



Conexión con bases de datos



Conectores

Los Sistemas Gestores de Bases de Datos (SGBD) tanto si son de bases de datos relacionales, XML nativas, de Objetos o No SQL tienen sus propios lenguajes para gestionar la información que contienen y que son diferentes a los lenguajes de programación.

Para que los programas puedan interactuar con los SGBD es necesario utilizar librerías adicionales que proporcionan ciertos métodos para la comunicación con las bases de datos (API) y se denominan *Conectores*.



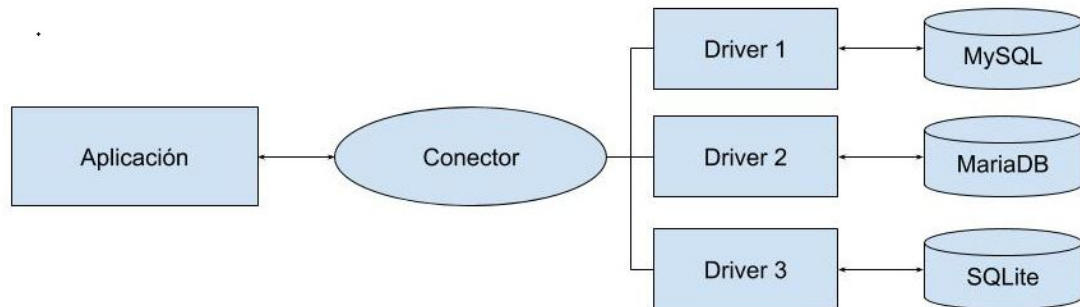
Conectores

Para trabajar en una base de datos relacional (que recordemos son el tipo de base de datos más difundida actualmente) se utiliza el lenguaje SQL, que aunque es un estándar, existen múltiples versiones y cada base de datos utiliza su propia versión de SQL y seguramente con modificaciones propias.

Para poder conectarnos a bases de datos desde nuestros programas utilizamos **drivers** de conexión específicos con la base de datos (específicos según el tipo de SGBD y versión). Estos drivers proporcionan interfaces comunes, que nos permiten abstraernos de las particularidades de cada base de datos.

Conectores

De esta forma, junto con los drivers, podemos utilizar los conectores como un modelo de arquitectura de desarrollo, creando una abstracción en la capa de acceso a datos de forma que la aplicación sea independiente de qué SGBD utilicemos y podamos utilizar uno u otro sin apenas tener que modificar nada de nuestra aplicación.





JDBC

La arquitectura Java Database Connectivity (JDBC) para Java se basa en drivers para la conexión con los SGBD.

Este conector proporciona una API común para la conexión con los drivers, que a su vez implementan esta interfaz común con las peculiaridades de cada base de datos.



Apertura y cierre de conexiones

Los Drivers de JDBC están disponibles en ficheros .jar que deben ser añadidos a nuestro proyecto como librerías.

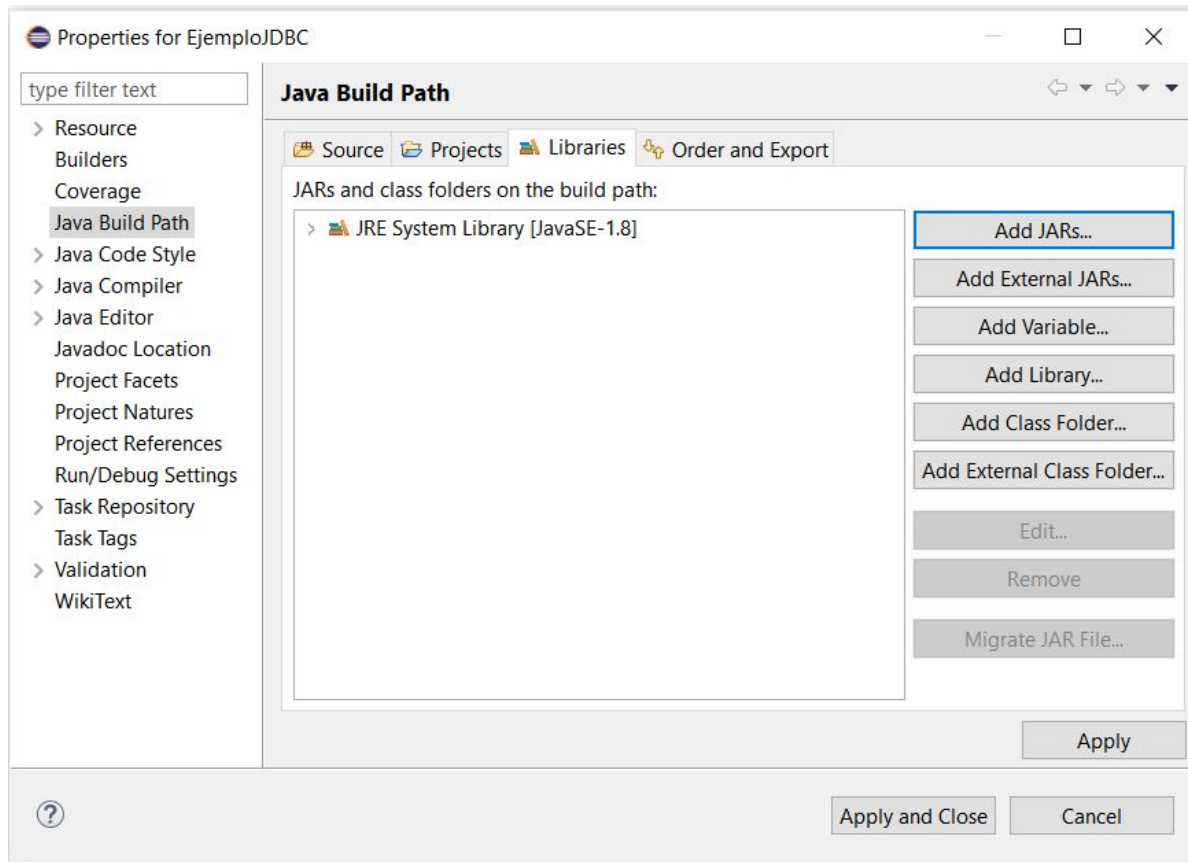
Estos drivers se pueden descargar de las páginas webs de los diferentes SGBD.

MariaDB: <https://downloads.mariadb.org/connector-java/>

MySQL: <https://www.mysql.com/products/connector/>




En Eclipse, para añadir estos .jar como librerías debemos ir a Configure Build Path... y dentro a la pestaña Librerías

Deberemos añadir los JARs. Si los añadimos como .jar externo se tomará una ruta absoluta, si seleccionamos la primera opción se tomará un jar de una ruta en nuestro proyecto (deberemos haberlo tenido que añadir previamente)













Apertura y cierre de conexiones

Una vez añadidos los .jar veremos que están referenciados en nuestro proyecto

- ▼  Referenced Libraries
 - >  mariadb-java-client-2.5.4.jar
 - >  mysql-connector-java-8.0.18.jar

Si desplegamos cualquiera de estos drivers de conexión veremos un Driver.class que nos hará falta más adelante.

- ▼  mariadb-java-client-2.5.4.jar
 - ▼  org.mariadb.jdbc
 - >  BasePreparedStatement.class
 - >  BlobOutputStream.class
 - >  CallableFunctionStatement.class
 - >  CallableParameterMetaData.class
 - >  CallableProcedureStatement.class
 - >  CallParameter.class
 - >  ClientSidePreparedStatement.class
 - >  Driver.class



Apertura y cierre de conexiones

Para poder hacer uso del driver JDBC en nuestra aplicación será necesario antes de nada que este se haya cargado en memoria. Para ello utilizamos la siguiente instrucción:

Class.forName(Nombre de la clase);

Para establecer la conexión se puede utilizar la clase **DriverManager**, con el método **getConnection**, que recibe una URL de conexión.

Esta URL tiene un formato de identificador de base de datos, host, puerto y nombre de la bbdd.

Por ejemplo:

jdbc:mysql://localhost:3306/miBaseDeDatos



Ejercicio

Crear una base de datos local usando WAMP y PHPMyAdmin o MySQLWorkbench llamada EjercicioBBDD.

Crear una clase llamada PersistenciaBBDD con los métodos:

obtenerConexión: Este método creará el objeto de tipo Connection mediante el DriverManager y lo devolverá.

cargarDriver: Método que realizará la instrucción de carga del driver (dentro se hace el método `Class.forName(...)`)



La interfaz Statement

Esta interfaz se utiliza para la ejecución de sentencias SQL. Se puede obtener un **Statement** mediante el método *getStatement()* de **Connection**.

Método	Funcionalidad
ResultSet executeQuery(String sql)	Ejecuta una sentencia Select y devuelve un ResultSet para acceder a los resultados.
ResultSet getResultSet()	Obtiene el conjunto de resultados tras la ejecución de una consulta.
int executeUpdate(String sql)	Ejecuta una sentencia Insert, Update o Delete y devuelve el número de filas afectadas.
boolean execute(String sql)	Se puede usar para ejecutar cualquier tipo de consulta. Es el método preferente en ejecución de sentencias DDL tales como Create, Alter y Drop.
void close()	Cierra el Statement.



Ejemplo de ejecución de consultas DDL

Para ejecutar consultas que crean, modifican o borran tablas, vistas y elementos de la base de datos se utiliza el método `execute()`.

```
private void CargarDriver() throws ClassNotFoundException {  
    Class.forName("org.mariadb.jdbc.Driver");  
} // CargarDriver  
  
public Connection obtenerConexion() throws SQLException {  
    String url = "jdbc:mysql://localhost:3306/Transparencias";  
    String usuario = "root";  
    String pass = "";  
    Connection c = DriverManager.getConnection(url, usuario, pass);  
    return c;  
} // obtenerConexion
```

```

public void CrearTablaClientes() {
    Connection c = null;
    try {
        CargarDriver();
        c= obtenerConexion();
        String consulta="CREATE TABLE CLIENTES "
            + "(DNI CHAR(9) NOT NULL, APELLIDOS VARCHAR(32) NOT NULL, CP CHAR(5), PRIMARY KEY(DNI))";
        Statement s=c.createStatement();
        s.execute(consulta);
        s.close();
    } catch (ClassNotFoundException ex) {
        System.out.println("Error al cargar el driver de la bbdd");
    } catch (SQLException e) {
        System.out.println("Error en la ejecución de la consulta");
        e.printStackTrace();
    } finally {
        try {
            if(c!=null && !c.isClosed())
                c.close();
        } catch (SQLException e) {
            System.out.println("Error al cerrar la conexión");
        }
    }
}

```



Consultas que modifican contenidos

Para realizar consultas de INSERT, UPDATE o DELETE utilizamos el método **executeUpdate()**, que devolverá el número de filas afectadas.

En el siguiente ejemplo se muestra la ejecución de una inserción de varias filas en la tabla creada anteriormente.

```

public void insertarClientes() {
    Connection c = null;
    try {
        CargarDriver();
        c= obtenerConexion();
        String consulta="INSERT INTO CLIENTES"
            + "(DNI,APELLIDOS,CP) VALUES "
            + "('78978978A','NADAL','15009'),"
            + "('12345678F','DJOKOVIC','15008'),"
            + "('55544587J','FEDERER','15007'),"
            + "('88965475K','MURRAY','15006');"
        Statement s=c.createStatement();
        s.executeUpdate(consulta);
        s.close();
    } catch (ClassNotFoundException ex) {
        System.out.println("Error al cargar el driver de la bbdd");
    } catch (SQLException e) {
        System.out.println("Error en la ejecución de la consulta");
        e.printStackTrace();
    } finally {
        try {
            if(c!=null && !c.isClosed())
                c.close();
        } catch (SQLException e) {
            System.out.println("Error al cerrar la conexión");
        }
    }
}
}

```



Consultas de selección

Para realizar consultas SELECT se puede utilizar el método **executeQuery()** que devuelve un **ResultSet** con los resultados en una estructura en memoria en forma de tabla.

Métodos de **ResultSet** para consultar contenidos

Método	Funcionalidad
boolean next()	El cursor de ResultSet apunta inicialmente al elemento anterior a la primera fila de resultados y puede moverse hasta la posición siguiente a la última fila de resultados. Este método mueve el cursor a la siguiente fila.
getXXX(int) getXXX(String)	Obtiene el contenido de la columna de la posición o nombre indicado. Hay diferentes métodos para diferentes tipos: <i>getInt()</i> , <i>getString()</i> , <i>getDate()</i> ,...
close()	Cierra el ResultSet . Debe hacerse siempre.



Ejemplo de ejecución de consulta de selección

Con nuestro ejemplo de la tabla Clientes, para recuperar los datos insertados anteriormente:

```
try {
    CargarDriver();
    c= obtenerConexion();
    String consulta="SELECT DNI,APELLIDOS,CP "
        + "FROM CLIENTES";
    Statement s=c.createStatement();
    ResultSet rs= s.executeQuery(consulta);
    while(rs.next()) {
        System.out.println("dni:" + rs.getString("DNI"));
        System.out.println("apellidos:" + rs.getString("APELLIDOS"));
        System.out.println("cp:" + rs.getString("CP"));
    }
    rs.close();
    s.close();
} catch (ClassNotFoundException ex) {
    System.out.println("Error al cargar el driver de la bbdd");
} catch (SQLException e) {
    System.out.println("Error en la ejecución de la consulta");
    e.printStackTrace();
} finally {
    try {
        if(c!=null && !c.isClosed())
            c.close();
    } catch (SQLException e) {
        System.out.println("Error al cerrar la conexión");
    }
}
```



Sentencias preparadas

En los ejemplos anteriores hemos visto ejecución de sentencias fijas, definiendo a mano la sentencia. En el caso de que queramos realizar, por ejemplo, sentencias de inserción según unos parámetros, podemos realizarlo de dos formas:

La primera sería mediante la creación de la sentencia de consulta en la cadena, por ejemplo con `String.format`: `String.format ("INSERT INTO CLIENTES (DNI,APELLIDOS,CP) VALUES ('%s','%s','%s')",dni,apellidos,cp);`



Sentencias preparadas

Este planteamiento plantea graves problemas:

- Seguridad: Al construir de esta forma las sentencias podemos hacer el sistema vulnerable a ataques mediante inyección SQL. Por ejemplo, en el ejemplo anterior, si alguno de esos valores se obtiene de una interfaz, se puede indicar (por ejemplo para el nombre) otro valor que no sea correcto: por ejemplo `'15301;drop database;'` nuestra consulta ejecutaría de forma correcta el insert pero también ejecutaría ese “drop database”, que eliminaría todas las tablas de la base de datos. Esto podría evitarse verificando el contenido de las variables pero resulta engorroso y complejo.
- Rendimiento: Cuando una consulta se envía al SGBC se compila antes de ejecutarse (analizandose y generando un plan de ejecución para ella), lo que consume un tiempo. Si entre consulta y consulta solo cambia el valor de algunas variables se compila independientemente aunque se genere el mismo plan de ejecución.



Sentencias preparadas

Las sentencias preparadas nos evitan estos problemas.

Para realizarlas utilizamos un **PreparedStatement**, que se obtiene mediante el método **getPreparedStatement** de **Connection**. A este método ya se le pasa la consulta SQL y en dicha consulta se indican mediante el carácter `?` los marcadores que serán sustituidos por valores.



Sentencias preparadas

Métodos de PreparedStatement

Método	Funcionalidad
ResultSet executeQuery() int executeUpdate() boolean execute()	Ejecutan la consulta que se pasó al método constructor.
setXXX(int pos, YYY valor)	Se utiliza para asignar un valor a un marcador indicando su posición (empiezan en 1). Los hay con distintos nombres dependiendo del tipo.
setNull(int pos, int tipoSQL)	Asigna un valor NULL a una columna. Debe indicarse el tipo, que está definido dentro de java.sql.Types

```

public void insertarCliente(String dni, String nombre, String cp) {
    Connection c = null;
    try {
        CargarDriver();
        c= obtenerConexion();
        String consulta="INSERT INTO CLIENTES"
            + "(DNI,APELLIDOS,CP) VALUES "
            + "(?,?,?);";
        PreparedStatement ps=c.prepareStatement(consulta);
        ps.setString(1, dni);
        ps.setString(2, nombre);
        ps.setString(3, cp);
        ps.executeUpdate();
        ps.close();
    } catch (ClassNotFoundException ex) {
        System.out.println("Error al cargar el driver de la bbdd");
    } catch (SQLException e) {
        System.out.println("Error en la ejecución de la consulta");
        e.printStackTrace();
    } finally {
        try {
            if(c!=null && !c.isClosed())
                c.close();
        } catch (SQLException e) {
            System.out.println("Error al cerrar la conexión");
        }
    }
}
}

```



Transacciones

Todos los SGBD relacionales tienen soporte para transacciones pero cada uno tiene sus particularidades. Por ejemplo, Oracle las tiene habilitadas por defecto, con lo que es necesario realizar un Commit para que se apliquen los cambios. Otras como MySQL las tienen deshabilitadas por defecto.

Además cada SGBD puede tener diferentes consultas para la gestión de las transacciones. JDBC proporciona una interfaz común para todas las bases de datos (la consulta que realiza la transacción la gestionará el driver) aunque si se desea también puede ejecutarse directamente la sentencia en SQL.

Las transacciones se definen con una sentencias de **Inicio de transacción**, tras la cual se ejecutarán sentencias y al finalizar se realiza un **Commit** para confirmar las sentencias. En caso de haber algún problema se puede abortar la ejecución de la transacción mediante un **Rollback**.



Transacciones

La interfaz Connection tiene una serie de métodos relacionados con las transacciones.

Métodos de Connection

Método	Funcionalidad
void setAutoCommit(boolean autoCommit)	Al indicar setAutoCommit(false) indicamos que no se realice una confirmación tras cada sentencia ejecutada. Sería equivalente a iniciar la transacción.
void commit()	Equivale a una sentencia commit en SQL.
void rollback()	Se descartan todos los cambios pendientes de commit. Se suele hacer en respuesta a una <i>SQLException</i> .



Ejemplo de inserción con transacciones

En las siguientes transparencias se muestra cómo podría hacerse una inserción de tres elemento en la tabla Clientes de forma que o se insertan los tres o ninguno mediante el uso de transacciones.

```
public void insertarClientesConTransaccion() {  
    Connection c = null;  
    try {  
        CargarDriver();  
        c = obtenerConexion();  
        c.setAutoCommit(false); // Inicio de transaction  
  
        String consulta = "INSERT INTO CLIENTES" + "(DNI,APELLIDOS,CP) VALUES " + "(?,?,?);";  
        PreparedStatement ps = c.prepareStatement(consulta);  
        ps.setString(1, "11111111G");  
        ps.setString(2, "MARTINEZ");  
        ps.setString(3, "11001");  
        ps.executeUpdate();  
  
        ps.setString(1, "22222222H");  
        ps.setString(2, "SANCHEZ");  
        ps.setString(3, "11002");  
        ps.executeUpdate();  
  
        ps.setString(1, "33333333I");  
        ps.setString(2, "VERDASCO");  
        ps.setString(3, "11003");  
        ps.executeUpdate();  
  
        c.commit(); // Commit de la transacción
```

```

        c.commit(); //Commit de la transacción
        ps.close();
    } catch (ClassNotFoundException ex) {
        System.out.println("Error al cargar el driver de la bbdd");
    } catch (SQLException e) {
        System.out.println("Error en la ejecución de la consulta");
        try {
            System.out.println("Se realiza el rollback");
            c.rollback(); //Rollback si ha habido un error en la ejecución de la consulta
        } catch (SQLException e1) {
            System.out.println("Error haciendo el rollback");
            e1.printStackTrace();
        }
        e.printStackTrace();
    } finally {
        try {
            if(c!=null && !c.isClosed())
                c.close();
        } catch (SQLException e) {
            System.out.println("Error al cerrar la conexión");
        }
    }
}
} //insertarClientesConTransaccion

```



Claves autogeneradas

Habitualmente se utiliza como clave primaria de las tablas un valor numérico autogenerado, de forma que sea el SGBD el que cree y asigne el valor a los nuevos registros.

La instrucción que indica si una clave es autoincremental varía según el tipo de base de datos utilizada. En MySql y en MariaDb se usa el identificador `AUTO_INCREMENT` (En Oracle, por ejemplo se usa el identificador `GENERATED BY DEFAULT AS IDENTITY`)

```
CREATE TABLE FACTURAS
(NUM_FACTURA INTEGER AUTO_INCREMENT NOT NULL, DNI_CLIENTE CHAR(9) NOT NULL,
PRIMARY KEY(NUM_FACTURA), FOREIGN KEY FK_FACT_DNI_CLIENTES (DNI_CLIENTE)
REFERENCES CLIENTES(DNI))
```



Claves autogeneradas

Una vez definida la tabla con la clave autogenerada queda saber cómo poder recuperar estos valores a la vez que se realiza la inserción. Esto podemos hacerlo a través de **Statement** o de **Connection** (al crear un **PreparedStatement**)

Métodos de Statement

Método	Funcionalidad
<code>int executeUpdate(String sql, int autogeneratedKeys)</code>	<u>autogeneratedKeys</u> puede tomar el valor: <ul style="list-style-type: none">• Statement.RETURN_GENERATED_KEYS• Statement.NO_GENERATED_KEYS
<code>ResultSet getGeneratedKeys()</code>	Devuelve un <code>ResultSet</code> con los valores de claves autogeneradas. Normalmente será solo uno.




Claves autogeneradas

Métodos de Connection

Método	Funcionalidad
PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)	Prepara una sentencia. Para que devuelva claves autogeneradas se debe indicar un valor para <u>autoGeneratedKeys</u> de PreparedStatement.RETURN_GENERATED_KEYS

En las siguientes transparencias se muestran métodos de generación de facturas obteniendo la clave generada, tanto con **Statement** como con **PreparedStatement**



```
CargarDriver();
c = obtenerConexion();
c.setAutoCommit(false); // Inicio de transaction

String consulta = "INSERT INTO FACTURAS" + "(DNI_CLIENTE) VALUES " + "(?)";
PreparedStatement ps = c.prepareStatement(consulta,Statement.RETURN_GENERATED_KEYS);
ps.setString(1, "11111111G");
ps.executeUpdate();

ResultSet rs= ps.getGeneratedKeys();
rs.next();//Obtenemos el siguiente valor del resultset (normalmente lo hacemos en el while)
int id= rs.getInt(1);//Obtenemos el valor del id generado
System.out.println("Id generado:" + id);

c.commit(); // Commit de la transacción
ps.close();
```




Ejecución por lotes

Como se ha indicado anteriormente, cada ejecución de una sentencia tiene un tiempo de compilación y preparación del plan de ejecución por parte del SGBD. Esto conlleva que si realizamos las sentencias una a una, pese a ser el mismo tipo de sentencia (inserción en la misma tabla, por ejemplo) pero con distintos valores, lleve mucho tiempo de ejecución.

Como alternativa, ante la ejecución de sentencias separadas está la posibilidad de añadir la sentencia a un lote de ejecución y, al finalizar, ejecutar el lote de forma completa.

Para añadir una sentencia a un lote de ejecución (el lote se crea solo al añadir una sentencia) se usa **`addBatch(String sql)`** en **Statement** y **`addBatch()`** en **PreparedStatement**.

Para ejecutar los lotes se utiliza **`executeBatch()`** o **`executeLargeBatch()`**. También **CallableStatement** (que veremos en las próximas diapositivas) permite la ejecución por lotes.

```
\
CargarDriver();
c = obtenerConexion();
c.setAutoCommit(false); // Inicio de transaction

String consulta = "INSERT INTO CLIENTES" + "(DNI, APELLIDOS, CP) VALUES " + "(?, ?, ?)";
PreparedStatement ps = c.prepareStatement(consulta);

ps.setString(1, "22222222F");
ps.setString(2, "Medvédev");
ps.setString(3, "21001");
ps.addBatch(); // Se añade al lote de ejecución

ps.setString(1, "33333333G");
ps.setString(2, "Tsitsipás");
ps.setString(3, "21002");
ps.addBatch(); // Se añade al lote de ejecución

ps.setString(1, "44444444H");
ps.setString(2, "Anisimova");
ps.setString(3, "21003");
ps.addBatch(); // Se añade al lote de ejecución

ps.executeBatch(); // Ejecución al lote
c.commit(); // Commit de la transacción
ps.close();
```



Procedimientos y funciones almacenados

El lenguaje SQL incluye lenguajes de programación estructurados basados en SQL, pero que incluyen sentencias condicionales y bucles.

Los procedimiento y funciones almacenados son bloques de este código que pueden tener parámetros de entrada, de salida y de entrada-salida. En el caso de las funciones también pueden devolver valores.

Estos bloques pueden ser invocados desde un intérprete SQL pero también desde una aplicación por medio de JDBC, por medio de la interfaz **CallableStatement**



Procedimientos y funciones almacenados

Un ejemplo de procedimiento almacenado, creado en SQL sería el siguiente:

El procedimiento recibirá el parámetro de entrada *in_dni* y devolverá una serie de filas con el dni y apellidos de clientes con el apellido menor en orden alfabético al del cliente de dni indicado. El otro parámetro de entrada *inout_long* se devuelve sumándole el tamaño del apellido del cliente con el dni indicado.

```
DELIMITER $$
CREATE PROCEDURE `listado_parcial_clientes`
(IN in_dni CHAR(9), INOUT inout_long INT)
BEGIN
    DECLARE apell VARCHAR(32) DEFAULT NULL;
    SELECT APELLIDOS FROM clientes WHERE DNI=in_dni INTO apell;
    SET inout_long = inout_long + LENGTH(apell);
    SELECT DNI,APELLIDOS FROM clientes
        WHERE APELLIDOS<=apell ORDER BY APELLIDOS;
END$$
DELIMITER ;
```

Procedimientos y funciones almacenados

Una llamada a este procedimiento desde un editor SQL podría ser el siguiente:

```
SET @tam=0;  
CALL listado_parcial_clientes('78978978A',@tam);  
SELECT @tam;
```

DNI	APELLIDOS
12345678F	DJOKOVIC
55544587J	FEDERER
11111111G	MARTINEZ
55555555H	Muguruza
88965475K	MURRAY
78978978A	NADAL

@tam

5



Procedimientos y funciones almacenados

Para realizar la llamada a procedimientos y funciones almacenados es necesario seguir un patrón en la llamada que se va a realizar, indicando el marcador para los parámetros (igual que hacíamos con las consultas de PreparedStatement):

Procedimientos: *{call nombre_procedimiento(?,?,...)}*

Funciones: *{? = call nombre_funcion(?,?,...)}*



Procedimientos y funciones almacenados

Para realizar la llamada y la asignación de parámetros con obtención de resultados se usa la interfaz `CallableStatement`.

Métodos de `CallableStatement`

Método	Funcionalidad
<code>setXXX(int pos, YYYY valor)</code>	Se utiliza para dar valor de diversos tipos a un parámetro indicado por su posición (empezando en 1).
<code>void registerOutParameter(int pos, int sqlType)</code>	Registra un parámetro de salida para poder obtener el valor devuelto en él.
<code>getXXX(int pos)</code>	Obtiene el valor de un parámetro de salida.
<code>ResultSet getResultSet()</code>	Obtiene el <code>ResultSet</code> resultado del procedimiento o función.

```

try {
    CargarDriver();
    c = obtenerConexion();
    CallableStatement st = c.prepareCall("{ call listado_parcial_clientes(?,?) }");

    st.setString(1, dni); // Asigno el valor del primer parámetro (el dni)
    st.setInt(2, 0); // Asigno el valor del segundo parámetro (el tamaño)
    // Registro el segundo parámetro como parámetro de salida
    st.registerOutParameter(2, java.sql.Types.INTEGER);
    if (!st.execute()) {
        System.out.println("El procedimiento no ha devuelto resultados");
    } else {
        ResultSet rs = st.getResultSet();

        // Ojo, los valores devueltos por el procedimiento se toman del
        // CallableStatement, no del ResultSet
        int inout_long = st.getInt(2);
        System.out.println("Valor devuelto de tamaño apellidos:" + inout_long);

        while (rs.next()) {
            System.out.println("DNI:" + rs.getString("DNI"));
            System.out.println("Apellidos:" + rs.getString("APELLIDOS"));
        }
    }
} catch (SQLException e) {

```