Manejo de conectores.

# 1.- Introducción.

Tradicionalmente, la programación de bases de datos ha sido como una torre de Babel: gran cantidad de productos de bases de datos en el mercado y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC** (**Java Database Connectivity**), 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. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

* Ser un **API con soporte de SQL**: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java.
* **Aprovechar** la experiencia de los API's de bases de datos existentes.
* **Ser lo más sencillo posible**.

Texto

Descripción generada automáticamente

Desfase objeto-relacional

El desfase o impedancia objeto-relacional es la **diferencia de aspectos que existen entre la POO y la base de datos**. Diferencias como:

* **Lenguaje de programación**: el programador debe aprender el lenguaje de POO y el lenguaje de acceso a datos.
* **Tipos de datos**: las BB.DD relacionales tienen restricciones en el uso de tipos, mientras que en la POO se pueden utilizar tipos de datos más complejos.
* **Paradigma de programación**: en el proceso de diseño y construcción del software se tiene que hacer una **traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R)** lo que implica que se tengan que **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 de negocio.

**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.

Imagina tener que almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

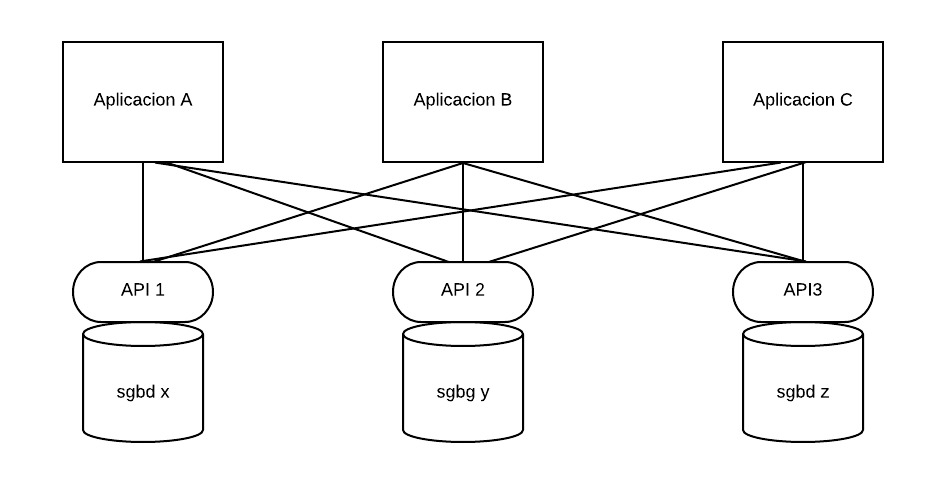
Este conjunto de dificultades técnicas son las que denominamos **Impedancia o desfase objeto-relacional**.

**Ejemplo**: 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.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

# 2.- Protocolos de acceso a bases de datos.

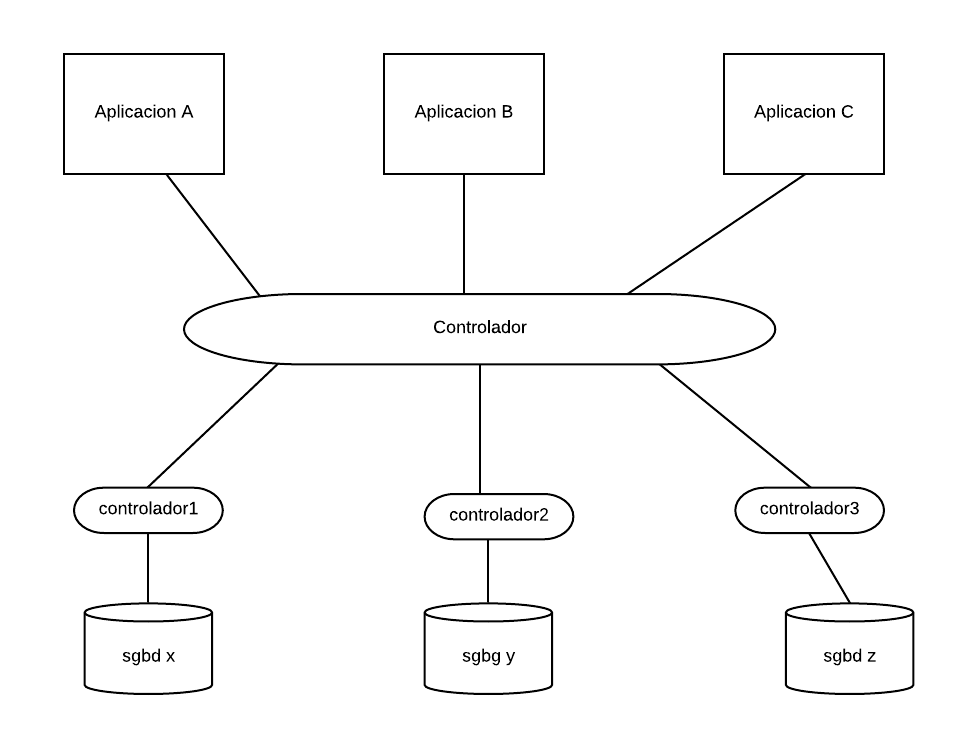
Inicialmente, cada empresa desarrolladora de un SGBD implementaba soluciones propietarias específicas para su sistema, pero pronto se dieron cuenta de que colaborando conjuntamente podían sacar mayor rendimiento y avanzar mucho más rápidamente. **Aquí cada SGBD tenía su propia conexión y su propio API**.

La llegada del **ODBC** (Open DataBase Conectivity) representó un avance sin precedentes en el camino hacia la interoperabilidad entre bases de datos y lenguajes de programación.

La mayoría de las empresas de SGBD incorporaron **drivers de conectividad** a las utilidades de sus sistemas y los lenguajes de programación desarrollaron bibliotecas específicas para soportar el API ODBC.

Sin embargo, aunque ODBC fue bien recibido para el acceso a bases de datos en Windows, no se introdujo bien en Java debido a su complejidad, lo que, entre otras cosas, ha impedido su transición fuera del entorno Windows.

**Sistema de conexión ODBC configurado usando diferentes drivers y un API estándar**.



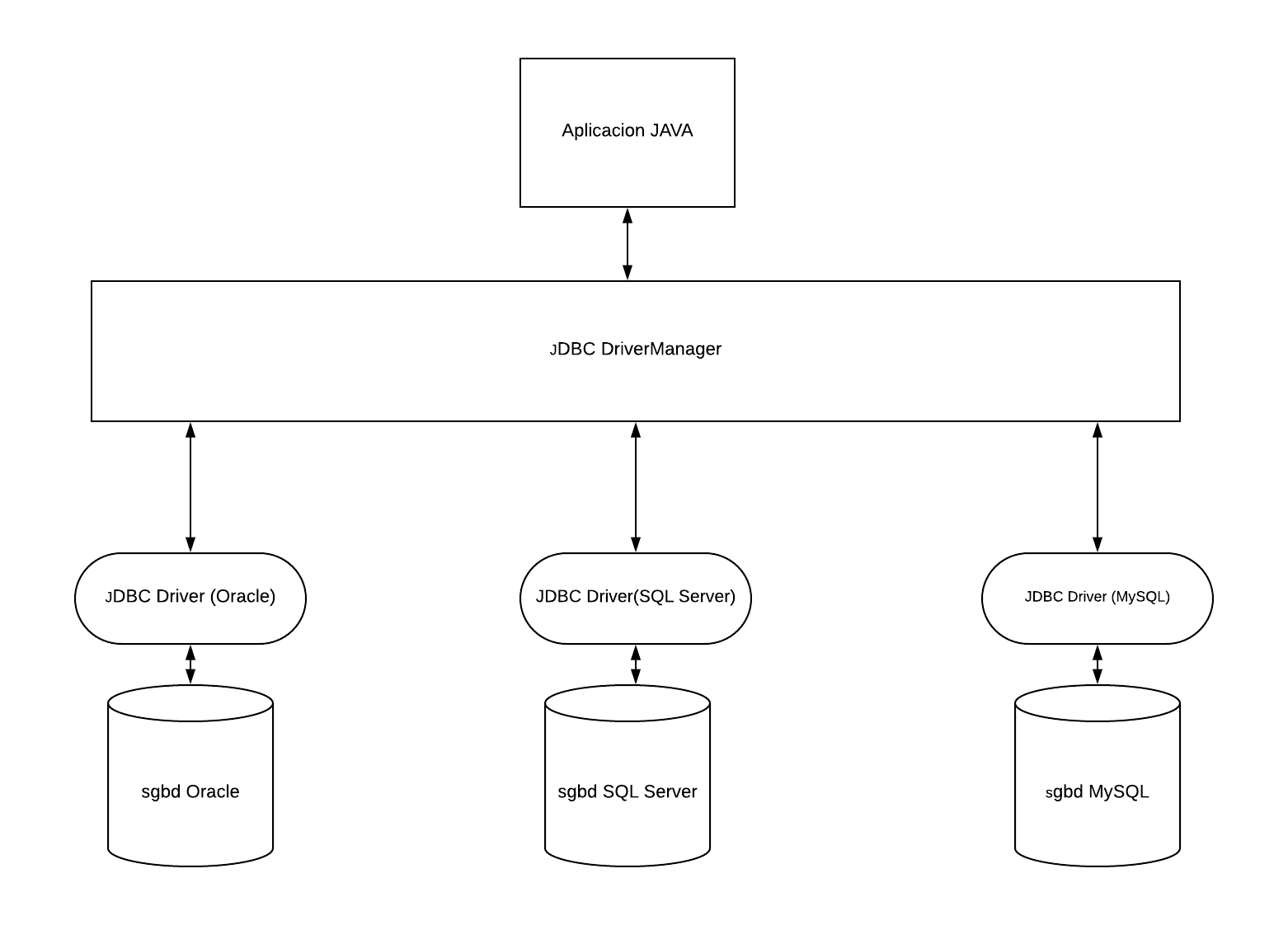
**JDBC (Java Database Connectivity)** se trata de un API bastante similar a **ODBC** en cuanto a funcionalidad, implementado y adaptado específicamente para Java.

La **funcionalidad** se encuentra **encapsulada en clases** (ya que Java es un lenguaje totalmente orientado a objetos) y además, **no depende de ninguna plataforma específica**, de acuerdo con la característica multiplataforma defendida por Java.

Sun desarrolló este API **(java.sql.\*)**para el acceso a bases de datos, con tres objetivos principales en mente:

* Ser un **API** con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java.
* Aprovechar la experiencia de los API's de bases de datos existentes.
* Ser lo más sencillo posible.

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



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.

A medida que cada controlador (de cada BB.DD) se carga en una Máquina Virutal de Java (JVM), **se registra con el DriverManager JDBC**. La clase **JDBC DriverManager** es responsable de localizar un controlador JDBC que necesita la aplicación. Las solicitudes de conexión a la BB.DD se especifican mediante una URL. Por ejemplo, para conectarnos a una BB.DD de Oracle, sería: **jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE**

Si el intento de conexión falla (no se encuentra ningún controlador que acepte la URL de conexión), el **Controlador** lanzará una **SQLException** a la aplicación. Si la conexión se completa con éxito, el controlador crea un objeto de conexión y lo devuelve a la aplicación.

# 3.- Conectores o Drivers.

Un conector o **driver** es un conjunto de **clases encargadas de implementar los interfaces del API** y acceder a la base de datos.

Un conector suele ser un fichero **.jar** que contiene una implementación de todas las interfaces del API JDBC. El conector lo proporciona el fabricante de la base de datos o bien un tercero.

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, y no dependa del driver en específico, puesto que trabajará con 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**:

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();

}

}

}

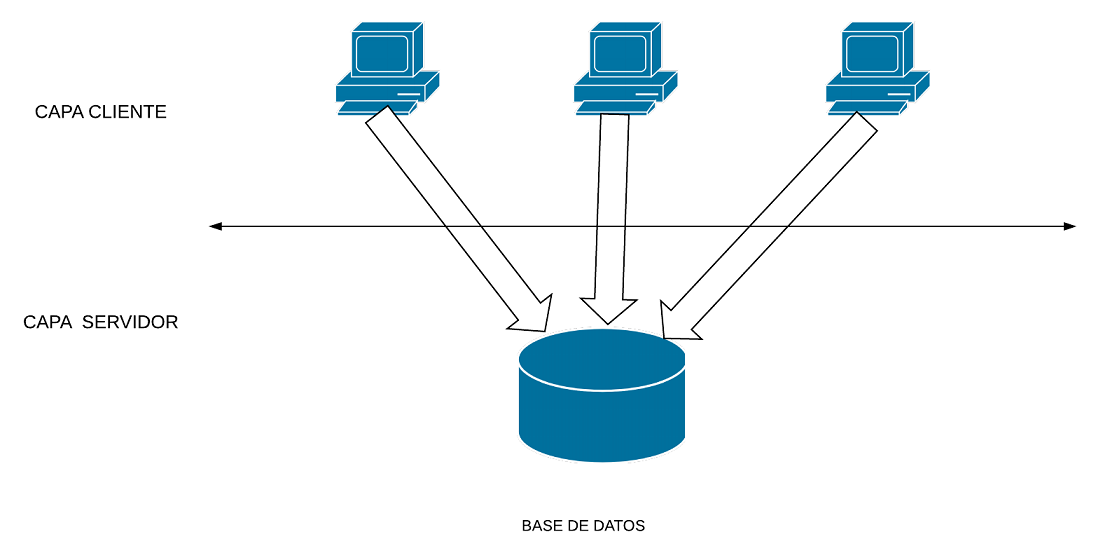
En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level (ANSI SQL-2 se refiere a los estándares adoptados por el American National Standards Institute en 1992. Entry Level se refiere a una lista específica de capacidades de SQL.) Los desarrolladores de drivers pueden establecer que sus conectores conocen estos estándares.

Hay **cuatro tipos de drivers JDBC**: Tipo 1, Tipo 2, Tipo 3 y Tipo 4, que veremos en apartados posteriores.

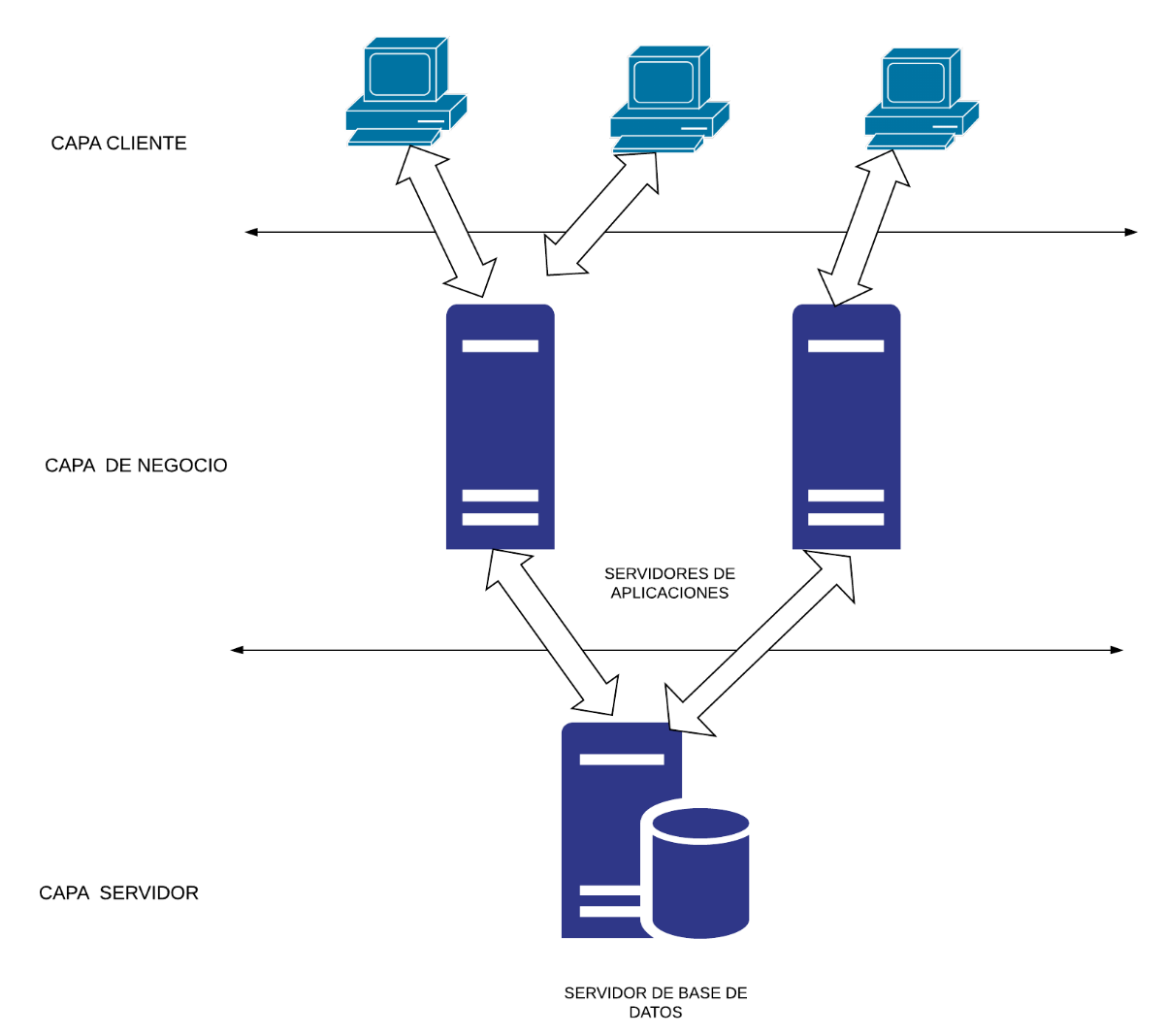
**Para saber más**: [base de datos con información sobre JDBC y tecnologías de base de datos de Oracle](https://www.oracle.com/java/technologies/javase/javase-tech-database.html).

# 4.- Arquitectura JDBC.

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



**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.



Hoy en día, hay cinco tipos de controladores JDBC en uso:

* **Tipo 1**: puente JDBC-ODBC.
* **Tipo 2**: controlador parcial de Java.
* **Tipo 3**: controlador Java puro para middleware de base de datos.
* **Tipo 4**: controlador Java puro para directo a base de datos.
* **Tipo 5**: controladores altamente funcionales con un rendimiento superior.

**Para saber más**: [introducción a JDBC](https://www.adictosaltrabajo.com/2006/05/04/introjdbc/).

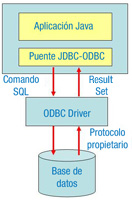
Texto

Descripción generada automáticamente

## 4.1.- Tipo 1 y 2.

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

|  |  |
| --- | --- |
| **Ventajas** | * **No se necesita un driver específico** de cada base de datos de tipo ODBC. * Está **soportado por muchos fabricantes**, por lo que tenemos **acceso a muchas Bases de Datos**. |
| **Desventajas** | * Hay **plataformas que no lo tienen implementado**. * 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** (Data Source Names, Nombres de fuentes de datos). |

El driver JDBC-ODBC Bridge **traduce las llamadas JDBC a llamadas ODBC** y las envía a la fuente de datos ODBC.

Este tipo de driver viene incluido en el JDK.

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

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. (Modelo de dos capas).

|  |  |
| --- | --- |
| **Ventajas** | * Rendimiento notablemente mejor que el JDBC-ODBC Bridge. |
| **Desventajas** | * La librería de las bases de datos 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[[1]](#footnote-1), por lo que son más eficientes.

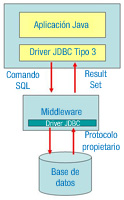
**Para saber más**: sobre [JNI](https://es.wikipedia.org/wiki/Java_Native_Interface) y [DSN](https://www.youtube.com/watch?v=s9rzjZ-ocg0) (DNS).

Texto

Descripción generada automáticamente

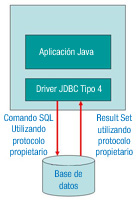
## 4.2.- Tipo 3 y 4.

**Tipo 3**: JDBC-Net pure Java driver

Las peticiones JDBC a la base de datos se pasan a través de la red al servidor de la capa intermedia ([middleware](https://ead.murciaeduca.es/pluginfile.php/1431852/mod_resource/content/6/AD03_Contenidos_Web/42_conectores_tipo_3_y_tipo_4.html#ta5b93431-18f9-be81-3428-06865a4a5da8)). Este servidor traduce este protocolo independiente del sistema gestor 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**.

Este driver está **basado en servidor**, por lo que no se necesita ninguna librería de base de datos en las máquinas clientes.  
Normalmente, un driver de tipo 3 proporciona soporte para balanceo de carga[[2]](#footnote-2), funciones avanzadas de administrador de sistemas tales como auditoría, etc.

**Tipo 4**: protocolo nativo.Est

Son conectores que **convierten directamente las llamadas JDBC al protocolo de red** usado por el SGBD.

Permite una **llamada directa desde la máquina cliente al servidor del SGBD**, y es una solución excelente para intranets.

|  |  |
| --- | --- |
| **Ventajas** | * No necesita traducción adicional o capa middleware, lo que **mejora el rendimiento**, siendo incluso mejor que en los tipos 1 y 2. * No necesita instalar **ningún software especial en el cliente o en el servido**r. |
| **Desventajas** | * Se necesita un driver diferente para cada base de datos. |

**Ejemplo de este tipo de conector**: Oracle Thin.

**Para saber más**: [sobre Oracle Thin y otros conectores de Oracle](http://web.archive.org/web/20120618210436/http:/www.devjoker.com/html/Como-conectar-a-ORACLE-con-Java-.html).

# 5.- Conexión a una BB.DD.

En Java, para establecer una conexión con una base de datos 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.
* **Devuelve un objeto *Connection***, que representa la conexión con la BB.DD

**Para saber más**: [Como instalar MySQL](https://www.ajpdsoft.com/), [Instalación y uso de MySQL Workbench](http://web.archive.org/web/20171117214534/http:/www.ecodeup.com/aprende-a-instalar-mysql-y-mysql-workbench-en-windows-10/).

## 5.1.- 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. Pero **normalmente es el administrador de la base de datos**, **a través de las herramientas del SGBD**, 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**. Aunque la mayoría de SGBD (como MySQL u Oracle) admiten esta sentencia desde línea de comandos.

Así pues, **mediante JDBC podemos conectarnos y manipular bases de datos**: crear tablas, modificarlas, borrarlas, añadir datos en las tablas, etc. Pero 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.

**Debes conocer**: si tienes olvidado SQL, [familiarízate con él](https://www.w3schools.com/sql/).

**Recomendación**: si necesitas [refrescar el concepto de clave primaria](https://es.wikipedia.org/wiki/Clave_primaria), en la Wikipedia puedes consultarlo. También puedes ver un [vídeo sobre la instalación de la BB.DD Oracle Express](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%203/Apuntes/AD03/web/html/ad03_descripcion_video_instalacion_de_oracle_express_y_creacion_de_tablas.html) y creación de tablas.

## 5.2.- Drivers de conexión (Oracle y MySQL).

### 5.2.1.- Drivers de MySQL.

MySQL **proporciona controladores basados ​​en estándares para JDBC, ODBC y ADO.Net**, lo que permite a los desarrolladores crear aplicaciones de base de datos en el lenguaje de programación de su elección. Puedes descargarlos en la [página de MySQL](https://www.mysql.com/products/connector/).

**MySQL** implementa el driver JDBC mediante su **conector MySQL Connector/ Connector / J** (se le conoce de las tres maneras). Además de implementar el JDBC, implementa una serie de extensiones de valor añadido al driver. También soporta el nuevo X DevAPI[[3]](#footnote-3).

J es un driver 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** (persistence frameworks) como Hibernate, las plantillas JDBC de Spring o el mapeo de sentencias SQL de MyBatis para reducir la cantidad de código JDBC para que pueda depurar, ajustar, proteger y mantener.

### 5.2.2.- Drivers de Oracle.

**JDBC Thin Driver utiliza sockets** de Java para conectarse directamente a Oracle. Proporciona su propia versión TCP/IP del protocolo SQL \* Net. Como es Java 100%, este controlador es independiente del sistema operativo y también puede ejecutarse desde un navegador web. Los formatos URL:

* **SID**: (ya no se recomienda para uso de Oracle).

jdbc: oracle: thin: [<user> / <password>] @ <host> [: <port>]: <SID>

* **Servicios**:

jdbc: oracle: thin: [<user> / <password>] @ // <host> [: <port>] / <service>

* **TNSNames**:

jdbc: oracle: thin: [<user> / <password>] @ <TNSName>

**JDBC OCI** (Oracle Call Interfaces) **funciona a través de SQL \* Net**. Los controladores OCI de JDBC le permiten **llamar al OCI directamente desde Java**, lo que proporciona un alto grado de **compatibilidad con una versión específica de Oracle**. Debido a que utilizan métodos nativos, son **específicos de la plataforma**.

jdbc: oracle: oci: @myhost: 1521: orcl

**JDBC KPRB driver** (default connection).

Este controlador de Oracle se utiliza principalmente para escribir procedimientos almacenados de Java y JSP (JavaServer Pages) almacenados. **Utiliza la sesión de base de datos predeterminada/actual** y, por lo tanto, no requiere un nombre de usuario, contraseña o URL de base de datos adicional.

Se puede obtener un identificador de la conexión predeterminada o actual (controlador KPRB) llamando al método **OracleDriver.defaultConnection ().**

**Para saber más**: [información sobre los tres drivers de Oracle](http://www.orafaq.com/wiki/JDBC).

## 5.3.- Instalar el conector de la base de datos.

### 5.3.1.- MySQL.

El 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.

Puedes seguir este [Tutorial para descargar e instalar el driver en Eclipse](https://rodin.uca.es/bitstream/handle/10498/18602/incorporacion-driver-JDBC-eclipse_Gamaza_Otte.pdf). O bien puedes seguir el [vídeo presentación siguiente](https://www.youtube.com/watch?v=cFLsynl91B0). Como verás, tan solo consiste en descargar un archivo, descomprimirlo y desde Eclipse, añadir el archivo **.jar**. **Es importante que compruebes el driver más adecuado para trabajar con tu servidor MySQL**.

**Nota**: en sistemas antiguos, para que **DriverManager** tuviera “registrados” los drivers, era necesario cargar la clase en la VM. 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()**, solo que estén en el classpath de la JVM.

**Ejemplo**:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class PrimeraConexion {

public static void main(String[] args) {

try {

// Establecemos la conexion 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();

}

}

}

### 5.3.2.- Oracle.

Se hace de la misma forma que en MySQL:

* Utilizaremos el driver ***ojdb8.jar*** que podemos descargar de la página de Oracle. Tendremos que configurar en Eclipse la librería en el ***Build Path***.
* **La cadena de conexión recomendada** es la que utiliza el ServiceName:
  + **host:port:service\_name**

El archivo ***Tnsnames.ora*** proporciona **información de las conexiones de la base de datos** que estas **utilizando**, la ruta es: **Oracle\_Home\app\oracle\product\11.2.0\server\network\ADMIN.**

En este caso el TNSNames.ora tiene una entrada correspondiente al servicio XE que equivale a la base de Datos que se va a usar:

XE =

(DESCRIPTION =

(ADDRESS = (PROTOCOL = TCP)(HOST = PC)(PORT = 1521))

(CONNECT\_DATA =

(SERVER = DEDICATED)

(SERVICE\_NAME = XE))

)

**Ejemplo** de conexión para esta base de datos:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class PrimeraConexion {

     public static void main(String[] args) {

          try {

               // Establecemos la conexion con la BD

               String urljdbc = "jdbc:oracle:thin:@localhost:1521:XE";

               conexion = DriverManager.getConnection(urljdbc, "ejemplo", "ejemplo");

               System.out.println("conecta");

               conexion.close(); // Cerrar conexión

          } catch (SQLException cn) {

               cn.printStackTrace();

          }

     }

}

## 5.4.- Conexión de la 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. Los objetos **Connection** mantendrán la capacidad de comunicarse con el SGBD mientras permanezcan abiertos. Para cerrarlos, se utiliza el método ***close()***.

**Método getConnection()**: El **objeto Connection** está totalmente vinculado a una fuente de datos. Por eso, con ***getConnection()*** (clase DriverManager) le especificamos de qué fuente se trata e indicarle una URL con los datos de acceso (usuario y 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.**

**¿Y cómo especifico la URL?** Por ejemplo:

jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE

* **Primer valor**: es el protocolo, que siempre es jdbc para los URL JDBC.
* **Segundo valor**: subprotocolo.
* **Tercer valor**: nombre del sistema para establecer la conexión con un sistema específico, dependerá del:
  + tipo de driver utilizado
  + del *host* donde se aloje el SGBD
  + del puerto que este use para escuchar sus peticiones
  + del nombre de la BB.DD o esquema con el que queremos trabajar.

Existen **dos valores especiales que pueden utilizarse para conectarse con la base de datos local**. Son ***\*LOCAL*** y ***localhost*** (ambos son sensibles a mayúsculas y minúsculas). También puede suministrarse un nombre de sistema específico, de la forma siguiente:

Prueba el siguiente ejemplo en un nuevo proyecto, configurar tu proyecto para incluir las bibliotecas JDBC de Oracle en la construcción de su ruta.

**proyecto --> boton derecho --> Build Path --> configure Build Path . Añade ojdbc6.jar**

[Para obtener mas información de los drivers de Oracle disponibles](https://docs.oracle.com/cd/E11882_01/appdev.112/e13995/oracle/jdbc/OracleDriver.html)  y de la sintaxis de url correspondiente a cada uno de ellos.

**Ejemplo de conexión a BB.DD Oracle**:

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class PrimeraConexion {

public static void main(String[] args) {

Connection conexion;

try {

//url oracle con formato jdbc:oracle:thin:@<hostname>:<port>:<servicename>

//urll con formato jdbc:oracle:thin:<usuario>@<hostname>:<port>:<servicename>

String urljdbc = "jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE";

//establecemos la conexion utilizando el metodo getConnection

// damos url, nombre del usuario y contraseña

conexion = DriverManager.getConnection(urljdbc, "ejemplo", "ejemplo");

System.out.println("conecta");

conexion.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

**Ejemplo de conexión a BB.DD 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.**

[Para obtener mas información de los drivers de mysql disponibles](https://www.mysql.com/products/connector/) 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 conexion con la BD

Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "root", "system");

// Cerrar conexión

conexion.close();

}

}

## 5.5.- Realización de consultas.

Para operar con una BB.DD mediante ejecución de consultas, nuestra aplicación deberá:

1. **Cargar el driver**.
2. **Establecer la conexión** con la BB.DD
3. **Enviar consultas** y **procesar** el resultado.
4. **Liberar los recursos**.
5. **Gestionar los errores** que se puedan producir.

Para hacer consultas en la BB.DD, tendremos que:

1. crear un objeto de tipo **Statement**
2. ejecutar la consulta con el método **executeQuery(sql)**
3. recorrer el **ResultSet** que devuelve este método.

**Ejemplo 1**:

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 conexion 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 resul = sentencia.executeQuery(sql);

               // Recorremos el resultado para visualizar cada fila

               // Se hace un bucle mientras haya registros y se van visualizando

               while (resul.next()) {

                    System.out.printf("%d, %s, %s %n", resul.getInt(1), resul.getString(2), resul.getString(3));

               }

               resul.close(); // Cerrar ResultSet

               sentencia.close(); // Cerrar Statement

               conexion.close(); // Cerrar conexión

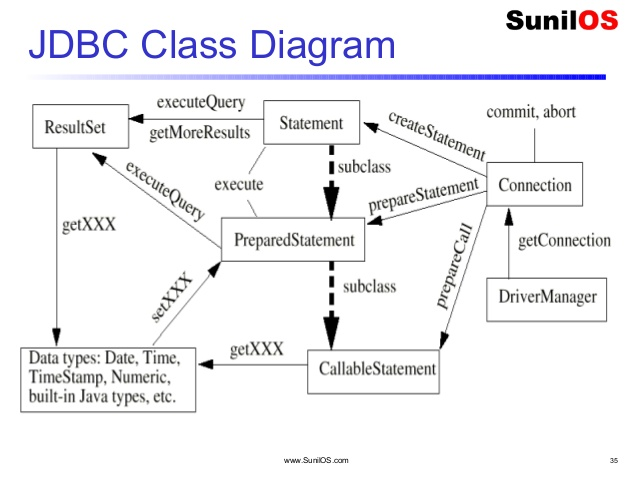
          } catch (SQLException cn) {

               cn.printStackTrace();

          }

     }

}



EL API JDBC distingue dos tipos de sentencias SQL:

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

**Ejemplo 2**: consultas **SELECT**. Uso del **ResultSet**: se acceden a los datos a través de un cursor que apunta a una fila de datos en dicho objeto **ResultSet**. Inicialmente, **el cursor se coloca antes de la primera fila**. Accedemos a todas las filas de una en una llamando al método ***next()***.

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:ejemplo/ejemplo@localhost:1521:XE";

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));

resul.close(); // Cerrar ResultSet

smt.close(); // Cerrar Statement

conexion.close(); // Cerrar conexión

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Ejemplo 3**: sentencias DML (UPDATE, DELETE, INSERT). Aunque son 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. A partir de una conexión activa, **instanciamos un objeto** **Statement.**
2. **Ejecutamos la sentencia** SQL, método **executeUpdate.**
   1. El método **executeUpdate** devuelve un nº entero que representa el nº de filas afectadas por la instrucción SQL.
3. **Cerramos** del objeto **Statement** instanciado.
4. **Cerramos** la **conexión** activa.

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 conexion

Connection conexion;

try {

String urljdbc = "jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE";

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

// cierra la conexion

conexion.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**Para saber más**: [sobre consultas preparadas](https://www.dataprix.com/ca/node/941).

## 5.6.- Ejecución de sentencias DML.

Las sentencias DML (Lenguaje de manipulación de datos), incluyen: INSERT, DELETE y UPDATE. El funcionamiento de todas ellas es bastante similar.

**Ejemplo**:

import java.sql.\*;

public class DMLDeletePrep {

    public static void main(String[] args) {

        try {

            // Carga el driver

            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Conexion

            Connection conexion = DriverManager.getConnection("jdbc:odbc:ORACLE-XE", "scott", "tiger");

            // PreparedStatement

            String sql = "DELETE FROM dept WHERE DEPTNO=?";

            System.out.println(sql);

            PreparedStatement sentencia = conexion.prepareStatement(sql);

            String deptno = args[0];

            sentencia.setInt(1, Integer.parseInt(deptno));

            // Ejecuta la sentencia

            int filas = sentencia.executeUpdate();

            System.out.println("Filas afectadas: " + filas);

            // Cerrar ResultSet

            sentencia.close();

            // Cerrar conexión

            conexion.close();

        } catch (ClassNotFoundException cn) {

            cn.printStackTrace();

        } catch (SQLException e) {

            e.printStackTrace();

        }

    }

}

## 5.7.- Ejecución de procedimientos almacenados en la BB.DD.

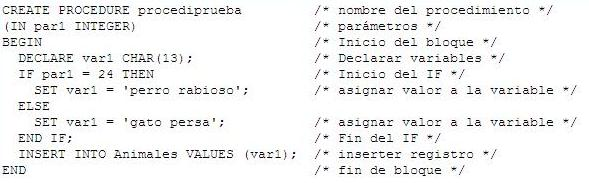
Un procedimiento almacenado es un procedimiento o subprograma que está almacenado en la base de datos.

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

Además, **estos procedimientos suelen ser de dos clases**:

* [**Procedimientos**](https://ead.murciaeduca.es/pluginfile.php/1431852/mod_resource/content/6/AD03_Contenidos_Web/57_ejecucin_de_procedimientos_almacenados_en_la_base_de_datos.html#t6afbc7ee-c074-9c77-b539-6e8c5f7a3efb) almacenados.
* **Funciones**, las cuales devuelven un valor que se puede emplear en otras sentencias SQL.

**Ejemplo** para crear un procedimiento almacenado sencillo para MySQL.



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

**Debes conocer**: en el capítulo 19 del [manual de referencia de MySQL](https://dev.mysql.com/doc/refman/8.0/en/) que puedes encontrar en el enlace siguiente, puedes familiarizarte con los comandos que puedes necesitar para realizar procedimientos almacenados y funciones.

### 5.7.1.- Ejecución de procedimientos almacenados en la BB.DD MySQL.

**Ejemplo**: procedimiento almacenado en MySQL que inserta datos en la tabla clientes. Desde el programa Java que realizamos, llamaremos para ejecutar a ese procedimiento almacenado. Por tanto, ¿cuál sería la secuencia que seguiríamos para realizar esto?

* **Si no tenemos creada la tabla de clientes, la creamos**.

delimiter $$

CREATE TABLE `clientes` (

`Cod\_Cliente` int(3) NOT NULL DEFAULT '0',

`Nombre` tinytext,

`Telefono` tinytext,

PRIMARY KEY (`Cod\_Cliente`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1$$

* **Creamos el procedimiento**:

-- --------------------------------------------------------------------------------

-- Routine DDL

-- --------------------------------------------------------------------------------

DELIMITER $$

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

* **Crear la clase Java para desde aquí**, llamar al procedimiento almacenado.
  + [Ejemplo](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%203/Apuntes/AD03/web/zip/ad03_cont_r28_02_inserconprocalma.zip).

try {

            // Cargar el driver de mysql

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

            // 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("Exceción: "+ cE.toString());

        }

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

SQL Exception:

com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException:

Duplicate entry '765' for key 'PRIMARY'

**Recomendación**: ve la [documentación siguiente](https://users.dcc.uchile.cl/~lmateu/CC60H/Trabajos/jfernand/). Tienes ejemplos JDBC para diversas finalidades: listar datos de una base de datos, llamar a funciones en Oracle, etc. [Wayback Machine (archive.org)](http://web.archive.org/web/20130613140711/https:/www.sicuma.uma.es/sicuma/Formacion/documentacion/JDBC-Oracle.pdf)

**Para saber más**: documento bastante más amplio que el anterior, sobre JDBC con Oracle, está [aquí](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%203/Recursos/JDBC-Oracle-120pags.pdf).

### 5.7.2.- Ejecución de procedimientos almacenados en la BB.DD Oracle.

Teniendo un procedimiento almacenado como el siguiente:

CREATE OR REPLACE PROCEDURE remove\_emp (employee\_id NUMBER) AS

tot\_emps NUMBER;

BEGIN

DELETE FROM employees

WHERE employees.employee\_id = remove\_emp.employee\_id;

tot\_emps := tot\_emps - 1;

END;

/

**Para ejecutar el procedimiento de arriba, haríamos**:

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class EjemploProcAlmacenado1 {

public static void main(String[] args) {

Connection cn = null;

try {

// Carga el driver de oracle

DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

// Conecta con la base de datos XE con el usuario scott y la contraseña tiger

cn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "scott", "tiger");

// Llamada al procedimiento almacenado

CallableStatement cst = cn.prepareCall("{call remove\_emp (104)}");

// Ejecuta el procedimiento almacenado

cst.execute();

System.out.println("borrado");

cst.close();

} catch (SQLException ex) {

System.out.println("Error: " + ex.getMessage());

} finally {

try {

cn.close();

} catch (SQLException ex) {

System.out.println("Error: " + ex.getMessage());

}

}

}

}

En el código:

* Las llamadas a los procedimientos almacenados, al igual que las **PreparedStatements** y las consultas simples se hacen sobre la conexión, en este caso con el método **prepareCall()**, que nos devuelve un **CallableStatement**.
* La llamada al procedimiento almacenado, además de ir entre comillas por ser un string tiene que ir también entre llaves y tiene el siguiente formato: “{call nombre\_procedimiento(params)}”
* El procedimiento se ejecuta cunado llamamos al **método execute**, y como es lógico, en el momento en el que se ejecute, tienen que estar definidos todos los parámetros tanto de entrada como de salida.

## 5.8.- Sentencias preparadas.

***PreparedStatement*** deriva de la clase más general ***Statement***.

Una ***Prepared Statement*** es una **sentencia** SQL de base de datos **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 se usa**:

1. **Simplemente ponemos interrogantes en donde irán los valores concretos** que vayamos a insertar. Cada interrogante se identifica luego con un número:
   * El primer parámetro es el nº del interrogante,
   * Y el segundo el valor que queremos insertar.
2. Luego con los métodos ***set()*** correspondientes (en función del tipo de dato), **rellenamos estos valores**.
3. Finalmente, llamamos a ***executeUpdate()***.
4. Con la misma PreparedStatement, podemos poner otros valores volviendo a llamar a los métodos ***set()*** y podríamos volver a hacer una actualización con ***executeUpdate()***.

No hay que preocuparse de hacer casting ni conversiones de ningún tipo, Los métodos ***setInt()***, ***setString()***, etc., ya se encargan de hacer todas las conversiones adecuadas para nosotros.

**Ejemplo**: utilizamos un único objeto ***PreparedStatement***, 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();

}

}

**Métodos que permiten pasar parámetros a la sentencia**:

* **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)
* etc...

## 5.9.- Captura de errores y liberación de recursos.

**Liberación de recursos**:

Las conexiones con una base de datos consumen muchos recursos en el sistema gestor. Por ello, 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**).

**Gestión de errores**:

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

**Esta excepción se puede producir tanto en el driver, como en la BB.DD**. Cuando se produce una excepción de este tipo, un objeto de tipo ***SQLException*** se pasará a la cláusula *Catch*.

El objeto **SQLException** pasado tiene los siguientes métodos disponibles para recuperar información adicional sobre la excepción:

* getErrorCode ()
* getMessage ()
* printStackTrace ()
* etc.

[Excepciones.doc (ugr.es)](http://elvex.ugr.es/decsai/java/pdf/B2-excepciones.pdf)

## 5.10.- Transacciones.

Cuando tenemos una serie de consultas SQL que deben ejecutarse en conjunto, con el uso de **transacciones** podemos **asegurarnos de que nunca nos quedaremos a medio camino de su ejecución**.

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

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

UPDATE cuentas SET saldo = saldo - cantidad WHERE cliente = “Antonio”;

UPDATE cuentas SET saldo = saldo + cantidad WHERE cliente = “Pedro”;

Pero, ¿qué ocurre si por algún imprevisto (un apagón de luz, etc.), el sistema “cae” después de que se ejecute la primera consulta, y antes de que se ejecute la segunda? 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.

Texto

Descripción generada automáticamente

El **control de la transacción** es **realizado por el objeto de la conexión**. Cuando se crea una conexión, **por defecto están activadas las transacciones**. 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 transacccion:  **conexion.setAutoCommit(false)**
2. Ejecutamos las sentencias
3. Finalizaremos de forma manual la transacción.**conexion.Commit.**

**Ejemplo**:

El siguiente programa 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());

updateTotal.executeUpdate();

con.commit();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

if (con != null) {

try {

System.err.print("Transaction is being rolled back");

con.rollback();

} catch(SQLException excep) {

JDBCTutorialUtilities.printSQLException(excep);

}

}

} finally {

if (updateSales != null) {

updateSales.close();

}

if (updateTotal != null) {

updateTotal.close();

}

con.setAutoCommit(true);

}

}

**Para saber más**: documentación muy extensa para programar con PL-SQL: procedimientos, funciones, triggers, etc., en el siguiente enlace: [Microsoft Word - 040628\_PLSQL\_Basico.doc (pbworks.com)](http://apuntesduoc.pbworks.com/w/file/fetch/54222708/040628_PLSQL_Basico.pdf). Otro interesante tutorial sobre transacciones y otras cuestiones con MySQL: [Microsoft Word - CursoMysql\_ses2.doc (uv.es)](https://www.uv.es/~jgutierr/MySQL_Java/CursoMysql_ses2.pdf)

## 5.11.- 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 **métodos** de la interfaz ***DatabaseMetaData*** **devuelven listas de información** en forma de objetos **ResultSet**, que se pueden recorrer en un bucle para obtener los **metadatos**. Los métodos **getString,  getInt**, etc, nos permitirán sacar esa información. [Consulta la documentación de la API JDBC.](https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html)

Algunos métodos **DatabaseMetaData** tienen **argumentos** que son **patrones**. Por ejemplo: "%" representa  cualquier subcadena de 0 o más caracteres, y "\_" representa un carácter.

**Algunos métodos útiles son**:

* getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)
  + Devuelve en un **ResultSet** la **descripción de las columnas de la tabla** disponibles en el catálogo especificado.
* getDriverName()
  + Devuelve en un ***String*** el nombre del controlador JDBC.
* getExportedKeys(String catalog, String schema, String table)
  + Devuelve en un **ResultSet** la descripción de las columnas de clave **Foránea** que hacen referencia a las columnas de clave principal de la tabla dada.
* getFunctions(String catalog, String schemaPattern, String functionNamePattern)
  + Devuelve en un **ResultSet** la **descripción del sistema** y las **funciones de usuario** disponibles en el catálogo dado.
* getMaxConnections()
  + Recupera el número máximo de conexiones simultáneas a esta base de datos que son posibles.
* getPrimaryKeys(String catalog, String schema, String table)
  + Devuelve en un **ResultSet**  una descripción de las columnas de clave primaria de la tabla dada.
* getProcedures(String catalog, String schemaPattern, String procedureNamePattern)
  + Recupera una descripción de los procedimietos almacenados disponibles en el catálogo dado.
* getSchemas()
  + Devuelve en un **ResultSet** los nombres de esquema disponibles en esta base de datos.
* getSQLKeywords()
  + 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.
* getStringFunctions()
  + Recupera una lista separada por comas de funciones de cadena disponibles con esta base de datos.
* supportsAlterTableWithAddColumn()
  + Recupera si esta base de datos admite **ALTER TABLE** para añadir columna. Devuelve un booleano
* supportsAlterTableWithDropColumn()
  + Recupera si esta base de datos es posible borrar una columna con ALTER **TABLE**. Devuelve un booleano.
* supportsTransactions()
  + Recupera si esta base de datos admite transacciones.

**Ejemplo:**

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());

}

}

}

## 5.12.- Recuperación y modificación de valores de ResultSet.

Un  objeto **ResultSet**es una tabla de datos que representa un conjunto de resultados de base de datos. Puede ser creado a través de cualquier objeto que implementa el interfaz **Statement**, incluyendo**PreparedStatement, CallableStatement**.

Es necesario un bucle para iterar a través de todos los datos en el**ResultSet**, utilizando el método ***ResultSet.next()*** se hace avanzar al cursor a la siguiente fila.

Los objetos **ResultSet** predeterminados no permiten la  actualización, además, solo se puede mover el cursor hacia adelante. Sin embargo, se pueden crear objetos **ResultSet** que se puedan **desplazar** hacia adelante y hacia atrás o moverse a una posición absoluta y **actualizarse**.

Hacer un ResultSet actualizable (actualización de tablas):

conexion.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

Además, es necesario **actualizar el dato en la fila actual** del objeto ResultSet con ***updateRow()***:

resulSet.updateFloat( "PRICE", f \* percentage);//recupera la columna con nombre "PRICE"

resulSet.updateRow();

Crear ResultSet recorrible en cualquier dirección y acceso mediante posición absoluta:

Con la opción **ResultSet.TYPE\_SCROLL\_SENSITIVE**.

**Ejemplo**:

public void modifyPrices(float percentage) throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement();

stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM COFFEES");

while (uprs.next()) {

float f = uprs.getFloat("PRICE"); //recupera la columna con nombre "PRICE"

uprs.updateFloat( "PRICE", f \* percentage); //también se puede utilizar updateFloat( 3, f \* percentage)

uprs.updateRow();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

**Para saber más**: de los [tipos de ResultSet](https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html#rs_update) que se pueden utilizar en Oracle.

**Ejemplo completo de uso ResultSet para actualización**:

import java.sql.\*;

import java.time.\*;

public class JDBCExample {

// JDBC driver name and database URL

static final String JDBC\_DRIVER = "com.mysql.jdbc.Driver";

static final String DB\_URL = "jdbc:oracle:thin:@localhost:1521:XE";

// Database credentials

static final String USER = "scott";

static final String PASS = "tiger";

public static void main(String[] args) {

Connection conn = null;

try {

// STEP 3: Open a connection

System.out.println("Connecting to database...");

conn = DriverManager.getConnection(DB\_URL, USER, PASS);

// STEP 4: Execute a query to create statment with

// required arguments for RS example.

System.out.println("Creating statement...");

Statement stmt = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);

// STEP 5: Execute a query

String sql = "SELECT EMPLOYEE\_ID, first\_name, last\_name, salary, email, hire\_date FROM Employees";

ResultSet rs = stmt.executeQuery(sql);

System.out.println("List result set for reference....");

printRs(rs);

// STEP 6: Loop through result set and add 5 in sal

// Move to BFR postion so while-loop works properly

rs.beforeFirst();

// STEP 7: Extract data from result set

while (rs.next()) {

// Retrieve by column name

int newAge = rs.getInt("salary") + 5;

rs.updateDouble("salary", newAge);

rs.updateRow();

}

System.out.println("List result set showing new salarys...");

printRs(rs);

// Insert a record into the table.

// Move to insert row and add column data with updateXXX()

System.out.println("Inserting a new record...");

rs.moveToInsertRow();

rs.updateInt("EMPLOYEE\_ID", 300);

rs.updateString("first\_name", "John");

rs.updateString("last\_name", "Paul");

rs.updateInt("salary", 40);

rs.updateString("email", "ana@ana");

rs.updateTimestamp("hire\_date", null);

// Commit row

rs.insertRow();

System.out.println("List result set showing new set...");

printRs(rs);

// Delete second record from the table.

// Set position to second record first\_name

rs.absolute(2);

System.out.println("List the record before deleting...");

// Retrieve by column name

int EMPLOYEE\_ID = rs.getInt("EMPLOYEE\_ID");

int age = rs.getInt("salary");

String first\_name = rs.getString("first\_name");

String last\_name = rs.getString("last\_name");

// Display values

System.out.print("EMPLOYEE\_ID: " + EMPLOYEE\_ID);

System.out.print(", salary: " + age);

System.out.print(", first\_name: " + first\_name);

System.out.println(", last\_name: " + last\_name);

// Delete row

rs.deleteRow();

System.out.println("List result set after deleting one records...");

printRs(rs);

// STEP 8: Clean-up environment

rs.close();

stmt.close();

conn.close();

} catch (SQLException se) {

// Handle errors for JDBC

se.printStackTrace();

} catch (Exception e) {

// Handle errors for Class.forName

e.printStackTrace();

} finally {

// finally block used to close resources

try {

if (conn != null)

conn.close();

} catch (SQLException se) {

se.printStackTrace();

} // end finally try

} // end try

System.out.println("Goodbye!");

}// end main

public static void printRs(ResultSet rs) throws SQLException {

// Ensure we start with first\_name row

rs.beforeFirst();

while (rs.next()) {

// Retrieve by column name

int EMPLOYEE\_ID = rs.getInt("EMPLOYEE\_ID");

int age = rs.getInt("salary");

String first\_name = rs.getString("first\_name");

String last\_name = rs.getString("last\_name");

// Display values

System.out.print("EMPLOYEE\_ID: " + EMPLOYEE\_ID);

System.out.print(", Salary: " + age);

System.out.print(", first\_name: " + first\_name);

System.out.println(", last\_name: " + last\_name);

}

System.out.println();

}// end printRs()

}// end JDBCExample

## 5.13.- Pool de conexiones.

En ocasiones, sobre todo cuando se trabaja en el ámbito de las aplicaciones distribuidas, en entornos web, es recomendable gestionar las conexiones de otro modo.

Veamos **cómo funciona la creación de conexiones** en una aplicación clásica de escritorio **cliente-servidor** y luego veamos los problemas se seguir con dicha estructura en una aplicación web.

Problemas en la creación de conexiones

En una app de escritorio, cada usuario que inicia la aplicación tiene una conexión con la BB.DD exclusiva para él. Mientras que, en web, cada usuario que se conecta a nuestra aplicación, se le crea una conexión y al salir de la aplicación, se cierra su conexión.

Si siguiéramos el mismo patrón de creación de conexiones de aplicaciones de escritorio en aplicaciones web acabaríamos con 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).

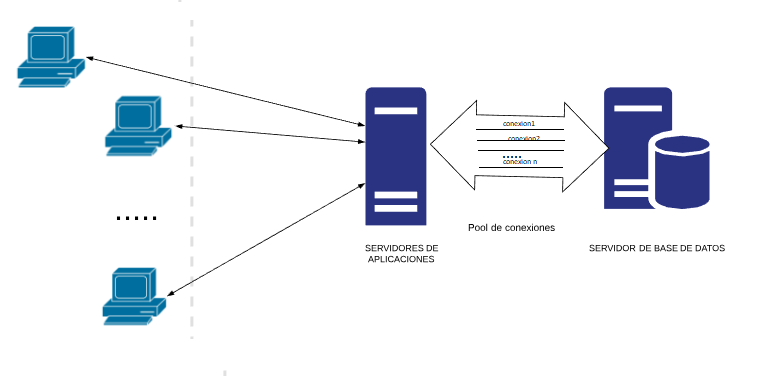
Por ende, el servidor de BB.DD acabaría cayendo debido a los recursos consumidos por todas las conexiones. Además, sería muy sencillo hacernos un ataque de denegación de servicio (DDoS)

Una solución que nos evitaría las conexiones sin usar sería que se creara la conexión al iniciar cada petición web y s ecerrara al finalizar dicha petición web. **¿El problema de eso?** Crear y cerrar una conexión es muy costoso. Tendríamos 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 tener tantas conexiones como usuarios ya que el número de usuarios es demasiado elevado.
* Cerrar la conexión.



Veamos cómo funciona

El servidor web tiene **“n” conexiones ya creadas y conectadas a la BB.DD** (Se llaman conexiones esperando).

* Cuando se lanzan peticiones web, la aplicación pide ese nº de conexiones al pool de conexiones.
  + Si se piden más conexiones de las que hay esperando en el pool se crearán en ese instante nuevas conexiones, con límite en el máximo que permita el pool.
* Cuando las peticiones web finalizan, las conexiones no se cierran, sino que se devuelven al pool indicándole que ya han acabado de usar las conexiones.
  + Si hay ya demasiadas conexiones esperando a ser usadas se cerrarán para ahorrar recursos en el servidor de base de datos.

Qué hemos conseguido con el pool

* Ahora las conexiones ya no se quedan abiertas, ya que cada conexión se pseudo-abre y se pseudo-cierra.
* No tenemos tantas conexiones como usuarios usan la aplicación, ya que solo se necesitan tantas como usuarios hay haciendo una petición en ese instante.
* Al iniciar un servidor Java EE (Enterprise Edition)[[4]](#footnote-4), automáticamente el pool crea un nº de conexiones físicas iniciales.

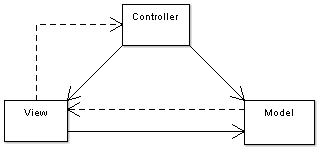
**Cuando un objeto Java del servidor J2EE necesita una conexión**, la solicita a través del método **dataSource.getConnection()**, la fuente de datos **javax.sql.DataSource**habla con el pool de conexiones y éste le entrega una conexión lógica**java.sql.Connection**. Esta conexión lógica la recibe, por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

**Cuando un objeto Java del servidor Java EE desea cerrar una conexión** a través del método ***connection.close()***, la fuente de datos ***javax.sql.DataSource*** habla con el pool de conexiones y devuelve la conexión lógica.

**Debes conocer**: [Pool de conexiones - ChuWiki (archive.org)](http://web.archive.org/web/20230201200039/http:/chuwiki.chuidiang.org/index.php?title=Pool_de_conexiones)

**Para saber más**: [BasicDataSource: Pool de conexiones (chuidiang.org)](https://old.chuidiang.org/java/mysql/BasicDataSource-Pool-Conexiones.php)

### 5.13.1.- Ejemplo de pool de conexiones MySQL.

El objetivo del proyecto siguiente es crear una aplicación grafica 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.

[(285) ▷ 13. ✅ CURSO JAVA: Cómo hacer un POOL de CONEXIONES?【Connection Pool Java】│Java Tutorial - YouTube](https://www.youtube.com/watch?v=pBXeGbsxk5Q)

[Ver código de ejemplo para MySQL](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%203/Resumen/ejemplo%20pool%20hilos%20mysql.java).

[Ver código de ejemplo por parte de ChatGPT](https://chat.openai.com/share/2c937051-ee84-40a2-b6ad-672704e660c1).

### 5.13.2.- Ejemplo de pool de conexiones Oracle.

[Ver código de ejemplo para Oracle](https://aula21-my.sharepoint.com/personal/5634224_alu365_murciaeduca_es/Documents/Documentos/FP/DAM/Curso%202/Materias/Acceso%20a%20datos/Evaluacion%201/Tema%203/Resumen/ejemplo%20pool%20hilos%20oraclejava.java).

[Ver código de ejemplo por parte de ChatGPT](https://chat.openai.com/share/1b43ba9c-006d-439d-af40-9b638e8e509e).

# 6.- Operaciones: ejecución de consultas.

**Resumen de lo que debe hacer nuestra aplicación para ejecutar consultas en la BB.DD**:

* **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.

**Tipos de sentencias que podemos utilizar**:

* **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.

Veamos a continuación cómo realizar una consulta en la base de datos que creamos en el apartado anterior, la base de datos **farmacia.mdb** (base de datos Access.

En este caso **utilizaremos** un **acceso mediante puente JDBC-ODBC**. Dicho puente da acceso a bases de datos ODBC.

Este **driver** está incorporado dentro de la distribución de Java, por lo que **no es necesario incorporarlo explícitamente en el**[**classpath**](https://ead.murciaeduca.es/pluginfile.php/1431852/mod_resource/content/6/AD03_Contenidos_Web/6_operaciones_ejecucin_de_consultas.html#t49f03823-b486-4831-4e78-815055f4f1d1) de una aplicación Java.

# 7.- Excepciones y cierre de conexiones.

Debemos tener en cuenta siempre que las **conexiones** con una base de datos consumen muchos recursos en el sistema gestor. Por ello, **conviene cerrarlas con el método** **close** siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (**Statement** y **PreparedStatement**) y los resultados (**ResultSet**) para liberar los recursos.

Respecto a las excepciones, en Java hay muchos tipos de excepciones, el paquete **java.lang.Exception** es el que contiene los tipos de excepciones.

**Para saber más**: [resumen esquematizado de conceptos vistos en este tema y más](https://www.adictosaltrabajo.com/2011/02/25/tutorial-basico-jdbc/).

## 7.1.- Excepciones.

El objeto Exception se pasa al código que se ha definido para gestionar la excepción (el que está dentro del catch).

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

public class prueba {

    public void estructura() {

        // Define la estructura para capturar excepciones trabajando con ficheros y con bases de datos

        try {

            // Código que puede generar una excepción

        } catch (FileNotFoundException e) {

            // Bloque para excepcion por fichero no encontrado

        } catch (IOException ioe) {

            // Bloque para excepcion por error de entrada-salida

        } catch (SQLException sqle) {

            // Bloque para excepcion por error con SQL

        } catch (Exception e) {

            // Bloque para excepcion por cualquier otro error

        } finally {

            // Instrucciones finales para, por ejemplo, limpieza

        }

    }

}

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.

Las **acciones** que se realizan **sobre una BB.DD** pueden lanzar la excepción **SQLException**. Este tipo de excepción proporciona entre otra información:

* Una **cadena** **describiendo el error**. Se obtiene con el método **getMessage()**.
* Un **código entero de error** **que especifica al fabricante de la BB.DD**.

**Para saber más**: [sobre excepciones en Java](http://elvex.ugr.es/decsai/java/pdf/B2-excepciones.pdf).

## 7.2.- Cierrre de conexiones.



1. JNI: Java Native Interface. [↑](#footnote-ref-1)
2. **Balanceo de carga**: se refiere a la distribución del tráfico de red entrante a través de un grupo de servidores backend, también conocido como Server Farm (conjunto de servidor) o Server Pool (conjunto de servidores). [↑](#footnote-ref-2)
3. X DevAPI: API común proporcionada por varios conectores de MySQL que permite acceder fácilmente a tablas relacionales y colecciones de documentos que se representan en JSON, desde una API con operaciones CRUD. X DevAPI utiliza el nuevo protocolo cliente-servidor X de la versión 8.0 de MySQL. Permite trabajar con datos relacionales y no relacionales usando expresiones, transacciones, índices y validaciones. Esta API está implementada por MySQL Shell y varios conectores de MySQL que soportan el protocolo X, como PHP, Python, Java y Node.js [↑](#footnote-ref-3)
4. Java EE: es una plataforma de programación para desarrollar y ejecutar software de aplicaciones Java con arquitectura de n niveles distribuida, basándose fundamentalmente en componentes de software modulares ejecutándose sobre un servidor de aplicaciones. [↑](#footnote-ref-4)