

ACCESO A DATOS  
TÉCNICO EN DESARROLLO DE APLICACIONES  
MULTIPLATAFORMA

## Manejo de conectores I

---

# ÍNDICE

<b>/ 1. Introducción y contextualización práctica</b>	<b>3</b>
<b>/ 2. Introducción al manejo de conectores</b>	<b>4</b>
2.1. El desfase objeto-relacional	4
2.2. Protocolos de acceso a base de datos	4
<b>/ 3. Conexiones: Componentes y tipos</b>	<b>5</b>
3.1. Componentes	5
3.2. Tipos	6
<b>/ 4. Caso práctico 1: “Diseño de arquitectura, aplicación registro de usuarios”</b>	<b>8</b>
<b>/ 5. Configuración de una conexión en código</b>	<b>8</b>
5.1. Establecer conexión	9
5.2. Operaciones con variables y excepciones	10
<b>/ 6. Ventajas e inconvenientes del uso de conectores</b>	<b>11</b>
<b>/ 7. Caso práctico 2: “Accedo a distintas BDD”</b>	<b>12</b>
<b>/ 8. Resumen y resolución del caso práctico de la unidad</b>	<b>13</b>
<b>/ 9. Bibliografía</b>	<b>14</b>
<b>/ 10. Webgrafía</b>	<b>14</b>

# OBJETIVOS



*Conocer el concepto desfase objeto-relacional.*

*Aprender los diferentes protocolos de acceso.*

*Identificar y desarrollar los diferentes tipos de conectores.*



## / 1. Introducción y contextualización práctica

En el siguiente capítulo, aprenderemos a **realizar una conexión a base de datos** desde nuestro aplicativo Java. Para ello, necesitaremos una serie de librerías para realizar las conexiones y traducciones necesarias.

Estudiaremos los distintos **tipos de protocolos JDBC y ODBC**.

Profundizaremos en los distintos tipos de conectores que existen, y mostraremos las ventajas e inconvenientes de cada uno de ellos.

Realizaremos una **instanciación de un driver** a modo práctico explicando paso a paso las distintas funcionalidades de los drivers.

Escucha el siguiente audio donde planteamos la contextualización práctica de este tema, encontrarás su resolución en el apartado Resumen y resolución del caso práctico.



Fig. 1.



Audio intro. "Conexión Driver"

<https://bit.ly/396f1JW>





## / 2. Introducción al manejo de conectores

Definimos **conector** como una serie de clases y librerías que realizan la labor de unir la capa de nuestra aplicación con la capa de base de datos. Este punto intermedio es nuestro conector, y es necesario para conectarnos a la base de datos y realizar consultas.

### 2.1. El desfase objeto-relacional

El concepto **desfase objeto-relacional** viene en relación a dos puntos: por un lado, tenemos la parte de la programación orientada objetos, es decir, el aplicativo, y por otro lado, tenemos la base de datos física. Muchas veces, surgen discrepancias debido a que la parte de la bases de datos tienen naturalezas distintas en comparación al aplicativo que se trabaja con programación orientada a objetos. A este problema lo definimos como **desfase objeto-relacional**.

El problema no es otro que las discordancias que surgen cuando trabajamos conjuntamente con una **base de datos relacional** y un aplicativo **orientado a objetos**. Algunos aspectos importantes del desfase son:

- Hay bastante **diferencia entre los datos** que se usan en las bases de datos relacionales y en la programación orientada a objetos. En las primeras se usan datos simples, mientras que en la orientada a objetos son objetos complejos.
- Implica realizar distintos diagramas, ya que vamos a tener que **realizar una traducción** desde los objetos del aplicativo Java a la base de datos relacional, por lo tanto, se crearán entidades en ambos sitios. Entidades distintas, aunque representen la misma unidad.



Fig. 2. Esquema: aplicativo-conector-base de datos.

### 2.2. Protocolos de acceso a base de datos

Realmente, un **conector o driver** es una serie de **clases implementadas (API)** que facilitan la conexión a la base de datos asociada.

Basándonos en el lenguaje **SQL**, disponemos de **dos protocolos** de conexión:

- **JDBC (Java Database Connectivity)**: este protocolo fue desarrollado por Sun.
- **ODBC (Open DataBaseConnectivity)**: desarrollado por Microsoft.



Audio 1. "Otros protocolos"  
<https://bit.ly/2WDslR5>





Una aplicación debe tener asociado siempre un conector. Cuando estamos desarrollando un aplicativo e introducimos un conector, no debemos de preocuparnos de demasiadas cuestiones. No tenemos que conocer los aspectos técnicos, ni cómo funcionan en su interior dichas bases de datos. Nos ocuparemos, básicamente, de cómo realizar la comunicación y de cómo funcione nuestro aplicativo.

Si usamos **el conector JDBC**, por ejemplo, no tendríamos que desarrollar un aplicativo para acceder a una base de datos Oracle y otro aplicativo o driver para acceder a una base de datos distinta, sino que nosotros desarrollaríamos nuestra aplicación y a la hora de realizar cualquier consulta, el conector interpretaría de una forma u otra dependiendo de la base de datos asociada.



Fig. 3 Ejemplo conector JDBC.

En este punto, nuestro aplicativo **necesitaría información de una base de datos**. Simplemente utilizando dicha **librería JDBC** e indicando las configuraciones de acceso a cada una de ellas, tendremos acceso a las mismas sin preocuparnos del lenguaje interno de cada una.

## / 3. Conexiones: Componentes y tipos

Debido a su extensión en el desarrollo de software, y su extenso uso, nos centraremos en el conector JDBC.

Dividiremos esta sección en dos puntos: **componentes** y tipos de **conexiones**.

### 3.1. Componentes

Podemos destacar cuatro componentes en el conector JDBC:

- **La API de JDBC:** con ella tenemos una serie de librerías y clases que nos facilitan el acceso a las bases de datos relacionales. También nos brinda la oportunidad de realizar consultas a la base de datos. Podemos encontrar estas clases en la paquetería: `java.sql` y `javax.sql`.
- **Paquete de pruebas JDBC:** estas clases se encargan de validar si un driver pasa los requisitos previstos por JDBC.
- **El gestor JDBC:** es el encargado de realizar la unión entre nuestra aplicación Java con el driver apropiado JDBC. Hay dos formas de realizar dicha operación:
  - Conexión directa
  - A través de un pool de conexiones
- **El puente JDBC-ODBC:** facilita el uso de los drivers ODBC como si estuviéramos trabajando con JDBC.

En relación a la arquitectura de conexión con JDBC, podemos identificar dos tipos de arquitectura:

- **Arquitectura en dos capas:** nuestra aplicación se conectará a la BDD a través de un driver. Tanto el driver como la aplicación deben localizarse en el mismo sistema o máquina.

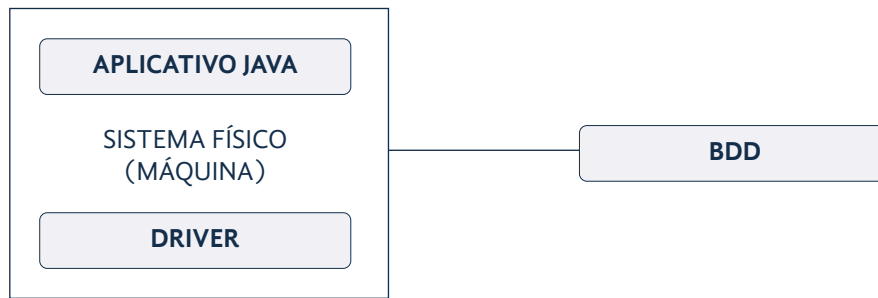



Fig. 4. Arquitectura de dos capas.

- **Arquitectura en tres capas:** nuestro aplicativo enviará las instrucciones a una capa intermedia (middleware). Esta capa cogerá la información y la enviará a la base de datos correspondiente traduciendo los comandos que el aplicativo haya enviado.




Fig. 5. Arquitectura de tres capas.



Vídeo 1. "Arquitectura tres capas"

<https://bit.ly/3h2UJ6C>



## 3.2. Tipos

A continuación, vamos a nombrar y a desarrollar los distintos **tipos de conexiones JDBC** disponibles.

- **Driver tipo 1 JDBC-ODBC:** este driver usa una API nativa, traduce las llamadas realizadas de JDBC a ODBC. Los datos devueltos por la base de datos se traducirán a JDBC cuando sean devueltos.

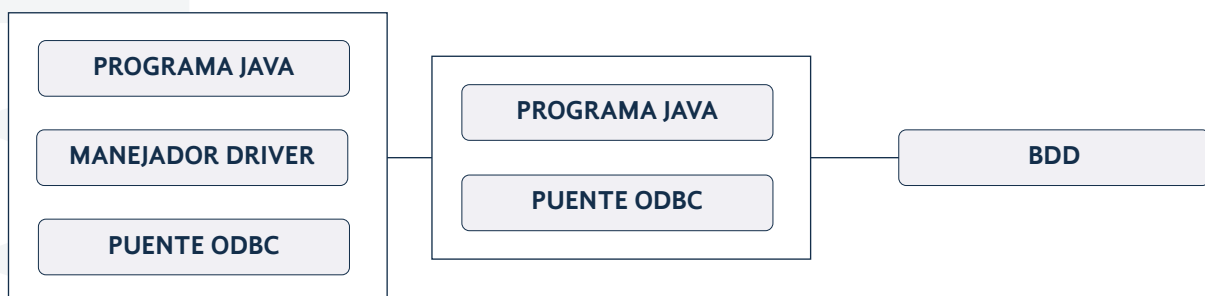


Fig.6. Driver tipo 1.



- **Driver tipo 2 JDBC Nativo:** estos drivers están escritos una parte en Java y otra parte, en código nativo. Las llamadas al API JDBC son traducidas en llamadas propias de la base de datos relacional que tengamos.



Fig.7. Driver tipo 2.

- **Driver tipo 3 JDBC net:** es un driver de tres capas cuyas solicitudes JDBC están siendo traducidas en un protocolo de red en una capa intermedia o middleware. Esta capa intermedia recibirá dichas solicitudes y las enviará a la base de datos usando un driver JDBC de tipo 1 o de tipo 2. Cabe destacar que es una arquitectura muy flexible.



Fig. 8. Driver tipo 3.

- **Driver tipo 4 protocolo nativo:** este tipo de driver realiza las llamadas mediante el servidor, usando el protocolo nativo del mismo. Estos drivers pueden desarrollarse al completo en Java, sin ningún problema. Si en el futuro se necesitara hacer un cambio de base de datos, evidentemente, habría que desarrollar otro driver nativo adaptado a la nueva base de datos relacional.



Fig.9. Driver tipo 4.



## / 4. Caso práctico 1: “Diseño de arquitectura, aplicación registro de usuarios”

**Planteamiento:** A Jorge, arquitecto de software de una empresa, le encargan el diseño técnico de la parte de datos de una pequeña aplicación que, simplemente, realiza un registro de usuarios guardando seis campos sencillos en una base de datos relacional.

**Nudo:** A Jorge le surge la duda de si montar una pequeña capa de datos con el driver en la capa cliente, o si, por el contrario, montar otra máquina intermedia que haga de traductor, donde estaría instalado el conector (de tres capas).

**Desenlace:** En este caso, Jorge debe diseñar una arquitectura simple de dos capas, similar a la de la Figura 3 vista en el apartado 3.1:

Esta arquitectura basada en dos capas será ideal para una aplicación simple que no requiere muchos recursos ni se prevé que vaya a tener multitud de consultas.

Para este caso, se puede instalar el conector (driver) en la misma máquina del cliente realizando las labores de traducción y comunicándose directamente con la base de datos.

Sin embargo, si fuera una aplicación (escritorio o web) cuyo propósito fuera gestionar una cantidad grande de consultas y necesitara balancearlas, incluso con algún tipo de caché, sería conveniente introducir una arquitectura de tres capas.

## / 5. Configuración de una conexión en código

Una vez vistos los tipos de conexión, los protocolos de acceso, y qué tipo de drivers son los más usados en el mercado, pasaremos a línea de código.

El primer paso sería descargar el driver (suele ser “.jar”) de conexión de la base de datos que vamos a utilizar y, a continuación, añadirlo a nuestro proyecto Java o nuestro aplicativo. Vamos a ver un ejemplo de conexión en línea de código:

```
private static final String DRIVER = "org.mysql.jdbc.Driver";  
private static final String URL_CONEXION = "jdbc:mysql://  
localhost:3306/Pruebas";
```

Código 1: Conexión driver URL.



**Enlace de interés...**

Fuente <http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc>

**El primer paso** para la conexión de una base de datos externa por medio de un driver de conexión **es definir algunos literales** que nos van a hacer falta, como el literal “Driver”, que hace referencia a la librería que hemos añadido a nuestro aplicativo, y la “URL CONEXION”, que hace referencia a la URL donde se alojará la información. Estas, podemos definir las como variables estáticas generales, ya que accederemos a ellas más tarde.





Como estamos realizando una prueba de desarrollo, hemos introducido el código en nuestro método main.

```
public static void main(String args[]) throws SQLException {  
    final String usuario = "user_db";  
    final String password = "password_db";  
    Connection dbConnection = null;  
    Statement statement = null;
```

Código 2: Conexión driver, credenciales.



### Enlace de interés...

Fuente: <http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc>

En el código anterior, podemos ver cómo **se definen ciertas variables de tipo String** que nos van a servir para realizar la conexión con la base de datos más tarde.

**Instanciamos** el usuario y la contraseña de nuestra conexión y también una variable de tipo **Connection** y otra **Statement**.

**Connection es una interfaz** que representa una conexión directa con una base de datos. El motivo de que sea una interfaz es porque tendrá distintas implementaciones posibles.

Como comentamos anteriormente, **JDBC ofrece distintas formas para realizar conexiones**. Nos centraremos en establecer la conexión con *"java.sql.DriverManager"*, recomendada para aquellos aplicativos que se hayan desarrollado en lenguaje Java.

Lo desarrollaremos en el siguiente apartado.

## 5.1. Establecer conexión

Podremos tener instaladas tantas conexiones como queramos. Cada conexión y cada base de datos utilizará los drivers JDBC, y, a su vez, cada uno de ellos implementará la interfaz *"java.sql.Driver"*. Con el método principal *connect()*, obtendremos el objeto *Connection* y estableceremos la conexión con base de datos:

```
try {  
    Class.forName(DRIVER);  
    dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password);  
    String selectTableSQL = "SELECT ID,USERNAME,PASSWORD,NOMBRE FROM Usuarios";  
    statement = dbConnection.createStatement();  
    ResultSet rs = statement.executeQuery(selectTableSQL);  
    while (rs.next()) {  
        String id = rs.getString("ID");  
        String usr = rs.getString("USERNAME");  
        String psw = rs.getString("PASSWORD");  
        String nombre = rs.getString("NOMBRE");  
        System.out.println("userid : " + id);  
        System.out.println("usr : " + usr);  
        System.out.println("psw : " + psw);  
        System.out.println("nombre : " + nombre);  
    }  
}
```

Código 3. Estableciendo conexión.



### Enlace de interés...

Fuente: <http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc>

El objetivo de la clase **DriverManager**, realmente, es **gestionar los drivers** que poseemos en nuestra aplicación y permitir en una misma capa el acceso a todos y cada uno de ellos. Algo que debemos tener en cuenta es que *DriverManager* necesita que todos y cada uno de los drivers estén registrados antes de su uso. Las conexiones deben quedar almacenadas antes de acceder a la base de datos.

Después de haber registrado el driver, se pueden usar los métodos estáticos para hacer “*getConnection*”, usándolo directamente para establecer conexiones.

Tal y como podemos observar en el código anterior, englobaremos una serie de operaciones en un bloque try/catch. **La primera instrucción que daremos es: “Class.forName()”**, de esta forma registraremos el driver que anteriormente hemos indicado en la variable estática “DRIVER”.

Una vez realizada una buena introducción de la clase “*DriverManager*” y de sus funcionalidades, si continuamos con el código, encontraremos cómo se está usando el **método “getConnection”**, al que le pasamos por parámetro la URL de conexión previamente definida: usuario y contraseña.

Todo esto nos devolverá un objeto de tipo *Connection*, en nuestro caso lo hemos llamado *dbConnection*.

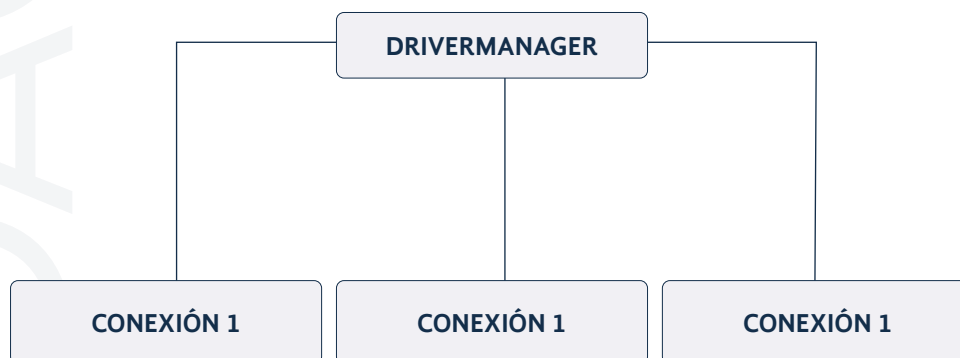


Fig.10. DriverManager.

## 5.2. Operaciones con variables y excepciones

Siguiendo con el ejemplo del Código 3, una vez que *DriverManager* nos ha devuelto la conexión a base de datos, realizaremos un ejemplo sencillo de **consulta simple** y **la almacenaremos en una variable de tipo String** para más tarde ser ejecutada.

Con la **variable Connection**, ejecutamos el método “createStatement” y lo **asignamos a la variable definida** al principio del ejercicio de tipo Statement. Más tarde, simplemente, tendremos que realizar la consulta con el método “executeQuery” pasándole como parámetro la query previamente definida en la variable de tipo *String*.

El resultado de la *query* se **asignará a una variable de tipo ResultSet**. Como podemos comprobar en el código, dicho *ResultSet* está envuelto en un bucle “while”, ya que por cada fila que nos devuelva esta tabla, podremos ir dando una vuelta más al bucle y seguir mostrando los resultados. En esta ocasión, hemos decidido realizar la operación de mostrar por pantalla tanto el ID, el USERNAME, el PASSWORD y el NOMBRE, que son columnas de la tabla *Usuarios* que hemos consultado de prueba.



A continuación, veremos la parte final del código donde vienen indicadas ciertas excepciones:

```
} catch (SQLException e) {  
    System.out.println(e.getMessage());  
} catch (ClassNotFoundException e) {  
    System.out.println(e.getMessage());  
} finally {  
    if (statement != null) {  
        statement.close();  
    }  
    if (dbConnection != null) {  
        dbConnection.close();  
    }  
}
```

Código 4. Excepciones de drivers.



### Enlace de interés...

Fuente: <http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc>

Inicialmente, nos encontramos con **SQLException**, dicha excepción es capturada si a la hora de ejecutar el método “executeQuery” algo va mal en base de datos, ya sea gramaticalmente, sintácticamente, etc.

La **excepción ClassNotFoundException** es lanzada y capturada en este punto si en nuestra línea: “Class.forName(DRIVER)”, el fichero del driver que le estamos indicando no encontrara la librería.

Por último, comentar y recordar que **la sentencia finally se ejecutará siempre**, hayamos capturado excepción o no. En esta, simplemente, se realizan los cierres de la clase *Statement* y del objeto *Connection* que, a su vez, en este punto pueden lanzar una excepción que será recogida y lanzada a la capa superior a través de la palabra clave “Throws” en la definición de nuestro método.



Vídeo 2. “Ejemplo conector”

<https://bit.ly/2WiHpDi>



## / 6. Ventajas e inconvenientes del uso de conectores

Una vez nombrados y explicados los cuatro tipos de conectores que nos podemos encontrar con más frecuencia, procedemos a estudiar sus ventajas e inconvenientes.

- **Drivers tipo 1**

- **Ventajas:**

- Solemos encontrarlos fácilmente, ya que se distribuyen con el paquete del lenguaje Java.
    - Acceso a gran cantidad de drivers ODBC.



- **Inconvenientes:**

- Rendimiento: demasiadas capas intermedias.
- Limitación de funcionalidad. (Características comunes de base de datos).
- No funcionan bien con applets. Problemas en navegadores.

- **Drivers tipo 2**

- **Ventajas:**

- Ofrecen rendimiento superior al de tipo 1. ya que son llamadas nativas.

- **Inconvenientes:**

- La librería de la BDD, forzosamente, se inicia en la parte de cliente. No se pueden usar en internet.
- Interfaz nativa Java. No movable entre plataformas.

- **Drivers tipo 3**

- **Ventajas:**

- No necesita librería del fabricante. No es necesario llevar al cliente este aspecto.
- Son los que mejor rendimiento dan en internet, muchas opciones de portabilidad y escalabilidad.

- **Inconvenientes:**

- Requieren de un código específico de BDD para la capa intermedia.

- **Drivers tipo 4**

- **Ventajas:**

- Buen rendimiento.
- No necesitan instalar un software especial ni en la parte del servidor, ni en la parte de cliente. Drivers de fácil acceso

- **Inconvenientes:**

- El usuario necesitará distinto software de conexión (driver) para cada base de datos.

## / 7. Caso práctico 2: “Accedo a distintas BDD”

**Planteamiento:** A José se le encarga un aplicativo, el cual tiene que tener acceso a bases de datos distintas, una Oracle y otra Mysql, con lo que necesitará implementar una capa de datos para soportar dichos accesos.

**Nudo:** Viendo la documentación, José estudia la clase *DriverManager* que puede serle de gran ayuda. ¿Cómo habría de implementarlo?



**Desenlace:** José tendrá que realizar una clase que realice las gestiones de conexión, e instancie cualquiera de los parámetros que se les vayan pasando de los diferentes drivers.

Para ello, en dicha clase, tendrá que implementar un método `getDBConnection()` en donde deberá pasar cuatro atributos por parámetro:

- Nombre del driver
- Url de conexión
- Usuario
- Contraseña

El método `getDBConnection()` podría ser así:

```
private Connection getDBConnection(String Driver, String url, String usuario, String password) {  
    Connection connection = null;  
    try {  
        Class.forName(Driver);  
        connection= DriverManager.getConnection(url, usuario, password);  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return connection;  
}
```

*Código 5. Ejemplo de conexiones.*

De esta forma, pasando esos cuatro parámetros, se podrán instanciar tantas conexiones con bases de datos como queramos o como sea necesario para la implementación de nuestro aplicativo.

En el caso de José, serán dos, una con los datos de la base de datos Oracle y otro con los datos de conexión de la base de datos Mysql.

## / 8. Resumen y resolución del caso práctico de la unidad

Después de haber estudiado esta unidad, ya conocemos el concepto de **conector**, lo que nos lleva a entender el concepto de **desfase objeto-relacional**, y cómo los aplicativos de distintos lenguajes tienen ciertas dificultades de comunicación a la hora de conectar con las bases de datos relacionales.

Para ello, hemos realizado el estudio de los **distintos protocolos**, deteniéndonos en el más usado, en este caso, JDBC.

Hemos profundizado, también, en los **distintos tipos de conectores o drivers** de dicho protocolo, y por último, hemos visto **cómo instalar un driver y las ventajas e inconvenientes** de los distintos tipos de conectores.

### Resolución del caso práctico inicial

Nuestro compañero Pedro tiene varios objetivos claros de cara a implementar la capa del modelado de datos de la aplicación web en la que está trabajando:



Carga del driver y almacenamiento de credenciales:

```
private static final String DRIVER = "nombre_del_Driver";  
private static final String URL_CONEXION = "url_de_la_base_De_datos";
```

Código 6. Credenciales 1.

En este punto, Pedro podrá **registrar la librería** .jar de la base de datos que haya introducido al classPath del proyecto, y también introducirá la cadena de conexión.

```
final String usuario = "usuario";  
final String password = "contraseña";
```

Código 7. User y pass.

Posteriormente, deberá **definir usuario y contraseña** en dos variables de tipo *string*. Si lo desea, también pueden ser variables globales.

```
Class.forName(DRIVER);  
Connection dbConnection = DriverManager.getConnection(URL_CONEXION, usuario, password)
```

Código 8. Instancia conexión.

Con el Código 8, **abriría la conexión**. Lo recomendable es que Pedro introduzca estas líneas en un método que se podría titular "getConnectionToDB" y nos devolvería la conexión ya establecida con nuestra URL de conexión, usuario y contraseña bien configurado y preparado para trabajar con la base de datos.

Sería algo así:

```
Class.forName(DRIVER);  
return DriverManager.getConnection(URL_CONEXION, usuario, password);
```

Código 9. Return connection.

## / 9. Bibliografía

Tendero, J. E. C. (2014). Acceso a Datos (GRADO SUPERIOR). Madrid. Grupo editorial ra-ma.

## / 10. Webgrafía

<https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/package-summary.html>

<http://decodigo.com/java-conexion-a-base-de-datos-con-jdbc>