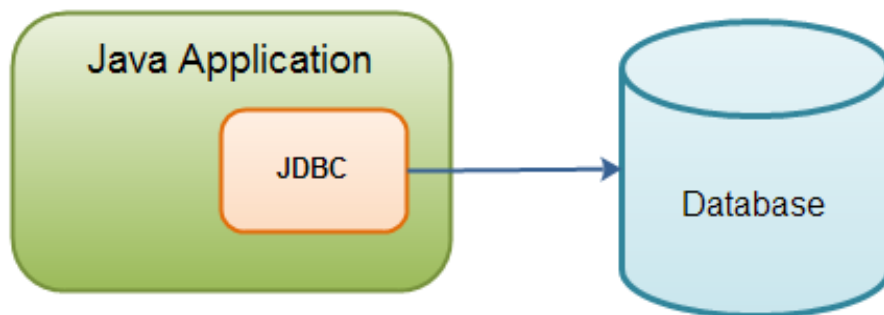


# U.T. 2: ACCESO A BASES DE DATOS RELACIONALES

1. INTRODUCCIÓN A JDBC.
2. LA INTERFAZ STATEMENT.
3. TRABAJANDO CON ResultSets.
4. USANDO ResultSet ACTUALIZABLES.
5. TRANSACCIONES.
6. OBJETOS PreparedStatement.
7. DBUTILS.
8. POOL DE CONEXIONES.
9. PROCEDIMIENTOS Y FUNCIONES ALMACENADAS.
  - 9.1. Procedimientos almacenados.
  - 9.2. Funciones almacenadas.
10. PATRÓN DAO.
11. PATRÓN MODELO-VISTA-CONTROLADOR.
  - 11.1. Ejemplo de aplicación MVC.



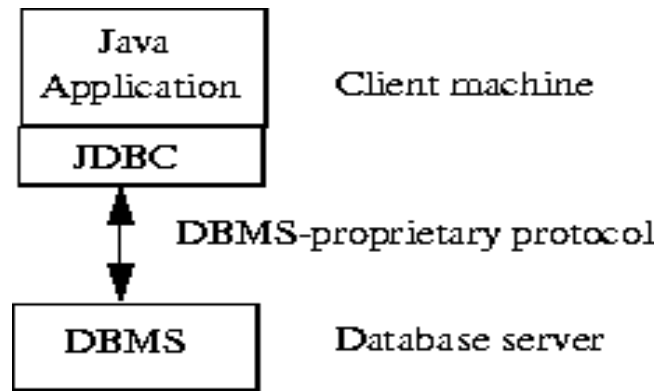
## 1. INTRODUCCIÓN A JDBC

JDBC (Java Data Base Connectivity) es la interfaz común que Java proporciona para poder conectarnos a cualquier SGBD Relacional. Proporciona una API completa para trabajar con Bases de Datos Relacionales de forma que, sea cual sea el motor (MySQL, PostgreSQL, Oracle ...) con el que conectemos, la API siempre será la misma. Simplemente tendremos que proveernos del controlador o *Driver* correspondiente para el gestor utilizado, que sí que dependerá totalmente de éste. Actualmente podemos encontrar un controlador JDBC para prácticamente cualquier SGBDR existente.

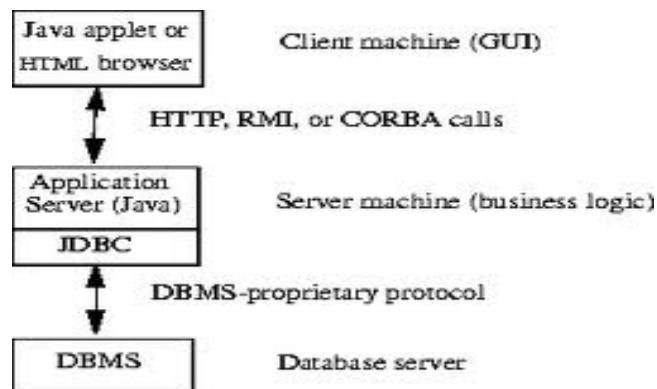
Ya que el controlador es lo único que depende exclusivamente del SGBD utilizado, es muy sencillo escribir aplicaciones cuyo código se pueda reutilizar si más adelante hemos de cambiar de motor de Base de Datos, o bien si queremos permitir que dicha aplicación pueda conectarse a más de un SGBD de forma que no tengamos que comprometernos a usar siempre un SGBD concreto.

2 modelos de arquitectura JDBC:

- en 2 capas:



- en 3 capas: el cliente interactúa desde un navegador y la aplicación se encuentra en un servidor intermedio, al otro extremo está el servidor de base de datos.



JDBC incluye un conjunto de clases e interfaces, que podemos agrupar atendiendo a las funciones que cumplen. Las principales pueden ser las siguientes:

- establecer una conexión con una base de datos
- ejecutar sentencias SQL
- procesar los resultados de la ejecución de las sentencias

Primer paso: establecer conexión -> se hará con la clase **DriverManager** y su método *getConnection...*, ¿pero para que se utiliza esta clase? Su función primordial es la de seleccionar el controlador adecuado para conectar la aplicación con la base de datos concreta. Para establecer una conexión en un origen de datos:

- la ya mencionada DriverManager
- Driver
- Connection
- Para ejecución de sentencias SQL:
  - *Statement*
  - *PreparedStatement*
  - *CallableStatement*
- Para obtención y modificación de los resultados de una sentencia SQL:
  - *ResultSet*

JDBC ofrece simplemente una especificación común para el acceso a bases de datos; el fabricante de un driver para un SGBD concreto tendrá que seguir las recomendaciones de esta especificación implementando la interfaz *Driver*.

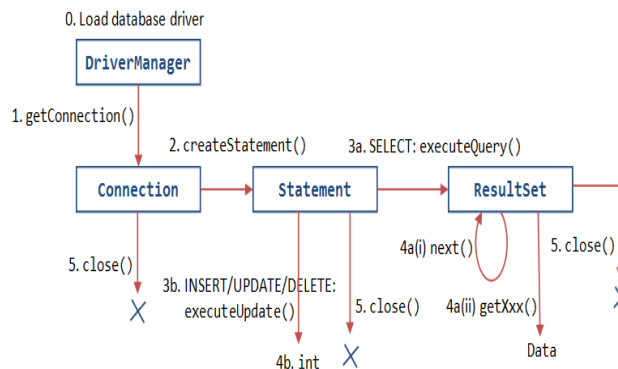
La clase *DriverManager* mantiene una lista de clases *Driver*. Todas ellas han de estar escritas con una sección estática que creará una instancia de la clase y, a continuación, se registrará con el método estático *DriverManager.registerDriver()*; este proceso se ejecuta automáticamente cuando se carga la clase del driver.

Por lo tanto, gracias al mecanismo anterior, un programador normalmente no tendría que llamar directamente al método *DriverManager.registerDriver()*, sino que será llamado automáticamente por el driver cuando es cargado. La manera aconsejada de cargar una clase *Driver*, y por tanto registrar con el *DriverManager*, es llamar al método estático *forName()* de la clase *Class*:

```
//String driver = "com.mysql.cj.jdbc.Driver";
String driver = "org.postgresql.Driver";
Class.forName(driver).newInstance();
```

Pero, con las últimas versiones de los drivers, incluso la carga del controlador se hace automáticamente. Por lo tanto, nuestros programas no requerirán las 2 líneas anteriores.

Una vez que las clases de los controladores se han cargado y registrado, se puede establecer la conexión con la base de datos. Un objeto **Connection** representa una conexión con una base de datos. La sesión de conexión incluirá la ejecución de sentencias SQL con sus consiguientes resultados.



La misma aplicación puede abrir más de una conexión con la misma base de datos, o con diferentes. Pero *Connection* no es una clase, sino una interfaz. ¿Como es que se instancia? La interfaz es una plantilla o especificación implementada en la práctica por el controlador concreto, de MySQL, PostgreSQL ... según corresponda. Por ejemplo:

```
String jdbcUrl = "jdbc:postgresql://localhost:5432/empresa";
Connection con = DriverManager.getConnection(jdbcUrl, "root", "");
```

## Conectando con una Base de Datos de MySQL

```
. . .
Connection con = null;

try {
    Class.forName("com.mysql.cj.jdbc.Driver");
    con = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/basededatos",
        "usuario", "contraseña");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
} catch (SQLException sqle) {
    sqle.printStackTrace();
}
. . .
```

## Conectando con una Base de Datos de PostgreSQL

```
. . .
Connection con = null;

try {
    Class.forName("org.postgresql.Driver").newInstance();
    con = DriverManager.getConnection(
        "jdbc:postgresql://localhost:5432/basededatos",
        "usuario", "contraseña");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
} catch (SQLException sqle) {
    sqle.printStackTrace();
}
. . .
```

## Desconectar de la Base de Datos

A la hora de desconectar, basta con cerrar la conexión con el método *close()* de la clase *Connection*, que será la misma operación independientemente del controlador utilizado. Podemos evitar haber de cerrar la conexión manualmente si hacemos uso de *try-with-resources*.

```
. . .
try (Connection con = DriverManager.getConnection("jdbc:postgresql://localhost:5432/basededatos") {
    // trabajamos con el objeto Connection y, al final, se cerrará automáticamente
} catch (SQLException sqle) {
    sqle.printStackTrace();
}
. . .
```

## 2. LA INTERFAZ STATEMENT

En el apartado anterior veíamos cómo establecer una conexión con una base de datos. Ahora vamos a dar un paso más en el acceso a bases de datos. Una vez establecida la conexión, podremos enviar sentencias SQL; para eso contamos con la interfaz *Statement*.

Tenemos tres tipos de objetos *Statement*. Son:

- *Statement*
- *PreparedStatement*
- *CallableStatement*.

*PreparedStatement* hereda de la interfaz *Statement*, mientras que *CallableStatement* lo hace de *PreparedStatement*.

*Statement* es utilizado para ejecutar una sentencia SQL simple sin parámetros; un objeto *PreparedStatement* es utilizado para ejecutar sentencias SQL precompilada con o sin parámetros de entrada, y se suele utilizar también para ejecutar sentencias SQL de uso frecuente. Finalmente, un objeto *CallableStatement* es utilizado para ejecutar una llamada a un procedimiento almacenado de una base de datos, que puede tener parámetros de entrada, de salida y entrada/salida.

Un *Statement* se crea con el método *createStatement()* de la clase *Connection*, como se puede observar en el siguiente fragmento de código:

```
conexion = DriverManager.getConnection(jdbcUrl, "root", "");
stmt = conexion.createStatement();
```

Observa cómo, de momento, el método `createStatement()` se está llamando sin parámetros, hecho que supondrá algunas limitaciones que veremos más adelante.

El objeto `Statement` proporciona básicamente 3 métodos de ejecución de sentencias: `execute()`, `executeUpdate()` y `executeQuery()`. La diferencia entre cada uno de estos métodos la mostramos a continuación:

- `executeQuery()`: se utiliza con sentencias SELECT (consultas o *queries*) y devuelve un `ResultSet`.

- `executeUpdate()`: se utiliza con sentencias INSERT, UPDATE y DELETE, o bien, con sentencias DDL SQL, como la misma creación de una nueva tabla.

- `execute()`: utilizado para cualquier sentencia DDL (de definición de datos como CREATE, para modificar o crear la estructura de las mesas), DML (de manipulación de datos como INSERT, UPDATE, DELETE o SELECT), o también órdenes específicas de la base de datos.

## Insertar datos

```
. . .
    String sql = "INSERT INTO productos VALUES (1,'azucar',3.15)";

    try (Statement stmt = con.createStatement()) {
        stmt.executeUpdate(sql);
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
. . .
```

Las operaciones de inserción, modificación o eliminación de datos las podemos hacer, como en el ejemplo anterior, con una sentencia precompilada (haciendo uso de `PreparedStatement`) o con sentencias completamente definidas (haciendo uso de `Statement`).

## Modificar datos

```
. . .
    String sql = "UPDATE productos SET precio = 3.55 WHERE id = 1";

    try (Statement stmt = con.createStatement()) {
        stmt.executeUpdate(sql);
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
. . .
```

## Eliminar datos

```
. . .
    String sql = "DELETE FROM productos WHERE id = 1";

    try (Statement stmt = con.createStatement()) {
        stmt.executeUpdate(sql);
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
. . .
```

## Consultas

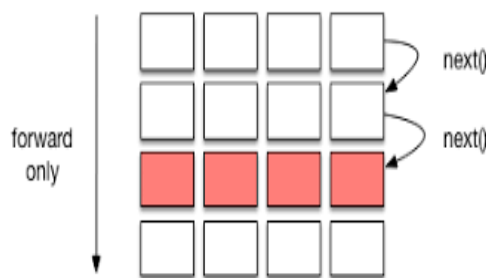
Veamos un ejemplo de uso del método `executeQuery()`. Este método devuelve un *ResultSet* que guardará los resultados obtenidos para su posterior uso:

```
String jdbcUrl = "jdbc:postgresql://localhost:5432/empresa";
Connection con = DriverManager.getConnection(jdbcUrl, "root", "");
/* Uso del método executeQuery
Observa como createStatement NO tiene parámetros: esto limitará el tipo de ResultSet */
Statement stmt = con.createStatement();
String sql = "select * from artículos";
ResultSet rs = stmt.executeQuery(sql);
```

EL método `createStatement()` de la interfaz *Connection* se encuentra sobrecargado. Si utilizamos su versión sin parámetros, en la hora de ejecutar sentencias SQL sobre el objeto *Statement* se obtendrá el tipo de objeto *ResultSet* por defecto, es decir, se obtendría un tipo de cursor de solo lectura y con movimiento únicamente hacia delante. Pero la otra versión del método `createStatement()` ofrece dos parámetros que nos permiten definir el tipo de *ResultSet* que se devolverá como resultado de la ejecución de la consulta, como se muestra en el siguiente código:

```
String jdbcUrl = "jdbc:mysql://localhost:5432/empresa";
Connection con = DriverManager.getConnection(jdbcUrl, "root", "");
// Uso del método executeQuery
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
String sql = "select * from artículos";
rs = stmt.executeQuery(sql);
```

El primero de los parámetros indica el tipo de objeto *ResultSet* que se creará, y el segundo de ellos indica si el *ResultSet* es solo de lectura o si permite modificaciones; este parámetro también se denomina tipo de concurrencia. Si no indicamos ningún parámetro en el método `createStatement()`, se creará un objeto *ResultSet* con los valores por defecto.

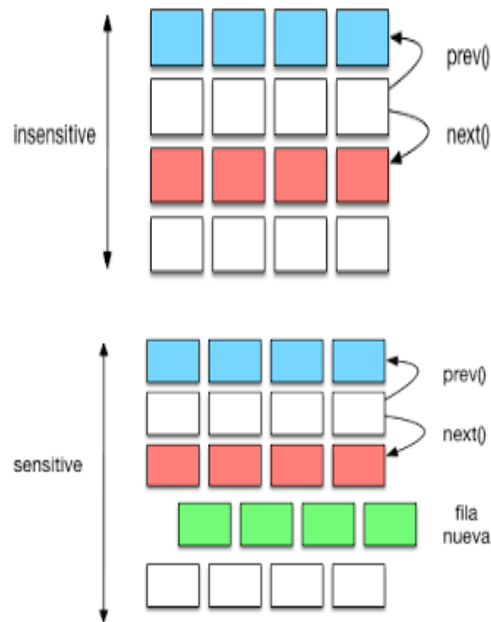


Los tipos de *ResultSet* diferentes que se pueden crear dependen de los valores elegidos para los parámetros. Estos valores se corresponden con constantes definidas en la interfaz *ResultSet*. Estas constantes, para el primer parámetro, se describen a continuación:

**TYPE\_FORWARD\_ONLY:** se crea un objeto *ResultSet* con movimiento únicamente hacia delante (forward-only). Es el tipo de *ResultSet* por defecto.

**TYPE\_SCROLL\_INSENSITIVE:** se crea un objeto *ResultSet* que permite todo tipo de movimientos. Pero este tipo de *ResultSet*, mientras está abierto, no será consciente de los cambios que se realizan sobre los datos que está mostrando, y por tanto no mostrará estas modificaciones.

**TYPE\_SCROLL\_SENSITIVE:** igual que el anterior permite todo tipo de movimientos, pero además permite ver los cambios que se realizan sobre los datos que contiene. Para que esos cambios puedan reflejarse en el *ResultSet* habremos de hacer uso del método `refreshRow()`.



Los valores que puede tener el segundo parámetro que define la creación de un objeto *ResultSet*, son también constantes definidas en la interfaz *ResultSet* y son los siguientes:

**CONCUR\_READ\_ONLY**: indica que el *ResultSet* es solo de lectura. Es el valor por defecto.

**CONCUR\_UPDATABLE**: permite realizar modificaciones sobre los datos que contiene el *ResultSet*.

### 3. TRABAJANDO CON ResultSets

En un objeto *ResultSet* se encuentran los resultados de la ejecución de una sentencia SQL. Es decir, contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece el acceso a los datos a través de una serie de métodos *getXXX()* para acceder a las columnas de la fila actual.

El método *next()* de la interfaz *ResultSet* es utilizado para desplazarse a la siguiente fila del *ResultSet*, haciendo que la próxima fila sea la actual; además de este desplazamiento básico, según el tipo de *ResultSet*, podremos realizar desplazamientos libres utilizando métodos como *last()*, *relative()* o *previous()*.

El aspecto que suele tener un *ResultSet* es una tabla con cabeceras de columnas y los valores correspondientes devueltos por una consulta. Un *ResultSet* mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve hacia abajo cada vez que el método *next()* es ejecutado. Inicialmente está posicionado antes de la primera fila; de este modo, el primer llamamiento a *next()* situará el cursor a la primera fila, pasando a ser la fila actual.

Las filas del *ResultSet* son devueltas de arriba a abajo según se va desplazando el cursor con los sucesivos usos de *next()*. Un cursor es válido hasta que el objeto *ResultSet*, o su *Statement*, sea cerrado.

Obteniendo datos del *ResultSet*: los métodos *getXXX()* ofrecen una forma de recuperar los valores de las columnas (campos) de la fila (registro) actual del *ResultSet*. No hace falta que las columnas sean recuperadas utilizando un orden determinado, pero, para una mayor portabilidad entre diferentes bases de datos, se recomienda que los valores de las columnas se recuperen de izquierda a derecha y solo una vez.



Para referirnos a una columna podemos utilizar su nombre, o bien, su número de orden. Por ejemplo, si la segunda columna de un objeto *rs* de la clase *ResultSet* se llama "titulo" y almacena datos de tipos String, se podrá recuperar su valor de las formas que muestra el siguiente fragmento de código:

```
// rs es un objeto de tipo ResultSet
String valor = rs.getString(2); // primera forma
String valor = rs.getString("titulo"); // segunda, completamente equivalente
```

Se ha de señalar que las columnas se numeran de izquierda a derecha empezando con la columna 1, y que los nombres de las columnas no son *case sensitive*, es decir, no distinguen entre mayúsculas y minúsculas.

La información referente a las columnas que contiene el *ResultSet* se encuentra disponible llamando al método *getMetaData()*; este método 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()* admite como parámetro el nombre de la columna y retorna un entero que será el índice correspondiente.

**Tipos de datos y conversiones:** cuando se lanza un método *getXXX()* sobre un *ResultSet* para obtener el valor de un campo del registro actual, el controlador JDBC convierte el dato que se quiere recuperar al tipo Java especificado. Por ejemplo, si utilizamos el método *getString()* y el tipo en la base de datos es VARCHAR, el controlador JDBC convertirá el dato VARCHAR a un objeto String de Java, por lo tanto el valor de retorno de *getString()* será un objeto de la clase String.

Esta conversión de tipo se puede realizar gracias a la clase *java.sql.Types*. En esta clase se definen lo que se denominan tipo de datos JDBC, que se corresponde con los tipos de datos SQL estándar. Esto nos permite abstraernos del tipo SQL específico de la base de datos con la cual estamos trabajando, puesto que los tipos JDBC son tipos de datos SQL genéricos.

La clase *Types* está definida como un conjunto de constantes (*final static*); estas constantes se usan para identificar los tipos SQL. Si el tipo del dato SQL que tenemos es específico de esta base de datos, y no se encuentra entre las constantes definidas por *Types*, se utilizará el tipo *Types.OTHER*.

**Desplazamiento en un ResultSet:** El método *next()* devuelve un valor booleano (tipo boolean de Java), true si el registro siguiente existe y false si hemos llegado al final y no hay más registros. Pero además disponemos de otros métodos para el desplazamiento y movimiento dentro de un objeto *ResultSet*.

-*boolean absolute(int registro)*: desplaza el cursor el número de registros indicado. Si el valor es negativo, se posiciona en el número de registro indicado pero empezando por el final. Este método devolverá false si nos hemos desplazado después del último registro o antes del primer registro del objeto *ResultSet*. Para poder utilizar este método, el objeto *ResultSet* tiene que ser de tipo *TYPE\_SCROLL\_SENSITIVE* o de tipo *TYPE\_SCROLL\_INSENSITIVE*. Un *ResultSet* de cualquier de estos dos tipos lo llamamos "Scrollable". Si a este método le pasamos un valor cero se lanzará una *SQLException*.

-*void afterLast()*: se desplaza al final del *ResultSet*, después del último registro.

-*void beforeFirst()*: mueve el cursor al inicio del objeto *ResultSet*, antes del primer registro. Solo se puede utilizar sobre *ResultSet* de tipos *Scrollable*.

-*boolean first()*: desplaza el cursor en la primera entrada. Devuelve *true* si el cursor se ha desplazado a un registro válido; por el contrario, tiene que devolver *false* en otro caso o si el *ResultSet* no contiene registros. Como los métodos anteriores, solo se puede utilizar en *ResultSet* de tipo "Scrollable".

-*boolean last()*: desplaza el cursor al último registro del objeto *ResultSet*. Devolverá *true* si el cursor se encuentra en un registro válido, y *false* en otro caso o si el *ResultSet* no tiene registros.

-*void moveToCurrentRow()*: mueve el cursor a la posición recordada, normalmente el registro actual. Este método solo tiene sentido cuando estamos situados dentro del *ResultSet* en un registro que se ha insertado, es decir, con *ResultSet* que permiten la modificación, definidos mediante la constante *CONCUR\_UPDATABLE*.

-*boolean previous()*: desplaza el cursor al registro anterior. Es el método contrario al método *next()*. Devolverá *true* si el cursor se encuentra en un registro o fila válidos, y *false* en caso contrario. Solo es válido este método con *ResultSet* de tipo *Scrollable*; en caso contrario lanzará una excepción *SQLException*.

-*boolean relative(int registros)*: mueve el cursor un número relativo de registros; este número puede ser positivo o negativo. Si el número es negativo el cursor se desplazará hacia el principio del objeto *ResultSet* el número de registros indicados, y si es positivo se desplazará hacia el final. Igualmente solo se puede utilizar si el *ResultSet* es *Scrollable*.

Otros métodos de la interfaz *ResultSet* relacionados con el desplazamiento:

- *boolean isAfterLast()*: indica si nos encontramos después del último registro del *ResultSet*. Solo se puede utilizar en *ResultSet* de tipos *Scrollable*.
- *boolean isBeforeFirst()*: indica si nos encontramos antes del primer registro del *ResultSet*. Solo se puede utilizar en *ResultSet* de tipos *Scrollable*.
- *boolean isFirst()*: indica si el cursor se encuentra en el primer registro. Solo se puede utilizar en *ResultSet* de tipos *Scrollable*.
- *boolean isLast()*: indica si nos encontramos en el último registro del *ResultSet*. Solo se puede utilizar en *ResultSet* de tipos *Scrollable*.
- *int getRow()*: devuelve el número de registro actual. El primer registro será el número 1, el segundo el 2, etc. Devolverá cero si no hay registro actual.

Si queremos recorrer un *ResultSet* completo hacia delante lo podremos hacer a través de un bucle en que se lanza el método *next()* sobre el objeto *ResultSet*. La ejecución de este bucle finalizará cuando llegamos al final del conjunto de registros.

```
. . .
    String sql = "SELECT nombre, precio FROM productos";

    try (Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {
        while (rs.next()) {
            System.out.println("nombre: " + rs.getString(1));
            System.out.println("precio: " + rs.getFloat(2));
        }
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
. . .
```

En el caso de las funciones agregadas, como *count* o *sum*, podremos tener en cuenta que sólo van a devolver un valor, por lo que no será necesario preparar el código para recorrer el cursor. Podremos acceder directamente a la primera fila del *ResultSet* y mostrar el resultado, tal y como se muestra en el siguiente ejemplo:

```

. . .
String sql = "SELECT COUNT(*) FROM productos";

try (Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(sql)) {
    rs.next();
    System.out.println("Cantidad de productos: " + rs.getInt(1));
} catch (SQLException sqle) {
    sqle.printStackTrace();
}
. . .

```

## 4. USANDO ResultSet ACTUALIZABLES

**Modificando los ResultSet.** Métodos *updateXXX()*: para poder modificar los datos que contiene un *ResultSet* hemos de crear un *ResultSet* modificable, haciendo uso de la constante *ResultSet.CONCUR\_UPDATABLE* al llamar a *createStatement()*.

Aunque un *ResultSet* que permita modificaciones suele permitir diferentes desplazamientos, es decir, se suele utilizar la constante *ResultSet.TYPE\_SCROLL\_INSENSITIVE* o *ResultSet.TYPE\_SCROLL\_SENSITIVE*, no es imprescindible: también puede ser del tipo solo hacia delante (forward-only).

Para modificar los valores de un registro existente se utilizan una serie de métodos *updateXXX()* de la interfaz *ResultSet*. Las XXX indican el tipo del dato al igual que pasa con los métodos *getXXX()* de esta misma interfaz. El proceso para realizar la modificación de una fila de un *ResultSet* es el siguiente: nos situamos sobre el registro que queremos modificar y llamamos a los métodos *updateXXX()* adecuados, pasándole como argumento los nuevos valores. Finalmente el método *updateRow()* hará que los cambios tengan efecto en la base a datos.

El método *updateXXX()* recibe dos parámetros: el campo o columna a modificar y el nuevo valor. La columna la podemos indicar por su número de orden o bien por su nombre, igual que en los métodos *getXXX()*.

El siguiente fragmento de código muestra cómo se puede modificar el campo "direccion" del último registro de un *ResultSet* con el resultado de una SELECT sobre la tabla de clientes:

```

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
// Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
System.out.println( "Situamos el cursor al final");
// Nos situamos en el último registro del ResultSet y modificamos
rs.last();
rs.updateString("dirección","c/ Pepe Ciges, 3");
rs.updateRow();

```

Si nos desplazamos dentro del *ResultSet* antes de lanzar el método *updateRow()*, se perderán las modificaciones realizadas. Y si queremos cancelar las modificaciones lanzaremos el método *cancelRowUpdates()* sobre el objeto *ResultSet*, en lugar del método *updateRow()*. Una vez se haya ejecutado *updateRow()*, *cancelRowUpdates()* no tendría ningún efecto.

Métodos *insertRow()* y *deleteRow()*: además de poder realizar modificaciones directamente sobre las filas de un *ResultSet*, también podemos añadir nuevas filas (registros) y eliminar las existentes. Estos métodos son: *moveToInsertRow()* y *deleteRow()*. Para insertar una nueva fila el proceso es:

1. El primer paso para insertar un registro o fila en un *ResultSet* es mover el cursor (puntero que indica el registro actual) del *ResultSet* llamando al método *moveToInsertRow()*.
2. El siguiente paso es dar un valor a cada uno de los campos que formarán parte del nuevo registro, para lo cual se utilizan los métodos *updateXXX()* adecuados.

3. Finalizamos el proceso llamando a *insertRow()*, que creará el nuevo registro tanto en el *ResultSet* como la tabla de la base de datos correspondientes. Hasta que no se ejecuta el método *insertRow()*, la fila no se incluye dentro del *ResultSet*, es una fila especial denominada "fila de inserción" (insert row) y es similar a un buffer completamente independiente del objeto *ResultSet*.
4. Cuando ya esté insertado nuestro nuevo registro en el *ResultSet*, podremos volver a la antigua posición en que nos encontrábamos, antes de haber lanzado el método *moveToInsertRow()*, mediante el método *moveToCurrentRow()*. Este método sólo se puede utilizar en combinación con el método *moveToInsertRow()*.

El siguiente fragmento de código da de alta un nuevo registro a la tabla de clientes:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
// Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
// Creamos un nuevo registro a la tabla de clientes
rs.moveToInsertRow();
rs.updateString(2, "Killy Lopez");
rs.updateString(3, "Wall Street 3674");
rs.insertRow();
```

Si no facilitamos valores en todos los campos del nuevo registro con los métodos *updateXXX()*, este campo tendrá un valor NULL, y si la base de datos no admite nulos se producirá una excepción *SQLException*.

Además de insertar filas en nuestro objeto *ResultSet*, también podremos eliminar filas o registros. El método a utilizar para esta tarea es el método *deleteRow()*. Para eliminar un registro no tenemos que hacer nada más que movernos a este registro, y ejecutar *deleteRow()* sobre el objeto *ResultSet* correspondiente.

El siguiente fragmento de código borra el último registro de la tabla de clientes:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
// Ejecutamos la SELECT sobre la tabla clientes
String sql = "select * from clientes";
rs = stmt.executeQuery(sql);
// Nos situamos a finales del ResultSet
rs.last();
rs.deleteRow();
```

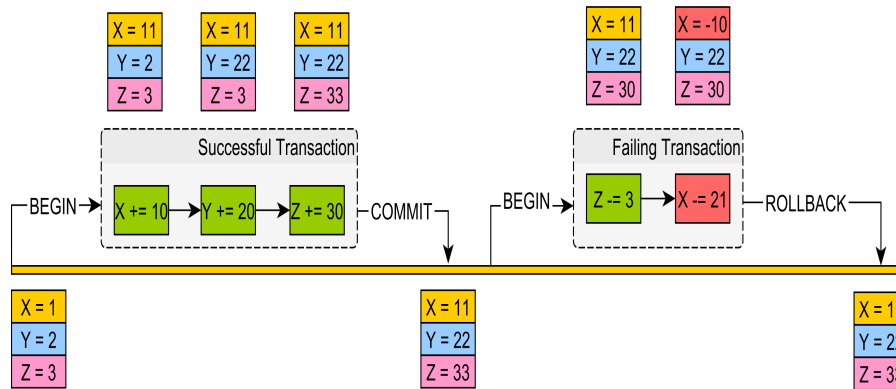
## 5. TRANSACCIONES

En algunos momentos podemos necesitar que varias sentencias SQL se ejecutan como un todo, y, si falla alguna de ellas, se deshagan los cambios realizados por las sentencias anteriores y se vuelva a la situación inicial. Así por ejemplo, al realizar una transferencia de una cuenta de un banco a otra cuenta, las dos sentencias encargadas de realizar la actualización en el saldo de cada cuenta se tienen que ejecutar, pero si una de ellas falla se tienen que deshacer los cambios realizados. Estas dos sentencias se han de ejecutar en una misma transacción.

Esas sentencias que se ejecutan en una misma transacción se han de ejecutar TODAS con éxito; si falla alguna se deshacen los cambios para evitar cualquier tipo de inconsistencia en la base de datos. Decimos que en una transacción se da un "todo o nada".

Hasta ahora, cuando hemos establecido una conexión en una base de datos con JDBC, por defecto ésta se ha creado en modo "autocommit" o "commit automático". Esto quiere decir que cada sentencia SQL modifica la base de datos nada más ejecutarse, es decir, se trata como una transacción que contiene una única sentencia. Ahora si queremos agrupar varias sentencias SQL en una misma transacción usaremos el método *setAutoCommit()* de la interfaz *Connection*, pasándole como parámetro el valor *false*, para indicar que cada sentencia no se ejecute automáticamente.

Una vez que se ha deshabilitado la manera *autocommit*, las sentencias que ejecutamos no modificarán la base de datos hasta que no ejecutemos el método *commit()* de la interfaz *Connection*. Todas las sentencias ejecutadas previamente a la llamada a *commit()* se incluirán en la misma transacción.



En el caso de una transferencia entre dos cuentas, y si suponemos que el número de la cuenta de origen de la transferencia es el 1 y el de destino el 2, el siguiente código efectuaría estas dos modificaciones dentro de una transacción:

```
try {
    con.setAutoCommit(false);
    int trans = 5000;
    int compteOrigen = 1;
    int compteDesti = 2;
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("UPDATE Cuentas SET cantidad = cantidad - "
        + trans + "WHERE NumCompte = " + compteOrigen);
    stmt.executeUpdate("UPDATE Cuentas SET cantidad = cantidad + "
        + trans + "WHERE NumCompte = " + compteDesti);
    con.commit();
}
catch (SQLException ex) {
    con.rollback();
    System.err.println("La transacción ha fallado.");
}
finally
{
    con.setAutoCommit(true);
}
```

## Procesamiento por lotes (Batch Processing)

JDBC Batch Processing (**procesamiento por lotes**): la interfaz *Statement* también soporta el procesamiento de múltiples sentencias en modo *batch*. De este modo minimizamos el número de accesos a la base de datos, factor crítico en algunas aplicaciones.

Por ejemplo, supongamos una aplicación que necesite realizar muchas sentencias de tipos INSERT y UPDATE. Si no utilizamos procesamiento *batch* podemos sobrecargar en exceso con accesos a la base de datos y disminuir el rendimiento general de la aplicación. Por eso, si utilizamos el procesamiento *batch* podemos agrupar varias sentencias como INSERT y UPDATE y lanzarlas hacia la base de datos en una única actualización. Para hacer esto es necesario saber si nuestra base de datos soporta procesamiento *batch*; podemos consultarlo con la interfaz *DatabaseMetaData* y llamar al método *supportBatchUpdates()*. Si devuelve true, quiere decir que nuestra base de datos sí que lo soporta.

```
Connection conn = DriverManager.getConnection(jdbcUrl, "root", "");
DatabaseMetaData dm = conn.getMetaData();
System.out.println("Soporta procesamiento batch ->" + dm.supportsBatchUpdates());
```

El procedimiento para utilizar el procesamiento por lotes consiste en llamar al método *addBatch()* que recibe como parámetro la sentencia SQL, y este proceso lo repetimos para cada una de las sentencias SQL que

queramos añadir en el lote. Una vez tengamos añadidas todas las sentencias llamamos al método `executeBatch()` de la interfaz `Statement`. Previamente hemos de haber desactivado el `autocommit` con el método `setAutoCommit()`, y finalmente invocar el método `commit()` para realizar los cambios.

En el siguiente ejemplo se crean 3 artículos nuevos en modo batch:

```
Connection con = DriverManager.getConnection(jdbcUrl, "root", "");
con.setAutoCommit(false);
Statement stmt = conn.createStatement();
// Añadimos sentencias SQL a manera Batch
String sql = "insert into artículos values(8, 'HD 120G', 100.0, 'HD120', 2)";
stmt.addBatch(sql);
sql = "insert into artículos values(9, 'HD 160G', 120.0, 'HD160', 2)";
stmt.addBatch(sql);
sql = "insert into artículos values(10, 'HD 200G', 140.0, 'HD200', 2)";
stmt.addBatch(sql);
int result [] = stmt.executeBatch();
conn.commit();
conn.setAutoCommit(true);
```

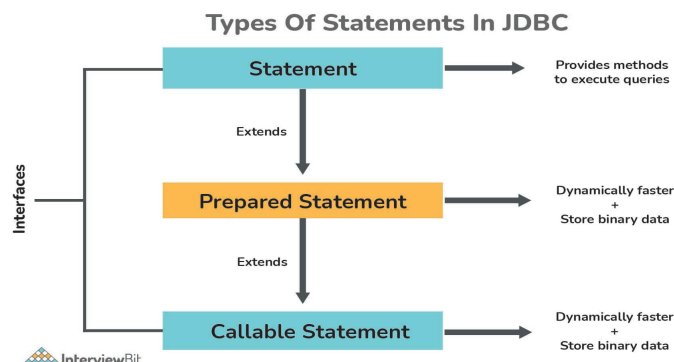
Observa como el método `executeBatch()` devuelve un array de enteros, indicativo de las filas afectadas por cada instrucción SQL.

Cuando trabajamos con JDBC podemos encontrar 4 tipos de excepciones: `SQLException`, `SQLWarning`, `BatchUpdateException` y `DataTruncation`. `SQLException` hereda de `Exception`, y por tanto tiene toda la funcionalidad para gestionar excepciones Java. Casi todos los métodos de la API JDBC lanzan una `SQLException`.

## 6. OBJETOS PreparedStatement

`PreparedStatement` es una versión extendida y poderosa de la interfaz `Statement`, de la cual hereda, que puede o no ser parametrizada; es decir, a diferencia de `Statement`, puede tomar parámetros de entrada lo que implica un mayor rendimiento en la ejecución de las sentencias. También se ejecuta a través de un protocolo de comunicación binario que no es SQL, que es básicamente un formato no textual utilizado para comunicarse entre clientes y servidores, lo que finalmente reduce el uso de ancho de banda, promoviendo así ejecuciones más rápidas en el servidor. En términos simples, es un objeto que representa una declaración SQL **precompilada**.

Un parámetro de entrada es aquel valor que no se especifica en la sentencia; en su posición la sentencia tendrá un signo de interrogación (?) por cada parámetro de entrada. Antes de ejecutar la sentencia se tiene que especificar un valor para cada parámetro a través de los métodos `setXXX()` apropiados. Estos métodos `setXXX()` los añade la interfaz `PreparedStatement`. Dado que las sentencias de los `PreparedStatement` están precompiladas, su ejecución será más rápida que la de los `Statement`. Por lo tanto, una sentencia SQL que será ejecutada varias veces se suele crear como un objeto `PreparedStatement` para ganar en eficiencia.



Los métodos `execute()`, `executeQuery()` y `executeUpdate()` están sobrecargados y en esta versión, es decir, para los objetos `PreparedStatement` no toman ningún tipo de argumentos; de este modo, a estos métodos

nunca se los tendrá que pasar como parámetro la sentencia SQL a ejecutar. Un *PreparedStatement* ya es una sentencia SQL por sí misma, a diferencia de lo que pasaba con las sentencias *Statement*, que poseían un significado sólo en el momento en que se ejecutaban.

Creando objetos *PreparedStatement*: se ha de llamar al método *prepareStatement()* del objeto *Connection*. En el siguiente ejemplo se puede ver como se crearía un *PreparedStatement* para una sentencia SQL con dos parámetros de entrada.

```
Connection con = DriverManager.getConnection(jdbcUrl,"root","");
PreparedStatement pstmt = con.prepareStatement("update facturas set serie = ? where id = ?");
```

**Utilizando los parámetros de entrada:** antes de poder ejecutar un objeto *PreparedStatement* se tiene que asignar un valor a cada parámetro. Esto se realiza, como hemos dicho, mediante los métodos *setXXX()*, donde XXX es el tipo apropiado para el parámetro. Por ejemplo, si el parámetro es de tipo long, el método a utilizar será *setLong()*.

El primer argumento de los métodos *setXXX()* es la posición ordinal del parámetro, y el segundo argumento es el valor a asignar. Por ejemplo, en el siguiente fragmento de código crea un *PreparedStatement* y, acto seguido, le asigna al primer parámetro un valor de tipo String y al segundo un entero largo:

```
String jdbcUrl = "jdbc:postgresql://localhost:5432/empresa";
con = DriverManager.getConnection(jdbcUrl,"root","");
pstmt = con.prepareStatement("update facturas set serie=? where id=?");
pstmt.setString(1, "B");
pstmt.setLong(2, 1);
pstmt.executeUpdate();
```

Una vez que se han asignado valores a los parámetros de entrada de una sentencia, el objeto *PreparedStatement* se puede ejecutar múltiples veces, hasta que sean borrados los parámetros con el método *clearParameters()*, aunque no hay que llamar a este método cuando se quieran modificar los parámetros de entrada, sino que al lanzar los nuevos métodos *setXXX()* los valores de los parámetros serán reemplazados. Para finalizar, llamamos al método *executeUpdate()* del objeto *PreparedStatement* con lo que se verán reflejados los cambios en la base de datos.

Como ya conocemos, *Statement* y *PreparedStatement* son clases que representan sentencias SQL para interactuar con el servidor de bases de datos. Veamos una diferencia más: *Statement* es más propenso a la **inyección de SQL** (observa la viñeta siguiente e investiga, utilizando un buscador de Internet, cómo se produce).



Otro ejemplo de transacción, ahora con *PreparedStatement*:

```
...
String descripcionProducto = "salsa de pisto";
float preciounit = 1.8f;
int idProveedor=10, idCategoria = 100, existencias =10;

String sqlAltaProducto = "INSERT INTO productos (productoid,proveedorid,categoriaid,
    descripcion, preciounit,existencia) VALUES (?, ?, ?, ?, ?, ?)";
String sqlAltaCategoria =
    "INSERT INTO categorias(categoriaid, nombrecat) " + "VALUES (?, ?)";
```

```

try ( Connection con = DriverManager.getConnection(jdbcUrl,"root","root"); )
{
    ResultSet idGenerados=null;
    //Inicia transacción
    try {
        con.setAutoCommit(false);

        PreparedStatement pstmt = con.prepareStatement(sqlAltaProducto,
            PreparedStatement.RETURN_GENERATED_KEYS);
        pstmt.setInt(1, 14);
        pstmt.setInt(2, idProveedor);
        pstmt.setInt(3, idCategoria);
        pstmt.setString(4, descripcionProducto);
        pstmt.setFloat(5, preciounit);
        pstmt.setInt(6, existencias);
        pstmt.executeUpdate();

        // Obtiene el id del producto que se acaba de registrar
        idGenerados = pstmt.getGeneratedKeys();
        idGenerados.next();
        int idProducto = idGenerados.getInt(1);
        // El identificador del producto añadido es idProducto
        idGenerados.close();
        sentencia.close();

        pstmt = con.prepareStatement(sqlAltaCategoria);
        idCategoria = 800;
        String nomCateg = "deportes";
        pstmt.setInt(1, idCategoria);
        pstmt.setString(2, nomCateg);
        pstmt.executeUpdate();
        // Valida la transacción
        con.commit();
        pstmt.close();
        con.setAutoCommit(true);
    }
    catch (SQLException e)
    {
        con.rollback();
        con.setAutoCommit(true);
        if (idGenerados != null)
            idGenerados.close();
        System.out.println(e.getMessage());
    }
} catch (SQLException sqle) {
    . . .
}

```

Otra duda que a veces nos surge a la hora de acceder a una base de datos es cómo conectar a un **esquema** (Schema) en aquellas bases de datos que tienen más de uno, ya que normalmente, si no los hay, conecta directamente al esquema público. Los esquemas, básicamente, representan la configuración o vista lógica, de todo o parte, de una base de datos relacional. Para conectarnos a uno en concreto bastará con indicarlo en la cadena de conexión:

```
jdbc:postgresql://localhost:5432/mydatabase?currentSchema=myschema
```

También conviene recordar que para mostrar los datos formateados podemos hacer uso de *printf*, en lugar del *print* o *println* ordinario:

```

public static void ejemploFormat() {
    String format = "%4d -%4.4s %-25.25s: %5.2f€\n";
    try (Connection con = DriverManager.getConnection(db, user, pass);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select * from product")) {
        while (rs.next()) {
            System.out.printf(format, rs.getInt("id"), rs.getString("reference"),
                rs.getString("name"), rs.getDouble("price"));
        }
    } catch (SQLException ex) {
        System.err.println(ex.getMessage());
    }
}

```

Puedes ver más información sobre el formateo en: <https://es.wikipedia.org/wiki/Printf>

## 7. DBUTILS

DbUtils es un pequeño conjunto de clases diseñadas para hacer mucho más fácil el trabajo con JDBC, centrándonos en lo que queremos: consultar, actualizar, insertar. Para utilizarlo habremos de incluir la dependencia correspondiente en el *pom.xml* de nuestro proyecto.

```

<dependency>
  <groupId>commons-dbutils</groupId>
  <artifactId>commons-dbutils</artifactId>
  <version>1.7</version>
  <scope>compile</scope>
</dependency>

```

Veamos un ejemplo de uso para una consulta.

```

import java.sql.*;
import org.apache.commons.dbutils.DbUtils;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.ResultSetHandler;
import org.apache.commons.dbutils.handlers.BeanHandler;

public class MainApp {
    static final String JDBC_DRIVER = "org.postgresql.Driver";
    static final String DB_URL = "jdbc:postgresql://localhost:5432/emp";
    static final String USER = "root";
    static final String PASS = "root";

    public static void main(String[] args) throws SQLException {
        Connection con = null;
        QueryRunner queryRunner = new QueryRunner();
        // Paso 1: Registrar JDBC driver (no necesario)
        DbUtils.loadDriver(JDBC_DRIVER);
        // Paso 2: Abrir la conexión
        System.out.println("Conectando a la base de datos...");
        con = DriverManager.getConnection(DB_URL, USER, PASS);
        // Paso 3: Crear un ResultSet Handler para manejar los objetos Empleado
        ResultSetHandler<Empleado> resultHandler = new BeanHandler<Empleado>(Empleado.class);
        try {
            // Obtener los empleados con apellido García

            Empleado emp = queryRunner.query(con, "SELECT * FROM Empleados WHERE
                apellido=?", resultHandler, "García");
            // Muestra valores
            System.out.print("ID: " + emp.getId());

```

```

        System.out.print(", Edad: " + emp.getAge());
        System.out.print(", Nombre: " + emp.getFirst());
        System.out.println(", Apellido: " + emp.getLast());
    } finally {
        DbUtils.close(con);
    }
}
}

```

El siguiente ejemplo mostrará cómo realizar una inserción en la tabla de Empleados con la ayuda de DbUtils. La sintaxis sería:

```

String insertQuery = "INSERT INTO Empleados (id,edad,nombre,apellido) VALUES (?, ?, ?, ?)";
int insertionRecords = queryRunner.update(con,insertQuery,104,30,"Juan","López");

```

donde el objeto *QueryRunner* se utiliza, en este caso, para insertar el empleado en la base de datos.

## 8. POOL DE CONEXIONES

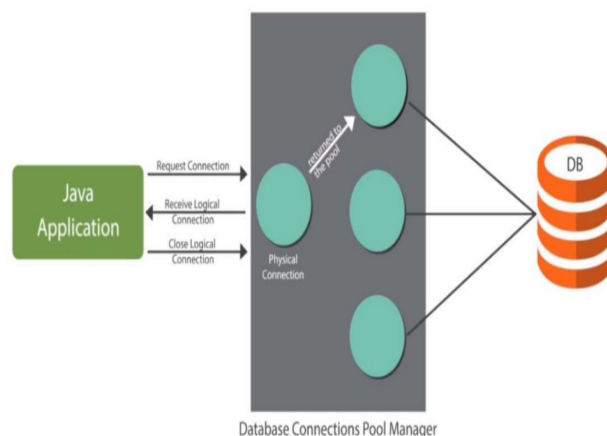
**Pool de conexiones.** En informática, se denomina *connection pool* (agrupamiento o *pool* de conexiones) a una colección de conexiones abiertas a una base de datos, de manera que puedan ser reutilizadas al realizar múltiples consultas o actualizaciones.

Cada vez que un programa cliente necesita comunicarse con una base de datos, establece una conexión utilizando un protocolo especializado. Esta conexión, generalmente se mantiene abierta el tiempo que dura la ejecución del programa y sólo es cerrada al finalizar el trabajo de la aplicación con la base de datos.

Este esquema, sin embargo, no es apropiado para aplicaciones multitarea en las que el mismo programa puede querer realizar en paralelo más de una operación sobre la base de datos. Este caso es típico de las aplicaciones que proveen servicio de páginas web a múltiples usuarios; en ellas, el número de operaciones sobre la base de datos y su cadencia dependen de la actividad de los usuarios de las páginas servidas.

Al mantener abierto un grupo de conexiones, estas son atribuidas a los diferentes hilos de ejecución únicamente el tiempo de una transacción con la base de datos. Cuando finaliza su uso, la conexión se pone a disposición de otro hilo de ejecución, en lugar de cerrarla o de asignarla permanentemente a un único hilo de ejecución.

Según las políticas de agrupamiento de conexiones, cuando todas están en uso se establecen nuevas conexiones, y si ello no es posible, se deja el hilo de ejecución en espera de la liberación de alguna conexión. A la inversa, si pasa mucho tiempo sin que se utilicen las conexiones, algunas de ellas o todas podrían ser cerradas. En Java, el esquema sería el siguiente:



Hay varios tipos de *pool*, dependiendo de qué tipo de aplicación usemos. Esta podría ser:

- de tipo JDBC Driver Manager → para aplicaciones monolíticas (consola, escritorio)
- aplicaciones o servicios web → (Java EE)

Nosotros, en nuestro ejemplo, usaremos la primera. Configurar un *pool* es dependiente de la base de datos con la que queramos conectar. En Postgres, podemos usar alguno de los frameworks de pool de conexiones desarrollado por Apache, por lo que primero debemos incluirlo en nuestro *pom.xml*:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.7.0</version>
</dependency>
```

Posteriormente voy a crear una clase envolvente que me servirá para facilitar las llamadas al pool, que llamaré, por ejemplo, *DBCPDataSource* (DBCP de DataBase Connection Pool) partiendo de un objeto *BasicDataSource*:

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

import org.apache.commons.dbcp2.BasicDataSource;

public class DBCPDataSource {

    private static BasicDataSource ds = new BasicDataSource();

    static {
        ds.setUrl("jdbc:postgresql://localhost:5432/product-manager");
        ds.setUsername("postgres");
        ds.setPassword("postgres");
        ds.setMinIdle(5);
        ds.setMaxIdle(10);
        ds.setMaxOpenPreparedStatements(100);
    }

    public static Connection getConnection() throws SQLException {
        return ds.getConnection();
    }

    private DBCPDataSource(){} }
}
```

Los parámetros a utilizar para el *BasicDataSource* se pueden consultar en <http://commons.apache.org/proper/commons-dbcp/configuration.html>.

Ahora ya en el programa principal podemos crear un método para probar el pool de conexiones:

```
public static void ejemploPool() {

    String format = "%4d - %4.4s %-25.25s: %5.2f€\n";
    try (Connection conn = DBCPDataSource.getConnection();
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery("select * from product")) {

        while (rs.next()) {
            System.out.printf(format, rs.getInt("id"), rs.getString("reference"), rs.getString("name"),
                rs.getDouble("price"));
        }
    } catch (SQLException ex) {
        System.err.println(ex.getMessage());
    }
}
```

Vemos la llamada a *getConnection()* como método estático de *DBCPDataSource* para obtener una conexión disponible del pool con una simple llamada. Ahora podríamos probar que todo funciona correctamente llamando a este método desde *main*. Finalmente, hemos de comentar que los *pools* no han de cerrarse,

aunque sí lo habrían de hacer las conexiones obtenidas. En el código anterior estamos haciendo uso de *try-with-resources* para que la conexión se cierre automáticamente al final.

## 9. PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

### 9.1. Procedimientos almacenados

PostgreSQL, al igual que Oracle y que (con más limitaciones) MySQL, permite definir procedimientos y funciones usando su lenguaje de programación propio, PL/pgSQL. Básicamente, la diferencia entre ambos, procedimientos y funciones, es que los primeros no retornan valor y las segundas sí.

La ejecución de procedimientos almacenados sigue la misma estructura que cualquiera de las sentencias SQL de los ejemplos anteriores, con la excepción de que usaremos la clase *CallableStatement* para representar al procedimiento y el método *execute()* de la misma para ejecutarlo.

```
. . .
String sentenciaSql = "call eliminar_todos_los_productos()";
CallableStatement procedimiento = null;

try {
    procedimiento = con.prepareCall(sentenciaSql);
    procedimiento.execute();
} catch (SQLException sqle) {
    . . .
}
. . .
```

Y en el caso de que el procedimiento almacenado tuviera algún parámetro:

```
. . .
String sentenciaSql = "call eliminar_producto(?)";
CallableStatement procedimiento = null;

try {
    procedimiento = con.prepareCall(sentenciaSql);
    procedimiento.setString(1, nombreProducto);
    procedimiento.execute();
} catch (SQLException sqle) {
    . . .
}
}
```

### 9.2. Funciones almacenadas

En el caso de las funciones almacenadas, para su ejecución recogeremos el valor devuelto en un *ResultSet*, y leeremos el valor contenido en él, ya que nuestras funciones nos devolverán siempre un solo valor (o null en el caso de que no devuelvan nada).

```
CallableStatement cstmt = con.prepareCall("{? = call nombrePoblacion(?, ?)}");
cstmt.setInt(2, codigoPostal);
cstmt.setString(3, pais);
cstmt.registerOutParameter(1, java.sql.Types.VARCHAR);
cstmt.execute();
String ciudad = cstmt.getString(1);
```

## 10. PATRÓN DAO

Un patrón de diseño es una solución probada que resuelve un tipo específico de problema en el desarrollo de software.

Existen una gran cantidad de patrones de diseño clasificados en distintas categorías (<https://refactoring.guru/es/design-patterns>), por ejemplo: de creación, estructurales, de comportamiento, interacción, etc. ¿Cuáles son las ventajas? Permiten tener el código bien organizado, legible y mantenible, además de facilitar la reutilización de código y aumentar la escalabilidad del proyecto.

El patrón Data Access Object (DAO) pretende principalmente independizar la aplicación de la forma de acceder a la base de datos, o cualquier otro tipo de repositorio de datos. Para ello se centraliza el código relativo al acceso al repositorio de datos, con sus operaciones (de inserción, borrado, actualización y consulta) en las clases llamadas DAO. Fuera de las clases DAO no debe haber ningún tipo de código que acceda al repositorio de datos.

Algunas de las ventajas de usar DAO son las siguientes:

- modificar la API de acceso: se podría cambiar el acceso a la base de datos de usar JDBC a usar Hibernate y sólo habría que modificar las clases DAO, sin afectar al resto de la aplicación.
- modificar el repositorio de datos: sería posible que el acceso de los usuarios se hiciera, mediante LDAP, a un servicio de directorio en lugar de tenerlos en una base de datos relacional. También bastaría con cambiar las clases DAO, el resto de la aplicación no sería necesaria modificarla.
- implementación de mecanismos de seguridad: al estar todo el código centralizado en las clases DAO podríamos fácilmente implementar políticas de seguridad en el acceso al repositorio de datos. Mientras que, en caso de no usarlo, sería imposible hacerlo ya que cualquiera podría acceder a los datos sin la obligación de pasar por dicha política de seguridad.

Los DAO, por lo tanto, serán las clases que acceden a la base de datos para cargar y guardar los modelos de nuestra aplicación. Típicamente las llamamos como la clase instanciable (Producto, por ejemplo) seguido de DAO (ProductoDAO), o le damos tal nombre a una interfaz que será implementada por las clases.

```
public class Producto {
    private Integer id;
    private String nombre;
    private Double precio;

    public Producto(Integer id, String name, Double price) {
        this.id = id;
        this.nombre = name;
        this.precio = price;
    }

    //... getters, setters, toString ...
}
```

La interfaz *ProductoDao* contará con las operaciones básicas: insertar, leer, actualizar y eliminar un producto (operaciones CRUD, Create Read Update Delete).

```
public interface ProductoDao {
    public void insert(Producto product);
    public void update(Producto product);
    public void delete(Integer id);
    public Producto read(Integer id);
}
```

La clase *ProductoDaoImpl* será la encargada de implementar la funcionalidad establecida por la interfaz *ProductoDao*.

```

public class ProductoDaoImpl implements ProductoDao {

    //static final String JDBC_DRIVER = "org.postgresql.Driver";
    static final String DB_URL = "jdbc:postgresql://localhost/bbdd";
    static final String DB_USER = "root";
    static final String DB_PASS = "root";

    @Override
    public void insert(Producto product) {
        Connection con = null;
        try {
            // abrir la conexion
            con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
            try (Statement stmt = con.createStatement()) {
                // enviar el comando insert
                stmt.executeUpdate("insert into productos values ("
                    + product.getId() + ","
                    + product.getName() + ","
                    + product.getPrice() + ");");
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public Producto read(Integer id) {
        Connection con = null;
        Producto product = null;

        try {
            // abrir la conexion
            con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASS);
            // consulta select (selecciona el producto con ID especificado)
            try (PreparedStatement ps = con.prepareStatement(
                "select * from productos where id = ?")) {
                // indicar el ID que buscamos
                ps.setInt(1, id);
                // ejecutar la consulta
                try (ResultSet rs = ps.executeQuery()) {
                    if (rs.next()) {
                        // obtener cada columna y mapearlas a la clase Producto
                        product = new Producto(id, rs.getString("nombre"),
                            rs.getDouble("precio"));
                    }
                }
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}

```

```

    } finally {
        if (con != null) {
            try {
                con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    return product;
}

```

La implementación del método *delete*, encargada de eliminar un producto no será mostrada aquí por no extendernos y por ser bastante similar al método *insert*. Para actualizar la información de un producto usaríamos el método *update*; éste también lo dejamos como práctica para el alumno.

Para finalizar vemos la clase principal.

```

public class ProductManager {
    public static void main(String[] args) {
        ProductDao product = new ProductDaoImpl();

        // agregar nuevo producto
        product.insert(new Product(100, "Arroz", 1.50));

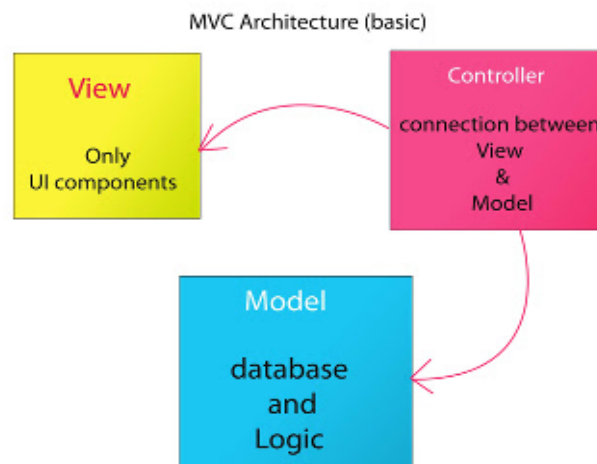
        // obtener el producto con ID = 100
        Product p = product.read(100);
        System.out.println(p);

        // eliminar el producto con ID = 100
        product.delete(100);
    }
}

```

## 11. PATRÓN MODELO-VISTA-CONTROLADOR

El patrón Modelo-Vista-Controlador o MVC (Model-View-Controller en inglés) es un patrón de diseño que busca separar, fundamentalmente en aplicaciones con GUI (interfaz gráfica de usuario), la lógica de la presentación, o **vista**, de la parte del acceso a los datos, **modelo**, facilitando así la reutilización de código y el mantenimiento de la aplicación.



**Modelo.** Esta capa representa todo lo que tiene que ver con el acceso a datos: guardar, actualizar, obtener datos, etc. Básicamente son las clases Java que representan los datos (clases POJO, Plain Old Java Object, o clases instanciables Java) y su acceso.

**Vista.** La vista tiene que ver con la presentación de datos del modelo, es la representación visual del modelo.

**Controlador.** Se encarga de conectar el modelo con las vistas, funciona como un puente entre ambos; el controlador recibe eventos generados por el usuario desde las vistas y se encarga de direccionar al modelo la petición correspondiente.

## 11.1. Ejemplo de aplicación MVC

A continuación se muestra el código completo de una pequeña aplicación con GUI de gestión con las operaciones más básicas (Alta, Modificar, Baja, Búsqueda) gestionada mediante colecciones Java.

Según el patrón MVC, la aplicación quedará separada en tres partes: *Model*, *View* y *Controller*. A esas 3 partes habrá que añadir las clases *POJOs* que son las clases instanciables que representan a los objetos o elementos que se gestionan en la aplicación. Contendrán atributos, constructores, getters, setters y los métodos que proceda implementar para cada clase. En este caso hay una única clase *POJO* que es *Animal.java*, puesto que la aplicación permite gestionar una base de datos de animales, mediante un array, de forma que queda como ejercicio muy interesante transformar esta aplicación en otra que gestione la información con Bases de Datos (MySQL ó PostgreSQL, por ejemplo). Será entonces cuando se aprecien las ventajas de la separación Modelo-Vista-Controlador y la reutilización real de código entre distintas versiones de esta aplicación. El aspecto de la aplicación sería algo como:

```
/**
 * Clase que representa a un Animal
 */
public class Animal {

    private String nombre;
    private String raza;
    private String características;
    private float peso;

    // Constructores, getters, setters, toString, etc
}
```

La clase *Animal* representa a los animales que vamos a gestionar y contendrá tantos atributos como características queramos almacenar. Además, incluiremos los correspondientes constructores, *getters* y *setters*. Si procede, podemos añadir otros métodos que mejoren el comportamiento de la clase.

```

/**
 * Clase que contiene la GUI de la Aplicación
 */
public class Ventana {
    private JPanel panel;
    private JTextField tfNombre;
    private JTextField tfRaza;
    // Resto de controles swing
    . . .

    public Ventana() {

        JFrame frame = new JFrame("Ventana");
        frame.setContentPane(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}

```

La clase Ventana contendrá exclusivamente el diseño de la ventana y la inicialización de la misma según convenga en cada caso. Ni siquiera los *Listeners* aparecerán en esta ventana.

```

/**
 * Controlador para la ventana
 * Recibe las entradas del usuario desde la ventana (View) y se
 * comunica con el Model si es necesario
 */
public class VentanaController implements ActionListener {

    private VentanaModel model;
    private Ventana view;

    private int posicion;

    public VentanaController(VentanaModel model, Ventana view) {
        this.model = model;
        this.view = view;
        anadirActionListener(this);
        posicion = 0;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        String actionCommand = event.getActionCommand();
        Animal animal = null;

        switch (actionCommand) {
            case "Nuevo":
                view.tfNombre.setText("");
                view.tfNombre.setEditable(true);
                view.tfCaracteristicas.setText("");
                view.tfCaracteristicas.setEditable(true);
                // Limpiar/Resetear resto de componentes
                . . .

```

```

        . . .
        view.btGuardar.setEnabled(true);
        break;
    case "Guardar":
        if (view.tfNombre.getText().equals("")) {
            Util.mensajeError("El nombre es un campo obligatorio", "Nuevo Animal");
            return;
        }
        animal = new Animal();
        animal.setNombre(view.tfNombre.getText());
        animal.setRaza(view.tfRaza.getText());
        animal.setCaracteristicas(view.tfCaracteristicas.getText());
        animal.setPeso(Float.parseFloat(view.tfPeso.getText()));
        model.guardar(animal);
        view.btGuardar.setEnabled(false);
        break;
    case "Modificar":
        animal = new Animal();
        animal.setNombre(view.tfNombre.getText());
        animal.setRaza(view.tfRaza.getText());
        animal.setCaracteristicas(view.tfCaracteristicas.getText());
        animal.setPeso(Float.parseFloat(view.tfPeso.getText()));
        model.modificar(animal);
        break;
    case "Eliminar":
        if (JOptionPane.showConfirmDialog(null, "¿Está seguro?", "Eliminar",
            JOptionPane.YES_NO_OPTION) == JOptionPane.NO_OPTION)
            return;
        model.eliminar();
        animal = model.getActual();
        cargar(animal);
        break;
        . . .
        . . .
    default:
        break;
}
}

. . .
. . .
}

// Carga los datos de un animal en la vista
private void cargar(Animal animal) {
    if (animal == null)
        return;

    view.tfNombre.setText(animal.getNombre());
    view.tfCaracteristicas.setText(animal.getCaracteristicas());
    . . .
    . . .
}

. . .
// Añade el Listener a todos los controles de la View
private void anadirActionListener(ActionListener listener) {

```

```

view.btNuevo.addActionListener(listener);
view.btGuardar.addActionListener(listener);

```

```

        . . .
        . . .
    }
}

```

Este código hace uso, para mostrar mensajes de error, de la siguiente clase.

```

public class Util {
    public static void mensajeError(String mensaje, String titulo) {
        JOptionPane.showMessageDialog(null, mensaje, titulo, JOptionPane.ERROR_MESSAGE);
    }

    public static void mensajeInformacion(String mensaje, String titulo) {
        JOptionPane.showMessageDialog(null, mensaje, titulo, JOptionPane.INFORMATION_MESSAGE);
    }
}

```

La clase `VentanaController` contiene el *Controller* de la ventana, que hará de intermediario entre la capa *View*, que es la GUI de la aplicación, y el *Model*, que contiene toda la lógica y el acceso a los datos. Además, se encarga de gestionar los eventos de cada componente de la GUI (implementando los *Listeners* necesarios).

También se incluyen en este componente todas las validaciones de datos relacionadas con los valores de entrada por parte del usuario y la visualización de los mensajes de errores que correspondan.

```

// Modelo para la ventana
public class VentanaModel {
    private ArrayList<Animal> listaAnimales;
    private int posicion;

    public VentanaModel() {
        listaAnimales = new ArrayList<>();
        posicion = 0;
    }

    /* Guarda un animal en la lista
       @param animal */
    public void guardar(Animal animal) {
        listaAnimales.add(animal);
        posicion++;
    }

    /* Modifica los datos del animal actual
       @param animalModificado */
    public void modificar(Animal animalModificado) {
        Animal animal = listaAnimales.get(posicion);
        animal.setNombre(animalModificado.getNombre());
        animal.setCaracteristicas(animalModificado.getCaracteristicas());
        animal.setRaza(animalModificado.getRaza());
        animal.setPeso(animalModificado.getPeso());
    }

    // Elimina el animal actual
    public void eliminar() {
        listaAnimales.remove(posicion);
    }

    public Animal getActual() {

```

```

    return listaAnimales.get(posicion);
}

/* Obtiene el animal que está en primera posición en la lista
   @return */
public Animal getPrimero() {
    posicion = 0;
    return listaAnimales.get(posicion);
}

/* Obtiene el animal que está en la posición anterior a la actual
   @return */
public Animal getAnterior() {
    if (posicion == 0)
        return null;
    posicion--;
    return listaAnimales.get(posicion);
}

/* Obtiene el animal que está en la posición siguiente a la actual
   @return */
public Animal getSiguiente() {
    if (posicion == listaAnimales.size() - 1)
        return null;
    posicion++;
    return listaAnimales.get(posicion);
}

/* Obtiene el animal que está en la última posición de la lista
   @return */
public Animal getUltimo() {
    posicion = listaAnimales.size() - 1;
    return listaAnimales.get(posicion);
}
}

```

La clase `VentanaModel`, puesto que es el *Model*, contendrá todos los métodos que proporcionan el acceso a los datos y la lógica de la aplicación. En este caso contiene todos los métodos que permiten obtener información de los animales, los resultados de las búsquedas, así como las operaciones de inserción, modificación y eliminación.

Aquí todas las operaciones se realizan sobre una colección `ArrayList`. Si quisiéramos que la aplicación utilizara una Base de Datos como motor de almacenamiento, sólo tendríamos que modificar este componente de la aplicación haciendo que los mismos métodos (incluso respetando su declaración: nombre, tipo devuelto y parámetros) utilicen una Base de Datos para realizar todas las operaciones. No sería necesario realizar ninguna modificación en el resto de la aplicación.

```

/* Aplicación para la gestión de animales utilizando el patrón MVC
 * Clase principal, que solamente lanza la aplicación */
public class Aplicacion {
    public static void main(String args[]) {
        Ventana view = new Ventana();
        VentanaModel model = new VentanaModel();
        VentanaController controller = new VentanaController(model, view);
    }
}

```

En la clase `Aplicacion`, que contiene el método `main` que lanzará la aplicación, simplemente tenemos que instanciar los tres componentes de nuestra aplicación (*Model*, *View* y *Controller*) y "conectarlos" mediante el último.

---

*Esta documentación, creada por Ricardo Cantó Abad, se distribuye bajo los términos de la licencia Creative Commons Atribución-NoComercial-CompartirIgual 3.0 Unported (CC BY-NC-SA 3.0) (<http://creativecommons.org/licenses/by-nc-sa/3.0/es/deed.ca>).*

