

Programación

Acceso a BBDD

Unidad 14

Jesús Alberto Martínez
versión 0.2



Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.
Basado en los apuntes de WirzJava, del CEEDCV y de los apuntes de programación de Joan Arnedo Moreno (Institut Obert de Catalunya, IOC).



Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención




Interesante

Unidad 14. Acceso a Bases de Datos

1 Introducción.....	3
2 Repaso del lenguaje SQL.....	4
2.1 Comandos.....	4
2.2 Clausulas.....	5
2.3 Operadores.....	5
2.4 Ejemplos de sentencias SQL.....	6
3 JDBC.....	9
3.1 Funciones del JDBC.....	9
3.2 Drivers JDBC.....	10
4 Acceso a bases de datos mediante código Java.....	11
4.1 Creación de un proyecto Maven.....	11
4.2 Añadir la librería JDBC al proyecto.....	12
4.3 Cargar el Driver.....	13
4.4 Clase DriverManager.....	14
4.5 Clase Connection.....	15
4.6 Clase Statement.....	15
4.7 Clase ResultSet.....	16
4.8 Gestión adecuada de las conexiones.....	17
4.9 Ejemplo completo.....	17
5 Navegabilidad y concurrencia (Statement).....	18
6 Consultas (ResultSet).....	19
6.1 Navegación de un ResultSet.....	19
6.2 Obteniendo datos del ResultSet.....	20
6.3 Tipos de datos y conversiones.....	21
6.4 Modificación.....	21
6.5 Inserción.....	23
6.6 Borrado.....	24
7 PreparedStatement.....	24
8 Programa gestor de clientes (I).....	25
9 Programa gestor de clientes usando DAO y Pool de Conexiones (II).....	28
9.1 El patrón Arquitectónico "Data Access Object" (DAO).....	29
Pool de conexiones.....	30
Implementación del interfaz.....	30
El interfaz.....	33
10 Anexo. Otros temas interesantes.....	34
10.1 Patrón de diseño Singleton.....	34
10.2 SQLite.....	35
10.3 Mapeo Objeto-Relacional. Hibernate.....	35
10.4 Transacciones.....	36
10.5 RowSet.....	37

1 Introducción

 Una base de datos es una colección de datos clasificados y estructurados que son guardados en uno o varios ficheros, pero referenciados como si de un único fichero se tratara.

Para crear y manipular bases de datos relacionales, existen en el mercado varios sistemas de gestión de bases de datos (SGBD); por ejemplo, Access, SQL Server, Oracle y DB2. Otros SGBD de libre distribución son MySQL/MariaDB y PostgreSQL.

Los datos de una base de datos relacional se almacenan en tablas lógicamente relacionadas entre sí utilizando campos clave comunes. A su vez, cada tabla dispone los datos en filas y columnas. Por ejemplo, piensa en una relación de clientes, a las filas se les denomina tuplas o registros y a las columnas campos.

Los usuarios de un sistema administrador de bases de datos pueden realizar sobre una determinada base operaciones como insertar, recuperar, modificar y eliminar datos, así como añadir nuevas tablas o eliminarlas. Esta operaciones se expresan generalmente en un lenguaje denominado SQL.

Antes de empezar necesitamos un sistema de gestión de base de datos instalado en nuestro ordenador en el que poder tener bases de datos a las que conectarnos desde nuestros programas escritos en lenguaje Java. Si ya tiene uno instalado (por ejemplo MySQL, MariaDB, PostgreSQL, etc.) puedes utilizarlo si quieres. De todos modos en esta unidad os proponemos utilizar XAMPP (Window/Linux Apache MariaDB PHP Perl) que incorpora un servidor web y la herramienta phpMyAdmin para trabajar con bases de datos. Lo puedes descargar desde la dirección: <https://www.apachefriends.org> Para su instalación tan solo debes seguir los pasos del instalador.



Veréis que las versiones más actuales de XAMPP han sustituido MySQL por MariaDB. No pasa nada, ambos son compatibles y pueden utilizarse indistintamente

2 Repaso del lenguaje SQL

En este apartado se da un repaso a los aspectos más relevantes del lenguaje SQL de manipulación de bases de datos relacionales que ya habrás visto en el módulo de Bases de Datos. Te vendrá bien para repasar conceptos. Si no has cursado dicho módulo, es muy importante que leas este apartado con atención. No es necesario tener un conocimiento avanzado sobre el lenguaje SQL, pero sí los aspectos más básicos que se cubren en este apartado.

SQL incluye sentencias tanto de creación de datos (CREATE) como de manipulación de datos (INSERT, UPDATE y DELETE) así como de consulta de datos (SELECT).

La sintaxis del lenguaje SQL está formado por cuatro grupos sintácticos que pueden combinarse en varias de las sentencias más habituales. Estos 4 grupos son: Comandos, Cláusulas, Operadores y Funciones. En este apartado repasaremos solo los tres primeros.

2.1 Comandos

Existen tres tipos de comandos en SQL:

- Los DDL (Data Definition Language), que permiten crear, modificar y borrar nuevas bases de datos, tablas, campos y vistas.
- Los DML (Data Manipulation Language), que permiten introducir información en la BD, borrarla y modificarla.
- Los DQL (Data Query Language), que permiten generar consultas para ordenar, filtrar y extraer información de la base de datos.

Los comandos DDL son:

- *CREATE*: Crear nuevas tablas, campos e índices.
- *ALTER*: Modificación de tablas añadiendo campos o modificando la definición de los campos.
- *DROP*: Instrucción para eliminar tablas, campos e índices.

Los comandos DML son:

- *INSERT*: Insertar registros a la base de datos.
- *UPDATE*: Instrucción que modifica los valores de los campos y registros especificados en los criterios.
- *DELETE*: Eliminar registros de una tabla de la base de datos.

El principal comando DQL es:

- *SELECT*: Consulta de registros de la base de datos que cumplen un criterio determinado.

2.2 Clausulas

Las cláusulas son condiciones de modificación, utilizadas para definir los datos que se desean seleccionar o manipular:

- *FROM*: Utilizada para especificar la tabla de la que se seleccionarán los registros.
- *WHERE*: Cláusula para detallar las condiciones que deben reunir los registros resultantes.
- *GROUP BY*: Utilizado para separar registros seleccionados en grupos específicos.
- *HAVING*: Utilizada para expresar la condición que ha de cumplir cada grupo.
- *ORDER BY*: Utilizada para ordenar los registros seleccionados de acuerdo a un criterio dado.

2.3 Operadores

Operadores lógicos:

- *AND*: Evalúa dos condiciones y devuelve el valor cierto, si ambas condiciones son ciertas.
- *OR*: Evalúa dos condiciones y devuelve el valor cierto, si alguna de las dos condiciones es cierta.
- *NOT*: Negación lógica. Devuelve el valor contrario a la expresión.

Operadores de comparación:

- *< [...]* menor que [...]
- *> [...]* mayor que [...]
- *<> [...]* diferente a [...]
- *<= [...]* menor o igual que [...]
- *>= [...]* mayor o igual que [...]
- *= [...]* igual que [...]
- *BETWEEN*: Especifica un intervalo de valores
- *LIKE*: Compara un modelo
- *IN*: Operadores para especificar registros de una tabla

2.4 Ejemplos de sentencias SQL

Para crear una base de datos: *CREATE DATABASE* <nombre>

Ejemplo: *CREATE DATABASE* tiendaonline

Para eliminar una base de datos: *DROP DATABASE* <base de datos>

Ejemplo: *DROP DATABASE* tiendaonline

Para utilizar una base de datos: *USE* <base de datos>

Ejemplo: *USE* tiendaonline

NOTA: Con *USE* indicamos sobre qué base de datos queremos trabajar (podemos tener varias).

Para crear una tabla: *CREATE TABLE* <tabla> (<columna 1> [,<columna 2>] ...)

Donde <columna n> se formula según la siguiente sintaxis:

<columna n> <tipo de dato> [*DEFAULT* <expresion>] [<const 1> [<const2>]...]

La cláusula *DEFAULT* permite especificar un valor por omisión para la columna y, opcionalmente, para indicar la forma o característica de cada columna, se pueden utilizar las constantes: *NOT NULL*, *UNIQUE* o *PRIMARY KEY*.

La cláusula *PRIMARY KEY* se utiliza para definir la columna como clave principal de la tabla. Esto supone que la columna no puede contener valores nulos ni duplicados. Una tabla puede contener una sola restricción *PRIMARY KEY*.

La cláusula *UNIQUE* indica que la columna no permite valores duplicados. Una tabla puede tener varias restricciones *UNIQUE*.

Ejemplo:

```
CREATE TABLE clientes(  
  nombre CHAR(30) NOT NULL,  
  dirección CHAR(30) NOT NULL,  
  teléfono CHAR(12) PRIMARY KEY NOT NULL,  
  observaciones CHAR(240) )
```

Para borrar una tabla:

```
DROP TABLE <tabla>
```

Ejemplo:

```
DROP TABLE clientes
```

Para insertar registros en una tabla:

```
INSERT [INTO] <tabla> [(<columna 1>[,<columna 2>]...)]
```

```
VALUES (<expresion 1>[,<expresion 2>]...),...
```

Ejemplo: Insertamos dos nuevos registros en la tabla clientes

```
INSERT INTO clientes VALUES ('Pepito Pérez','VALENCIA','963003030','Ninguna'),  
('María Martínez','VALENCIA','961002030','Ninguna')
```

Para modificar registros ya existentes en una tabla:

```
UPDATE <tabla> SET <columna1 = (<expresion1> | NULL) [<columna2 = ...]>...
```

```
WHERE <condición de búsqueda>
```

Ejemplo: Cambiamos el campo 'dirección' del registro cuyo teléfono es '963003030'

```
UPDATE clientes SET direccion = 'Puerto de Sagunto'  
WHERE telefono = '963003030'
```

Para borrar registros de una tabla.

```
DELETE FROM <tabla> WHERE <condición de búsqueda>
```

Ejemplo: Borramos los registros cuyo campo 'telefono' es 963003030.

```
DELETE FROM clientes WHERE telefono='963003030'
```

Para realizar una consulta, es decir, obtener registros de una base de datos:

```
SELECT [ALL | DISTINCT] <lista de selección>
```

FROM <tablas>

WHERE <condiciones de selección>

[ORDER BY <columna1> [ASC|DESC][, <columna2>[ASC|DESC]]...]

Algunos ejemplos sencillos:

Obtenemos todos los registros de la tabla 'clientes'.

*SELECT * FROM clientes;*

Obtenemos solo el campo 'nombre' de todos los registros de la tabla 'clientes':

SELECT nombre FROM clientes;

Obtenemos solo los campos 'nombre' y 'telefono' de la tabla 'clientes':

SELECT nombre FROM clientes;

Obtenemos todos los registros de 'clientes' ordenados por nombre:

*SELECT * FROM clientes ORDER BY nombre;*

Obtenemos solo los nombres y teléfonos de los clientes, ordenados por nombre:

SELECT nombre, telefono FROM clientes ORDER BY nombre;

Obtenemos todos los registros de 'clientes' donde el campo 'telefono' es mayor que 1234

*SELECT * FROM clientes WHERE telefono > '1234';*

Obtenemos todos los registros de 'clientes' en los que el campo 'telefono' empieza por 91

*SELECT * FROM clientes WHERE telefono LIKE '91*';*

Para ver ejemplos más complejos y/o que impliquen datos de varias tablas podéis consultar este material complementario: <https://www.cs.us.es/blogs/bd2013/files/2013/09/Consultas-SQL.pdf>

3 JDBC

Java puede conectarse con distintos SGBD y en diferentes sistemas operativos. Independientemente del método en que se almacenen los datos debe existir siempre un mediador entre la aplicación y el sistema de base de datos y en Java esa función la realiza JDBC.



Para la conexión a las bases de datos utilizaremos la API estándar de JAVA denominada JDBC (Java Data Base Connection)

JDBC es un API incluido dentro del lenguaje Java para el acceso a bases de datos. Consiste en un conjunto de clases e interfaces escritas en Java que ofrecen un completo API para la programación con bases de datos, por lo tanto es la única solución 100% Java que permite el acceso a bases de datos.

JDBC es una especificación formada por una colección de interfaces y clases abstractas, que todos los fabricantes de drivers deben implementar si quieren realizar una implementación de su driver 100% Java y compatible con JDBC (JDBC-compliant driver). Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma.



No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, SQL Server, etc.

Además podrá ejecutarse en cualquier sistema operativo que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma. Otras APIS que se suelen utilizar bastante para el acceso a bases de datos son DAO (Data Access Objects) y RDO (Remote Data Objects), y ADO (ActiveX Data Objects), pero el problema que ofrecen estas soluciones es que sólo son para plataformas Windows.

JDBC tiene sus clases en el paquete `java.sql` y otras extensiones en el paquete `javax.sql`.

3.1 Funciones del JDBC

Básicamente el API JDBC hace posible la realización de las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular datos.
- Procesar los resultados de la ejecución de las sentencias.

3.2 Drivers JDBC

Los drivers nos permiten conectarnos con una base de datos determinada. Existen cuatro tipos de drivers JDBC, cada tipo presenta una filosofía de trabajo diferente. A continuación se pasa a comentar cada uno de los drivers:

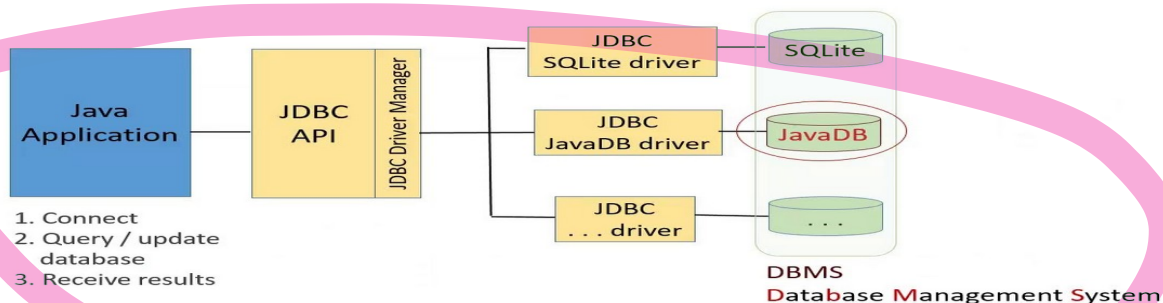
- JDBC-ODBC bridge plus ODBC driver (tipo 1): permite al programador acceder a fuentes de datos ODBC existentes mediante JDBC. El JDBC-ODBC Bridge (puente JDBC-ODBC) implementa operaciones JDBC traduciéndolas a operaciones ODBC, se encuentra dentro del paquete *sun.jdbc.odbc* y contiene librerías nativas para acceder a ODBC.

Al ser usuario de ODBC depende de las dll de ODBC y eso limita la cantidad de plataformas en donde se puede ejecutar la aplicación.

- Native-API partly-Java driver (tipo 2): son similares a los drivers de tipo 1, en tanto en cuanto también necesitan una configuración en la máquina cliente. Este tipo de driver convierte llamadas JDBC a llamadas de Oracle, Sybase, Informix, DB2 u otros SGBD. Tampoco se pueden utilizar dentro de applets al poseer código nativo.
- JDBC-Net pure Java driver (tipo 3): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos mediante el uso de una biblioteca JDBC en el servidor. La ventaja de esta opción es la portabilidad.

- JDBC de Java cliente (tipo 4): Estos controladores están escritos en Java y se encargan de convertir las llamadas JDBC a un protocolo independiente de la base de datos y en la aplicación servidora utilizan las funciones nativas del sistema de gestión de base de datos sin necesidad de bibliotecas. La **ventaja** de esta opción es la **portabilidad**. Son como los drivers de tipo 3 pero sin la figura del intermediario y tampoco requieren ninguna configuración en la máquina cliente. Los drivers de tipo 4 se pueden utilizar para servidores Web de tamaño pequeño y medio, así como para intranets.

JDBC - Java Database Connectivity



4 Acceso a bases de datos mediante código Java

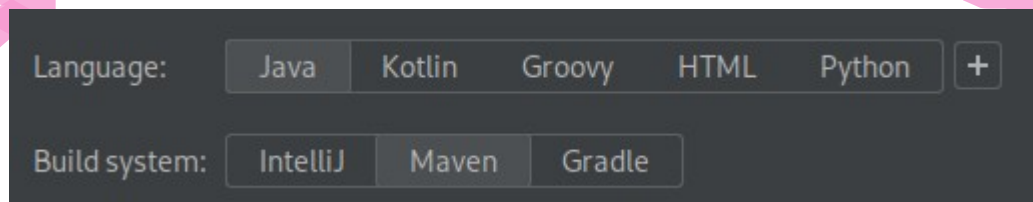
En este apartado se ofrece una introducción a los aspectos fundamentales del acceso a bases de datos mediante código Java. En los siguientes apartados se explicarán algunos aspectos en mayor detalle, sobre todo los relacionados con las clases Statement y ResultSet.

4.1 Creación de un proyecto Maven

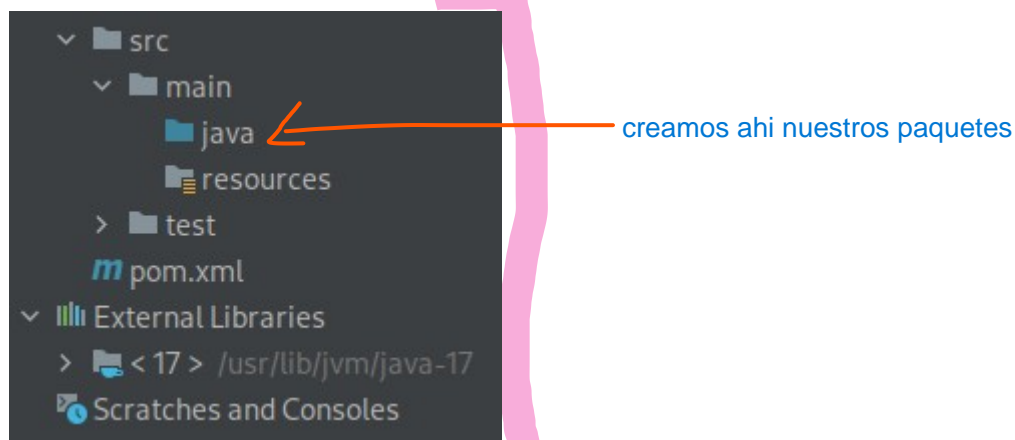
Maven es una **herramienta de gestión de proyectos** de software ampliamente utilizada en el desarrollo de aplicaciones Java. Proporciona una forma estructurada y eficiente de administrar las **dependencias**, compilar, empaquetar y distribuir proyectos Java. Maven sigue el enfoque de "Convención sobre configuración", lo que significa que se basa en una estructura de directorios predefinida y convenciones establecidas para simplificar y agilizar el proceso de desarrollo.

Es una **herramienta muy poderosa**, y su uso y funcionamiento escapa de los límites de nuestro módulo, pero para este tema vamos a usar la herramienta.

Para ello creamos un nuevo proyecto, pero esta vez seleccionamos como "Build system" a Maven.



Nos creará el proyecto con la estructura predeterminada de Maven, dentro del directorio src dos directorios, main y test. Dentro del directorio main dos directorios, java y resources. En el **directorio java** es donde **crearemos y editaremos las clases de nuestro programa**.



Y un fichero, **pom.xml**, este fichero es el fichero de configuración de Maven.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>Tema14</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

</project>
```

4.2 Añadir la librería JDBC al proyecto

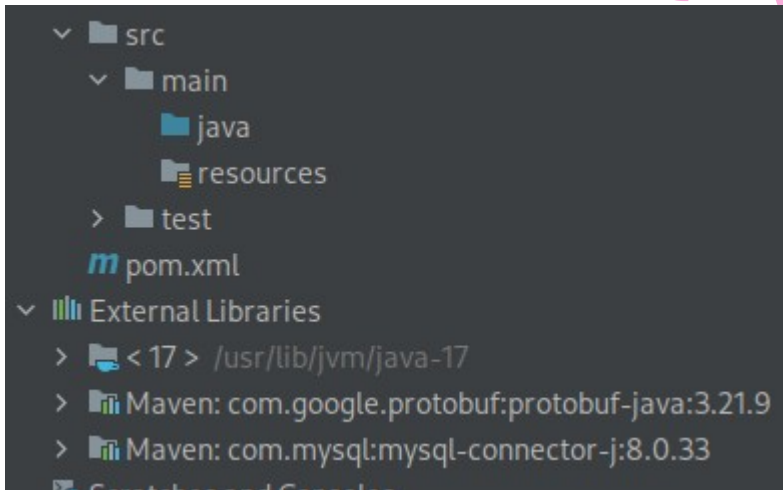
Para poder utilizar la librería JDBC en un proyecto Java primero deberemos añadirla al proyecto. Usar Maven nos va a facilitar esa tarea. En el fichero pom.xml debemos indicar que queremos usar una librería.

Para buscar cómo incorporar la librería en nuestro fichero, nos vamos a la página web <https://mvnrepository.com> y buscamos “mysql”. Seleccionamos “Mysql connector Java” y la versión que queremos (en este ejemplo la 8.0.33) y nos proporciona un código para incorporarlo en nuestro proyecto maven.

Copiamos este código y lo copiamos en la sección dependencies del fichero pom.xml. Tendremos que crear la sección si no está creada.

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
  </dependency>
</dependencies>
```

Ahora tenemos que hacer que Maven actualice los cambios, tenemos un botón con una M o **botón derecho sobre el proyecto**, opción **Maven**, **"Reload project"**. Después de realizar esta acción tenemos incorporado la librería JDBC a nuestro proyecto.



Maven se ha encargado de descargar las librerías indicadas.

La otra opción sin Maven es descargar el fichero JAR correspondiente (lo podemos bajar desde esta misma web) e incorporarlo como librería a nuestro proyecto.

4.3 Cargar el Driver

En un proyecto Java que realice conexiones a bases de datos es necesario, antes que nada, utilizar `Class.forName(...)` para cargar dinámicamente el Driver que vamos a utilizar. Esto solo es necesario hacerlo una vez en nuestro programa. Puede lanzar excepciones por lo que es necesario utilizar un bloque `try-catch`.

```
try {  
    Class.forName("com.mysql.cj.jdbc.Driver");  
} catch (Exception e) {  
    // manejamos el error  
}
```

Hay que tener en cuenta que las clases y métodos utilizados para conectarse a una base de datos (explicados más adelante) funcionan con todos los drivers disponibles para Java (JDBC es solo uno, hay muchos más). Esto es posible ya que el estándar de Java solo los define como interfaces (interface) y cada librería driver los implementa (define las clases y su código). Por ello es necesario utilizar `Class.forName(...)` para indicarle a Java qué driver vamos a utilizar.

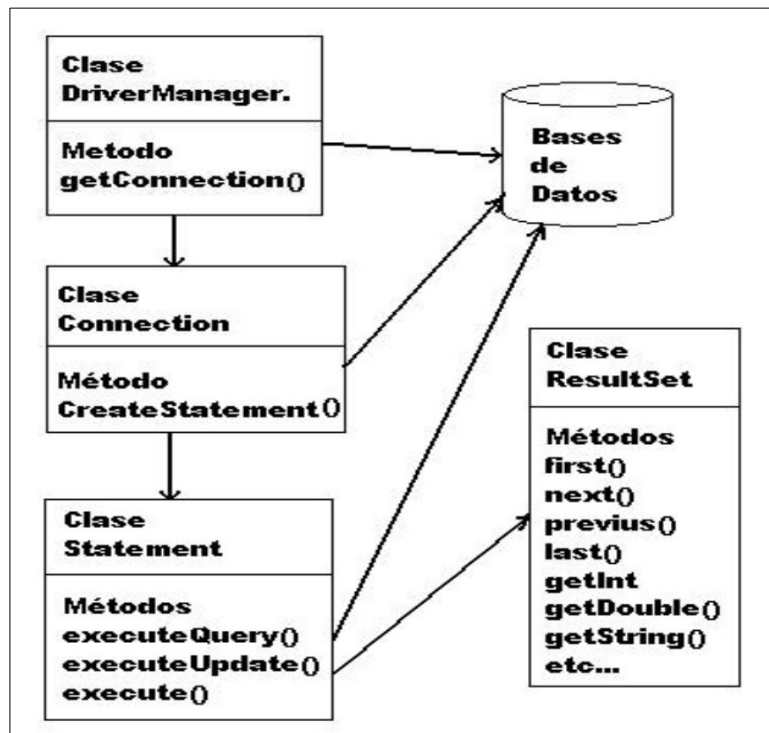
Este nivel de abstracción facilita el desarrollo de proyectos ya que si necesitáramos utilizar otro sistema de base de datos (que no fuera MySQL) solo necesitaríamos cambiar la línea de código que carga el driver y

poco más. Si cada sistema de base de datos necesitara que utilizáramos distintas clases y métodos todo sería mucho más complicado.

Las cuatro clases fundamentales que toda aplicación Java necesita para conectarse a una base de datos y ejecutar sentencias son: DriverManager, Connection, Statement y ResultSet.



Esta fase ya no es necesaria desde Java 8



4.4 Clase DriverManager

La clase `java.sql.DriverManager` es la capa gestora del driver JDBC. Se encarga de manejar el Driver apropiado y permite crear conexiones con una base de datos mediante el método estático `getConnection(...)` que tiene dos variantes:

```

DriverManager.getConnection(String url)
DriverManager.getConnection(String url, String user, String password)
  
```

Este método intentará establecer una conexión con la base de datos según la URL indicada. Opcionalmente se le puede pasar el usuario y contraseña como argumento (también se puede indicar en la propia URL). Si la conexión es satisfactoria devolverá un objeto `Connection`.

Ejemplo de conexión a la base de datos 'prueba' en localhost:

```
String url = "jdbc:mysql://localhost:3306/prueba";  
Connection conn = DriverManager.getConnection(url, "root", "");
```

Este método puede lanzar dos tipos de excepciones (que habrá que manejar con un try-catch):

- `SQLException`: La conexión no ha podido producirse. Puede ser por multitud de motivos como una URL mal formada, un error en la red, host o puerto incorrecto, base de datos no existente, usuario y contraseña no válidos, etc.
- `SQLException`: Se ha superado el `LoginTimeout` sin recibir respuesta del servidor.

4.5 Clase Connection

Un objeto `java.sql.Connection` representa una sesión de conexión con una base de datos. Una aplicación puede tener tantas conexiones como necesite, ya sea con una o varias bases de datos.

El método más relevante es `createStatement()` que devuelve un objeto `Statement` asociado a dicha conexión que permite ejecutar sentencias SQL. El método `createStatement()` puede lanzar excepciones de tipo `SQLException`.

```
Statement stmt = conn.createStatement();
```

Cuando ya no la necesitemos es aconsejable cerrar la conexión con `close()` para liberar recursos.

```
conn.close();
```

4.6 Clase Statement

Un objeto `java.sql.Statement` permite ejecutar sentencias SQL en la base de datos a través de la conexión con la que se creó el `Statement` (ver apartado anterior). Los tres métodos más comunes de ejecución de sentencias SQL son `executeQuery(...)`, `executeUpdate(...)` y `execute(...)`. Pueden lanzar excepciones de tipo `SQLException` y `SQLException`.

- `ResultSet executeQuery(String sql)`: Ejecuta la sentencia `sql` indicada (de tipo `SELECT`). Devuelve un objeto `ResultSet` con los datos proporcionados por el servidor.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM vendedores");
```

- `int executeUpdate(String sql)`: Ejecuta la sentencia `sql` indicada (de tipo `DML` como por ejemplo `INSERT`, `UPDATE` o `DELETE`). Devuelve un el número de registros que han sido insertados, modificados o eliminados.

```
int nr = stmt.executeUpdate("INSERT INTO vendedores VALUES (1,  
                        'Pedro Gil', '2017-04-11', 15000);")
```

Cuando ya no lo necesitemos es aconsejable cerrar el statement con close() para liberar recursos.

```
stmt.close();
```

4.7 Clase ResultSet

Un objeto `java.sql.ResultSet` contiene un conjunto de resultados (datos) obtenidos tras ejecutar una sentencia SQL, normalmente de tipo SELECT. Es una estructura de datos en forma de tabla con registros (filas) que podemos recorrer para acceder a la información de sus campos (columnas).

ResultSet utiliza internamente un cursor que apunta al 'registro actual' sobre el que podemos operar. Inicialmente dicho cursor está situado antes de la primera fila y disponemos de varios métodos para desplazar el cursor. El más común es next():

- `boolean next()`: Mueve el cursor al siguiente registro. Devuelve true si fué posible y false en caso contrario (si ya llegamos al final de la tabla).

Algunos de los métodos para obtener los datos del registro actual son:

- `String getString(String columnLabel)`: Devuelve un dato String de la columna indicada por su nombre. Por ejemplo: `rs.getString("nombre")`
- `String getString(int columnIndex)`: Devuelve un dato String de la columna indicada por su nombre. La primera columna es la 1, no la cero. Por ejemplo: `rs.getString(2)`

Existen métodos análogos a los anteriores para obtener valores de tipo int, long, float, double, boolean, Date, Time, Array, etc. Pueden consultarse todos en la [documentación oficial de Java](#).

- | | |
|---|--|
| • <code>int getInt(String columnLabel)</code> | <code>int getInt(int columnIndex)</code> |
| • <code>double getDouble(String columnLabel)</code> | <code>double getDouble(int columnIndex)</code> |
| • <code>boolean getBoolean(String columnLabel)</code> | <code>boolean getBoolean(int columnIndex)</code> |
| • <code>Date getDate(String columnLabel)</code> | <code>int getDate(int columnIndex)</code> |
| • etc. | |

Más adelante veremos cómo se realiza la modificación e inserción de datos.

Todos estos métodos pueden lanzar una `SQLException`.

Veamos un ejemplo de cómo recorrer un ResultSet llamado rs y mostrarlo por pantalla:


```
while(rs.next()) {  
    int id      = rs.getInt("id");  
    String nombre = rs.getString("nombre");  
    Date fecha   = rs.getDate("fecha_ingreso");  
    float salario = rs.getFloat("salario");  
    System.out.println(id + " " + nombre + " " + fecha + " " + salario);  
}
```

4.8 Gestión adecuada de las conexiones

Como hemos visto anteriormente, conexiones, sentencias y conjuntos de resultados, (connection, statement, resultSet) tienen que cerrarse una vez que se han usado.

Esto debe realizarse por varios motivos:

- Eficiencia y rendimiento
- Recursos limitados
- Mantenimiento de la integridad de los datos
- Seguridad

Para realizar una gestión adecuada de las conexiones se recomienda usar:

- try-with-resources.
- pools de conexiones
- control de transacciones
- sentencias precompiladas (preparedStatement)

4.9 Ejemplo completo

Veamos un ejemplo completo de conexión y acceso a una base de datos utilizando todos los elementos mencionados en este apartado.

```
String url = "jdbc:mysql://localhost:3306/tienda";  
String sentenciasql="select * from clientes";  
try (Connection conn = DriverManager.getConnection(url,"root","");  
    Statement st= conn.createStatement();  
    ResultSet rs=st.executeQuery(sentenciasql)){  
  
    JOptionPane.showMessageDialog(null,"Conexión establecida ",  
        "Tema14",JOptionPane.INFORMATION_MESSAGE);
```

```
while (rs.next()){
    System.out.print("Id: "+rs.getInt(1));
    System.out.print(" Nombre: "+rs.getString("nombre"));
    System.out.println(" Dirección: "+rs.getString(3));
}

} catch (SQLException e) {
    JOptionPane.showMessageDialog(null,"Error en la conexión a la base de datos");
    e.printStackTrace();
}
```

5 Navegabilidad y concurrencia (Statement)

Cuando invocamos a *createStatement()* sin argumentos, como hemos visto anteriormente, al ejecutar sentencias SQL obtendremos un *ResultSet* por defecto en el que el cursor solo puede moverse hacia adelante y los datos son de solo lectura. A veces esto no es suficiente y necesitamos mayor funcionalidad.

Por ello el método *createStatement()* está sobrecargado (existen varias versiones de dicho método) lo cual nos permite invocarlo con argumentos en los que podemos especificar el funcionamiento.

- *Statement createStatement(int resultSetType, int resultSetConcurrency)*: Devuelve un objeto *Statement* cuyos objetos *ResultSet* serán del tipo y concurrencia especificados. Los valores válidos son constantes definidas en *ResultSet*.

El argumento *resultSetType* indica el tipo de *ResultSet*:

- *ResultSet.TYPE_FORWARD_ONLY*: *ResultSet* por defecto, forward-only y no-actualizable.
 - Solo permite movimiento hacia delante con *next()*.
 - Sus datos NO se actualizan. Es decir, no reflejará cambios producidos en la base de datos. Contiene una instantánea del momento en el que se realizó la consulta.
- *ResultSet.TYPE_SCROLL_INSENSITIVE*: *ResultSet* desplazable y no actualizable.
 - Permite libertad de movimiento del cursor con otros métodos como *first()*, *previous()*, *last()*, etc. además de *next()*.
 - Sus datos NO se actualizan, como en el caso anterior.
- *ResultSet.TYPE_SCROLL_SENSITIVE*: *ResultSet* desplazable y actualizable.
 - Permite libertad de movimientos del cursor, como en el caso anterior.
 - Sus datos SÍ se actualizan. Es decir, mientras el *ResultSet* esté abierto se actualizará automáticamente con los cambios producidos en la base de datos. Esto puede suceder incluso mientras se está recorriendo el *ResultSet*, lo cual puede ser conveniente o contraproducente según el caso.

El argumento `resultSetConcurrency` indica la concurrencia del `ResultSet`:

- `ResultSet.CONCUR_READ_ONLY`: Solo lectura. Es el valor por defecto.
- `ResultSet.CONCUR_UPDATABLE`: Permite modificar los datos almacenados en el `ResultSet` para luego aplicar los cambios sobre la base de datos (más adelante se verá cómo).

El `ResultSet` por defecto que se obtiene con `createStatement()` sin argumentos es el mismo que con `createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY)`.

6 Consultas (ResultSet)

6.1 Navegación de un ResultSet

Como ya se ha visto, en un objeto `ResultSet` se encuentran los resultados de la ejecución de una sentencia SQL. Por lo tanto, un objeto `ResultSet` contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece métodos de navegación por los registros como `next()` que desplaza el curso al siguiente registro del `ResultSet`.

Además de este método de desplazamiento básico, existen otros de desplazamiento libre que podremos utilizar siempre y cuando el `ResultSet` sea de tipo `ResultSet.TYPE_SCROLL_INSENSITIVE` o `ResultSet.TYPE_SCROLL_SENSITIVE` como se ha dicho antes.

Algunos de estos métodos son:

- `void beforeFirst()`: Mueve el cursor antes de la primera fila.
- `boolean first()`: Mueve el cursor a la primera fila.
- `boolean next()`: Mueve el cursor a la siguiente fila. Permitido en todos los tipos de `ResultSet`.
- `boolean previous()`: Mueve el cursor a la fila anterior.
- `boolean last()`: Mueve el cursor a la última fila.
- `void afterLast()`: Mover el cursor después de la última fila.
- `boolean absolute(int row)`: Posiciona el cursor en el número de registro indicado. Hay que tener en cuenta que el primer registro es el 1, no el cero. Por ejemplo `absolute(7)` desplazará el cursor al séptimo registro. Si el valor es negativo se posiciona en el número de registro indicado pero empezando a contar desde el final (el último es el -1). Por ejemplo si tiene 10 registros y llamamos `absolute(-2)` se desplazará al registro n.º 9.

- `boolean relative(int registros)`: Desplaza el cursor un número relativo de registros, que puede ser positivo o negativo. Por ejemplo si el cursor está en el registro 5 y llamamos a `relative(10)` se desplazará al registro 15. Si luego llamamos a `relative(-4)` se desplazará al registro 11.

Los métodos que devuelven un tipo `boolean` devolverán `'true'` si ha sido posible mover el cursor a un registro válido, y `'false'` en caso contrario, por ejemplo si no tiene ningún registro o hemos saltado a un número de registro que no existe.

Todos estos métodos pueden producir una excepción de tipo `SQLException`.

También existen otros métodos relacionados con la posición del cursor.

- `int getRow()`: Devuelve el número de registro actual. Cero si no hay registro actual.
- `boolean isBeforeFirst()`: Devuelve `'true'` si el cursor está antes del primer registro.
- `boolean isFirst()`: Devuelve `'true'` si el cursor está en el primer registro.
- `boolean isLast()`: Devuelve `'true'` si el cursor está en el último registro.
- `boolean isAfterLast()`: Devuelve `'true'` si el cursor está después del último registro.

6.2 Obteniendo datos del ResultSet

Los métodos `getXXX()` ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del *ResultSet*. No es necesario que las columnas sean obtenidas utilizando un orden determinado.

Para designar una columna podemos utilizar su nombre o bien su número (empezando por 1).

Por ejemplo si la segunda columna de un objeto *ResultSet* se llama "título" y almacena datos de tipo `String`, se podrá recuperar su valor de las dos formas siguientes:

```
// rs es un objeto ResultSet
String valor = rs.getString(2);
String valor = rs.getString("titulo");
```

Es importante tener en cuenta que las columnas se numeran de izquierda a derecha y que la primera es la número 1, no la cero. También que las columnas no son case sensitive, es decir, no distinguen entre mayúsculas y minúsculas.



La información referente a las columnas de un *ResultSet* se puede obtener llamando al método `getMetaData()` que devolverá un objeto *ResultSetMetaData* que contendrá el número, tipo y propiedades de las columnas del *ResultSet*.

Si conocemos el nombre de una columna, pero no su índice, el método *findColumn()* puede ser utilizado para obtener el número de columna, pasándole como argumento un objeto *String* que sea el nombre de la columna correspondiente, este método nos devolverá un entero que será el índice correspondiente a la columna.

6.3 Tipos de datos y conversiones

Cuando se lanza un método *getXXX()* determinado sobre un objeto *ResultSet* para obtener el valor de un campo del registro actual, el driver JDBC convierte el dato que se quiere recuperar al tipo Java especificado y entonces devuelve un valor Java adecuado. Por ejemplo si utilizamos el método *getString()* y el tipo del dato en la base de datos es *VARCHAR*, el driver JDBC convertirá el dato *VARCHAR* de tipo SQL a un objeto *String* de Java.

Algo parecido sucede con otros tipos de datos SQL como por ejemplo *DATE*. Podremos acceder a él tanto con *getDate()* como con *getString()*. La diferencia es que el primero devolverá un objeto Java de tipo *Date* y el segundo devolverá un *String*.

Siempre que sea posible el driver JDBC convertirá el tipo de dato almacenado en la base de datos al tipo solicitado por el método *getXXX()*, pero hay conversiones que no se pueden realizar y lanzarán una excepción, como por ejemplo si intentamos hacer un *getInt()* sobre un campo que no contiene un valor numérico.

6.4 Modificación

Para poder modificar los datos que contiene un *ResultSet* necesitamos un *ResultSet* de tipo modificable. Para ello debemos utilizar la constante *ResultSet.CONCUR_UPDATABLE* al llamar al método *createStatement()* como se ha visto antes.

Para modificar los valores de un registro existente se utilizan una serie de métodos *updateXXX()* de *ResultSet*. Las *XXX* indican el tipo del dato y hay tantos distintos como sucede con los métodos *getXXX()* de este mismo interfaz: *updateString()*, *updateInt()*, *updateDouble()*, *updateDate()*, etc.

La diferencia es que los métodos *updateXXX()* necesitan dos argumentos:

- La columna que deseamos actualizar (por su nombre o por su número de columna).
- El valor que queremos almacenar en dicha columna (del tipo que sea).

Por ejemplo para modificar el campo 'edad' almacenando el entero 28 habría que llamar al siguiente método, suponiendo que *rs* es un objeto *ResultSet*:

```
rs.updateInt("edad", 28);
```

También podría hacerse de la siguiente manera, suponiendo que la columna "edad" es la segunda:

```
rs.updateInt(2, 28);
```

Los métodos updateXXX() no devuelven ningún valor (son de tipo void). Si se produce algún error se lanzará una SQLException.

Posteriormente hay que llamar a updateRow() para que los cambios realizados se apliquen sobre la base de datos. El Driver JDBC se encargará de ejecutar las sentencias SQL necesarias. Esta es una característica muy potente ya que nos facilita enormemente la tarea de modificar los datos de una base de datos. Este método devuelve void.

En resumen, el proceso para realizar la modificación de una fila de un ResultSet es el siguiente:

1. Desplazamos el cursor al registro que queremos modificar.
2. Llamamos a todos los métodos updateXXX(...) que necesitemos.
3. Llamamos a updateRow() para que los cambios se apliquen a la base de datos.

Es importante entender que hay que llamar a updateRow() antes de desplazar el cursor. Si desplazamos el cursor antes de llamar a updateRow(), se perderán los cambios.

Si queremos cancelar las modificaciones de un registro del ResultSet podemos llamar a cancelRowUpdates(), que cancela todas las modificaciones realizadas sobre el registro actual.

Si ya hemos llamado a updateRow() el método cancelRowUpdates() no tendrá ningún efecto.

El siguiente código de ejemplo muestra cómo modificar el campo 'dirección' del último registro de un ResultSet que contiene el resultado de una SELECT sobre la tabla de clientes. Supondremos que conn es un objeto Connection previamente creado:

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

// Vamos al último registro, lo modificamos y actualizamos la base de datos
rs.last();
rs.updateString("direccion", "C/ Pepe Ciges, 3");
rs.updateRow();
```

6.5 Inserción

Para insertar nuevos registros necesitaremos utilizar, al menos, estos dos métodos:

- `void moveToInsertRow()`: Desplaza el cursor al 'registro de inserción'. Es un registro especial utilizado para insertar nuevos registros en el `ResultSet`. Posteriormente tendremos que llamar a los métodos `updateXXX()` ya conocidos para establecer los valores del registro de inserción. Para finalizar hay que llamar a `insertRow()`.
- `void insertRow()`: Inserta el 'registro de inserción' en el `ResultSet`, pasando a ser un registro normal más, y también lo inserta en la base de datos.

El siguiente código inserta un nuevo registro en la tabla 'clientes'. Supondremos que `conn` es un objeto `Connection` previamente creado:

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);

// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

// Creamos un nuevo registro y lo insertamos
rs.moveToInsertRow();
rs.updateString(2, "Killy Lopez");
rs.updateString(3, "Wall Street 3674");
rs.insertRow();
```

Los campos cuyo valor no se haya establecido con `updateXXX()` tendrán un valor `NULL`. Si en la base de datos dicho campo no está configurado para admitir nulos se producirá una `SQLException`.

Tras insertar nuestro nuevo registro en el objeto `ResultSet` podremos volver a la anterior posición en la que se encontraba el cursor (antes de invocar `moveToInsertRow()`) llamando al método `moveToCurrentRow()`. Este método sólo se puede utilizar en combinación con `moveToInsertRow()`.

6.6 Borrado

Para eliminar un registro solo hay que desplazar el cursor al registro deseado y llamar al método:

- `void deleteRow()`: Elimina el registro actual del `ResultSet` y también de la base de datos.

El siguiente código borra el tercer registro de la tabla 'clientes':

```
// Creamos un Statement scrollable y modificable
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                       ResultSet.CONCUR_UPDATABLE);


// Ejecutamos un SELECT y obtenemos la tabla clientes en un ResultSet
String sql = "SELECT * FROM clientes";
ResultSet rs = stmt.executeQuery(sql);

// Desplazamos el cursor al tercer registro
rs.absolute(3)
rs.deleteRow();
```

7 PreparedStatement

Son una extensión de `Statement` que previene la inyección de código con una forma distinta de componer el `String` que forma la consulta, de forma que JDBC "precompilará" la consulta antes de enviarla. Esta clase dispone igualmente de los métodos `executeQuery()` y `executeUpdate()`. Se genera así:

```
PreparedStatement ps = conexion.prepareStatement();
```

 La inyección de código es una técnica de ataque a una base de datos que se basa en introducir (inyectar) código malicioso en las partes dinámicas de la consulta, por ejemplo, valores solicitados al usuario. Ejemplo: Supón que construyes esta cadena que será una consulta en tu base de datos:

```
String consulta = " select * from tabla where usuario = '" + user + "' " en
```

la que `user` es una variable que introduce el usuario, con intención de que la consulta similar a:

```
select * from tabla where usuario = 'Pepe'
```

Pero imagina que el usuario introdujese: `xxx' or '1'='1` , el resultado final sería:

```
select * from tabla where usuario = 'xxx' or '1'='1'
```

produciendo un resultado totalmente diferente.

Ejemplo.

Definimos la sentencia

```
PreparedStatement pstmt=conexion.prepareStatement("select * from country  
where code=?");
```

Establecemos el valor de los parámetros, cada parámetro se representa por una interrogación, usando `setString`, `setBoolean`, `setDouble`, etc. donde el primer parámetro es la posición del parámetro y el segundo el valor.

```
pstmt.setString(1,"BEL");
```

Y ejecutamos la sentencia con `executeQuery()` o `executeUpdate()` según corresponda

```
ResultSet resultado=pstmt.executeQuery();
```

A tener en cuenta:

- Aunque los parámetros sean textos o fechas, no tenemos que preocuparnos de las comillas en la consulta, al sustituir la interrogación por el valor, JDBC se encarga de ello.
- Si el parámetro es una fecha, el valor debe ser de tipo `java.sql.Date`, no `LocalDate`, por lo que si tuviésemos una variable llamada 'fec' de tipo `LocalDate`, la llamada debería ser:

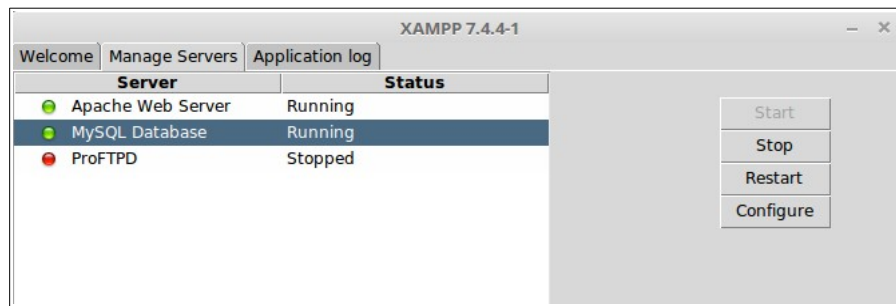
```
pstmt.setDate(1, java.sql.Date.valueOf(fec));
```

8 Programa gestor de clientes (I)

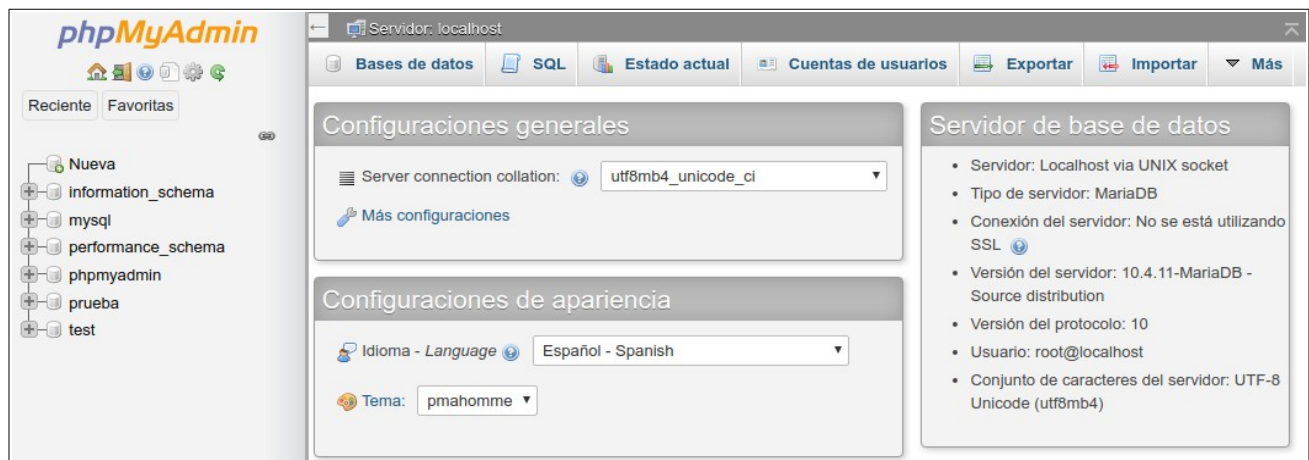
Veamos un ejemplo de un programa de gestión de clientes de una tienda, con interfaz gráfica de usuario y acceso a base de datos MySQL. Este programa permitirá consultar y modificar la información de los clientes así como darlos de alta y de baja. El código del proyecto y la base de datos pueden encontrarse en el aula virtual.

En la base de datos tendremos una sola tabla con los campos id, nombre y dirección. Cabe destacar que el campo id es autoincremental, es decir que su valor se incrementará al insertar un registro en la tabla, por lo que al crear nuevos registros no es necesario darle un valor.

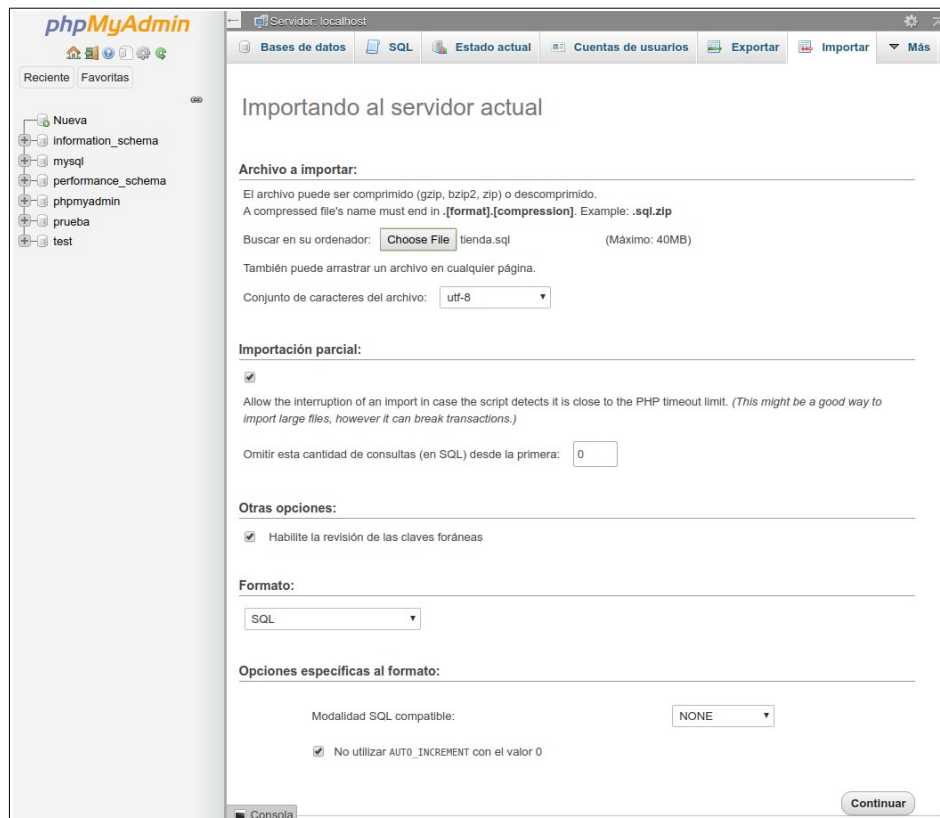
Primero necesitamos importar la base de datos a nuestro servidor MySQL. Abrimos el panel de control XAMPP y nos aseguramos de que Apache y MySQL están funcionando. En caso contrario los iniciamos.



Abrimos un navegador web, vamos a <http://localhost> y accedemos a phpMyAdmin. También podemos acceder directamente con <http://localhost/phpmyadmin/>. Esta aplicación web alojada en nuestro servidor Apache nos permite interactuar con el servidor MySQL de una forma sencilla.

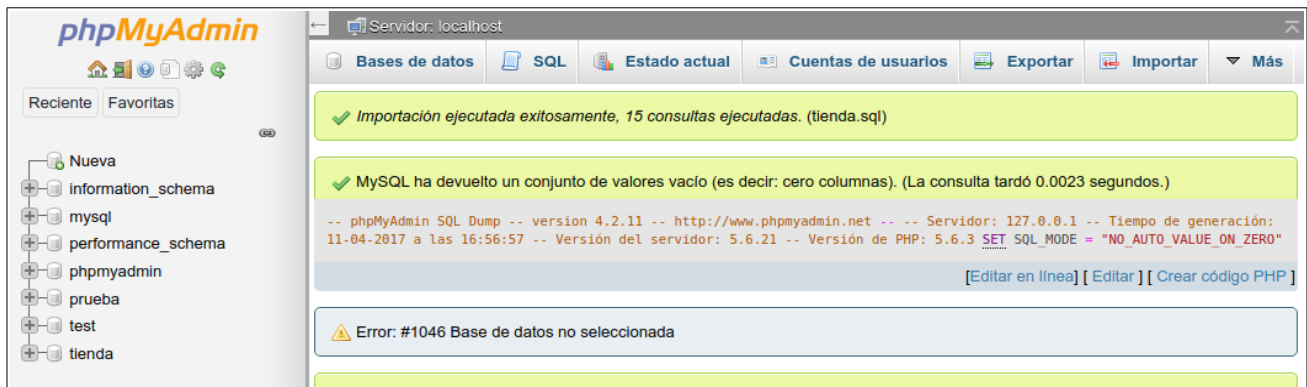


A continuación hacemos clic en 'Importar' y seleccionamos el archivo 'tienda.sql' que podéis descargar del aula virtual. Este archivo contiene las sentencias sql para crear la tabla que necesitamos.



The screenshot shows the 'Importar' (Import) screen in phpMyAdmin. The left sidebar shows a database tree with 'tienda' selected. The main area is titled 'Importando al servidor actual' (Importing to the current server). Under 'Archivo a importar:' (File to import:), it says 'El archivo puede ser comprimido (gzip, bzip2, zip) o descomprimido. A compressed file's name must end in .[format].[compression]. Example: .sql.zip'. The 'Buscar en su ordenador:' (Search on your computer:) field contains 'tienda.sql' and has a 'Choose File' button. Below it, 'Conjunto de caracteres del archivo:' (File character set:) is set to 'utf-8'. The 'Importación parcial:' (Partial import:) section has a checked checkbox and a text box for 'Omitir esta cantidad de consultas (en SQL) desde la primera:' (Skip this number of queries (in SQL) from the first:), with the value '0'. The 'Otras opciones:' (Other options:) section has a checked checkbox for 'Habilite la revisión de las claves foráneas' (Enable foreign key checks). The 'Formato:' (Format:) dropdown is set to 'SQL'. The 'Opciones específicas al formato:' (Format-specific options:) section has a 'Modalidad SQL compatible:' (SQL compatibility mode:) dropdown set to 'NONE' and a checked checkbox for 'No utilizar AUTO_INCREMENT con el valor 0' (Do not use AUTO_INCREMENT with the value 0). A 'Continuar' (Continue) button is at the bottom right.

Hacemos clic en 'Continuar' y si todo va bien la base de datos 'tienda' se importará correctamente.



The screenshot shows the phpMyAdmin interface after the import. The left sidebar shows the database tree with 'tienda' selected. The main area shows a green success message: 'Importación ejecutada exitosamente, 15 consultas ejecutadas. (tienda.sql)' (Import executed successfully, 15 queries executed. (tienda.sql)). Below it, another green message says: 'MySQL ha devuelto un conjunto de valores vacío (es decir: cero columnas). (La consulta tardó 0.0023 segundos.)' (MySQL has returned an empty set of values (i.e.: zero columns). (The query took 0.0023 seconds.)). Below these, there is a blue box with a warning icon and the text: 'Error: #1046 Base de datos no seleccionada' (Error: #1046 Database not selected). At the bottom, there is a green bar.

Ya tenemos la base de datos en nuestro servidor MySQL.

A Java Swing window titled "Gestor de clientes" with a standard Mac OS X title bar (minimize, maximize, close buttons). The window has a light gray background. At the top, there are four small square buttons with three dots inside, followed by three larger buttons labeled "Nuevo", "Editar", and "Borrar". Below these buttons are three text input fields: "id:" followed by a small box, "Nombre:" followed by a medium-width box, and "Dirección:" followed by a long-width box. At the bottom of the window are two buttons labeled "Aceptar" and "Cancelar".

⚡ En el aula virtual podéis encontrar el código a modo de ejemplo. Este ejemplo usa ResultSet para toda la gestión.

9 Programa gestor de clientes usando DAO y Pool de Conexiones (II)

En el aula virtual tenemos otro programa de gestión de Clientes pero esta vez usando DAO y un Pool de Conexiones.

A Java Swing window titled "Gestión de clientes (2)" with a standard Mac OS X title bar. The window has a light gray background. At the top, there is a dropdown menu labeled "Clientes:". Below the dropdown are three text input fields: "Id:", "Nombre:", and "Dirección:". At the bottom of the window are three buttons labeled "Nuevo", "Editar", and "Borrar".

9.1 El patrón Arquitectónico “Data Access Object” (DAO)

El patrón Arquitectónico “Data Access Object” (DAO) permite separar la lógica de acceso a datos de la lógica de negocio, de tal forma que el DAO encapsula y aísla el proceso de acceso a datos del resto de la aplicación.

Si recordáis, cuando hablamos de interfaz gráfico, tratábamos de separar la parte de interacción con el usuario de la lógica del programa, para poder utilizar la parte lógica en otros entornos. Esto es algo similar, pero en relación con el acceso a la base de datos. Así, sería muy fácil cambiar de gestor e incluso de tipo de base de datos (BD orientada a objetos, BD NoSQL, etc...) ya que el resto de la aplicación no se vería afectada.

Por otra parte, en las actualizaciones y correcciones referente al acceso a la base datos, es más fácil localizar el código afectado.

La idea sería entonces tener una clase que agrupase todas estas operaciones. Supongamos que tenemos una aplicación que gestiona una clase llamada Cliente.

```
public class Cliente {
    long id;
    String nombre;
    String direccion;

    public Cliente(long id, String nombre, String direccion) {
        this.id = id;
        this.nombre = nombre;
        this.direccion = direccion;
    }
    // Junto con getters/setters
}
```

y que tenemos los datos de esa clase en una base datos, en una tabla llamada Cliente.

Vamos a definir un interfaz, Dao, que defina las operaciones que se deben realizar con los datos. Vamos a definirla de una forma parametrizada, para permitir que se pudiera usar con otras clases.

```
public interface Dao <T>{
    Optional<T> obtener(long id);
    List<T> obtenerTodos();
    void guardar(T t);
    void actualizar(T t);
    void borrar(T t);
}
```

Esa <T> significa que vamos a poder indicar el tipo de datos, la clase, que vamos a usar. Nos fijamos en las operaciones que define la interfaz, operaciones que podemos usar en aplicaciones CRUD (Create – Read – Update – Delete) es decir, de manejo de datos. Pero esta interfaz no define como se implementa,

podemos tener una clase que la implemente sobre una base de datos, y otra clase que lo implemente sobre ficheros serializados, etc.

Vamos a ver la definición sobre una base de datos MySQL del Dao para Cliente.

```
public class ClienteDaoMysql implements Dao<Cliente>{  
    // Un DataSource es un intermediario para obtener las conexiones con la base de datos  
    private DataSource dataSource;
```

Pool de conexiones

Un DataSource es una interfaz que va definir una forma de gestionar las conexiones de forma automática, define lo que se conoce como un "pool de conexiones".

En un "pool de conexiones", se crean y mantienen varias conexiones abiertas a la base de datos, en lugar de abrir y cerrar una conexión para cada solicitud individual. Estas conexiones se almacenan en un "pool" y se pueden compartir entre múltiples hilos o procesos.

Cuando una aplicación necesita realizar una operación en la base de datos, toma una conexión del pool de conexiones disponible. Una vez que se completa la operación, la conexión se devuelve al pool para que esté disponible para ser reutilizada por otras solicitudes.

En estas aplicaciones sencillas no sería necesario usarlo, pero en la práctica, cuando se usan aplicaciones con conexiones a bases de datos, siempre se usa un gestor de conexiones.

En esta implementación vamos a usar la clase BasicDataSource del paquete org.apache.commons.dbcp2 que debemos incorporar a nuestro proyecto, la forma más sencilla es incorporándolo a nuestro fichero de dependencias Maven.

Implementación del interfaz

Seguimos con nuestra clase de implementación del Dao, y vemos el constructor. En él vamos a configurar la configuración del DataSource, muy similar a la configuración de un DriverManager, establecemos la dirección con el protocolo y driver, usuario y contraseña.

Habría muchos más parámetro que configurar, número de conexiones simultaneas, tiempos de espera, etc,

Para completarlo un poco, recogemos los datos de la conexión desde un fichero properties, fichero que se encuentra en el directorio resources que nos ha creado Maven, y vemos cómo accedemos a el de una forma especial, sin especificar realmente el directorio, porque está definido de una forma especial, y con el método getResource accedemos directamente.

```

public ClienteDaoMysql(){
    // Configurar el BasicDataSource con los datos de la base de datos
    Properties datos=new Properties();
    try (InputStream inputStream =
        Main.class.getClassLoader().getResourceAsStream("db_config.properties")) {
        datos.load(inputStream);
    } catch (IOException e) {
        e.printStackTrace();
    }
    BasicDataSource basicDataSource = new BasicDataSource();
    basicDataSource.setUrl("jdbc:mysql://" + datos.getProperty("db.servidor")
        + ":3306/" + datos.getProperty("db.baseDatos"));
    basicDataSource.setUsername(datos.getProperty("db.usuario"));
    basicDataSource.setPassword(datos.getProperty("db.clave"));
    this.dataSource = basicDataSource;
}

```

Seguimos con el resto de métodos, que son similares entre si, obtienen la conexión del gestor de conexiones, definen la sentencia necesaria, estableciendo parámetros usando el objeto que se le pasa al método y ejecutan la sentencia. Siempre prefiriendo sentencias pre compiladas.

```

@Override
public void guardar(Cliente cliente) {
    try (Connection conn=dataSource.getConnection();
        PreparedStatement pst=conn.prepareStatement("insert into clientes(nombre,direccion)
values (?,?)")) {

        pst.setString(1,cliente.getNombre());
        pst.setString(2,cliente.getDireccion());

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

@Override
public void actualizar(Cliente cliente) {
    try (Connection conn=dataSource.getConnection();
        PreparedStatement pst=conn.prepareStatement("update clientes set nombre=?,
direccion=? where id=?")) {

        pst.setString(1,cliente.getNombre());
        pst.setString(2,cliente.getDireccion());
        pst.setLong(3,cliente.getId());

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

```

```
@Override
public void borrar(Cliente cliente) {
    try (Connection conn=dataSource.getConnection();
        PreparedStatement pst=conn.prepareStatement("delete from clientes where id=?")) {

        pst.setLong(1,cliente.getId());

        pst.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
@Override
public Optional<Cliente> obtener(long id) {
    try (Connection conn = dataSource.getConnection();
        PreparedStatement stmt = conn.prepareStatement("SELECT nombre, direccion FROM
clientes WHERE id = ?")) {

        stmt.setLong(1, id);

        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                String nombre = rs.getString("nombre");
                String direccion = rs.getString("direccion");

                Cliente cliente = new Cliente(id, nombre, direccion);
                return Optional.of(cliente);
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return Optional.empty();
}
```

```
@Override
public List<Cliente> obtenerTodos() {
    List<Cliente> clientes = new ArrayList<>();
    try (Connection conn = dataSource.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT id, nombre, direccion FROM clientes")) {

        while (rs.next()) {
            long id = rs.getLong("id");
            String nombre = rs.getString("nombre");
            String direccion = rs.getString("direccion");

            Cliente cliente = new Cliente(id, nombre, direccion);
        }
    }
}
```



```

        clientes.add(cliente);
    }
} catch (SQLException e) {
    e.printStackTrace();
}
return clientes;
}

```

El interfaz

Ya tenemos definido el modelo (la clase Cliente), tenemos definido el controlador (la clase Dao), y ahora nos falta definir la vista (el interfaz), que haga uso del controlador para modificar objetos del modelo.

Vemos aquí sólo una parte, cómo definimos y creamos el Dao

```

private Dao<Cliente> dao;
public GestionClientesGUI() {

    dao=new ClienteDaoMysql();
    rellenarCombo();
}

```

Y por ejemplo cómo rellenamos el comboBox de Clientes.

```

/**
 * Rellena dos datos del comboBox listaClientes con los datos obtenidos del Dao
 */
private void rellenarCombo() {
    // Borramos primero los datos que contiene para poder llamarlo varias veces
    listaClientes.removeAllItems();
    // Si definiéramos otro Dao con otro tipo de acceso, esta línea no cambiaría
    List<Cliente> clientes=dao.obtenerTodos();
    // Añadimos objetos Cliente al componente
    clientes.forEach(listaClientes::addItem);
    // Esto hace que no esté seleccionado nada después de rellenarlo
    listaClientes.setSelectedIndex(-1);
}

```

En el aula virtual está disponible el código completo del proyecto.

10 Anexo. Otros temas interesantes

10.1 Patrón de diseño Singleton

Singleton es un patrón de diseño que define una clase de la cual solamente queremos tener una instancia. Un ejemplo de utilización podría ser un servicio del sistema operativo o la conexión a una base de datos (solo queremos una única conexión, sobre la que luego haremos las operaciones sobre la base de datos, pero no queremos una nueva instancia cada vez que hagamos una operación sobre la base de datos)

Para implementarla, podemos seguir los siguientes pasos:

- Definir un único constructor como privado, así no se podrán crear instancias desde el exterior.
- Obtener siempre la instancia a través de un método estático, que llama al constructor solo la primera vez. Las veces siguientes que invoquemos a ese método devolverá esa primera instancia creada.

⚡ Un constructor devuelve la referencia en memoria de la instancia que crea. Este método estático “simula” el comportamiento de un constructor.

Ejemplo:

```
class ClaseSingleton {
    private static ClaseSingleton instance = null;
    private ClaseSingleton() {}

    //Evita la instanciación directa
    public static ClaseSingleton getInstance() {
        if (instance == null)
            instance = new ClaseSingleton();
        return instance;
    }
}
```

Podemos usarlo y comprobar su funcionamiento así:

```
// ClaseSingleton cls0 = new ClaseSingleton(); //Error
ClaseSingleton cls1 = ClaseSingleton.getInstance();
ClaseSingleton cls2 = ClaseSingleton.getInstance();
```

```
// Comprobamos así, viendo que la referencia es la misma
System.out.println("Instancia Singleton 1: " + cls1);
System.out.println("Instancia Singleton 2: " + cls2);
} catch (SQLException e) { e.printStackTrace(); }
```

10.2 SQLite

SQLite es una librería gratuita que implementa un motor de base de datos sencillo, autocontenido, sin necesidad de servidor alguno y ni de configuración.

Toda la información de una base de datos (tablas, índices, triggers, etc.) queda almacenada en un solo archivo, que es fácilmente portable entre distintas plataformas, simplemente copiando un único archivo. Como inconvenientes, cabe citar que no tiene gestión de usuarios ni privilegios y que tiene pocos tipos de datos (dispone de tipos Integer, Real, Text y Blob pero no Boolean o Date/Time).

Todas estas características la hacen idónea en muchas situaciones: en pequeños proyectos o proyectos con poco tratamiento de base de datos, en fase de prueba, cuando vamos a presentar un prototipo previo a nuestros clientes, aplicaciones móviles, etc.

Necesitaremos usar el driver JDBC de SQLite.

10.3 Mapeo Objeto-Relacional. Hibernate

El desfase objeto-relacional surge cuando en el desarrollo de una aplicación con un lenguaje orientado a objetos se hace uso de una base de datos relacional. En cuanto al desfase, ocurre que en nuestra aplicación Java tendremos la definición de clases con sus atributos y métodos mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y viceversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente.

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos, como hemos visto previamente.

Si contamos con un framework como Hibernate, esta misma operación se traduce en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican a éste con que tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

```
@Entity
@Table(name = "actor", catalog = "db_peliculas")
public class Actor {
    private Integer id;
    private String nombre;
    private Date fechaNacimiento;
    // Constructor/es
    public Actor() {...}
    ...
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "id")
    public Integer getId() {
        return this.id;
    }
    public void setId(Integer id) {
        this.id = id;
        @Column(name = "nombre")
        public String getNombre() {
            return this.nombre;
        }
    }
    ...
}
```

Luego, las operaciones de base de datos, las haremos a través de métodos hibernate sin ver el SQL que hay por detrás. Ejemplo:

```
Session session = HibernateUtil.getCurrentSession();
session.beginTransaction();
session.save(unObjeto);
session.getTransaction().commit();
session.close();
```

Más información en: <https://datos.codeandcoke.com/apuntes:hibernate>

10.4 Transacciones

Una transacción es un conjunto de operaciones sobre una base de datos que se deben ejecutar como una unidad. Hay ocasiones en las que es necesario que varias operaciones sobre la base de datos se realicen en bloque, es decir, que se ejecuten o todas o ninguna, pero no que se realicen unas sí y otras no. Si se ejecutan parcialmente hasta que una da error, el estado de la base de datos puede quedar inconsistente. En este caso necesitaríamos un mecanismo para devolverla a su estado anterior, pudiendo deshacer todas las operaciones realizadas.

El objeto `Connection` por defecto realiza automáticamente cada operación sobre la base de datos. Esto significa que cada vez que se ejecuta una instrucción, se refleja en la base de datos y no puede ser deshecha. Por defecto está habilitado el modo auto-commit en la conexión.

Los siguientes métodos en la interfaz `Connection` son utilizados para gestionar las transacciones en la base de datos:

```
void setAutoCommit(boolean valor)
void commit()
void rollback()
```

Para iniciar una transacción deshabilitamos el modo auto-commit mediante el método `setAutoCommit(false)`. Esto nos da el control sobre lo que se realiza y cuándo se realiza. Una llamada al método `commit()` realizará todas las instrucciones emitidas desde la última vez que se invocó el método `commit()`.

Una llamada a `rollback()` deshacerá todos los cambios realizados desde el último `commit()`. Una vez se ha emitido una instrucción `commit()`, esas transacciones no pueden deshacerse con `rollback()`. En muchas ocasiones, deberemos hacer `rollback()` de operaciones en la sección `catch` de los métodos, si se produce una excepción en una operación previa.

10.5 RowSet

`ResultSet` no funciona para hacer actualizaciones con todos los gestores de base de datos. `RowSet` es una evolución de `ResultSet`, que soluciona este problema y ofrece otras ventajas, como las siguientes:

- `ResultSet` mantiene la conexión con la base de datos permanentemente, y `RowSet` puede ser conectado o no.
- `RowSet` es `Serializable`, puede ser transmitido por la red y puede ser tratado como un `JavaBean`, `ResultSet` no tiene ninguna de estas características.