmt.setInt(1, dep); //número departamento
pstmt.setString(2, dnombre); //nombre
pstmt.setString(3, loc); //localidad

//ejecutamos la consulta de actualización
int filas = pstmt.executeUpdate(); //filas afectadas
```

A continuación se muestra una tabla organizada por tipo de dato y método `set()` correspondiente para la asignación de un valor a un parámetro de la `PreparedStatement`:

Tipo de dato en Java	Método PreparedStatement	Notas / Consideraciones
int	<code>setInt(int parameterIndex, int value)</code>	Índice empieza en 1.
long	<code>setLong(int parameterIndex, long value)</code>	Útil para BIGINT en bases de datos.
short	<code>setShort(int parameterIndex, short value)</code>	Menos usado, para SMALLINT.
byte	<code>setByte(int parameterIndex, byte value)</code>	Para TINYINT.
boolean	<code>setBoolean(int parameterIndex, boolean value)</code>	Convierte a BIT/BOOLEAN según DB.
float	<code>setFloat(int parameterIndex, float value)</code>	Precisión simple.
double	<code>setDouble(int parameterIndex, double value)</code>	Precisión doble.
String	<code>setString(int parameterIndex, String value)</code>	Para CHAR, VARCHAR, TEXT.



Acceso a Datos

U2: Manejo de conectores

Tipo de dato en Java	Método PreparedStatement	Notas / Consideraciones
Date (java.sql.Date)	setDate(int parameterIndex, Date value)	Solo fecha (YYYY-MM-DD).
Time (java.sql.Time)	setTime(int parameterIndex, Time value)	Solo hora (HH:MM:SS).
Timestamp (java.sql.Timestamp)	setTimestamp(int parameterIndex, Timestamp value)	Fecha y hora (YYYY-MM-DD HH:MM:SS).
byte[]	setBytes(int parameterIndex, byte[] value)	Para BLOB / VARBINARY.

6. Procedimientos almacenados

Un procedimiento almacenado es un conjunto de sentencias, almacenadas en el servidor con un nombre determinado, y que los programas cliente pueden ejecutar simplemente indicando el nombre del procedimiento y pasándole opcionalmente los parámetros necesarios.

Estos procedimientos suelen ser de dos clases:

- Procedimientos almacenados. Realizan una tarea concreta con o sin parámetros.
- Funciones almacenadas. Realizan una tarea con o sin parámetros y devuelven un valor que se puede emplear en otras sentencias SQL

El conjunto de sentencias almacenadas en el servidor se encuentra enlazado (compilado) y optimizado para su ejecución repetida; lo que brinda grandes ventajas de rendimiento y una disminución importante de carga de tráfico en la red.

Además de optimizar el rendimiento del entorno, el uso de procedimientos almacenados permite incrementar las medidas de seguridad de un entorno de bases de datos, ya que elimina la necesidad de dar permisos directos a las tablas por parte de aplicaciones y clientes, y en su lugar se asignan permisos para ejecutar los procedimientos almacenados en el servidor.

El estándar SQL establece la sintaxis de las sentencias que hay que utilizar para definir procedimientos o funciones almacenados, una sintaxis que es implementada con notables diferencias en cada SGDBR.

Un procedimiento almacenado típico tiene:

- un nombre
- una lista de parámetros
- sentencias SQL
- sentencias condicionales y/o repetitivas extensión de SQL
- declaración de variables



Acceso a Datos

U2: Manejo de conectores

En el caso de MySQL:

- los procedimientos almacenados pueden definirse con parámetros de entrada (IN), de salida (OUT), de entrada/salida (INOUT) o sin ningún parámetro.
- Las funciones no pueden recibir parámetros de salida (OUT) ni de entrada/salida (INOUT).
- Cuando se omite el tipo de parámetro, se supone IN.

EJEMPLO. Procedimiento que sube el sueldo a los empleados de un departamento. El procedimiento recibe dos parámetros de entrada que son el número de departamento y el % de subida como valor entero.

```
DELIMITER //
CREATE PROCEDURE subida_salario (dep INT, subida INT)
BEGIN
UPDATE empleados
SET salario=salario + (salario*subida/100)
WHERE dept_no=dep;
END //
DELIMITER ;
```

Para invocar al procedimiento basta poner su nombre y pasar dos parámetros de entrada que representen al departamento y el % de subida en formato entero.

```
call subida_salario(10, 5); //ejecutar el procedimiento para el departamento 10 con subida del 5%
```

La invocación desde un cliente gráfico de MySQL como Workbench, sería de la siguiente forma:

```
CALL subida_salario(10,5);
```

EJEMPLO. Procedimiento almacenado que recibe el número de un departamento y muestra los datos de los empleados de ese departamento. También calcula el total de empleados y deja ese total en un parámetro de salida.

```
DELIMITER //
CREATE PROCEDURE emple_depar (IN dep INT, OUT cuenta INT)
BEGIN
SELECT * FROM empleados
WHERE dept_no=dep;

SELECT COUNT(*) INTO cuenta
FROM empleados
WHERE dept_no=dep;
END //
DELIMITER ;
```



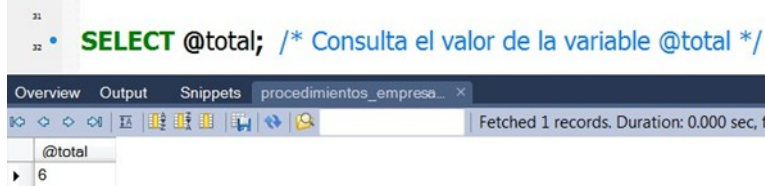
Acceso a Datos

U2: Manejo de conectores

La invocación desde un cliente gráfico de MySQL como Workbench, sería de la siguiente forma:

```
CALL emple_depar(10, @total); /* Ejecuta el procedimiento. @total variable para el parámetro de salida*/
```

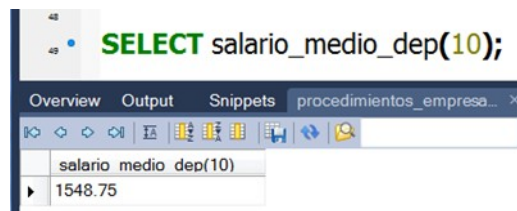
Para recuperar el valor del parámetro de salida desde un cliente de MySQL se consulta la variable @total:



EJEMPLO. Función que devuelve el salario medio de los empleados de un departamento. La función recibe el código o número del departamento.

```
DELIMITER //  
CREATE FUNCTION salario_medio_dep (dep INT)  
RETURNS DOUBLE  
DETERMINISTIC  
BEGIN  
DECLARE salario_medio DOUBLE;  
SELECT AVG (salario) INTO salario_medio  
FROM empleados  
WHERE dept_no=dep;  
RETURN salario_medio;  
END //  
DELIMITER ;
```

La ejecución de la función desde un cliente MySQL sería:





Acceso a Datos

U2: Manejo de conectores

6.1. CallableStatement

La interfaz `CallableStatement`, que hereda de `PreparedStatement`, permite que se puedan llamar desde Java a los procedimientos almacenados.

Para crear un objeto de este tipo se llama al método `prepareCall(String)` de un objeto `Connection`.

El siguiente ejemplo declara la llamada al procedimiento `subida_salario` que tiene dos parámetros y para indicar los parámetros se utilizan marcadores de posición (?) numerados internamente desde 1 en adelante y que se corresponden en el mismo orden con los parámetros del procedimiento.

```
String sql= "{call subida_salario (?, ?)}";  
CallableStatement procAlmacenado = conexion.prepareCall(sql);
```

Hay 4 formas de declarar las llamadas a los procedimientos y las funciones que dependen del uso u omisión de parámetros, y de la devolución de valores. Son las siguientes:

- `{call nombre_procedimiento}`

Para un procedimiento almacenado sin parámetros.

- `{?= call nombre_funcion}`

Para una función almacenada que devuelve un valor y no recibe parámetros, el valor se recibe a la izquierda del igual y es el primer parámetro.

- `{call nombre_procedimiento (?, ?, ...)}`

Para un procedimiento almacenado que recibe parámetros.

- `{?=call nombre_funcion(?, ?, ...)}`

Para una función almacenada que devuelve un valor (primer parámetro) y recibe varios parámetros.

Otras consideraciones a tener en cuenta al utilizar `CallableStatement`:

- Los valores de los parámetros de entrada se establecen mediante métodos `setter` heredados de `PreparedStatement`.
- Los tipos de los parámetros de salida y valores devueltos deben ser registrados antes de ejecutar el procedimiento almacenado. Para realizar el registro debe usarse el método `registerOutParameter(int indice, int tipoSQL)`. El primer parámetro es la posición y el segundo es una constante definida en la clase `java.sql.Types`.

Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR en la base de datos, en la llamada al método se escribe:

```
procAlmacenado.registerOutParameter(2, Types.VARCHAR);
```



Acceso a Datos

U2: Manejo de conectores

Una vez ejecutada la llamada al procedimiento, los valores de los parámetros OUT e INOUT se obtienen en los métodos `getXXX(indice)` similares a los utilizados para obtener los valores de las columnas en un `ResultSet`.

- Una sentencia `CallableStatement` puede devolver uno o más objetos `ResultSet`, que habrá que procesar adecuadamente.
- Para ejecutar una sentencia `CallableStatement` se usará el método `execute()` o `executeUpdate()`.

EJEMPLO. El siguiente ejemplo ejecuta el procedimiento `emple_depar` (IN `depar` INT, OUT `cuenta` INT). Recuerda que este procedimiento al ejecutarse, también devuelve un `ResultSet`.

```
public class CallableStatementOUT {
    //valores de las variables
    static int depar=10, totEmple=0;
    /* parámetros de conexión con la base de datos MySql */
    private static final String url = "jdbc:mysql://localhost/empresa";
    private static final String user = "root";
    private static final String password = "";

    public static void main(String[] args)
    {
        /* objeto de conexión */ Connection conexion = null;

        /* objeto ResultSet para recoger los registros devueltos */ ResultSet result = null;
        try
        {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            /* abre la conexión */
            conexion = DriverManager.getConnection(url, user, password);

            /*construir la orden de llamada al procAlmacenado*/
            String sql="{call emple_depar (?,?)}";
            /* 1. asignar la llamada al procAlmacenado a una CallableStatement*/
            CallableStatement procAlmacenado = conexion.prepareCall(sql);

            /*2. establecer el valor de los parámetros del procedimiento almacenado*/
            procAlmacenado.setInt(1, depar); //primer parámetro

            /*3. registro del parámetro de salida OUT*/
            procAlmacenado.registerOutParameter(2, Types.INTEGER);

            /*4. ejecutar el procedimiento almacenado*/
            procAlmacenado.execute();

            /*5. Obtener el valor de los parámetros de salida*/
            totEmple=procAlmacenado.getInt(2);
            System.out.println("Total de empleados departamento "+ depar +" son: " +
            totEmple);
            /*6 Obtener el ResultSet devuelto*/
            result=procAlmacenado.getResultSet();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```




Acceso a Datos

U2: Manejo de conectores

```
/*7. Procesar el resultSet*/
System.out.println("\nNumEmp, apellido,      oficio, salario ");
while (result.next())
{
    System.out.printf("%4d %-15s %-15s %6.2f \n", result.getInt(1),
        result.getString(2), result.getString(3), result.getFloat("salario"));
}
}
catch (SQLException ex)
{
    System.out.println(ex.getMessage());
}
finally
{
    /*liberar recursos*/
    try
    {
        conexion.close();
    }
    catch (SQLException exq)
    {
        System.out.println(exq.getMessage());
    }
}
}
} //main
} //class
```

7. ResultSetMetaData

Se pueden obtener metadatos (datos sobre datos) a partir de un [ResultSet](#) mediante la interfaz [ResultSetMetaData](#): número de columnas devueltas, el tipo de las columnas, el nombre,...

Para ello, se usará el método [getMetaData\(\)](#) del objeto [ResultSet](#) que devuelve una referencia a un objeto [ResultSetMetaData](#) con el que se obtendría información acerca de las columnas devueltas.

Algunos métodos interesantes son:

Método	Descripción
getColumnCount()	Devuelve el número de columnas devueltas por la consulta.
getColumnName(indice columna)	Devuelve el nombre de la columna.
getColumnTypeName(indice)	Devuelve el nombre del tipo de dato que contiene la columna específico del sistema de bases de datos.
isNullable(indice)	Devuelve 0 si la columna puede contener valores nulos.



Acceso a Datos

U2: Manejo de conectores

El siguiente segmento de código muestra el uso de la interfaz para conocer más información acerca de las columnas devueltas por la consulta ***SELECT * FROM departamentos***. En este caso, al usar ***SELECT **** se desconoce el nombre de las columnas devueltas.

```
public class EjemploResultSetMetaData
{
    public static void main(String[] args)
    {
        try
        {
            Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo",
                "root", "");

            Statement sentencia = conexion.createStatement();
            ResultSet rs = sentencia.executeQuery("SELECT * FROM departamentos");

            ResultSetMetaData rsmd = rs.getMetaData();

            int nColumnas = rsmd.getColumnCount();
            String nula;
            System.out.printf("Número de columnas recuperadas: %d\n", nColumnas);

            for (int i = 1; i <= nColumnas; i++)
            {
                System.out.printf("Columna %d: %n ", i);
                System.out.printf("  Nombre: %s %n    Tipo: %s %n ", rsmd.getColumnName(i),
                    rsmd.getColumnTypeName(i));

                if (rsmd.isNullable(i) == 0)
                    nula = "NO";
                else
                    nula = "SI";

                System.out.printf("  Puede ser nula?: %s %n ", nula);

            } // for

            sentencia.close();
            rs.close();
            conexion.close();
        }
        catch (SQLException e)
        {
            System.out.println(e.getMessage());
        }
    }
} // fin de main
```

8. Ejecución de sentencias de descripción de datos.

Normalmente, cuando desarrollamos una aplicación JDBC, conocemos las estructuras de las tablas y datos que estamos manejando, es decir, conocemos las columnas que tienen y cómo están relacionadas entre sí, etc.

Sin embargo, es posible que no conozcamos la estructura de las tablas de una base de datos. En este caso la información de la base de datos la podemos obtener a través de los metaobjetos, que no son más que objetos que proporcionan información sobre la base de datos.



Acceso a Datos

U2: Manejo de conectores

8.1. DatabaseMetaData

La interface `DatabaseMetaData` proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información.

El método `getMetaData()` de la interface `Connection`, devuelve un objeto `DatabaseMetaData` con el que se obtiene información de la base de datos.

EJEMPLO 4. Conectamos con la base de datos empresa y se muestra información sobre el producto de base de datos, el driver, la URL para acceder a la base de datos, el nombre de usuario y las tablas y las vistas del esquema actual.

```
public class EjemploDataBaseMetaData
{
    private static final String url = "jdbc:mysql://localhost/empresaz";
    private static final String user = "root";
    private static final String password = "";

    public static void main(String[] args)
    {
        /* variables locales */
        String nombre, driver, url_name, usuario, catalogo, esquema, tabla, tipo;

        /* objeto de conexión */
        Connection conexion = null;

        /* objeto ResultSet para recoger los registros devueltos por la consulta */
        ResultSet result = null;

        /* objeto de metadatos */
        DatabaseMetaData dbmd;
        try
        {
            //establece conexión
            conexion = DriverManager.getConnection(url, user, password);

            //crea objeto de metadatos y obtiene información
            dbmd = conexion.getMetaData();

            nombre = dbmd.getDatabaseProductName();
            driver = dbmd.getDriverName();
            url_name = dbmd.getURL();

            usuario = dbmd.getUserName();

            /* imprime valores */
            System.out.println("INFORMACIÓN SOBRE LA BASE DE DATOS:");
            System.out.println("=====");
            System.out.println("Nombre: " + nombre);
            System.out.println("Driver: " + driver);
            System.out.println("URL: " + url_name);
            System.out.println("Usuario: " + usuario + "\n");

            /* obtiene la información de las tablas y vistas existentes */
            result = dbmd.getTables(null, "empresaz", null, null);

            /* mientras puede avanzar, asigna valores obtenidos */
            while (result.next())
            {
                catalogo = result.getString(1);
```



Acceso a Datos

U2: Manejo de conectores

```
        esquema = result.getString(2);
        tabla = result.getString(3);
        tipo = result.getString(4);

        /* imprime */
        System.out.println(tipo + " -Catálogo: " + catalogo + ", Esquema: "
            + esquema + ", Nombre: " + tabla);
    }
}
catch (SQLException ex)
{
    System.out.println(ex.getMessage());
}
}
```

Algunos métodos de interés:

- `getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)`

Devuelve un objeto `ResultSet` que proporciona información sobre las tablas y vistas de la base de datos. Cada fila de `ResultSet` que devuelve `getTables()` tiene información sobre una tabla.

Necesita 4 parámetros:

- **Parámetro 1: catálogo de la base de datos.**

El método obtiene las tablas del catálogo indicado. Si se establece a valor **null**, indicamos todos los catálogos (o catálogo actual dependiendo del SGBD).

- **Parámetro 2: esquema de la base de datos.**

Obtiene las tablas del esquema indicado. Si se establece a valor **null**, indica el esquema actual (o todos los esquemas dependiendo del SGBD).

- **Parámetro 3: es un patrón en el que se indica el nombre de las tablas que queremos que obtenga el método.**

Se puede utilizar el carácter guión bajo o porcentaje. Por ejemplo “de%” obtendría todas las tablas cuyo nombre empieza por “de”.

- **Parámetro 4: es un array de String, en el que indicamos qué tipos de tablas queremos TABLE (para tablas), VIEW (para vistas).**

Si se establece a null, nos devolverá todos los tipos ya sean tablas o vistas. El siguiente ejemplo nos devolvería sólo tablas:



Acceso a Datos

U2: Manejo de conectores

Las columnas del ResultSet que devuelve el método son:

1. TABLE_CAT. El nombre del catálogo al que pertenece la tabla.
2. TABLE_SCHEM. Nombre del esquema al que pertenece la tabla
3. TABLE_NAME. El nombre de la tabla o vista.
4. TABLE_TYPE. El tipo TABLE o VIEW
5. REMARKS. Comentarios
6. •

Para obtener los resultados del ejemplo también podríamos haber puesto en el código anterior:

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

- `getColumns(String catalogo, String esquema, String nombre_tabla, String nombre_columna)`

Devuelve un [ResultSet](#) con información sobre las columnas de una tabla o tablas.

Para el nombre de la tabla y de la columna se pueden utilizar los caracteres comodín `_` y `%`. El valor null en los 4 parámetros indica que obtiene información de todas las columnas y tablas del esquema actual.

El siguiente ejemplo muestra información sobre todas las columnas de la tabla departamentos:

```
System.out.println("COLUMNAS TABLA DEPARTAMENTOS:");
System.out.println("=====");

/* obtiene las columnas de la tabla 'Departamentos' */
result = dbmd.getColumns(null, "empresa", "departamentos", null);

/* mientras puede avanzar */
while (result.next())
{
    /* asigna valores */
    nombreCol = result.getString("COLUMN_NAME");
    tipoCol = result.getString("TYPE_NAME");
    tamCol = result.getString("COLUMN_SIZE");
    nula = result.getString("IS_NULLABLE");

    /* imprime */
    System.out.println(" Columna: " + nombreCol + ", Tipo: "
        + tipoCol + ", Tamaño: " + tamCol + ", ¿Puede ser Nula?: "+ nula);
}
```

El resultado se puede ver en la siguiente imagen:

```
COLUMNAS TABLA DEPARTAMENTOS:
=====
Columna: dept_no, Tipo: TINYINT, Tamaño: 3, ¿Puede ser Nula?: NO
Columna: dnombre, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula?: YES
Columna: loc, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula?: YES
```



Acceso a Datos

U2: Manejo de conectores

- `getPrimaryKeys (String catalogo, String esquema, String tabla)`

Devuelve un `ResultSet` con la lista de columnas que forman la clave primaria.

El siguiente ejemplo muestra la clave primaria de la tabla “departamentos”

```
ResultSet result = dbmd.getPrimaryKeys(null, "empresa", "departamentos");
pkDep = "";
separador = "";
while (result.next())
{
    pkDep += separador + result.getString("COLUMN_NAME");
    separador = "+";
}
System.out.println("Clave primaria: " + pkDep);
```

- `getExportedKeys (String catalogo, String esquema, String tabla)`

Devuelve un `ResultSet` con la lista de todas las claves foráneas que utilizan la clave primaria de esta tabla.

El siguiente ejemplo muestra las tablas y sus claves ajenas que referencian a la tabla “departamentos”, en este caso solo la tabla “empleados”

```
result = dbmd.getExportedKeys(null, "empresa", "departamentos");
while (result.next())
{
    fk_name = result.getString("FKCOLUMN_NAME");
    pk_name = result.getString("PKCOLUMN_NAME");
    fk_table = result.getString("FKTABLE_NAME");
    pk_table = result.getString("PKTABLE_NAME");

    System.out.println("Clave ajena: " + fk_name + " en la tabla " + fk_table);
    System.out.println("Clave primaria: " + pk_name + " en la tabla " + pk_table);
}
```

- `getImportedKeys (String catalogo, String esquema, String tabla).`

Devuelve un `ResultSet` con la lista de claves ajenas existentes en la tabla. Se utiliza igual que el método anterior.

- `getProcedures (String catalogo, String esquema, String procedure)`

Devuelve un `ResultSet` con la lista de procedimientos almacenados.

El siguiente ejemplo mostraría los procedimientos y funciones que tiene el esquema de nombre “empresa”

```
result = dbmd.getProcedures(null, "empresa", null);
```



Acceso a Datos

U2: Manejo de conectores

```
while (result.next())
{
    proc_name = result.getString("PROCEDURE_NAME");
    proc_type = result.getString("PROCEDURE_TYPE");
    System.out.println("Nombre Procedimiento: " + proc_name + " -Tipo " + proc_type);
}
```

9. Pool de Conexiones

Un pool de conexiones consiste en tener un conjunto de conexiones ya realizadas a la base de datos que podrán ser reutilizadas en distintas peticiones. Es un mecanismo que administra y reutiliza las conexiones para mejorar el rendimiento y evitar la sobrecarga de abrir y cerrar conexiones repetidamente. En lugar de crear nuevas conexiones cada vez que una es requerida, mediante el pool se le asigna una conexión disponible, y una vez termina de usarla la conexión no se cierra, sino que se devuelve al pool para ser reutilizada. Si todas las conexiones están ocupadas, el pool puede crear nuevas (hasta un límite) o hacer que los usuarios esperen.

Los pool de conexiones se usan principalmente en aplicaciones que necesitan gestionar un gran número de conexiones y se emplea sobre todo en sistemas que requieren acceso frecuente y concurrente a bases de datos.

Para crear un pool de conexiones es necesario el uso de frameworks como HikariCP (uno de los más usados en java), Apache DBCP o Tomcat JDBC Pool (integrado en servidores Tomcat).

9.1. Apache Commons DBCP (DBCP2)

DBCP2 (Apache Commons DBCP versión 2) es una implementación de connection pooling para JDBC que gestiona un conjunto de conexiones a la base de datos reutilizables.

Evita el coste de crear/cerrar conexiones repetidamente y ofrece control sobre:

- tamaño de pool.
- tiempos de espera.
- pruebas de conexión.
- detección de conexiones abandonadas,
- etc.

Para usarlo en un proyecto gradle, hay que añadir la dependencia siguiente:

```
implementation group: 'org.apache.commons', name: 'commons-dbcp2', version: '2.13.0'
```




Acceso a Datos

U2: Manejo de conectores

9.1.1. Configuración básica de un DataSource con BasicDataSource

BasicDataSource es la clase más simple para crear un **DataSource** administrado por DBCP2.

Se puede configurar por código o por propiedades. Las propiedades importantes:

Método	Descripción
<code>setDriverClassName(String)</code>	nombre de clase del driver JDBC.
<code>setUrl(String)</code>	JDBC URL.
<code>setUsername, setPassword.</code>	Para establecer el nombre de usuario y el password de conexión a la base de datos.
<code>setInitialSize(int)</code>	Número de conexiones a crear cuando se inicie el pool. Es opcional. Si no se indica el valor por defecto es 0.
<code>setMaxTotal(int)</code>	Establece el número máximo de conexiones en el pool.
<code>setMaxIdle(int)</code>	Establece el número máximo de conexiones inactivas.
<code>setMinIdle(int)</code>	Establece el número mínimo de conexiones inactivas.
<code>setMaxWaitMillis(long)</code>	Establece cuánto esperar al pedir una conexión antes de lanzar excepción.
<code>setValidationQuery(String)</code> <code>setDefaultAutoCommit(false)</code> y usar <code>setTestWhileIdle/testOnBorrow</code>	o para validar conexiones.
<code>setRemoveAbandonedOnBorrow(true)</code> y <code>setRemoveAbandonedTimeout(int)</code>	para detectar/conexiones abandonadas

9.2. Hikari CP

HikariCP es un pool de conexiones JDBC ligero, rápido y fiable pensado para producción. Su objetivo es permitir con alta concurrencia y consumo de memoria bajo. Es la opción por defecto en muchos proyectos modernos cuando se busca rendimiento.

Para su configuración se usa la clase **HikariConfig** cuyos métodos más destacables son:

Método	Descripción
<code>setPoolName</code>	Asigna un nombre al pool.
<code>setJdbcUrl</code>	JDBC URL.
<code>setUsername, setPassword.</code>	Para establecer el nombre de usuario y el password de conexión a la base de datos.
<code>setMaximumPoolSize(int)</code>	Establece el número máximo de conexiones en el pool.
<code>setMinimumIdle(int)</code>	Establece el número mínimo de conexiones inactivas.



Acceso a Datos

U2: Manejo de conectores

Método	Descripción
setIdleTimeout(long)	Establece el tiempo de espera antes de liberar conexiones. Se establece en ms.
setConnectionTimeout(long)	Establece el tiempo máximo de espera para obtener conexión. Se establece en ms. Si alcanza dicho tiempo lanza excepción.
setMaxLifetime(long)	Establece el tiempo máximo de vida de una conexión. Se establece en ms.

También ofrece la clase [HikariDataSource](#) para obtener el [DataSource](#). Esta clase recibe como parámetro un objeto de tipo [HikariConfig](#).

```
private static HikariDataSource dataSource;  
dataSource = new HikariDataSource(config);
```

Muy importante hacer uso del método [close\(\)](#) de [HikariDataSource](#) para para liberar recursos al cerrar la app.

Por último, una clase muy útil es [HikariPoolMXBean](#) que permite obtener información sobre el estado del pool. Para obtener un objeto de este tipo se debe usar el método [getHikariPoolMXBean\(\)](#) del [dataSource](#) (objeto instancia de [HikariDataSource](#)).

Entre sus métodos se pueden destacar:

Método	Descripción
getTotalConnections()	Devuelve el número de conexiones totales del pool.
getActiveConnections()	Devuelve el número de conexiones activas del pool.
getIdleConnections()	Devuelve el número de conexiones inactivas del pool.
getThreadsAwaitingConnection()	Devuelve el número de conexiones en espera.