



UD02. JDBC: Conectividad Java con Bases de Datos

2º Curso CFGS Desarrollo de Aplicaciones Multiplataforma

Una guía completa para dominar la conectividad entre aplicaciones Java y bases de datos relacionales mediante JDBC

¿Qué es JDBC?



Conectividad Universal

JDBC (Java Database Connectivity) facilita la conexión a múltiples tipos de bases de datos relacionales como SQLite, H2, Apache Derby, MySQL, PostgreSQL, Oracle o SQL Server, proporcionando una interfaz estándar e independiente del proveedor.



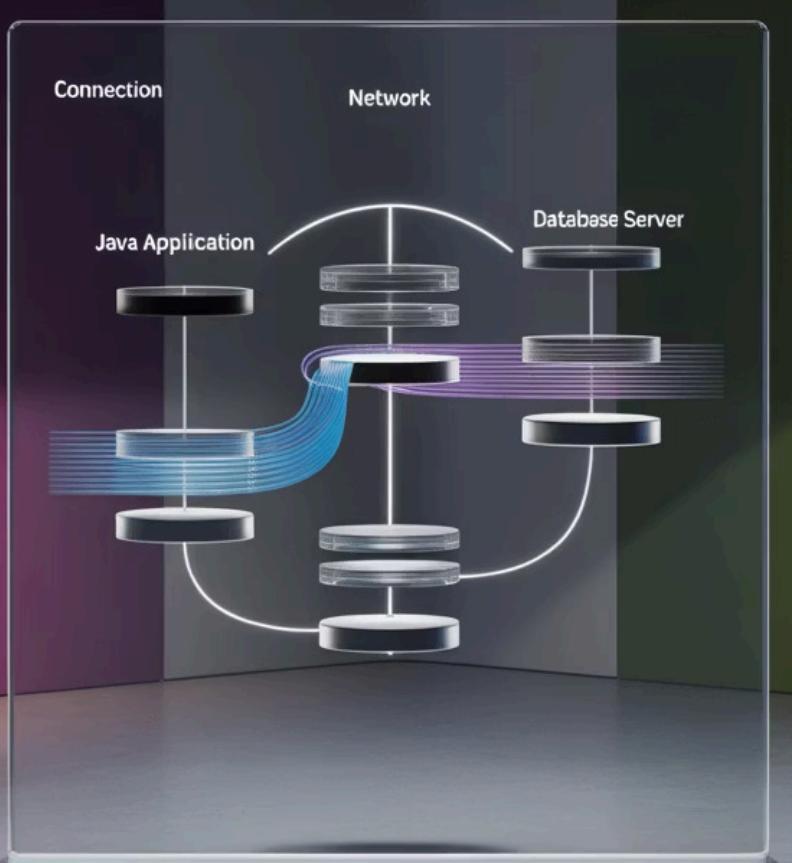
Ejecución SQL Integrada

Permite la ejecución de sentencias SQL directamente desde programas Java, posibilitando realizar consultas, modificaciones o la manipulación de la estructura de la base de datos mediante código Java.

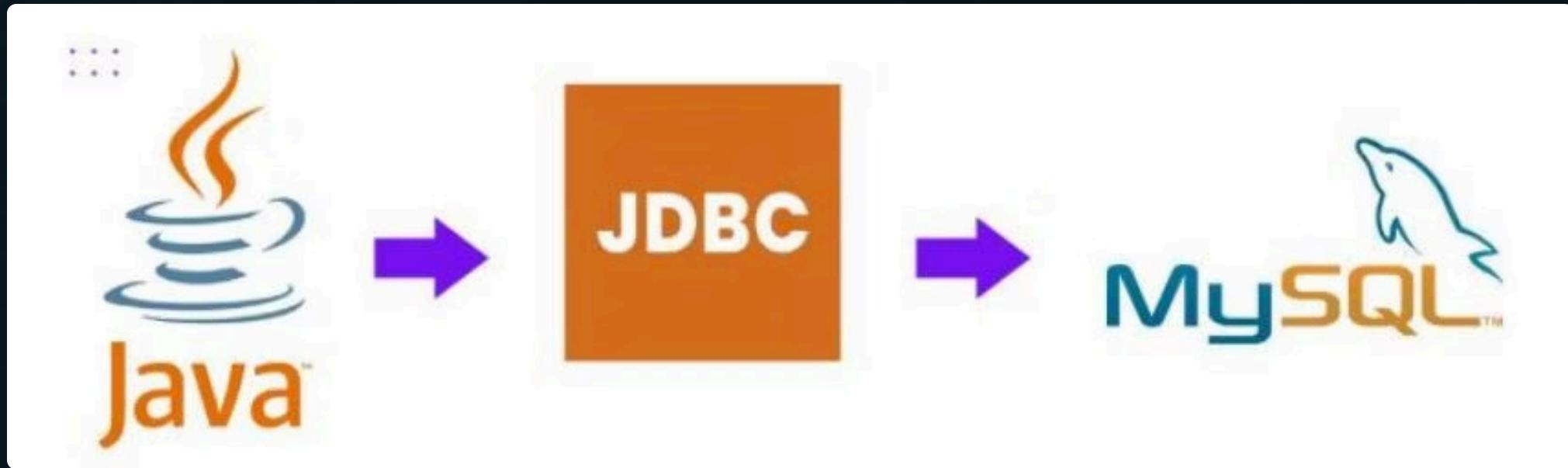


Driver JDBC

Es un controlador que actúa como intermediario entre la aplicación Java y la base de datos. Se trata de una librería .jar que podemos descargar desde la web oficial o incluir mediante gestores como Maven.



Arquitectura y Flujo de JDBC



Modelo Cliente-Servidor

La aplicación Java (cliente) utiliza las clases e interfaces de JDBC para interactuar con la base de datos, mientras que el Driver JDBC traduce las llamadas de la API a comandos comprensibles por el sistema de gestión de base de datos.

01

Establecer Conexión

Utilizando DriverManager y la URL de conexión específica para conectar con la base de datos objetivo.

02

Crear y Ejecutar Sentencias

Mediante Statement o PreparedStatement para ejecutar consultas SQL de forma segura y eficiente.

03

Procesar Resultados

Usando ResultSet para navegar y extraer datos de los resultados obtenidos.

04

Liberar Recursos

Cerrar conexión, Statement y ResultSet para liberar memoria (automático con try-catch-resources).

Acceso



Servidor de Base de Datos

Aplicación que almacena los datos y permite acceder y modificarlos manteniendo su seguridad.

Incluye únicamente una interfaz básica en modo consola.

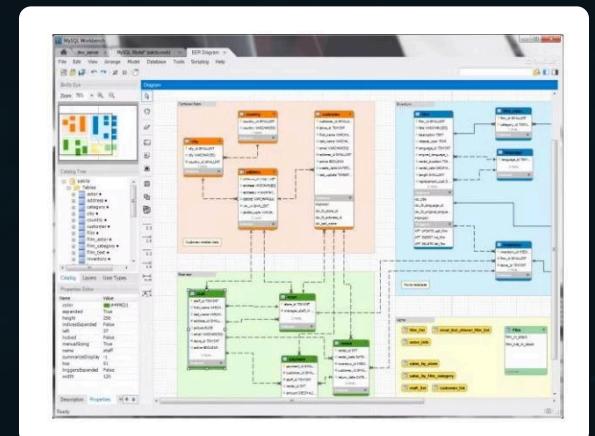
- [MySQL](#) - Sistema de gestión más popular
- [MariaDB](#) - Fork de MySQL con mejoras
- [PostgreSQL](#) - Base de datos objeto-relacional
- SQL Server, Oracle - Soluciones empresariales



Interfaz de Acceso

Aplicación que permite acceder al servidor de base de datos de forma más gráfica e intuitiva para facilitar la administración y consultas.

- MySQL Workbench - Herramienta oficial de MySQL
- DBeaver - Cliente universal multiplataforma
- phpMyAdmin - Interfaz web para MySQL
- HeidiSQL - Cliente ligero para Windows



Instalación del Driver JDBC



Descarga Manual

Descargar la librería JAR del conector desde la web oficial e incluirlo manualmente en el proyecto dentro del IDE (IntelliJ o cualquier otro).

Gestor de Dependencias

Mediante Maven buscando la dependencia en mvnrepository.com y copiando el código XML correspondiente al archivo pom.xml del proyecto. Este método es más profesional y mantenable.

- **Recomendación:** Utiliza siempre Maven o Gradle para gestionar dependencias en proyectos profesionales, ya que facilita las actualizaciones y la gestión de versiones.

Conexión desde Java

Parámetros de Conexión

Usuario

Nombre del usuario con privilegios adecuados en la base de datos o esquema específico que vamos a utilizar.

Contraseña

Contraseña correspondiente al usuario de base de datos para autenticación y acceso seguro.

URL de Conexión

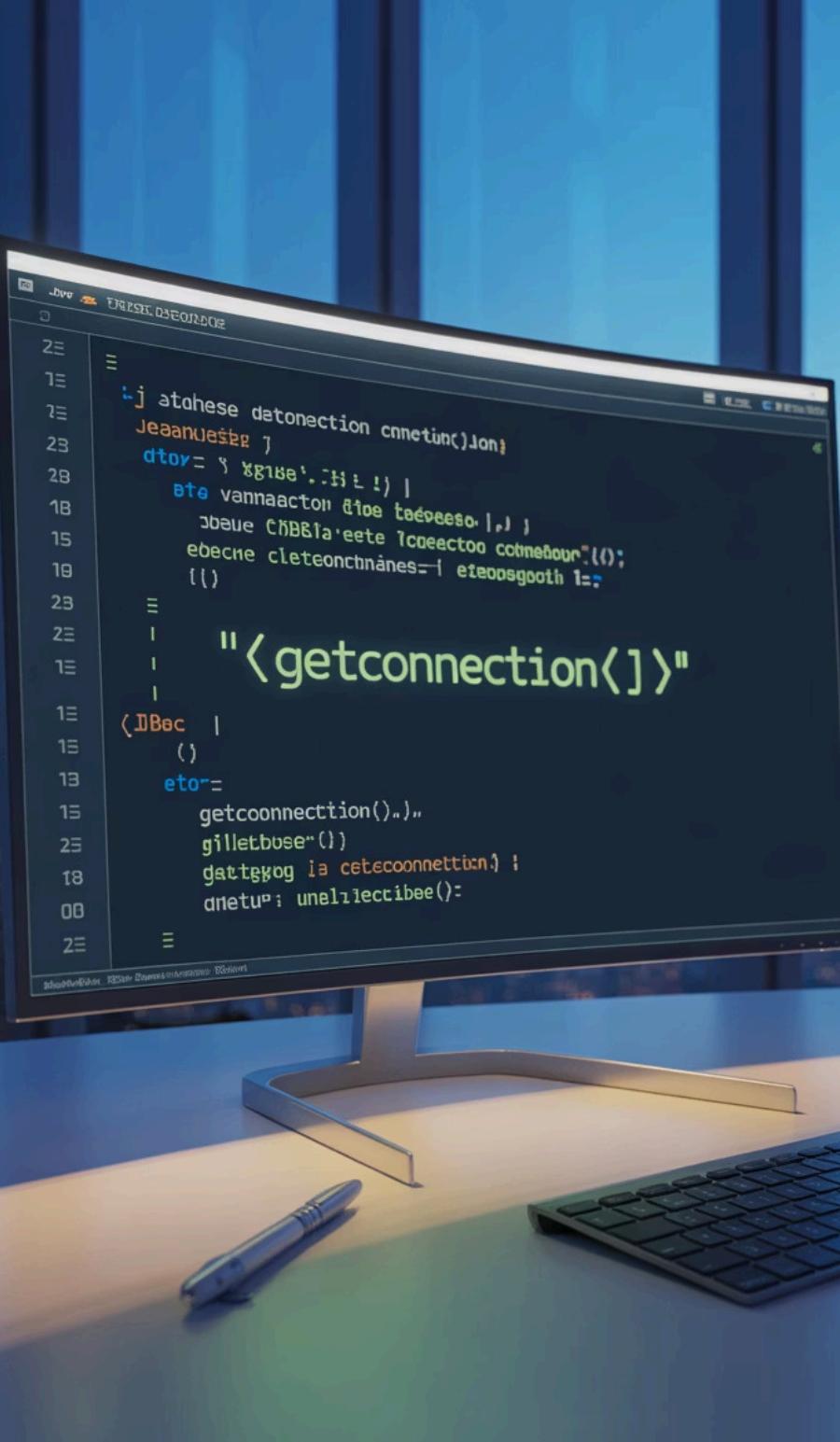
MySQL: jdbc:mysql://localhost:3306/nombreBaseDeDatos

MariaDB: jdbc:mariadb://localhost:3306/nombreBaseDeDatos

SQLite: jdbc:sqlite:D:/AD/SQLite/nombreBaseDeDatos.db

Oracle: jdbc:oracle:thin:@localhost:1521:XE

....



Ejecución de Consultas SQL

Utilizaremos la estructura **try-catch-resources** para que los recursos se cierren automáticamente y evitar memory leaks.



Connection

Representa una conexión activa a una base de datos. Se utiliza para enviar comandos SQL y recibir resultados del servidor.



Statement

Interfaz que permite enviar sentencias SQL sin parámetros a la base de datos. Ideal para consultas estáticas.



PreparedStatement

Similar a Statement, pero permite sentencias SQL con parámetros y ofrece mayor seguridad contra inyección SQL.



ResultSet

Objeto que representa los datos devueltos por una consulta SELECT y permite recorrer los resultados .



SQLException

Excepción que maneja errores durante la interacción con la base de datos, proporcionando información detallada del problema.

Statement vs PreparedStatement

Statement

Para consultas simples sin parámetros del usuario o sentencias DDL (Data Definition Language):

- Consultas estáticas predefinidas
- Creación/modificación de tablas
- Sin riesgo de inyección SQL

Métodos principales:

- `executeQuery(String sql)` - SELECT
- `executeUpdate(String sql)` - INSERT/UPDATE/DELETE
- `execute(String sql)` - DDL

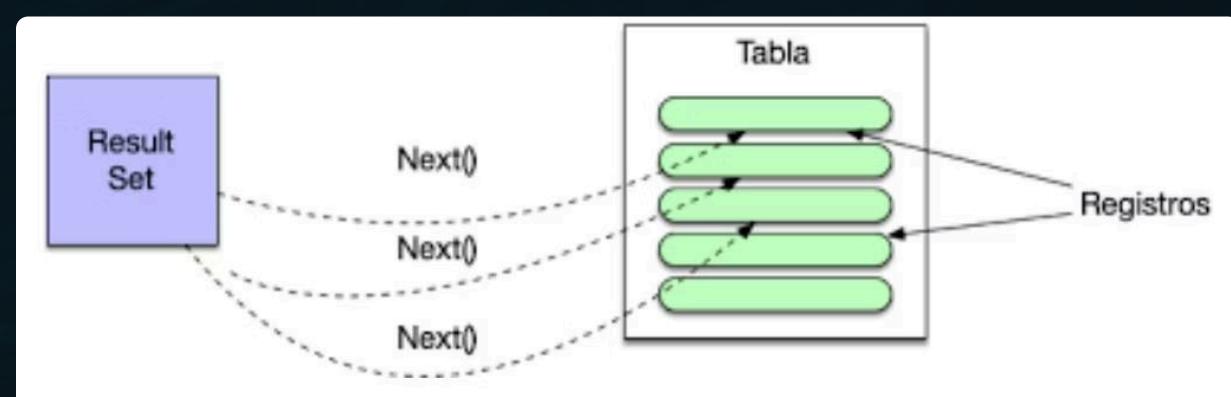
PreparedStatement

Para consultas con parámetros del usuario. **Obligatorio** para prevenir ataques de inyección SQL:

- Consultas con datos de usuario
- Mayor seguridad y rendimiento
- Reutilizable con diferentes parámetros

Métodos principales:

- `executeQuery()` - SELECT preparado
- `executeUpdate()` - INSERT/UPDATE/DELETE
- `execute()` - DDL preparado



ResultSet: Manejando Resultados de Consultas

ResultSet es una interfaz fundamental en JDBC que representa un conjunto de resultados obtenidos al ejecutar una consulta SQL (normalmente un SELECT). Proporciona métodos esenciales para navegar a través de los datos recuperados y extraer valores de las filas.

Navegación del Cursor en ResultSet

next()

Avanza el cursor al siguiente registro del ResultSet. Devuelve true si existe una fila válida y false si se llega al final.

first()

Mueve el cursor a la primera fila del ResultSet. Retorna true si hay una fila disponible y false si está vacío.

last()

Posiciona el cursor en la última fila del ResultSet. Devuelve true si hay una fila y false si el conjunto de resultados está vacío.

absolute(int row)

Mueve el cursor a una fila específica por su número de índice. Un número positivo cuenta desde el inicio, mientras que uno negativo cuenta desde el final.

relative(int rows)

Desplaza el cursor un número relativo de filas hacia adelante (positivo) o hacia atrás (negativo) desde la posición actual.

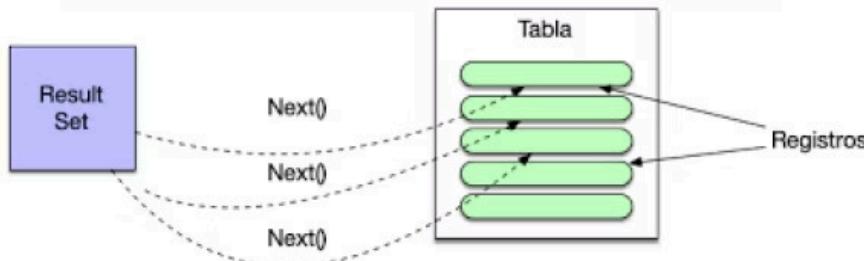
Métodos para Obtener Datos

Utiliza estos métodos para extraer los valores de las columnas de la fila actual del ResultSet, especificando el nombre de la columna.

- `getInt(String columnLabel)`: Recupera un valor entero.
- `getString(String columnLabel)`: Recupera un valor de cadena de texto.
- `getDouble(String columnLabel)`: Recupera un valor de tipo doble.
- `getDate(String columnLabel)`: Recupera un valor de fecha.
- `getBoolean(String columnLabel)`: Recupera un valor booleano.
- `getFloat(String columnLabel)`: Recupera un valor de tipo flotante.
- `getLong(String columnLabel)`: Recupera un valor de tipo largo (long).

Ejemplo

```
public void consultarDatos() {  
    try (Connection conn = ConexionBD.conectar();  
         Statement stmt = conn.createStatement()) {  
  
        String sql = "SELECT * FROM usuarios";  
        ResultSet rs = stmt.executeQuery(sql);  
  
        while (rs.next()) {  
            System.out.println("ID: " + rs.getInt("id"));  
            System.out.println("Nombre: " + rs.getString("nombre"));  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```



```
public void insertarUsuario(String nombre, String email) {  
    String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";  
  
    try (Connection conn = ConexionBD.conectar();  
         PreparedStatement pstmt = conn.prepareStatement(sql)) {  
  
        pstmt.setString(1, nombre);  
        pstmt.setString(2, email);  
        pstmt.executeUpdate();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Gestión de Excepciones y Transacciones

SQLException

Clase base para excepciones SQL que debe manejarse explícitamente.

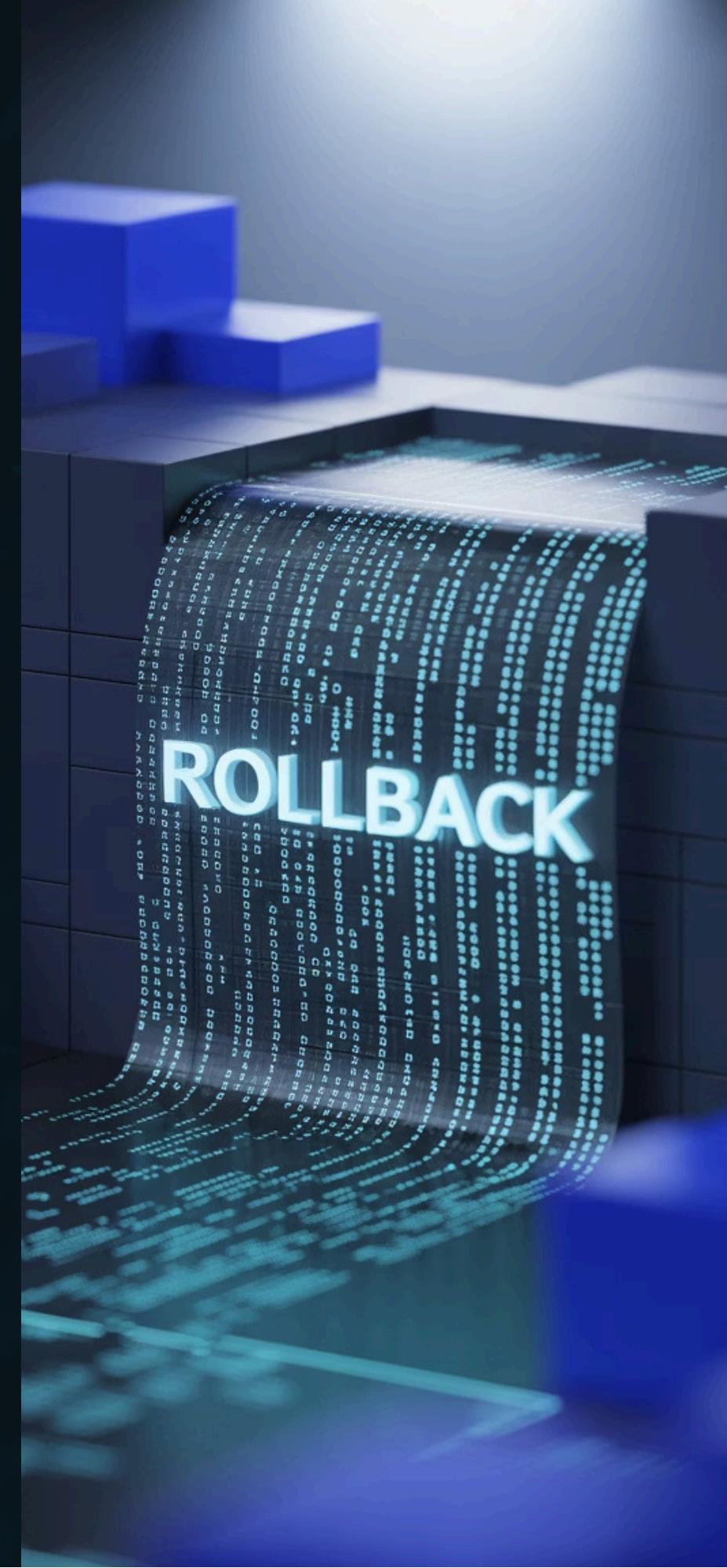
Métodos útiles: `getMessage()`, `getSQLState()` y `getErrorCode()` para obtener información detallada del error.

Transacciones ACID

Conjunto de operaciones que se ejecutan de forma atómica. Todas deben completarse con éxito para confirmar cambios, o hacer rollback si alguna falla para garantizar consistencia de datos.

```
try (Connection conn = ConexionBD.conectar()) {
    conn.setAutoCommit(false);

    try (Statement stmt = conn.createStatement()) {
        stmt.executeUpdate("UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1");
        stmt.executeUpdate("UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2");
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        e.printStackTrace();
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```



Patrón DAO y Mejores Prácticas

Data Access Object (DAO)

Patrón de diseño que proporciona una capa de abstracción entre la lógica de negocio y el acceso a datos, permitiendo cambiar la implementación de persistencia sin afectar el resto de la aplicación.

01

Interfaz DAO

Define métodos CRUD (Create, Read, Update, Delete) que implementarán todas las clases concretas.

02

Implementación Concreta

Clase que implementa la interfaz con código específico para cada sistema de base de datos.

03

Pool de Conexiones

Sistema para reutilizar conexiones abiertas en lugar de crear nuevas para cada operación, mejorando el rendimiento.

- **Próximo paso:** En la siguiente unidad aprenderemos Spring Data JPA, que automatizará gran parte de este proceso y gestionará el pool de conexiones por nosotros usando HikariCP.

PATRON DAO: EJEMPLO

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Usuario {  
    private int id;  
    private String nombre;  
    private String email;  
}
```

```
public class DatabaseConnection {  
    private final String url = "jdbc:mysql://localhost:3306/tu_base_de_datos";  
    private final String user = "tu_usuario";  
    private final String password = "tu_contraseña";  
  
    public Connection getConnection() throws SQLException {  
        return DriverManager.getConnection(url, user, password);  
    }  
}
```

```
public interface UsuarioDAO {  
    void agregarUsuario(Usuario usuario);  
    Usuario obtenerUsuarioPorId(int id);  
    List<Usuario> obtenerTodosLosUsuarios();  
    void actualizarUsuario(Usuario usuario);  
    void eliminarUsuario(int id);  
}
```

Se implementan los métodos del Interfaz

```
public class UsuarioDAOImpl implements UsuarioDAO {  
    private final DatabaseConnection databaseConnection;  
  
    public UsuarioDAOImpl(DatabaseConnection databaseConnection) {  
        this.databaseConnection = databaseConnection;  
    }  
  
    @Override  
    public void agregarUsuario(Usuario usuario) {  
        String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";  
        try (Connection conn = databaseConnection.getConnection();  
             PreparedStatement pstmt = conn.prepareStatement(sql)) {  
            pstmt.setString(1, usuario.getNombre());  
            pstmt.setString(2, usuario.getEmail());  
            pstmt.executeUpdate();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    @Override  
    public Usuario obtenerUsuarioPorId(int id) {  
        String sql = "SELECT * FROM usuarios WHERE id = ?";  
        try (Connection conn = databaseConnection.getConnection();  
             PreparedStatement pstmt = conn.prepareStatement(sql)) {  
            pstmt.setInt(1, id);  
            try (ResultSet rs = pstmt.executeQuery()) {  
                if (rs.next()) {  
                    return new Usuario(rs.getInt("id"),  
                                      rs.getString("nombre"),  
                                      rs.getString("email"));  
                }  
            }  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

PATRON DAO: EJEMPLO

- Nuestro programa principal queda mucho más limpio:

```
public static void main(String[] args) {  
    DatabaseConnection dbConnection = new DatabaseConnection();  
    UsuarioDAO usuarioDAO = new UsuarioDAOImpl(dbConnection);  
  
    // Agregar un nuevo usuario  
    Usuario nuevoUsuario = new Usuario(0, "Juan Pérez", "juan@example.com");  
    usuarioDAO.agregarUsuario(nuevoUsuario);  
  
    // Obtener un usuario por ID  
    Usuario usuario = usuarioDAO.obtenerUsuarioPorId(1);  
    System.out.println("Usuario: " + usuario);  
  
    // Listar todos los usuarios  
    List<Usuario> usuarios = usuarioDAO.obtenerTodosLosUsuarios();  
    System.out.println("Todos los usuarios: " + usuarios);  
  
    // Actualizar un usuario  
    if (usuario != null) {  
        usuario.setNombre("Juan Actualizado");  
        usuarioDAO.actualizarUsuario(usuario);  
    }  
  
    // Eliminar un usuario  
    usuarioDAO.eliminarUsuario(1);  
}
```

PATRON DAO: VARIAS IMPLEMENTACIONES DEL INTERFAZ

- Si quisieramos hacer varias implementaciones para utilizar varias bases de datos (o para cambiar de una a otra) se haría de esta forma:

```
public interface UsuarioDAO {  
    void agregarUsuario(Usuario usuario);  
    Usuario obtenerUsuarioPorId(int id);  
    List<Usuario> obtenerTodosLosUsuarios();  
    void actualizarUsuario(Usuario usuario);  
    void eliminarUsuario(int id);  
}
```

```
public class UsuarioDAOImplOracle implements UsuarioDAO {  
    private final DatabaseConnection dbConnection;  
  
    public UsuarioDAOImplOracle(DatabaseConnection dbConnection) {  
        this.dbConnection = dbConnection;  
    }  
  
    @Override  
    public void agregarUsuario(Usuario usuario) {  
        String sql = "INSERT INTO usuarios (nombre, email) VALUES (?, ?)";  
        try (Connection conn = dbConnection.getConnection();  
             PreparedStatement stmt = conn.prepareStatement(sql)) {  
            stmt.setString(1, usuario.getNombre());  
            stmt.setString(2, usuario.getEmail());  
            stmt.executeUpdate();  
        } catch (SQLException e) {  
            throw new RuntimeException("Error al insertar usuario en Oracle", e);  
        }  
    }  
}
```

```
public class UsuarioDAOImplMySQL implements UsuarioDAO {  
    private final DatabaseConnection dbConnection;  
  
    public UsuarioDAOImplMySQL(DatabaseConnection dbConnection) {  
        this.dbConnection = dbConnection;  
    }  
}
```

```
public class UsuarioDAOImplPostgreSQL implements UsuarioDAO {  
    private final DatabaseConnection dbConnection;  
  
    public UsuarioDAOImplPostgreSQL(DatabaseConnection dbConnection) {  
        this.dbConnection = dbConnection;  
    }  
}
```

PATRON DAO: RELACIONES

- Para relacionar unas tablas con otras se puede hacer de forma más manual, poniendo **la clave ajena como una propiedad de tipo numérico**:
- O poniendo una **propiedad que contenga un objeto de la entidad relacionada** (o una colección de éstas). Esto complica el código de las clases DAO pero facilita luego el código de la aplicación. En las siguientes unidades veremos que los ORM lo hacen de esta forma, pero nosotros no tendremos que escribir el código del DAO/Repository sino que lo hará el ORM por nosotros.

```
public class Usuario {  
    private int id;  
    private String nombre;  
    private String email;
```

```
public class Pedido {  
    private int id;  
    private String descripcion;  
    private int idUsuario; // ID del usuario relacionado
```

```
public class Usuario {  
    private int id;  
    private String nombre;  
    private String email;  
    private List<Pedido> pedidos; // Relación Uno a Muchos
```

```
public class Pedido {  
    private int id;  
    private String descripcion;  
    private Usuario usuario; // Relación Muchos a Uno (referencia al usuario)
```

Pool de Conexiones

La conexión a una base de datos conlleva una sobrecarga de rendimiento considerable, debido a procesos como el establecimiento de la comunicación, la autenticación y la verificación de privilegios.

Para optimizar la eficiencia, en lugar de abrir y cerrar una conexión por cada operación, la práctica recomendada es establecer un conjunto (pool) de conexiones que se mantengan activas y reutilizables a lo largo del ciclo de vida de la aplicación.

En aplicaciones reales, esta gestión no se realiza manualmente, sino que se delega a librerías especializadas, como HikariCP, que administran eficazmente el pool de conexiones.

En el alcance de esta unidad, no implementaremos directamente la gestión de conexiones. Más adelante, con Spring Data JPA, esta funcionalidad será manejada automáticamente por la plataforma (la cual, dicho sea de paso, utiliza HikariCP internamente).

Sin embargo, es fundamental comprender el concepto y sus beneficios. Para obtener más detalles y una explicación visual, puede consultar este vídeo:



Ejemplos

🎯 Problema que resuelve:

ANTES (sin DAO):



```
Main.java
└── Lógica de negocio
    └── SQL queries mezcladas
        └── Gestión de JDBC
            └── Presentación de datos
```

✗ Todo mezclado en un solo lugar

DESPUÉS (con DAO):

```
Main.java (Presentación)
    ↓ usa
    IUsuarioDAO (Interfaz)
        ↓ implementa
    UsuarioDAOImpl (Acceso a datos)
        ↓ gestiona
    Base de Datos
```

✓ Cada capa hace una sola cosa



Estructura del patrón DAO:

1 Capa de Modelo (Usuario.java)

- Representa una entidad de la BD
- Solo tiene atributos, getters y setters
- NO tiene lógica de negocio ni SQL

2 Capa de Interfaz (IUsuarioDAO.java)

- Define QUÉ operaciones se pueden hacer
- NO define CÓMO se hacen

3 Capa de Implementación (UsuarioDAOImpl.java)

- Implementa la interfaz
- Contiene **TODO** el código JDBC
- Contiene todas las queries SQL
- Gestiona conexiones y excepciones

4 Capa de Utilidades (ConexionBD.java)

- Centraliza la gestión de conexiones
- Un solo lugar para cambiar la configuración

5 Capa de Presentación (Main.java)

- **SOLO** usa el DAO
- **NO** tiene código SQL ni JDBC
- Se enfoca en la lógica de la aplicación

PASO 1: Código Original

```
ublic class Main_Paso1_Original {
    public static void main(String[] args) {
        System.out.println("≡ PASO 1: CÓDIGO ORIGINAL (TODO EN EL MAIN)");

        String url = "jdbc:sqlite:D:/Acceso a Datos/SQLiteDatabaseBrowser";

        try (Connection conexion = DriverManager.getConnection(url);
            PreparedStatement psUsuarios = conexion.prepareStatement(
                "SELECT * FROM usuarios WHERE localidad LIKE ?");
            PreparedStatement psTelefonos = conexion.prepareStatement(
                "SELECT * FROM telefonos WHERE cod = ?")) {

            System.out.println("✓ Conexión establecida");
            psUsuarios.setString(1, "T%");

            try (ResultSet rsUsuarios = psUsuarios.executeQuery()) {
                while (rsUsuarios.next()) {
                    int cod = rsUsuarios.getInt("cod");
                    String nombre = rsUsuarios.getString("nombre");
                    String apellidos = rsUsuarios.getString("apellidos");
                    String direccion = rsUsuarios.getString("direccion");
                    String localidad = rsUsuarios.getString("localidad");
                    ...
                }
            }
        }
    }
}
```

Objetivo: Entender JDBC básico

- Todo en el main
- Fácil de seguir línea por línea
- **Problemas visibles:** código largo, duplicación potencial



PASO 2: Con Clase Usuario (Modelo)

👉 Objetivo: Separar datos de lógica

- Crear clase Usuario (POJO)
- Método mostrarUsuario() para separar presentación
- **Ventaja:** Código más organizado
- **Problema aún presente:** JDBC en el main

```
class Usuario {  
    private int cod;  
    private String nombre;  
    private String apellidos;  
    private String direccion;  
    private String localidad;  
    private List<String> telefonos;  
  
    public Usuario(int cod, String nombre, String apellidos, String di  
        this.cod = cod;  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.direccion = direccion;  
        this.localidad = localidad;  
        this.telefonos = new ArrayList<>();  
    }  
}
```

```

try (Connection conexion = DriverManager.getConnection(url);
     PreparedStatement psUsuarios = conexion.prepareStatement(
         "SELECT * FROM usuarios WHERE localidad LIKE ?");
     PreparedStatement psTelefonos = conexion.prepareStatement(
         "SELECT * FROM telefonos WHERE cod = ?")) {

    System.out.println("✓ Conexión establecida");
    psUsuarios.setString(1, "T%");

    try (ResultSet rsUsuarios = psUsuarios.executeQuery()) {
        while (rsUsuarios.next()) {
            // MEJORA: Crear objeto Usuario en lugar de variables
            Usuario usuario = new Usuario(
                rsUsuarios.getInt("cod"),
                rsUsuarios.getString("nombre"),
                rsUsuarios.getString("apellidos"),
                rsUsuarios.getString("direccion"),
                rsUsuarios.getString("localidad"))
        }
    }

    // Cargar teléfonos
}

```

```

// MEJORA: Método para mostrar usuario (separa presentación)
private static void mostrarUsuario(Usuario usuario) {
    System.out.println("-----");
    System.out.printf("ID: %d%n", usuario.getCod());
    System.out.printf("Nombre: %s %s%n", usuario.getNombre(), usuario.getApellido());
    System.out.printf("Dirección: %s, %s%n", usuario.getDireccion(),
    System.out.println("Teléfonos:");

    if (usuario.getTelefonos().isEmpty()) {
        System.out.println("    (sin teléfonos)");
    } else {
        for (String telefono : usuario.getTelefonos()) {
            System.out.printf("    - %s%n", telefono);
        }
    }
    System.out.println("-----\n");
}

```

// MEJORAS LOGRADAS: //

- ✓ Código más organizado con la clase Usuario
- ✓ Método *mostrarUsuario()* separa presentación

PERO TODAVÍA:

- ✗ SQL y JDBC siguen en el main
- ✗ No es reutilizable en otras clases

PASO 3: Con Métodos de Acceso a Datos

👉 Objetivo: Encapsular operaciones de BD

- Métodos: obtenerUsuariosPorLocalidad(), mapearUsuario(), cargarTelefonos()
- Main más limpio y legible
- Ventaja: Reutilización dentro de la clase
- Problema aún presente: No reutilizable desde otras clases

Movemos la lógica JDBC a métodos separados

```
public static void main(String[] args) {
    System.out.println("≡ PASO 3: CON MÉTODOS DE ACCESO A DATOS");

    // MEJORA: El main ahora es mucho más simple y legible
    List<Usuario> usuarios = obtenerUsuariosPorLocalidad("T");

    if (usuarios.isEmpty()) {
        System.out.println("No se encontraron usuarios.");
        return;
    }

    for (Usuario usuario : usuarios) {
        mostrarUsuario(usuario);
    }
}
```

```
private static List<Usuario> obtenerUsuariosPorLocalidad(String localidad) {
    List<Usuario> usuarios = new ArrayList<>();

    try (Connection conn = DriverManager.getConnection(URL);
         PreparedStatement psUsuarios = conn.prepareStatement(
             "SELECT * FROM usuarios WHERE localidad LIKE ?");
         PreparedStatement psTelefonos = conn.prepareStatement(
             "SELECT * FROM telefonos WHERE cod = ?")) {

        psUsuarios.setString(1, localidad + "%");

        try (ResultSet rs = psUsuarios.executeQuery()) {
            while (rs.next()) {
                Usuario usuario = mapearUsuario(rs);
                cargarTelefonos(usuario, psTelefonos);
                usuarios.add(usuario);
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
// NUEVO: Método para convertir ResultSet a Usuario
private static Usuario mapearUsuario(ResultSet rs) throws SQLException {
    return new Usuario(
        rs.getInt("cod"),
        rs.getString("nombre"),
        rs.getString("apellidos"),
        rs.getString("direccion"),
        rs.getString("localidad")
    );
}

// NUEVO: Método para cargar teléfonos de un usuario
private static void cargarTelefonos(Usuario usuario, PreparedStatement ps) {
    ps.setInt(1, usuario.getCod());
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) {
            usuario.addTelefono(rs.getString("telefono"));
        }
    }
}
```

MEJORAS LOGRADAS:

- ✓ Main mucho más simple y legible
 - ✓ Métodos específicos para cada tarea
 - ✓ Código SQL separado en métodos
- PERO TODAVÍA:
- ✗ Los métodos están en la misma clase
 - ✗ No podemos reutilizarlos desde otras clases

PASO 4: Con Clase de Acceso a Datos (Pre-DAO)

👉 Objetivo: Separar acceso a datos en su propia clase

- Clase `UsuarioDataAccess` con todos los métodos JDBC
- **Ventaja:** Reutilizable desde cualquier clase
- Main súper simple: solo llama métodos
- **Casi es un DAO, pero...** falta la interfaz

```
// NUEVA CLASE: Toda la lógica de acceso a datos en un solo lugar
class UsuarioDataAccess {
    private static final String URL = "jdbc:sqlite:D:/Acceso a Datos/Sistema.db";

    // Método público que usa el main
    public List<Usuario> obtenerPorLocalidad(String localidad) {
        List<Usuario> usuarios = new ArrayList<>();

        try (Connection conn = DriverManager.getConnection(URL);
             PreparedStatement psUsuarios = conn.prepareStatement(
                     "SELECT * FROM usuarios WHERE localidad LIKE ?"));
            PreparedStatement psTelefonos = conn.prepareStatement(
                     "SELECT * FROM telefonos WHERE cod = ?")) {

            psUsuarios.setString(1, localidad + "%");

            try (ResultSet rs = psUsuarios.executeQuery()) {
                while (rs.next()) {
                    Usuario usuario = mapearUsuario(rs);
                    cargarTelefonos(usuario, psTelefonos);
                    usuarios.add(usuario);
                }
            }
        }
    }
}
```

```
// Podemos agregar más métodos fácilmente
public Usuario obtenerPorCod(int cod) {
    try (Connection conn = DriverManager.getConnection(URL);
        PreparedStatement ps = conn.prepareStatement(
            "SELECT * FROM usuarios WHERE cod = ?")) {

        ps.setInt(1, cod);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                return mapearUsuario(rs);
            }
        }
    } catch (SQLException e) {
        System.err.println("X ERROR: " + e.getMessage());
    }
    return null;
}
```

```

public class Main_Paso4_ConClaseAccesoDatos {

    public static void main(String[] args) {
        System.out.println("≡ PASO 4: CON CLASE DE ACCESO A DATOS ≡\n");

        // MEJORA: Crear instancia de la clase de acceso a datos
        UsuarioDataAccess dataAccess = new UsuarioDataAccess();

        // El main es ahora súper simple
        List<Usuario> usuarios = dataAccess.obtenerPorLocalidad("T");

        for (Usuario usuario : usuarios) {
            mostrarUsuario(usuario);
        }

        System.out.println("\n--- Ahora podemos hacer más cosas fácilmente ---\n");

        // Obtener un usuario específico
        Usuario usuario1 = dataAccess.obtenerPorCod(1);
        if (usuario1 != null) {
            System.out.println("Usuario específico: " + usuario1.getNombre());
        }

        // Obtener todos los usuarios
        List<Usuario> todos = dataAccess.obtenerTodos();
        System.out.println("Total de usuarios: " + todos.size());
    }
}

```

MEJORAS LOGRADAS:

- ✓ Clase separada para acceso a datos
- ✓ Código reutilizable desde cualquier clase
- ✓ Fácil agregar nuevos métodos
- ✓ Main súper simple

SIGUIENTE PASO: → Agregar una INTERFAZ (DAO completo)

PASO 5: Patrón DAO Completo ✨

👉 Objetivo: Implementar DAO profesional

- Interfaz `IUsuarioDAO` define el **QUÉ**
- Clase `UsuarioDAOSQLite` define el **CÓMO**
- **Ventajas:**
 - Cambio fácil de BD: `new UsuarioDAOSQLite() → new UsuarioDAOMySQL()`
 - Testable: puedes crear mocks de la interfaz
 - Código profesional y escalable

```
// NUEVO: Interfaz que define el contrato (QUÉ se puede hacer)
interface IUsuarioDAO {
    List<Usuario> obtenerPorLocalidad(String localidad);
    Usuario obtenerPorCod(int cod);
    List<Usuario> obtenerTodos();
    boolean insertar(Usuario usuario);
    boolean actualizar(Usuario usuario);
    boolean eliminar(int cod);
}

// Implementación para SQLite (CÓMO se hace)
class UsuarioDAOSQLite implements IUsuarioDAO {
    private static final String URL = "jdbc:sqlite:D:/Acceso a Datos/SQLiteDatabaseBrowserPortable/prueba.db";

    @Override
    public List<Usuario> obtenerPorLocalidad(String localidad) {
        List<Usuario> usuarios = new ArrayList<>();

        try (Connection conn = DriverManager.getConnection(URL);
            PreparedStatement ps = conn.prepareStatement(
                "SELECT * FROM usuarios WHERE localidad LIKE ?")) {

            ps.setString(1, localidad + "%");
            ...
        }
    }
}
```

```
@Override
public Usuario obtenerPorCod(int cod) {
    try (Connection conn = DriverManager.getConnection(URL);
        PreparedStatement ps = conn.prepareStatement(
            "SELECT * FROM usuarios WHERE cod = ?")) {

        ps.setInt(1, cod);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                return mapearUsuario(rs);
            }
        }
    } catch (SQLException e) {
        System.err.println("X ERROR: " + e.getMessage());
    }
    return null;
}

@Override
public List<Usuario> obtenerTodos() {
    List<Usuario> usuarios = new ArrayList<>();

    try (Connection conn = DriverManager.getConnection(URL);
        PreparedStatement ps = conn.prepareStatement("SELECT * FROM usuarios")) {
```

```

@Override
public boolean insertar(Usuario usuario) {
    try (Connection conn = DriverManager.getConnection(URL);
        PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO usuarios (nombre, apellidos, direccion, localidad) VALUES (?, ?, ?, ?)")) {

        ps.setString(1, usuario.getNombre());
        ps.setString(2, usuario.getApellidos());
        ps.setString(3, usuario.getDireccion());
        ps.setString(4, usuario.getLocalidad());

        return ps.executeUpdate() > 0;
    } catch (SQLException e) {
        System.err.println("✗ ERROR: " + e.getMessage());
        return false;
    }
}

```

// Si más adelante necesitas MySQL, solo creas otra implementación:
// class UsuarioDAOMySQL implements IUsuarioDAO { ... }

```

public class Main_Paso5_PatronDAOCompleto {

    public static void main(String[] args) {
        System.out.println("≡≡ PASO 5: PATRÓN DAO COMPLETO ≡\n");

        // VENTAJA: Programamos contra la interfaz, no la implementación
        // Podemos cambiar fácilmente de SQLite a MySQL
        IUsuarioDAO usuarioDAO = new UsuarioDAOSQLite();
        // IUsuarioDAO usuarioDAO = new UsuarioDAOMySQL(); // Cambio fácil!

        // El código del main es idéntico sin importar la BD
        List<Usuario> usuarios = usuarioDAO.obtenerPorLocalidad("T");

        System.out.println("Usuarios encontrados: " + usuarios.size() + "\n");

        for (Usuario usuario : usuarios) {
            mostrarUsuario(usuario);
        }
    }
}

```