TEMA 2.- MANEJO DE CONECTORES.

Objetivos

- El desfase objeto-relacional.
- Protocolos de acceso a bases de datos. Conectores.
- Establecimiento de conexiones.
- Ejecución de sentencias de descripción de datos.
- Ejecución de sentencias de modificación de datos.
- Ejecución de consultas.
- Utilización del resultado de una consulta.
- Ejecución de procedimientos almacenados en la base de datos.
- Gestión de transacciones.

Contenidos

1.1.- Bases de datos orientadas a objetos.

Las bases de datos orientadas a objetos Las bases de datos orientadas a objetos (BDOO) están, como su nombre indica especialmente diseñadas para trabajar con datos de tipo objeto mientras que las bases de datos tradicionales trabajan con campos y registros.

Los modelos Orientados a Objetos los datos manejados por la base de datos serán clases y objetos permitiendo almacenar y recuperar los datos de una forma eficiente y sencilla.

Una de las principales Bases de Datos Orientadas a Objetos disponibles en el mercado es db4o que es Open Source para Java y se distribuye bajo licencia GPL.

1.2.- Características de las Bases de Datos Orientadas a Objetos.

Las características de las Bases de Datos Orientadas a Objetos son:

- Se entienden las Bases de Datos OO como un sistema de modelado del mundo real. Cada entidad del mundo real será un objeto en la base de datos.
- Los objetos tienen un identificador único que los identifica y los diferencia de los demás objetos del mismo tipo.
- El resultado de las consultas son objetos por lo tanto la velocidad de procesamiento se aumenta.
- En las bases de datos orientadas cuando se trata de realizar consultas complejas resulta más adecuado una base de datos relacional accedida mediante SQL.

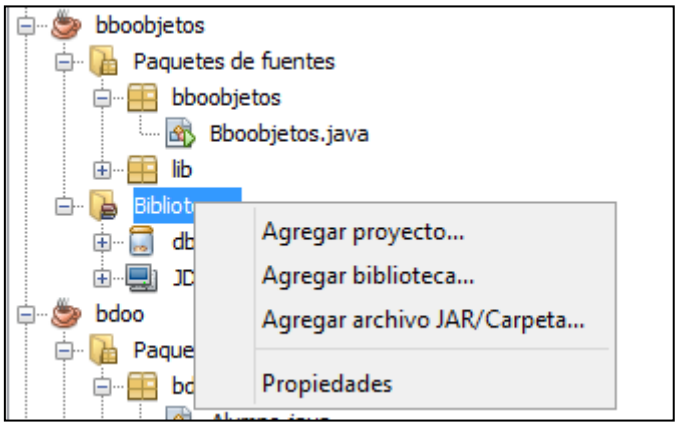
1.3.- Instalación del Gestor de Base de Datos Orientadas a Objetos.

El sistema gestor de base de datos Db4o es una verdadera base de datos de objetos. Muchas otras bases de datos de objetos lo que hacen es utilizar internamente mecanismos relacionales para almacenar los objetos lo que las hace que no sean una verdadera base de datos de

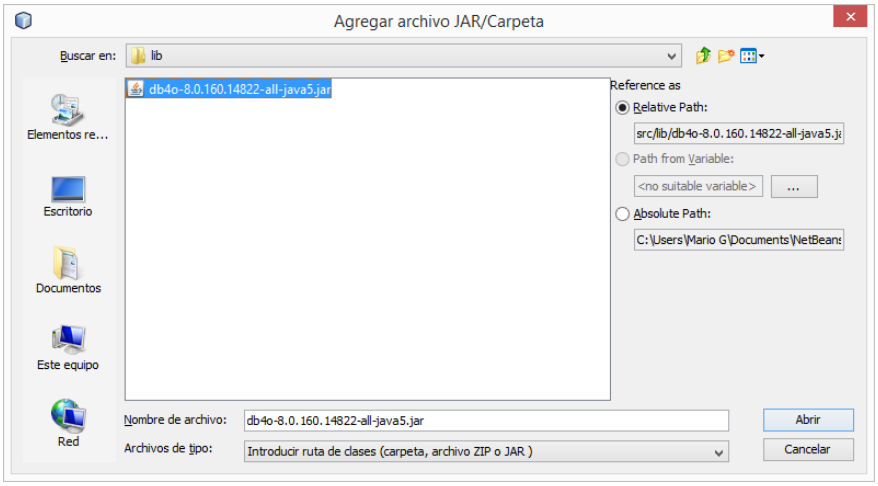
objetos (se suelen denominar bases de datos híbridas). El contenido de los objetos son almacenados en la base de datos.

Una de las ventajas de db4o es que es un simple jar distribuible con cualquier aplicación sin necesidad de tener que instalar nada. No necesita drivers de tipo JDBC o similar.

La instalación consistirá en copiar el fichero db4o-*.jar en la carpeta del proyecto de Netbeans. Después pulsaremos el botón derecho sobre la carpeta Bibliotecas del proyecto y seleccionaremos la opción Añadir JAR/Carpeta.



Por último seleccionamos el el fichero db4o-*.jar.



1.4.- Operaciones Básicas con la Base de Datos.

Antes de nada debemos crear una clase que va a ser la candidata a utilizar cuando interactuemos con la base de datos. Por ejemplo podemos crear una clase Alumno que almacene los alumnos de un colegio:

```
package bboobjetos;
public class Alumno {
    int  id;
    String nombre;
    String apellidos;
```

```
    double nota;
    String observaciones;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nobre) {
        this.nombre = nobre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public double getNota() {
        return nota;
    }

    public void setNota(double nota) {
        this.nota = nota;
    }

    public String getObservaciones() {
        return observaciones;
    }

    public void setObservaciones(String observaciones) {
        this.observaciones = observaciones;
    }

    public Alumno(int id, String nombre, String apellidos, double nota,
String observaciones) {
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.nota = nota;
        this.observaciones = observaciones;
    }

    @Override
```

```
    public String toString() {
        return "Alumno{" + "id=" + id + ", nombre=" + nombre + ", apellidos="
+ apellidos + ", nota=" + nota + ", observaciones=" + observaciones + '}';
    }
}
```

1.5.- Crear/Acceder a la base de datos.

Para conectar/acceder a la base de datos debemos crear una instancia de la clase ObjectContainer.

Ejemplo:

```
ObjectContainer base;
```

Mediante el método openFile de la clase Db4oEmbedded podemos abrir/crear una base de datos.

Ejemplo:

```
base = Db4oEmbedded.openFile("alumnos.db4o");
```

Debemos importar **`com.db4o.*`** para tener acceso y poder trabajar con objetos en la base de datos.

1.6.- Cerrar la base de datos.

Para cerrar la base de datos utilizamos el método close() de la instancia de la clase ObjectContainer creada en la apertura/creación de la base de datos.

Ejemplo:

```
base.close();
```

1.7.- Almacenar objetos en la base de datos.

Las sentencias que permiten trabajar sobre la base de datos se colocan sobre un bloque:

```
try{
      :     :     :
} catch (Exception e){
      :     :     :
} finally {
      :     :     :
}
```

Para almacenar objetos en la base de datos utilizamos el método store() de la instancia de la clase ObjectContainer.

Ejemplo:

```
Alumno alumno1 = new Alumno(1,"Luis","Solis Mas",8.75,"Aprobado");
base.store(alumno1);
```

Código completo de alta de alumno:

```
package bboobjetos;
import com.db4o.*;
public class Bboobjetos {
    public static void main(String[] args) {
        ObjectContainer base;
        base = Db4oEmbedded.openFile("alumnos.db4o");
        try {
            Alumno alumno1 = new Alumno(1,"Luis","Solis
Mas",8.75,"Aprobado");
            base.store(alumno1);
            System.out.println(alumno1.getNombre()+" Almacenado");
            Alumno alumno2 = new Alumno(2,"Marta","Romero
Serra",3.25,"Suspendido");
            base.store(alumno2);
            System.out.println(alumno2.getNombre()+" Almacenado");
        } catch (Exception e){
            System.out.println("Error de la base de datos.");
        } finally {
            base.close();
        }
    }
}
```

1.8.- Listar objetos de la base de datos.

En las bases de datos relacionales las recuperaciones de datos se realiza mediante el lenguaje especifico de base de datos relacionales SQL.

En las Bases de Datos Orientadas a Objetos utilizaremos otro lenguaje llamado QBE. El método queryByExample permite lanzar operaciones sobre la base de datos.

Para recuperar todos los objetos la base de datos se pasa un resultado vacio (String a null y numéricos a 0) al método queryByExample de la instancia creada a la base de datos. El resultado de la consulta se asigna a una instancia de la clase ObjectSet.

El método hasNext() devuelve true mientras el ObjectSet tenga más objetos que recuperar. El método next() pasa al siguiente objeto del conjunto de ObjectSet devueltos por la consulta.

Ejemplo:

```
Alumno miAlumno = new Alumno(0, null, null, 0, null);
ObjectSet resultado = base.queryByExample(miAlumno);
System.out.println("Recuperados "+resultado.size()+" Objetos");
```

```
while(resultado.hasNext()) {
	System.out.println(resultado.next());
}
```

1.9.- Consultar un objeto de la base de datos.

El método queryByExample permite lanzar operaciones sobre la base de datos.

Para recuperar un objeto de la base de datos se pasa al método queryByExample un resultado con los datos del objeto que queremos recuperar poniendo los atributos que no tienen condición a null si se trata de String o a 0 si se trata de atributos numéricos. El resultado de la consulta se asigna a una instancia de la clase ObjectSet.

El método hasNext() devuelve true mientras el ObjectSet tenga más objetos que recuperar. El método next() pasa al siguiente objeto del conjunto de ObjectSet devueltos por la consulta.

Ejemplo:

```
int id=2;
Alumno miAlumno = new Alumno(id, null, null, 0, null);
ObjectSet resultado = base.queryByExample(miAlumno);
System.out.println("Recuperados "+resultado.size()+" Objetos");
while(resultado.hasNext()) {
	System.out.println(resultado.next());
}
```

1.10.- Actualizar objetos en la base de datos.

Para actualizar un objeto basta con modificarlo y llamar al método store() para que se actualice en la base de datos.

Ejemplo:

```
	int id = 2;
	double nota=6.80;
	Alumno a = new Alumno(id, null, null, 0,null);
	ObjectSet resultado = base.queryByExample(a);
	Alumno miAlumno = (Alumno)resultado.next();
	miAlumno.setNota(nota);
	base.store(miAlumno);
	System.out.println("Registro Actualizado");
```

1.11.- Borrar Objetos de la Base de Datos.

Para borrar objetos de la base de datos utilizamos el método delete() sobre el objeto recuperado de la base de datos.

Ejemplo:

```
int id = 1;
Alumno a = new Alumno(id, null, null, 0,null);
ObjectSet resultado = base.queryByExample(a);
Alumno miAlumno = (Alumno)resultado.next();
base.delete(miAlumno);
System.out.println("Registro Eliminado");
```

1.12.- Consultas Nativas (NativeQuery NQ).

Para realizar la consulta nativa, hay que crear un objeto de una clase anónima que implementa la interfaz Predicate. Un Predicate permite definir las condiciones que un objeto determinado debe cumplir .

El método match() define las condiciones de la consulta. Aquellos objetos que devuelven true son los que formaran parte del resultado.

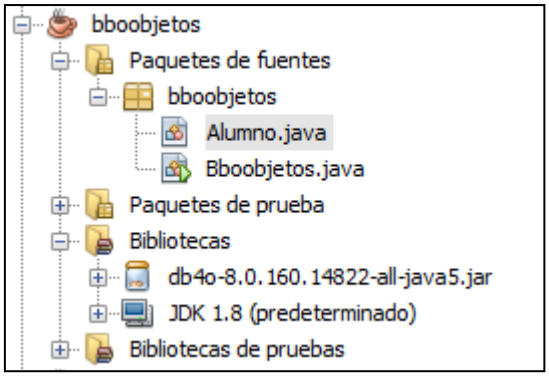
El método query() devuelve un ObjectSet que contiene todos los objetos que cumplen las condiciones.

Debemos importar com.db4o.query.* y debemos sobreescribir el método match() retornando la condición que deben cumplir los objetos.

Ejemplo:

```
Predicate<Alumno> consulta=new Predicate<Alumno>(){
	@Override
	public boolean match(Alumno a){
		return (a.getNota() >= 6 && a.getNota() <=9);
	}
};
ObjectSet resultado = base.query(consulta);
while(resultado.hasNext()) {
	System.out.println(resultado.next());
}
```

1.13.- Código Completo del Ejemplo.



a) Clase Alumno:

```
package bboobjetos;
public class Alumno {
    int  id;
    String nombre;
    String apellidos;
    double nota;
    String observaciones;
```

```
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nobre) {
        this.nombre = nobre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }

    public double getNota() {
        return nota;
    }

    public void setNota(double nota) {
        this.nota = nota;
    }

    public String getObservaciones() {
        return observaciones;
    }

    public void setObservaciones(String observaciones) {
        this.observaciones = observaciones;
    }

    public Alumno(int id, String nombre, String apellidos, double nota,
String observaciones) {
```

```
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.nota = nota;
        this.observaciones = observaciones;
    }

    @Override
    public String toString() {
        return "Alumno{" + "id=" + id + ", nombre=" + nombre + ", apellidos="
+ apellidos + ", nota=" + nota + ", observaciones=" + observaciones + '}';
    }
}
```

b) Clase bdoobjetos.

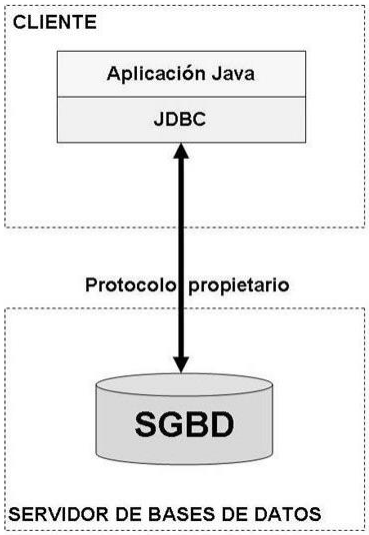
```
package bboobjetos;
import com.db4o.*;
import com.db4o.query.*;
public class Bboobjetos {
    public static void main(String[] args) {
        ObjectContainer base;
        base = Db4oEmbedded.openFile("alumnos.db4o");
        try {
            insertar(base);
            consultar(base, 2);
            actualizar(base);
            eliminar(base);
            listar(base);
            consultarNQ(base);
        } catch (Exception e){
            System.out.println("Error de Base de Datos");
        } finally {
            base.close();
        }
    }

    public static void insertar(ObjectContainer base){
        Alumno alumno1 = new Alumno(1,"Luis","Solis Mas",8.75,"Aprobado");
        base.store(alumno1);
        System.out.println(alumno1.getNombre()+" Almacenado");
        Alumno alumno2 = new Alumno(2,"Marta","Romero
Serra",3.25,"Suspendido");
        base.store(alumno2);
        System.out.println(alumno2.getNombre()+" Almacenado");
    }

    public static void listar(ObjectContainer base){
        Alumno miAlumno = new Alumno(0, null, null, 0, null);
        ObjectSet resultado = base.queryByExample(miAlumno);
        System.out.println("Recuperados "+resultado.size()+" Objetos");
        while(resultado.hasNext()) {
```

```
            System.out.println(resultado.next());
        }
    }

    public static void consultar(ObjectContainer base, int id){
        Alumno miAlumno = new Alumno(id, null, null, 0, null);
        ObjectSet resultado = base.queryByExample(miAlumno);
        System.out.println("Recuperados "+resultado.size()+" Objetos");
        while(resultado.hasNext()) {
                System.out.println(resultado.next());
        }
    }

    public static void actualizar(ObjectContainer base){
        int id = 2;
        double nota=6.80;
        Alumno a = new Alumno(id, null, null, 0,null);
        ObjectSet resultado = base.queryByExample(a);
        Alumno miAlumno = (Alumno)resultado.next();
        miAlumno.setNota(nota);
        base.store(miAlumno);
        System.out.println("Registro Actualizado");
    }

    public static void eliminar(ObjectContainer base){
        int id = 1;
        Alumno a = new Alumno(id, null, null, 0,null);
        ObjectSet resultado = base.queryByExample(a);
        Alumno miAlumno = (Alumno)resultado.next();
        base.delete(miAlumno);
        System.out.println("Registro Eliminado");
    }

    public static void consultarNQ(ObjectContainer base){
        Predicate<Alumno> consulta=new Predicate<Alumno>(){
            @Override
            public boolean match(Alumno a){
                return (a.getNota() >= 6 && a.getNota() <=9);
            }
        };
        ObjectSet resultado = base.query(consulta);
        System.out.println("Alumnos con nota entre 6 y 9:
"+resultado.size());
        while(resultado.hasNext()) {
                System.out.println(resultado.next());
        }
    }
}
```

2.1.- ¿Qué es JDBC?

Java tiene una librería la cual permite interactuar con fuentes de datos (incluidas bases de datos) de tal manera que podemos: Conectarnos a una fuente de datos, enviar consultas a la base de datos y recuperar datos de una consulta y manejarlos.

JDBC son las siglas de Java DataBase Connectivity y es un driver que permite conectar a la base de datos y manipular los datos utilizando SQL.



El cliente accede directamente a los datos a través del driver JDBC. Los comandos son enviados a la base de datos y los resultados son devueltos a la aplicación cliente. Es una configuración cliente/servidor donde la máquina del usuario que corre la aplicación Java es el cliente y el servidor es donde reside la base de datos. Servidor y cliente pueden estar en máquinas diferentes en la misma red local o a través de Internet.

Para trabajar con JDBC se necesitará:

- Establecer la conexión con la Base de datos.
- Crear un objeto Statement.
- Ejecutar la sentencia SQL.
- Leer el resultset o resultado de la consulta.

2.2.- Conexión con la base de datos

JDBC se conecta a las bases de datos utilizando la clase DriverManager que es la forma más sencilla de conectarse a una base de datos.

La conexión con la base de datos se realiza especificando una dirección URL o cadena de conexión y se realiza por medio de una instancia de tipo Connection e importando java.sql.*.

Las partes de la cadena de conexión son:

```
driver:ProtocoloDriver:DetalleConexion
```

Ejemplo:

```
jdbc:mysql://localhost:3306/videoclub
```

Podemos establecer la conexión a la base de datos utilizando una instancia de tipo Connection y pasándole al método getConnection la cadena de conexión o URL, el usuario y la contraseña.

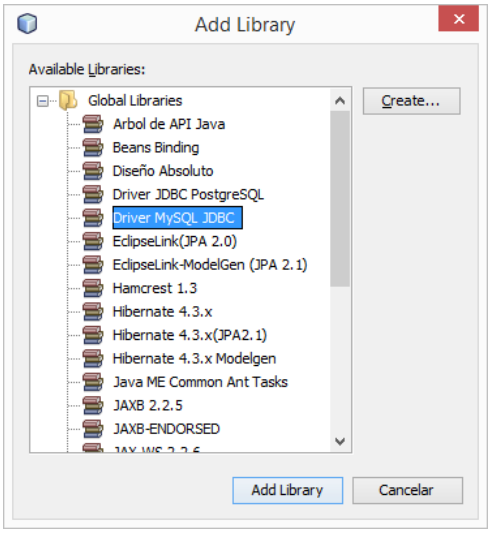
```
Connection miConexion =
      DriverManager.getConnection("jdbc:mysql://localhost:3306/videoclub",
      "root","");
```

Cuando JDBC encuentra un error al trabajar con una base de datos lanza una SQLException y el método SQLException.getMessage nos devuelve una cadena con el error producido. Por tanto, la conexión debe tratarse dentro de un bloque try-catch que maneje las excepciones que se puedan producir como por ejemplo que no se encuentre el driver de conexión, que el servidor de base de datos no esté activo, que la cadena de conexión sea errónea, que la base de datos no exista, que el usuario o contraseña no sean correctos, …

Ejemplo:

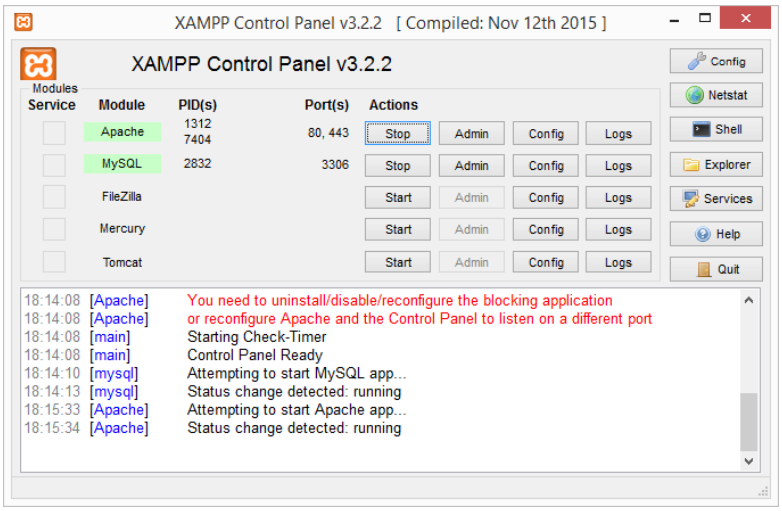
```
try {

    Connection miConexion =
      DriverManager.getConnection("jdbc:mysql://localhost:3306/videoclub",
      "root", "");
    System.out.println("Conexión establecida con éxito!!!");
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

Para poder establecer la conexión de forma satisfactoria de debemos de añadir el driver JDBC correspondiente a la base de datos que queremos manejar, en este caso el driver mysql. Para ello, pulsaremos botón derecho sobre la carpeta Bibliotecas del proyecto Netbeans y pulsaremos la opción Añadir Biblioteca.

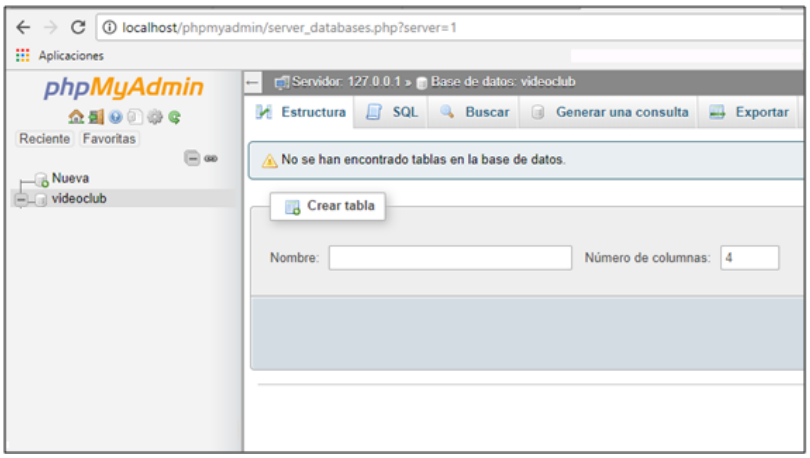


Seleccionaremos la librería "Driver MYSQL JDBC" y pulsaremos el botón "Add Library".

Además, el servidor de base de datos debe estar activo. En el desarrollo del tema se utilizará una base de datos MySQL, por lo que debemos tener activo el servidor instalando por ejemplo el paquete de software que proporciona XAMPP o WAMP.



La gestión de la base de datos la podemos realizar a través de un navegador accediendo a la url: **http://localhost/phpmyadmin**



2.3.- Crear y ejecutar sentencia de consulta SQL

La clase Statemnet se utiliza para crear y ejecutar sentencias SQL. Un objeto de la clase Statement se crea mediante el método createStatement del objeto Connection.

Ejemplo:

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

Una vez creada la instancia del tipo Statement podemos ejecutar una sentencia de consulta SQL con el método executeQuery. El resultado de la sentencia de consulta SQL se devuelve en un objeto del tipo ResultSet que es una tabla virtual.

Ejemplo:

```
ResultSet resultado = sentencia.executeQuery("SELECT * FROM PELICULAS");
```

El acceso a los datos almacenados en el Resultset se realiza mediante un puntero a una zona de memoria donde residen los datos recuperados por la sentencia SQL.

Inicialmente se coloca en una posición anterior a la primera posición de los datos recuperados y mediante la llamada al método next() vamos posicionándonos en la siguiente fila de los datos recuperados.

Podemos recorrer todos los datos obtenidos como resultado con un bucle. Al final del bucle cuando ya no existen más datos el método next() devuelve false.

```
while (resultado.next()) {
      String codpelicula = resultado.getString("codpelicula");
      String titulo = resultado.getString("titulo");
      String tema = resultado.getString("tema");
      int duracion = resultado.getInt("duracion");
      double precio = resultado.getDouble("precio");
      System.out.println("Codigo Pelicula: "+codpelicula);
      System.out.println("Título: "+titulo);
      System.out.println("Tema: "+tema);
      System.out.println("Duración: "+duracion);
      System.out.println("Precio: "+precio);
      System.out.println("Precio con IVA: "+precio*1.21);
      System.out.println("******************************");
}
```

Debemos cerrar la conexión con la base de datos mediante este comando close().

Ejemplo:

```
miConexion.close();
```

2.4.- Devolver el número de registros de una tabla.

```
package basedatosvideoclub;
import java.sql.*;
public class BaseDatosVideoclub {
    public static void main(String[] args) {

        try {
            Connection miConexion =

DriverManager.getConnection("jdbc:mysql://localhost:3306/videoclub",
             "root", "");
            System.out.println("La conexión se ha establecido!!!");
            Statement sentencia = miConexion.createStatement();
            ResultSet resultado = sentencia.executeQuery("SELECT * FROM
PELICULAS");
```

```
            if (resultado.last()){
                System.out.println("Cantidad de Registros : "  +
resultado.getRow());
            } else {
                System.out.println("No Hay Registros ");
            }
            miConexion.close();
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

2.5.- Crear una tabla en la Base de Datos.

La clase Statemnet se utiliza para crear y ejecutar sentencias SQL. Un objeto de la clase Statement se crea mediante el método createStatement del objeto Connection.

Ejemplo:

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

Una vez creada la instancia del tipo Statement podemos ejecutar una sentencia de creación de tabla SQL con el método executeUpdate.

Ejemplo: Tendremos creada de antemano la Base de Datos Empresa

```
package basedatos;
import java.sql.*;
public class BaseDatos {
    public static void main(String[] args) {
        Connection conexion = null;
        try {
            conexion =
DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa","root","");
            System.out.println("Conexión Establecida!");
        }catch (SQLException e) {
            System.out.println("Error: "+e.getMessage());
        }
        String sentencia = "create table empresa.empleados (id_empleado
integer NOT NULL,"
                + "nombre varchar(40) NOT NULL, "
                + "apellidos varchar(40) NOT NULL, "
                + "poblacion varchar(20) NOT NULL, "
                + "codpostal varchar(5) NOT NULL, "
                + "sueldo decimal(8,2) NOT NULL, "
                + "PRIMARY KEY (id_empleado))";
        Statement stmt = null;
        try {
            stmt = conexion.createStatement();
```

```
            stmt.executeUpdate(sentencia);
            conexion.close();
        } catch (SQLException e) {
            System.out.println("Error: "+e.getMessage());
        }
    }
}
```

2.6.- Insertar registros en la Base de Datos.

La clase Statemnet se utiliza para crear y ejecutar sentencias SQL. Un objeto de la clase Statement se crea mediante el método createStatement del objeto Connection.

Ejemplo:

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

Una vez creada la instancia del tipo Statement podemos ejecutar una sentencia de inserción SQL con el método executeUpdate.

Por Ejemplo:

```
sentencia.executeUpdate("INSERT INTO PELICULAS VALUES (5000,'La forma del agua','drama',145,3)");
```

2.7.- Modificar registros en la Base de Datos.

La clase Statemnet se utiliza para crear y ejecutar sentencias SQL. Un objeto de la clase Statement se crea mediante el método createStatement del objeto Connection.

Ejemplo:

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

Una vez creada la instancia del tipo Statement podemos ejecutar una sentencia de actualización SQL con el método executeUpdate.

Por Ejemplo:

```
sentencia.executeUpdate("UPDATE PELICULAS SET PRECIO = 2.00 WHERE CODPELICULA=5000");
```

2.8.- Eliminar registros en la Base de Datos.

La clase Statemnet se utiliza para crear y ejecutar sentencias SQL. Un objeto de la clase Statement se crea mediante el método createStatement del objeto Connection.

Ejemplo:

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

Una vez creada la instancia del tipo Statement podemos ejecutar una sentencia de eliminación SQL con el método executeUpdate.

Por Ejemplo:

```
sentencia.executeUpdate("DELETE FROM PELICULAS WHERE CODPELICULA=5000");
```

2.9.- Consultas Preparadas.

Las sentencias preparadas son consultas que permiten la parametrización, es decir son consultas en las que los valores de los campos de la clausula where pueden variar, indicando las posiciones de los datos que van a cambiar. Los parámetros se especifican con el carácter ?.

La clase PreparedStatemnet se utiliza para crear y ejecutar consultas preparadas. Un objeto de la clase PreparedStatement se crea mediante el método prepareStatement del objeto Connection, utilizando ? para el parámetro de la consulta.

Ejemplo:

```
PreparedStatement sentencia =
  miConexion.prepareStatement("SELECT * FROM PELICULAS WHERE TEMA=?
AND PRECIO > ?");
```

Una vez creada la instancia del tipo PreparedStatement, debemos indicar cuál es el valor para cada uno de los parámetros de la consulta. Para ello, utilizaremos el método setString(númeroParametro, valor).

Ejemplo:

```
sentencia.setString(1, "Drama");
sentencia.setString(2, 3);
```

Por último podemos ejecutar una sentencia con una consulta preparada con el método executeQuery. El resultado de la sentencia de consulta SQL se devuelve en un objeto del tipo ResultSet que es una tabla virtual.

Ejemplo:

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

El acceso a los datos almacenados en el Resultset se realiza mediante un puntero a una zona de memoria donde residen los datos recuperados por la sentencia SQL.
Inicialmente se coloca en una posición anterior a la primera posición de los datos recuperados y mediante la llamada al método next() vamos posicionándonos en la siguiente fila de los datos recuperados.

Podemos recorrer todos los datos obtenidos como resultado con un bucle. Al final del bucle cuando ya no existen más datos el método next() devuelve false.

```
while (resultado.next()) {
```

```
        String codpelicula = resultado.getString("codpelicula");
        String titulo = resultado.getString("titulo");
        String tema = resultado.getString("tema");
        Integer duracion = resultado.getInt("duracion");
        Double precio = resultado.getDouble("precio");
        System.out.println(codpelicula+" - "+titulo+" - "+tema+" -
"+duracion+" - "+precio+" - "+precio*1.21);
}
```

2.10.- Transacciones.

Nos saldrá algo así en el examen

En entornos multiusuario hay que controlar el acceso concurrente a la BD para evitar inconsistencias.

La Atomicidad es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias. Si esta operación consiste en una serie de pasos, todos ellos ocurren o ninguno.

Por ejemplo: En el caso de una transacción bancaria o se ejecuta tanto el depósito como la deducción o ninguna acción es realizada.

Transacción: Unidad de trabajo lógica.

- Un conjunto de instrucciones sobre la BD
- Si algo va mal, hacer marcha atrás al estado anterior
- Si todo va bien, hacer efectivos los cambios

Por defecto se funciona en modo **autocommit:**

- Cada instrucción sobre la BD va en su propia transacción.
- Es seguro pero ineficiente.
- Se desactiva con ***conexion.setAutoCommit(false)***

La interfaz Connection ofrece métodos para las transacciones:

- **commit()**
  - Hace efectivos todos los cambios desde el último commit/rollback.
  - Libera los bloqueos de la BD que tuviera el objeto Connection.
- **rollback()**
  - Deshace todos los cambios realizados en la transacción.
  - Libera los bloqueos de la BD que tuviera el objeto Connection.

Por ejemplo, imaginemos una transferencia de dinero entre dos cuentas bancarias. La primera operación es restar de la cuenta A la cantidad a transferir para luego ingresarla en la cuenta B. En el caso de que haya algún problema para actualizar el saldo de la cuenta B y esta operación no se realice, nos podemos encontrar con que en la cuenta A se ha retirado una cantidad de dinero y en la cuenta B no se ha hecho efectiva la transferencia. La base de datos en ese momento se quedaría inconsistente.

Cuando se realiza una operación de modificación de base de datos (borrado, inserción o actualización) los cambios se producen cuando la sentencia se ejecuta. No hace falta ejecutar una orden para actualizar cambios en la base de datos. Esto es así porque está habilitado el modo auto-commit en la conexión con la base de datos. Para deshabilitar el modo auto-commit en la base de datos hay que ejecutar la siguiente sentencia:

```
conexion.setAutoCommit(false);
```

El método rollback generalmente se utiliza cuando se lanza una excepción. En el momento que se lanza la excepción esto nos indica que algo ha pasado pero nunca nos va a decir qué datos se han modificado y cuáles no. Por lo tanto el rollback deshace todos los cambios hechos hasta el momento desde el último COMMIT, anulando los pasos realizados en la transacción que ha producido el error.

Ejemplo:

Fichero banco.sql

```
CREATE DATABASE bancos;

CREATE TABLE cuentas (
       NUMCUENTA int(11) NOT NULL,
       NOMBRE varchar(30) NOT NULL,
       SALDO decimal(10,2) NOT NULL,
       PRIMARY KEY (NUMCUENTA));

INSERT INTO cuentas (NUMCUENTA, NOMBRE, SALDO) VALUES
(1, 'Luis', '1500.00'),
(2, 'Pedro', '1800.00');
```

Proyecto Java: Transaccion

```
package transaccion;
import java.sql.*;
public class Transaccion {
    public static void main(String[] args) {
        //  Definimos una variable del tipo conexión
        Connection conexion;
        try {
            // Establecemos una conexión a la base de datos banco de mysql
            conexion                                                              =
DriverManager.getConnection("jdbc:mysql://localhost:3306/banco","root","");

            //Ejecutamos una transacción entre las cuentas 1 y 2 por 500€ en la base
de datos banco.
            transaccion(conexion,"banco",1,2,500);
        } catch (Exception e) {
            System.out.println("Mensaje: "+e.getMessage());
        }
    }

    static  void  transaccion(Connection  conexion,  String  BDNombre,int  cuentaA,int
cuentaB,int cantidad) throws SQLException {
        // Declaramos una instancia del tipo Statement para crear las consultas
```

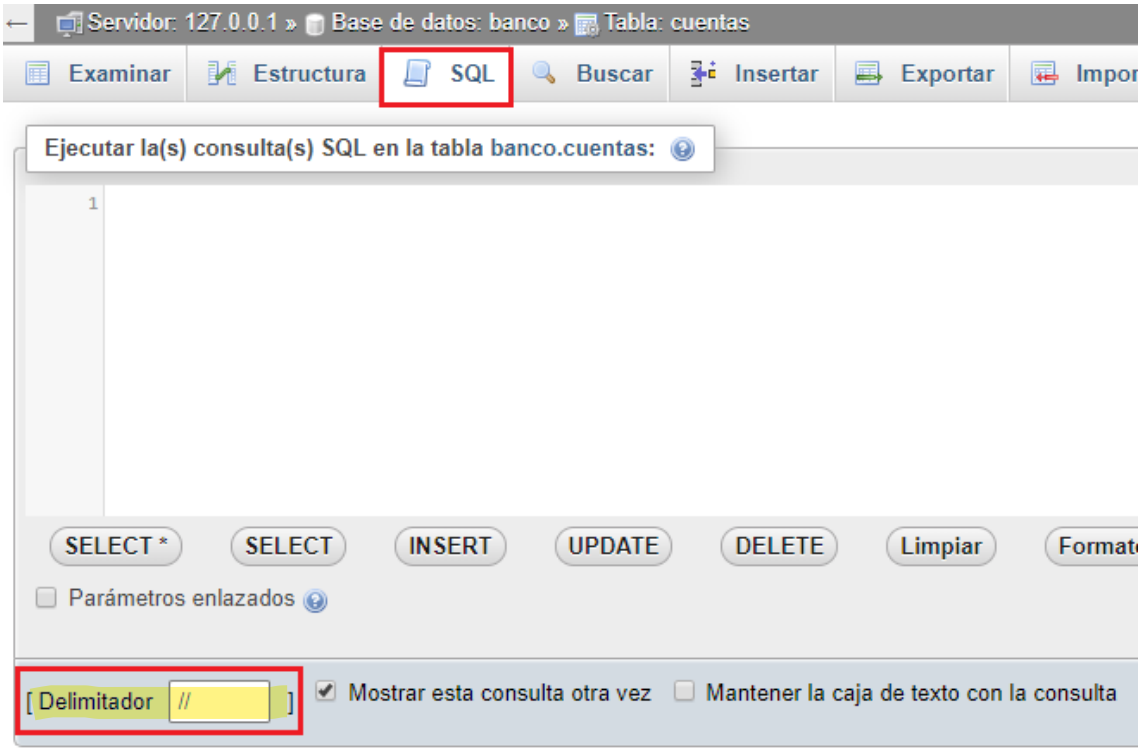
```
        Statement sentencia = null;
        // Creamos dos variables que almacenarán los registros afectados por las
actualizaciones
        int registrosA;
        int registrosB;
        // Creamos dos variables que almacenarán saldos de las cuentas afectadas por
la transacción
        double saldoA;
        double saldoB;
        // Declaramos la variable que almacenará el conjunto de registros
        ResultSet registros;
        // Creamos las sentencias
        String actualizaA = "update " + BDNombre + ".CUENTAS " +"set SALDO = SALDO -
"+cantidad+" where NUMCUENTA = "+cuentaA;
        String actualizaB = "update " + BDNombre + ".CUENTAS " +"set SALDO = SALDO +
"+cantidad+" where NUMCUENTA = "+cuentaB;
        String consultaA = "select SALDO from cuentas where NUMCUENTA = "+cuentaA;
        String consultaB = "select SALDO from cuentas where NUMCUENTA = "+cuentaB;
        try {
            // Desactivamos el autoCommit para que no se realicen los cambios en la
base de datos hasta que no se active
            conexion.setAutoCommit(false);
            // Creamos la sentencia
            sentencia = conexion.createStatement();
            // Ejecutamos las consultas
            registrosA = sentencia.executeUpdate(actualizaA);
            registrosB = sentencia.executeUpdate(actualizaB);
            // Obtener saldos de Cuentas
            registros = sentencia.executeQuery(consultaA);
            registros.next();
            saldoA = registros.getDouble("SALDO");
            registros = sentencia.executeQuery(consultaB);
            registros.next();
            saldoB= registros.getDouble("SALDO");
            // Si alguna de las actualizaciones no devuelve registros se lanza una
excepción
            if (registrosA == 0 || registrosB == 0 || saldoA < 0 || saldoB < 0){
                throw new ExcepcionActualizacion();
            }
            System.out.print("Transaccion realizada con Éxito");
        } catch (ExcepcionActualizacion e ) {
            // La excepción indica que se hace un Rollback
            System.out.print(e.getMessage());
            // Realizamos el Rockball deshaciendo las sentencias del bloque de la
transacción
            // realizadas hasta el momento.
            conexion.rollback();
        } finally {
            // Activamos el autoCommit
            conexion.setAutoCommit(true);
            // Cerramos la conexión a la base de datos
            conexion.close();
        }
    }
}

class ExcepcionActualizacion extends Exception {
```

```
    public ExcepcionActualizacion() {
        // Muestra el mensaje que ha habido un error
        super("Error. Se anulará la transacción");
    }
}
```

2.11.- Procedimiento Almacenado.

En el examen tendremos dos tipos unos tendremos que hacer el procedimiento y otros las funciones

Acceder al entorno phpmyadmin y en la base de datos que queremos crear el procedimiento pulsa SQL y cambiar el delimitador a // para indicar el final del procedimiento.



A continuación escribimos el procedimiento. Por ejemplo:

```
CREATE PROCEDURE ingreso(IN nC INT, IN cantidad decimal(10,2))
BEGIN
	UPDATE CUENTAS SET SALDO = SALDO + cantidad WHERE NUMCUENTA = nC;
END;
//
```

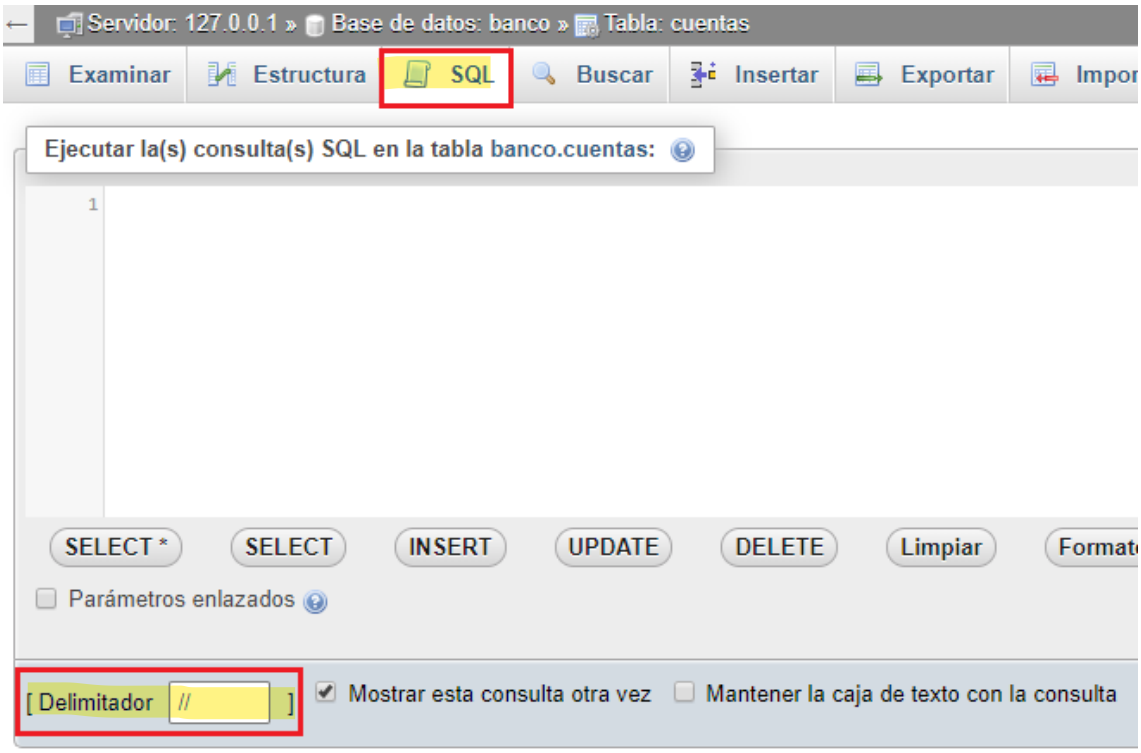
Y pulsamos **Continuar**. El procedimiento se almacenará en la base de datos mysql en la tabla proc. Además se muestra dentro de la base de datos en el apartado ***Procedimientos***.

Para ejecutar el procedimiento utilizaremos CALL:

```
CALL ingreso(2, 200);
```

2.12.- Funciones.

Acceder al entorno phpmyadmin y en la base de datos que queremos crear la función pulsa SQL y cambiar el delimitador a // para indicar el final de la función.



A continuación escribimos la función. Por ejemplo:

Devuelve

```
CREATE FUNCTION consultaSaldo(nC INT) RETURNS decimal(10,2)
BEGIN
       DECLARE cantidad DECIMAL(10,2);
       SET cantidad=0;
       SELECT saldo INTO cantidad FROM CUENTAS WHERE NUMCUENTA = nC;
    RETURN cantidad;
END;
//
```

Ya no indicamos la entrada ni salida

Asigno

Que coincida con el que le paso como parámetro

Y pulsamos **Continuar**. La función se almacenará en la base de datos mysql en la tabla proc. Además se muestra dentro de la base de datos en el apartado ***Funciones***.

Para ejecutar la función utilizaremos SELECT:

```
SELECT consultaSaldo(2);
```

2.13.- Funciones y Procedimientos almacenados desde Java.

Para poder ejecutar un procedimiento necesitamos una variable del tipo **CallableStatement** que almacena una llamada al método **prepareCall** sobre la conexión a la base de datos. Las llamadas a los procedimiento deben ir entre llaves. Por ejemplo: **`{Call ingreso(2,200)}`**

Ejemplo:

```
package procsubida;
import java.sql.*;
public class ProcSubida {
    public static void main(String[] args) {
 //  Definimos una variable del tipo conexión
```

```
        Connection conexion;
        try {
            // Establecemos una conexión a la base de datos banco de mysql
            conexion                                                    =
DriverManager.getConnection("jdbc:mysql://localhost:3306/banco","root","");

            // Construimos la sentencia
            String sentencia = "{CALL ingreso(?,?)}";

            // Preparamos la sentencia
            CallableStatement                procedimiento              =
conexion.prepareCall(sentencia);

            // Damos valor a los argumentos
            procedimiento.setInt(1, 2);
            procedimiento.setDouble(2,350);

            // Ejecutamos el procedimiento
            procedimiento.executeUpdate();
            System.out.println("Ingreso realizado!!!");

            // Cerramos conexión
            conexion.close();
        } catch (Exception e) {
            System.out.println("Mensaje: "+e.getMessage());
        }
    }
}
```

Si deseamos ejecutar una función utilizaremos parámetros de salida de la consulta preparada con **registerOutParameter(numParametro, tipoDato).** La sentencia de llamada debe ir entre llaves y asignada a un parámetro. Por ejemplo: **`{ ? = call consultaSaldo(?) }`**

Ejemplo:

```
package funcsaldo;
import java.sql.*;
public class FuncSaldo {
    public static void main(String[] args) {
        //  Definimos una variable del tipo conexión
        Connection conexion;
        try {
            // Establecemos una conexión a la base de datos banco de mysql
            conexion                                                    =
DriverManager.getConnection("jdbc:mysql://localhost:3306/banco","root","");

            // Construimos la sentencia
            String sentencia = "{? = CALL consultaSaldo(?)}";

            // Preparamos la sentencia
            CallableStatement funcion = conexion.prepareCall(sentencia);
```

```
            // Damos valor a los argumentos
            int nC = 2;
            funcion.registerOutParameter(1, Types.DECIMAL);
            funcion.setInt(2, nC);

            // Ejecutamos el procedimiento
            funcion.executeUpdate();
            System.out.println("Saldo de la cuenta "+nC+" es de
"+funcion.getDouble(1)+" €");

            // Cerramos conexión
            conexion.close();
        } catch (Exception e) {
            System.out.println("Mensaje: "+e.getMessage());
        }
    }
}
```

2.14.- Creación de Savepoints.

Los Savepoints permiten realizar un rollback(), es decir deshacer las sentencias ejecutadas sobre la base de datos hasta un punto (savepoint).

Para crear un Savepoint crearemos una instancia del tipo Savepoint.

```
Savepoint punto = conexion.setSavepoint();
```

Para deshacer todas las sentencias ejecutadas sobre la base de datos hasta el Savepoint punto ejecutaremos la sentencia rollback sobre el punto:

```
conexion.rollback(punto);
```

Ejemplo:

```
package savepoints;
import java.sql.*;
public class SavePoints {
    public static void main(String[] args) {
        // Creamos instancia de conexion
        Connection conexion;
        try{
            // Realizamos la conexion a la base de datos videoclub
            conexion =
DriverManager.getConnection("jdbc:mysql://localhost:3306/videoclub","root",""
);

            // Desactivamos el autocommit para poder hacer rollback
            conexion.setAutoCommit(false);

            // creamos la sentencia INSERT como consulta preparada
            String sentenciaSQL = "INSERT INTO peliculas VALUES(?,?,?,?,?)";
```

```
            PreparedStatement sentencia =
conexion.prepareStatement(sentenciaSQL);

            // Inserción 1
            sentencia.setInt(1, 6001);
            sentencia.setString(2, "Batman");
            sentencia.setString(3, "Aventuras");
            sentencia.setInt(4, 134);
            sentencia.setDouble(5, 3.5);
            sentencia.executeUpdate();

            // Inserción 2
            sentencia.setInt(1, 6002);
            sentencia.setString(2, "Batman");
            sentencia.setString(3, "Aventuras");
            sentencia.setInt(4, 134);
            sentencia.setDouble(5, 3.5);
            sentencia.executeUpdate();

            // Inserción 3
            sentencia.setInt(1, 6003);
            sentencia.setString(2, "Batman");
            sentencia.setString(3, "Aventuras");
            sentencia.setInt(4, 134);
            sentencia.setDouble(5, 3.5);
            sentencia.executeUpdate();

            // Crea un savepoint para poder deshacer con rollback hasta el
savepoint punto
            Savepoint punto = conexion.setSavepoint();

            // Inserción 4
            sentencia.setInt(1, 6004);
            sentencia.setString(2, "Batman");
            sentencia.setString(3, "Aventuras");
            sentencia.setInt(4, 134);
            sentencia.setDouble(5, 3.5);
            sentencia.executeUpdate();

            // Inserción 5
            sentencia.setInt(1, 6005);
            sentencia.setString(2, "Batman");
            sentencia.setString(3, "Aventuras");
            sentencia.setInt(4, 134);
            sentencia.setDouble(5, 3.5);
            sentencia.executeUpdate();

            // deshace los dos últimos inserts hasta el savepoint punto
            conexion.rollback(punto);

            // Hace commit de los 3 primeros inserts que están antes del
savepoint punto
```

```
            conexion.setAutoCommit(true);

            // Mostramos mensaje de inserción de los 3 primeros INSERTS
            System.out.println("Se han realizado los 3 primeros inserts hasta
el Savepoint punto");

            // cierra la base de datos
            conexion.close();
        }catch(SQLException e){
            System.out.println(e.getMessage());
        }
    }
}
```