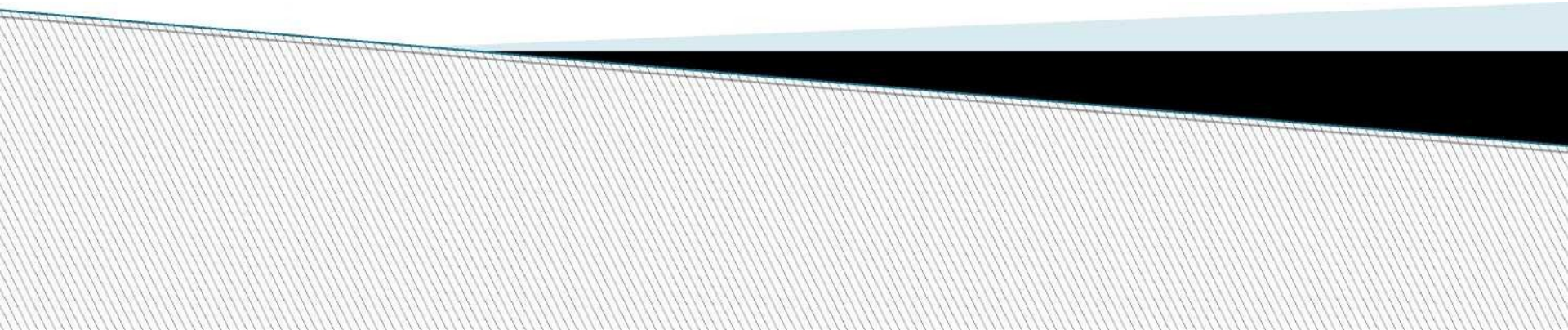


Acceso Java a base de datos.

JDBC



Acceso a datos

- ▶ Cuando abordamos el desarrollo de una aplicación en Java, uno de los primeros requerimientos que debemos resolver es la integración con una **base de datos** para guardar, actualizar y recuperar la información que utiliza nuestra aplicación.
- ▶ Se llama “persistencia” de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento.

Acceso a datos


- ▶ La persistencia en bases de datos relacionales se suele implementar mediante :
 - El desarrollo de la funcionalidad específica utilizando la tecnología JDBC.

JDBC

- ▶ JDBC es el acrónimo de Java Database Connectivity.
- ▶ Un API (Application programming interface) incluida en Java, que describe o define una librería estándar para acceso a fuentes de datos, principalmente orientado a Bases de Datos relacionales que usan SQL (Structured Query Language).
- ▶ Consiste en un **conjunto de clases** e interfaces, escritas en Java, que ofrecen un completo API **para la programación de bases de datos**.

API JDBC

Es una solución cien por cien Java para el acceso a bases de datos. Básicamente el API JDBC hace posible realizar las siguientes tareas:

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

API JDBC

JDBC al estar escrito completamente en Java 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, Sysbase, o SQL Server.

Además podrá ejecutarse en cualquier sistema que posea una Máquina Virtual de Java.

Packages del API JDBC

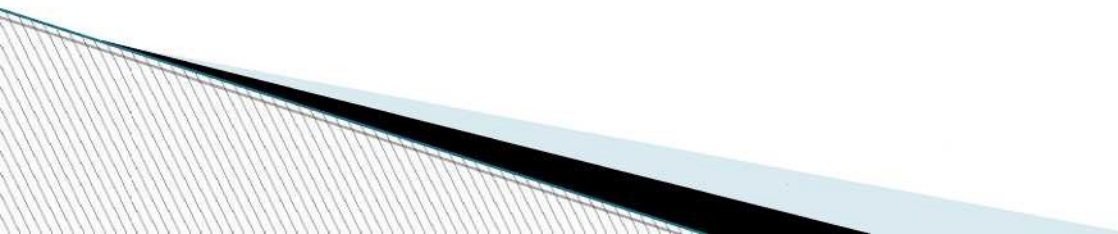
El API JDBC proporciona una interfaz de programación para acceso a datos de Base de Datos Relacionales desde el lenguaje de programación Java a través de dos paquetes (packages):

- ▶ El paquete **java.sql** es el corazón de la API JDBC 2.0.
- ▶ El paquete **javax.sql** es la API de Extensión Estándar JDBC 2.0; y proporciona la funcionalidad de fuente de datos (objeto **DataSource**), y agrupación de conexiones (**connection pooling**).

Package JAVA.SQL

Es el núcleo central de JDBC, en el que se encuentran las clases, interfaces y excepciones.

Las mas importantes son:



Package JAVA.SQL

java.sql.DriverManager

java.sql.Connection

java.sql.Statement

Subtipos **java.sql.PreparedStatement**

java.sql.CallableStatement **java.sql.ResultSet**

java.sql.SQLException

java.sql.ResultSetMetaData j

java.sql.DatabaseMetaData

Package JAVA.SQL

- ▶ **java.sql.DriverManager:** Es la clase gestora de los drivers. Se encarga de cargar y seleccionar el driver adecuado para realizar la conexión con una base de datos determinada.
- ▶ **java.sql.Connection:** Representa una conexión con una base de datos.
- ▶ **java.sql.Statement:** Actúa como un contenedor para ejecutar sentencias SQL sobre una base de datos. Este interfaz tiene otros dos subtipos:
 - java.sql.PreparedStatement**
 - **java.sql.CallableStatement**

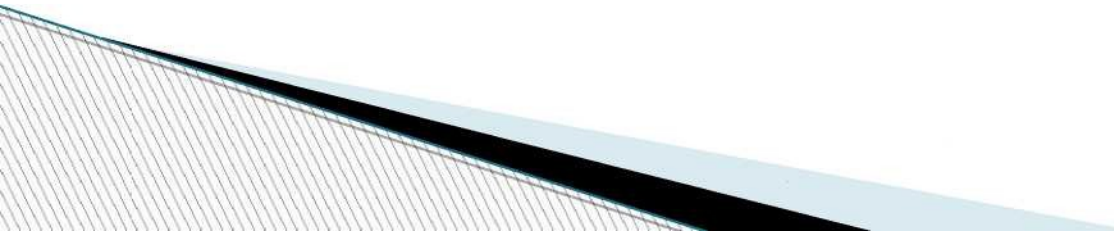
Subtipos de `java.sql.Statement`:

- `java.sql.PreparedStatement` para la ejecución de sentencias SQL precompiladas a las que se les pueden pasar parámetros de entrada;
- `java.sql.CallableStatement` que permite ejecutar procedimientos almacenados de una base de datos.

Package JAVA.SQL

- ▶ **java.sql.ResultSet:** controla el acceso a los resultados de la ejecución de una consulta, es decir, de un objeto Statement. Permite también la modificación de estos resultados.

Package JAVA.SQL

- ▶ **java.sql.SQLException:** para tratar las excepciones que se produzcan al manipular la base de datos, ya sea durante el proceso de conexión, desconexión u obtención y modificación de los datos.
 - ▶ **java.sql.ResultSetMetaData:** este interfaz ofrece información detallada relativa a un objeto ResultSet determinado.
 - ▶ **java.sql.DatabaseMetaData:** ofrece información detallada sobre la base de datos a la que nos encontramos conectados.
- 

Acceso a BBDD mediante JDBC

Hasta ahora, hemos visto las diferentes partes del **API JDBC**.

- ▶ Veamos como usar este API JDBC en la práctica usando el driver adecuado de Base de Datos.
- ▶ Los drivers nos permiten conectarnos con una base de datos determinada y lanzar consultas para manipularla.
- ▶ Veremos el proceso de forma específica para las operaciones de manipulación de datos (**insert** / **delete** / **update**) y para las de lectura (**select**).

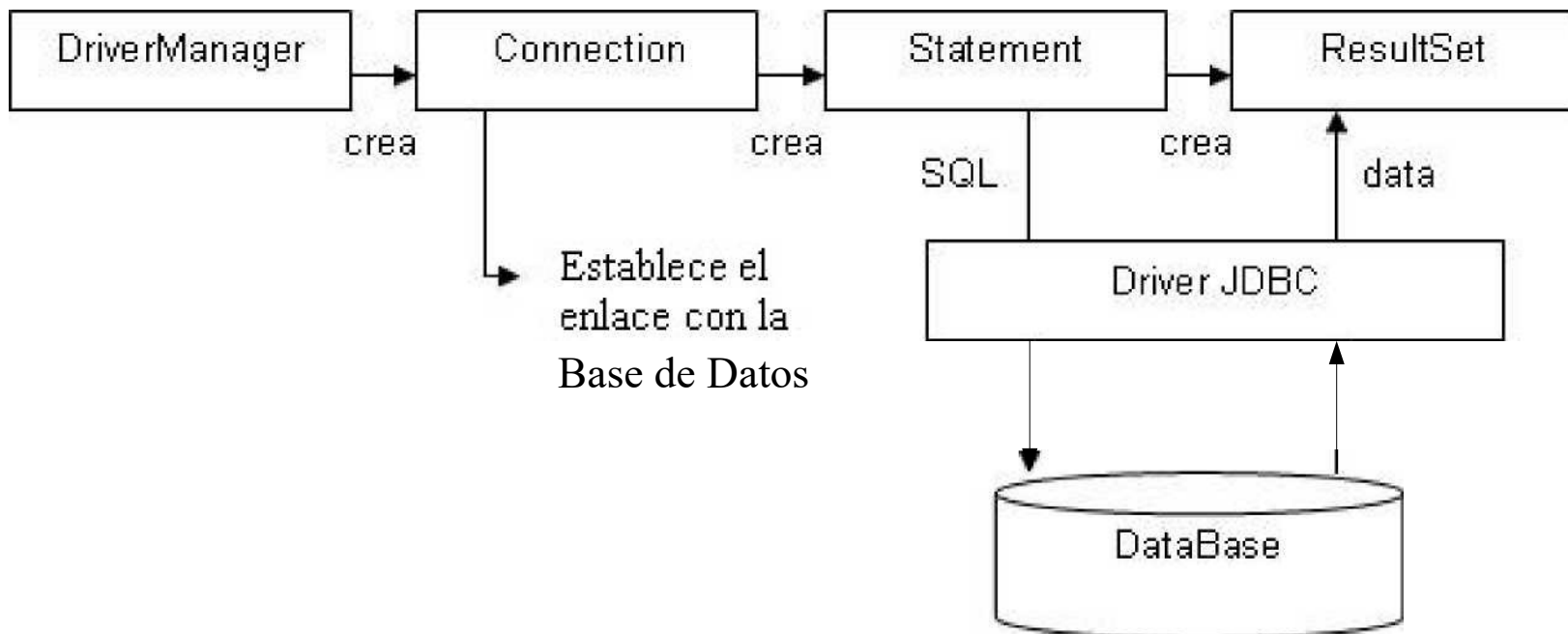
Proceso genérico

- 1. Establecer la conexión.** Permite hacer uso del driver elegido para acceder a la Base de Datos.
- 2. Ejecución SQL mediante Statements** (sentencias). Ejecutará las sentencias que queramos lanzar. Si son de tipo selección se devolverá un objeto de tipo `ResultSet` con las filas seleccionadas. Si son **de actualización, borrado o inserción no devolverá** ningún objeto de tipo `ResultSet`.
- 3. Obtener datos mediante ResultSet.** En el caso de que la sentencia sea de selección se habrá devuelto un `ResultSet` que deberemos recorrer accediendo a cada campo de cada fila devuelta.
- 4. Liberar recursos.** Es muy importante liberar los recursos en cuanto nos sea posible, dado que en las aplicaciones Web hay muchas peticiones simultáneas queriendo acceder a los mismos recursos.

Proceso genérico

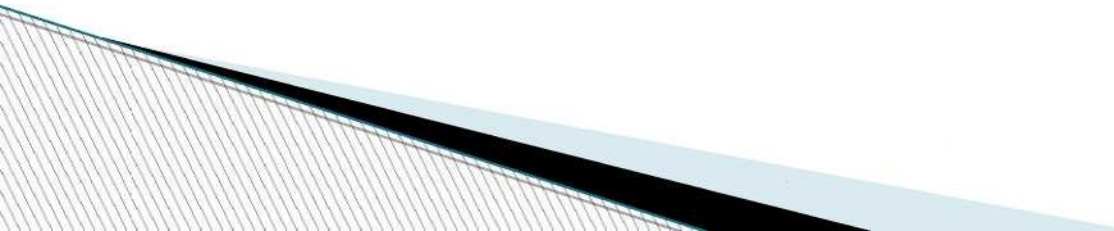
Para llevar a cabo este proceso las principales clases java que se usarán son **DriverManager**, **Connection**, **Statement** y **ResultSet**.

Si los representamos esquemáticamente, junto a su función principal dentro del proceso:



Proceso genérico

Cada instancia de cada clase se crea por medio de otra.

- > Un objeto de tipo **Connection** se crea mediante la clase **DriverManager**.
 - > Un objeto **Statement** se crea mediante un objeto de tipo **Connection**.
 - > Un objeto **ResultSet** se crea mediante un objeto **Statement**.
- 

Establecer la conexión: carga del driver

Establecer una conexión con el controlador (driver) de base de datos que queremos utilizar, implica dos pasos:

- 1 Cargar el driver
- 2 Hacer la conexión.

La carga del driver la realizaremos usando el método

Class.forName(driver).

Si, por ejemplo, queremos utilizar el driver **MySQL-Connector/J** se cargaría la siguiente línea de código:

```
Class.forName("com.mysql.jdbc.driver");
```

Establecer la conexión: **hacer la conexión**

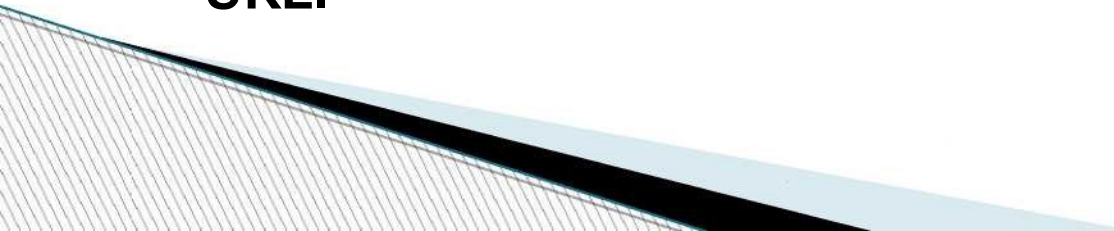
Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

El objetivo es conseguir un objeto del tipo

java.sql.Connection a través del método

DriverManager.getConnection(String url)

Cuando este método es invocado, el DriverManager tratará de usar el driver especificado anteriormente para conectar a la base de datos especificada en la **URL**.



Establecer la conexión: **hacer la conexión**

- ▶ Ese driver debería entender el “**subprotocolo**” que especifica la **URL**.
- ▶ Esa URL tendrá la siguiente estructura de nombrado de bases de datos: **jdbc:<subprotocol>:<subname>**

Donde:

jdbc: parte fija

Subprotocol: mecanismo de acceso a la base de datos

Subname: dependerá del subprotocolo.

JDBC recomienda seguir también la
convención de nombrado URL:

//hostname:port/subsubname.

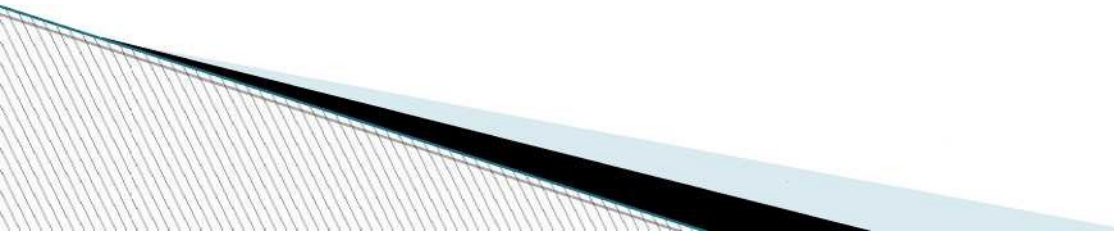
Establecer la conexión: **hacer la conexión**

Ejemplo: si estamos utilizando JDBC para acceder a una base de datos MySQL llamada "**CientesDB**" nuestro URL podría ser:

`jdbc:mysql://localhost:3306/CientesDB.`

De este modo **un ejemplo** para establecer la conexión es:

```
Class.forName("com.mysql.jdbc.Driver");  
String urlBD = "jdbc:mysql://localhost:3306/CientesDB ";  
Connection con;  
Con  =  
java.sql.DriverManager.getConnection(urlBD,"root","root");
```



Hacer la conexión

- ▶ Una vez establecida la conexión, esta se usa para **pasar sentencias SQL** a la base de datos.
- ▶ JDBC no pone ninguna restricción sobre los tipos de sentencias que pueden enviarse, esto proporciona gran flexibilidad, permitiendo el uso de sentencias específicas de la base de datos, siempre que ésta las soporte.


Ejecución SQL mediante Statements

JDBC suministra **tres clases** para el envío de sentencias SQL y tres métodos en la interfaz Connection para crear instancias de estas tres clases.

Son los siguientes:

Statement – creada por el método **createStatement**.

El **Objeto statement** se usa para enviar sentencias SQL simples, sin parámetros. El objeto Statement proporciona básicamente **3 métodos**, **execute()**, **executeUpdate()** y **executeQuery()**, que actúan como conductores de información con la base de datos.



Ejecución SQL mediante Statements

Diferencia entre cada uno de estos métodos ::

Método Uso recomendado

executeQuery() Se utiliza con sentencias **SELECT** y devuelve un **ResultSet**.

executeUpdate() Se utiliza con sentencias **INSERT**, **UPDATE** y **DELETE**, o bien, con sentencias DDL SQL.

execute() Utilizado para cualquier sentencia DDL, DML o comando específico de la base de datos.

Ejecución SQL mediante Statements

- > PreparedStatement – Hereda de Statement. Creada por el método **prepareStatement**.

Un **Objeto PreparedStatement** se usa para ejecutar sentencias SQL que toman uno o más parámetros como argumentos de entrada (parámetros IN).

Estos parámetros se especifican mediante un signo de interrogación (?) en la sentencia. Se denominan sentencias SQL precompiladas.

Ejecución SQL mediante Statements

Este objeto tiene una serie de métodos (**setXXX**) que fijan los valores de los parámetros **IN**, los cuales son enviados a la base de datos, cuando se procesa la sentencia SQL.

Es más eficiente y rápida que su antecesor.



Ejemplo para lanzar una sentencia es:

```
PreparedStatement stmt;
```

```
stmt = con.prepareStatement( "INSERT INTO cliente VALUES (?, ?, ?, ?, ?) );
```

```
    stmt.setInt(1, "143765987");
```

```
    stmt.setString(2, "Benito");
```

```
    stmt.setString(3, "Perez");
```

```
    stmt.setString(4, "Garrido");
```

```
    stmt.setFloat(5, 1000);
```

```
    stmt.executeUpdate();
```



Ejecución SQL mediante Statements

- > **CallableStatement**: creado por el método **prepareCall**. Los Objetos **CallableStatement** se usan para ejecutar **procedimientos almacenados SQL**.

Un **Objeto CallableStatement** hereda métodos para el manejo de los parámetros IN de **PreparedStatement**, y añade métodos para el manejo de los parámetros OUT e INOUT.

Ejecución SQL mediante Statements

Clases	Métodos
--------	---------

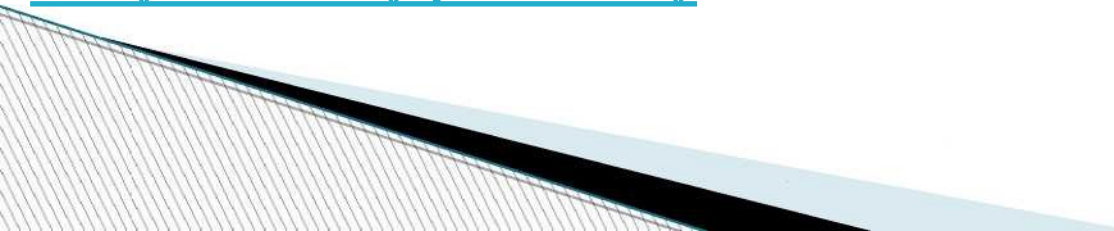
Statement	<code>execute()</code> <code>executeUpdate()</code> <code>executeQuery()</code>
-----------	---

Obtener datos mediante ResultSet

En un **objeto ResultSet** se encuentran los **resultados** de la ejecución de una sentencia SQL.

Un objeto ResultSet contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece el acceso a los datos de las filas a través de una serie de métodos **getXXX()** que permiten acceder a las columnas de la fila actual.

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



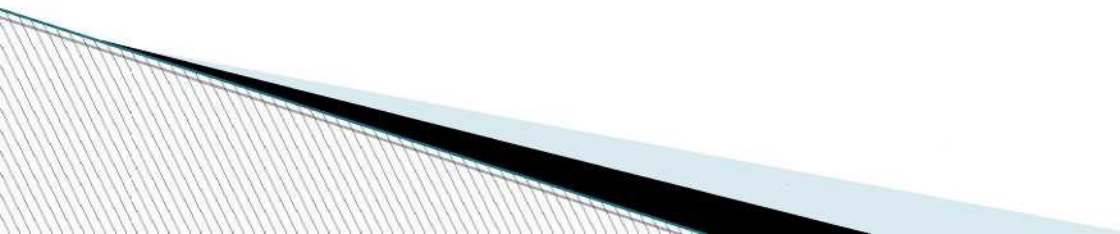
Obtener datos mediante ResultSet

- 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 lanzado. Inicialmente está posicionado antes de la primera fila.
De esta forma, la primera llamada a `next()` situará el cursor en la primera fila, pasando a ser la fila actual.

Obtener datos mediante ResultSet

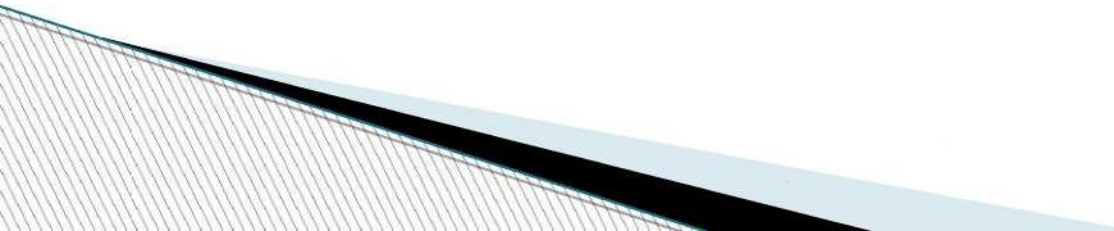
Las filas del ResultSet son devueltas de arriba abajo, según se va desplazando el cursor, con las sucesivas llamadas al método `next()`.

Un cursor es valido hasta que el objeto ResultSet o su objeto padre Statement, es cerrado.



Un ejemplo para recorrer un ResultSet es:

```
ResultSet rs = stmt.executeQuery ("SELECT * FROM cliente");  
  
while (rs.next()) {  
  
    String d = rs.getString("DNI ");  
    String n = rs. getString ("Nombre");  
    System.out.println(d + " " + n);  
  
}
```



Liberar recursos

En la programación Web hay muchos accesos simultáneos a un mismo recurso: la base de datos.

Es por ello que tenemos que liberar las conexiones tan pronto nos sea posible. A pesar de que el cierre de una conexión provoca el cierre de todos sus Statements asociados y el cierre de un Statement provoca el cierre de todos sus ResultSet asociados se aconseja realizar el cierre ordenado nosotros mismos.

Esto es, por ejemplo:



Liberar recursos

Try {

if (rs != null) rs.close(); //Cerramos el resulset

if (stmt != null) stmt.close(); //Cerramos el Statement

if (con!= null) con.close(); //Cerramos la conexión

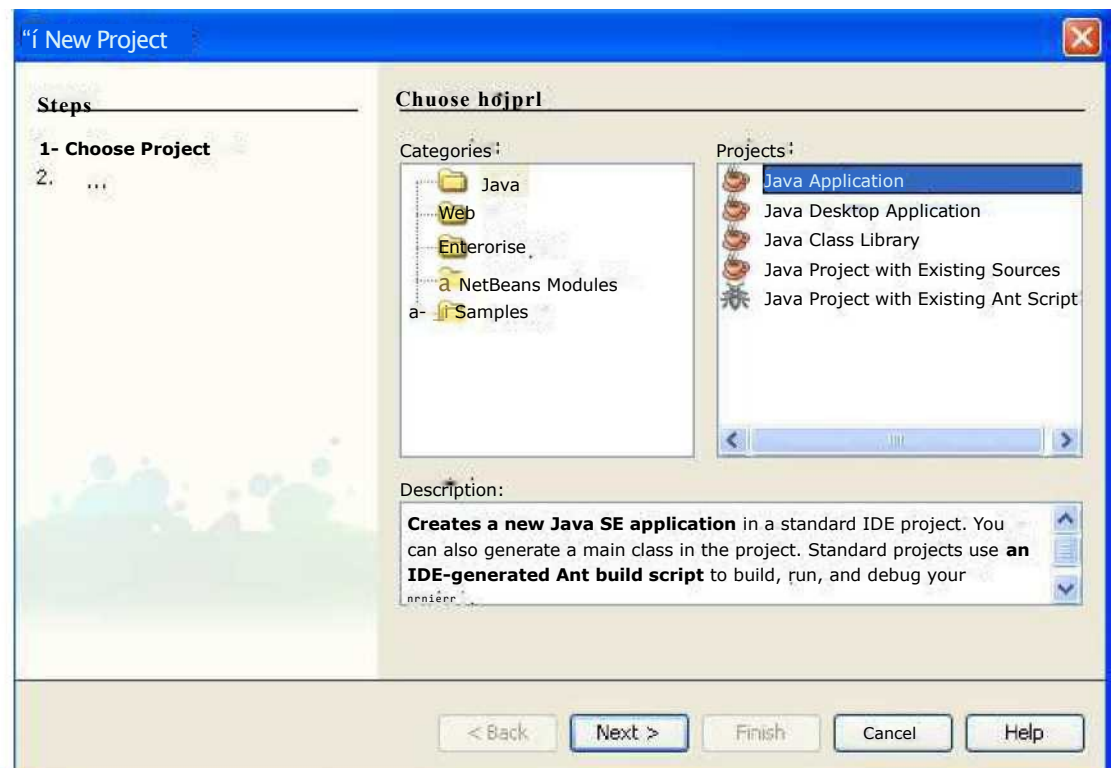
}

catch (SQLException ex) { ex.printStackTrace(); }

Ejemplo desde Net Beans

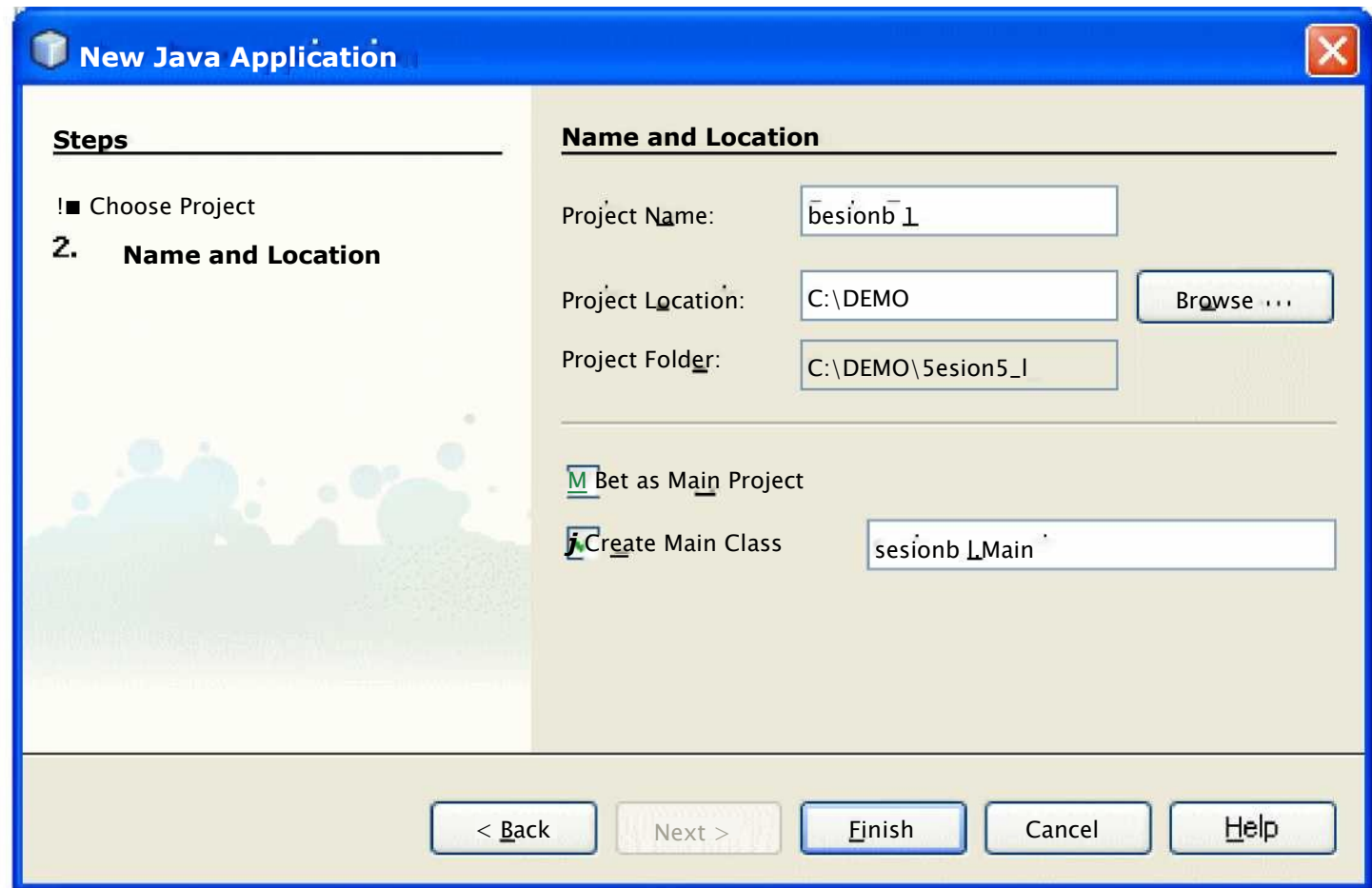
Para las pruebas sobre la base de datos usaremos un aplicación que muestre su salida por consola. Para ello :

1. Crearemos un proyecto de tipo Java / Java Application



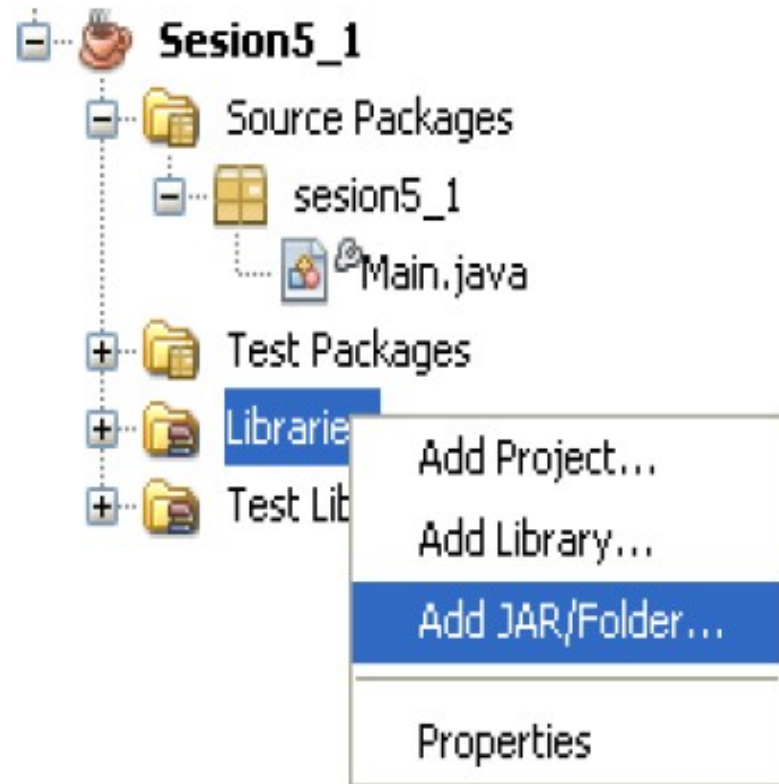
Ejemplo desde NetBeans

2. Llamaremos al proyecto **tema9** y dejaremos que cree por nosotros la clase principal (Main)

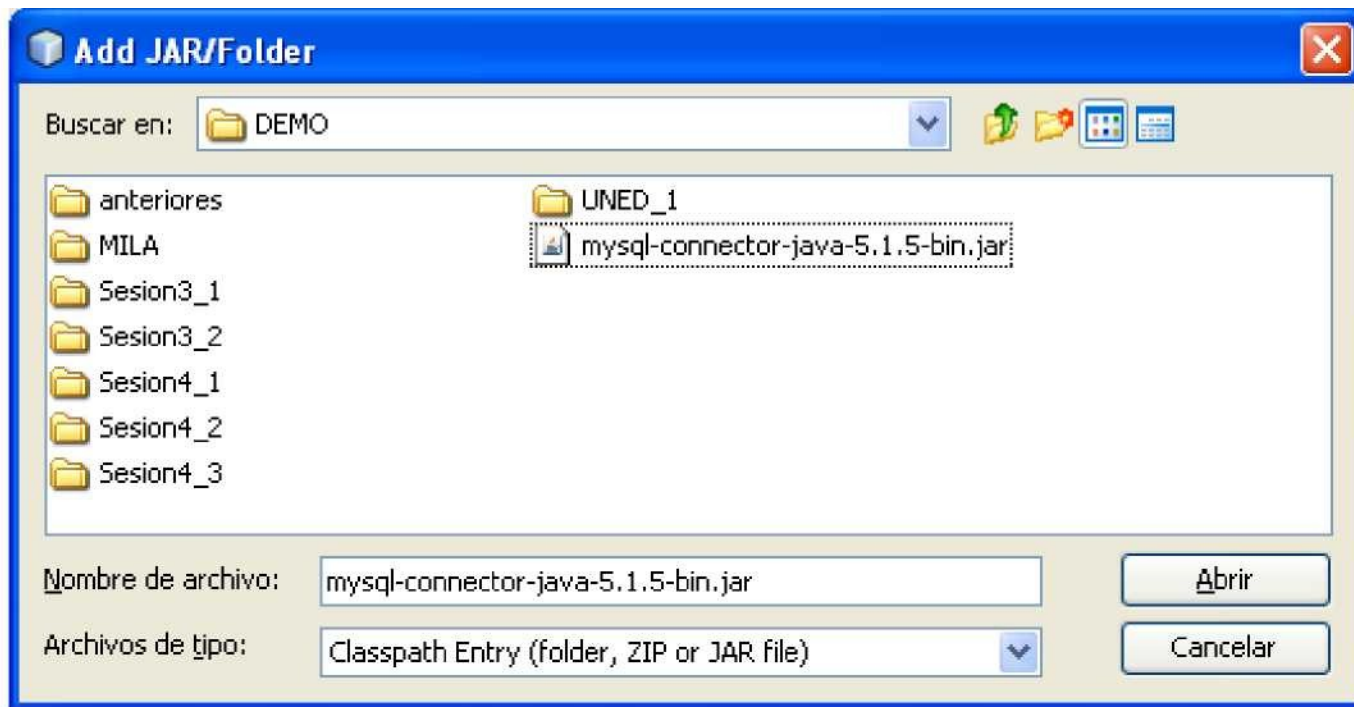


Ejemplo desde NetBeans

3. Añadiremos la biblioteca **mysql-connector** al proyecto mediante la opción Add Jar del menú contextual de biblioteca

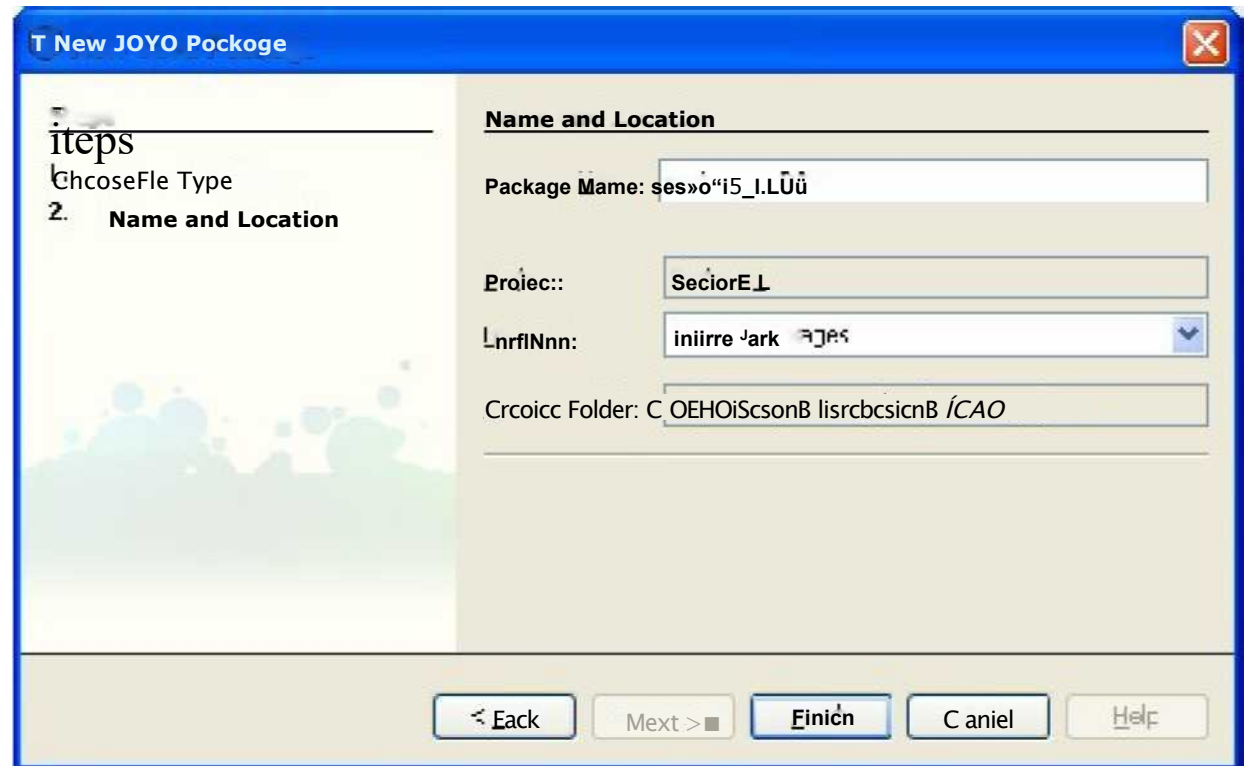


Previamente habremos descargado de <http://www.mysql.com/products/connector/j/> el fichero comprimido que al desempaquetar veremos que contiene el jar **mysql-connector-java-5.1.22-bin**. Que corresponde al driver de MySQL



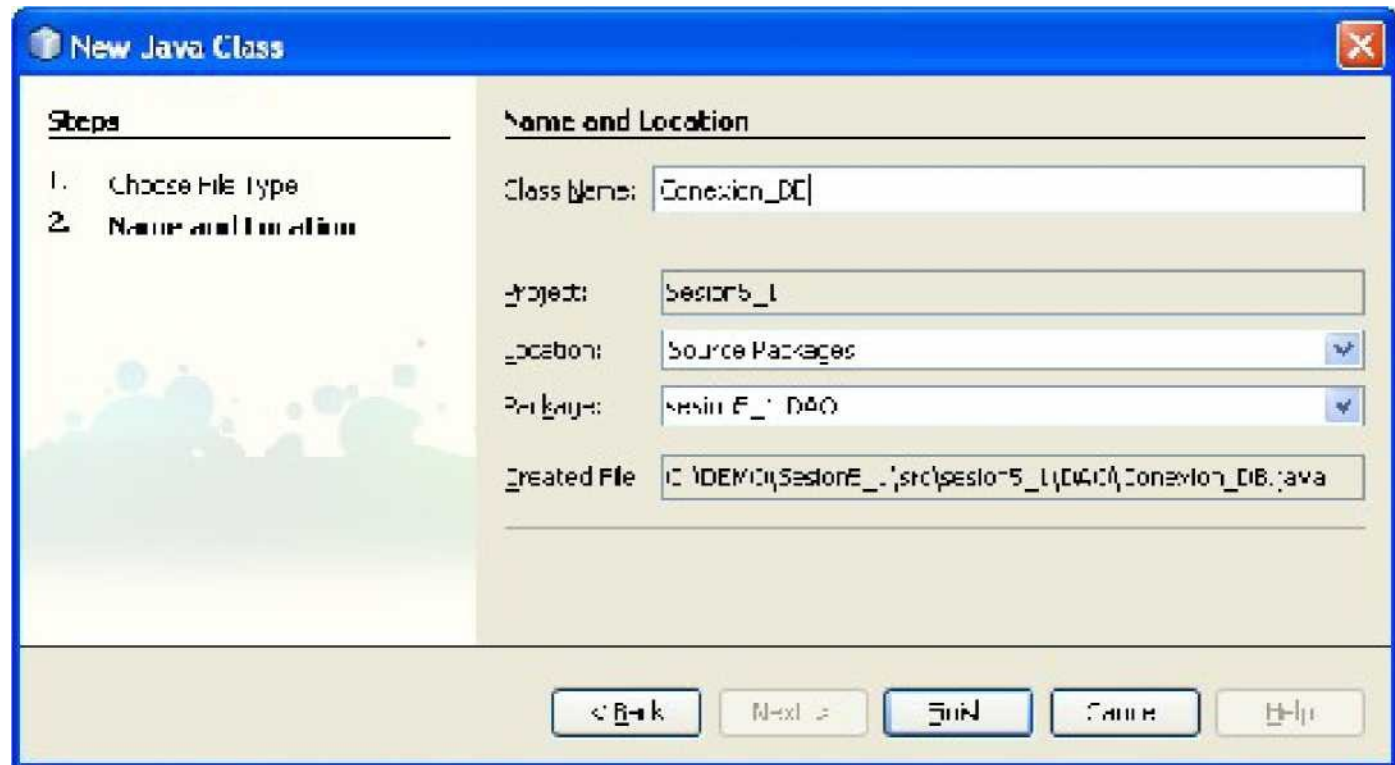
Ejemplo desde NetBeans

A continuación crearemos una clase llamada **Conexion_DB**, en la cual añadiremos unos métodos para abrir y cerrar la conexión a la Base de Datos Ciclismo ya creada. Esta clase la crearemos **en un package llamado DAO** (data access object). Esto es:



Ejemplo desde NetBeans

En este package creamos la clase **Conexion_DB**



Y esta clase tendrá el siguiente código

```
package DAO;

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

public class Conexion_DB {
    public Connection AbrirConexion() throws Exception
    {
        Connection con=null;
        try {
            Class.forName("com.mysql.jdbc.Driver");//cargar driver
            String urlObdc =("jdbc:mysql://localhost/ciclismo");
            con=java.sql.DriverManager.getConnection(urlObdc,"root","");
            return con;
        } catch (ClassNotFoundException | SQLException e) { //excepciones
            e.printStackTrace();
            throw new Exception("Ha sido imposible establecer la conexion"+e.getMessage());
        }
    }
}
```

Y la clase continua con el siguiente código:

```
public void cerrarConexion(Connection con) throws Exception
{
    try {
        if (con!=null) con.close();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new Exception("Ha sido imposible cerrar la conexión"+e.getMessage());
    }
}
```

Por otro lado tendremos una clase principal, cuyo código será

```
import java.sql.Connection;
import DAO.Conexion_DB;

public class Main {
    public static void main(String[] args) {
        try {
            Conexion_DB conexion_DB = new Conexion_DB();
            System.out.println("Abrir conexión");
            Connection con = conexion_DB.AbrirConexion();
            System.out.println("Conexión abierta");

            System.out.println("Cerrar Conexión");
            conexion_DB.cerrarConexion(con);
            System.out.println("Cerrando Conexión");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo desde NetBeans

Observaremos, una vez le demos a la ejecución de este código únicamente establece y cierra la conexión con Base de Datos. La salida será:

```
run:
```

```
Abrir conexión
```

```
Conexión abierta
```

```
Cerrar Conexión
```

```
Cerrando Conexión
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

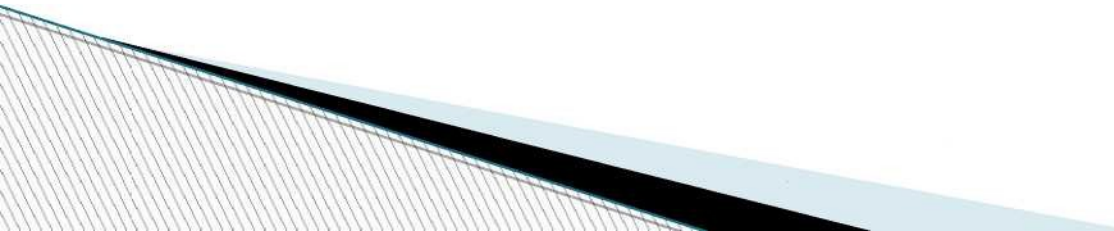
De este modo, tendremos una primera clase base (Conexion_DB) para el resto de los ejemplos.

Accesos básicos (Insert/ Update / Delete)

Veamos casos concretos del proceso genérico de acceso a datos descrito anteriormente.

Uno de los procesos habituales, a la hora de realizar mantenimiento sobre una tabla, es el de la actualización, borrado e inserción de datos.

Veamos como realizar el acceso. Dado que no se devuelven filas de la Base de Datos no utilizaremos la clase ResultSet.



Accesos básicos (Insert/ Update / Delete)

Actualización de datos. **UPDATE**

UPDATE tabla SET campo1 = valor1,
campo2=valor2

WHERE campo=condicion

Se actualizarán todas las filas que cumplan la condición (WHERE campo=condicion). Y la actualización consistirá en asignar nuevos valores a ciertas columnas.

(SET campo=valor).

Para poder ejecutar esta sentencia de UPDATE mediante JDBC, es necesario realizar los pasos explicados anteriormente

Establecer la conexión

// carga el driver de la bbdd.

```
String sDriver = "com.mysql.jdbc.Driver";
```

```
Class.forName(sDriver).newInstance();
```

// abrir conexión con la bbdd.

```
String sURL = "jdbc:mysql://localhost:3306/CientesDB";
```

```
con = DriverManager.getConnection(sURL,"root","password");
```

Ejecución SQL mediante Statements:

Una vez tenemos la conexión, preparamos la sentencia apoyándonos en la clase **PreparedStatement**.

PreparedStatement se utiliza cuando se va a realizar una sustitución de alguno de los valores de la condición, sino, se podría utilizar **Statement** directamente para ejecutar la sentencia.

PreparedStatement stmt;

```
stmt = con.prepareStatement ("UPDATE Usuarios SET Apel ='Lopez' WHERE  
DNI=?");
```


```
stmt.setString(1 ,"143765987");
```

Solo nos queda ejecutar la actualización. Para ello hay que ejecutar el método

ExecuteUpdate() del **PreparedStatement**

```
int retorno = stmt.executeUpdate();
```

Dicho método devolverá el número de filas que se han actualizado. Será un valor entero desde 0 al número de filas actualizadas.



Liberación de recursos:

```
try {  
    if (rs != null) rs.close(); //Cerramos el resulset  
  
    if (stmt != null) stmt.close(); //Cerramos el Statement  
  
    if (con != null) con.close(); //Cerramos la conexión  
} catch (SQLException ex) {  
  
    ex.printStackTrace();  
  
}
```

En el caso que en el proceso, desde la conexión a la ejecución de la sentencia, ocurriese un error, se produciría una excepción `SQLException` que capturaremos incluyendo todo el código entre la sentencia try-catch.

Accesos básicos (Select). Uso de ResultSets

Para poder ejecutar sentencias de tipo **SELECT** mediante JDBC, es necesario realizar los pasos explicados anteriormente en el proceso genérico:

establecer la conexión (exactamente igual que en la actualización)

Ejecución SQL mediante Statements

PreparedStatement stmt;

```
stmt = con.prepareStatement("SELECT * FROM cliente WHERE DNI=?");  
stmt.setString(1, "143765987");  
ResultSet rs = stmt.executeQuery();
```

Es idéntico al anterior **PERO** en vez de devolver un entero devuelve un ResultSet con las filas obtenidas de la base de Datos

Recorrer el Resultset

Utilizaremos el método **next()** del interfaz ResultSet para desplazarnos al registro siguiente dentro de un ResultSet.


El método **next()** devuelve un valor booleano, true si el registro siguiente existe y false si hemos llegado al final del objeto ResultSet, es decir, no hay más registros.

Los métodos **getXXX**(Idcolumna) del interfaz **ResultSet** ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del **ResultSet**.
Donde: **XXX** Es el tipo apropiado de la columna en la base de datos y se utilizan para recuperar el valor de cada columna.

Por ejemplo, si la primera columna de cada fila almacena un valor del tipo **VARCHAR** de SQL.

El método para recuperar un valor **VARCHAR** es **getString**.

Si la segunda columna de cada fila almacena un valor del tipo **FLOAT** de SQL, y el método para recuperar valores de ese tipo es **getFloat**.

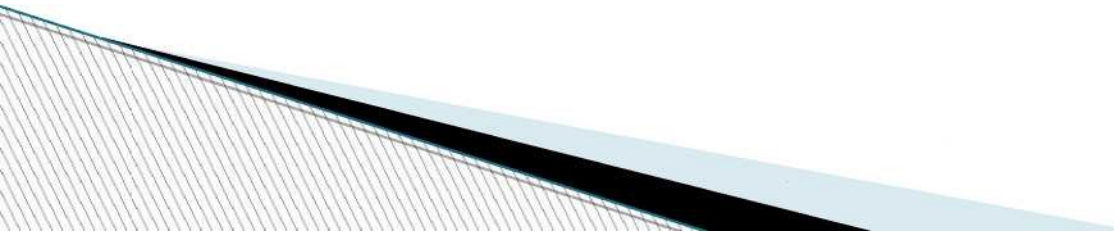


Idcolumnna

Para designar una columna podemos utilizar su nombre o bien su número de orden en la fila.

Por ejemplo si la segunda columna de un objeto **rs** de la clase **ResultSet** se llama "título" y almacena datos de tipo 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);  
String valor=rs.getString("titulo");
```



Por ejemplo, el siguiente código accede a los valores almacenados en la fila actual de **rs** e imprime una línea con el **DNI** seguido por tres espacios y el nombre.

Cada vez que se llama al método **next()**, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en **rs**.

```
while (rs.next()) {  
    String d = rs.getString("Dni");  
    String n = rs. getString ("nombre");  
    System.out.println(d + "   " + n);  
}
```



Liberar los recursos (igual que en la actualización)

```
try {  
    if (rs != null) rs.close(); //Cerramos el resulset  
  
    if (stmt != null) stmt.close(); //Cerramos el Statement  
    if (con != null) con.close(); //Cerramos la conexión  
  
} catch (SQLException ex) {  
    ex.printStackTrace();  
}
```

Al igual que antes, desde la conexión a la ejecución de la sentencia, puede producirse una excepción SQLException.

► Es por ello que no nos queda más remedio que capturar dicha excepción y ejecutar todo el código entre la sentencia try-catch.