Acceso a datos

2° DAM

<Qué se entiende por acceso a datos>

Según el IEEE (Institute of Electrical and Electronics Engineers) software es:

“El conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación”.

Una definición más conocida asumida por el área de la ingeniería del software es:

“El software es programas + datos”.

En ambas definiciones se ha resaltado la palabra datos.

<Qué se entiende por datos>

Los datos son, sencillamente, lo que necesitan los programas para realizar la misión para la que fueron programados.

Los datos pueden entenderse como persistentes o no persistentes.

- Persistentes son aquellos que el programa necesita que sean guardados en un sitio “seguro” para que pueda recuperar su estado anterior. Por ejemplo, en un teléfono móvil, la agenda con los números de teléfono.
- No persistentes son aquellos que no es necesario que el programa guarde entre ejecución y ejecución ya que solo son necesarios mientras la aplicación se está ejecutando. Por ejemplo, los teléfonos móviles llevan un registro de las aplicaciones que se están ejecutando en un momento dado: agenda, navegador, apps, etc.

Desde un punto de vista lógico, con independencia del sistema operativo, la persistencia se consigue con bases de datos y sistemas de ficheros.

Necesitaremos saber

- Generar y manejar ficheros XML y esquemas XML
- Manejar sistemas gestores de bases de datos relacionales y SQL
- Manejar Java, programación básica y entornos de desarrollo

1. Conector JDBC

ACCESO A DATOS

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

Contenido:

- Repaso de Java y MySQL
- API JDBC
- Driver
- Conexión
- Ejecución de sentencias
- Clase ResultSet
- SQL Injection
- Sentencias parametrizadas

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

Ejercicio:

- Crearemos un modelo de una temática distinta cada alumno/a, pero con aspectos en común: productos, clientes y pedidos. Por ejemplo: Ikea, Decathlon, Amazon, etc.
- Dicho modelo lo haremos:
  - Modelo de dominio en Java (POJOs con relaciones)
  - Esquema relacional en MySQL (DDL), con claves y restricciones

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

¿Qué es un POJO?

Un POJO es una clase Java muy simple que:

- No hereda de ninguna clase especial (más allá de Object).
- No implementa interfaces de frameworks obligatorios.
- No tiene anotaciones “extrañas” ni dependencias.
- Solo define atributos (campos), constructores, getters/setters, y a veces métodos auxiliares (toString(), equals(), hashCode()).

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

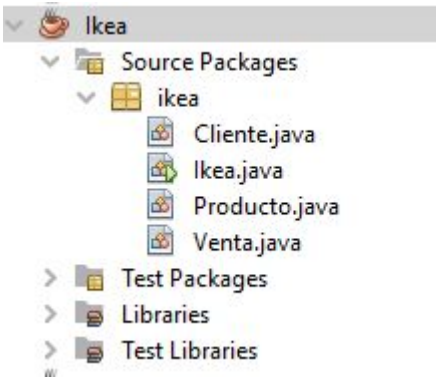
Ejemplo Ikea

- Propongamos una posible solución para uno de estos ejercicios basándonos en la temática de Ikea y dando por hecho, como en la vida real, que un cliente puede hacer un pedido de varios artículos.

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

Ejemplo Ikea

- Propondremos el uso de NetBeans para Java y de PHPMyAdmin para la BD.



1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

Ejemplo Ikea

- Una vez creados, haremos las pruebas, es decir:
  - Crear objetos en Java en una clase Test
  - Realizar INSERTS en la base de datos
- Si todo lo hemos hecho correctamente, podremos simular la misma venta en un sitio y en otro.

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

La mayoría de las aplicaciones, tras recoger una serie de datos y procesarlos, generan distintos resultados en forma de nuevos datos. Todos estos datos necesitan ser almacenados de forma permanente para su posterior uso. Sin la posibilidad de que una aplicación guarde la información necesaria sería imposible llevar a cabo muchas tareas.

Entre las técnicas de persistencia hemos visto que es posible guardar datos mediante ficheros. Otra alternativa consiste en utilizar los servicios de un sistema gestor de base de datos (SGBD) para que almacene y custodie toda la información necesaria de una aplicación.

1. CONEXIÓN A BASE DE DATOS: JDBC. Introducción

Ningún sistema informático, volviendo al ejemplo de IKEA, recoge una venta accediendo a la base de datos y haciendo un INSERT. Se usa una interfaz (página web) o una app o la misma aplicación de los cajeros al comprar en la tienda física.

En este tema, nuestra interfaz será la aplicación en JAVA.

1.1. API JDBC

INTRODUCCIÓN

JDBC son las siglas de Java DataBase Connectivity, una API (Interfaz de Programación de Aplicaciones) de Java que permite a los programas conectarse a bases de datos, enviar consultas SQL y manejar los resultados.

JDBC es el “puente” entre un programa en Java y una base de datos (por ejemplo MySQL).

¿Por qué existe JDBC?

- Un programa en Java no “sabe hablar SQL” directamente.
- La base de datos (MySQL, Oracle, PostgreSQL…) tampoco “sabe hablar Java”.
- JDBC actúa de traductor: tu aplicación envía instrucciones en SQL a la base de datos y recibe los resultados en objetos de Java.

1.1. API JDBC

Componentes principales de JDBC

Driver JDBC

- Es una librería que implementa la comunicación con un gestor concreto (MySQL, Oracle, etc.).
- Ejemplo: mysql-connector-j.jar para MySQL.

Connection

- Representa la conexión abierta con la base de datos.
- Se obtiene indicando URL, usuario y contraseña.

Statement / PreparedStatement

- Se utilizan para enviar sentencias SQL (SELECT, INSERT, UPDATE...).
- PreparedStatement es más seguro y eficiente que Statement.

ResultSet

- Es la tabla de resultados que devuelve un SELECT.
- Permite recorrer fila por fila y columna por columna.

1.1. API JDBC

RECORDEMOS

- API (Applications Programming Interface) es un conjunto de clases que trabajan de forma coordinada y conjunta proporcionando ciertos servicios, como por ejemplo interactuar con una base de datos, configurar una red o gestionar los movimientos de un robot.
- JDBC: Java DataBase Connectivity. Es una API que permite ejecutar sentencias SQL en un SGBD desde una aplicación Java, que funcionará como un cliente que accede a los servicios del servidor de base de datos.
- SGBD (sistema gestor de base de datos) es un conjunto de software que permite almacenar y gestionar información en una base de datos. También proporciona servicios de recuperación e integridad de datos, control de acceso de usuarios, copias de seguridad, etcétera.

1.1. API JDBC

La API de JDBC está compuesta por un sinfín de clases que trabajan de forma conjunta. Las principales son:

- DriverManager: permite manipular los distintos drivers. Con cada driver se puede acceder a un SGBD distinto.
- Connection: crea una conexión entre la aplicación y la base de datos.
- Statement: representa una sentencia SQL que ejecutará el servidor de base de datos.
- PreparedStatement: también representa una sentencia SQL, que permite configurar o parametrizar fácilmente valores en la consulta, como por ejemplo la edad de un alumno o su fecha de nacimiento en una condición.
- ResultSet: representa una tabla con el resultado que genera el SGBD tras ejecutar una sentencia de consulta de información (SELECT).

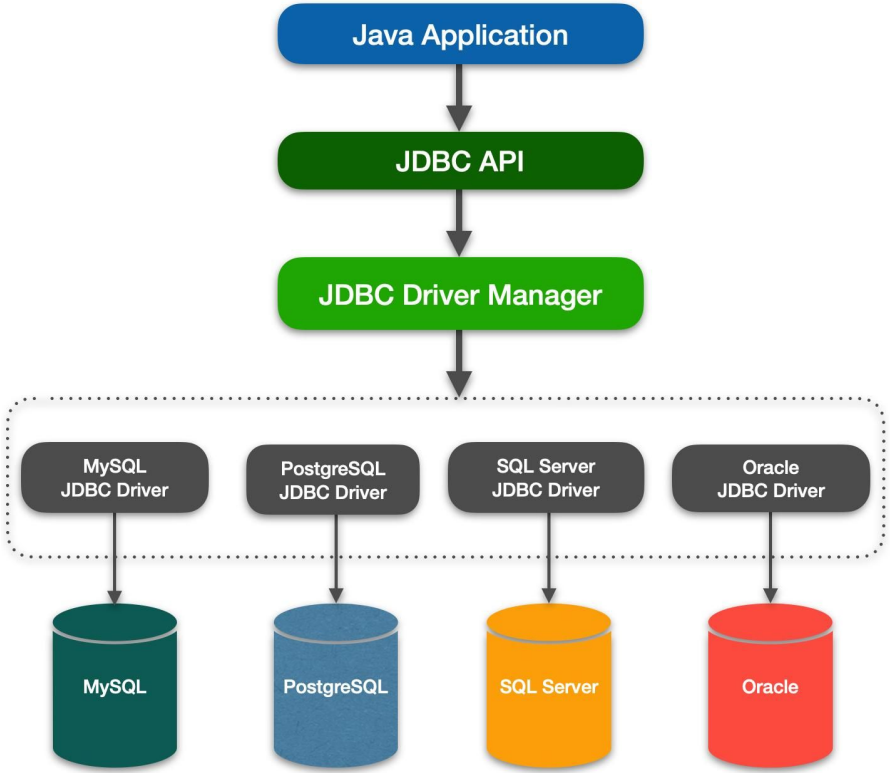
1.2. Driver

Cada fabricante de un SGBD, al desarrollar su producto, usa mecanismos propios que establecen una conexión con la base de datos y permiten acceder a sus servicios. Las clases que componen la API de JDBC no conocen estos detalles propios de cada producto.

Por lo tanto, ¿cómo es posible que las clases de la API de JDBC finalmente lleguen a establecer conexión con el servidor de base de datos? El mecanismo es muy sencillo: entre las clases que componen la API de JDBC existe una especial (que se denomina Driver) que será específica de cada SGBD. La clase Driver o driver de JDBC se añadirá a nuestro programa en función del SGBD que hayamos seleccionado.

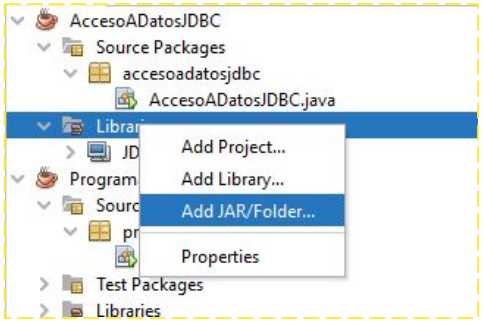
De esta forma, el driver permite que cualquier aplicación Java, a través de la API JDBC sea capaz de utilizar los servicios de cualquier SGBD. Y así se evita que cada aplicación Java tenga que conocer los detalles internos propios de cada SGBD.

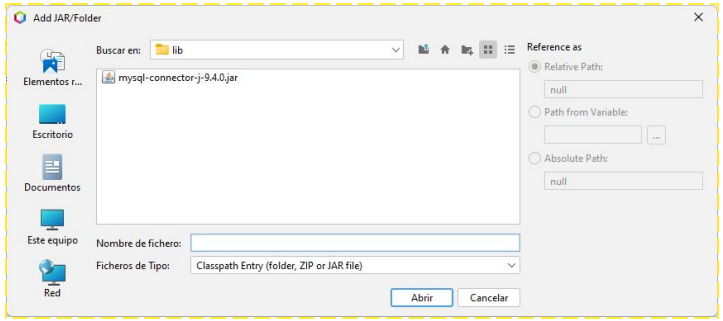
1.2. Driver


Java Application
JDBC API
JDBC Driver Manager
MySQL JDBC Driver
PostgreSQL JDBC Driver
SQL Server JDBC Driver
Oracle JDBC Driver
MySQL
PostgreSQL
SQL Server
Oracle

1.2. Driver

La manera de añadir un driver a nuestro proyecto consiste en insertar la biblioteca que contiene la clase correspondiente al driver. Si usamos Netbeans, hacemos click derecho en "Libraries" en nuestro proyecto > Add Jar/Folder.





https://dev.mysql.com/downloads/connector/j/ (Seleccionar Platform Independent y No thanks, download.

Ahora solo queda seleccionar el fichero **.jar** que incluye el driver adecuado. Dicho fichero tendrá que descargarse de la web del fabricante del SGBD. **Y es recomendable añadir el .jar en la carpeta del proyecto.**

1.3. Conexión

Antes de trabajar con la base de datos, hay que crear una conexión entre nuestra aplicación y el SGBD. Esta conexión funciona como un tubo que comunica ambas partes, permitiendo que las sentencias SQL viajen desde la aplicación al SGBD y los resultados de las consultas se muevan en sentido contrario. Mientras la necesitemos, la conexión deberá permanecer abierta; una vez que ya no sea útil, hay que cerrarla. En el caso de que una conexión no se cierre, quedará abierta consumiendo recursos del SGBD.

1.3. Conexión

Para crear una conexión disponemos del método estático DriverManager:

```
Connection getConnetion(String url, String usuario, String password);
```

El método devuelve un objeto de tipo Connection, que representa la conexión establecida entre nuestra aplicación y la base de datos.

Suponiendo que nuestra base de datos se llama "Instituto» y que disponemos del usuario «Pepe» con contraseña «12345», la manera de crear una conexión será:

```
Connection con;

String url = "jdbc:mysql://localhost/Instituto";

con = DriverMananager.getConnection (url, "Pepe", "12345");
```

1.3. Conexión

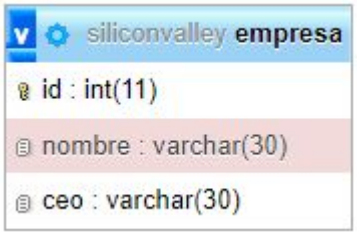
El método getConnection() puede producir algún error al crear la conexión como SQLException o SQLTimeoutException, por eso necesitamos añadir el código en un bloque try-catch.

Cuando ya no necesitemos una conexión, es necesario cerrarla mediante el método:

```
void close();
```

1.3. Conexión

Un ejemplo de conectar una base de datos MySQL llamada **siliconvalley** con un programa Java constaría de llamar den un atributo de tipo Connection y dos métodos, uno que realiza la conexión y otro que realiza la sentencia SQL en el SGBD.



1.3. Conexión

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

public class Gestor_conexion {

 Connection conn1 = null;
 public Gestor_conexion() {
      // ABRE UNA CONEXIÓN A UNA BASE DE DATOS QUE SE SUPONE MYSQL Y QUE TIENE LAS
TABLAS
    // Y LOS USUARIOS CREADOS SEGÚN ESTE EJEMPLO.

    try {
      //RECUERDA: PARA EJECUTAR ESTE CÓDIGO ES NECESARIO TENER mYSQL FUNCIONANDO Y
LAS TABLAS Y USUARIOS CREADOS
       String url1 = "jdbc:mysql://localhost:3306/siliconvalley";
       String user = "root";
       String password = "";
       conn1 = DriverManager.getConnection(url1, user, password);
       if (conn1 != null) {
          System.out.println("Conectado a siliconvalley");
       }
    } catch (SQLException ex) {
       System.out.println("ERROR:La dirección no es válida o el usuario y clave");
    }
 }

}
```

1.3. Conexión

```
public class Test {

  public static void main(String[] args) {
    //RECUERDA:
    // deben estar instalados los drivers MySQL en NetBeans para que funcione
    // Se debe crear la estructura de tablas indicadas para que funcione.
      // Se debe dar acceso a MySQL con usuario "root" y clave "" para que
funcione
    Gestor_conexion gestor = new Gestor_conexion();
 }

}
```

1.3. Conexión

```
public void cerrarConexion (){
  //SE CIERRA LA CONEXIÓN
   try {

   conn1.close();

 } catch (SQLException ex) {
      System.out.println("ERROR: al cerrar la conexión");
 }
```

1.4. Ejecución de consultas

Para ejecutar una sentencia SQL (SELECT, INSERT, UPDATE o DELETE) tendremos que utilizar un objeto de la clase Statement. Estos objetos, al igual que ocurre con la conexión, no se crean mediante el operador new, sino mediante métodos que se encargan de construir, configurar y devolver los objetos que necesitamos.
Lo primero que haremos será crear un objeto de tipo Statement:

```
Statement sentencia = con.createStatement();
```

El objeto con es de tipo Connection y representa la conexión creada entre la aplicación y el SGBD. Disponemos de dos métodos para ejecutar sentencias SQL.

- El primero está reservado a las consultas (SELECT) que devuelven datos con el resultado proporcionado por el SGBD.
- El segundo método está reservado para las sentencias INSERT, UPDATE o DELETE que no devuelven resultados.

1.4.1. Ejecución de consultas (SELECT)

El método executeQuery() de Statement ejecuta una consulta y devuelve el resultado de ésta mediante un objeto de tipo ResultSet. Su prototipo es:

```
ResultSet executeQuery(String sql)
```

Veamos cómo consultar una tabla Alumnos:

```
String sql = "SELECT * FROM Alumnos";
Statement sentencia = con.createStatement();
ResultSet rs = sentencia.executeQuery(sql);
```

Una vez que obtenemos los resultados de la consulta (englobados en un objeto de tipo ResultSet), queda trabajar con ellos. Esto se verá en un apartado posterior. Al igual que la mayoría de las clases que componen la API de JDBC, el método executeQuery() lanza, en el caso de que exista algún problema, dos excepciones:

- SQLException
- SQLTimeoutException

1.5. Clase ResultSet

Cuando se ejecuta una consulta (SELECT) obtenemos los resultados encapsulados en un objeto de tipo ResultSet, que representa una tabla con los datos que genera la consulta.
Por ejemplo, si necesitamos consultar el número, nombre y nota media de los alumnos que hayan nacido después del 1 de enero de 2009, ejecutaremos el código:

```
String sql = "SELECT num, nombre, media FROM Alumnos WHERE fnac > '2009-01-01'";
Statement sentencia = con.createStatement();
ResultSet rs = sentencia.executeQuery(sql);
```

El objeto rs contiene los datos obtenidos de esta consulta:

num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50

1.5. Clase ResultSet

La clase ResultSet tiene una forma de trabajar muy parecida a un iterador. Dispone de un cursor que apunta en cada momento a una única fila (llamada fila activa). Este sistema solo permite acceder a los datos de la fila activa, para trabajar con todos los datos del objeto ResultSet tendremos que ir moviendo el cursor de fila en fila.

Justo en el momento en el que se crea un objeto ResultSet el cursor se encuentra delante de la primera fila. Por lo tanto, hay que realizar un primer avance del cursor para colocarlo en la primera fila y comenzar a trabajar con los datos de esta.

num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50

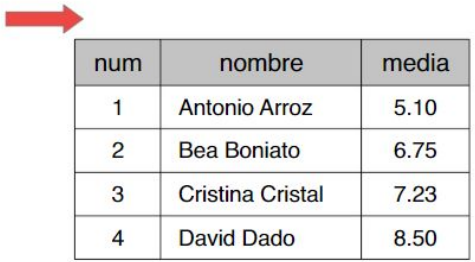
num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50

1.5. Clase ResultSet

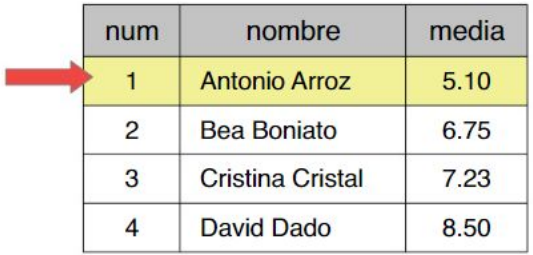
La forma de pasar de una fila a la siguiente se lleva a cabo mediante el método:

- boolean next () : mueve el cursor a la siguiente fila. Devuelve verdadero o falso si ha sido posible realizar el movimiento. Dicho en otras palabras, devuelve false cuando el cursor termina de recorrer todas las filas de datos y true en caso contrario.

El booleano que devuelve **next( )** es fundamental para conocer cuándo hemos terminado de procesar todas las filas de un ResultSet. Es habitual utilizar **rs.next ( )** como la condición de un bucle while.



num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50



num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50

1.5. Clase ResultSet

Existen métodos que mueven y posicionan el cursor a nuestro antojo a través de las filas de un ResultSet. Estos métodos sólo son aplicables en objetos ResultSet cuyo tipo permite el desplazamiento del cursor de dos maneras (TYPE_ SCROLL_INSENSITIVE y TYPE_SCROLL_SENSITIVE) que veremos más adelante.

Todos los métodos que mueven el cursor devuelven un booleano que indica si ha sido posible desplazar el cursor (true) o, por el contrario, el movimiento del cursor no puede realizarse (en este caso devuelven false). Un movimiento del cursor no podrá llevarse a cabo si la fila especificada no existe o el tipo de cursor no permite ese tipo de movimiento.

Alguno de estos métodos son:

- boolean next ( ) : avanza el cursor y activa la siguiente fila.
- boolean previous ( ) : retrocede el cursor y activa la fila anterior.
- boolean first () : coloca el cursor en la primera fila, activándola.
- boolean last ( ) : coloca el cursor en la última fila, activándola.
- boolean absolute (int numeroFila) : mueve el cursor a la enésima fila del ResultSet. El parámetro numeroFila especifica el número de fila, que se enumera desde el 1. También el -1 es la última, -2 penúltima, etc.

1.5. Clase ResultSet

La posibilidad de mover el cursor puede provocar que perdamos la noción de dónde se encuentra. Por ello, existe un conjunto de métodos que permiten sondear si el cursor se halla en algunas posiciones establecidas. Estos métodos preguntan: ¿está el cursor en cierta posición? Todos devuelven un booleano para indicar sí o no.

- boolean isBeforeFirst ( ) : especifica si el cursor se encuentra delante de la primera fila (es la posición inicial cuando se crea un objeto ResultSet).
- boolean isAfterLast (indica si el cursor está colocado justo detrás de la última fila. Esta posición es la que se alcanza tras recorrer el ResultSet mediante while (rs.next ()).
- boolean isFirst (): devuelve true si el cursor está apuntando a la primera fila.
- boolean isLast ( ) : devuelve true si el cursor apunta a la última fila.

1.5. Clase ResultSet

Ahora podemos extraer los datos de la fila activa. Para ello disponemos de los métodos:

- getString (String nombreCampo) : devuelve el valor del campo como una cadena.
- int getInt (String nombreCampo) : devuelve el valor del campo como un entero.
- Double getDouble (String nombreCampo): devuelve el valor del campo como un real.
- Date getDate (String nombreCampo) : devuelve el valor del campo como una fecha.

Ahora podemos extraer los datos de la fila activa. Por ejemplo:

```
//mostrará "Número de alumno: 1"
System.out.println("Número del alumno: " + rs.getInt("num"));
//mostrará "Nombre: Antonio Arroz"
System.out.println("Nombre: " + rs.getString("nombre"));
Double nota = rs.getDouble("media"); //la variable nota será igual a 5.10
```

1.5. Clase ResultSet

Podemos extraer los datos de las filas y ejecutar repetidas veces rs.next( ), pero llegará un momento en el que hayamos visitado todas las filas del ResultSet, y rs.next ( ) devolverá falso, indicando que no existen más filas que activar; en ese momento el cursor se coloca justo detrás de la última fila.

```
ResultSet rs = sentencia.executeQuery(sql);
while(rs.next()){
  variable1 = rs.getString("nombre");
  variable2 = rs.getInt("num");
  ...
}
```

num	nombre	media
1	Antonio Arroz	5.10
2	Bea Boniato	6.75
3	Cristina Cristal	7.23
4	David Dado	8.50

1.5.1. Tipos de ResultSet

El objeto ResultSet que hemos utilizado puede denominarse por defecto, ya que es de solo lectura (se pueden extraer los datos del objeto ResultSet, pero no modificarlos) y su cursor siempre avanza hacia delante. Es posible utilizar otros tipos de ResultSet que permiten la modificación de sus datos y poder mover el cursor hacia delante o atrás, así como posicionarlo en cualquier fila.

Para obtener otros tipos de ResultSet es necesario crear los objetos Statement mediante el método sobrecargado de Connection:

```
Statement createStatement(int tipoResultSet, int concurrencia)
```

->

1.5.1. Tipos de ResultSet

El parámetro tipoResultSet admite cualquiera de las constantes:

- ResultSet.TYPE_FORWARD_ONLY: indica que el cursor solo podrá moverse hacia delante.
- ResultSet.TYPE_SCROLL_INSENSITIVE: el cursor podrá desplazarse hacia delante o atrás. Los datos contenidos en el ResultSet son una copia de los datos almacenados en la base de datos, por lo tanto, el ResultSet no es sensible a los posibles cambios que se produzcan en la base de datos.
- ResultSet.TYPE_SCROLL_SENSITIVE: el cursor podrá desplazarse hacia delante o atrás. El ResultSet es sensible a los cambios que ocurran en la base de datos. Cualquier modificación de los datos que contiene el ResultSet en la base de datos produce una modificación en los datos del ResultSet. En este caso el ResultSet actúa de una forma similar a una vista.

El segundo parámetro, concurrencia, indica la posibilidad de modificar los datos contenidos en el objeto ResultSet:

- ResultSet.CONCUR_READ_ONLY: los datos contenidos en el ResultSet serán de solo lectura.
- ResultSet.CONCUR_UPDATABLE: desde la aplicación es posible modificar los datos contenidos en el objeto ResultSet.

Cuando usamos el método createStatement(), sin parámetros, los objetos ResultSet obtenidos por defecto serán de tipo TYPE_FORWARD_ONLY y CONCUR_READ_ONLY.

1.5.1. Tipos de ResultSet

ResultSet.CONCUR_UPDATABLE **permite modificar tanto el objeto** ResultSet **como, indirectamente, las filas de la base de datos**.

1. El ResultSet actúa como una “vista” sobre la tabla
   Cuando creamos un ResultSet con concurrencia updatable, podemos modificar los valores de sus filas directamente desde Java usando métodos como:

```
rs.updateString("nombre", "NuevoNombre");
rs.updateDouble("salario", 3000);
rs.updateRow();   // <- Aquí se actualiza en la base de datos
```

   Es decir:

- Los métodos updateXXX() modifican el contenido del objeto ResultSet (en memoria).
- Pero no actualizan la base de datos hasta que llamas a updateRow().

1.5.1. Tipos de ResultSet

2. Ejemplo:

```
Statement st = conexion.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);

ResultSet rs = st.executeQuery("SELECT * FROM empleados");

while (rs.next()) {
  if (rs.getString("nombre").equals("Luis")) {
     rs.updateDouble("salario", 2500);        // Cambia el valor en el ResultSet
     rs.updateRow();                          // Aplica el cambio en la base de datos
  }
}
```

- updateDouble() cambia solo en memoria.
- updateRow() envía el cambio al SGBD, ejecutando internamente un UPDATE.

1.5.1. Tipos de ResultSet

3. Añadir o eliminar filas también es posible.
Con CONCUR_UPDATABLE se pueden incluso insertar o borrar filas desde el propio ResultSet:

```
rs.moveToInsertRow();
rs.updateString("nombre", "Ana");
rs.updateDouble("salario", 2000);
rs.insertRow();                          // Inserta la nueva fila en la base de datos
```

Y para borrar:

```
rs.deleteRow();          // Elimina la fila actual en la base de datos
```

1.5.1. Tipos de ResultSet

Acción	Método JDBC	Qué modifica inmediatamente	Cuándo se refleja en la base de datos
Modificar un valor existente	updateXXX()	Solo el objeto ResultSet (en memoria)	No se actualiza todavía
Confirmar el cambio	updateRow()	Actualiza la fila actual del ResultSet	Se actualiza en la base de datos
Insertar una nueva fila	insertRow()	Añade la fila al ResultSet	Se inserta en la base de datos
Eliminar una fila	deleteRow()	Marca la fila actual como eliminada	Se borra en la base de datos

1.5.2. JDBC y programación orientada a objetos / ORM

El ResultSet no es un objeto de nuestro modelo, sino un mecanismo para leer o modificar datos “crudos” directamente desde la base de datos.

Cuando trabajamos con JDBC:

- El ResultSet actúa como una tabla temporal en memoria, donde cada fila representa un registro de la BD.
- Lo correcto en un programa bien estructurado es leer esos datos y transformarlos en objetos Java (por ejemplo, Empleado, Producto, etc.), que tengan sus atributos y constructores.
- El paso de “traducir” las filas a objetos se llama mapeo objeto-relacional (ORM).

1.5.2. JDBC y programación orientada a objetos / ORM

Enfoque	Qué se manipula	Cómo se insertan datos
JDBC directo (sin ORM)	Filas y columnas de la BD	Mediante sentencias SQL (INSERT, UPDATE, etc.) o métodos del ResultSet (no recomendado en diseño orientado a objetos)
Con ORM o diseño por clases	Objetos de tipo Empleado, Producto, etc.	Creando objetos con constructores y luego guardándolos con código que traduce a SQL (por ejemplo, Hibernate, o una capa DAO propia)

1.6. SQL Injection

SQL Injection es una técnica de hacking muy conocida, sencilla y fácil de utilizar. Consiste en inyectar código SQL en una consulta mediante la entrada de datos. Veamos un ejemplo: suponemos que tenemos una aplicación que, en un momento dado, nos pide el nombre de un alumno para eliminarlo.
El código que utilizaremos tendrá el aspecto:

```
String nombre;
nombre = new Scanner(System.in).nextLine( ) ; //pedimos el nombre al usuario nombre
String sql = "DELETE FROM Alumnos WHERE nombre = ' "+ nombre + " ' ";
```

Si el usuario introduce como nombre «Federico García», la consulta que se ejecutará en el servidor de base de datos, será:

```
DELETE FROM Alumnos
WHERE nombre = 'Federico García'
```

lo que provocaría que se eliminara a dicho alumno.

1.6. SQL Injection

Pero si el usuario tiene conocimientos de SQL e introduce de forma malintencionada código SQL como si fuera el nombre del alumno, este código se añadirá a la sentencia. El hacker lo que busca es eliminar completamente nuestra base de datos. Para ello, cuando se le solicita el nombre del alumno, puede teclear:

```
xxx' OR '1' = '1
```

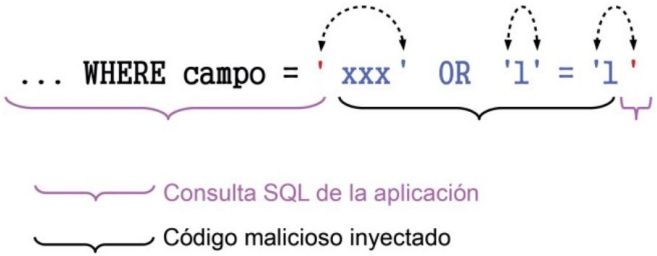
La cadena introducida por el usuario se complementa con la nuestra, resultando:

```
DELETE FROM Alumnos
WHERE nombre = 'xxx' OR '1' = '1'
```

Como la cadena '1' es siempre igual a '1', esto provocará que se eliminen todos los registros de la tabla Alumnos.

1.6. SQL Injection

La extraña posición de las comillas utilizadas en el código inyectado tiene como finalidad completar las comillas de la consulta de la aplicación y utilizar el código propio. En la condición sería más cómodo usar 1 = 1, pero el hecho de que exista una comilla final en la consulta de la aplicación es lo que fuerza a la comparación '1' = ‘1’.



1.6. SQL Injection

EJERCICIO

Implementar un programa que solicite el nombre de un alumno y lo elimine. Aprovechar esta aplicación para practicar la técnica de SQL Injection.

1.7. Sentencias parametrizadas

Como se ha visto, para evitar SQL Injection, no es buena idea construir sentencias como concatenación de cadenas a partir de datos introducidos por el usuario/programador. En lugar de esto, usaremos sentencias parametrizadas, que son aquellas que incluyen unos marcadores o parámetros que se sustituyen por valores. Este mecanismo permite adaptar y reutilizar la misma consulta varias veces. En JDBC la interfaz PreparedStatement representa una consulta parametrizada.
Veamos el concepto de consulta parametrizada con un ejemplo: si deseo consultar a todos los alumnos de un curso concreto cuya nota media es superior a cierta nota de corte. La consulta parametrizada se formará:

```
SELECT * FROM Alumnos WHERE curso = ? AND media > ?
```

1.7. Sentencias parametrizadas

En la consulta se usa el símbolo de cierre de interrogación (?) para introducir un parámetro. Si me interesa que el curso sea 1A y la nota de corte sea 6,0, podemos asignar estos valores a los parámetros:

- El primer parámetro: ? = 1A.
- El segundo parámetro: ? = 6.0

De esta forma la consulta queda:

```
SELECT * FROM Alumnos WHERE curso = '1A' AND media > 6.0
```

Una ventaja de este tipo de consultas es que no es necesario prestar atención a las comillas, solo hay que asignar valores a los parámetros y es el propio JDBC lo que determina qué campos se entrecomillan.
En el caso de tener que reutilizar la consulta, es tan simple como asignar nuevos valores a los parámetros.

1.7. Sentencias parametrizadas

Veamos el código necesario para utilizar las consultas con parámetros:

```
//Consulta con parámetros. Cada parámetro se indica con ?
String sql = "SELECT nombre, media FROM Alumnos " +
        "WHERE curso = ? AND " +
        " media > ?";
//Creamos un objeto de tipo PreparedStatement:
PreparedStatement sentencia = con.preparedStatement(sql);
//asignamos los parámetros:
sentencia.setString(1, curso); //el primer ? corresponde al curso
sentencia.setDouble(2, notaCorte); //el segundo ? corresponde a la nota de corte
ResultSet rs = sentencia.executeQuery(); //ejecutamos la consulta
... //trabajamos con los datos restantes
```

1.7. Sentencias parametrizadas

Para asignar los parámetros disponemos de los métodos:

- *void* setString(*indice*, *valor*)
- *void* setInt(*indice*, *valor*)
- *void* setDouble(*indice*, *valor*)
- *void* setBoolean(*indice*, *valor*)
- *void* setDate(*indice*, *valor*)

En todos los métodos, el primer parámetro indica el índice del parámetro de la consulta (los parámetros se comienzan a contar desde 1). Y el segundo parámetro es el valor que asignar.

1.7. Sentencias parametrizadas

ACTIVIDAD

Escribir un programa que muestre todos los alumnos de un curso cuya nota es mayor que cierta nota de corte. Tanto el curso como la nota de corte serán introducidos por el usuario, usando para ello la clase Scanner.