# Práctico 9: Acceso a Base de Datos con Jdbc

## Objetivo:

**Desarrollar habilidades en el uso de JDBC para que el estudiante sea capaz de desarrollar aplicaciones Java que interactúen con bases de datos relacionales, aplicando los principios de la arquitectura JDBC, configurando entornos adecuados, implementando operaciones CRUD seguras y eficientes, y estructurando el código siguiendo buenas prácticas y diseño modular.**

***Resultados de aprendizaje:***

1. **Comprender los fundamentos y la arquitectura de JDBC:** El estudiante será capaz de describir el propósito de JDBC en aplicaciones Java, explicar su arquitectura incluyendo los componentes clave (JDBC API, DriverManager, Driver JDBC) y distinguir entre los diferentes tipos de drivers, destacando las ventajas del Type 4 en términos de portabilidad y rendimiento.
2. **Preparar y configurar correctamente el entorno de desarrollo para JDBC:** El estudiante será capaz de instalar y configurar las herramientas necesarias (JDK LTS, NetBeans 25, MySQL Server, DBeaver), integrar el driver JDBC y verificar la conexión funcional con una base de datos relacional desde una aplicación Java.
3. **Establecer conexiones robustas y seguras entre Java y bases de datos:** El estudiante será capaz de implementar conexiones a bases de datos utilizando JDBC clásico y pool de conexiones, aplicando el cierre automática de conexiones y la preparación de la consulta para garantizar el uso eficiente de recursos y la seguridad.
4. **Desarrollar operaciones CRUD utilizando JDBC con y sin control transaccional:** El estudiante será capaz de ejecutar operaciones de creación, lectura, actualización y eliminación de datos sobre una base de datos desde Java, tanto con JDBC directo como mediante un pool de conexiones, aplicando manejo de transacciones para asegurar la consistencia de los datos.
5. **Construir una aplicación CRUD completa siguiendo buenas prácticas de diseño modular:** El estudiante será capaz de desarrollar un sistema CRUD aplicando buenas prácticas como el uso del patrón DAO, separación por capas (model, dao, service, config), depender de interfaces (abstracciones) y no de clases concretas y el uso de HikariCP para la gestión eficiente de conexiones.

## ¿Qué es una Kata y cómo se utiliza en programación?

**Una kata es un ejercicio de programación diseñado para mejorar habilidades de codificación mediante la repetición y el aprendizaje progresivo.** El término proviene de las artes marciales, donde las katas son secuencias de movimientos que se practican repetidamente para perfeccionar la técnica.


En programación, las katas ayudan a los programadores a reforzar conceptos, mejorar la comprensión del código y desarrollar buenas prácticas. **Se recomienda resolver una kata varias veces, intentando mejorar el código en cada iteración, utilizando mejores estructuras, nombres más claros y principios de diseño.**

## Importante:

**Intentar resolver cada kata sin mirar la solución.**

**Comprobar la solución y corregir errores si es necesario.**

**Repetir 2 o 3 veces para mejorar la comprensión, lógica y el código.**

**Experimentar con diferentes valores para reforzar el aprendizaje.**

### Resolver Katas

**A continuación, te presento 4 enunciados de katas en Java para fortalecer los conceptos vistos en el módulo 9.**

**Kata 1: CRUD de Categorías con GestorCategorias**

**Kata 2:.CRUD de Categorías con buenas prácticas**

**Kata 3: Gestión de Productos vinculados a Categorías**

**Kata 4: Gestión de Pedidos con múltiples Productos y Categorías**

---

## Kata 1: CRUD de Categorías con GestorCategorias (estudiar actividades: 1-2)

**Enunciado:**
**Implementar una aplicación java para CRUD que permita gestionar categorías de productos (sin relación con productos aún).**

**Clases/Interfaces:**

`GestorCategorias`

**Tabla en Base de Datos**

`Categoría`

**Atributos (para `Categoria`):**

- `id: int` **(autoincremental)**
- `nombre: String`
- `descripcion: String`

**Métodos (para `GestorCategorias`):**

- `agregarCategoria(nombre, descripcion)`
- `mostrarCategoria(id)`
- `listarCategorias()`
- `actualizarCategoria(id, nombre, descripcion)`
- `eliminarCategoria(id)`

🎯 **Tarea a realizar:**

1. **Crear tabla `categorias` en base de datos MySql.**
2. **Implementar los métodos CRUD en `GestorCategorias`**
3. **Validar que no existan categorías duplicadas por nombre.**

---

## 🥋🥋 Kata 2: CRUD de Categorías con buenas prácticas

*(estudiar actividades: 1 - 2 - 3)*

---

### 🗒️ Enunciado:

Implementar una aplicación Java con JDBC para realizar operaciones **CRUD** (crear, leer, actualizar, eliminar) sobre categorías de productos, aplicando el patrón por **capas DAO-Service-Model**.

Se trabajará exclusivamente con la entidad **Categoría** (sin relación aún con productos), y se aplicarán validaciones, organización del código en paquetes y reutilización mediante una interfaz genérica.

## 🧱 Clases / Interfaces:

### 📦 `model`

- `Categoria.java`
  Clase que representa la entidad de dominio.

### 📦 `dao`

- `GenericDAO<T>`
  Interfaz genérica con métodos: `crear`, `leer`, `actualizar`, `eliminar`, `listar`.
- `CategoriaDAOImpl.java`
  Implementa `GenericDAO<Categoria>` con lógica JDBC y SQL.

### 📦 `service`

- `GenericService<T>`
  Interfaz de lógica de negocio genérica.
- `CategoriaServiceImpl.java`
  Implementa `GenericService<Categoria>`. Aplica validaciones antes de llamar al DAO.

---

## 🗃️ Tabla en la Base de Datos

### `categorias`

| Campo | Tipo | Restricciones |
|---|---|---|
| id | int | PRIMARY KEY, AUTO_INCREMENT |
| nombre | varchar(100) | NOT NULL, UNIQUE |
| descripcion | varchar(255) | |

---

## 🧬 Atributos (de `Categoria`):

private int id;
private String nombre;
private String descripcion;

---

## 🛠️ Métodos Clave a Implementar:

**En `CategoriaDAOImpl`**

- `crear(Categoria c)`
- `leer(int id)`
- `actualizar(Categoria c)`
- `eliminar(int id)`
- `listar()`
- ✅ `boolean existeNombre(String nombre)`

**En `CategoriaServiceImpl`**

- `crear(Categoria c)`:
  - Validar que `nombre` no esté vacío.
  - Verificar duplicado usando `existeNombre(nombre)` antes de insertar.

---

## 🎯 Tarea a realizar:

1. 📦 **Mantener la tabla `categorias` creada en la Kata 1.**
2. 🧱 **Diseñar las clases según el patrón de capas (model, dao, service).**
3. ⚙️ **Implementar la lógica JDBC en `CategoriaDAOImpl`.**
4. 🧠 **Agregar validaciones en `CategoriaServiceImpl`:**
   - Validar campos vacíos.
   - Verificar que no exista una categoría con el mismo nombre.
5. 🧪 **Probar operaciones CRUD desde un `Main` de prueba.**

---

# 🥋🥋 Kata 3: Gestión de Productos vinculados a Categorías

🚀 *con validación de stock en capa de servicio*

---

## 🧾 Enunciado:

Extender el sistema para gestionar productos vinculados a categorías (1:N). Cada producto pertenece a una categoría válida, y debe tener stock (`cantidad > 0`) para ser insertado o actualizado. Aplicar arquitectura en capas (`model`, `dao`, `service`) y validaciones de integridad referencial y de negocio.

---

## 🧱 Clases / Interfaces

### 📦 `model`

- `Categoria.java` (sin cambios)
- `Producto.java`

### 📦 `dao`

- `GenericDAO<T>`
- `CategoriaDAOImpl`
- `ProductoDAOImpl`

### 📦 `service`

- `GenericService<T>`
- `CategoriaServiceImpl`
- `ProductoServiceImpl`

---

## 🗃️ Tablas en la Base de Datos

### 📋 `productos`

| Campo | Tipo | Restricciones |
|---|---|---|
| id | INT | PRIMARY KEY AUTO_INCREMENT |
| nombre | VARCHAR(100) | NOT NULL |
| descripcion | VARCHAR(255) | |
| precio | DECIMAL | NOT NULL |
| cantidad | INT | NOT NULL |
| id_categoria | INT | FOREIGN KEY → `categorias(id)` |

ALTER TABLE productos ADD CONSTRAINT fk_categoria FOREIGN KEY (id_categoria) REFERENCES categorias(id);

---

## 🧬 Atributos

**`Producto.java`**

private int id;
private String nombre;
private String descripcion;
private double precio;
private int cantidad;
private Categoria categoria;

---

## 🛠️ Métodos clave

**En `ProductoDAOImpl`**

- `crear(Producto p)`
- `leer(int id)`
- `actualizar(Producto p)`
- `eliminar(int id)`
- `listar()`
- `listarPorCategoria(int idCategoria)`
- `boolean existeCategoria(int idCategoria)`

### 🎯 Tarea a realizar

1. 🧱 Crear o modificar la tabla `productos`, asegurando que contenga `cantidad`.
2. 📦 Agregar el atributo `cantidad` a `Producto.java`.
3. ⚙️ Implementar validaciones en `ProductoServiceImpl`:
    - Nombre no vacío.
    - Precio > 0.
    - **Cantidad > 0.**
    - Categoría existente.
4. 🔍 Implementar método `listarPorCategoria(idCategoria)` y `mostrarProductoConCategoria(id)` con `JOIN`.

---

## 🥷🥷🥷🥷 Kata 4: Gestión de Pedidos con múltiples Productos y Categorías

---

### 🗒️ Enunciado:

Implementar el modelo de **Pedido**, en el que un cliente realiza una compra compuesta por múltiples productos. Cada fila del pedido representa un **ítem** (producto + cantidad), y cada producto está vinculado a una **categoría**.

El objetivo es insertar un pedido con múltiples ítems, calcular el total del pedido y persistir todo en la base de datos utilizando **JDBC**, **DAOs**, y **services** con validaciones de negocio e integridad referencial.

---

### 📐 Relación entre entidades:

Pedido 1 ---- N ItemPedido N ---- 1 Producto N ---- 1 Categoria

---

### 🧱 Clases / Interfaces

#### 📦 `model`

- `Categoria`
- `Producto`
- `ItemPedido`
- `Pedido`

### 📦 dao

- `GenericDAO<T>`
- `ProductoDAOImpl`
- `CategoriaDAOImpl`
- `ItemPedidoDAOImpl`
- `PedidoDAOImpl`

### 📦 service

- `ProductoServiceImpl`
- `PedidoServiceImpl`

---

## 🗃️ Tablas en la Base de Datos

### 📋 pedidos

| Campo | Tipo | Restricciones |
|---|---|---|
| id | INT | PRIMARY KEY AUTO_INCREMENT |
| fecha | DATE | NOT NULL |
| total | DECIMAL | NOT NULL |

### 📋 items_pedido

| Campo | Tipo | Restricciones |
|---|---|---|
| id | INT | PRIMARY KEY AUTO_INCREMENT |

| | | |
|---|---|---|
| pedido_id | INT | FOREIGN KEY → `pedidos(id)` |
| producto_id | INT | FOREIGN KEY → `productos(id)` |
| cantidad | INT | NOT NULL |
| subtotal | DECIMAL | NOT NULL (cantidad * producto.precio) |

---

## 🧬 Atributos

### `ItemPedido.java`

private int id;
private Producto producto;
private int cantidad;
private double subtotal;

### `Pedido.java`

private int id;
private LocalDate fecha;
private List<ItemPedido> items;
private double total;

---

## 🛠️ Métodos clave

### En `PedidoServiceImpl`

- `crearPedido(Pedido pedido)`
    - Validar que haya al menos un ítem.
    - Validar que la cantidad de cada ítem sea mayor a cero.
    - Validar existencia del producto y su stock disponible.
    - Descontar stock de productos.
    - Calcular subtotales e insertar `items_pedido`.
    - Calcular total del pedido y persistirlo.
- `mostrarDetallePedido(int pedidoId)`
    - Mostrar cada ítem con nombre de producto, categoría, cantidad, subtotal.

- Mostrar total general.

---

## ✅ Validaciones de negocio

- ✔️ Cada ítem debe tener cantidad > 0
- ✔️ El producto debe existir
- ✔️ El stock del producto debe ser suficiente
- ✔️ La categoría del producto debe existir
- ✔️ El total del pedido se calcula automáticamente

---

## 🎯 Tarea a realizar

1. Crear tablas `pedidos` e `items_pedido` con claves foráneas.
2. Implementar clases modelo y DAOs para `Pedido` e `ItemPedido`.
3. Implementar en `PedidoServiceImpl` la lógica de:
    - Validación de productos y stock.
    - Inserción en `pedidos` e `items_pedido` (usar transacciones).
    - Cálculo de total del pedido.
4. Mostrar un pedido completo:
    - Producto, categoría, cantidad, subtotal.
    - Total del pedido.

---

## 🧪 En main

- Simular una compra desde `Main.java` cargando 2 productos en el pedido.
- Hacer rollback si algún ítem tiene stock insuficiente.

- Mostrar todos los pedidos con sus detalles usando `JOIN`.