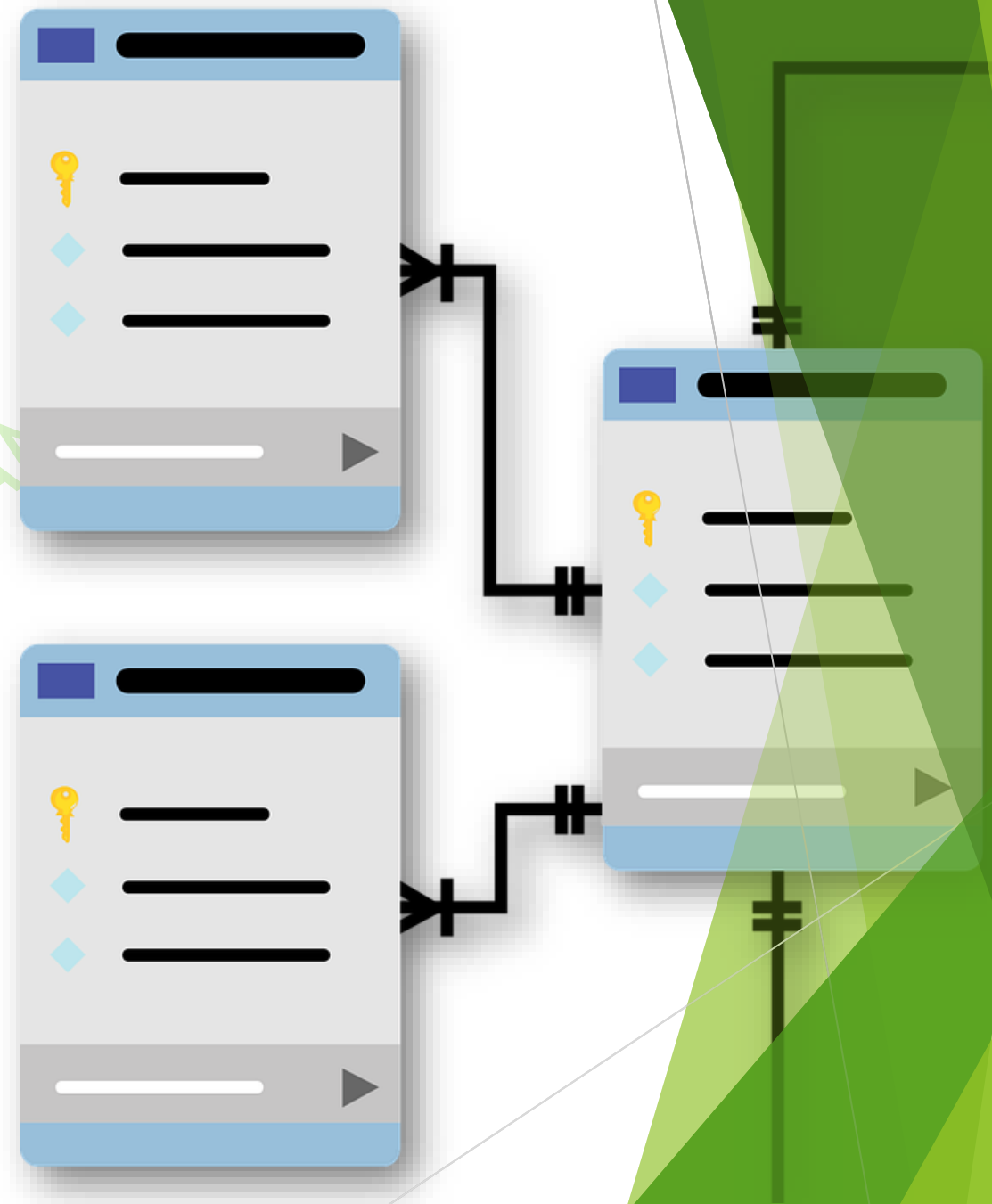


# UD 6 - MANEJO DE BASES DE DATOS RELACIONALES



# UD 6 - MANEJO DE BBDD RELACIONALES

## Índice

1. Introducción
2. Java Data Base Connectivity
3. Conexión a una BD MariaDB
4. Las interfaces Statement y ResultSet
5. PreparedStatement y CallableStatement
6. Los tipos de datos SQL vs Java
7. Transacciones
8. La persistencia en una aplicación por capas

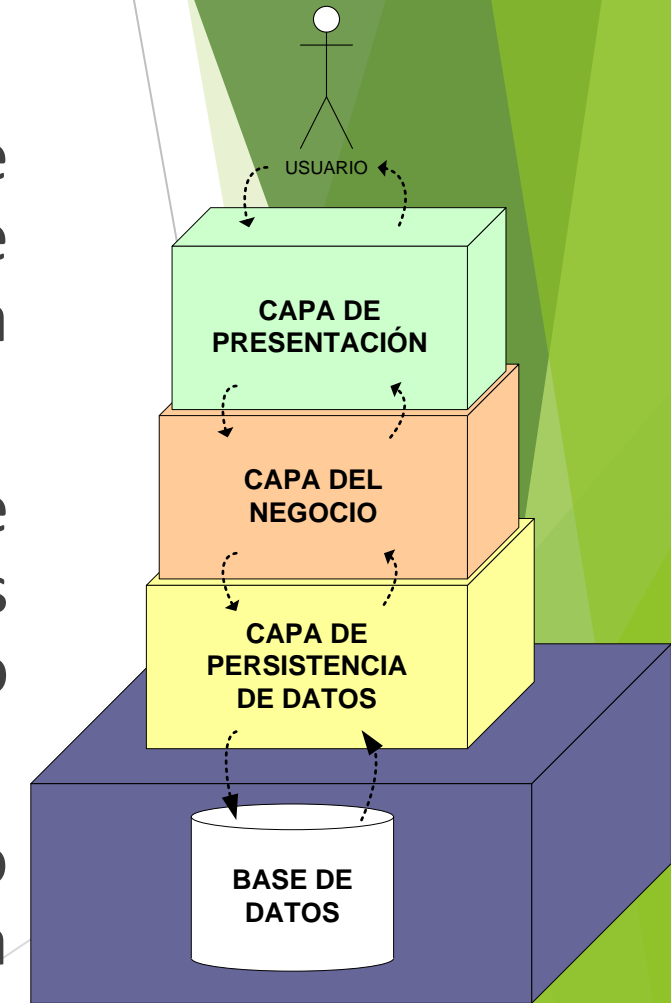
ANEXO I – El patrón DAO

# UD 6 - MANEJO DE BBDD RELACIONALES

## 1 - Introducción

### 1 – Introducción

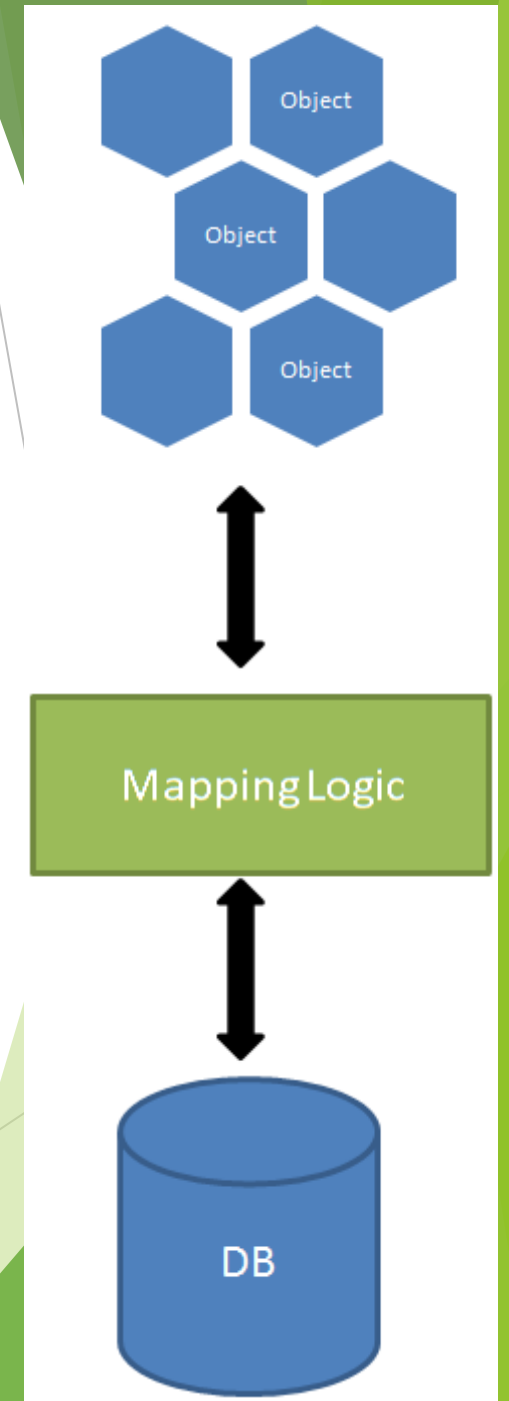
- ▶ Recordemos que el software de una aplicación se estructura de modo que cada capa tiene su responsabilidad y es más fácil de aislar las distintas piezas de código. Esto mejora la “mantenibilidad” del código y la “adaptación al cambio”.
- ▶ En este tema vamos a trabajar en la capa de persistencia que permite almacenar información de forma “persistente”, es decir, que si se apaga el ordenador no se pierde (al contrario que la memoria RAM, que es “volátil”).
- ▶ La información se puede almacenar también en ficheros, pero las bases de datos permiten un acceso a más rápido y versátil a los datos y, además, incorporan medidas de seguridad y recuperación ante fallos.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 1 - Introducción

- ▶ Vamos a estudiar cómo Java permite acceder a cualquier base de datos, sea del proveedor que sea y también las operaciones más usuales.
- ▶ También vamos a analizar el impacto de manejar dos modelos conceptuales distintos: el modelo relacional de una BD y el modelo orientado a objetos del lenguaje Java.
- ▶ Para salvar las diferencias entre ambos modelos se debe generar mucho código repetitivo y poco original ("*boilerplate*" = soso, del montón) o bien usar herramientas de tipo ORM (Object-Relational Mapping) que permiten hacerlo todo de una forma más cómoda, pero que si no se usan correctamente pueden sobrecargar el sistema.

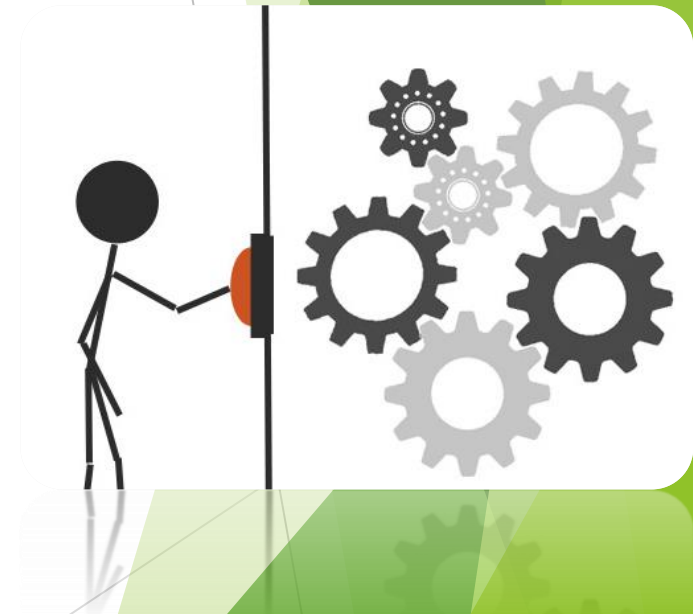


# UD 6 - MANEJO DE BBDD RELACIONALES

## 2 - Java Data Base Connectivity

### 2 – Java Data Base Connectivity

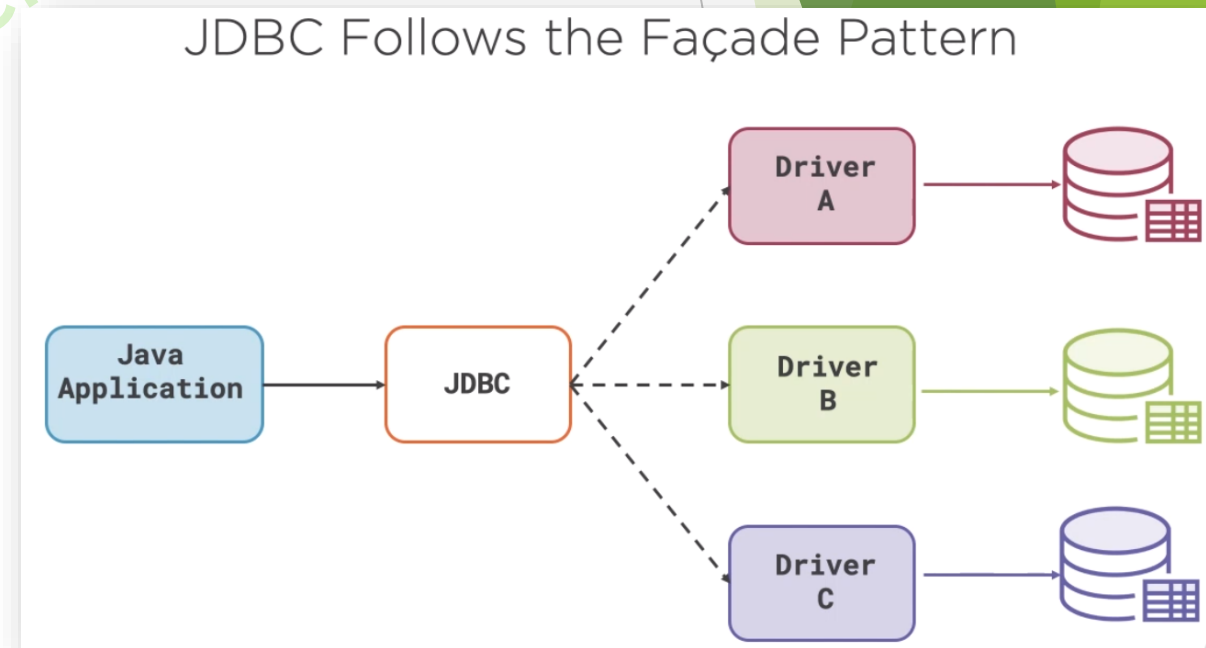
- ▶ Los creadores de Java debían dotar al lenguaje de un mecanismo homogéneo para conectar a cualquier base de datos así que usaron el “**poder de las interfaces**” para conseguirlo.
- ▶ La idea consistía en crear interfaces que describieran las cosas que se pueden hacer con una base de datos y dejar a cada proveedor de BBDD que implemente estas interfaces para su producto concreto.
- ▶ De esta idea nace el estándar **JDBC**. El paquete **java.sql** define las interfaces que se deben implementar así como los tipos de datos que se van a usar para manejar cualquier base de datos.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 2 - Java Data Base Connectivity

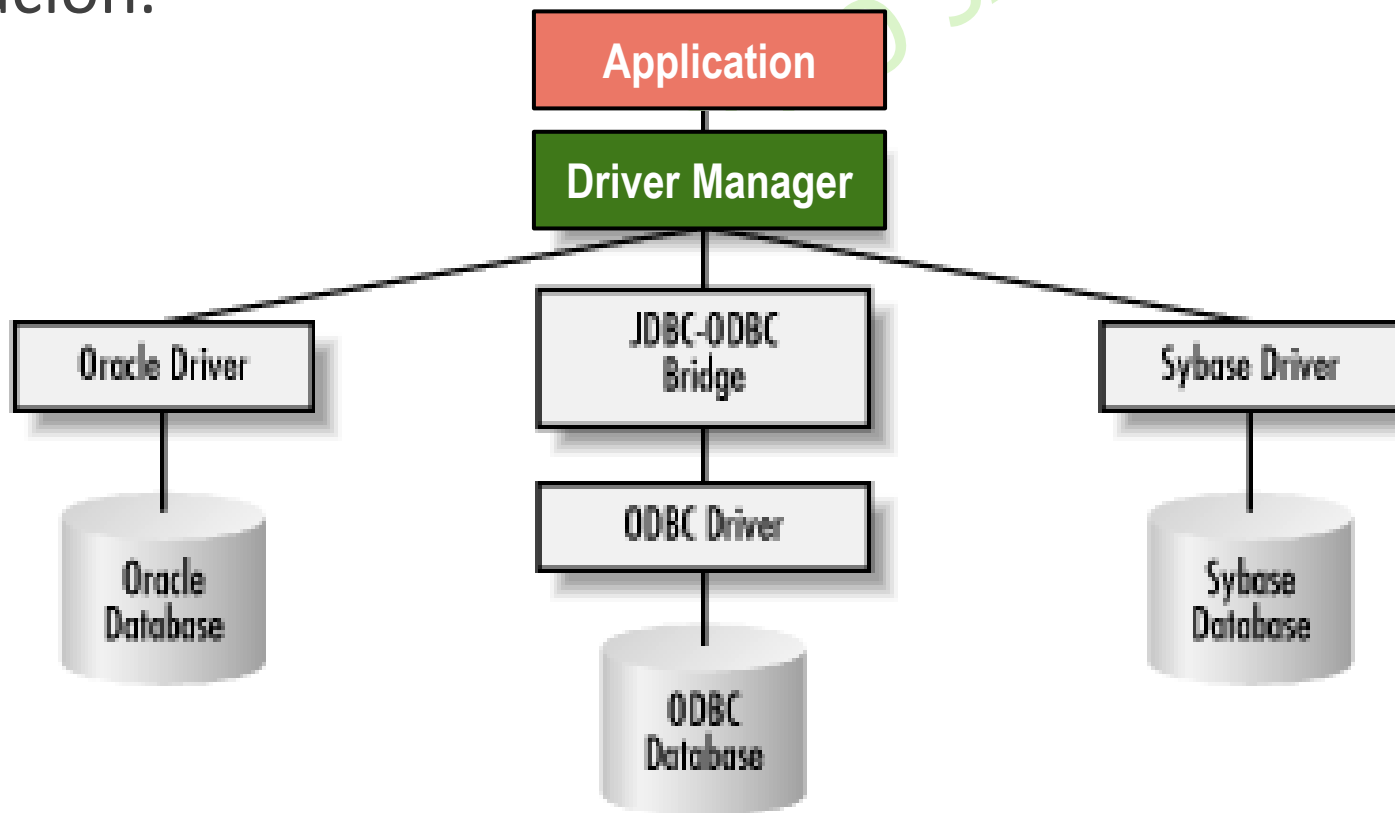
- ▶ Lo que se hace es aplicar un patrón de diseño llamado “**fachada**” de modo que se obliga a que todos los proveedores de BBDD “presenten” la misma fachada/interfaz para manejar su producto. De este modo, a ojos de la aplicación, todas las BBDD se manejan de la misma forma.
- ▶ Cada proveedor debe crear una clase que implemente la interfaz **Driver** en la que se realiza todo el trabajo, cumpliendo así con la “fachada de JDBC”.
- ▶ De este modo existe la clase “Driver de JDBC para Oracle” o el “Driver de JDBC para MariaDB”...



# UD 6 - MANEJO DE BBDD RELACIONALES

## 2 - Java Data Base Connectivity

- Dado que una aplicación puede necesitar **conectarse a varias bases de datos**, el lenguaje Java proporciona una clase llamada **DriverManager** que permite cargar y gestionar distintos drivers en un aplicación.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB X

### 3 – Conexión a una BBDD MariaDB

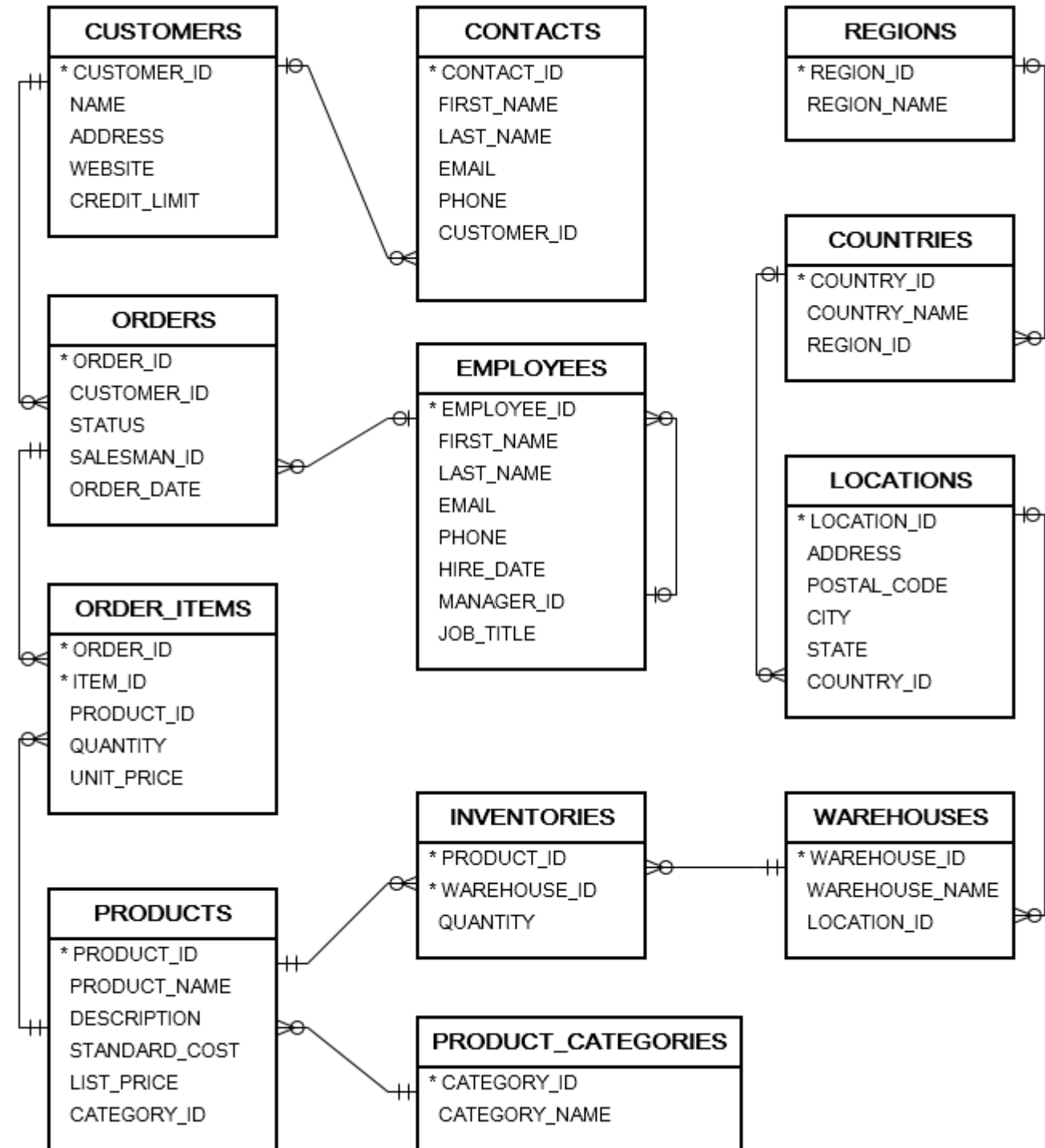
#### ENTORNO DE TRABAJO: COMPANY DB

- Para trabajar esta unidad didáctica vamos crear el siguiente esquema de una compañía ficticia extraído de la siguiente página:

<https://www.oracletutorial.com/getting-started/oracle-sample-database/>

- Para ello entramos en HeidiSQL como usuario root y lanzaremos el siguiente script:

*company\_create\_user.sql*

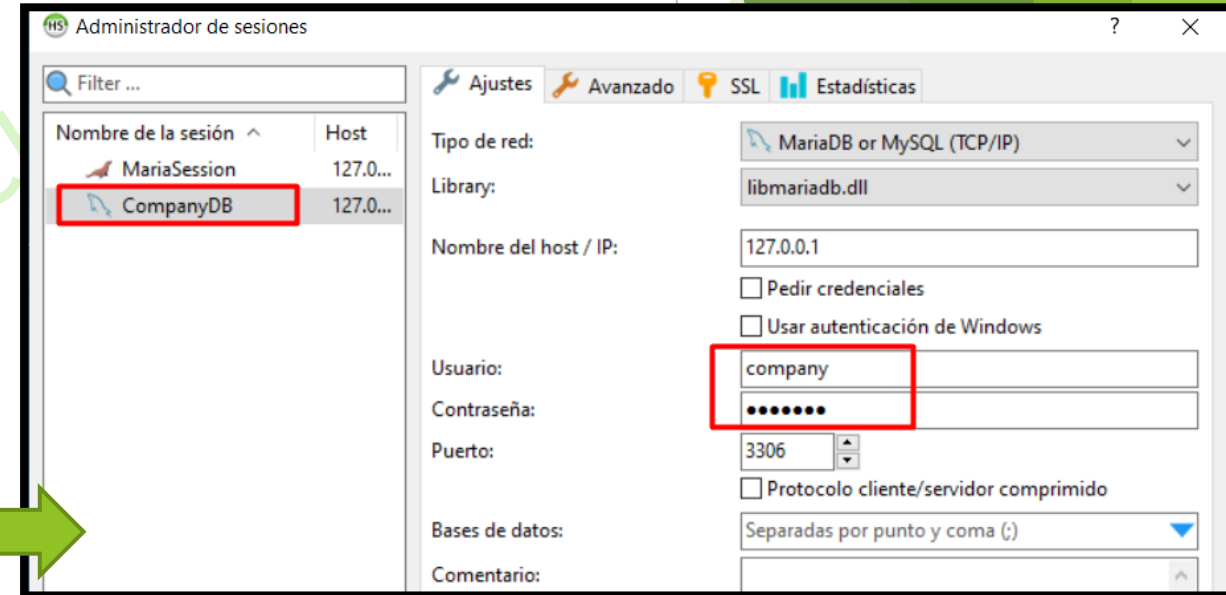




# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

- ▶ El script anterior crea una base de datos llamada 'company\_db' y un usuario con privilegios sobre esa BD identificado por: company/company
- ▶ Ahora tenemos que crear una nueva conexión para entrar con este usuario:



- ▶ Por último, hay que lanzar los siguientes scripts para crear el esquema de tablas y poblarlas con datos:

*company\_schema.sql*  
*company\_data.sql*



Si accidentalmente hemos borrado algo que no debíamos y queremos volver a tener el esquema como nuevo hay que ejecutar el script *company\_drop.sql* y después los 3 anteriores

# UD 6 - MANEJO DE BBDD RELACIONALES

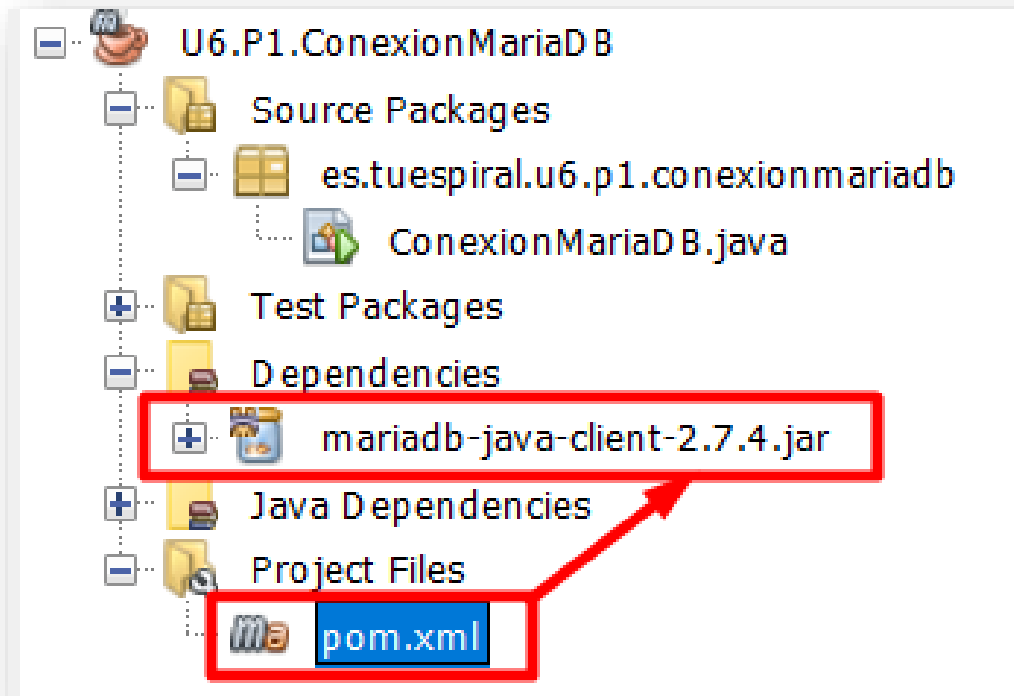

## 3 - Conexión a una BD MariaDB

### CONEXIÓN CON LA BD DESDE EL CÓDIGO

Veamos cómo se lleva todo esto al código. Seguimos los pasos:

1. Abrimos un nuevo proyecto de Java (proyecto tipo Maven)
2. Añadimos al fichero pom.xml la siguiente dependencia:

```
<dependencies>  
  <dependency>  
    <groupId>org.mariadb.jdbc</groupId>  
    <artifactId>mariadb-java-client</artifactId>  
    <version>2.7.4</version>  
  </dependency>  
</dependencies>
```



# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD

### 3. Escribimos el siguiente código

Descarga el fichero  
U6.P1.MDB.Conexion y prueba si  
te funciona correctamente

```
--- exec-maven-plugin:1.5.0:ex  
Conexión válida: true  
Estado del autocommit: true  
-----  
BUILD SUCCESS
```

```
public class ConexionMariaDB {  
  
    public static void main(String args[]) {  
        final String USER = "company";  
        final String PASS = "company";  
        final String DB_NAME = "company_db";  
        final String CONN_URL = "jdbc:mariadb://localhost:3306/" + DB_NAME;  
        Connection conn = null;  
  
        try {  
            conn = DriverManager.getConnection(CONN_URL, USER, PASS);  
            System.out.println("Conexión válida: " + conn.isValid(10));  
            System.out.println("Estado del autocommit: " + conn.getAutoCommit());  
        } catch (SQLException e) {  
            System.out.println("Ocurrió una excepción al conectar a la BD");  
        } finally {  
            try {  
                if (conn != null) {  
                    conn.close();  
                }  
            } catch (SQLException ex) {  
                System.out.println("Ocurrió una excepción al cerrar la BD");  
            }  
        }  
    }  
}
```

# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

- ▶ La cadena de conexión a la BD de MaríaDB es:

Diagrama de la cadena de conexión JDBC para MariaDB:

`jdbc:mariadb://localhost:3306/company_db`

Etiquetas de los componentes:

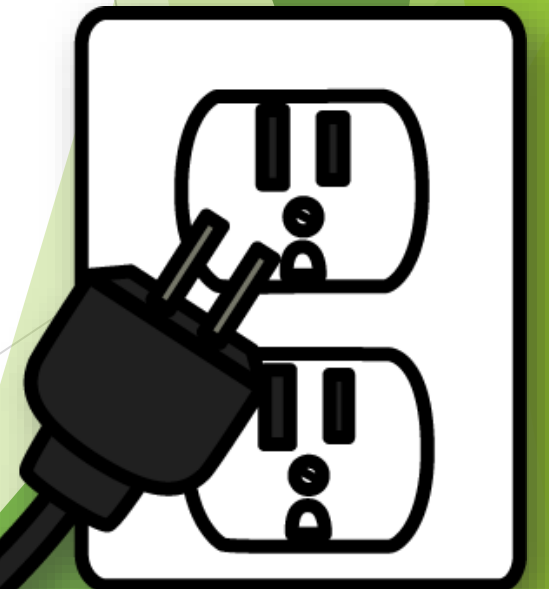
- `jdbc`: Fijo
- `mariadb`: Nombre del driver
- `//localhost:3306`: máquina:puerto
- `company_db`: Nombre de la BD

- ▶ `Connection conn = DriverManager.getConnection(url, user, pass);`

le pide a la clase `DriverManager` que busque el driver indicado por la *url*, lo cargue en memoria, lo registre y que intente abrir una conexión usando *user* y *pass*. Si lo consigue, devuelve un objeto que implementa la interfaz `Connection`.

- ▶ `conn.isValid(10)` testea la conexión, haciendo una prueba que puede tardar hasta 10 segundos, y devuelve *true* si todo fue bien y *false* en caso contrario.

Cada proveedor de BBDD tiene su propia cadena de conexión en JDBC



# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

- ▶ `conn.getAutoCommit()` permite conocer la configuración del parámetro autocommit de la BD.

```
▶ } finally {  
    try {  
        if (conn != null)  
            conn.close();  
    } catch(SQLException e) {  
        System.out.println("Ocurrió una excepción al cerrar la BD");  
    }  
}
```

`conn.close()` cierra la conexión. Fíjate que lo metemos en un bloque *finally* para garantizar que se cierra en cualquier caso. Dado que la operación de cierre también puede lanzar una excepción tenemos que encerrarla en un *try...catch* o bien “lanzarla hacia arriba”.





# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

### Gestión de la conexión

- ▶ Crear un objeto *Connection* es **costoso** en tiempo de cómputo, así que las aplicaciones suelen **reutilizar las conexiones**. Para ello se crea un “**pool de conexiones**” que es como un almacén que te “presta” conexiones.
- ▶ Cerrar la conexión permite que pueda ser “reutilizada” correctamente por otros objetos o hilos de programación. Además, en algunas ocasiones una conexión sin cerrar puede “bloquear” el acceso a la base de datos y hacer que toda la aplicación se caiga.
- ▶ *Es el equivalente a limpiar tu mesa en el McDonald's para que pueda usarla el siguiente cliente o dejarla hecha una porquería.*

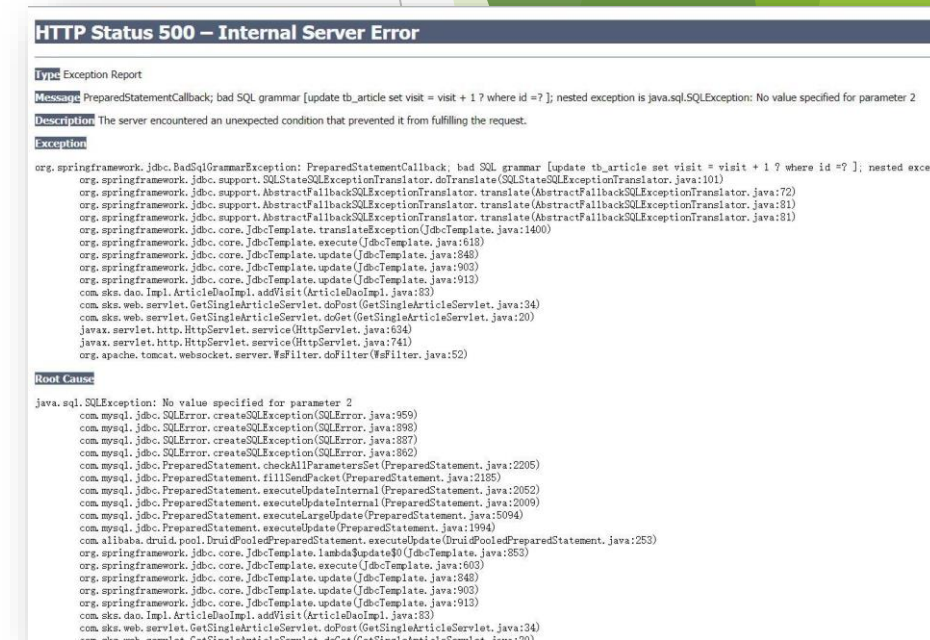


# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

### Tratamiento de las excepciones:

- ▶ JDBC solo lanza excepciones **SQLException** que son de tipo **“checked exception”** así que, como siempre, tendremos que “capturarlas” o “lanzarlas hacia arriba”.
- ▶ Si lanzamos esta excepción hacia arriba y llega al *main* se imprimiría la pila de llamadas. Esto se considera una **mala práctica** porque podría dar información a un atacante sobre la estructura de nuestras tablas.
- ▶ Así que normalmente esta excepción se captura y se **lanza otra excepción** con nombre más concreto y que oculta los detalles de la SQLException. Ejemplo: `ClienteNotFoundException`, `ClienteCreateException`...



The screenshot shows an "HTTP Status 500 - Internal Server Error" page. It contains a "Type: Exception Report" section. The "Message" is "PreparedStatementCallback; bad SQL grammar [update tb\_article set visit = visit + 1 ? where id =? ]; nested exception is java.sql.SQLException: No value specified for parameter 2". The "Description" states: "The server encountered an unexpected condition that prevented it from fulfilling the request." The "Exception" section lists the stack trace, starting with "org.springframework.jdbc.BadSqlGrammarException: PreparedStatementCallback; bad SQL grammar [update tb\_article set visit = visit + 1 ? where id =? ]; nested exception is java.sql.SQLException: No value specified for parameter 2". The "Root Cause" section shows the final exception: "java.sql.SQLException: No value specified for parameter 2".

# UD 6 - MANEJO DE BBDD RELACIONALES

## 3 - Conexión a una BD MariaDB

- ▶ La interfaz **Connection** define muchos métodos. Los que más nos interesan en este punto del tema son:
  - ▶ **commit()** -> persiste los cambios no guardados desde el último commit/rollback.
  - ▶ **rollback()** -> deshace los cambios no guardados desde el último commit/rollback.
  - ▶ **setAutoCommit(boolean autoCommit)** -> establece el valor del autocommit de la conexión.
  - ▶ **getAutoCommit()** -> obtiene el valor del autocommit de la conexión.
  - ▶ **createStatement()** -> devuelve un objeto que implementa la interfaz Statement con la que podremos crear sentencias SQL.

*Make the  
Connection*

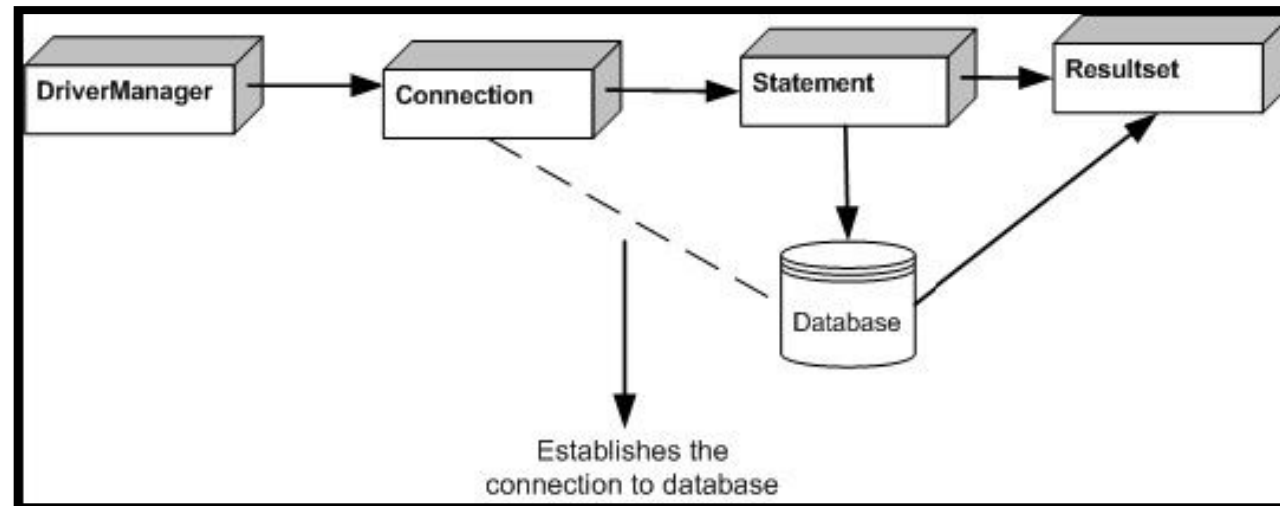


# UD 6 - MANEJO DE BBDD RELACIONALES

## 4 - Las interfaces Statement y ResultSet

### 4 – Las interfaces Statement y ResultSet

- ▶ La interfaz *Statement* nos permite crear y ejecutar sentencias en la base de datos. Los objetos que cumplen con esta interfaz los conseguimos pidiéndoselos a *Connection*.
- ▶ Por otro lado, la interfaz *ResultSet* nos permite procesar los datos obtenidos tras utilizar el método *executeQuery* de la interfaz *Statement*.
- ▶ La secuencia de pasos se muestra en la figura:



# UD 6 - MANEJO DE BBDD RELACIONALES

## 4 - Las interfaces Statement y ResultSet

- ▶ Abrimos y estudiamos el código **U6.P2.MDB.StatementResultSet**.
- ▶ Explicamos algunos elementos del código:

```
Connection conn = null;
Statement st = null;
ResultSet rs = null;

try {
    conn = DBUtils.getConnection();
    st = conn.createStatement();
    rs = st.executeQuery("SELECT * FROM CUSTOMERS WHERE customer_id <= 10");
```

```
while(rs.next()) {
    long id = rs.getLong(1);
    String name = rs.getString(2);
    double credit = rs.getDouble(5);
    System.out.println("Id = "+id+", name = "+name+", credit = "+credit);
}
```

- Creamos una conexión con la clase de utilidades DBUtils.
- Al objeto Connection le pedimos que nos proporcione un objeto Statement.
- Al objeto Statement le pedimos que ejecute una query y recogemos el resultado en un objeto ResultSet
- OJO: las sentencias SQL no llevan ;

- Recorremos el ResultSet y nos quedamos con los datos de las columnas que nos interesa mostrar.
- OJO: Los ResultSets solo se pueden recorrer UNA VEZ.

# UD 6 - MANEJO DE BBDD RELACIONALES

## 4 - Las interfaces Statement y ResultSet

- Hay que tener cuidado de respetar el número de las columnas y la concordancia de los tipos de dato al extraerlos del ResultSet.

```
long id = rs.getLong(1);  
String name = rs.getString(2);  
double credit = rs.getDouble(5);
```

#	Nombre	Tipo de datos
1	customer_id	INT
2	name	VARCHAR
3	address	VARCHAR
4	website	VARCHAR
5	credit_limit	DECIMAL

```
Id = 1, name = Raytheon, credit = 100.0  
Id = 2, name = Plains GP Holdings, credit = 100.0  
Id = 3, name = US Foods Holding, credit = 100.0  
Id = 4, name = AbbVie, credit = 100.0  
Id = 5, name = Centene, credit = 100.0  
Id = 6, name = Community Health Systems, credit = 100.0  
Id = 7, name = Alcoa, credit = 100.0  
Id = 8, name = International Paper, credit = 100.0  
Id = 9, name = Emerson Electric, credit = 100.0  
Id = 10, name = Union Pacific, credit = 200.0
```

También podemos usar el nombre de la columna:  
`long id = rs.getLong("customer_id");`

Realizamos los ejercicios 1-3  
del boletín de problemas

# UD 6 - MANEJO DE BBDD RELACIONALES

## 4 - Las interfaces Statement y ResultSet

### Método executeUpdate

- ▶ Solo usaremos executeQuery para sentencias tipo SELECT.
- ▶ Si queremos ejecutar un INSERT, un DELETE, un UPDATE, un CREATE TABLE o cualquier otra sentencia de DDL debemos usar **executeUpdate**.
- ▶ La firma de este método es: **int executeUpdate(String sql)**
  - ▶ Si se ejecuta un INSERT, DELETE o UPDATE devuelve el número de filas afectadas.
  - ▶ Si se ejecuta un CREATE TABLE o cualquier otra sentencia DDL siempre devuelve cero.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 4 - Las interfaces Statement y ResultSet

- Abrimos y estudiamos el código **U6.P3.MDB.StatementExecuteUpdate**.


```
conn = DBUtils.getConnection();
st = conn.createStatement();
int numFilas = st.executeUpdate("INSERT INTO CUSTOMERS "
    + "(name, address, website, credit_limit) "
    + "VALUES ('PepePhone', 'Avda. Pepe Phone S/N', null, 4000)");

System.out.println("Se han insertado "+numFilas+" filas");

numFilas = st.executeUpdate("DELETE FROM CUSTOMERS "
    + "WHERE NAME = 'PepePhone'");
System.out.println("Se han eliminado "+numFilas+" filas");
```

Usamos `executeUpdate` para lanzar dos sentencias (que no son `SELECT`) y mostramos el número de registros afectados.

También hemos ampliado la clase `DBUtils` para que ahora también pueda cerrar los objetos `Statement` y `ResultSet`



```
--- exec-maven-plugin:1.5.0
Se han insertado 1 filas
Se han eliminado 1 filas
```

Realizamos los ejercicios 4-6 del boletín de problemas

# UD 6 - MANEJO DE BBDD RELACIONALES

## 5 - PreparedStatement y CallableStatement

### 5 – PreparedStatement y CallableStatement

- Si queremos que nuestro programa permita consultar los datos de un cliente buscando por el nombre del cliente podríamos hacer:

```
System.out.println("Dime el nombre del cliente para ver sus datos:");
String nombre = sc.nextLine();
rs = st.executeQuery("SELECT * FROM CUSTOMERS WHERE NAME = '"+nombre+"'");

while(rs.next()) {
    long id = rs.getLong(1);
    String name = rs.getString(2);
    String address = rs.getString(3);
    String website = rs.getString(4);
    double credit = rs.getDouble(5);
    System.out.println("Id = "+id+", name = "+name+", address = "+address+
        ", website = "+website+", credit = "+credit);
}
```





# UD 6 - MANEJO DE BBDD RELACIONALES

## 5 - PreparedStatement y CallableStatement

### ► Ejecución “sin malas intenciones”:

Dime el nombre del cliente para ver sus datos:

Exelon

Id = 192, name = Exelon, address = Via Luminosa 162, Firenze, , website = http://www.

Se obtiene 1 registro

### ► Ejecución con una inyección de SQL:

Dime el nombre del cliente para ver sus datos:

MeLoInvento' OR '1'='1

Id = 177, name = United Continental Holdings, address = 2904 S Sal  
Id = 180, name = INTL FCStone, address = 5344 Haverford Ave, Phila  
Id = 184, name = Publix Super Markets, address = 1795 Wu Meng, Mua  
Id = 187, name = ConocoPhillips, address = Walpurgisstr 69, Munich  
Id = 190, name = 3M, address = Via Frenzy 6903, Roma, , website =  
Id = 192, name = Exelon, address = Via Luminosa 162, Firenze, , we  
Id = 208, name = Tesoro, address = Via Notoriosa 1942, Firenze, ,  
Id = 207, name = Northwestern Mutual, address = 1831 No Wong, Peki  
Id = 200, name = Enterprise Products Partners, address = Via Notor

Se obtienen TODOS los registros

Código disponible en  
U6.P4.MDB.InyeccionSql

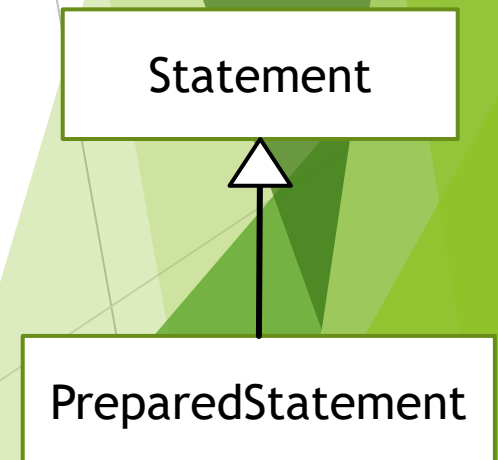
# UD 6 - MANEJO DE BBDD RELACIONALES

## 5 - PreparedStatement y CallableStatement

- ▶ Imagina el fallo de seguridad si escribimos este tipo de código para validar un usuario y contraseña.
- ▶ Para solventar estos problemas y obtener otros beneficios se usa: *PreparedStatement*.
- ▶ **PreparedStatement es una interfaz** que extiende de Statement y que nos va a aportar dos beneficios:
  - ▶ **Compone sentencias parametrizadas de forma segura.** Los objetos que implementen esta interfaz deben “**ESCAPAR**” los caracteres peligrosos, consiguiendo que dichos caracteres sean interpretados como texto y no como un símbolo sintáctico en la sentencia. En el ejemplo anterior, si escapamos los caracteres peligrosos la base de datos ejecutaría la siguiente sentencia:

```
SELECT * FROM CUSTOMERS WHERE NAME = 'MeloInvento OR \'1\'=\'1\''
```

```
11011010101101  
1011HACKED1111  
01010010000101
```





# UD 6 - MANEJO DE BBDD RELACIONALES

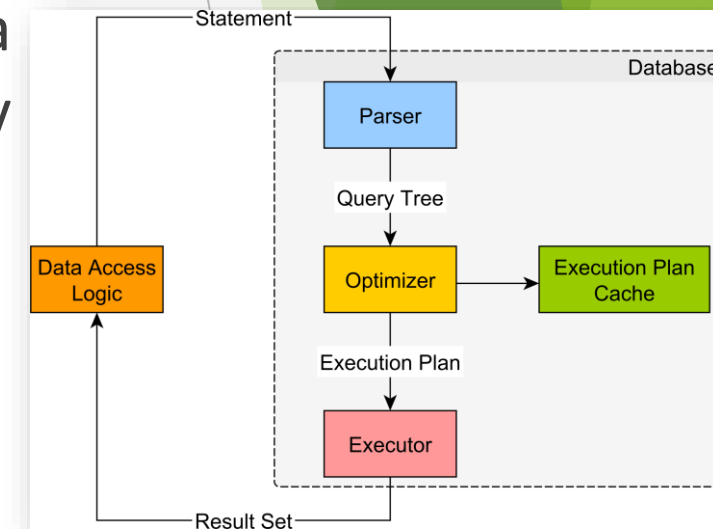
## 5 - PreparedStatement y CallableStatement

- **Precompila las sentencias SQL.** Las BBDD normalmente permiten que los lenguajes de programación les envíen las sentencias que van a ejecutar con anterioridad a la ejecución. Esto permite que la BD pueda “preparar” la ejecución de la sentencia. Esta preparación consiste en compilar la sentencia y elaborar un plan de ejecución estimado de la misma.

- Definición de plan de ejecución (ejemplo con SQL Server):

<https://www.sqlshack.com/es/planes-de-ejecucion-de-consultas-sql-server-viendo-los-planes/>

Es bastante habitual lanzar varias veces una misma sentencia con distintos valores en sus parámetros. Utilizar PreparedStatement acelera la ejecución de una misma sentencia repetidas veces.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 5 - PreparedStatement y CallableStatement

### Sintaxis y uso de un objeto PreparedStatement

```
Connection conn = null;
PreparedStatement st = null;
ResultSet rs = null;

try {
    conn = DBUtils.getConnection();

    System.out.println("Dime el nombre del cliente para ver sus datos:");
    String nombre = sc.nextLine();

    st = conn.prepareStatement("SELECT * FROM CUSTOMERS WHERE NAME = ?");
    st.setString(1, nombre);

    rs = st.executeQuery();
}
```

Al objeto Connection le pedimos que nos prepare una sentencia y marcamos los parámetros con ?

Después, le decimos que nos “rellene” los parámetros (los ?) con datos. Esto “saneará” los datos en caso de que hubiera caracteres raros

Código disponible en  
U6.P5.MDB.PreparedStatement

Y lo ejecutamos como una query normal

Dime el nombre del cliente para ver sus datos:  
MeLoInvento' OR '1'='1  
No se encuentra al cliente MeLoInvento' OR '1'='1



**Bye bye SQL  
Injection**

# UD 6 - MANEJO DE BBDD RELACIONALES


## 5 - PreparedStatement y CallableStatement

- PreparedStatement se usa muchísimo. Vemos otro ejemplo:

```
System.out.println("ALTA DE NUEVO CLIENTE");
System.out.println("Dime el nombre del cliente:");
String nombre = sc.nextLine();
System.out.println("Dime su dirección postal:");
String direccion = sc.nextLine();
System.out.println("Dime su sitio web:");
String web = sc.nextLine();
System.out.println("Dime su límite de crédito:");
double limite = sc.nextDouble();
```

```
st = conn.prepareStatement("INSERT INTO CUSTOMERS (name, address, "+
    "website, credit_limit) VALUES (?, ?, ?, ?)");
st.setString(1, nombre);
st.setString(2, direccion);
st.setString(3, web);
st.setDouble(4, limite);
```

```
int numFilas = st.executeUpdate();
System.out.println(numFilas+" filas insertadas");
```



```
ALTA DE NUEVO CLIENTE
Dime el nombre del cliente:
Sotero SA
Dime su dirección postal:
Avda. Sotero Hernandez S/N
Dime su sitio web:
http://www.iessoterohernandez.es
Dime su límite de crédito:
5000
1 filas insertadas
```

Código disponible en  
U6.P6.MDB.InsertPreparedStatement

Realizamos los ejercicios 7-8 del  
boletín de problemas

# UD 6 - MANEJO DE BBDD RELACIONALES

## 5 - PreparedStatement y CallableStatement

### ► CallableStatement

- Esta interfaz permite ejecutar un procedimiento almacenado (en PL/SQL o equivalente) en una base de datos. Aunque no la estudiaremos, sí tenemos que saber para qué se usa.
- Un procedimiento almacenado (stored procedure) es un programa que se guarda en la BD y que puede ser utilizado en las siguientes situaciones:
  - Para validar datos antes/después de una operación (mediante triggers).
  - Para realizar procesamiento masivo de registros (1000 registros o más) de forma más eficiente.
  - Para garantizar que un tarea se realiza de forma “transaccional”.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 6 - Los tipos de datos SQL vs Java

### 6 - Los tipos de datos SQL vs tipos Java

- La mayoría de las conversiones entre los tipos de datos SQL y los tipos de datos Java están bastante claras, **pero con las fechas HAY LÍO.**

Tipo SQL	Tipos Java
Numeric(n), int, tinyint, bigint	int o long (o sus Wrappers)
Numeric(n, d), float, real	float, double (o Wrappers), BigDecimal
char, varchar, text	char, String
Date	java.util.Date, java.sql.Date, java.time.Date ¿? ¡Confusión!

- Vamos a aclararlo de una vez para siempre.




# UD 6 - MANEJO DE BBDD RELACIONALES

## 6 - Los tipos de datos SQL vs Java

- Primero vamos a observar la siguiente tabla:

Lenguaje	Tipo	Dato de ejemplo	Basado en
SQL	DATE	12/04/2019	UTC
Java	java.util.Date	12/04/2019 17:16:49.413481	UTC
Java	java.sql.Date	12/04/2019 00:00:00.000000	UTC
Java	Java.time.LocalDate	12/04/2019	Otro sistema

- El tipo SQL DATE se basa en el sistema **universal de tiempo coordinado** (UTC) que internamente guarda el número de **milisegundos transcurridos desde el 01/01/1970 GMT**. Esto permite poder comparar dos fechas de forma inequívoca sin importar la franja horaria donde se estableció la hora.
- **SQL DATE codifica una fecha sin HH:MM:SS**, que es lo normal. Sin embargo, **java.util.Date sí lleva información HH:MM:SS.** 

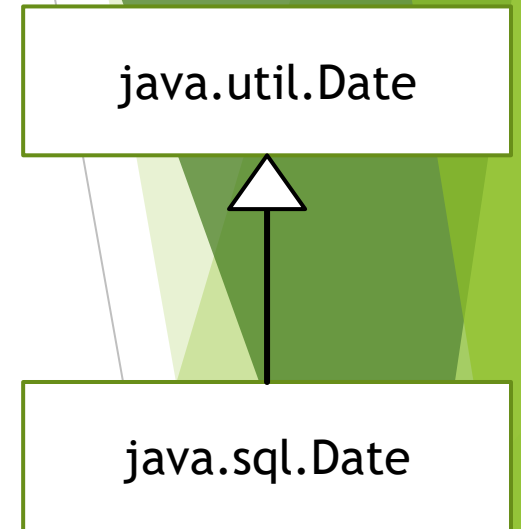




# UD 6 - MANEJO DE BBDD RELACIONALES

## 6 - Los tipos de datos SQL vs Java

- ▶ Y a “los Javeros” se les ocurrió un “apaño”. Crearon **java.sql.Date** como hija de **java.util.Date** pero “normalizando” (poniendo a cero) la parte de las horas, minutos, segundos y milisegundos.
- ▶ Así que cada vez que leamos o escribamos una fecha en la base de datos debemos utilizar un tipo **java.sql.Date**.
- ▶ Sólo nos quedará aprender a convertir entre este formato y los otros formatos de fecha disponibles en Java:
  - ▶ **java.util.Date**: ya desaconsejado (*deprecated*), aunque todavía se ve mucho código con este tipo.
  - ▶ **java.time.LocalDate**: es la recomendada desde java 1.8 y la deberíamos usar preferiblemente en nuestras clases.



Descarga y estudia  
el código  
U6.P7.MDB.ConversionSqlDate

# UD 6 - MANEJO DE BBDD RELACIONALES

## 7 - Transacciones

### 7 - Transacciones

- ▶ El concepto de **transacción** en un programa o base de datos procede del mundo financiero y nos debe evocar mentalmente a una **transferencia bancaria**.
- ▶ En una transferencia bancaria si el sujeto A transfiere 20 euros al sujeto B ocurren las siguientes operaciones:
  - ▶ Al sujeto A se le restan 20 euros de su cuenta.
  - ▶ Al sujeto B se le suman 20 euros a su cuenta
- ▶ ¿Qué pasaría si se corta la luz cuando se le ha hecho la resta al sujeto A pero todavía no se le ha hecho la suma al sujeto B? **el dinero desaparece, ERROR GRAVE...**





# UD 6 - MANEJO DE BBDD RELACIONALES

## 7 - Transacciones

► Una **transacción** es un conjunto de operaciones que deben realizarse “atómicamente”, es decir, de forma indivisible. O hacemos todas las operaciones o ninguna.

► Las BBDD permiten la realización de transacciones y para conseguirlo se basan en tres elementos:

► **Segmento de rollback:** es una zona del disco duro, distinta a la zona de datos, en la que se van grabando los cambios que realizan las operaciones de la transacción. De algún modo permiten “simular” que se realizan los cambios.

► **Sentencia commit:** si todo fue bien entonces ejecutamos un *commit* y los cambios realizados en el segmento de rollback se copian sobre el segmento de datos.

► **Sentencia rollback:** si algo falló se ejecuta un *rollback* que aborta la transacción y el segmento de datos queda intacto.

Transaction without problems

```
BEGIN TRANSACTION;  
✓ STEP 1  
✓ STEP 2  
✓ STEP 3  
✓ STEP 4  
...  
✓ STEP n  
COMMIT;
```

Incomplete Transaction

```
BEGIN TRANSACTION;  
✓ STEP 1  
✓ STEP 2  
✗ STEP 3 → ROLLBACK;  
STEP 4  
...  
STEP n  
COMMIT;
```

# UD 6 - MANEJO DE BBDD RELACIONALES

## 7 - Transacciones

- ▶ La interfaz **Connection** nos permite preparar una transacción y llevarla a cabo:

- ▶ **Preparación:** tenemos que desactivar la opción de **autocommit** para conseguir que realicen las operaciones de forma atómica.

```
conn.setAutoCommit(false);
```

- ▶ **Transacción:** hacemos los cambios necesarios sobre la BD, lanzando INSERTS, DELETE... lo que sea necesario.

- ▶ Si todo fue bien hacemos un `conn.commit();` para finalizar la transacción y que se reflejen los cambios en el segmento de datos.

- ▶ Si algo falló entonces hacemos un `conn.rollback();` para abortar la transacción, deshaciendo los cambios producidos en el segmento de rollback. En esta situación también debería lanzarse una excepción.

Descarga y estudia  
el código  
U6.P8.MDB.Transaccion

Realizamos los  
ejercicios 9-10 del  
boletín de problemas



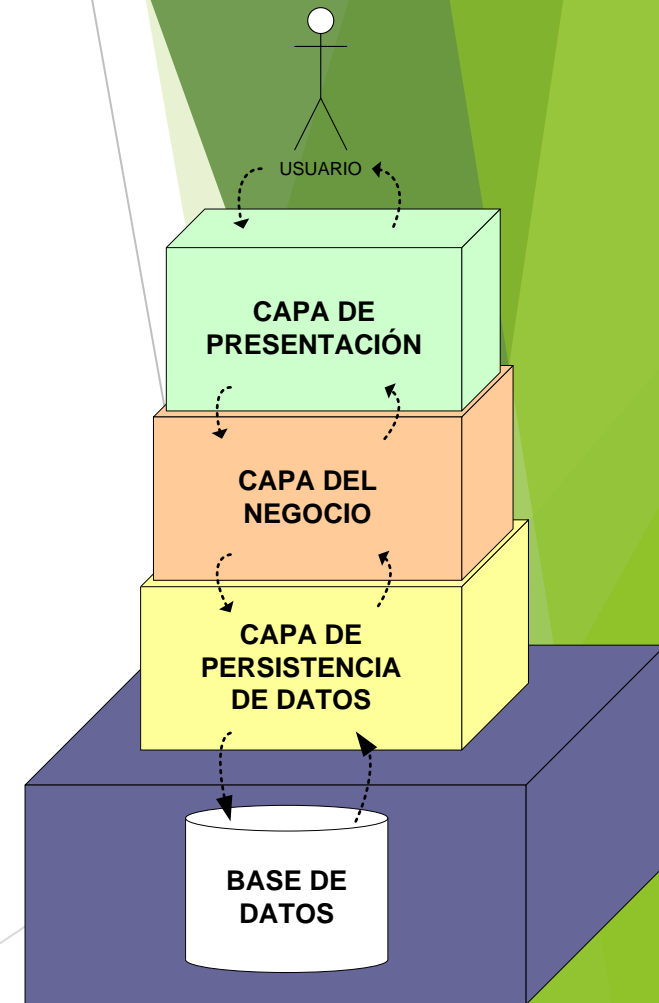
# UD 6 - MANEJO DE BBDD RELACIONALES

## 8 - La persistencia en una aplicación por capas

### 8 – La persistencia en una aplicación por capas

- ▶ Recordemos que en la **capa de negocio** tenemos las clases que representan un concepto propio del negocio: factura, pedido, cliente... con sus propiedades y métodos.
- ▶ Para poder persistir los objetos en una BD relacional necesitamos generar código para **salvar las diferencias entre dos modelos distintos**:
  - ▶ **Modelo relacional**: los datos se almacenan como columnas de una tabla. Las tablas se relacionan mediante claves primaria-ajena y se busca que se cumpla la 3ª Forma Normal para garantizar la coherencia de los datos.
  - ▶ **Modelo de objetos**: los datos se almacenan en las propiedades del objeto, pero no se tiene en cuenta si los datos se almacenan en una tabla o varias. Ejemplo: si una clase Cliente tiene como propiedad una lista de cuentas corrientes deberá persistirse en, al menos, dos tablas:

objeto Cliente -> tabla cliente + tabla cuentas



# UD 6 - MANEJO DE BBDD RELACIONALES

## 8 - La persistencia en una aplicación por capas

- ▶ Es fácil imaginar la **complejidad** que puede alcanzar el código necesario para persistir una capa de negocio de tamaño mediano o grande. Hacerlo “a mano” conlleva afrontar los siguientes **problemas**:
  - ▶ Mapear entre objetos y tablas adecuando los tipos de datos.
  - ▶ Gestionar las claves primarias y ajenas en las tablas manteniendo la integridad referencial.
  - ▶ Gestionar el “ciclo de vida” de los objetos. Ejemplo: objetos que “nacen” en memoria y deben ser persistidos. Objetos que “nacen” en la BD, se modifican en la aplicación y deben persistirse los cambios...
  - ▶ Gestionar los identificadores de los objetos. Dado que un objeto puede “nacer” en memoria su identificador estará a *null* hasta que se persista y la BD le asigne una clave primaria.
  - ▶ Gestionar las transacciones...



# UD 6 - MANEJO DE BBDD RELACIONALES

## 8 - La persistencia en una aplicación por capas

- Los **frameworks de Object-Relational Mapping (ORM)** vienen a salvarnos de estos problemas (y crearnos otros problemas nuevos, claro)



- Un ORM es una capa que se sitúa entre la BD y nuestra capa de negocio y que hace toda la conversión entre ambos modelos de forma transparente.
- El impacto en la capa de negocio consiste en definir una **clases con anotaciones llamadas entidades** que indican cómo se relacionan las propiedades: `@Entity`, `@OneToMany`, `@ManyToOne`... de este modo, el ORM deduce cuál es la estructura de tablas adecuada en la BD.

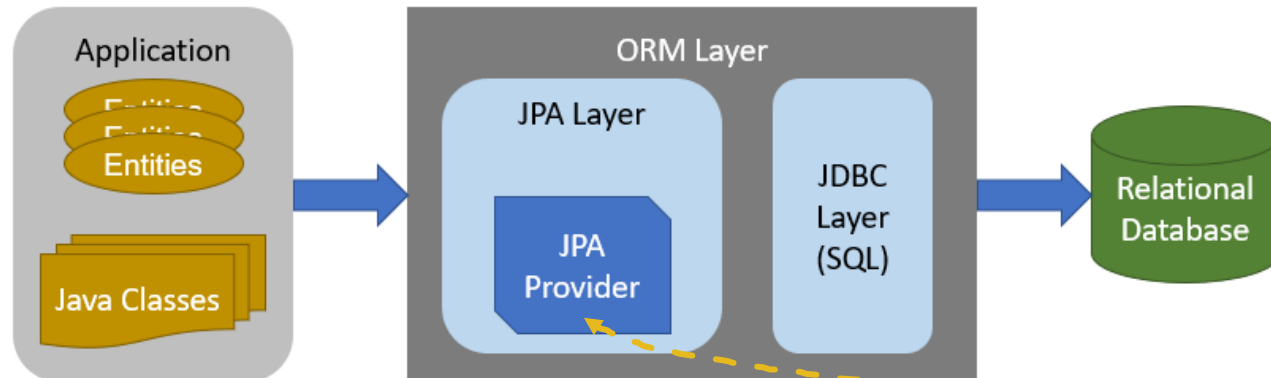




# UD 6 - MANEJO DE BBDD RELACIONALES

## 8 - La persistencia en una aplicación por capas

- ▶ Una vez configurado el ORM obtendremos los siguientes beneficios:
  - ▶ El esquema de las tablas de la BD se crean automáticamente.
  - ▶ Nuestras clases de la capa de negocio tendrán disponibles métodos para realizar búsquedas: `findAll()`, `findById(Long id)` o para grabar datos: `persist()`...
- ▶ Dado que hay muchos ORM para Java y cada uno tiene sus peculiaridades, los “padres” del lenguaje decidieron “estandarizarlos” y crearon la **Java Persistence API (JPA)** que no es más que un conjunto de **interfaces** que deben implementar los ORM. De este modo podremos cambiar de ORM sin apenas tocar el código.



# UD 6 - MANEJO DE BBDD RELACIONALES

## 8 - La persistencia en una aplicación por capas

### ► Ventajas e inconvenientes de los ORM:

Ventajas	Inconvenientes
<ul style="list-style-type: none"><li>• Reducción de los tiempos y costes del desarrollo.</li><li>• Gestión automática del ciclo de vida de los objetos y de las transacciones.</li><li>• Mejor adaptación al cambio.</li><li>• Abstracción de la peculiaridades de la base de datos usada.</li><li>• Seguridad de la capa de acceso a datos contra ataques.</li></ul>	<ul style="list-style-type: none"><li>• Curva de aprendizaje del nuevo "léxico" asociado al ORM.</li><li>• El desarrollador "pierde el control" de lo que está haciendo al delegarlo al ORM.</li><li>• El ORM es un componente "pesado", siempre será más lento que una solución hecha a mano en JDBC.</li><li>• Los ORM tienen tendencia a volverse lentos si no se optimizan bien.</li></ul>

- El estudio de un framework ORM se realizará en el 2º curso del ciclo formativo.



# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

### Haciéndolo “a mano”. El patrón Data Access Object (DAO)

- ▶ Si nos enfrentamos a un problema donde **el rendimiento en velocidad sea crítico** y no queremos utilizar un ORM tendríamos que desarrollar la capa de persistencia. El código resultante es un poco complejo y repetitivo, a cambio ganamos en velocidad y en el control total de nuestro código.
- ▶ El patrón de diseño DAO permite solucionar el problema de la persistencia consiguiendo un **acoplamiento bajo** entre la capa del negocio y la de persistencia, mejorando así la adaptación al CAMBIO.
- ▶ La idea de este patrón consiste en tener un conjunto de objetos (los DAO) **que concentren todo el trabajo con la BD**. De este modo, los objetos de la capa de negocio solo tienen que llamar a un DAO para pedirle sus servicios: *guárdame este objeto, dame un objeto de tipo X buscándolo por su identificador, dame todos los objetos de este tipo...*




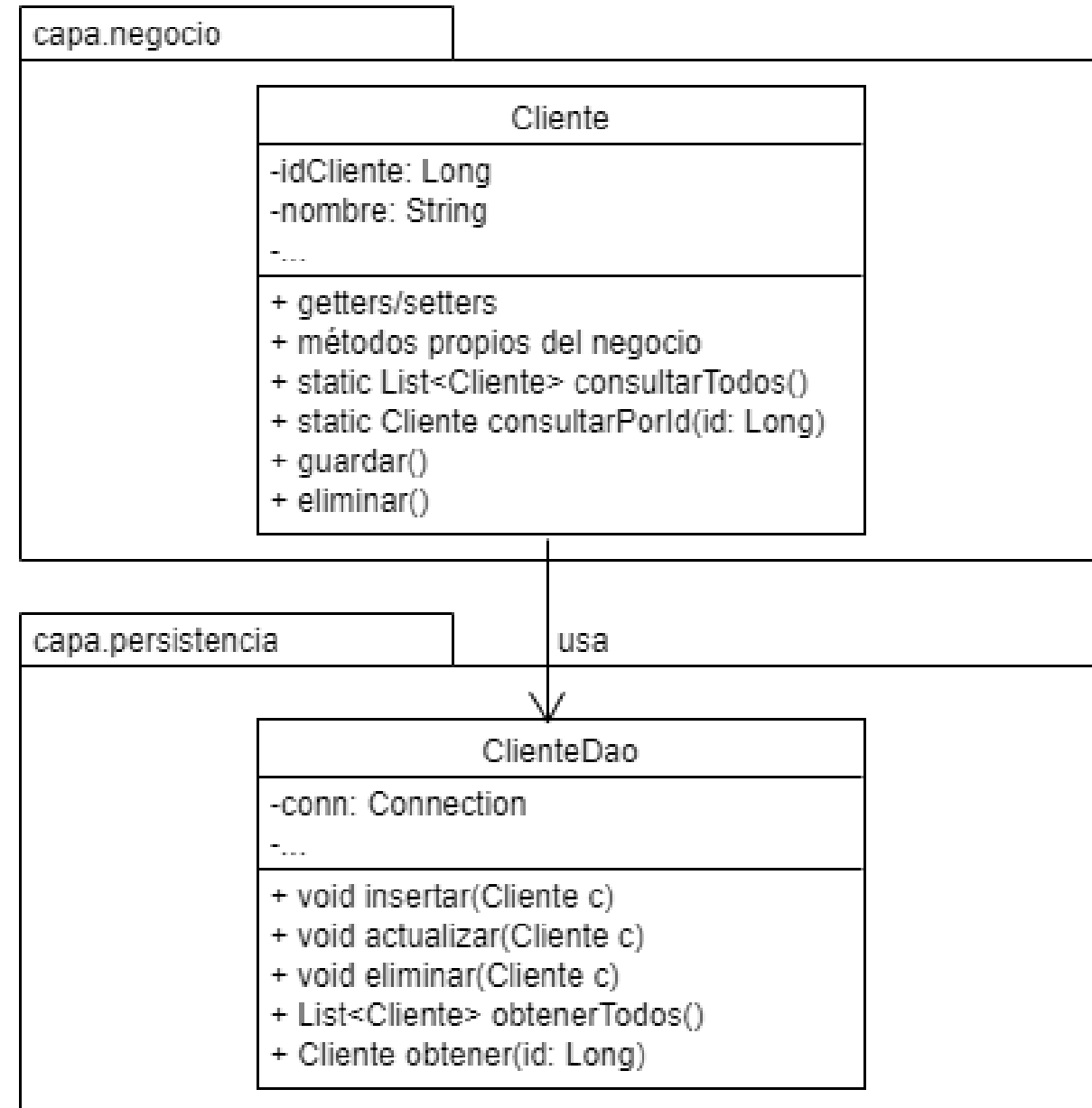
El **acoplamiento** entre dos piezas de software mide el grado de interdependencia entre ambas piezas. Lo ideal es que el acoplamiento sea lo **más bajo posible**.



# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

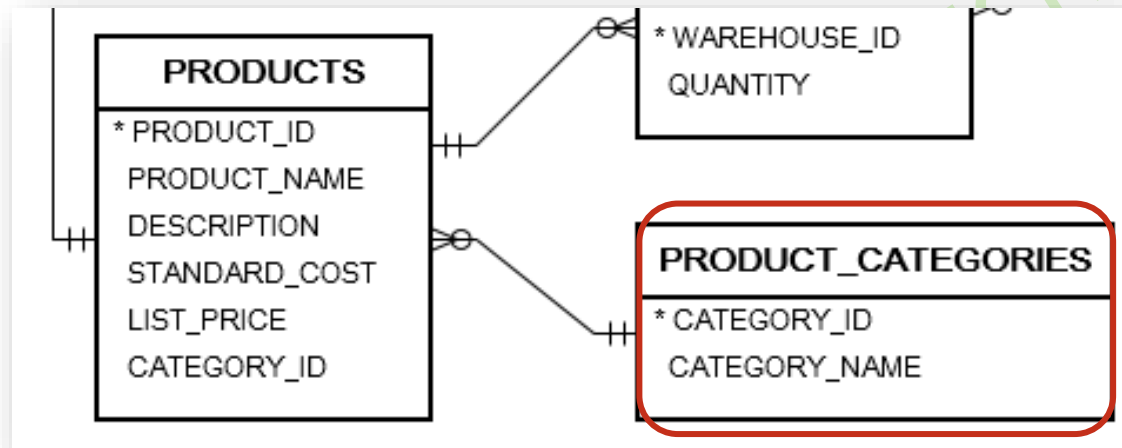
- ▶ El patrón DAO gráficamente sería algo así: 
- ▶ Cada clase de la capa de negocio tiene tu correspondiente objeto DAO.
- ▶ Cada objeto DAO implementa, como mínimo, las 4 operaciones básicas: **Create**, **Retrieve**, **Update**, **Delete** (crear, devolver, actualizar y borrar). A esto coloquialmente se le llama “**hacer el CRUD de una clase**”.
- ▶ Cada objeto DAO concentra toda la gestión necesaria para persistir el objeto de la capa de negocio que corresponda.
- ▶ El objeto de la capa de negocio utiliza el DAO, olvidándose de los detalles de la BD, para crear otros métodos: consultarTodos, guardar... que son los que ofrece a su capa superior.



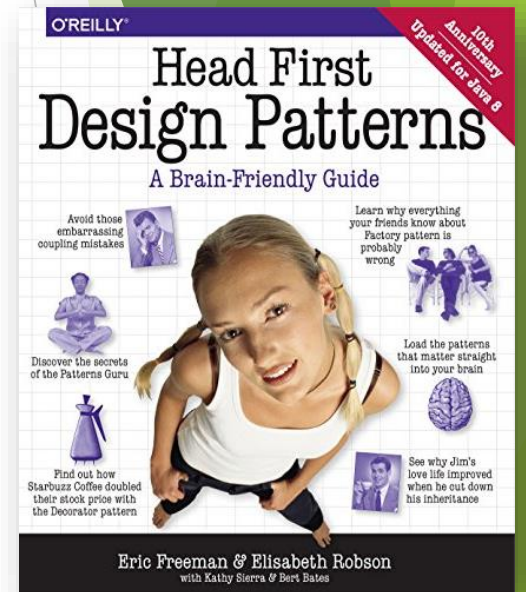
# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

- A modo de ejemplo he aplicado este patrón a la tabla PRODUCT\_CATEGORIES nuestro esquema COMPANY.



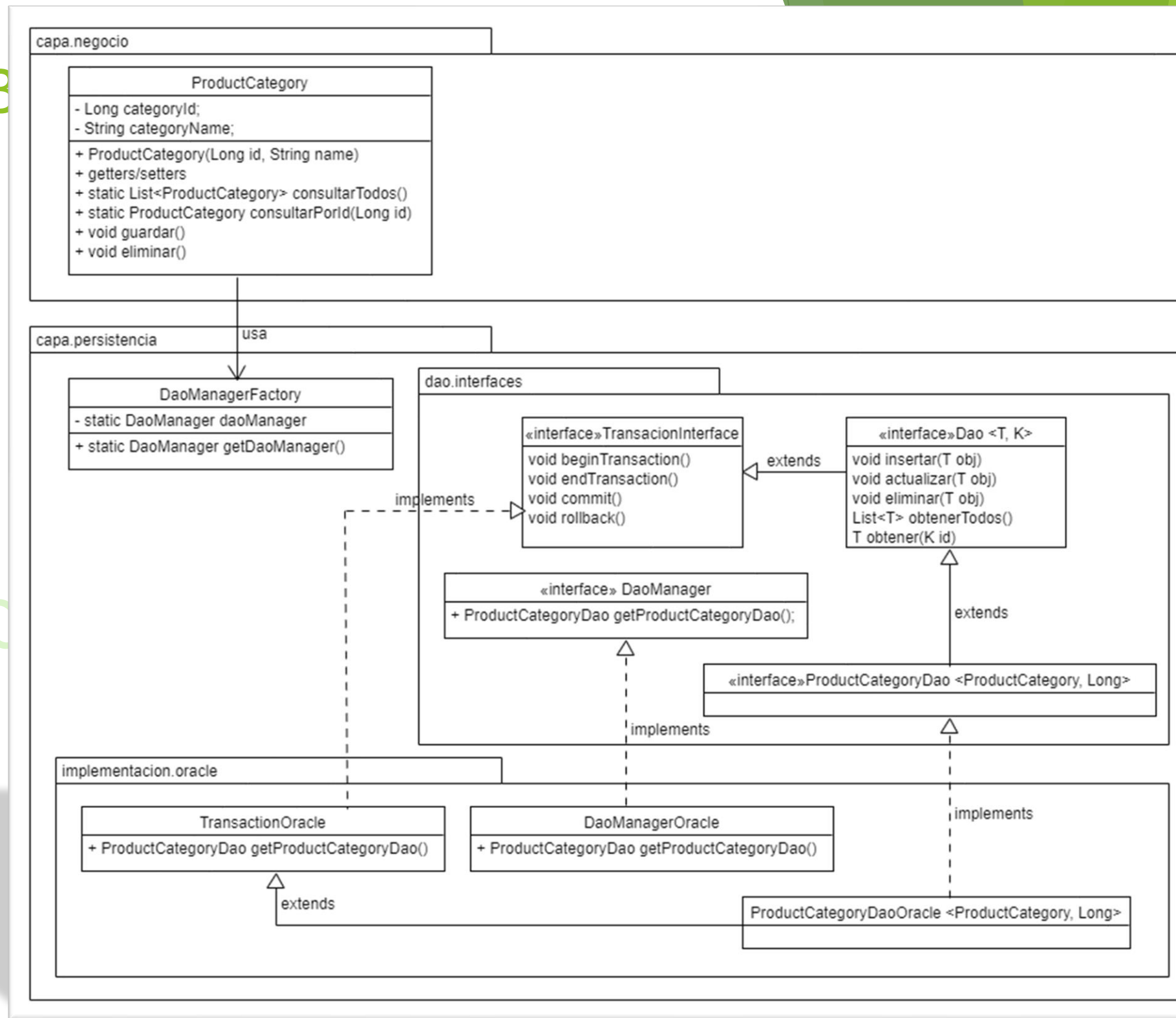
- Las primeras veces que nos enfrentamos a un patrón nuevo se nos hace extraño. **Hay que estudiarlos detenidamente, buscando la “intención de diseño” y el beneficio que aporta.**
- Para rizar el rizo, he aplicado además, otros patrones de diseño en nuestro código de ejemplo. El código resultante es complejo, por este motivo, debe ser leído y estudiado CON MUUUUCHA CALMA.



# UD 6 - MANEJO DE BB

## ANEXO I - EL PATRÓN

- El diagrama UML del código es:



# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

### ► Algunas claves para entender el código de la **capa de persistencia**:

- Creamos una jerarquía de interfaces (Dao y ProductCategoryDao) para conseguir **separar la funcionalidad de la implementación**. La idea es conseguir el mismo efecto que cuando hacemos:

**Funcionalidad**

List lista = new ArrayList();

**Implementación**

- De este modo, crearemos una clase (ProductCategoryDaoOracle) que implementa ProductCategoryDao para una base de datos Oracle. Pero podríamos hacer otra implementación para MySQL o cualquier otra BD.
- Por otro lado, creamos una interfaz DaoManager que también separa funcionalidad e implementación. La clase DaoManagerOracle implementa esta interfaz con la idea de centralizar el acceso a los objetos DAO.



# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

- ▶ Creamos una clase **DaoManagerFactory** para que el código de la capa de negocio no tenga que escoger una implementación de DaoManager. Esta decisión se toma en un único punto de nuestro código, la clase DaoManagerFactory, permitiendo cambiar de implementación (Oracle, MySQL...) simplemente tocando una línea de código (*patrón FactoryMethod*)
- ▶ Algunas claves para entender el código de la **capa de negocio**:
  - ▶ La clase **ProductCategory** crea unos métodos (consultarTodos, guardar, eliminar...) que ofrece a su capa superior. Estos métodos son los únicos que conectan con la BD a través del objeto DAO.
  - ▶ Los métodos de ProductCategory usan un objeto que implementa la interfaz ProductCategoryDao. El lío está en que este objeto se lo pedimos a un objeto que implementa la interfaz DaoManager, que a su vez se lo pedimos a la clase DaoManagerFactory... ¡casi na!

Descarga y estudia CON  
CALMA  
el código  
U6.P9.MDB.DaoCategorias

Si no lo entiendes,  
no pasa nada... Es  
cuestión de tiempo





# UD 6 - MANEJO DE BBDD RELACIONALES

## ANEXO I - EL PATRÓN DAO

### ► Para saber más:

- **PATRÓN DAO** - Lista de reproducción sobre JDBC de Makigas en Youtube:

<https://www.youtube.com/playlist?list=PLTd5ehlJ0goMKGkcD6cB7enP0nnyYiEzw>

Todos los vídeos son interesantes, pero los vídeos del 11 al 15 hacen un DAO completo para una aplicación de gestión de alumnado.

- **ORM** - Lista de reproducción sobre JPA/Hibernate de Makigas en Youtube:

<https://www.youtube.com/playlist?list=PLTd5ehlJ0goPcnQs34i0F-Kgp5JHX8UUv>

MUY  
RECOMENDABLE





# UD 6 - MANEJO DE BBDD RELACIONALES

## Cierre de la unidad

- ▶ En esta unidad hemos dejado la capa de negocio y hemos presentado las tareas que pueden desarrollarse en la **capa de persistencia**.
- ▶ Hemos trabajado con JDBC para acceder directamente a una base de datos y hemos aprendido a escribir este tipo de código.
- ▶ Sin embargo, **el código resultante es repetitivo, tedioso y propenso a errores** (“boilerplate”). Esto nos invita a usar “un intermediario” para acceder a la base de datos que nos facilite las cosas.
- ▶ **Los ORM son ampliamente utilizados** porque automatizan el manejo de la BBDD, permitiendo ahorrar tiempo y esfuerzo de desarrollo.
- ▶ Es probable que no uséis directamente JDBC en la empresa, sin embargo haberlo estudiado nos permite **valorar que cada vez que usamos un ORM estamos “a hombros de gigantes”**.



## UD 6 - MANEJO DE BBDD RELACIONALES

*The End*