# JAVA JDBC

Antes de comenzar con JDBC vamos a instalar una base de datos local para poder trabajar con ella.

## Instalación de Xampp.

### Ventajas de usar MySQL localmente con XAMPP:

1. **Más fácil de configurar**: No tienes que preocuparte por la conectividad remota ni por problemas de red.
2. **Control total**: Tendrás acceso total a la base de datos para probar y aprender.
3. **Rendimiento más rápido**: La base de datos localmente será más rápida que conectarse a un servidor remoto.
4. **Ideal para desarrollo**: Puedes trabajar sin depender de un servidor externo y hacer pruebas rápidamente.

### Pasos para configurar una base de datos local con XAMPP

1. **Descargar e instalar XAMPP**:
   - Ve al sitio web oficial de XAMPP: https://www.apachefriends.org/index.html.
   - Descarga la versión de XAMPP para Windows (o para el sistema operativo que estés usando).
   - Sigue el asistente de instalación y asegúrate de instalar **Apache** y **MySQL** (que son los que necesitarás).
2. **Iniciar MySQL en XAMPP**:
   - Una vez instalado XAMPP, abre el **Panel de Control de XAMPP**.
   - Haz clic en **Start** al lado de **MySQL** para iniciar el servidor MySQL.
   - Si todo está correcto, el estado de MySQL debe mostrar "Running".
3. **Acceder a phpMyAdmin**:
   - En el Panel de Control de XAMPP, también puedes hacer clic en el botón **Admin** al lado de MySQL.
   - Esto abrirá el navegador y te llevará a **phpMyAdmin** (la interfaz gráfica para gestionar bases de datos MySQL).
   - Desde allí, puedes crear bases de datos, usuarios y tablas.
4. **Crear una base de datos**:
   - En **phpMyAdmin**, haz clic en la pestaña **Bases de datos**.
   - Crea una nueva base de datos, por ejemplo, test_db.

# JDBC

JDBC (Java Database Connectivity) es una API que permite a las aplicaciones Java conectarse a bases de datos y ejecutar consultas SQL.

## Agregar el driver JDBC

Cada base de datos tiene su propio driver JDBC. Debes descargarlo e incluirlo en tu proyecto. Algunos ejemplos de drivers son:

- **MySQL:** mysql-connector-java
- **PostgreSQL:** postgresql-42.x.x.jar
- **SQL Server:** mssql-jdbc

MySQL :: Download Connector/J

## Establecer conexión con la base de datos

Creamos un archivo **ConexionMySQL.java**

Para conectarnos a nuestra base de datos local debemos configurar correcta los datos de conexión.

```
String url = "jdbc:mysql://localhost:3306/test_db";
String usuario = "root";  // Por defecto el usuario es 'root'
String contraseña = "";  // Por defecto, no hay contraseña para 'root' en XAMPP
```

## Pasos generales con JDBC:

Cuando trabajas con JDBC para conectar una aplicación Java a una base de datos MySQL, los pasos generalmente son los mismos. Aquí te los resumo para que puedas ver cómo funcionan en otros contextos:

1. **Importar las clases necesarias:** Siempre necesitarás importar las clases que permiten manejar la conexión y excepciones relacionadas con SQL:

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

2. **Definir los parámetros de conexión:** Dependiendo de la base de datos con la que trabajes, solo necesitas ajustar estos parámetros:

- **URL de la base de datos:** La URL sigue el formato general jdbc:mysql://<host>:<puerto>/<nombre_base_datos>.
- **Usuario:** El usuario con el que te conectas (por ejemplo, root en MySQL local, o un usuario que hayas creado en tu base de datos).
- **Contraseña:** La contraseña asociada al usuario.

**3. Cargar el driver (si es necesario):** En versiones recientes de JDBC, el driver puede cargarse automáticamente, pero es una buena práctica incluir este paso. Usualmente se hace con la línea:

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

Esto solo es necesario si tu aplicación no lo hace automáticamente, pero para versiones más antiguas de MySQL o si estás configurando algo manualmente, es útil.

4. **Establecer la conexión:** Usando el **DriverManager**, te conectas a la base de datos:

```
Connection conn = DriverManager.getConnection(url, usuario, contraseña);
```

Aquí **url** es la URL de tu base de datos, **usuario** es el nombre de usuario y **contraseña** es la contraseña que le corresponde.

**5. Manejar excepciones:** Es importante manejar los posibles errores, como problemas de conexión o de carga de clases. Para ello, utilizamos un bloque **try-catch**:

```
try {
    // Conexión exitosa
} catch (SQLException | ClassNotFoundException e) {
    // Manejo de errores
}
```

**6. Cerrar la conexión:** Una vez que hayas terminado de trabajar con la base de datos, debes cerrar la conexión para liberar recursos:

```
conn.close();
```

Puedes ver el archivo java para la conexión a base de datos en el Anexo 1, mientras te familiarizar con Jdbc te puede servir de guía.

Compilar nuestro archivo de conexión:

```
javac -cp ".;mysql-connector-java-9.2.0.jar" ConexionMySQL.java
```

Ejecutamos y comprobamos que hemos establecido la conexión correctamente:

```
java -cp ".;mysql-connector-java-9.2.0.jar" ConexionMySQL
```

En este comando:

- **.** significa el directorio actual (donde están tus archivos .java o .class).
- **mysql-connector-java-9.2.0.jar** es el archivo JAR con el conector de MySQL que necesitas para conectar tu aplicación Java con la base de datos.

Ahora que ya hemos probado que nuestra conexión a la base de datos funciona, continuamos como sigue:

La mejor práctica es **separar la lógica de conexión a la base de datos** de la lógica de negocio o de las operaciones específicas que vas a realizar con la base de datos. Esto hace que tu código sea más modular, reutilizable y fácil de mantener.

### ¿Cómo organizarlo?

1. **Crea un archivo Java para la conexión a la base de datos:** Este archivo se encargará solo de establecer la conexión con la base de datos y proporcionar un objeto Connection que pueda ser usado en otros archivos. ConexionMySQL.java, modificamos nuestro archivo anterior eliminando la clase main, ya que no es necesaria.

2. **Utiliza el archivo de conexión en tu aplicación principal o en otros archivos:** En los archivos donde necesitas interactuar con la base de datos (por ejemplo, para insertar datos, hacer consultas, etc.), **importas** y utilizas el archivo ConexionMySQL para obtener la conexión.

Así queda nuestro archivo de conexión:

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

public class ConexionMySQL {

    public static Connection getConexion() {
        // Parámetros de conexión
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contraseña = "mi_contraseña";
        Connection conn = null;

        try {
            // Cargar el driver de MySQL
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establecer la conexión con la base de datos
            conn = DriverManager.getConnection(url, usuario, contraseña);

            System.out.println("Conexión exitosa a la base de datos.");

        } catch (SQLException | ClassNotFoundException e) {
            System.out.println("Error de conexión: " + e.getMessage());
        }

        return conn;
    }
}
```

Este archivo proporciona el método getConexion() que devuelve una conexión a la base de datos. Este método se puede invocar en cualquier otra clase que necesite una conexión.

## *Archivo Main.java (para la lógica de negocio):*

En tu archivo principal o en cualquier clase que necesite acceder a la base de datos, importas ConexionMySQL y usas el método getConexion() para obtener la conexión.

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

// Importar la clase ConexionMySQL
import ConexionMySQL;

public class Main {

    public static void main(String[] args) {
        // Obtener la conexión utilizando el método getConexion() de ConexionMySQL
        Connection conn = ConexionMySQL.getConexion();

        if (conn != null) {
            try {
                // Hacer algo con la conexión (por ejemplo, una consulta)
                Statement stmt = conn.createStatement();
                String sql = "SELECT * FROM tabla_example";
                stmt.executeQuery(sql);

                // Aquí iría la lógica para trabajar con la base de datos
            } catch (SQLException e) {
                System.out.println("Error al interactuar con la base de datos: " + e.getMessage());
            } finally {
                try {
                    if (conn != null) {
                        conn.close();  // Cerrar la conexión cuando ya no sea necesaria
                    }
                } catch (SQLException e) {
                    System.out.println("Error al cerrar la conexión: " + e.getMessage());
                }
            }
        }
    }
}
```

### Ventajas de separar la lógica de la conexión:

1. **Modularidad:** El código relacionado con la base de datos se mantiene separado del resto de la lógica de la aplicación. Esto hace que sea más fácil modificar o actualizar la parte de la conexión sin afectar otras partes de tu aplicación.
2. **Reutilización:** Puedes reutilizar la clase ConexionMySQL en otros proyectos o en diferentes partes de tu aplicación si necesitas hacer una conexión a la base de datos.
3. **Mantenimiento:** Si en algún momento necesitas cambiar los parámetros de la conexión (como la URL o las credenciales), solo tendrás que modificar la clase ConexionMySQL. Todo el código que usa la conexión no se verá afectado.

4. **Mejor organización:** Mantener el código organizado facilita el desarrollo y la depuración, ya que no tendrás que buscar la lógica de conexión entre el código de las consultas o la lógica de negocio.

A continuación, te proporcionaré ejemplos de cómo realizar las operaciones **CRUD** (Crear, Leer, Actualizar, Borrar) con JDBC, utilizando la conexión que ya hemos establecido.

## 1. Consultar datos (READ)

Para consultar datos de una base de datos, utilizamos la sentencia SQL SELECT y luego procesamos los resultados obtenidos con un ResultSet.

```
// Leer datos de la base de datos
String sql = "SELECT id, nombre, edad FROM empleados"; // Consulta SQL
Statement stmt = conn.createStatement();  // Crear un Statement para ejecutar la consulta
ResultSet rs = stmt.executeQuery(sql);    // Ejecutar la consulta

while (rs.next()) {  // Iterar sobre los resultados
    int id = rs.getInt("id");
    String nombre = rs.getString("nombre");
    int edad = rs.getInt("edad");

    // Imprimir los resultados
    System.out.println("ID: " + id + ", Nombre: " + nombre + ", Edad: " + edad);
}
```

Explicación:

- **ResultSet rs = stmt.executeQuery(sql)**: Ejecuta la consulta y almacena los resultados en un ResultSet.
- **rs.next()**: Mueve el cursor al siguiente registro del ResultSet. Retorna true si hay más resultados, de lo contrario false.
- **rs.getInt("columna")**: Recupera los valores de las columnas por su nombre o índice. Puedes usar otros métodos como getString, getDouble, etc., según el tipo de dato.

## 2. Insertar datos (CREATE)

Para insertar datos, utilizamos la sentencia SQL INSERT INTO y un PreparedStatement para evitar inyecciones SQL.

```
// Insertar un nuevo registro
String sql = "INSERT INTO empleados (nombre, edad) VALUES (?, ?)";  // Consulta SQL con parámetros
PreparedStatement pstmt = conn.prepareStatement(sql);  // Preparar la sentencia

// Asignar valores a los parámetros
pstmt.setString(1, "Juan Pérez");  // Primer parámetro: nombre
pstmt.setInt(2, 28);               // Segundo parámetro: edad

int filasAfectadas = pstmt.executeUpdate();  // Ejecutar la inserción

System.out.println("Filas insertadas: " + filasAfectadas);
```

Explicación:

- **PreparedStatement pstmt = conn.prepareStatement(sql)**: Crea una sentencia preparada con la consulta INSERT. Los ? se usan como marcadores de parámetros.
- **pstmt.setString(1, "Juan Pérez")**: Asigna el valor "Juan Pérez" al primer parámetro ? en la consulta.
- **pstmt.executeUpdate()**: Ejecuta la inserción. Devuelve el número de filas afectadas (en este caso, la cantidad de registros insertados).

## 3. Actualizar datos (UPDATE)

Para actualizar datos en la base de datos, usamos la sentencia SQL UPDATE y un PreparedStatement.

```
// Actualizar un registro
String sql = "UPDATE empleados SET edad = ? WHERE nombre = ?";  // Consulta SQL con parámetros
PreparedStatement pstmt = conn.prepareStatement(sql);  // Preparar la sentencia

// Asignar valores a los parámetros
pstmt.setInt(1, 30);          // Primer parámetro: edad
pstmt.setString(2, "Juan Pérez");  // Segundo parámetro: nombre

int filasAfectadas = pstmt.executeUpdate();  // Ejecutar la actualización

System.out.println("Filas actualizadas: " + filasAfectadas);
```

Explicación:

- **UPDATE empleados SET edad = ? WHERE nombre = ?**: La consulta SQL actualiza la columna edad de la tabla empleados donde el nombre coincide con el parámetro proporcionado.
- **pstmt.setInt(1, 30)**: Establece el valor de edad a 30.
- **pstmt.setString(2, "Juan Pérez")**: Establece el valor de nombre a "Juan Pérez".

## 4. Eliminar datos (DELETE)

Para eliminar registros, usamos la sentencia SQL DELETE y un PreparedStatement.

```
// Eliminar un registro
String sql = "DELETE FROM empleados WHERE nombre = ?";  // Consulta SQL con parámetros
PreparedStatement pstmt = conn.prepareStatement(sql);  // Preparar la sentencia

// Asignar valor al parámetro
pstmt.setString(1, "Juan Pérez");  // Eliminar el registro con nombre "Juan Pérez"

int filasAfectadas = pstmt.executeUpdate();  // Ejecutar la eliminación

System.out.println("Filas eliminadas: " + filasAfectadas);
```

Explicación:

- **DELETE FROM empleados WHERE nombre = ?**: La consulta SQL elimina el registro de la tabla empleados cuyo nombre coincide con el parámetro.
- **pstmt.setString(1, "Juan Pérez")**: Establece el nombre que será utilizado para buscar el registro a eliminar.

### Consideraciones adicionales:

**executeUpdate()**: Este método se usa para las consultas que modifican datos (como INSERT, UPDATE, DELETE). Devuelve el número de filas afectadas.

**executeQuery()**: Este método se usa para ejecutar consultas de solo lectura (como SELECT), y devuelve un ResultSet con los resultados.

**PreparedStatement vs Statement**: Usa PreparedStatement siempre que trabajes con consultas que involucren parámetros dinámicos. Es más seguro, ya que evita inyecciones SQL.

## Resumen de las operaciones CRUD:

1. **Crear (INSERT)**: pstmt.executeUpdate() con una consulta INSERT INTO.
2. **Leer (SELECT)**: stmt.executeQuery() con una consulta SELECT.
3. **Actualizar (UPDATE)**: pstmt.executeUpdate() con una consulta UPDATE.
4. **Borrar (DELETE)**: pstmt.executeUpdate() con una consulta DELETE.

## Resumen de uso de createStatement() y prepareStatement()

1. **createStatement()**:
   - **Uso**: Se utiliza cuando **no tienes parámetros** en tu consulta SQL o cuando la consulta es simple.
   - **Método**: Statement stmt = conn.createStatement();
   - **Consulta**: Es ideal para consultas estáticas, como consultas fijas que no requieren parámetros dinámicos.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM empleados");
```

- **Ventaja**: Más simple para consultas sin parámetros.
- **Desventaja**: Puede ser más vulnerable a inyecciones SQL si no se escapan los valores correctamente.

2. **prepareStatement()**:

- **Uso**: Se utiliza cuando la consulta SQL tiene parámetros dinámicos o cuando necesitas ejecutar la misma consulta varias veces con diferentes valores.
- **Método**: PreparedStatement pstmt = conn.prepareStatement("SQL con parámetros");
- **Consulta**: Es ideal para consultas con parámetros, como INSERT, UPDATE, DELETE o consultas SELECT con condiciones dinámicas.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM empleados WHERE nombre = ?");
pstmt.setString(1, "Juan Pérez");
ResultSet rs = pstmt.executeQuery();
```

- **Ventaja**: Evita inyecciones SQL, mejora el rendimiento al reutilizar la consulta con diferentes valores.
- **Desventaja**: Requiere un poco más de código debido a los parámetros.

Resumen:

1. **createStatement()**:
   - **Consulta simple sin parámetros**.
   - Ejemplo: Statement stmt = conn.createStatement();
   - **Uso**: Consultas fijas sin variables dinámicas.
   - **Método**: stmt.executeQuery() o stmt.executeUpdate().

2. **prepareStatement()**:
   - **Consulta con parámetros dinámicos**.
   - Ejemplo: PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM empleados WHERE nombre = ?");
   - **Uso**: Consultas donde los valores cambian, como INSERT, UPDATE, DELETE, o SELECT con WHERE dinámico.
   - **Método**: pstmt.executeQuery() o pstmt.executeUpdate().

## Cerrar la conexión.

Es una buena práctica cerrar la conexión después de realizar cualquier operación en la base de datos para liberar los recursos utilizados y evitar posibles fugas de memoria o bloqueos. Puedes implementar esta lógica de cierre de la conexión dentro de tu clase de **ConexionMySQL**, o en una clase de utilidad, para llamarla siempre que termines una operación.

### Pasos para manejar la conexión correctamente:

1. **Cerrar la conexión, el Statement y el ResultSet**: Después de realizar cualquier consulta, actualización, inserción o eliminación, debes cerrar:
   - La conexión a la base de datos.
   - El Statement o PreparedStatement utilizado para ejecutar la consulta.
   - El ResultSet (si lo estás utilizando) para evitar que sigan consumiendo recursos.

2. **Usar finally para asegurarse de que se cierren**: Es importante cerrar todos estos recursos en un bloque finally, para garantizar que se cierre incluso si ocurre una excepción.

Ejemplo de cómo manejar la conexión y cerrarla en ConexionMySQL:

Puedes agregar un método que se encargue de cerrar la conexión y los demás recursos. Aquí te dejo una forma de hacerlo:

*Modificación en ConexionMySQL:*

```
import java.sql.*;

public class ConexionMySQL {
    private static final String URL = "jdbc:mysql://localhost:3306/mi_base_de_datos";
    private static final String USER = "root";
    private static final String PASSWORD = "mi_contraseña";

    public static Connection getConexion() {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(URL, USER, PASSWORD);
            System.out.println("Conexión exitosa");
        } catch (SQLException e) {
            System.out.println("Error de conexión: " + e.getMessage());
        }
        return conn;
    }

    public static void cerrarConexion(Connection conn, Statement stmt, ResultSet rs) {
        try {
            if (rs != null) {
                rs.close();  // Cerrar ResultSet si es que se utilizó
            }
            if (stmt != null) {
                stmt.close();  // Cerrar Statement si es que se utilizó
            }
            if (conn != null) {
                conn.close();  // Cerrar conexión
            }
        } catch (SQLException e) {
            System.out.println("Error al cerrar recursos: " + e.getMessage());
        }
    }
}
```

¿Por qué cerrar la conexión y los recursos?

- **Eficiencia**: Mantener las conexiones abiertas innecesariamente consume recursos del sistema. Las bases de datos tienen un número limitado de conexiones simultáneas.
- **Evitar fugas de memoria**: Si no se cierran adecuadamente, los recursos como las conexiones, los Statements y los ResultSets siguen en memoria, lo que puede generar problemas a largo plazo.
- **Bloqueos**: Las conexiones abiertas sin liberar pueden llevar a bloqueos de base de datos o tiempos de espera excesivos si otras operaciones intentan acceder a esos recursos.

**Resumen**

1. **Agregar el driver JDBC** (descargar JAR).
2. **Conectarse a la base de datos** con DriverManager.getConnection().
3. **Ejecutar consultas SQL** con Statement o PreparedStatement.
4. **Cerrar conexiones** cuando ya no se necesiten.

Anexo 1.

```java
// Importar clases necesarias para manejar la conexión con la base de datos
import java.sql.Connection;   // Para manejar la conexión con la base de datos
import java.sql.DriverManager; // Para obtener la conexión a través de DriverManager
import java.sql.SQLException;  // Para manejar errores relacionados con SQL

// Definir la clase principal que contiene el método main
public class ConexionMySQL {

    // Método principal que se ejecuta al iniciar el programa
    public static void main(String[] args) {

        // Definir la URL de conexión a la base de datos (servidor, puerto y nombre de la base de datos)
        String url = "jdbc:mysql://localhost:3306/test_db";  // URL de conexión (ajustar según tu base de datos)

        // Usuario para la base de datos
        String usuario = "root";  // El usuario predeterminado en MySQL con XAMPP es "root"

        // Contraseña para la base de datos
        String contraseña = "";  // Por defecto, el usuario "root" no tiene contraseña en XAMPP

        try {
            // Intentar cargar el driver de MySQL (en versiones recientes puede no ser necesario, pero es una buena práctica)
            Class.forName("com.mysql.cj.jdbc.Driver"); // Registra el driver MySQL

            // Establecer la conexión con la base de datos
            Connection conn = DriverManager.getConnection(url, usuario, contraseña);

            // Si la conexión es exitosa, se muestra un mensaje
            System.out.println("Conexión exitosa a la base de datos!");

            // Cerrar la conexión después de usarla
            conn.close();
        } catch (SQLException | ClassNotFoundException e) {
            // Si ocurre un error en cualquier parte del proceso (conexión o carga del driver), se captura y muestra
            System.out.println("Error de conexión:");
            e.printStackTrace();  // Imprime el error completo
        }
    }
}
```

Explicación línea por línea:

1. **import java.sql.Connection;**
   Importa la clase **Connection**, que es responsable de representar una conexión activa con una base de datos. Usaremos esta clase para interactuar con la base de datos.
2. **import java.sql.DriverManager;**
   Importa la clase **DriverManager**, que maneja el registro de drivers de bases de datos y establece conexiones con ellos. Nos ayuda a obtener una conexión usando la URL de la base de datos, usuario y contraseña.

3. **import java.sql.SQLException;**
   Importa la clase **SQLException**, que se usa para manejar errores relacionados con SQL. Si algo sale mal durante la conexión o la ejecución de consultas, un **SQLException** se lanzará.
4. **public class ConexionMySQL {**
   Define una clase pública llamada **ConexionMySQL**. Esta clase contendrá el código necesario para conectar a la base de datos MySQL.
5. **public static void main(String[] args) {**
   El método **main** es el punto de entrada de la aplicación Java. Este es el método que se ejecutará cuando arranques tu programa.
6. **String url = "jdbc:mysql://localhost:3306/test_db";**
   Aquí defines la **URL de conexión** a la base de datos.
   - **jdbc:mysql://**: Especifica que estás usando el protocolo JDBC para conectarte a MySQL.
   - **localhost**: Indica que la base de datos está en tu computadora local.
   - **3306**: Es el puerto predeterminado de MySQL.
   - **test_db**: Es el nombre de la base de datos a la que te vas a conectar.
7. **String usuario = "root";**
   Define el **usuario** que se va a utilizar para autenticar la conexión a la base de datos. En muchos sistemas de desarrollo local (como XAMPP), el usuario por defecto es **root**.
8. **String contraseña = "";**
   Define la **contraseña** del usuario. En el caso de XAMPP, el usuario **root** no tiene contraseña, por lo que la dejamos vacía.
9. **try {**
   Inicia un bloque **try**, que se utiliza para intentar ejecutar el código que puede generar un error. En este caso, todo lo relacionado con la conexión a la base de datos se coloca dentro de este bloque.
10. **Class.forName("com.mysql.cj.jdbc.Driver");**
    Carga el **driver** de MySQL necesario para establecer la conexión. Esta línea no siempre es necesaria en versiones recientes del conector, ya que el driver puede registrarse automáticamente, pero es una buena práctica incluirlo. Este código dice a Java que cargue la clase **com.mysql.cj.jdbc.Driver** que contiene la implementación del driver JDBC para MySQL.
11. **Connection conn = DriverManager.getConnection(url, usuario, contraseña);**
    Aquí se establece la **conexión** con la base de datos usando la URL, el usuario y la contraseña previamente definidos. Si la conexión es exitosa, se almacena en el objeto **conn**.
12. **System.out.println("Conexión exitosa a la base de datos!");**
    Si la conexión se establece correctamente, se imprime el mensaje **"Conexión exitosa a la base de datos!"** en la consola.

13. **conn.close();**
    Después de usar la conexión, la cerramos usando **conn.close()** para liberar los recursos y evitar fugas de memoria.
14. **catch (SQLException | ClassNotFoundException e) {**
    Si ocurre un error durante la conexión o al intentar cargar el driver, se captura en el bloque **catch**. Aquí estamos manejando dos tipos de excepciones:
    - **SQLException**: Se lanza cuando hay un error relacionado con la base de datos (por ejemplo, problemas con la URL, el usuario o la contraseña).
    - **ClassNotFoundException**: Se lanza si no se puede encontrar el driver de MySQL en el classpath.
15. **System.out.println("Error de conexión:");**
    Si ocurre un error, se imprime el mensaje **"Error de conexión:"** para indicar que hubo un problema al intentar conectar a la base de datos.
16. **e.printStackTrace();**
    Este método imprime la **pila de llamadas** del error, es decir, proporciona detalles sobre el tipo de error que ocurrió, la línea en el código donde ocurrió y el flujo de ejecución que llevó a ese error. Es útil para depurar el problema.
17. **}**
    Cierra el bloque **catch** y el bloque **try**.
18. **}**
    Cierra el método **main**.
19. **}**
    Cierra la clase **ConexionMySQL**.