

## UD3.- Manejo de conectores.

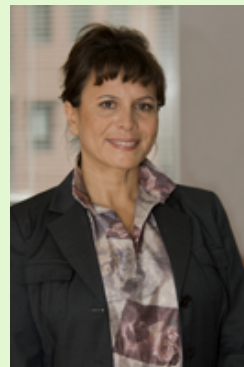


### Caso práctico

Esta mañana **Ada** está reunida con **María** y **Juan**.

Están discutiendo unos detalles de un proyecto, en el que la empresa está embarcada desde hace tiempo, y quieren retomarlo porque lo tenían un poco aparcado.

En esta ocasión se trata de una **franquicia de inmobiliarias**. El gerente que gestiona las franquicias contactó, hace tiempo, con BK Programación para que le hicieran una página web, en la que colgar información sobre los inmuebles en venta o alquiler.



**María** participó entonces activamente en el proyecto, así como **Juan**, por la amplia experiencia de ambos en desarrollo web.

Desarrollaron en su momento lo más básico del proyecto, pero ahora, **el gerente de las inmobiliarias está apremiando a BK Programación** para que terminen el resto del proyecto. Sobre todo, necesita realizar un montón de consultas sobre la base de datos relacional donde guardan los datos de la inmobiliaria: clientes, inmuebles, ventas realizadas, etc.

**Juan y María** van a aprovechar a **Ana y Antonio** para que les echen una mano y avanzar rápidamente en el proyecto.



**Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.**

[Aviso Legal](#)

# 1.- Introducción.

Actualmente, las bases de datos relacionales constituyen el sistema de almacenamiento probablemente más extendido, aunque otros sistemas de almacenamiento de la información se estén abriendo paso poco a poco.



Columna 1	Columna 2	Columna 3	Columna 4	Columna 5
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9
6	7	8	9	10
7	8	9	10	11
8	9	10	11	12
9	10	11	12	13
10	11	12	13	14

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).

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

El sistema gestor de bases de datos, en inglés conocido como: Database Management System (**DBMS**) gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: Relational Database Management System (**RDBMS**).

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**.



## Autoevaluación

Señala si la afirmación siguiente es verdadera o falsa.

**JDBC tiene como uno de sus principales objetivos facilitar el acceso a bases de datos relacionales.**

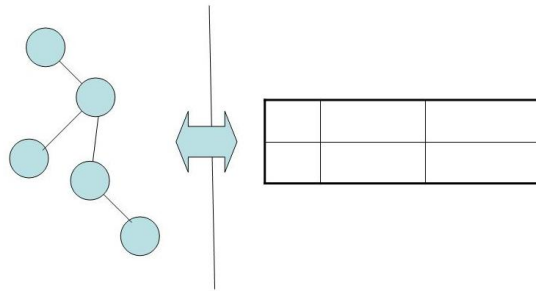
☒ Verdadero ☐ Falso

**Verdadero**

JDBC hace que sea muy sencilla la conexión de ese tipo de bases de datos.

# El desfase objeto-relacional.

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:** 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) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas (o filas), 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.

Esto es sencillo para un caso simple, pero complicado si el objeto posee muchas propiedades, o bien se necesita 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 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.

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.



## Pregunta Verdadero-Falso

El desfase objeto-relacional tiene que ver con que el paradigma de orientación a objetos tiene una naturaleza matemática y el modelo relacional no.

☐ Verdadero ☐ Falso

**Falso**

La afirmación es falsa, es el modelo relacional el que posee una naturaleza matemática.

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



### Caso práctico

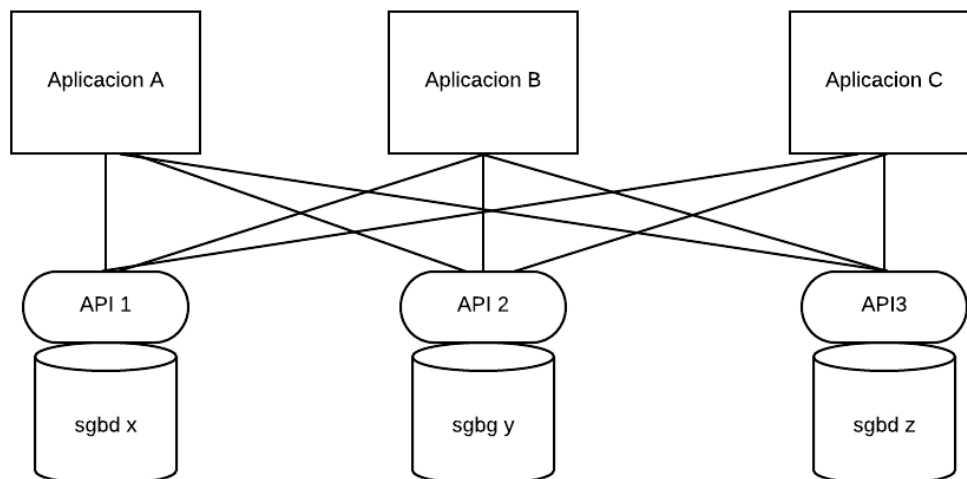
**María** les pregunta a **Antonio** y a **Ana** -¿Estáis dispuestos a realizar consultas sobre bases de datos? Nos vais a tener que echar una mano a **Juan** y a mi para acabar pronto el trabajo. -¡Cuenta con ello! -responde **Antonio**. -Hacer consultas es lo que más me gusta, ¡soy una máquina en eso!



**Ana** comenta que a ella también le apasiona el tema, y que en el ciclo de DAM lo han estudiado muy bien. -¿Qué base de datos tienen implantada, qué protocolo de acceso hay que usar?

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.

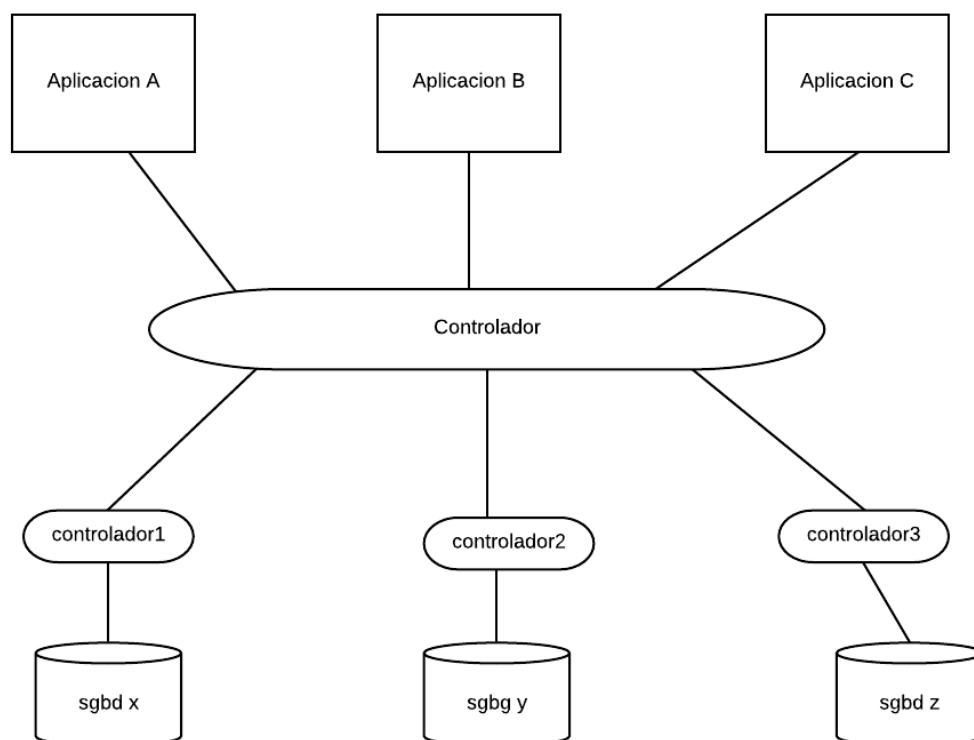
Cada SGBD tiene su propia conexión y su propio API



La llegada del ODBC representó un avance sin precedentes en el camino hacia la interoperabilidad entre bases de datos y lenguajes de programación. La mayoría de empresas desarrolladoras de sistemas gestores de bases de datos 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, la verdad es que no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

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



**JDBC (Java Database Connectivity)** se trata de un API bastante similar a **ODBC** en cuanto a funcionalidad, implementado específicamente para usar con el lenguaje Java, adaptado a las especificidades de Java. Es decir, la funcionalidad se encuentra capsulada 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. La idea en el desarrollo de JDBC era intentar ser tan sencillo como fuera posible, pero proporcionando a los desarrolladores la máxima flexibilidad.

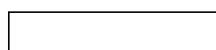
Java, mediante **JDBC**, permite simplificar el acceso a bases de datos relacionales, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos.

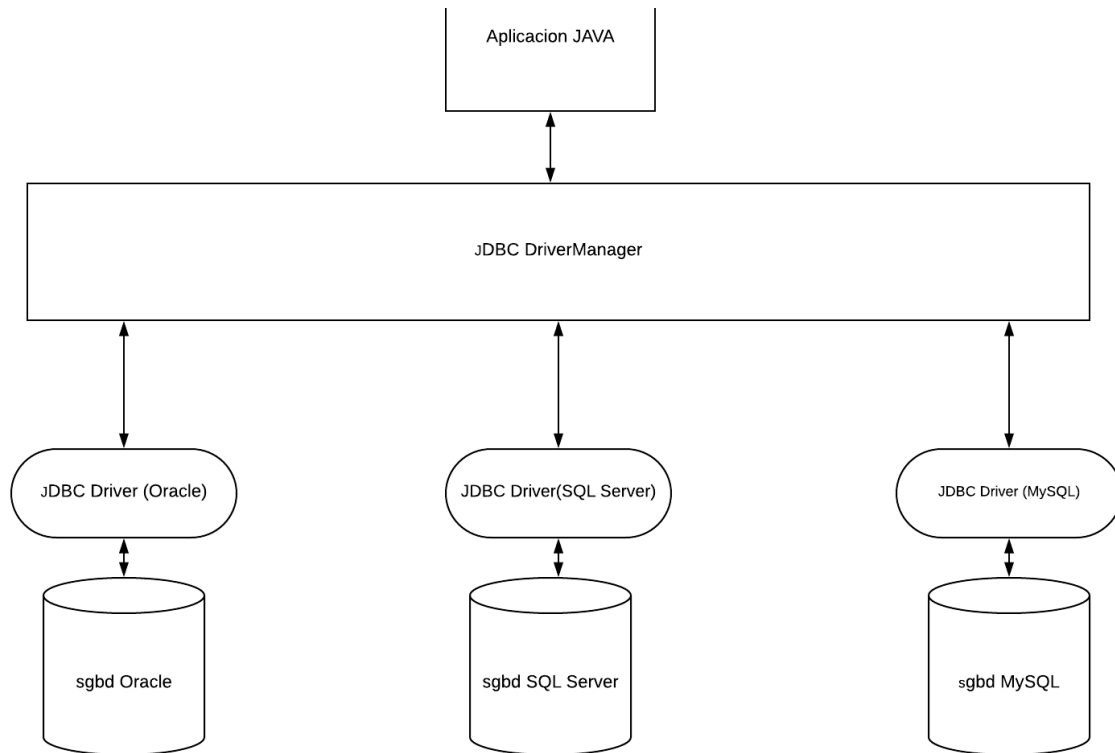
Sun desarrolló este API (`java.sql.*`) para el acceso a bases de datos, con tres objetivos principales en mente:

- ✓ 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.

Los desarrolladores de Los Sistemas Gestores de Bases de Datos proporcionan la implementación de esa interfaz (Drivers).

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 portátiles: puede ejecutar una aplicación Java sin modificaciones en cualquier sistema que tenga instalado una máquina virtual java (Java runtime environment).

Una aplicación Java que utiliza JDBC puede comunicarse con muchas bases de datos con pocas modificaciones, si es que las hay. Al igual que ODBC, JDBC proporciona una manera consistente de conectarse a una base de datos, ejecutar comandos y recuperar los resultados. 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 (Uniform Resource Locator). Una URL de JDBC es similar a las URL que utiliza con un navegador web. Para conectarnos a la base de datos Oracle, por ejemplo: `jdbc:oracle:thin:ejemplo/ejemplo@localhost:1521:XE`

A medida que cada controlador se carga en una Máquina Virtual de Java (VM), se registra con el **DriverManager JDBC**. Cuando una aplicación solicita una conexión, el DriverManager pregunta a cada **Controlador** si puede conectarse a la base de datos especificada en la URL dada. Tan pronto como encuentra un controlador adecuado, la búsqueda se detiene y el controlador intenta establecer una conexión con la base de datos. Si el intento de conexión falla, 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.

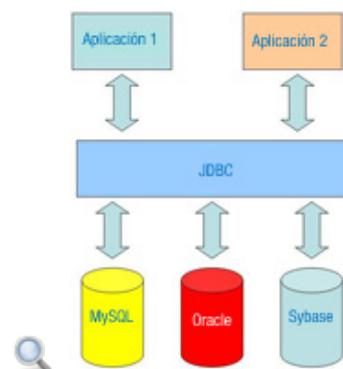
### 3.- Conectores o Drivers.

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

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un driver adecuado. 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 conector lo proporciona el fabricante de la base de datos o bien un tercero.



El código de nuestra aplicación no depende 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:

```
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.getStrin
    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.

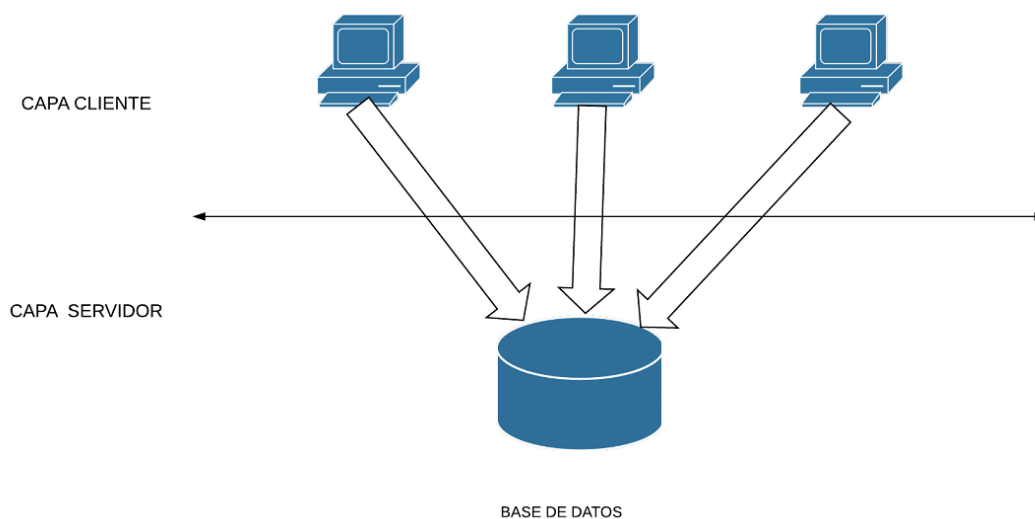


[JDBC y otras tecnologías de Oracle](#)

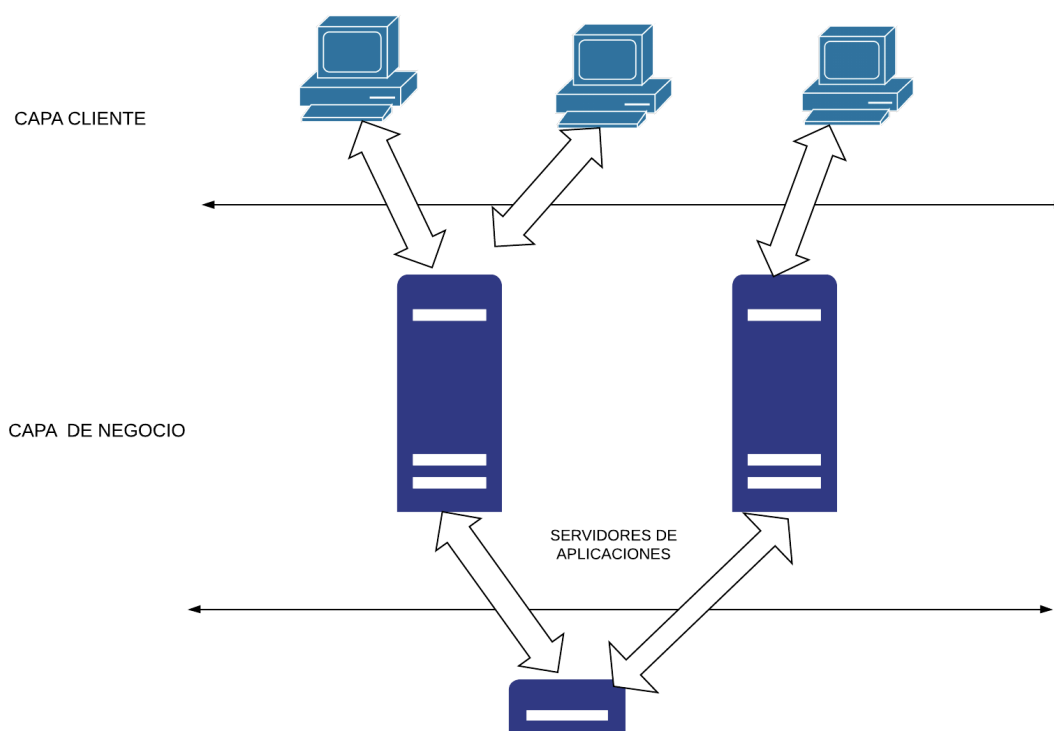
## 4.- Arquitectura JDBC.

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

Los comandos o instrucciones del usuario se envían a la base de datos y los resultados se devuelven al usuario. La fuente de datos puede estar ubicada en otra máquina a la que el usuario se conecte por red. A esto se denomina configuración  cliente/servidor, con la máquina del usuario como cliente y la máquina que aloja los datos como servidor.



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 a la capa intermedia, desde la que luego se le envían al usuario.



CAPA SERVIDOR

SERVIDOR DE BASE DE  
DATOS


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

En este enlace puedes ver también estos conceptos introductorios que estamos viendo sobre JDBC:

 [Introducción a JDBC](#)



## Autoevaluación

Di si la siguiente afirmación es verdadera o falsa:

**ODBC es lo mismo que JDBC, el primero para Windows y el segundo para Linux y Mac.**

☐ Verdadero ☐ Falso

**Falso**

Son dos APIs distintas.

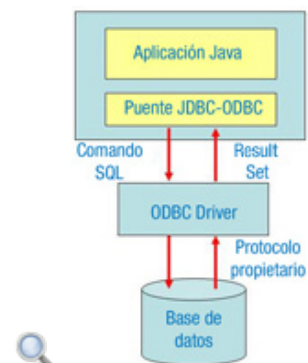
## 4.1.- Conectores tipo 1 y tipo 2.

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

Proporcionan un puente entre el API JDBC y el API ODBC. El driver JDBC-ODBC Bridge traduce las llamadas JDBC a llamadas ODBC y las envía a la fuente de datos ODBC.

Como ventajas destacar:

- ✓ 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.



Como desventajas señalar:

- ✓ 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).

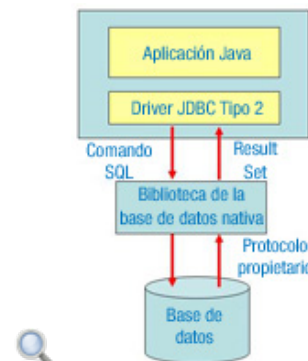
Este tipo de driver va incluido en el JDK.

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

Convierten las llamadas JDBC a llamadas específicas de la base de datos para bases 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, este conector destaca por ofrecer un rendimiento notablemente mejor que el JDBC-ODBC Bridge.



Como inconveniente, señalar que la librería de la 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, por lo que son más eficientes.



### Para saber más

En este enlace puede ver información sobre JNI y sobre DSN.





## Autoevaluación

Señala si la siguiente afirmación es verdadera o falsa:

**El problema de los conectores tipo 2 es que no pueden utilizarse para Internet.**

☒ Verdadero ☐ Falso

### Verdadero

La afirmación es correcta, porque hay que cargar librerías de la base de datos en la máquina cliente.

## 4.2.- Conectores tipo 3 y tipo 4.

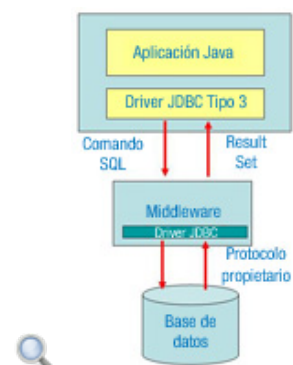
### Tipo 3: JDBC-Net pure Java driver.

Tiene una aproximación de tres capas. Las peticiones JDBC a la base de datos se pasan a través de la red al servidor de la capa intermedia (middleware). 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, funciones avanzadas de administrador de sistemas tales como auditoría, etc.



### Tipo 4: Protocolo nativo.

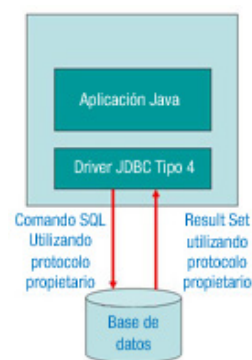
En este caso se trata de conectores que convierten directamente las llamadas JDBC al protocolo de red usando por el sistema gestor de la base de datos. Esto permite una llamada directa desde la máquina cliente al servidor del sistema gestor de base de datos y es una solución excelente para acceso en 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**.



## Para saber más

En la siguiente web puedes ver información sobre Oracle Thin y otros conectores de Oracle.

 [Conectores Oracle](#)

## 5.- Conexión a una base de datos.



A man with dark hair and a beard, wearing a light blue button-down shirt, stands in a room with large, colorful, abstract wall art. He is gesturing with his hands as if speaking. The background features a large, stylized handprint in wood tones and a blue and white abstract pattern above a doorway.

Para acceder a una base de datos y así poder operar con ella, **lo primero que hay que hacer es conectarse a dicha base de datos.**

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 {
        // Crear el socket de origen
        ClientSocket myConn = new ClientSocket();

        // Crear el socket para conectar con MySQL con localhost,
        // especificando la base de datos llamada "base"
        // con usuario "root" y contraseña "" servidor de MySQL con "localhost"
        String connectString = "jdbc:mysql://localhost:3306/base?user=root&password=";

        // Crear la conexión
        Connection con = myConn.getConnection(connectString);

    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    } catch (ClassNotFoundException e) {
        System.out.println("Exception: " + e.getMessage());
    }
}
```



En el enlace siguiente puedes ver cómo instalar MySQL:

## Instalar MySQL

Además del servidor, puedes descargarte e instalar una herramienta gráfica que permite entre otras cosas trabajar para crear tablas, editarlas, añadir datos a las tablas, etc. con MySQL. Aquí puedes ver cómo descargarlo y utilizarlo:



[Instalación y uso de MySQL WorkBench](#)



## 5.1.- Creación de la base de datos.



### Caso práctico

**Antonio** le ha pedido a **Juan** participar en uno de los proyectos que desarrolla en la empresa. Juan es el responsable de un proyecto de aplicaciones para oficinas de farmacia. **Juan** no está muy convencido al principio, porque Antonio está bastante centrado en el proyecto de las inmobiliarias y piensa que quizás sea demasiada carga para él, pero **María** interviene en la conversación y anima a **Juan** a que le eche una mano a **Antonio**. **Juan** va a crear una base de datos con Microsoft Access para guardar la información de los productos de la farmacia.



—**Antonio**, presta atención a lo que te digo —le dice **Juan**.

Juan le explica a Antonio, que 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 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.



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.

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.



### Debes conocer


Si tienes olvidado SQL, o no lo has visto antes, deberías familiarizarte con él:

 [Tutorial de SQL](#)

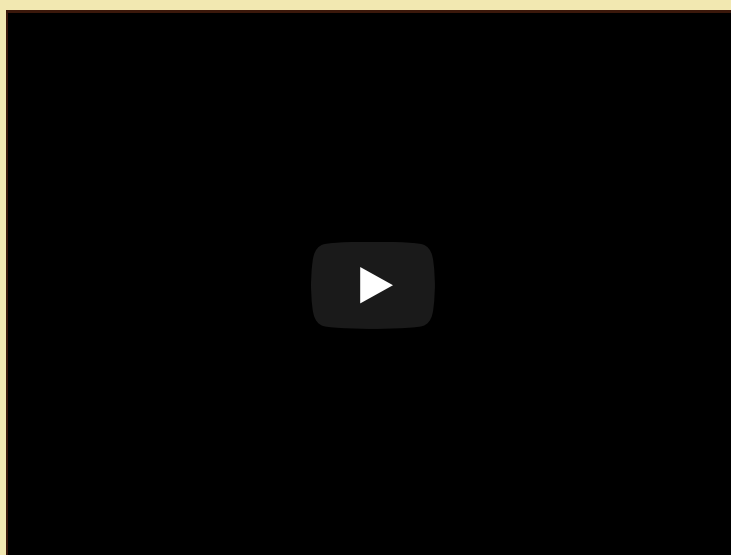


## Recomendación

Si necesitas refrescar el concepto de clave primaria, en la wikipedia puedes consultarlo.

 [Clave primaria](#)


Aquí puedes ver un vídeo sobre la instalación de la base de datos Oracle Express y creación de tablas.



[Resumen textual alternativo](#)

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

### Drivers MySQL

**MySQL proporciona controladores basados en estándares para JDBC, ODBC y .Net, lo que permite a los desarrolladores crear aplicaciones de base de datos en el idioma de su elección.** Puedes descargarlos en la  [pagina de MySQL](#).

Respecto a la implementación de JDBC, MySQL proporciona conectividad para aplicaciones cliente desarrolladas en el lenguaje de programación Java con MySQL Connector / J. Connector / J implementa 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 (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.

Podrás encontrar más información de los drivers JDBC disponibles en la  [pagina de MySQL](#).

### 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 de Oracle.** Debido a que es 100% Java, este controlador es independiente de la plataforma y también puede ejecutarse desde un navegador web.

Los formatos de 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)

**El controlador JDBC KPRB de Oracle se utiliza principalmente para escribir procedimientos almacenados Java y JSP (JavaServer Pages).** Utiliza la sesión de base de datos 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 ()`.

Encontraras la documentación de estos tres tipos de drivers es la pagina :  [JDBC Oracle](#).


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

Vamos a utilizar el IDE Eclipse para conectar java a las bases de datos, necesitamos instalar el conector de la base de datos.

Puedes seguir este  [tutorial](#), que explica la descarga del driver y la configuración necesaria para utilizarlo en un proyecto Eclipse. O bien, puedes seguir el video presentación

Como verás, tan sólo consiste en descargar un archivo, descomprimirlo y desde Eclipse añadir el fichero .jar que constituye el driver que necesitamos. Es importante que compruebes el driver mas adecuado para trabajar con tu Servidor MySQL.



[Curso Java. Acceso a BBDD. JDBC.](#)

**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:

```
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.out.println("conecta");
            conexion.close(); // Cerrar conexión
        } catch (SQLException cn) {
            cn.printStackTrace();
        }
    }
}
```

### Oracle

Para Oracle la configuración del driver se hace de la misma forma que en MySQL:

- ✓ Utilizaremos el driver `ojdbc8.jar` que podemos descargar en la  [pagina Oracle](#). Desde Eclipse añadimos esta librería: configuramos **Build Path**.
- ✓ La cadena de conexión recomendada es la que utiliza el `ServiceName`:

**host:port:service\_name**

Consulta si es necesario, el archivo Tnsnames.ora que proporciona informacion 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))
  )
```

La URL seria para esta base de datos:

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

### 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
    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.- 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.

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 en pedir la conexión hay que especificar de qué fuente se trata siguiendo el protocolo JDBC e indicando la url de los datos, y en su caso el usuario y password. 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 forma más sencilla de URL JDBC es una lista de tres valores separados mediante dos puntos. El primer valor de la lista representa el protocolo, que es siempre `jdbc` para los URL JDBC. El segundo valor es el subprotocolo. El tercer valor es el nombre de sistema para establecer la conexión con un sistema específico, dependerá del tipo de controlador utilizado, del *host* donde se aloje el SGBD, del puerto que este use para escuchar las peticiones y del nombre de la base de datos 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](#) y de la sintaxis de url correspondiente a cada uno de ellos.

Para conectarnos a una base de datos Oracle 11XE:

```
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>
            //ur 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");
            conexion.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

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`.

 [Para 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.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")
        // Cerrar conexión
        conexion.close();
    }
}
```



## 5.5.- Realización de consultas.

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

1. **Cargar** el driver necesario para comprender el protocolo que usa la base de datos en cuestión.
2. **Establecer** una conexión con la base de datos.
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.

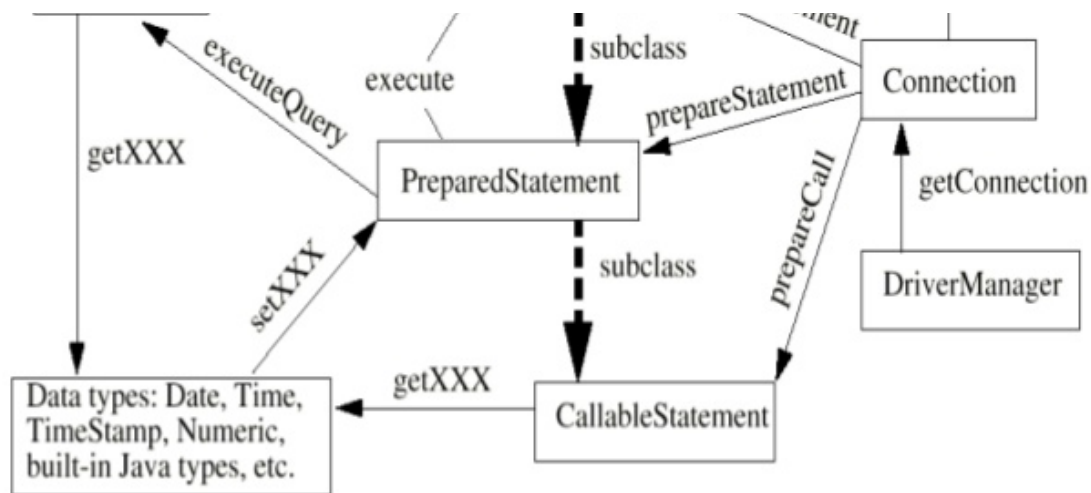
### Ejemplo 1:

```
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 conexión con la BD
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/ejemplo", "root", "s
// 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 reg
(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();
}
}
}
```

## JDBC Class Diagram

**SunilOS**





www.SunilIOS.com

35

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`. Se accede a los datos a través de un cursor. Este cursor es un puntero que apunta a una fila de datos en el objeto `ResultSet`. Inicialmente, el cursor se coloca antes de la primera fila. Accedemos a todas las filas de una a una llamando el método `next`, ya que cada invocación `next` avanzará a la siguiente fila.

### Ejemplo 2:

```
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();
        }
    }
}
```

```
}  
}
```

## 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. A partir de una conexión activa, instanciamos un objeto `Statement`.
2. Ejecutamos la sentencia SQL, método `executeUpdate`.
3. Cerramos el objeto `Statement` instanciado.
4. Cerramos la conexión activa.

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

Veamos el ejemplo siguiente, insertamos una línea y luego borramos 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 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 I  
            // 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

Puedes ver un ejemplo más de consultas preparadas en el siguiente enlace:



[Consultas preparadas](#)

## 5.6.- Ejecución de sentencias DML.

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

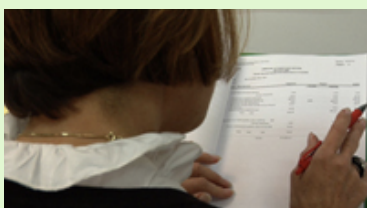
El funcionamiento de todas ellas en la conexión es bastante similar. A continuación se muestra un ejemplo de borrado en una conexión Oracle:

```
import java.sql.*;
public class DMLDeletePrep {
    public static void main(String[] args) {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection conexion = DriverManager.getConnection ("jdbc:odbc:ORACLE-XE","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 conexion
            conexion.close();
        }
        catch (ClassNotFoundException cn) {cn.printStackTrace();}
        catch (SQLException e) {e.printStackTrace();}
    }
}
```

## 5.7.- Ejecución de procedimientos almacenados en la base de datos.



### Caso práctico




Ada está terminando de diseñar unos procedimientos almacenados para un proyecto que está realizando la empresa, para unos grandes almacenes. En esos grandes almacenes utilizan MySQL como base de datos, puesto que ese sistema gestor soporta la ejecución de dichos procedimientos. Ada está pensando en pedir ayuda a alguien más de la empresa, porque se está dando cuenta de que habrá que realizar bastante código para toda la funcionalidad que se necesita.

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** almacenados.
- ✓ **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.

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

```
CREATE PROCEDURE procedimiento
  (IN par1 VARCHAR(13))
  SET var1 = 'perro rabioso';
  IF par1 = 'gato persa' THEN
    SET var1 = 'gato persa';
  END IF;
  INSERT INTO animales VALUES (var1);
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.



## Debes conocer

En el capítulo 19 del manual de referencia de MySQL que puedes encontrar en el enlace siguiente, puedes familiarizarte con los comandos que puedes necesitar para realizar procedimientos almacenados y funciones:



[Manual de referencia MySQL](#)

## 5.7.1.- Ejecución de procedimientos almacenados en la base de datos MySQL.

A continuación, vamos a realizar un procedimiento almacenado en MySQL, que simplemente insertará 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. Por simplicidad, en este ejemplo, trabajamos sobre la base de datos que viene por defecto en MySQL, el esquema denominado: test. Para crear la tabla de clientes, el script correspondiente es:

 [Crear tabla en MySQL.](#) (1 KB)

- ✓ Creamos el procedimiento almacenado en la base de datos. Sería tan fácil como lo que ves en el siguiente enlace:

 [DDL rutina insertar cliente.](#) (1 KB)

- ✓ Crear la clase Java para desde aquí, llamar al procedimiento almacenado:

 [Llamar al procedimiento almacenado.](#) (1 KB)

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

Te recomendamos que veas la documentación que hay en la siguiente dirección. Tienes ejemplos JDBC para diversas finalidades: listar datos de una base de datos, llamar a funciones en Oracle, etc.


 [Ejemplos JDBC](#)





## Para saber más

Un documento bastante más amplio que el anterior, sobre JDBC con Oracle, está en la siguiente dirección:

 [Documentación JDBC con Oracle](#)

## 5.7.2.- Ejecución de procedimientos almacenados en la base de datos Oracle.

---

Tomando un procedimiento de almacenamiento tal que:

```
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 almacenamiento de arriba:

```
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 `PreparedStatement` 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)}`”, dando los parámetros reales espera el procedimiento almacenado.
- ✓ El procedimiento se ejecuta cuando 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.

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

Una `PreparedStatement` 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.

### Veamos como se usa :

Símplemente ponemos interrogantes donde irán los valores concretos que vayamos a insertar. Cada interrogante se identifica luego con un número, de forma que el primer interrogante que aparece es el 1, el segundo el 2, etc. Luego, con los métodos `set()` correspondientes rellenos esos valores. El primer parámetro es el número del interrogante y el segundo el valor que queremos insertar. Finalmente llamamos a `executeUpdate()`. Luego, con la misma `PreparedStatement`, ponemos otros valores volviendo a llamar a los métodos `set()` y volvemos a llamar a `executeUpdate()`

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 `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 emp
        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)  
etc...
```

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

---

### Liberación de recursos

Debemos tener en cuenta siempre que las conexiones con una base de datos consumen muchos recursos en el sistema gestor, y por lo tanto en el sistema informático en general. 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`) 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`.

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

```
getErrorCode ()  
getMessage ()  
printStackTrace ()  
etc.
```



### Para saber más



[Excepciones en Java.](#) (88 KB)

## 5.10.- Transacciones.



### Caso práctico

Hay un aspecto sobre bases de datos, que **Ana** estudió en el ciclo formativo, y que no había tenido ocasión de ver en un caso real, y es el de las **transacciones en una base de datos**. Es un tema que le apasiona y le pide a María que le muestre alguna que haya realizado ella, para estudiarla a fondo y aprender con sus consejos.



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.



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.

El ejemplo típico que se pone para hacer más clara la necesidad de transacciones en algunos casos es el de una transacción bancaria. Por ejemplo, si una cantidad de dinero es transferida de la cuenta de Antonio a la cuenta de Pedro, se necesitarían dos consultas:

- ✓ En la cuenta de Antonio para quitar de su cuenta ese dinero:

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

- ✓ 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 “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 creará que ha realizado la transferencia. Pedro, sin embargo, creará que todavía no le han realizado la transferencia.



## Autoevaluación

**Respecto a las transacciones, señala la respuesta correcta:**

- ☐ En caso de una transacción con tres operaciones, no se podrá deshacer en ningún caso.
- ☐ Una transacción permite recuperar los datos si se produce un fallo, aportando por tanto más seguridad en la realización de operaciones en la base de datos.
- ☐ Una transacción se hará sólo si las operaciones involucradas operan con dinero.
- ☐ Ninguna es correcta.

¡Incorrecto, el número de operaciones es indiferente!

¡Exacto, así es!

¡No es correcto! No tiene por qué ser con dinero.

¡No! Hay una correcta.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

De manera más formal el control de la transacción es realizado por el objeto de la conexión. Cuando se crea una conexión, por defecto esta es 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 transacción: `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) {
        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.

Commit o  
Rollback

Una transacción tiene dos finales posibles, **COMMIT o ROLLBACK**. Si se finaliza correctamente y sin problemas se hará con **COMMIT**, con lo que los cambios se realizan en la base de datos, y si por alguna razón hay un fallo, se deshacen los cambios efectuados hasta ese momento, con la ejecución de **ROLLBACK**.

Por defecto, al menos en MySQL o con Oracle, en una conexión trabajamos en modo **autocommit** con valor **true**. Eso significa que cada consulta es una transacción en la base de datos.

Por tanto, 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**.

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;
```

Es conveniente planificar bien la aplicación para minimizar el tiempo en el que se tengan transacciones abiertas ejecutándose, ya que consumen recursos y suponen bloqueos en la base de datos que puede parar otras transacciones. En muchos casos, un diseño cuidadoso puede evitar usos innecesarios que se salgan fuera del modo estándar **AutoCommit**.




## Para saber más

Hay una documentación muy extensa para programar con PL-SQL: procedimientos, funciones, triggers, etc., en el siguiente enlace:

 [Programación con PL-SQL](#)

Interesante tutorial sobre transacciones y otras cuestiones con MySQL.

 [MySQL](#)

## 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.

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 una repetitiva para obtener los metadatos. Para ello utilizaremos metodos `getString`, `getInt`, etc.  [Consulta la documentación de la API JDBC.](#)

Algunos métodos `DatabaseMetaData` 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:

```
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.

```
getDriverVersion()
```

Devuelve en un `String`, el número de versión de este controlador JDBC como a `String`.

```
getExportedKeys(String catalog, String schema, String table)
```

Devuelve en un `ResultSet` la descripción de las columnas de clave Foranea 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 procedimientos 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** con 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", "e
            java.sql.DatabaseMetaData dbmd = conexion.getMetaData();// Creamos objeto DatabaseMeta
            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,
            }
            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 objeto `ResultSet` predeterminados no permiten la actualización, además, solo se puede mover el cursor hacia adelante. Sin embargo, puede crear objetos `ResultSet` que se pueden desplazar hacia adelante y hacia atrás o moverse a una posición absoluta y actualizarse.

Para poder utilizar el `ResultSet` para actualizaciones de tablas, tendremos que crear la consulta con la opción `ResultSet.CONCUR_UPDATABLE`, es un parámetro opcional de la creación de la sentencia:

```
conexion.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Además, es necesario actualizar el dato en la fila actual del objeto `ResultSet`:

```
resultSet.updateFloat( "PRICE", f * percentage); //recupera la columna con nombre "PRICE"  
resultSet.updateRow();
```

Con la opción `ResultSet.TYPE_SCROLL_SENSITIVE` se crea un objeto `ResultSet` cuyo que puede recorrerse hacia adelante y hacia atrás en relación con la posición actual y con una posición absoluta.

Veamos el ejemplo siguiente, el método `modifyPrices` (float percentage) multiplica el precio del café por un porcentaje dentro de la repetitiva que recorre, utilizando el objeto `ResultSet`, todas las filas de la tabla `coffees` utilizando el objeto `ResultSet`.

### 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(
        uprs.updateRow();
    }
} catch (SQLException e ) {
    JDBCUtilities.printSQLException(e);
} finally {
    if (stmt != null) { stmt.close(); }
}
}

```



## Para saber más



[Para saber mas de los tipos de ResultSet 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);
            // STEP 5: Execute a query
            String sql = "SELECT EMPLOYEE_ID, first_name, last_name, salary, email,
            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 salaries...");
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.

Acabamos de ver cómo se realiza una conexión a una base de datos. 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.

Antes de ver en qué consiste, veamos cómo funciona la creación de conexiones en una aplicación clásica de escritorio cliente-servidor y luego veamos los problemas que se seguirán con dicha estructura en una aplicación web.



### Problemas en la creación de conexiones.

En una aplicación de escritorio se crea una conexión de base de datos al iniciar la aplicación y se cierra al finalizar la aplicación. Es decir que cada usuario que inicia la aplicación tiene una conexión en exclusiva para él. Y obviamente sería imposible compartirlas 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 de la aplicación que se cierre su conexión. Esto que aparentemente es sencillo tiene unos problemas debido a la diferente naturaleza de las aplicaciones de escritorio y las web.

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).

Consecuencia de lo anterior al tener tantas conexiones a la base de datos se acabaría cayendo el servidor de base de datos debido a los recursos consumidos por todas las conexiones.

Podemos pensar que nuestro servidor puede aguantar todas esas conexiones ya que podemos tener pocos usuarios pero no suele ser así debido a:

- ✓ Si no se cierran las conexiones aun teniendo 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 y saturando el servidor haciendo que se crearan gran cantidad de conexiones que no se usen.

Una solución que nos evitaría las conexiones sin usar sería que se creara la conexión al iniciar cada petición web y se cerrara al finalizar dicha petición web. ¿El problema de eso? Crear y cerrar una conexión es muy costoso. Lo que tendríamos es una aplicación lentísima.

### Pool de conexiones.

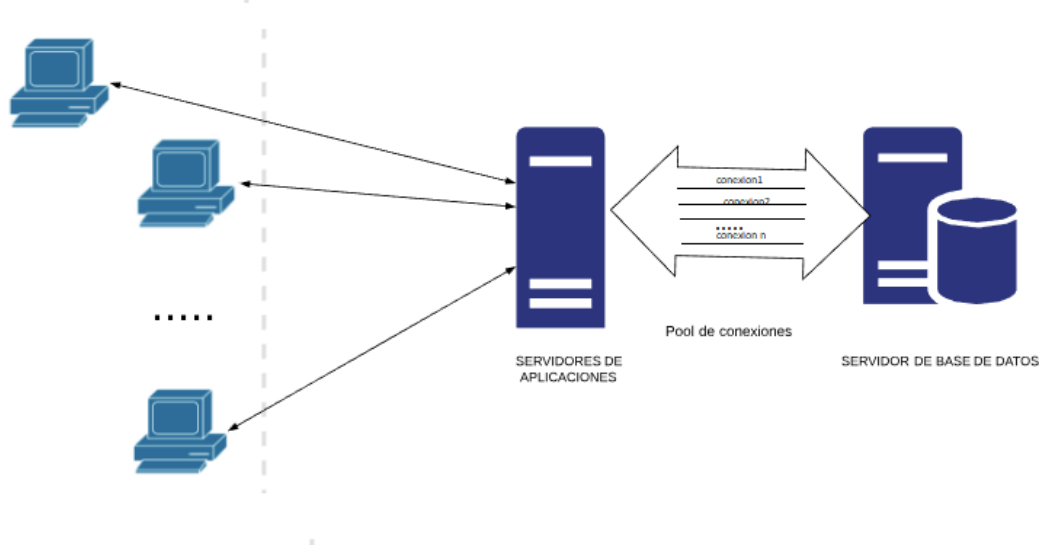
El concepto de Pool de conexiones se ha estandarizado desde la versión 3.0 de JDBC.

La solución del pool de conexiones tiene que solucionar los siguientes problemas:

- ✓ No tener tantas conexiones como usuarios ya que el número de usuarios es demasiado

elevado.

- ✓ Cerrar la conexión.



### Veamos como funciona:

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 esperando en el pool “n-m” conexiones. Esta operación es muy rápida ya que la conexión ya está creada y solo hay que marcarla como que alguien la está usando. Ahora hay “m” conexiones activas que está usando la aplicación.
- ✓ 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. Ahora vuelve a quedar “n” conexiones esperando en el pool. Esta operación también es muy rápida ya que realmente no se cierra ninguna conexión sino que simplemente se marcan como que ya no las están usando nadie.
- ✓ 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.

Al devolver una conexión al pool , ésta se queda esperando para que otra petición la pueda usar. 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 quedarán abiertas cuando el usuario se marcha del portal ya que cada conexión se pseudo-abre y pseudo-cierra con cada petición.
- ✓ No tenemos tantas conexiones como usuarios usan la aplicación ya que solo se necesitan tantas como usuarios hay haciendo una petición en ese instante. Pensemos por un momento en facebook. ¿cuandos usuarios están conectados a facebook? Supongamos “x”. Pero ¿cuantos están realmente haciendo una petición y no viendo los datos que se han servido? Supongamos “y”. Obviamente “y” es mucho menor que “x”. Con lo que nos hemos ahorrado “x-y” conexiones.
- ✓ 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 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 le devuelve la conexión lógica en cuestión.

Si hay un pico en la demanda de conexiones a la base de datos, el pool de conexiones de forma transparente crea más conexiones físicas de objetos tipo `Connection`. Si por el contrario las conexiones a la base de datos disminuyen, el pool de conexiones, también de forma transparente elimina conexiones físicas de objetos de tipo `Connection`.



## Debes conocer


El siguiente enlace es bastante interesante, mostrando ejemplos de código.

 [Ejemplo de pool de conexiones](#)



## Para saber más

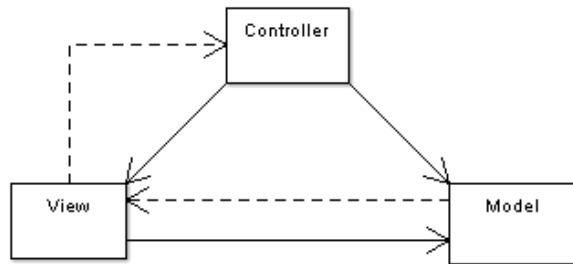
En el siguiente enlace puedes ver un ejemplo paso a paso con NetBeans y MySQL, sobre la creación de un pool de conexiones.

 [Como crear un Pool de Conexiones en Java con NetBeans y MySQL](#)

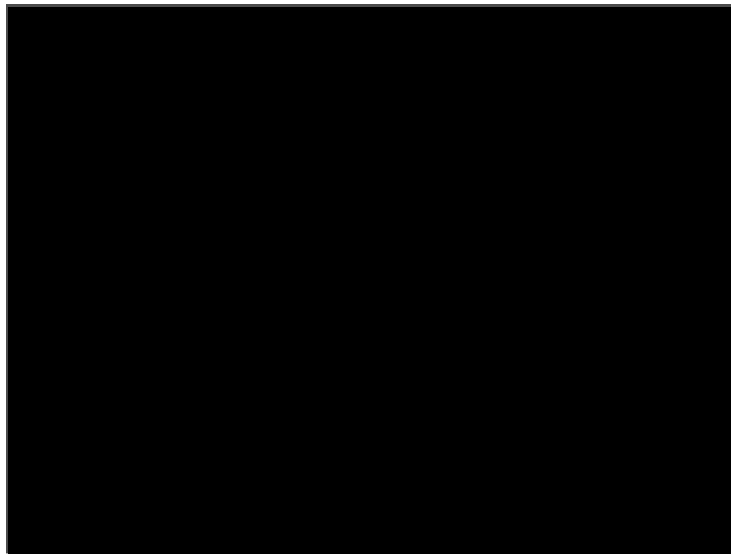
## 5.13.1.- Ejemplo de pool de conexiones MySQL.

---

El objetivo del proyecto siguiente es crear una aplicacion 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.



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



[Resumen textual alternativo](#)

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

---

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

```
package PoolConexiones;
/*
    DESCRIPTION
    The code sample shows how to use the DataSource API to establish a connection
    to the Database. You can specify properties with "setConnectionProperties".
    This is the recommended way to create connections to the Database.
    Note that an instance of oracle.jdbc.pool.OracleDataSource doesn't provide
    any connection pooling. It's just a connection factory. A connection pool,
    such as Universal Connection Pool (UCP), can be configured to use an
    instance of oracle.jdbc.pool.OracleDataSource to create connections and
    then cache them.
    Step 1: Enter the Database details in this file.
            DB_USER, DB_PASSWORD and DB_URL are required
    Step 2: Run the sample with "ant DataSourceSample"
    NOTES
    Use JDK 1.7 and above
    MODIFIED    (MM/DD/YY)
    nbsundar    02/17/15 - Creation
*/
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 {
    // The recommended format of a connection URL is the long format with the
    // connection descriptor.
    final static String DB_URL= "jdbc:oracle:thin:@localhost:1521:XE";
    final static String DB_USER = "scott";
    final static String DB_PASSWORD = "tiger";
    /*
    * The method gets a database connection using
    * oracle.jdbc.pool.OracleDataSource. It also sets some connection
    * level properties, such as,
    * OracleConnection.CONNECTION_PROPERTY_DEFAULT_ROW_PREFETCH,
    * OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES, etc.,
    * There are many other connection related properties. Refer to
    * the OracleConnection interface to find more.
    */
    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);
}
}
/*
 * Displays first_name and last_name from the employees table.
 */
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) + " ");
        }
    }
}
}
```



## 6.- Operaciones: ejecución de consultas.



### Caso práctico

**Juan y Antonio** están inmersos en la creación de consultas para los informes que la aplicación de farmacias debe aportar a los usuarios de la misma. Realmente es **Juan** el que está realizándolas, pero Antonio empieza a comprender el asunto y quiere participar más.



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.

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

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 de una aplicación Java.



### Autoevaluación

**Respecto a las consultas con JDBC, señala la respuesta correcta:**

- ☐ Podemos crear una tabla con una consulta de actualización.
- ☐ Hay que liberar los recursos antes de cargar el driver de la base de datos.
- ☐ No es posible realizar consultas si no usamos un puente ODBC.
- ☐ Todas son correctas.

¡Exacto! Con la sentencia correspondiente DDL.

¡Incorrecto! No es imprescindible.

¡No es correcto! Eso no es cierto.

¡No es la opción correcta! Hay una cierta.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

## 7.- Excepciones y cierre de conexiones.



### Caso práctico

**Cuando Ada se reunió con Juan y María**, les hizo hincapié en que debían ser especialmente **rigurosos en la programación de los proyectos**, tratando los posibles errores de las aplicaciones que programaran, para evitar la finalización abrupta del programa, y también el cierre adecuado de las conexiones a bases de datos, para evitar perder la información. Así mismo, les comunicó que **transmitieran esto mismo a Ana y a Antonio**.



Debemos tener en cuenta siempre que las conexiones con una base de datos consumen muchos recursos en el sistema gestor, y por lo tanto en el sistema informático en general. 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.


Una excepción es una situación que no se puede resolver y que provoca la detención del programa de manera abrupta. Se produce por una condición de error en tiempo de ejecución.

En Java hay muchos tipos de excepciones, el paquete `java.lang.Exception` es el que contiene los tipos de excepciones.



### Para saber más

Aquí puedes ver un resumen esquematizado de conceptos vistos en este tema y más:

 [Bases de datos con Java](#)

## 7.1.- Excepciones.

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.

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

} catch (FileNotFoundException fnde) {
    // 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
}
```

 [Código con la estructura para capturar excepciones.](#) (1 KB)

**El bloque de instrucciones del `try` es el que se ejecuta, y si en él ocurre un error que dispara una excepción, entonces se mira si es de tipo fichero no encontrado; si es así, se ejecutarían las instrucciones del bloque del fichero no encontrado.** Si no era de ese tipo la excepción, se mira si es del siguiente tipo, o sea, de entrada salida, y así sucesivamente.

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 base de datos pueden lanzar la excepción `SQLException`. Este tipo de excepción proporciona entre otra información:

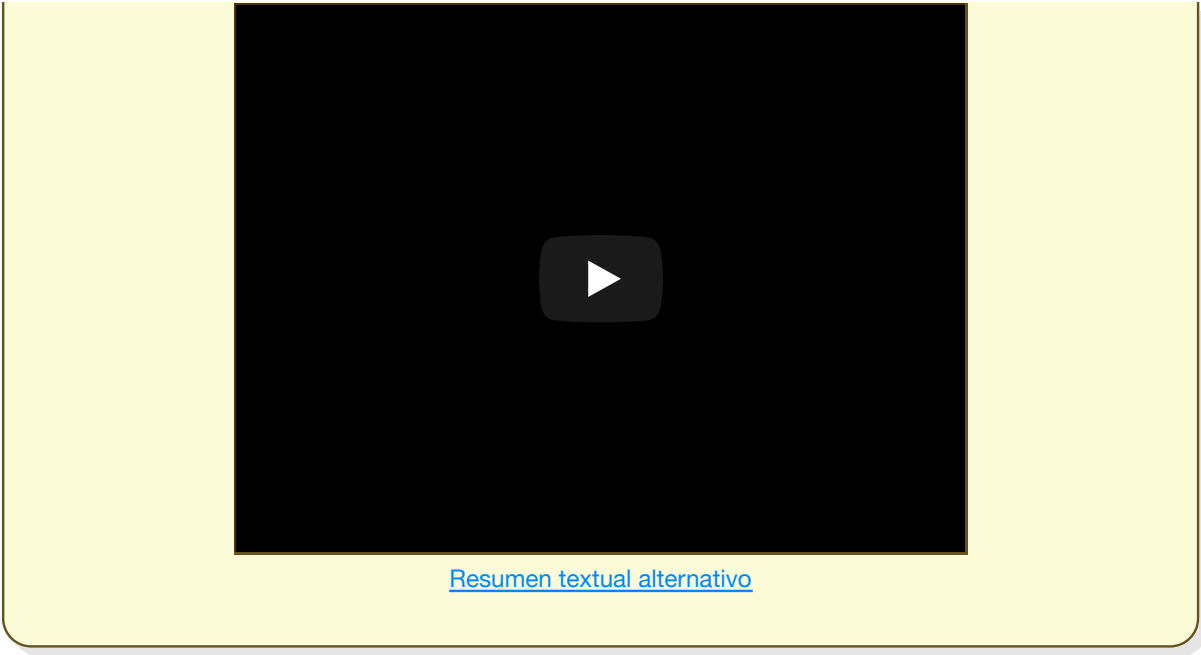
- ✓ Una cadena de caracteres describiendo el error. Se obtiene con el método `getMessage`.
- ✓ Un código entero de error que especifica al fabricante de la base de datos.



### Para saber más

Para saber más sobre excepciones, puedes consultar estos dos enlaces:

 [Excepciones en Java.](#) (88 KB)



## 7.2.- Cierre de conexiones.

Como ya hemos dicho, 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.

Podríamos por tanto tener un ejemplo de cómo hacer esto:

```
Connection con = null;
try {
    con = DriverManager.getConnection("jdbc:odbc:source");
    System.out.println("Conexión realizada con éxito.");
    // Aquí hacemos lo que necesitamos
} catch (SQLException e) {
    // Aquí hacemos 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());
        }
    }
}
```



### Para saber más









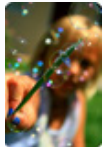
Vídeo resumiendo parte de lo visto en el tema:



[Resumen textual alternativo](#)

## Anexo.- Licencias de recursos.

### Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: krollian. Licencia: CC-by-nc. Procedencia: <a href="http://www.flickr.com/photos/krollian/3705826490/">http://www.flickr.com/photos/krollian/3705826490/</a>		Autoría: pcambraf. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/pcambra/3347911070/">http://www.flickr.com/photos/pcambra/3347911070/</a>
	Autoría: jimw. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/jimwinstead/24124753/">http://www.flickr.com/photos/jimwinstead/24124753/</a>		Autoría: hongiv. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/hongiv/85186002/">http://www.flickr.com/photos/hongiv/85186002/</a>
	Autoría: 姒儿喵喵. Licencia: CC-by-nc. Procedencia: <a href="http://www.flickr.com/photos/crystaljingsr/3914729343/">http://www.flickr.com/photos/crystaljingsr/3914729343/</a>		Autoría: Amortize. Licencia: CC-by-nc-sa. Procedencia: <a href="http://www.flickr.com/photos/amortize/527435776/">http://www.flickr.com/photos/amortize/527435776/</a>
	Autoría: Chavezonico. Licencia: CC-by-nc-sa. Procedencia: <a href="http://www.flickr.com/photos/chavezonico/3806410897/">http://www.flickr.com/photos/chavezonico/3806410897/</a>		Autoría: MrDoS. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/mrdos/3353224081/">http://www.flickr.com/photos/mrdos/3353224081/</a>
	Autoría: Andrew Murdoch Licencia: CC-by-nc-sa. Procedencia: <a href="http://www.flickr.com/photos/andrewmurdoch/3233287999/sizes/s/in/photostream/">http://www.flickr.com/photos/andrewmurdoch/3233287999/sizes/s/in/photostream/</a>		Autoría: Andrés Rueda. Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/andresrueda/3274955487/">http://www.flickr.com/photos/andresrueda/3274955487/</a>
	Autoría: Monroe's Dragonfly Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/monroesdragonfly/2739734655/">http://www.flickr.com/photos/monroesdragonfly/2739734655/</a>		Autoría: Ivan Walsh Licencia: CC-by. Procedencia: <a href="http://www.flickr.com/photos/ivanwalsh/5169156843/">http://www.flickr.com/photos/ivanwalsh/5169156843/</a>