

Unidad 2. Manejo de conectores.

Resumen

En esta unidad veremos las herramientas necesarias para desarrollar aplicaciones que gestionen información que esté almacenada en distintos tipos de base de datos. Dentro de las bases de datos que se utilizarán, estas serán:

- Base de datos relacionales: MySQL.
- Base de datos objeto relacional: PostgreSQL.
- Base de datos orientada a objetos.

Las dos primeras serán analizadas a nivel teórico y práctico mientras que el último tipo será analizado desde un punto de vista teórico.

Contenidos

1. Introducción.
2. Protocolos de acceso a base de datos.
3. Establecimiento de conexiones..
4. Ejecución de sentencias de descripción y de modificación de datos.
5. Ejecución de consultas
6. Manejo del resultado de las consultas.
7. Ejecución de procedimientos almacenados en una base de datos
8. Gestión de transacciones.

Objetivos

1. Conocer las características de los distintos tipos de bases de datos.
2. Establecer protocolos de acceso a bases de datos.
3. Establecer conexiones.
4. Gestionar objetos.

5. Trabajar con tipos de datos básicos y estructurados.
6. Ejecutar sentencias de descripción y modificación de datos
7. Ejecutar consultas.
8. Ejecutar procedimientos almacenados
9. Gestionar transacciones

Contenidos

1. Introducción
2. Características
3. Ventajas e inconvenientes
4. Gestores de Bases de Datos Orientadas a Objetos
5. API
6. Características de las bases de datos objeto-relacionales
7. Gestores de bases de datos objeto-relacionales
8. Gestión de bases de datos objeto-relacionales con Java

Introducción

SOFTWARE A UTILIZAR

Para esta unidad el software que habrá que utilizar será **MySQL Workbench** como base de datos relacional y **PostgreSQL** como base de datos objeto relacional.

Actualmente, las **bases de datos relacionales** constituyen el sistema de almacenamiento más extendido.

DEFINICIÓN

Una base de datos relacional se puede definir, de una manera simple, como aquella que **presenta la información en tablas con filas y columnas**.

Una **tabla o relación** es una colección de objetos del mismo tipo (filas o tuplas).

A continuación, se muestra una imagen de una tabla:



idreg	Temporada	Lugar	equicasero	golescasero	equiforaneo	golesforaneo
1	1955-1956	París	Real Madrid		4 Stade Reims	3
2	1956-1957	Madrid	Real Madrid		2 Fiorentina	0
3	1957-1958	Bruselas	Real Madrid		3 A.C. Milán	2
4	1958-1959	Stuttgart	Real Madrid		2 Stade Reims	0
5	1959-1960	Glasgow	Real Madrid		7 Eintrach de Fra	3
6	1960-1961	Berna	Benfica		3 Barcelona	2
7	1961-1962	Amsterdam	Bentica		5 Real Madrid	3
8	1962-1963	Londres	A.C. Milán		2 Benfica	1
9	1963-1964	Viena	Inter de Milán		3 Real Madrid	1
10	1964-1965	Milán	Inter de Milán		1 Benfica	0
11	1965-1966	Bruselas	Real Madrid		2 Partizán de Bel	1
12	1966-1967	Lisboa	Celtic de Glasgow		2 Inter de Milán	1
13	1967-1968	Londres	Manchester United		4 Benfica	1
14	1968-1969	Madrid	A.C. Milán		4 Ajax de Amster	1
15	1969-1970	Milán	Feyenoord		4 Celtic de Glasg	1
16	1970-1971	Londres	Ajax de Amsterdam		2 Panathinaikos	1
17	1971-1972	Rotterdam	Ajax de Amsterdam		2 Inter de Milán	0
18	1972-1973	Belgrado	Ajax de Amsterdam		1 Juventus	0
19	1973-1974	Bruselas	Bayern de Munich		1 Atl. De Madrid	1
20				4		1
21	1974-1975	París	Bayern de Munich	2	Leeds United	0
22	1975-1976	Glasgow	Bayern de Munich	1	Saint Etienne	0
23	1976-1977	Roma	Liverpool	3	Borussia Moen	1

En cada **tabla** de una base de datos se elige **una clave primaria** para identificar de manera **única** a cada fila de la misma.

El sistema gestor de bases de datos (SGBD) **gestiona el modo en que los datos se almacenan, mantienen y recuperan.**

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

Esta API tiene tres objetivos principales:

1. Soporta SQL: permite construir sentencias SQL.
2. Aprovechar la experiencia de los APIs de bases de datos existentes.
3. Es lo más sencilla posible.

El desfase objeto-relacional.



DEFINICIÓN

El **desfase objeto-relacional**, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos.

Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación:** el programador debe conocer el lenguaje de programación orientado a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos:** en las bases de datos relacionales siempre hay restricciones en el uso de tipos, mientras que la programación orientada a objetos utiliza tipos de datos mas complejos.
- **Paradigma de programación:** es necesaria hacer una traducción de los objetos del modelo orientado a objetos a tablas y tuplas del modelo Entidad-Relación (E/R), lo que implica diseñar dos diagramas diferentes para el diseño de la aplicación.

El **modelo relacional trata con relaciones y conjuntos** debido a su naturaleza matemática. Sin embargo, **el modelo de POO trata con objetos y las asociaciones entre ellos**. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos.

La escritura (y de manera similar la lectura) mediante JDBC implica:

- Abrir una conexión.

- Crear una sentencia en SQL.
- Copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto.

Esto es sencillo para un caso simple, pero complicado si el objeto posee muchas propiedades. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

DEFINICIÓN

Este **problema** es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en POO.

EJEMPLO

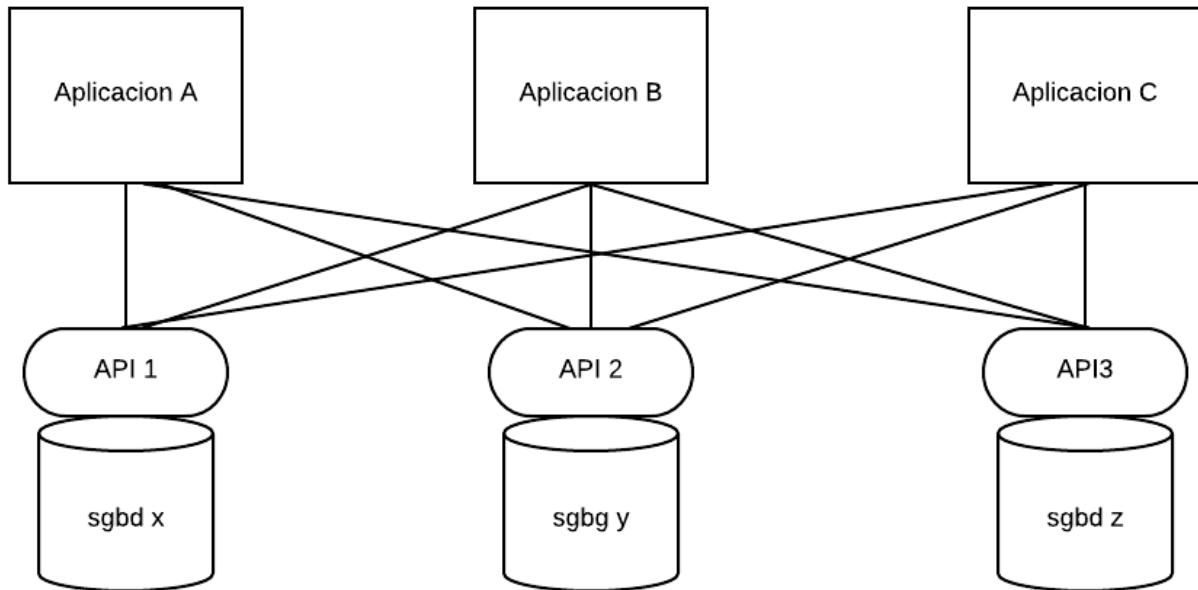
Como ejemplo de desfase objeto-relacional, podemos poner el siguiente:

Supón que tienes un objeto `Agenda Personal` con un atributo que sea una colección de objetos de la clase `Persona`. Cada persona tiene un atributo `teléfono`.

Al transformar este caso a relacional, se ocuparía más de una tabla para almacenar la información, conllevando varias sentencias SQL y bastante código.

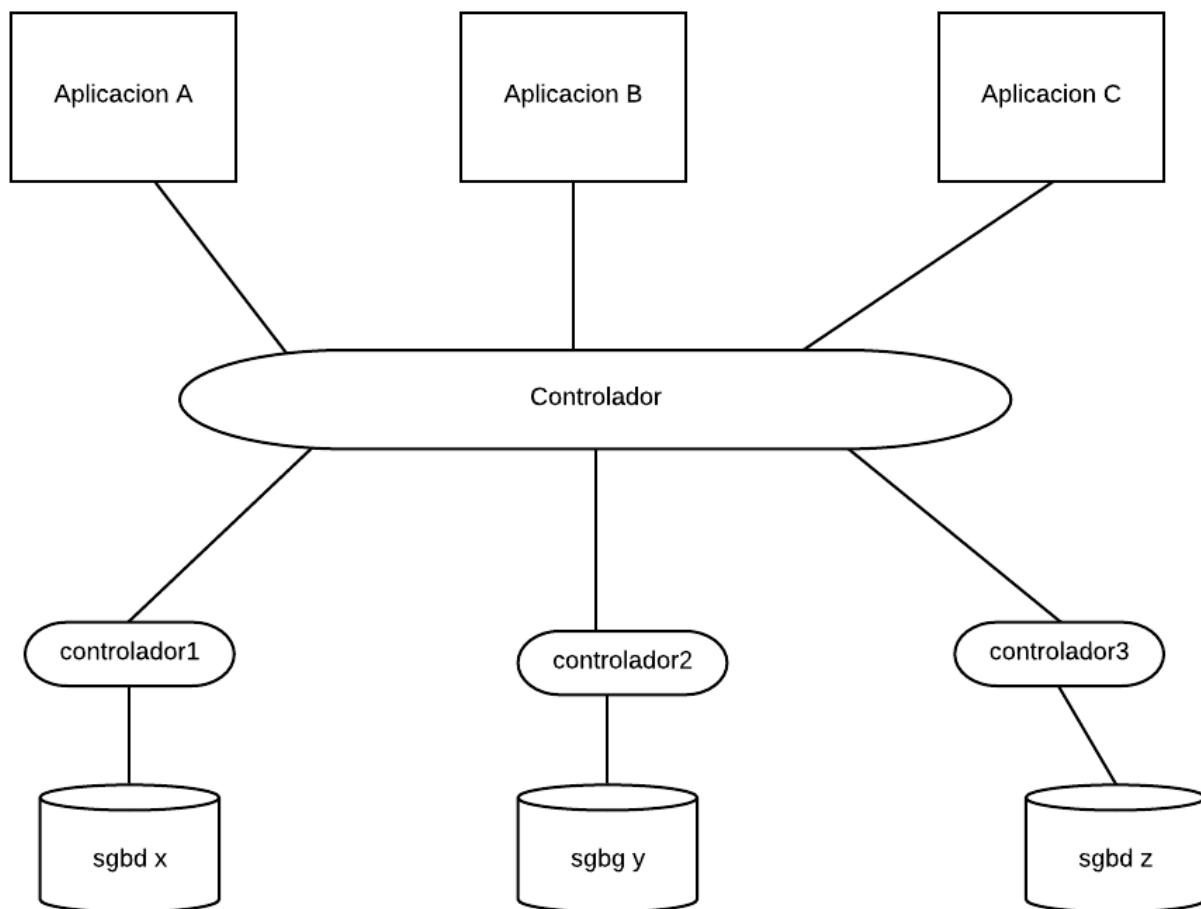
Protocolos de acceso a bases de datos

Inicialmente, cada empresa desarrolladora de un SGBD implementaba soluciones propietarias, pronto se dieron cuenta de que colaborando podían sacar mayor rendimiento y avanzar mucho más rápidamente.



La llegada del **ODBC** representó un avance en la **interoperabilidad** entre bases de datos y lenguajes de programación. La mayoría de empresas desarrolladoras de SGBDs incorporaron los **drivers** de conectividad a las utilidades de sus sistemas y los lenguajes de programación más importantes desarrollaron bibliotecas específicas para soportar el API ODBC.

Aunque la industria aceptó ODBC como medio principal para acceso a bases de datos en Windows, en Java no se introduce bien debido a su complejidad.



Por es razón surge JDBC.

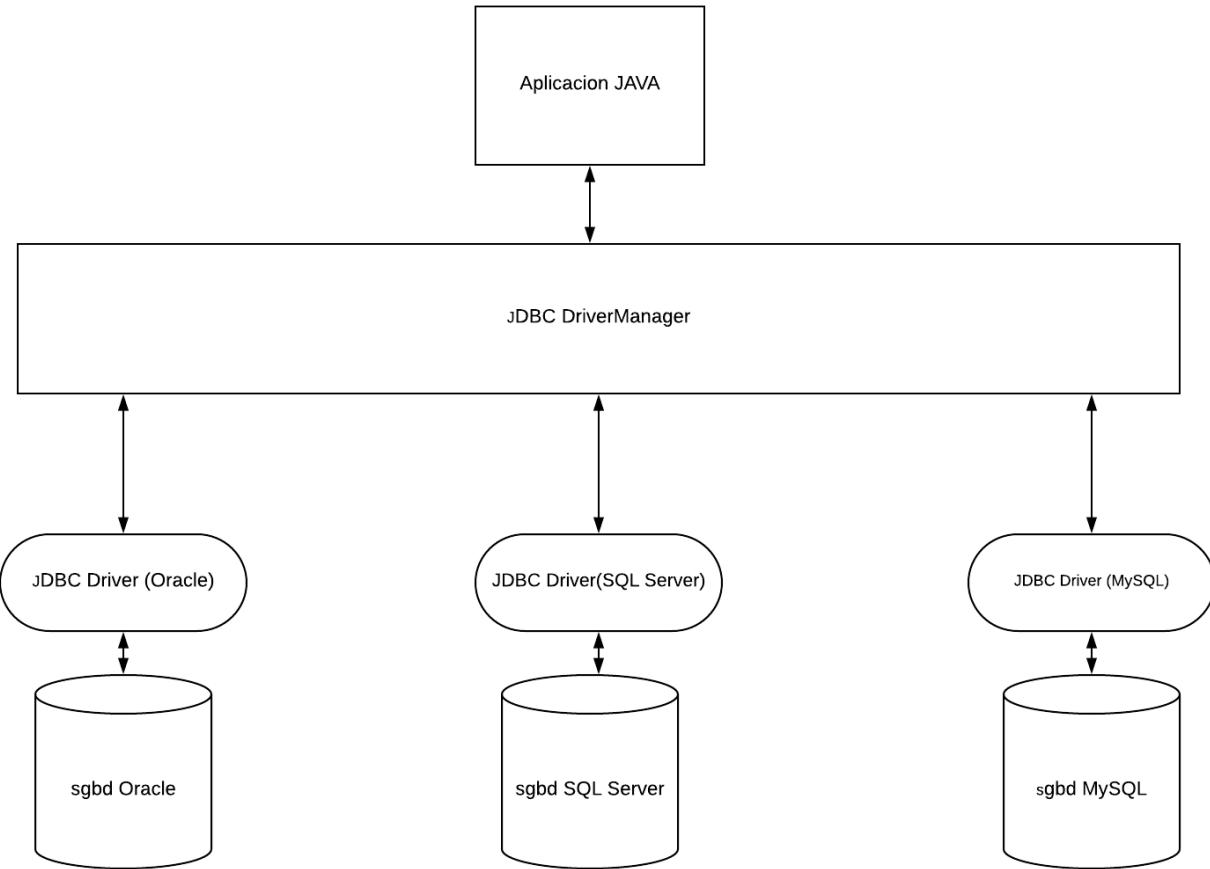


DEFINICIÓN

JDBC se trata de un API, similar a ODBC en cuanto a funcionalidad, implementada específicamente para usarse con Java.

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

JDBC es similar en estructura a ODBC. Una aplicación JDBC está compuesta de varias capas, como se muestra en la figura:



La **capa superior** en este modelo es la aplicación Java. Las aplicaciones Java son **portables**, es decir, pueden ejecutarse sin modificaciones en cualquier sistema que tenga instalado una máquina virtual Java.

Esta aplicación utiliza JDBC puede comunicarse con muchas bases de datos sin realizar ninguna modificación. Al igual que ODBC, JDBC no impone un lenguaje de comando común: puede usar la sintaxis específica de Oracle cuando está conectado a un servidor Oracle y la sintaxis específica de MySQL cuando está conectado a un servidor MySQL.

La clase **JDBC DriverManager** es responsable de localizar un controlador JDBC que necesita la aplicación. Cuando una aplicación cliente solicita una conexión de base de datos, la solicitud se expresa en forma de una URL, por ejemplo: `jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE`

El proceso de interacción es el siguiente:

1. Se carga el controlador en una Máquina Virtual de Java (VM).
2. Se registra con el DriverManager JDBC.
3. Aplicación solicita una conexión
4. El DriverManager pregunta a cada Controlador si puede realizar la conexión en la URL dada.
5. Se busca el controlador adecuado y se intenta establecer la conexión

6. Si falla, el Controlador lanzará una SQLException a la aplicación.
7. Si la conexión se completa, el controlador crea un objeto de conexión y lo devuelve a la aplicación.

Protocolos de acceso a bases de datos

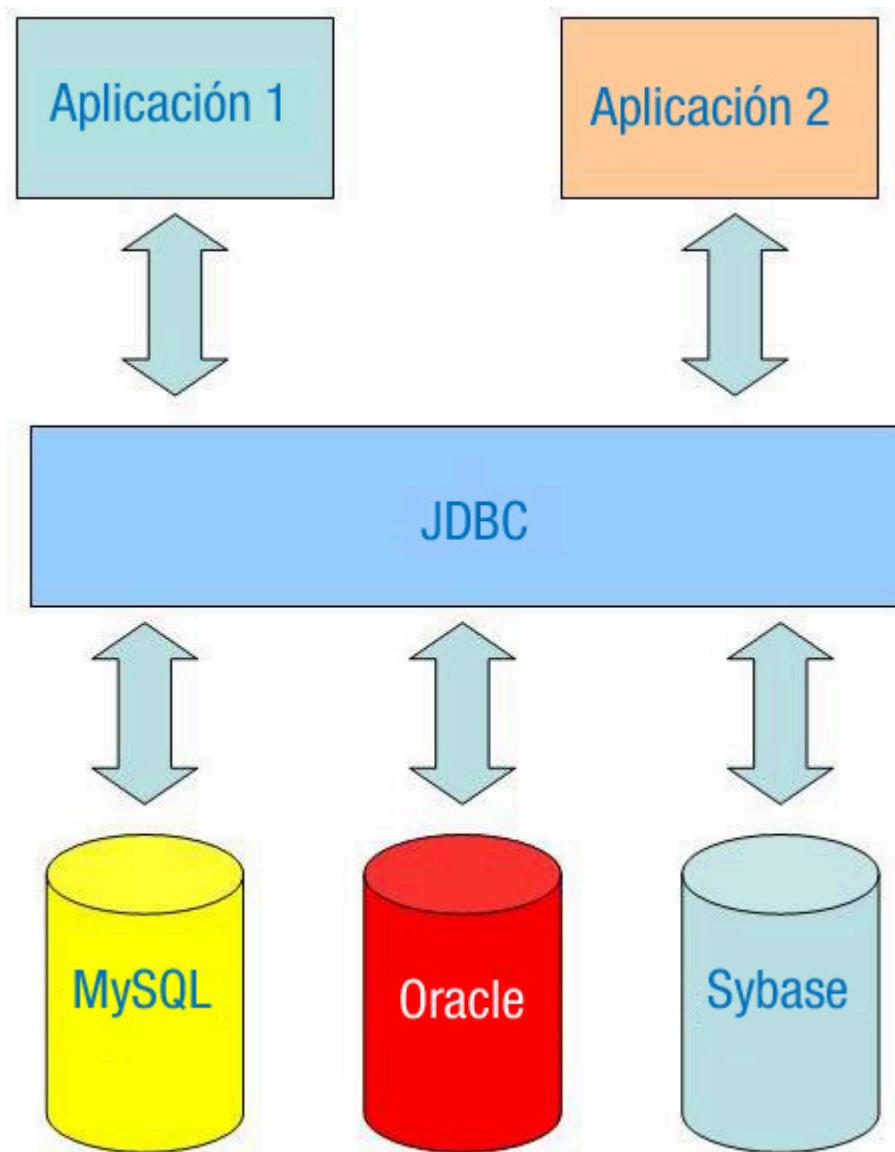


DEFINICIÓN

Un **conector o driver** es un conjunto de clases encargadas de implementar los interfaces del API y acceder a la base de datos. Gracias a un conector un programa es capaz de conectarse a una base de datos y lanzar consultas.

Un conector suele ser un fichero **.jar** que contiene una implementación de todas las interfaces del API JDBC.

Cuando se construye una aplicación de base de datos, JDBC oculta lo específico de cada base de datos, de modo que el programador se ocupe sólo de su aplicación.



El código de nuestra aplicación no dependerá del driver, puesto que trabajamos mediante los paquetes **java.sql** y **javax.sql**.

JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.

EJEMPLO

Crea un programa en Java que se conecte a una base de datos Oracle y ejecute una consulta sobre ella.

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

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

public class ConsultaSencilla {
    public static void main(String[] args) {
        Connection conexion;
        try {
            String urljdbc = "jdbc:oracle:thin:@localhost:1521:XE";
            //la base de datos esta en local y el servicio Oracle es XE
            se utiliza thin driver
            conexion = DriverManager.getConnection(urljdbc, "ejemplo",
            "ejemplo");
            Statement smt = conexion.createStatement();
            ResultSet rset = smt.executeQuery("select empno, ename, job
            from emp order by 1");
            while (rset.next())
                System.out.println("empleado numero " +
            rset.getString(1) + " nombre " + rset.getString(2) + " oficio " +
            rset.getString(3));
            conexion.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

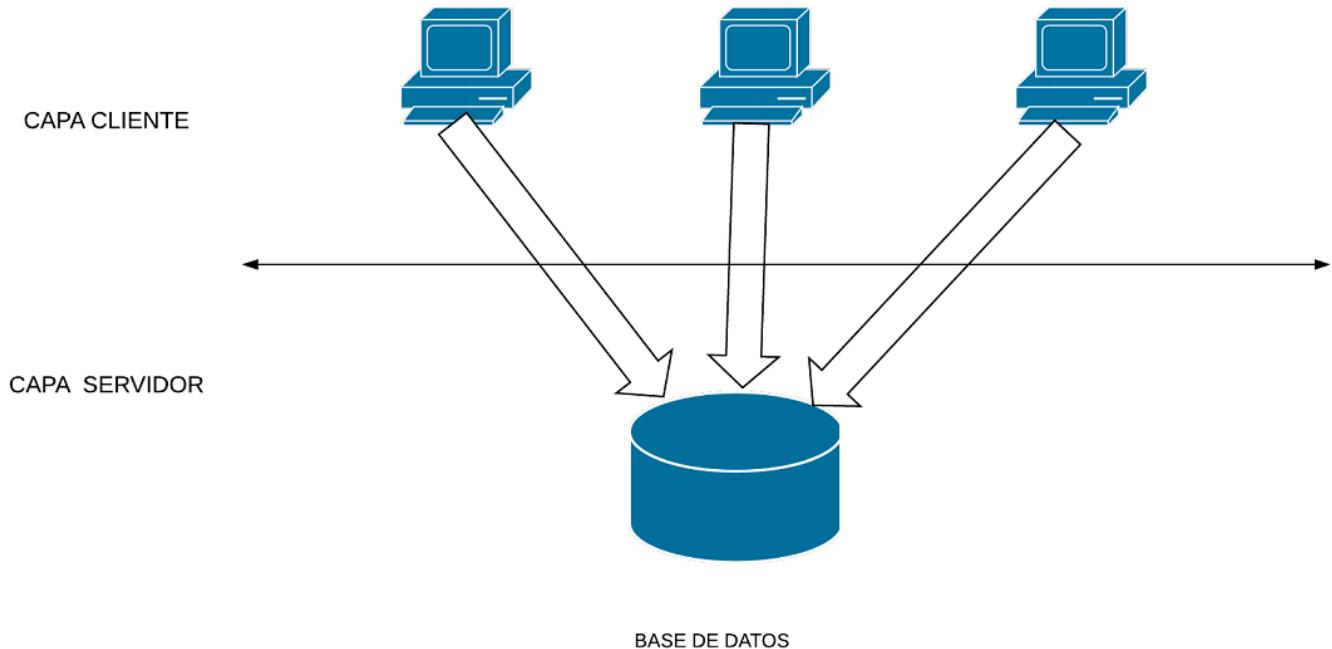
Introducción

Modelo de dos capas

En el **modelo de dos capas**, una aplicación se comunica directamente a la fuente de datos. Esto necesita un **conector JDBC** que pueda comunicarla con la fuente de datos específica a la que acceder.

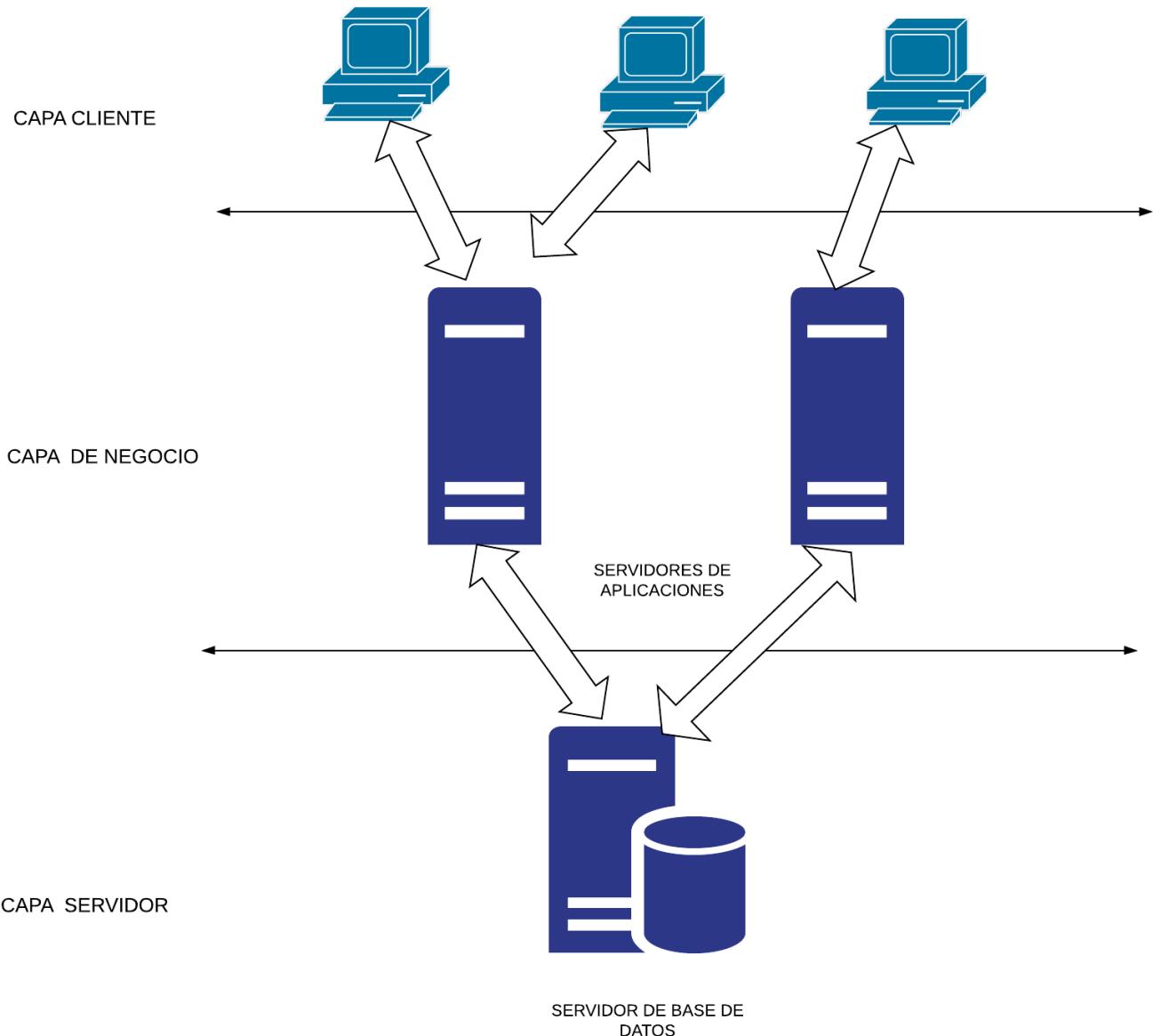
Las principales características de este modelo son:

- Los comandos o instrucciones del usuario se envían a la BD y los resultados se devuelven al usuario.
- Sigue una **estructura cliente/servidor**: La fuente de datos (**servidor**) puede estar ubicada en otra máquina a la que el usuario (**cliente**) se conecte por red.



Modelo de tres capas

En el **modelo de tres capas**, los comandos se envían a una capa intermedia de servicios, la cual envía los comandos a la fuente de datos. La fuente de datos procesa los comandos y envía los resultados de vuelta la capa intermedia, desde la que luego se le envían al usuario.



Tipos de controladores

Hoy en día, hay **cuatro** tipos de controladores JDBC en uso:

- Tipo 1: puente JDBC-ODBC
- Tipo 2: controlador parcial de Java
- Tipo 3: controlador puro para middleware de base de datos
- Tipo 4: controlador puro para directo a base de datos

En los siguientes apartados hablaremos de cada uno de estos tipos de controladores en más detalle.

Conectores tipo 1 y tipo 2

Tipo 1: puente JDBC-ODBC.

Los **conectores tipo 1** se denominan también **JDBC-ODBC Bridge (puente JDBC-ODBC)**.

El driver JDBC-ODBC Bridge traduce las llamadas JDBC a llamadas ODBC y las envía a la fuente de datos ODBC, de este modo el conector funciona a modo de puente entre JDBC y ODBC.

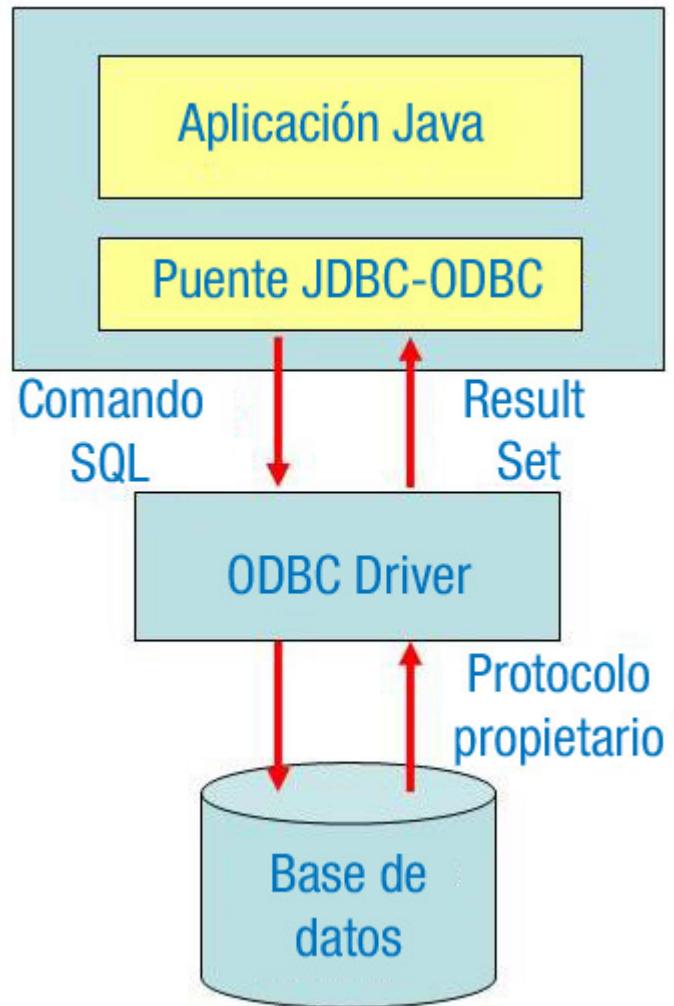
Como ventajas de este conector destacamos:

- No se necesita un driver específico de cada BD de tipo ODBC.
- Está soportado por muchos fabricantes, por lo que tenemos acceso a muchas BD.

Como desventajas podemos señalar:

- Hay plataformas que no lo implementan.
- El rendimiento no es óptimo ya que la llamada JDBC se realiza a través del puente hasta el conector ODBC y de ahí al interface de conectividad de la base de datos. El resultado recorre el camino inverso.
- Se tiene que registrar manualmente en el gestor de ODBC teniendo que configurar el DSN.

Este tipo de driver va incluido en el JDK.



Tipo 2: controlador parcial de Java (API nativa).

Los **conectores tipo 2** se conocen también como: **API nativa**

Convierten llamadas JDBC a llamadas específicas de la base de datos como SQL Server, Informix, Oracle, o Sybase.

El conector tipo 2 se comunica directamente con el servidor de bases de datos, por lo que es necesario que haya código en la máquina cliente.

Como ventaja:

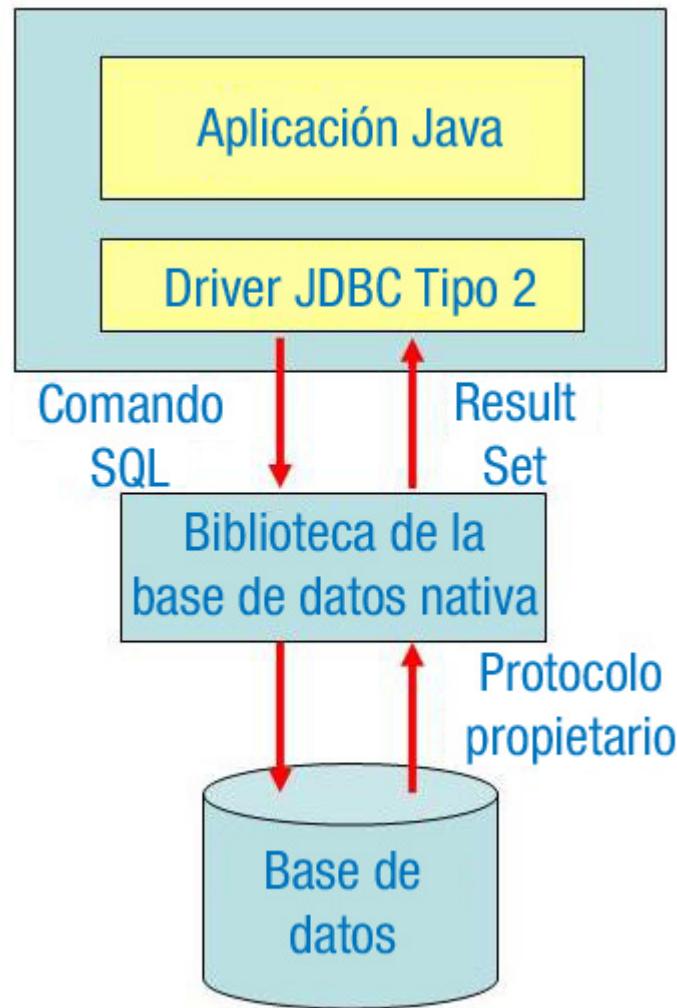
- Ofrecer un rendimiento mejor que el JDBC-ODBC Bridge.

Como inconveniente:

- La librería de la BD del vendedor necesita cargarse en cada máquina cliente.

- Por esta razón los drivers tipo 2 no pueden usarse para Internet.

Los drivers Tipo 1 y 2 utilizan código nativo vía JNI, por lo que son más eficientes.



SABER MÁS

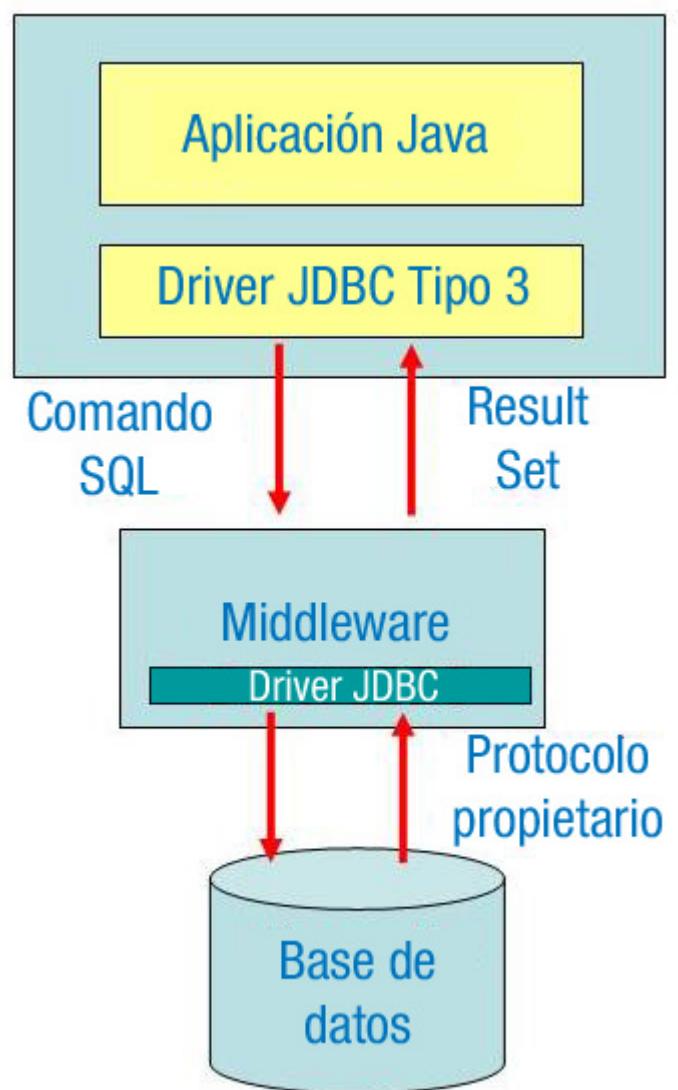
Si tienes interés por saber como funciona JNI y cual es su propósito, en este [enlace](#) puedes obtener más información

Conectores tipo 3 y tipo 4

Tipo 3: controlador puro para middleware (JDBC-Net pure Java driver).

Tiene una aproximación de **tres capas**. Las peticiones JDBC se pasan a través de la red al servidor de la capa intermedia (**middleware**). Este servidor traduce este protocolo a protocolo específico del sistema gestor y se envía a la base de datos. Los resultados se mandan de vuelta al middleware y se enrutan al cliente.

Es útil para aplicaciones en Internet.



Ventaja:

- Está basado en servidor.

- No necesita ninguna librería de base de datos en las máquinas clientes.
- Proporciona:
 - soporte para balanceo de carga
 - funciones de administrador, etc.

Tipo 4: controlador puro directo (Protocolo nativo).

Son conectores que convierten directamente las llamadas JDBC al protocolo de red usando por el sistema gestor de la base de datos.

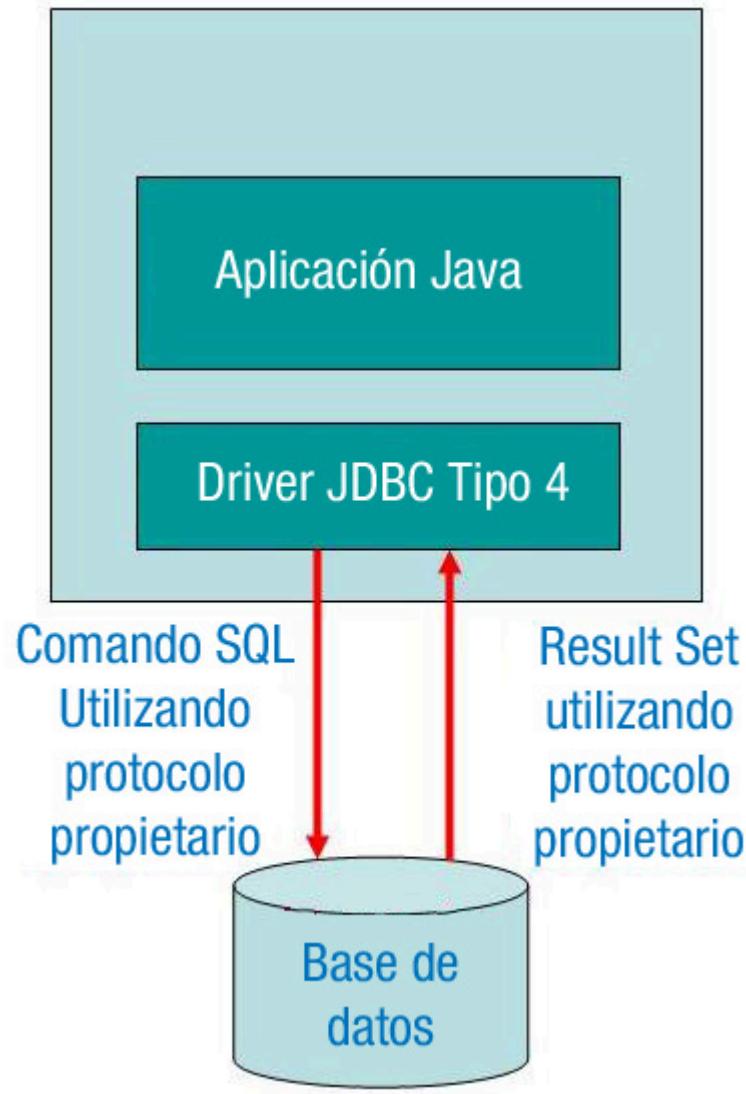
Esto permite una llamada directa desde la máquina cliente al servidor del sistema gestor de base de datos y es una solución excelente para acceso en intranets.

Como ventaja se tiene que no es necesaria traducción adicional o capa middleware, lo que mejora el rendimiento, siendo éste mejor que en el caso de los tipos 1 y 2.

Además, no se necesita instalar ningún software especial en el cliente o en el servidor.

Como inconveniente, de este tipo de conectores, el usuario necesita un driver diferente para cada base de datos.

Un ejemplo de este tipo de conector es Oracle Thin.



Introducción

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

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

La ejecución de este método devuelve un objeto `Connection` que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, `DriverManager` itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún conector adecuado, se lanza una `SQLException`.

Veamos un ejemplo con comentarios, para conectarnos a una base de datos MySQL:

```
public static void main (String[] args){
    try{

        //Cadena de conexión para conectar con MySQL en localhost, seleccionar
        la
        // base de datos llamada `test` con usuario y contraseña del servidor
        de
        // MySQL: root y admin
        String connectionUrl = "jdbc:mysql://localhost/test?
        user=root&password=admin";

        //Obtener la conexión
        Connection con = DriverManager.getConnection(connectionUrl);

    }catch(SQLException e){
        System.out.println("SQL Exception: " + e.toString());
    }catch(ClassNotFoundException cE){
        System.out.println("Exception: " + cE.toString());
    }
}
```

Creación de la base de datos

Una base de datos puede crearse utilizando las herramientas proporcionadas por el fabricante de la base de datos, o por medio de sentencias SQL desde un programa Java.

Normalmente es el administrador, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos.

No todos los conectores JDBC soportan la creación de la base de datos mediante el **lenguaje de definición de datos (DDL)**. Es decir, la sentencia `CREATE DATABASE` no es parte del estándar SQL, sino que es dependiente del sistema gestor de la base de datos.

Mediante JDBC podemos:

- Conectarnos
- Manipular:
 - Crear tablas
 - Modificarlas
 - Borrarlas
 - Añadir datos
 - etc.

No obstante, la creación en sí de la base de datos la hacemos con la herramienta específica para ello.

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos, sus tablas, claves, y todo lo necesario.

También, como en el caso de MySQL, o de Oracle, y la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la línea de comandos de MySQL o de Oracle, con las sentencias SQL apropiadas.



PARA RECORDAR

Si tienes algo olvidado SQL en este [enlace](#) tienes un tutorial sobre él

Drivers de conexión

Drivers de MySQL

MySQL proporciona controladores basados en estándares para JDBC, ODBC y .Net, lo que permite a los desarrolladores crear aplicaciones en el idioma de su elección.

DIVERS DE MYSQL

Puedes descargar estos controladores para MySQL en el siguiente [enlace](#).

Respecto a la implementación de JDBC, MySQL proporciona conectividad para aplicaciones cliente desarrolladas en el lenguaje de programación Java con MySQL Connector/J. La API de Java Database Connectivity (JDBC), así como una serie de extensiones de valor añadido al driver. También soporta el nuevo X DevAPI.

MySQL Connector/J es un controlador JDBC Tipo 4. Existen diferentes versiones disponibles que son compatibles con las especificaciones JDBC 3.0 y JDBC 4.2. El controlador es una implementación Java pura del protocolo MySQL y no se basa en las bibliotecas cliente de MySQL.

Para los programas a gran escala que usan patrones de diseño comunes de acceso a datos, podemos usar:

- Marcos de persistencia como Hibernate
- Plantillas JDBC de Spring
- Mapeo de sentencias SQL de MyBatis

Lo que permiten reducir la cantidad de código JDBC para que pueda depurar, ajustar, proteger y mantener.

Instalar el conector de la base de datos

A continuación, explicaremos como instalar los dos tipos de conectores de bases de datos:

MySQL.

Entre nuestra aplicación Java y el Sistema Gestor de Base de Datos (SGBD), se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

La función del conector es traducir los comandos del API JDBC al protocolo nativo del SGBD.



PASOS PARA CONECTAR JAVA A UNA BD

Para poder usar el conector en cualquier IDE es necesario buscar el .jar correspondiente y descargarlo. Para descargarlo podéis usar el siguiente [enlace de Maven Repository](#). En él podéis descargar el fichero .jar.

Acto seguido, sería necesario importar la librería al IDE. El procedimiento de importación es diferente en cada IDE por lo que tendréis que saber cómo hacerlo en el que useis.



COMPROBACIÓN

Es importante que compruebes el driver mas adecuado para trabajar con tu Servidor MySQL.



PARA AMPLIAR

En el siguiente [vídeo](#) encontraréis una visión gráfica de lo explicado hasta el momento

Nota: En sistemas antiguos, para que `DriverManager` tuviera "registrados" los drivers, era necesario cargar la clase en la máquina virtual. Para eso es el `Class.forName()`, simplemente carga la clase con el nombre indicado.

A partir de JDK 6, los drivers JDBC 4 ya se registran automáticamente y no es necesario el `Class.forName()`, sólo que estén en el classpath de la JVM.

EJEMPLO DE CONEXIÓN

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

public class PrimeraConexion {
    public static void main(String[] args) {
        try {
            // Establecemos la conexión con la BD
            Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "root",
"system");
            System.out.println("conecta");
            conexion.close(); // Cerrar conexión
        } catch (SQLException cn) {
            cn.printStackTrace();
        }
    }
}
```

JDBC conexión a una base de datos

La API JDBC, aparte de algunas clases específicas, mayoritariamente está compuesto de interfaces que el controlador implementa dándoles la funcionalidad adecuada.

Para asegurar la **interoperabilidad**, las aplicaciones **no referenciarán** nunca las **clases** concretas de ningún controlador **sino** las **interfaces** estándares de la API JDBC. Para ello, **la aplicación nunca podrá instanciar directamente los objetos** JDBC con una sentencia new, sino que **se crearán** indirectamente **llamando** algún **método** de alguna clase u objeto ya existente.

La interfaz `Connection` representa una conexión a la base de datos, una vía de comunicación entre la aplicación y el SGBD. Los objetos `Connection` mantendrán la capacidad de comunicarse con el sistema gestor mientras permanezcan abiertos. Esto es, desde que se crean hasta que se cierran utilizando el método `close`.

El objeto `Connection` está totalmente vinculado a una fuente de datos, por eso hay que especificar de qué fuente al pedir la conexión, indicando la url de los datos, el usuario y contraseña.

El objeto `Connection` se obtiene utilizando el método `getConnection()` de la clase `DriverManager`. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión:

- `getConnection(String url)`
- `getConnection(String url, Properties info)`
- `getConnection(String URL, String user, String password)`

Cuando se presenta con una URL específica, `DriverManager` itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún conector adecuado, se lanza una `SQLException`.

La URL JDBC más sencilla está formada por tres valores separados mediante dos puntos:

- El **primer** valor: representa el protocolo, que es siempre jdbc para los URL JDBC.
- El **segundo** valor: representa el subprotocolo.
- El **tercer** valor: representa el nombre de sistema para establecer la conexión, dependerá de:

- el tipo de controlador utilizado
- el host donde se aloje el SGBD
- el puerto que este use para escuchar las peticiones
- el nombre de la base de datos o esquema con el que se trabajará.

CONEXIÓN A LOCAL

Existen dos valores especiales que pueden utilizarse para conectarse a una base de datos local. Estos valores son: ***LOCAL** y **localhost** (ambos son sensibles a mayúsculas y minúsculas).

EJEMPLO CONEXIÓN BASE DE DATOS MYSQL

Para establecer la conexión con la BD MySQL, únicamente hay que utilizar el driver JDBC correspondiente y configurar el proyecto para añadir la librería mysql-connector-java-5.1.47-bin.jar.

[En este enlace](#) puedes obtener mas información de los drivers de MySQL disponibles y de la sintaxis de URL correspondiente a cada uno de ellos.

La URL para una BD local es "jdbc:mysql://localhost/base datos".

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

public class PrimeraConexion {

    public static void main(String[] args) throws SQLException {
        // Establecemos la conexión con la BD
        Connection conexion =
        DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "root",
        "system");
        // Cerrar conexión
        conexion.close();
    }
}
```

Realización de consultas

Para operar con una base de datos y ejecutar consultas, nuestra aplicación deberá:

1. Cargar el driver de la BD a utilizar.
2. Establecer una conexión con la BD.
3. Enviar consultas SQL y procesar el resultado.
4. Liberar los recursos al terminar.
5. Gestionar los errores que se puedan producir.

Para hacer consultas en la base de datos tendremos que crear un objeto de tipo `Statement`, ejecutar la consulta con el método `executeQuery(sql)` y recorrer el `ResultSet` que devuelve este método.

A continuación, mostraremos un par de ejemplos donde se realicen y procesen distintas consultas



EJEMPLO 1

Realice un programa Java que se conecte a una base de datos MySQL llamada `ejemplo` mediante el usuario `root` y la contraseña `system`. Una vez conectado se tendrán que listar todos los datos de la tabla departamentos `dept`

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

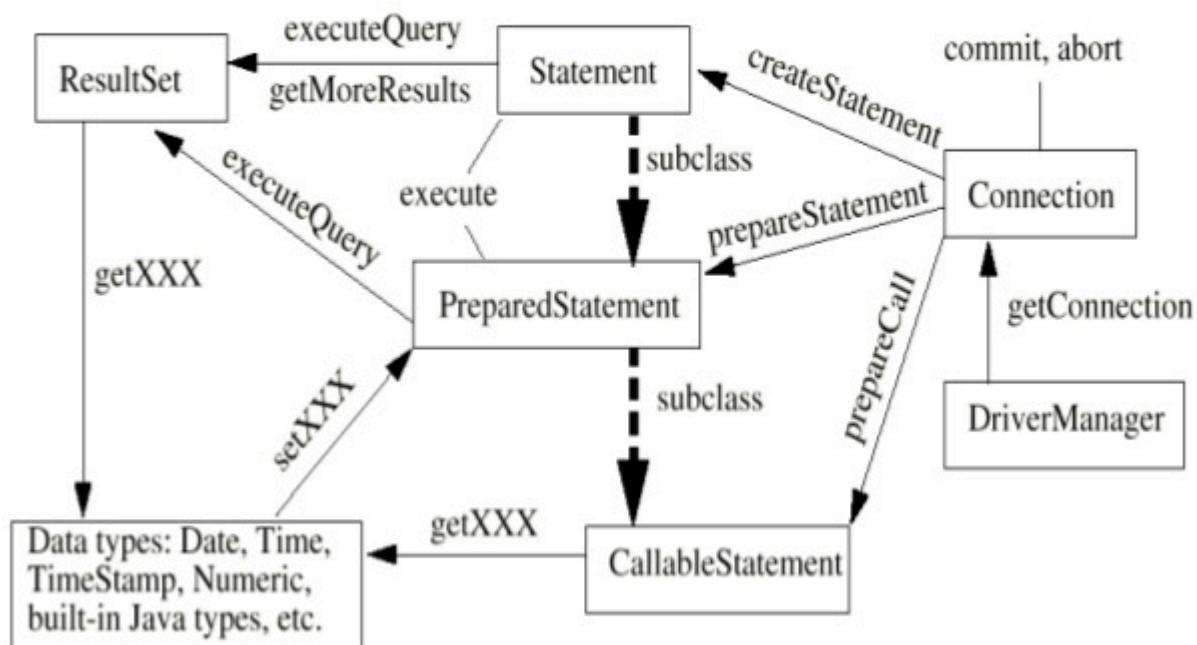
public class PrimeraConsulta {
    public static void main(String[] args) {
        try {
            // Establecemos la conexión con la BD
            Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "root",
"system");
            // creamos el objeto Statement
            Statement sentencia = conexion.createStatement();
            // ejecutamos la consulta
            String sql = "SELECT * FROM dept";
            ResultSet result = sentencia.executeQuery(sql);
```

```
// Recorremos el resultado para visualizar cada fila      //
Se hace un bucle mientras haya registros y se van visualizando
while (result.next()) {
    System.out.printf("%d, %s, %s %n", result.getInt(1),
result.getString(2), result.getString(3));
}
result.close(); // Cerrar ResultSet
sentencia.close(); // Cerrar Statement
conexion.close(); // Cerrar conexión
} catch (SQLException cn) {
    cn.printStackTrace();
}
}

}
```

JDBC Class Diagram

SunilOS



El API JDBC distingue dos tipos de sentencias SQL:

- **Consultas:** SELECT devuelven de ninguna o varias filas.
- **Actualizaciones:** INSERT, UPDATE, DELETE, sentencias DDL.

Veamos en primer lugar consultas que devuelven datos (SELECT)

Los datos se devuelven utilizando un objeto `ResultSet` el cual se iterará mediante un cursor. Este cursor es un puntero que apunta a una fila de datos del objeto `ResultSet`. Inicialmente, el cursor se coloca antes de la primera fila y irá avanzando a las siguientes filas al utilizar el método `next`.

A continuación, veamos como ejecutar sentencias DML (UPDATE, DELETE, INSERT)

Pese a que se trata de sentencias muy dispares, desde el punto de vista de la comunicación con el SGBD se comportan de manera muy similar, siguiendo el siguiente patrón:

1. Instanciamos un objeto `Statement` a partir de una conexión activa.
2. Ejecutamos la sentencia SQL, método `executeUpdate`.
3. Cerramos del objeto `Statement` instanciado.
4. Cerramos la conexión activa.



MÉTODO EXECUTEUPDATE

El método `executeUpdate` devuelve un número entero que representa el número de filas afectadas por la instrucción SQL.



EJEMPLO 3

Realice un programa Java que se conecte a una base de datos MySQL llamada `ejemplo` mediante el usuario `scott` y la contraseña `tiger`. Una vez conectado se tendrán que insertar una línea y luego borrar todas las filas de la tabla `jobs`

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

public class Actualizacion {

    public static void main(String[] args) {
        // establecemos la conexión
        Connection conexion;
        try {

            String urljdbc = "jdbc:mysql://localhost/ejemplo";
            conexion = DriverManager.getConnection(urljdbc, "scott",
                "tiger");
        }
    }
}
```

```
// crea la sentencia
Statement stm = conexion.createStatement();

    // ejecuta la actualizacion la ejecucion devuelve 1, numero
    de filas afectadas
    System.out.println(stm.executeUpdate("insert into jobs values
('ID TEACH', 'PROFESOR IT', 2500, 5000)"));

    // valida los datos
    // ejecuta el borrado, la ejecucion devuelve 19, numero de
filas afectadas
    System.out.println(stm.executeUpdate("delete from jobs"));
    stm.close(); // Cerrar Statement
    conexion.close(); // cierra la conexion

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

}
```

Ejecución de sentencias DML



SENTENCIAS DML

Las sentencias DML (Lenguaje de manipulación de datos), incluyen fundamentalmente tres tipos de operaciones inserciones (**INSERT**), borrados (**DELETE**) y modificaciones (**UPDATE**).

El funcionamiento de todas ellas en la conexión es bastante similar. A continuación se muestra un ejemplo de borrado en una conexión MySQL:

```
import java.sql.*;
public class DMLDeletePrep {
    public static void main(String[] args) {
        try{

            Connection conexion = DriverManager.getConnection
("jdbc:mysql://localhost/ejemplo","scott", "tiger");
            String deptno=args[0];
            String sql= "DELETE FROM dept WHERE DEPTNO=?";
            System.out.println(sql);
            PreparedStatement sentencia = conexion.prepareStatement(sql);
            sentencia.setInt(1,Integer.parseInt(deptno));
            int filas = sentencia.executeUpdate();
            System.out.println("Filas afectadas: " + filas);
            sentencia.close();
            //Cerrar conexión
            conexion.close();

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

Ejecución de procedimientos almacenados



PROCEDIMIENTO ALMACENADO

Un **procedimiento almacenado** es un procedimiento o subprograma que está almacenado en la base de datos.

Muchos sistemas gestores de bases de datos los soportan, por ejemplo: MySQL, Oracle, etc.

Podemos diferenciar dos clases de procedimientos:

1. **Procedimientos almacenados:** no devuelven un valor
2. **Funciones:** las cuales devuelven un valor que se puede emplear en otras sentencias SQL.

Un procedimiento almacenado típico tiene:

- Un nombre.
- Una lista de parámetros.
- Unas sentencias SQL.

Un ejemplo de sentencia para crear un procedimiento almacenado sencillo para MySQL, aunque sería similar en otros sistemas gestores:

```
CREATE PROCEDURE procediprueba (IN par1 INTEGER)      /* Nombre del
procedimiento y parámetro */
BEGIN
    /* Inicio del bloque */
    /* Declaración de
variables */
    DECLARE var1 CHAR(13);
    IF par1 = 25 THEN
        /* Inicio del IF */
        SET var1 = 'perro rabioso';
        /* Asignar valor a la
variable */
    ELSE
        SET var1 = 'gato persa';
        /* Asignar valor a la
variable */
    END IF;
    /* Fin del IF */
    /* Insertar registro */
    INSERT INTO Animales VALUES (var1);
    /* Fin del procedimiento
*/
END
*/
```

Como se ve en los comentarios, este procedimiento admite un parámetro, llamado `par1`.

También se declara una variable a la que llamamos `var1` y es de tipo carácter y longitud 13. Si el valor que le llega de parámetro es igual a 24, entonces se asigna a la variable `var1`, la cadena `perro rabioso` y en caso contrario se le asignará la cadena: `gato persa`. Finalmente, se inserta en la tabla `Animales` el valor que se asignó a la variable `var1`.

Procedimientos almacenados en MySQL

A continuación, vamos a realizar un procedimiento almacenado en MySQL, que insertará datos en la tabla clientes. Desde el programa Java que realizamos, llamaremos para ejecutar a ese procedimiento almacenado.

La secuencia que seguiremos para realizar esto será:

1. Si no tenemos creada la tabla de clientes, crearla.

```
CREATE TABLE `clientes` (
  `Cod_Cliente` int(3) NOT NULL DEFAULT '0',
  `Nombre` tinytext,
  `Telefono` tinytext,
  PRIMARY KEY (`Cod_Cliente`)
)
```

2. Creamos el procedimiento almacenado en la base de datos.

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `insertaCliente` (IN Cod_Cliente
INTEGER, IN Nombre TinyText,
IN Telefono TinyText)
BEGIN
  INSERT INTO clientes VALUES (Cod_Cliente, Nombre, Telefono);
END
```

3. Crear la clase Java para desde aquí, llamar al procedimiento almacenado:

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

public class inserConProcAlma {
```

```
public static void main(String[] args) {
    try {
        // Cadena de conexión para conectar con MySQL en localhost,
        // seleccionar la base
        // de datos llamada 'test' con usuario y contraseña del servidor
        // de MySQL: root y admin
        String connectionUrl = "jdbc:mysql://localhost/test?
user=root&password=admin";
        // Obtener la conexión
        Connection con = DriverManager.getConnection(connectionUrl);

        // El procedimiento almacenado tendrá tres parámetros
        CallableStatement prcProcedimientoAlmacenado = con.prepareCall("{"
call insertaCliente(?, ?,?) }");

        // cargar parametros en el procedimiento almacenado
        prcProcedimientoAlmacenado.setInt("Cod_Cliente", 765);
        prcProcedimientoAlmacenado.setString("Nombre", "Antonio Pérez");
        prcProcedimientoAlmacenado.setString("Telefono", "950121314");

        // ejecutar el procedimiento
        prcProcedimientoAlmacenado.execute();

    } catch (SQLException e) {
        System.out.println("SQL Exception: "+ e.toString());
    } catch (ClassNotFoundException cE) {
        System.out.println("Excepción: "+ cE.toString());
    }
}
```

CLAVES PRIMARIAS

Si hemos definido la tabla correctamente, con su clave primaria, y ejecutamos el programa, intentando insertar una fila con la misma clave primaria que otra fila, obtendremos un mensaje al capturar la excepción de este tipo:

```
SQL Exception:
com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:
Duplicate entry '765' for key 'PRIMARY'
```

MÁS EJEMPLOS

Si quieres ver más ejemplos JDBC para diversas finalidades: listar datos de una BD, llamar a funciones en Oracle, etc., puedes consultar el siguiente [enlace](#)

Sentencias preparadas

A veces es útil usar un objeto `PreparedStatement` para enviar sentencias de SQL. Este tipo especial de declaración se deriva de la clase más general `Statement`.



PREPARED STATEMENT

Una **Prepared Statement** es una sentencia SQL precompilada.

Al estar precompilada, su ejecución será más rápida que una SQL normal, por lo que es adecuada cuando vamos a ejecutar la misma sentencia SQL (con distintos valores) muchas veces.

Cómo utilizarlo

1. Se utilizarán los interrogantes para indicar dónde van a ir los valores a insertar.
2. Cada interrogante se identifica luego con un número, de forma que el primer interrogante que aparece es el 1, el segundo el 2, etc.
3. Utilizamos los métodos `set()` para llenar los valores. Donde el primer parámetro es el número del interrogante y el segundo el valor que queremos insertar.
4. Llamamos al método `executeUpdate()`.
5. Si queremos volver a ejecutarla con otros valores utilizaremos los métodos `set()` para ir modificando los parámetros afectados.



INFORMACIÓN SOBRE LOS PARÁMETROS

Fíjate que a la hora de poner los interrogantes, **no** nos hemos preocupado de poner **comillas en los valores de texto ni conversiones de ningún tipo**.

Las llamadas a `setInt()`, `setString()`, ya hacen todas las conversiones adecuadas para nosotros.

El siguiente ejemplo utiliza un único objeto `PreparedExpresion`, dándole valores a los parámetros:

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

public class actualizaSentenciaPreparada {
    public static void main(String[] args) throws SQLException {

        String urljdbc = "jdbc:oracle:thin:@localhost:1521:XE";
        Connection conexion = DriverManager.getConnection(urljdbc, "scott",
        "tiger");
        PreparedStatement pstmt = conexion.prepareStatement("UPDATE EMP SET
SAL = ? WHERE empno = ?");
        pstmt.setInt(1, 1);
        pstmt.setInt(2, 7844 );
        pstmt.executeUpdate();
        pstmt.close();
        conexion.close();
    }
}
```

Los métodos que permiten pasar parámetros a la sentencia empiezan por `set`:

- `setBigDecimal(int parameterIndex, BigDecimal x)`
- `setBinaryStream(int parameterIndex, InputStream x)`
- `setDate(int parameterIndex, Date x)`
- `setDouble(int parameterIndex, double x)`
- `setInt(int parameterIndex, int x)`
- `setString(int parameterIndex, String x)`

Captura de errores y liberación de recursos

Liberación de recursos

Debido a que las conexiones con una BD consumen muchos recursos en el sistema gestor y en el sistema informático, conviene cerrarlas con el método `close` (de la clase `Connection`) siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar la ejecución de procedimientos almacenados `CallableStatement`, las consultas (`Statement` y `PreparedStatement`) y los resultados (`ResultSet`) para liberar los recursos.

Gestión de errores.

El manejo de excepciones permite manejar condiciones excepcionales tales como errores definidos por el programa de una manera controlada.

Cuando se produce una condición de excepción, se lanza una excepción. La ejecución del programa actual se detiene y el control se redirige a la cláusula de captura (catch) más cercana. Si no existe, la ejecución del programa finaliza.

El manejo de excepciones de JDBC es muy similar al manejo de excepciones de Java, pero para JDBC, la excepción más común con la que tratará es `java.sql.SQLException`.

Se puede producir una `SQLException` tanto en el controlador como en la base de datos. Cuando se produce una excepción de este tipo, un objeto de tipo `SQLException` se pasará a la cláusula catch.

Transacciones



TRANSACCIONES

Una transacción permite que un conjunto de consultas SQL se ejecuten en su totalidad, impidiendo dejar la BD en un estado inconsistente al ejecutar algunas consultas y otras no.

Las transacciones tienen la característica de poder **deshacer** los cambios efectuados en las tablas, si no se han podido realizar todas las operaciones que forman parte de dicha transacción.

Por eso, las bases de datos que soportan transacciones son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor que almacena la base de datos, ya que las consultas se ejecutan o no en su totalidad.

Al ejecutar una transacción, el motor de base de datos garantiza: **atomicidad, consistencia, aislamiento y durabilidad** (ACID) de la transacción (o conjunto de comandos) que se utilice.



EJEMPLO TÍPICO

El ejemplo típico de una transacción es de una transacción bancaria. Si una cantidad de dinero es transferida de la cuenta de Antonio a la cuenta de Pedro, se necesitarían dos consultas:

1. En la cuenta de Antonio para quitar de su cuenta ese dinero:

```
UPDATE cuentas SET saldo = saldo - cantidad WHERE cliente = "Antonio";
```

2. En la cuenta de Pedro para añadir ese dinero a su cuenta:

```
UPDATE cuentas SET saldo = saldo + cantidad WHERE cliente = "Pedro";
```

Pero, ¿qué ocurre si por algún imprevisto (un apagón de luz, etc.), el sistema se "cae" después de que se ejecute la primera consulta? Antonio tendrá una cantidad de dinero menos en su cuenta y creerá que ha realizado la transferencia. Pedro, sin embargo, creerá que todavía no le han realizado la transferencia.

El control de la transacción es realizado por el objeto de la conexión. Cuando se crea una conexión, por defecto, está en el modo activado. Esto significa que cada operación DML (INSERT,

UPDATE, DELETE) es tratada como transacción por sí misma que se valida automáticamente en cuanto se ejecute.

A veces necesitamos agrupar varias sentencias SQL en una única transacción, para ello:

1. Modificamos el autocommit antes de ejecutar las consultas que deban estar en la misma transaccion:

```
conexion.setAutoCommit(false)
```

2. Ejecutamos las sentencias.

3. Finalizaremos de forma manual la transacción.

```
conexion.commit()
```

EJEMPLO

El siguiente programa, se actualiza la tabla CAFFEE con los datos de las ventas que se envían al programa como un objeto de tipo HashMap (colección que almacena datos asociando una clave a un valor), las actualizaciones se tratan como una única transacción de forma que si se ha podido completar todas las actualizaciones, se validan y en caso de que falle alguna, el gestor de errores deshace los cambios (con.rollback())

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
throws SQLException {
    PreparedStatement updateSales = null;
    PreparedStatement updateTotal = null;
    String updateString = "update " + dbName + ".COFFEES set SALES = ?
where COF_NAME = ?";
    String updateStatement = "update " + dbName + ".COFFEES set TOTAL =
TOTAL + ? where COF_NAME = ?";
    try {
        con.setAutoCommit(false);
        updateSales = con.prepareStatement(updateString);
        updateTotal = con.prepareStatement(updateStatement);
        for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
            updateSales.setInt(1, e.getValue().intValue());
            updateSales.setString(2, e.getKey());
            updateSales.executeUpdate();
            updateTotal.setInt(1, e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
        }
    } catch (SQLException ex) {
        con.rollback();
    }
}
```

```
        updateTotal.executeUpdate();
        con.commit();
    }
} catch (SQLException e) {
    JDBCUtilities.printSQLException(e);
    if (con != null) {
        try {
            System.err.print("Transaction is being rolled back");
            con.rollback();
        } catch(SQLException excep) {
            JDBCUtilities.printSQLException(excep);
        }
    }
} finally {
    if (updateSales != null) {
        updateSales.close();
    }
    if (updateTotal != null) {
        updateTotal.close();
    }
    con.setAutoCommit(true);
}
}
```

Commit y Rollback.

Una transacción tiene dos finales posibles:

- Si se finaliza correctamente y sin problemas se hará con **COMMIT**, con lo que los cambios se realizan en la base de datos.
- Si hay un fallo, se deshacen los cambios efectuados hasta ese momento, con la ejecución de **ROLLBACK**.



AUTOCOMMIT

Por defecto, las conexiones con MySQL y Oracle trabajan en modo autocommit con valor **true**. Eso significa que cada consulta es una transacción en la base de datos.

Si queremos definir una transacción de varias operaciones, estableceremos el modo autocommit a false con el método **setAutoCommit** de la clase Connection.

En modo **no autocommit** las transacciones quedan definidas por las ejecuciones de los métodos `commit` y `rollback`. Una transacción abarca desde el último commit o rollback hasta el siguiente commit. Los métodos commit o rollback forman parte de la clase Connection.

EJEMPLO

En la siguiente porción de código de un procedimiento almacenado, puedes ver un ejemplo sencillo de cómo se puede utilizar commit y rollback: tras las operaciones se realiza el commit, y si ocurre una excepción, al capturarla realizaríamos el rollback.

```
BEGIN
```

```
...
```

```
SET AUTOCOMMIT OFF
```

```
update cuenta set saldo=saldo + 250 where dni='12345678-L';
```

```
update cuenta set saldo=saldo - 250 where dni='89009999-L';
```

```
COMMIT;
```

```
...
```

```
EXCEPTION
```

```
WHEN OTHERS THEN
```

```
ROLLBACK ;
```

```
END;
```

Consultas al diccionario de datos

La Interfaz `DatabaseMetaData` (de la API `java.sql`) permite obtener información sobre la base de datos en su conjunto.

Algunos de sus métodos devuelven listas de información en forma de objetos `ResultSet`, que se pueden recorrer de forma repetitiva para obtener los metadatos. Para ello utilizaremos métodos `getString`, `getInt`, etc.

Otros métodos tienen argumentos que son patrones de cadena. Es posible usar patrones dentro de estas cadenas, "%" representa cualquier subcadena de 0 o más caracteres, y "_" representa un carácter.

Algunos métodos útiles son:

Métodos	Descripción
<code>getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)</code>	Devuelve en un <code>ResultSet</code> la descripción de las columnas de la tabla disponibles en el catálogo especificado.
<code>getDriverName()</code>	Devuelve en un <code>String</code> , el nombre del controlador JDBC.
<code>getDriverVersion()</code>	Devuelve en un <code>String</code> , el número de versión de este controlador JDBC como a <code>String</code> .
<code>getExportedKeys(String catalog, String schema, String table)</code>	Devuelve en un <code>ResultSet</code> la descripción de las columnas de clave Foranea que hacen referencia a las columnas de clave principal de la tabla dada.

Métodos	Descripción
<code>getFunctions(String catalog, String schemaPattern, String functionNamePattern)</code>	Devuelve en un ResultSet la descripción del sistema y las funciones de usuario disponibles en el catálogo dado.
<code>getMaxConnections()</code>	Recupera el número máximo de conexiones simultáneas a esta base de datos que son posibles.
<code>getPrimaryKeys(String catalog, String schema, String table)</code>	Devuelve en un ResultSet una descripción de las columnas de clave primaria de la tabla dada.
<code>getProcedures(String catalog, String schemaPattern, String procedureNamePattern)</code>	Recupera una descripción de los procedimientos almacenados disponibles en el catálogo dado.
<code>getSchemas()</code>	Devuelve en un ResultSet los nombres de esquema disponibles en esta base de datos.
<code>getSQLKeywords()</code>	Recupera una lista separada por comas de todas las palabras clave de SQL de esta base de datos que NO son también palabras clave de SQL: 2003.
<code>getStringFunctions()</code>	Recupera una lista separada por comas de funciones de cadena disponibles con esta base de datos.
<code>supportsAlterTableWithAddColumn()</code>	Recupera si esta base de datos admite ALTER TABLE con añadir columna. Devuelve un booleano
<code>supportsAlterTableWithDropColumn()</code>	Recupera si esta base de datos es posible borrar una columna con

Métodos	Descripción
	ALTER TABLE. Devuelve un booleano.
<code>supportsTransactions()</code>	Recupera si esta base de datos admite transacciones.

EJEMPLO

A continuación, se describe un ejemplo en Java con MySQL para la obtención de los metadatos

```
public class EjemploMetadata {
    public static void main(String[] args) {
        try {
            Connection conexion =
DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "ejemplo",
"ejemplo");
            java.sql.DatabaseMetaData dbmd = conexion.getMetaData();//
Creamos objeto DatabaseMetaData

            ResultSet resul = null;
            String nombre = dbmd.getDatabaseProductName();
            String driver = dbmd.getDriverName();
            String url = dbmd.getURL();
            String usuario = dbmd.getUserName();

            System.out.println("INFORMACIÓN SOBRE LA BASE DE DATOS:");
            System.out.printf("Nombre : %s %n", nombre);
            System.out.printf("Driver : %s %n", driver);
            System.out.printf("URL     : %s %n", url);
            System.out.printf("Usuario: %s %n", usuario);

            // Obtener información de las tablas y vistas que hay
            resul = dbmd.getTables(null, "ejemplo", null, null);
            while (resul.next()) {
                String catalogo = resul.getString(1); // columna 1
                String esquema = resul.getString(2); // columna 2
                String tabla = resul.getString(3); // columna 3
                String tipo = resul.getString(4); // columna 4
                System.out.printf("%s - Catalogo: %s, Esquema: %s,
Nombre: %s %n", tipo, catalogo, esquema, tabla);
            }
        }
    }
}
```

```
conexion.close(); // Cerrar conexion

} catch (SQLException e) {
    System.out.println(e.getMessage());
    System.out.println(e.getErrorCode());
    System.out.println(e.getSQLState());
}
}
```

Pool de conexiones

A continuación, vamos a estudiar como funciona la creación de conexiones en una aplicación clásica de escritorio cliente-servidor y luego veremos los problemas de seguir con dicha estructura en una aplicación web.

Problemas en la creación de conexiones

En una aplicación de escritorio cada usuario que la inicia mantiene una conexión exclusiva con la base de datos, la cual se crea al iniciar la aplicación y se cierra al finalizarla. Como es observable, sería imposible compartir dicha conexión ya que cada aplicación estará en un ordenador independiente.

En una aplicación Web podríamos seguir un esquema similar, en el que cada usuario nuevo que se conecta a nuestra aplicación se le crea una conexión y al salir se cierre. Esto, aunque parezca sencillo, presenta ciertos problemas debido a la diferente naturaleza de las aplicaciones de escritorio y las web.

Este mismo patrón de creación aplicado a las aplicaciones web conllevaría tener una cantidad enorme de conexiones activas (debido al gran número de usuarios) y con gran cantidad de conexiones abiertas sin usar (debido a usuarios que abandonan el portal y no lo hemos detectado) lo que podría provocar que el servidor de bases de datos se cayera debido a los recursos consumidos por todas las conexiones.

Podemos llegar a pensar que nuestro servidor puede aguantar ya que tendremos pocos usuarios pero no suele ser así debido a:

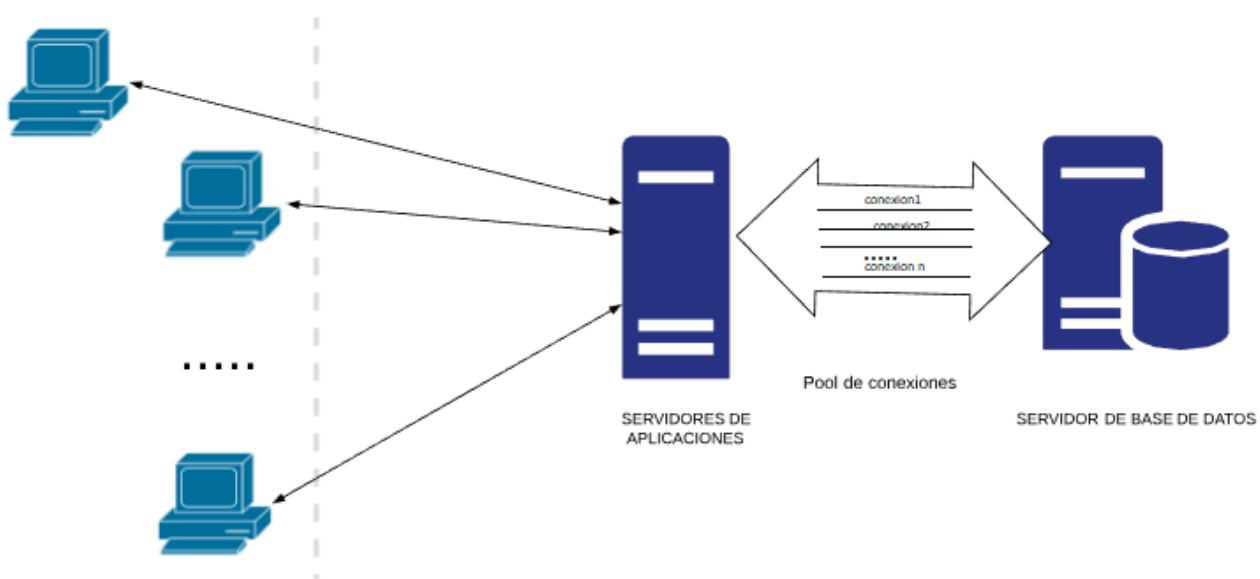
- Si no se cierran las conexiones aun con pocos usuarios es probable que acabemos saturando al servidor de conexiones sin usar.
- Las aplicaciones Web pueden tener picos de mucho tráfico, donde sería normal que se excediera la capacidad de nuestro servidor.
- Sería muy sencillo hacernos un ataque de denegación de servicio (DDS) y saturando el servidor haciendo que se crearan gran cantidad de conexiones que no se usen.

Una posible solución sería crear la conexión al iniciar cada petición web y cerrarla al finalizar. ¿El problema de eso? Crear y cerrar una conexión es muy costoso. Lo que tendríamos es una aplicación lentísima.

Pool de conexiones

El concepto de Pool de conexiones se ha estandarizado desde la versión 3.0 de JDBC. La solución del pool de conexiones tiene que solucionar los siguientes problemas:

- No tenemos tantas conexiones como usuarios (número de usuarios es demasiado elevado).
- Cerrar la conexión.



A partir de la imagen anterior, podemos definir el funcionamiento del **pool de conexiones**.

El servidor web tiene **n** conexiones ya creadas y conectadas a la base de datos (Se llaman **conexiones esperando**).

Cuando llegan **m** peticiones web, la aplicación pide **m** conexiones al pool de conexiones, quedando en estado de espera en el pool **n-m** conexiones. Esta operación es muy rápida ya que al estar creada la conexión solo hay que marcarla como "**en uso**".

Cuando las peticiones web finalizan, las **conexiones no se cierran** sino que **se devuelven al pool** indicándole que ya se han acabado de usar las conexiones. Esta operación también es muy rápida ya que realmente no se cierran sino que se marcan como "**en espera**"

Si se piden **más conexiones de las que hay esperando** en el pool **se crearán** en ese instante nuevas conexiones **hasta el máximo de conexiones que permita el pool**. Si hay ya demasiadas conexiones esperando se cerrarán para ahorrar recursos en el servidor de base de datos.

Con esto conseguimos:

- Evitar conexiones abiertas cuando el usuario se marcha ya que cada conexión se pseudo-abre y pseudo-cierra con cada petición.
- Tenemos un número estable de conexiones independiente del número de usuarios que utilicen la aplicación.
- Al iniciar un servidor Java EE, automáticamente el pool de conexiones crea un número de conexiones físicas iniciales.

Cuando un objeto Java del servidor J2EE necesita una conexión, la solicita a través del método `dataSource.getConnection()`, la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y éste le entrega una conexión lógica `java.sql.Connection`. Esta conexión lógica la recibe por último, el objeto Java.

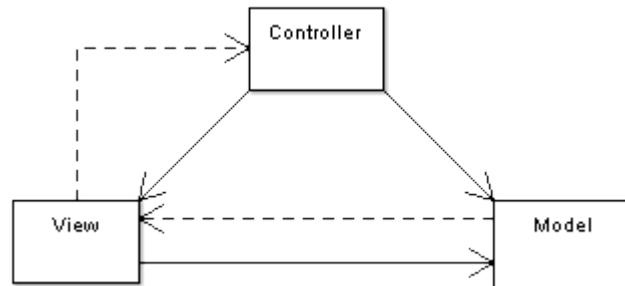
Cuando un objeto Java desea cerrar una conexión utiliza el método `connection.close()`, la fuente de datos `javax.sql.DataSource` habla con el pool de conexiones y le devuelve la conexión lógica en cuestión.

Si hay un pico en la demanda de conexiones, el pool de forma transparente crea más conexiones físicas de objetos tipo `Connection`. Si baja la demanda, el pool de forma transparente elimina conexiones físicas de objetos de tipo `Connection`.

EJEMPLO DE POOL DE CONEXIONES

A continuación se muestra un ejemplo de un proyecto en el que se crea una aplicación gráfica que permita consultar datos almacenados en una Base de Datos.

Utiliza el modelo vista controlador, la vista se crea con JSP, el modelo establece la conexión a una base de datos Mysql y la parte lógica se desarrolla con Servlets.



Para aprender a realizar un pool de conexiones con MySQL, se aporta el siguiente vídeo explicativo:

En el siguiente [vídeo](#) se explica paso a paso cómo realizar este ejercicio utilizando un pool de conexiones.

EJEMPLO DE POOL DE CONEXIONES CON ORACLE

El siguiente código establece una conexión con la base de datos Oracle, utilizando un Pool de conexiones Oracle:

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleConnection;
import java.sql.DatabaseMetaData;

public class DataSourceSample {

    final static String DB_URL= "jdbc:oracle:thin:@localhost:1521:XE";
    final static String DB_USER = "scott";
    final static String DB_PASSWORD = "tiger";

    public static void main(String args[]) throws SQLException {

        Properties info = new Properties();
        info.put(OracleConnection.CONNECTION_PROPERTY_USER_NAME,
DB_USER);
        info.put(OracleConnection.CONNECTION_PROPERTY_PASSWORD,
DB_PASSWORD);

        info.put(OracleConnection.CONNECTION_PROPERTY_DEFAULT_ROW_PREFETCH,
"20");

        info.put(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUIRED");

        OracleDataSource ods = new OracleDataSource();
        ods.setURL(DB_URL);
        ods.setConnectionProperties(info);

        // With AutoCloseable, the connection is closed automatically.
        try (OracleConnection connection = (OracleConnection)
ods.getConnection()) {

            // Get the JDBC driver name and version
            DatabaseMetaData dbmd = connection.getMetaData();
            System.out.println("Driver Name: " + dbmd.getDriverName());
            System.out.println("Driver Version: " +
dbmd.getDriverVersion());
        }
    }
}
```

```
// Print some connection properties
System.out.println("Default Row Prefetch Value is: " +
connection.getDefaultRowPrefetch());
System.out.println("Database Username is: " +
connection.getUserName());
System.out.println();

// Perform a database operation
printEmployees(connection);

}

}

public static void printEmployees(OracleConnection connection) throws
SQLException {

    // Statement and ResultSet are AutoCloseable and closed
    automatically.
    try (Statement statement = connection.createStatement()) {
        try (ResultSet resultSet = statement.executeQuery("select
first_name, last_name from employees")) {
            System.out.println("FIRST_NAME" + " " + "LAST_NAME");
            System.out.println("-----");
            while (resultSet.next())
                System.out.println(resultSet.getString(1) + " "+
resultSet.getString(2) + " ");
        }
    }
}
```

Características de las BD objeto-relacionales



BASES DE DATOS OBJETO-RELACIONALES

Las BDOR las podemos ver como un híbrido de las BDR y las BDOO que intenta aunar los beneficios de ambos modelos, aunque por descontado, ello suponga renunciar a algunas características de ambos.

Los **objetivos** que persiguen estas bases de datos son:

- Mejorar la representación de los datos mediante la orientación a objetos.
- Simplificar el acceso a datos, manteniendo el sistema relacional.

En una BDOR se siguen almacenando **tablas en filas y columnas**, aunque la estructura de **las filas no** está restringida a **contener escalares o valores atómicos, sino** que las columnas pueden almacenar **tipos estructurados** (tipos compuestos como vectores, conjuntos, etc.) y las **tablas** pueden ser **definidas en función de otras**, que es lo que se denomina **herencia directa**.

Esto es posible porque internamente tanto las **tablas** como las **columnas son** tratados como **objetos**, esto es, **se realiza un mapeo objeto-relacional** de manera transparente.

Como consecuencia de esto, aparecen las siguientes **características**:

- **Tipos definidos por el usuario:** Se pueden crear nuevos tipos de datos definidos por el usuario, y que son compuestos o estructurados, esto es, será posible tener en una columna un atributo multivaluado (un tipo compuesto).
- **Tipos Objeto:** Posibilidad de creación de objetos como nuevo tipo de dato que permiten relaciones anidadas.
- **Reusabilidad.** Posibilidad de guardar esos tipos en el gestor de la BDOR, para reutilizarlos en tantas tablas como sea necesario
- **Creación de funciones:** Posibilidad de definir funciones y almacenarlas en el gestor. Las funciones pueden modelar el comportamiento de un tipo objeto, en este caso se llaman métodos.
- **Tablas anidadas:** Se pueden definir columnas como arrays o vectores multidimensionales, tanto de tipos básicos como de tipos estructurados, esto es, se pueden anidar tablas.

- **Herencia** con subtipos y subtablas.

Introducción

Podemos decir que un sistema gestor de bases de datos objeto-relacional (SGBDOR) contiene dos tecnologías; la tecnología relacional y la tecnología de objetos, pero con ciertas restricciones.

A continuación te indicamos algunos ejemplos de gestores objeto-relacionales:

- Oracle
- PostgreSQL
- MySQL



POSTGRESQL

Como base de datos objeto relacional a desarrollar se utilizará **PostgreSQL**, el cual puede descargarse desde el siguiente [enlace](#)

Creación de un Schema

Para realizar una mejor gestión de los objetos de una base de datos y con el fin de "empaquetar" los objetos, PostgreSQL permite la creación de **Schema**

SCHEMA

Un **Schema** sería como un espacio de nombres que contiene objetos de la base de datos de tal forma que nos aseguramos que el nombre de estos elementos es único.

En un Schema se pueden incluir:

- Tablas
- Tipos de datos
- Indices
- Funciones
- Vistas
- Operadores

Existen las siguientes razones para elegir crear un **schema**:

- Permiten que varios usuarios utilicen la base de datos sin que exista interferencia entre ellos.
- Permiten organizar los objetos de la base de datos en grupos lógicos haciéndolos más manejables.
- Permiten separar las aplicaciones de terceros entre sí de forma que no colisiones los nombres de sus objetos.

La sintaxis para la creación de un schema es la siguiente:

```
CREATE SCHEMA nombreEsquema
```

Para indicar que un objeto, tabla, etc es creado dentro de un determinado Schema deberá ponerse *nombreEsquema*. antes del nombre del elemento.

EJEMPLO

```
CREATE TABLE nombreEsquema.nombreTabla(...);
```

```
CREATE TYPE nombreEsquema.nombreObjeto(...);
```

Estructura de un tipo

💡 TIPO

Un **tipo** de objeto representa una entidad del mundo real. Encapsula datos y operaciones, por lo que en la especificación sólo se pueden declarar atributos y métodos, pero no constantes, excepciones, cursor o tipos.

Al menos un atributo es requerido y los métodos son opcionales. Se compone de los siguientes elementos:

- **Nombre:** que sirve para identificar el tipo de los objetos.
- **Atributos:** modelan la estructura y los valores de los datos de ese tipo. Cada atributo puede ser de un tipo de datos básico o de un tipo de usuario.
- **Métodos:** procedimientos o funciones escritos en lenguaje PL/SQL.

Los tipos de objetos actúan como plantillas para los objetos de cada tipo.

Podemos diferenciar tres estructuras de sintaxis diferentes:

- Creación de tipo básico

```
CREATE TYPE name AS  
( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )
```

🔥 EJEMPLO

Creación de un tipo `compfoo` formado por un entero y una cadena de texto

```
CREATE TYPE compfoo AS (f1 int, f2 text);
```

- Creación de enumerado o array

```
CREATE TYPE name AS ENUM  
( [ 'label' [, ... ] ] )
```

```
CREATE TYPE name AS RANGE (
```

```

SUBTYPE = subtype
[ , SUBTYPE_OPCCLASS = subtype_operator_class ]
[ , COLLATION = collation ]
[ , CANONICAL = canonical_function ]
[ , SUBTYPE_DIFF = subtype_diff_function ]
[ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
)

```

EJEMPLO

Creación de un enumerado llamado `estado_bug` y utilización en la tabla `bug`

```

CREATE TYPE estado_bug AS ENUM ('nuevo', 'abierto', 'cerrado');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);

```

Creación de un array

```

CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff =
float8mi);

```

- Creación de tipo a partir de funciones

```

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , SUBSCRIPT = subscript_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
)

```

```
[ , PREFERRED = preferred ]
[ , DEFAULT = default ]
[ , ELEMENT = element ]
[ , DELIMITER = delimiter ]
[ , COLLATABLE = collatable ]
)
```

EJEMPLO

Creación del tipo `caja` y utilización en la definición de una tabla.

```
CREATE TYPE caja;
```

```
CREATE FUNCTION mi_caja_in_function(cstring) RETURNS caja AS ... ;
CREATE FUNCTION mi_caja_out_function(caja) RETURNS cstring AS ... ;
```

```
CREATE TYPE caja (
    INTERNALLENGTH = 16,
    INPUT = mi_caja_in_function,
    OUTPUT = mi_caja_out_function
);
```

SABER MÁS

Si se quiere saber más sobre qué es cada elemento se puede consultar el [siguiente enlace](#)

Tipos disponibles en PostgreSQL

Numeric	Character	Date/Time	Monetary	Binary
Boolean	Geometric	JSON	Enumerated	Text-search
UUID	Network Address Types	Composite	Object Identifiers	Pseudo
BitString	XML	Range	Arrays	pg_lns

Creación de tablas

Para la creación de una tabla utilizaremos la siguiente sintaxis:

```
CREATE TABLE [IF NOT EXISTS] nombre (
    nombre_atributo tipo_atributo [restricciones],
);
```



EJEMPLOS

```
CREATE TABLE films (
    code      char(5) CONSTRAINT firstkey PRIMARY KEY,
    title     varchar(40) NOT NULL,
    did       integer NOT NULL,
    date_prod date,
    kind      varchar(10) UNIQUE, // Restricción de valores únicos
    len       interval hour to minute
);
```

```
CREATE TABLE distributors (
    did      integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, // Clave primaria por defecto
    name    varchar(40) NOT NULL CHECK (name <> '')
);
```

```
// Creación de una tabla con un array
CREATE TABLE array_int (
    vector int[][][]
);
```

```
CREATE TABLE distributors (
    did      integer CHECK (did > 100), // Propia restricción en la creación
    name    varchar(40)
);
```

```
CREATE TABLE films (
    code      char(5),
    title     varchar(40),
    did       integer,
    CONSTRAINT code_title PRIMARY KEY(code,title) // Creación clave
```

```
primaria compuesta  
);
```

Relaciones entre tablas

Relación 1 a 1

Para representar una relación 1 a 1 se utilizaría un código similar al siguiente:

```
CREATE TABLE persona (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255),
    -- Otras columnas de persona
    direccion_id INT UNIQUE REFERENCES direccion(id)
);

CREATE TABLE direccion (
    id SERIAL PRIMARY KEY,
    calle VARCHAR(255),
    ciudad VARCHAR(255),
    -- Otras columnas de dirección
);
```

Relación 1 a varios

Para representar una relación 1 a varios se utilizaría un código similar al siguiente:

```
CREATE TABLE departamento (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255),
    -- Otras columnas del departamento
);

CREATE TABLE empleado (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255),
    salario NUMERIC,
    -- Otras columnas del empleado
    departamento_id INT REFERENCES departamento(id)
);
```

Relación varios a varios

Para representar una relación varios a varios se utilizaría un código similar al siguiente:

```
CREATE TABLE estudiante (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255),
    -- Otras columnas del estudiante
);

CREATE TABLE curso (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(255),
    -- Otras columnas del curso
);

-- Tabla de enlace para representar la relación muchos a muchos
CREATE TABLE inscripcion (
    estudiante_id INT REFERENCES estudiante(id),
    curso_id INT REFERENCES curso(id),
    PRIMARY KEY (estudiante_id, curso_id)
);
```

Acceso a objetos

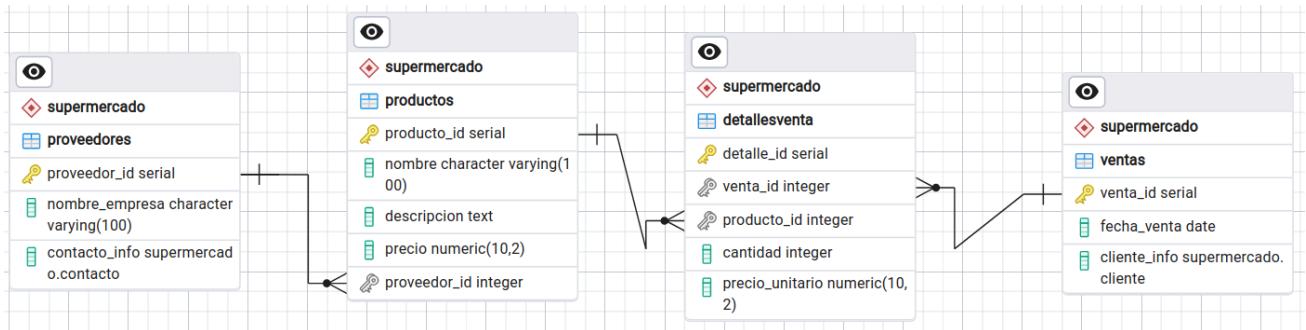
Una de las principales características de las bases de datos objeto relacional es que en una celda (combinación fila-columna) se puede almacenar objetos. Estos objetos a su vez pueden almacenar atributos.

Para poder acceder a estos atributos se empleará el punto (.) de forma muy similar a su funcionamientos en otro lenguajes de programación como Java o C++.



EJEMPLO DE ACCESO A LOS CAMPOS DE UN OBJETO

Dado el siguiente modelo Entidad-Relación de una base de datos denominada *supermercado*:



Cuyo código SQL es el siguiente:

▼ Código SQL

```
CREATE SCHEMA supermercado;
```

```
-- Tipo de datos compuesto para la información de contacto
```

```
CREATE TYPE supermercado.Contacto AS (
    telefono VARCHAR(15),
    direccion VARCHAR(255),
    email VARCHAR(100)
);
```

```
-- Tipo de datos compuesto para la información del proveedor
```

```
CREATE TYPE supermercado.Proveedor AS (
    proveedor_id INTEGER,
    nombre_empresa VARCHAR(100),
```

```
    contacto_info supermercado.Contacto
);

-- Tipo de datos compuesto para la información del cliente
CREATE TYPE supermercado.Cliente AS (
    nombre_cliente VARCHAR(100),
    direccion_cliente VARCHAR(255),
    email_cliente VARCHAR(100),
    edad INTEGER
);

-- Tabla para los proveedores
CREATE TABLE supermercado.Proveedores (
    proveedor_id SERIAL PRIMARY KEY,
    nombre_empresa VARCHAR(100),
    proveedor_info supermercado.Proveedor
);

-- Tabla para los productos
CREATE TABLE supermercado.Productos (
    producto_id SERIAL PRIMARY KEY,
    nombre VARCHAR(100),
    descripcion TEXT,
    precio DECIMAL(10, 2),
    proveedor_id INT UNIQUE REFERENCES
supermercado.Proveedores(proveedor_id)
);

-- Tabla para las ventas
CREATE TABLE supermercado.Ventas (
    venta_id SERIAL PRIMARY KEY,
    fecha_venta DATE,
    cliente_info supermercado.Cliente
);

-- Tabla de detalles de la venta
CREATE TABLE supermercado.DetallesVenta (
    detalle_id SERIAL PRIMARY KEY,
    venta_id INTEGER,
    producto_id INTEGER,
    cantidad INTEGER,
    precio_unitario DECIMAL(10, 2),
    FOREIGN KEY (venta_id) REFERENCES supermercado.Ventas(venta_id),
    FOREIGN KEY (producto_id) REFERENCES
supermercado.Productos(producto_id)
);
```

Se pueden observar que existen tres tipos de datos que hacen referencia a objetos que son *Contacto*, *Proveedor* y *Cliente* que están asignados, respectivamente, a los campos **contacto_info** en el tipo Proveedor, **proveedor_info** en la tabla Proveedores y **cliente_info** en la tabla Ventas.

Si se quisiera listar el nombre y el email de todos los clientes de la tabla Ventas se utilizaría la siguiente consulta:

```
select (cliente_info).nombre_cliente, (cliente_info).email_cliente from  
supermercado.ventas;
```

En este caso, se utilizan los paréntesis para indicar que el campo *cliente_info* es un tipo objeto. **Estos paréntesis solo son necesarios cuando se realiza una consulta.**

Una vez se indica que el tipo de datos es de tipo objeto, se puede utilizar el operador punto para acceder a los campos de dicho objeto. En este caso lo utilizamos para acceder a los campos **nombre_cliente** y **email_cliente**.



EJEMPLO DE ACCESO A LOS CAMPOS DE UN OBJETO DENTRO DE OTRO OBJETO

Siguiendo la misma lógica que en el ejemplo anterior, si una tabla tiene un campo objeto que, a su vez tiene un campo que también es un objeto se podría hacer de la siguiente manera.

Si se quiere obtener el *email* de un proveedor se realizaría la siguiente consulta:

```
select ((proveedor_info).contacto_info).email from  
supermercado.proveedores;
```

Primero se accede al campo *proveedor_info* indicando con los paréntesis que es un objeto.

Dentro de ese objeto se accede al campo *contacto_info* como este campo vuelve a ser un objeto se vuelve a encerrar toda la expresión entre paréntesis.

Finalmente, accedemos al campo objetivo, el *email*.

Consultar



CLÁUSULA SELECT

Para consultar datos de una tabla se utilizará la cláusula **SELECT**

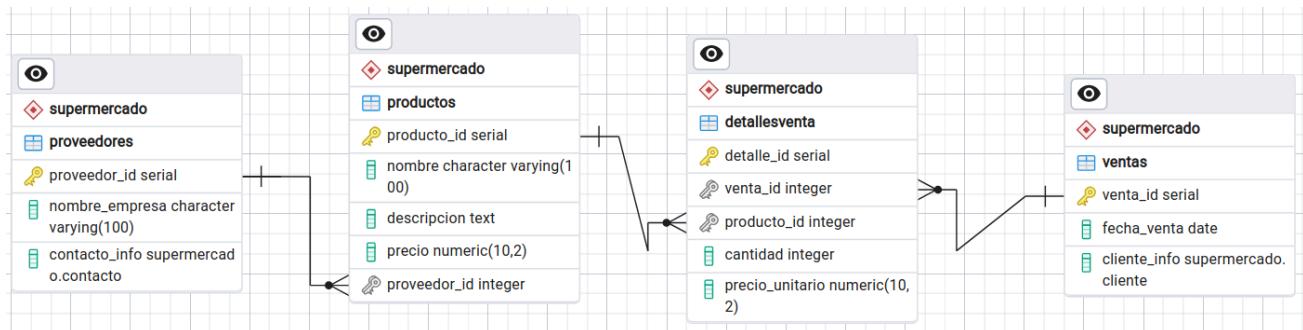
La sintaxis es la siguiente:

```
SELECT columna1, columna2, ...
FROM nombre_tabla
WHERE condición;
```



CONSULTA BÁSICA

Dada la base de datos del supermercado analizada en la sección sobre acceso a objetos la cual tenía el siguiente modelo relacional.



Si se quisiera listar la **descripción** de un producto, se utilizaría la siguiente consulta:

```
select descripcion from supermercado.productos;
```

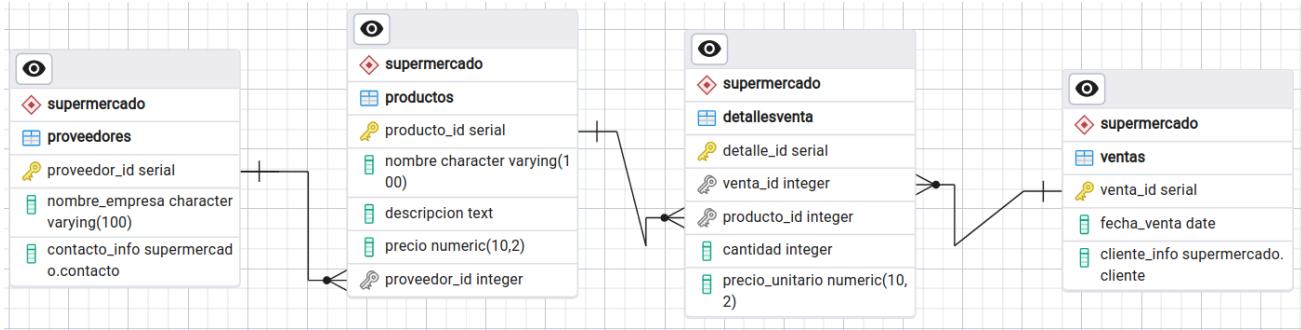


CONSULTAR EL CAMPO DE UN OBJETO

Para la misma base de datos supermercado, se pide listar el **nombre** y el **email** de todos los clientes de la tabla Ventas. La consulta resultante será la siguiente:

```
select (cliente_info).nombre_cliente, (cliente_info).email_cliente from
supermercado.ventas;
```

CONSULTAR EL CAMPO DE UN OBJETO QUE A SU VEZ ESTÁ DENTRO DE UN OBJETO.



Para la misma base de datos supermercado, se pide obtener el **email** de un proveedor. La consulta resultante será la siguiente:

```
select ((proveedor_info).contacto_info).email from
supermercado.proveedores;
```

Insertar

CLÁUSULA INSERT

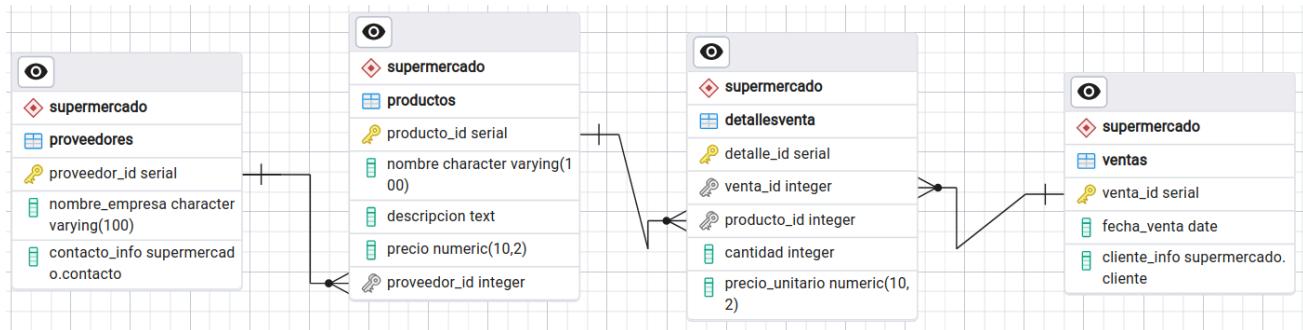
Para insertar una nueva fila en una tabla utilizaremos la cláusula **INSERT**

La sintaxis es la siguiente:

```
INSERT INTO nombre_tabla(columna1, columna2, ...)
VALUES
    (valor1, valor1, ...);
```

INSERCIÓN DE DATOS SIMPLES

Dada la base de datos del supermercado analizada en la sección sobre [acceso a objetos](#) la cual tenía el siguiente modelo relacional.



Para añadir datos simples a una tabla utilizaríamos una expresión del siguiente estilo:

```
INSERT INTO supermercado.Productos (nombre, descripcion, precio,
proveedor_id)
VALUES
    ('Producto 1', 'Descripción del producto 1', 10.50, 1),
    ('Producto 2', 'Descripción del producto 2', 8.75, 2);
```

INSERTAR DATO COMPLEJO

Siguiendo con la misma base de datos del ejemplo anterior.

Si por el contrario, se quisieran insertar datos en datos compuestos (objetos) la expresión que habría que utilizar sería haciendo uso de la cláusula **ROW**.

```
INSERT INTO supermercado.Proveedores (nombre_empresa, contacto_info)
VALUES
    ('Proveedor A', ROW('234567890', 'Calle 1, Ciudad A',
    'proveedora@email.com')),
    ('Proveedor B', ROW('876543210', 'Calle 2, Ciudad B',
    'proveedorb@email.com'));
```

Actualizar

CLÁUSULA UPDATE

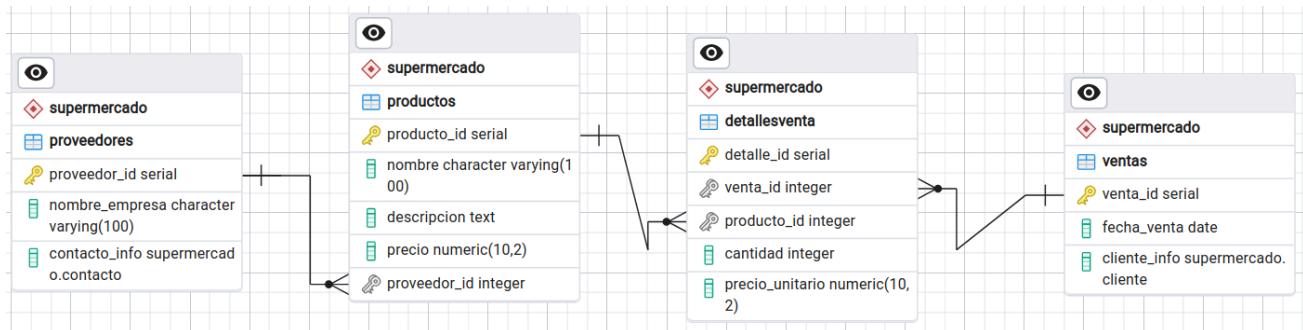
Para actualizar los valores de una fila en una tabla utilizaremos la cláusula **UPDATE**

La sintaxis es la siguiente:

```
UPDATE nombre_tabla
SET columna1 = valor1,
    columna1 = valor1, ...
WHERE
    condición;
```

MODIFICACIÓN DA DATOS SIMPLES

Dada la base de datos del supermercado analizada en la sección sobre acceso a objetos la cual tenía el siguiente modelo relacional.



```
UPDATE supermercado.productos SET descripción = 'Nueva descripción' WHERE
producto_id = 1;
```

MODIFICACIÓN DA UN DATO COMPLETO

Siguiendo con la misma base de datos del ejemplo anterior.

```
UPDATE supermercado.ventas SET cliente_info = ROW('Cliente A', 'Dirección
A', 'clienteA@email.com') WHERE venta_id = 1;
```

MODIFICACIÓN DA UN CAMPO DE UN DATO COMPLEJO

Siguiendo con la misma base de datos del ejemplo anterior.

```
UPDATE supermercado.ventas SET cliente_info.nombre_cliente = 'Cliente A'  
WHERE venta_id = 1;
```

Borrar



CLÁUSULA DELETE

Para borrar valores de una tabla utilizaremos la cláusula **DELETE**

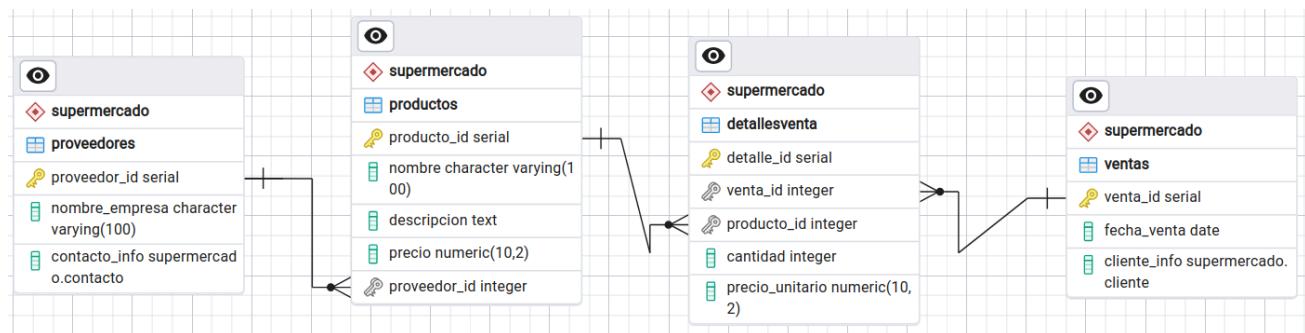
La sintaxis es la siguiente:

```
DELETE FROM nombre_tabla  
WHERE condición;
```



EJEMPLO

Dada la base de datos del supermercado analizada en la sección sobre [acceso a objetos](#) la cual tenía el siguiente modelo relacional.



```
DELETE FROM supermercado.ventas WHERE venta_id = 1;
```

Upsert



CLÁUSULA UPSERT

La cláusula **Upsert** es una característica de un DBMS que permite a una sentencia DML realizar diferentes acciones cuando se almacena o actualiza una fila.

La sintaxis es la siguiente:

```
INSERT INTO nombre_tabla(lista_columna)
VALUES(lista_valores)
ON CONFLICT target action;
```

Donde:

- **target**: puede ser:
 - el nombre de una columna.
 - ON CONSTRAINT nombre_restriccion: donde *nombre_restriccion* puede ser una restricción UNIQUE.
 - WHERE *predicado*: cláusula WHERE con una condición booleana.
- **action** puede ser:
 - DO NOTHING: si la fila existe en la tabla, no hacer nada
 - DO UPDATE SET columnaA=valorA, ... WHERE condición: actualiza ciertos campos de la tabla dependiendo de la condición

Gestión de transacciones

TRANSACCIÓN

Una **transacción** es un conjunto de sentencias que se ejecutan formando una unidad de trabajo, esto es, en forma indivisible o atómica, o se ejecutan todas o no se ejecuta ninguna.

Mediante la gestión de transacciones, los sistemas gestores proporcionan un acceso concurrente a los datos almacenados, mantienen la **integridad** y **seguridad** de los datos, y proporcionan un **mecanismo de recuperación** de la base de datos ante fallos.

Las transacciones deben cumplir el criterio ACID.

- **Atomicidad:** Se deben cumplir todas las operaciones de la transacción o no se cumple ninguna; no puede quedar a medias.
- **Consistencia:** la transacción solo termina si la base de datos queda en un estado consistente.
- **Aislamiento (Isolation):** Las transacciones sobre la misma información deben ser independientes, para que no interfieran sus operaciones y no se produzca ningún tipo de error.
- **Durabilidad:** Cuando la transacción termina el resultado de la misma perdura, y no se puede deshacer aunque falle el sistema.

Alguno sistemas también proporcionan savepoints.

SAVEPOINTS

Un **punto de salvaguarda** o **savepoint** permite descartar selectivamente partes de un transacción, justo antes de acometer el resto.

Después de definir un punto como punto de salvaguarda, se puede retroceder al mismo.

De esta forma se descartan todos los cambios hechos por la transacción después del punto de salvaguarda, pero se mantienen todos los anteriores.

Conexión

Para realizar la conexión entre Java y una base de datos de PostgreSQL utilizaremos el siguiente código:

```
Connection conexion = null;

try {
    conexion =
    DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
    "usuario", "password");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
} catch (SQLException sqle) {
    sqle.printStackTrace();
} catch (InstantiationException ie) {
    ie.printStackTrace();
} catch (IllegalAccessException iae) {
    iae.printStackTrace();
}
```

Pudiendo ser acortado a:

```
Connection conexion = null;

try {
    conexion =
    DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
    "usuario", "password");
} catch (Exception e) {
    e.printStackTrace();
}
```

Creación de una tabla

Haciendo uso de la sintaxis de PostgreSQL, podemos crear una tabla en Java de la siguiente manera

```
Connection conexion = null;
Statement stmt = null;

try {
    conexion =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
"usuario", "password");
    stmt = conexion.createStatement();
    String sql = "CREATE TABLE COMPANY " +
"(ID INT PRIMARY KEY      NOT NULL," +
" NAME          TEXT      NOT NULL, " +
" AGE           INT       NOT NULL, " +
" ADDRESS        CHAR(50), " +
" SALARY         REAL)";
    stmt.executeUpdate(sql);
    stmt.close();
    conexion.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Insertar valores

Haciendo uso de la sintaxis de PostgreSQL, podemos insertar valores en una tabla usando Java de la siguiente manera:

```
Connection conexion = null;
Statement stmt = null;

try {
    conexion =
    DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
    "usuario", "password");

    conexion.setAutoCommit(false);

    System.out.println("Opened database successfully");

    stmt = conexion.createStatement();
    String sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
    + "VALUES (1, 'Paul', 32, 'California', 20000.00 );";
    stmt.executeUpdate(sql);
    sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
    + "VALUES (2, 'Allen', 25, 'Texas', 15000.00 );";
    stmt.executeUpdate(sql);
    sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
    + "VALUES (3, 'Teddy', 23, 'Norway', 20000.00 );";
    stmt.executeUpdate(sql);
    sql = "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) "
    + "VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 );";
    stmt.executeUpdate(sql);

    stmt.close();

    conexion.commit();

    conexion.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Actualizar valores

Haciendo uso de la sintaxis de PostgreSQL, podemos actualizar los valores de una fila usando Java de la siguiente manera:

```
Connection conexion = null;
Statement stmt = null;

try {
    conexion =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
"usuario", "password");

    String updateSQL = "UPDATE tu_tabla SET columna1 = ?, columna2 = ? WHERE
condicion";

    // Crear una PreparedStatement con la sentencia SQL
    PreparedStatement preparedStatement =
conexion.prepareStatement(updateSQL);

    // Establecer los nuevos valores de las columnas
    preparedStatement.setString(1, "nuevo_valor_columna1");
    preparedStatement.setString(2, "nuevo_valor_columna2");

    // Establecer la condición para la actualización
    // Puedes utilizar una condición WHERE para especificar qué filas se deben
actualizar
    // Ejemplo: preparedStatement.setInt(3, id); // id sería el valor para la
condición

    // Ejecutar la actualización
    int rowsUpdated = preparedStatement.executeUpdate();

    if (rowsUpdated > 0) {
        System.out.println("Actualización exitosa. Filas actualizadas: " +
rowsUpdated);
    } else {
        System.out.println("Ninguna fila actualizada. La condición no coincide
con ningún registro.");
    }

    conexion.close();
} catch (Exception e) {
```

```
e.printStackTrace();  
}
```

Eliminar valores

Haciendo uso de la sintaxis de PostgreSQL, podemos eliminar valores de una tabla usando Java de la siguiente manera:

```
Connection conexion = null;
Statement stmt = null;

try {
    conexion =
    DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos",
    "usuario", "password");

    // Definir la sentencia SQL de eliminación
    String deleteSQL = "DELETE FROM tu_tabla WHERE condicion";

    // Crear una PreparedStatement con la sentencia SQL
    PreparedStatement preparedStatement =
    conexion.prepareStatement(deleteSQL);

    // Establecer la condición para la eliminación
    // Puedes utilizar una condición WHERE para especificar qué filas se deben
    eliminar
    // Ejemplo: preparedStatement.setInt(1, id); // id sería el valor para la
    condición

    // Ejecutar la eliminación
    int rowsDeleted = preparedStatement.executeUpdate();

    if (rowsDeleted > 0) {
        System.out.println("Eliminación exitosa. Filas eliminadas: " +
    rowsDeleted);
    } else {
        System.out.println("Ninguna fila eliminada. La condición no coincide
    con ningún registro.");
    }

    conexion.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Operaciones: ejecución de consultas

Para operar con una base de datos, ejecutando las consultas necesarias, nuestra aplicación deberá hacer:

- **Cargar** el driver necesario para comprender el protocolo que usa la base de datos en cuestión.
- **Establecer** una conexión con la base de datos.
- **Enviar** consultas SQL y procesar el resultado.
- **Liberar** los recursos al terminar.
- **Gestionar** los errores que se puedan producir.

Podemos utilizar los siguientes tipos de sentencias:

- **Statement**: para sentencias sencillas en SQL.
- **PreparedStatement**: para consultas preparadas, como por ejemplo las que tienen parámetros.
- **CallableStatement**: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

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

Excepciones y cierre de conexiones

Cuando se produce un error durante la ejecución de un programa, se genera un objeto asociado a esa excepción. Ese objeto es de la clase `Exception` o de alguna de sus subclases. Este objeto se pasa entonces al código que se ha definido para gestionar la excepción.

En una porción de programa donde se trabajara con ficheros, y con bases de datos podríamos tener esta estructura para capturar las posibles excepciones.



EJEMPLO DE BLOQUE DE CAPTURA DE ERRORES

```
try{
    // Bloque de instrucciones del try

}catch(FileNotFoundException fnfe){
    // Bloque para excepción por fichero no encontrado
}catch(IOException ioe){
    // Bloque para excepción por error de entrada salida
}catch(SQLException sqle){
    // Bloque para excepción por error con SQL
}catch(Exception e){

}finally{
    // Instrucciones finales para, por ejemplo, limpieza
}
```

El bloque de instrucciones del `try` es el que se ejecuta. Si en él ocurre un error que dispara una excepción, entonces se comprueba si hay una sentencia `catch` que la pueda gestionar.

Las instrucciones que hay en el bloque del `finally`, **se ejecutarán siempre, se haya producido una excepción o no**, ahí suelen ponerse instrucciones de limpieza, de cierre de conexiones, etc.

Al realizar acciones sobre una base de datos se pueden lanzar excepción de tipo `SQLException`. Este tipo de excepción proporciona la siguiente información:

- Una cadena de caracteres describiendo el error. Se obtiene con el método `getMessage()`.
- Un código entero de error que especifica al fabricante de la base de datos.

Como se comentó en anteriores secciones, al trabajar con bases de datos se consumen muchos recursos por parte del sistema gestor, así como del resto de la aplicación. Por esta razón, resulta totalmente conveniente cerrarlas con el método `close()` cuando ya no se utilizan.



EJEMPLO DE CIERRE DE CONEXIÓN CON BLOQUE TRY

```
Connection con = null;

try{
    con = DiverManager.getConnection("jdbc:odbc:admdb");
    System.out.println("Conexión realizada con éxito.");

    // Aquí hacemos lo que necesitemos
}catch(SQLException e){

    // Aquí haríamos lo necesario para gestionar la excepción SQL

} finally{
    // Si hay conexión la cerramos
    if (con != null){
        try{
            con.close();
        }catch(SQLException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Patrón MVC

El patrón MVC (Modelo-Vista-Controlador) es un patrón de arquitectura de software que separa los datos de una aplicación:

- Por una parte la lógica de negocio.
- Por otra parte la interfaz de usuario.

Su uso está muy extendido porque mejora la calidad y eficiencia del código.

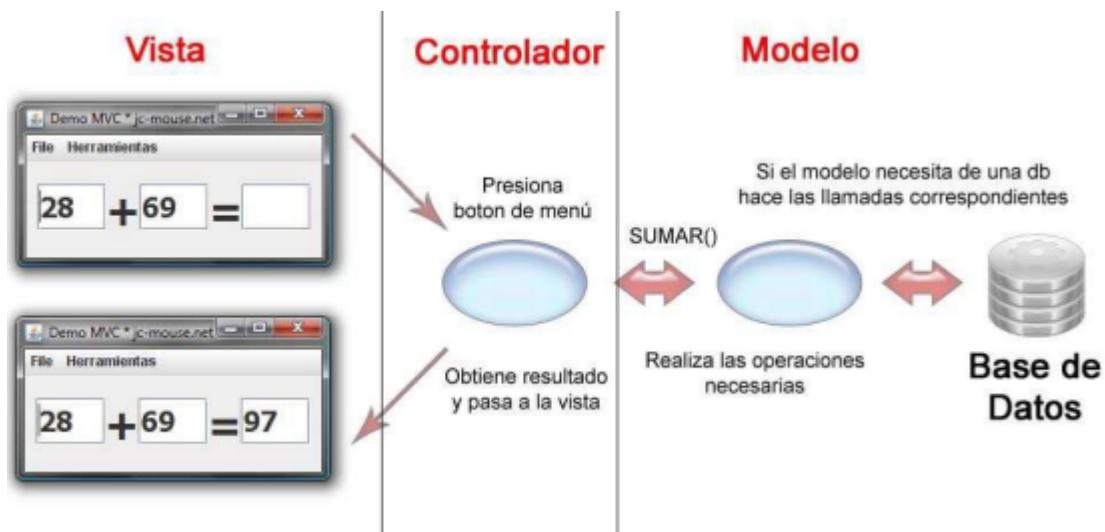
Este patrón organiza una aplicación en tres bloques, que son independientes entre sí:

- **Modelo:** representa la lógica de la aplicación y sus datos.
- **Vista:** representa la parte del código con la que interacciona el usuario. Es decir, está formada por las diferentes interfaces de nuestra aplicación.
- **Controlador:** es el bloque que se encarga de comunicar la vista con el modelo.

Entre las ventajas de este patrón encontramos:

- Hay una separación entre el almacenamiento de los datos y su representación visual.
- Facilita la reutilización de código.
- Facilita el mantenimiento, porque los cambios que se produzcan en una parte de la aplicación no afectarán a las demás.

La siguiente figura describe el flujo de una aplicación diseñada bajo el patrón MVC:



El funcionamiento sería el siguiente:

1. El usuario realiza una petición (evento) en la interfaz de la aplicación (pulsa un botón, un link...). Esta petición es recogida por la vista y la envía al controlador.
2. El controlador examina la petición y decide lo que debería hacer la aplicación. Entonces, el controlador llama al modelo para que ejecute una acción determinada.
3. El modelo ejecuta la acción (llamando a la base de datos si es necesario) y le devuelve el resultado al controlador.
4. El controlador le indica a la vista los resultados que debe mostrar.
5. La vista muestra estos resultados en la interfaz gráfica, de forma que el usuario los pueda visualizar.

Implementacion

A continuación, se va a detallar la función de cada uno de los elementos necesarios para implementar un patrón MVC de forma correcta:

Main

Esta clase Java será la encargada de:

- Crear las instancias de los tres componentes principales: Modelo, Vista y Controlador.
- Vincular el controlador con la vista y el modelo.
- Vincula la vista con el controlador.
- Arrancar la Vista.



EJEMPLO DE MAIN

▼ Ejemplo de Main

```
public class main {  
  
    public static void main(String[] args) {  
        // Crea la instancia del modelo  
        Modelo modelo = new Modelo();  
        // Crea la instancia de la vista  
        Vista vista = new Vista();  
        // Crea la instancia del controlador  
        // Vincula el controlador con la vista y  
        Controlador control = new Controlador (vista, modelo);  
        // Arranca la interfaz (vista):  
        vista.arranca();  
        // Vincula la vista con el modelo  
        vista.setControlador(control);  
    }  
}
```

Controlador

En el proyecto del tema anterior, cada vista implementaba la interfaz ActionListener por lo que controlador y vista estaban todo en la misma clase. Ahora vamos a separar interfaz de lógica de funcionamiento.

Para ello, es necesario que el controlador sea una clase diferente que implemente la interfaz **ActionListener**. De esta forma cuando ocurra una pulsación de un botón será nuestro controlador el encargado de gestionarla y decidir qué hacer con ella.

Además, al implementar esta interfaz será obligatorio añadir el método **actionPerformed(ActionEvent e)** ya que es este método el que se encarga de gestionar los eventos (al igual que el proyecto anterior). Dentro de este método, usaremos la función **e.getActionCommand()** para saber cuál es el nombre del evento que ha generado la pulsación para saber qué hacer con él.

A mayores de esta interfaz, el controlador necesitará disponer de un constructor que reciba el modelo y la vista. Esto es necesario para:

- Poder obtener y modificar la información de las interfaces de la vista
- Poder llamar a las funciones del modelo con la información necesaria.



EJEMPLO DE CONTROLADOR

▼ Ejemplo de Controlador

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controlador implements ActionListener{
    private Vista vista;
    private Modelo modelo;

    // Constructor del controlador con la vista y el modelo
    public Controlador(Vista vista, Modelo modelo){
        this.vista = vista;
        this.modelo = modelo;
    }

    // Método actionPerformed para gestionar los eventos de los
    // botones
```

```
@Override  
public void actionPerformed(ActionEvent e) {  
    // Obtenemos el nombre del evento que generó la pulsación  
    if(e.getActionCommand().equals("SUMAR")){  
        // Llamamos a la vista para obtener la información de la  
        interfaz  
        String[] param = vista.getCampos();  
        // Llamamos al modelo para realizar la lógica  
        modelo.sumar(Integer.parseInt(param[0]),  
        Integer.parseInt(param[1]));  
        // Obtenemos el resultado del modelo  
        int resultado = modelo.getSuma();  
        // Actualizamos en la interfaz el resultado  
        vista.escribirResultado(Integer.toString(resultado));  
    }  
}  
}
```

Vista

Formada por el conjunto de clases que permiten mostrar la interfaz gráfica de la aplicación. A parte de todo el código necesario para crear la interfaz dispondrá de:

- Una variable del tipo del Controlador.
- Un método para ejecutar/arrancar la interfaz gráfica.
- Un método para establecer la vinculación del controlador con la vista.

A cada botón de los que disponga la vista tendrá que aplicársele dos funciones:

- **addActionListener(Controlador controlador)**: permite indicarle al botón en cuestión que cuando se le pulse tendrá que usar el controlador como gestor de eventos.
- **setActionCommand(String evento)**: permite darle un nombre de identificación a la pulsación de un evento.

De esta forma, cuando se pulse un botón el controlador detectará la ocurrencia del evento y gracias a setActionCommand() podremos identificar qué botón ha sido pulsado.



EJEMPLO DE VISTA

▼ Ejemplo de Vista

```
public class Vista extends javax.swing.JFrame {  
  
    // Código de la vista  
  
    // Variable para almacenar el controlador que se utilizará  
    private Controlador controlador;  
  
    // Constructor de la vista  
    public Vista() {  
    }  
  
    // Método para arrancar/iniciar la interfaz  
    public void arranca(){  
        initComponents();  
        setLocationRelativeTo(null);  
        setVisible(true);  
    }  
  
    // Método para vincular el controlador con la vista  
    public void setControlador(Controlador controlador){  
        // Vinculación  
        this.controlador = controlador;  
        // Vincular el botón con el controlador  
        jButton1.addActionListener(controlador);  
        // Darle un nombre al evento de pulsación del botón  
        jButton1.setActionCommand("SUMAR");  
    }  
  
    // Función para recuperar la información de la vista  
    public String[] getCampos(){  
        return new String[]{jTextField1.getText(),  
        jTextField2.getText()};  
    }  
  
    // Función para modificar la vista  
    public void escribirResultado(String resultado){  
        this.jTextPane1.setText(resultado);  
    }  
}
```

Modelo

Es la clase Java que dispone de toda la lógica de operaciones. Las funciones podrán devolver el resultado directamente o bien almacenarlo en una variable interna de la clase.

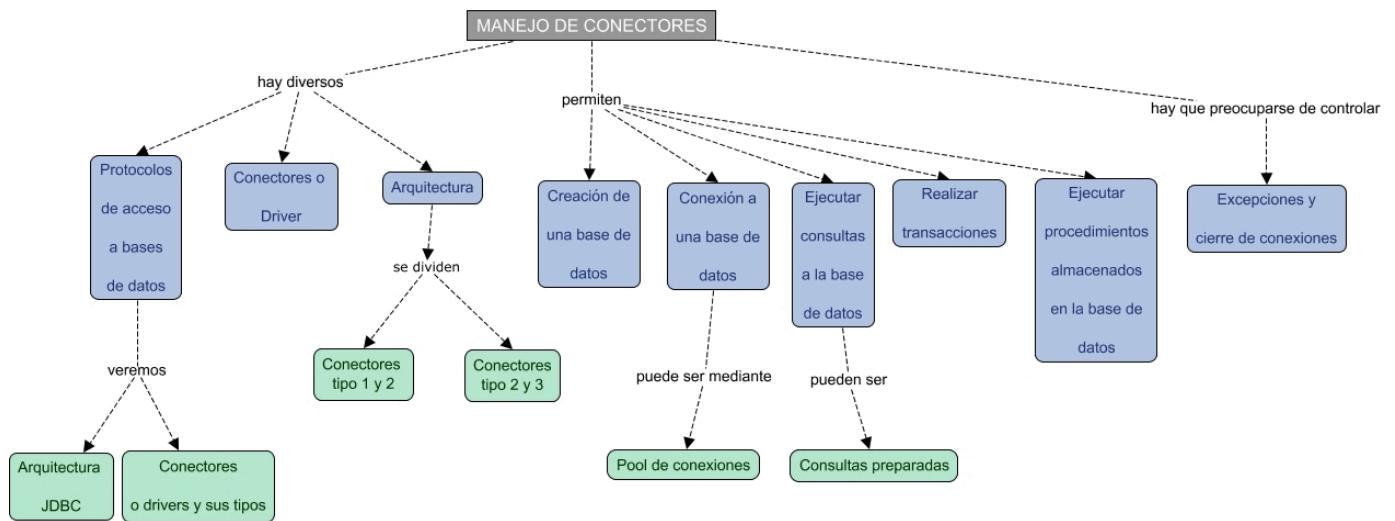


EJEMPLO DE MODELO

▼ Ejemplo de Modelo

```
public class Modelo {  
  
    // Variable para almacenar el resultado  
    private int resultadoSuma;  
  
    // Función para generar el resultado  
    public void sumar(int elemento1, int elemento2){  
        this.resultadoSuma = elemento1 + elemento2;  
    }  
  
    // Función para recuperar el resultado  
    public int getSuma(){  
        return this.resultadoSuma;  
    }  
}
```

Mapa conceptual



Chuleta SQL

Consultas básicas

- Filtrar columnas: `SELECT col1, col2, col3, ... FROM table1`
- Filtrar filas: `... WHERE col4 = 1 AND col5 = 2`
- Agrupar información: `... GROUP by ...`
- Limitar los datos agregados: `... HAVING count(*) > 1 ...`
- Ordenar los resultados: `... ORDER BY col2`

Palabras claves útiles para SELECT

- `DISTINCT`: return unique results
- `BETWEEN a AND b`: limit the range, the values can be numbers, text, or dates
- `LIKE`: pattern search within the column text
- `IN (a, b, c)`: check if the value is contained among given.

Modificadores de información

- Actualiza información específica con la cláusula WHERE: `UPDATE table1 SET col1 = 1 WHERE col2 = 2`
- Insertar valores manualmente: `INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME) VALUES (1, 'Rebel', 'Labs')`
- Insertar valores de forma manual usando una consulta: `INSERT INTO table1 (ID, FIRST_NAME, LAST_NAME) SELECT id, last_name, first_name FROM table2`

Vistas



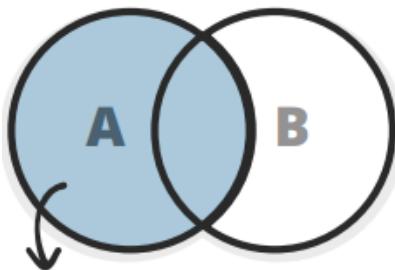
VISTA

Una vista es una tabla virtual en la cual se muestran los resultados de una consulta.

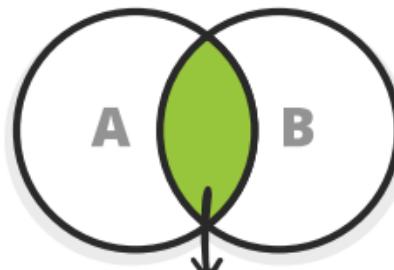
Pueden ser usadas para crear tablas virtuales que muestre consultas complejas:

```
CREATE VIEW view1 AS SELECT col1, col2 FROM table1 WHERE ...
```

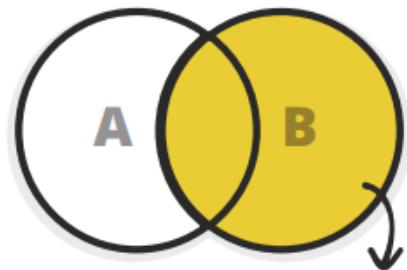
La diversión de los JOINs



LEFT OUTER JOIN - all rows from table A, even if they do not exist in table B



INNER JOIN - fetch the results that exist in both tables



RIGHT OUTER JOIN - all rows from table B, even if they do not exist in table A

En la imagen anterior:

- **RIGHT OUTER JOIN**: todas las filas de la tabla B, incluso si no existen en la tabla A.
- **INNER JOIN**: selecciona los resultados que existen en ambas tablas.
- **LEFT OUTER JOIN**: todas las filas de la tabla A, incluso si no existen en la tabla B

Actualizaciones usando consultas con JOIN

- Se pueden usar los JOINs en cláusulas UPDATE para actualizar los datos

```
UPDATE t1 SET a = 1 FROM table1 t1 JOIN table2 t2 ON t1.id = t2.t1_id WHERE t1.col1 = 0 AND t2.col2 IS NULL;
```

Semi JOINs

Se pueden usar sub consultas en lugar de JOINs:

```
SELECT col1, col2 FROM table1 WHERE id IN (SELECT t1_id FROM table2 WHERE date > CURRENT_TIMESTAMP)
```

Índices

Si las consultas se hacen sobre consultas, mejor indexarlas.

```
CREATE INDEX index1 ON table1 (col1)
```

RECORDATORIO

No olvidar, los índices permiten:

- Evitar sobreposición de índices
- Evitar indexar muchas columnas.
- Permite aumentar la velocidad de las operaciones de actualización (UPDATE) y las de eliminación (DELETE)

Funciones de gran utilidad Useful Utility Functions

- Convertir cadenas a fechas: `TO_DATE` (Oracle, PostgreSQL), `STR_TO_DATE` (MySQL)
- Devolver el primer argumento no nulo: `COALESCE (col1, col2, "default value")`
- Devuelve el tiempo actual: `CURRENT_TIMESTAMP`
- Une el resultado de dos resultados: `SELECT col1, col2 FROM table1 UNION / EXCEPT / INTERSECT SELECT col3, col4 FROM table2;`

`Union`: devuelve la información de ambas consultas. `Except`: las filas de la primera consulta que no están en la segunda. `Intersect`: las filas que son devueltas en ambas consultas

Reporting

Use aggregation functions

- `COUNT`: devuelve el número de filas
- `SUM`: suma acumulativa de un conjunto de valores
- `AVG`: devuelve la media de valores de un grupo
- `MIN / MAX`: valor más pequeño / más grande

Patrón singleton



PATRÓN SINGLETON

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Problema

El patrón Singleton resuelve dos problemas al mismo tiempo:

- Garantizar que una clase tenga una única instancia:** permitiendo controlar el acceso a un recurso compartido, ej.: el acceso a una base de datos.
- Proporcionar un punto de acceso global a dicha instancia:** permite funcionar como una variable global pero con más seguridad, ya que de esta forma es más difícil sobrescribir el contenido de esas variables y descomponer la aplicación.

Solución

Implementar el patrón Singleton consta de dos pasos:

- Hacer privado el constructor** por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
- Crear un método de creación estático que actúe como constructor.** Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.



EJEMPLO DE IMPLEMENTACIÓN DEL PATRÓN SINGLETON

El patrón Singleton permite crear una única instancia de acceso a la BD.

```
public class Database{  
  
    private static Connection connection;  
    private final String usuario = "root";
```



```
private final String clave = "abc123.";
private final String url ="jdbc:mysql://localhost:3306/school";

// El constructor del singleton siempre debe ser privado para evitar
// llamadas de construcción directas con el operador `new`.
private Database(){
    try {
        this.connection = DriverManager.getConnection(url, usuario,
clave);
    }catch(SQLException sqle) {
        System.out.println("Error al abrir la conexión");
    }
}

// El método estático que controla el acceso a la instancia
// singleton.
public static Connection getInstance(){
    if (Database.connection == null)
        new Database();
    return Database.connection;
}
}
```

Patrón DAO (Data Access Object)



PATRÓN DAO

El **patrón de diseño DAO** permite separar la lógica de acceso a datos de los Objetos de negocios, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Problema

Una de las grandes problemáticas al momento de acceder a los datos, es que la implementación y formato de la información puede variar según la fuente de los datos.

Mezclar la lógica de acceso a datos con la lógica de negocio es un problema ya que es necesario tener en cuenta las dos partes lo que aumenta el grado de complejidad del sistema.

Si el sistema solo emplea una fuente de datos la problemática no es muy compleja, sin embargo si esta fuente de datos cambia o aumenta el número de ellas la complejidad crece de forma exponencial.

Solución

El patrón DAO propone **separar** por completo la **lógica de negocio de la lógica para acceder a los datos**. DAO proporcionará los métodos CRUD necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.



MÉTODOS CRUD

Las siglas CRUD vienen del inglés y hacen referencia a los métodos de:

- Create (Crear)
- Read (Leer)
- Update (Actualizar)

- Delete (Eliminar)

EJEMPLO DE USO DEL PATRÓN DAO

Dada la siguiente tabla:

inquilinos	
!	idInquilinos INT
◆	dni VARCHAR(8)
◆	nombres VARCHAR(150)
◆	paterno VARCHAR(150)
◆	materno VARCHAR(150)
◆	telefono VARCHAR(40)
◆	correo VARCHAR(200)
◆	deuda DECIMAL(6,2)
◆	fecha_ingreso DATE

Si queremos gestionar esa tabla lo primero que tendríamos que hacer es crear una interfaz que tenga la definición de los métodos que queremos implementar.

```
public interface DaoInquilinosInterfaz {
    public List <Inquilinos> inquilinosSel(); // Devuelve la
    lista de inquilinos con todos sus datos
    public Inquilinos inquilinosGet(Integer id); // Devuelve el
    inquilino cuyo id coincida con el parámetro pasado
    public String inquilinosIns(Inquilinos inquilino); // Permite
    insertar un nuevo inquilino
    public String inquilinosUpd(Inquilinos inquilino); // Permite
    actualizar el valor de los atributos de un inquilino
    public String inquilinosDel(List<Integer> ids); // Permite
    eliminar inquilinos
}
```

Una vez creada la interfaz lo que tenemos que hacer es crear una clase que implemente esa interfaz:

```
public class DaoInquilinosImpl implements DaoInquilinosInterfaz {
    private final ConectaBD conectaDb;
```

```
public DaoInquilinosImpl() {
    this.conectaDb = new ConectaBD();
}

@Override
public List<Inquilinos> inquilinosSel() {
    // Código java de la consulta a la bd y gestiones pertinentes
}

@Override
public List<Inquilinos> inquilinosSel() {
    // Código java de la consulta a la bd y gestiones pertinentes
}

@Override
public Inquilinos inquilinosGet(Integer id) {
    // Código java de la consulta a la bd y gestiones pertinentes
}

@Override
public String inquilinosIns(Inquilinos inquilino) {
    // Código java de la consulta a la bd y gestiones pertinentes
}

@Override
public String inquilinosDel(List<Integer> ids) {
    // Código java de la consulta a la bd y gestiones pertinentes
}

@Override
public String inquilinosUpd(Inquilinos inquilino) {
    // Código java de la consulta a la bd y gestiones pertinentes
}
}
```