**JDBC**

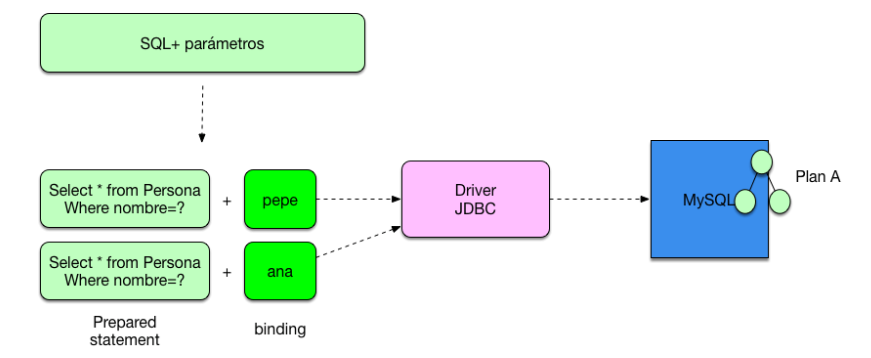
Paquete JAVA.SQL

<https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/package-summary.html>

**PreparedStatement**

A dos consultas diferentes (por sus parametros diferentes) **se crean dos planes de ejecución diferentes** aunque ambas consultas sean realmente muy similares y únicamente entre e**n juego el valor del parámetro que las pasamos** . Para solventar este problema existen **los JDBC Prepared statement** . Estas estructuras permiten mantener **las consultas neutras** sin tener en cuenta los parámetros que se las pasa ya que realiza un binding de ellos.

De esta forma cuando la base de datos genera un hash para el plan de ejecución ambas consultas , la que consulta por pepe y la que consulta por ana devuelven **el mismo hash y comparten el plan de ejecución**.



Ejemplo:

String consulta = "select \* from Persona where nombre = ? ";

Connection conexion= DriverManager.getConnection("jdbc:mysql://localhost/prueba", "root", "root");

PreparedStatement sentencia= conexion.prepareStatement(consulta);

sentencia.setString(1, "pepe");

ResultSet rs = sentencia.executeQuery();

**Codigo Eclipse:**

****

Ejemplo corriéndolo con un proyecto de Java en lugar de Maven.

1. Debo bajarme el mysql.connector.java.(version).jar a mi proyecto.
2. Click botón derecho sobre el Proyecto. => Properties => Libraries => ADD/JAR FOLDER

Y en mi caso tenia descargado el connector sql en la siguiente ruta:

C:\Users\lrlopez\.m2\repository\mysql\mysql-connector-java\8.0.20\

1. Entonces importe ese .jar y luego fue posible impactar en la base desde un proyecto de Java.

A diferencia del Statement, PreparedStatement trabaja en forma precompilada. Cuando es ejecutado la BDD simplemente ejecuta la sentencia sql sin tener que compilarla primero. La otra ventaja es que nos permite trabajar con parámetros.

Probamos Insertar, Modificar, Consultar y Eliminar.

**Estructura TRY-WITH-RESOURCES**



**Manejo de Excepciones**

La idea es ir lanzando excepciones un poco mas elaboradas. Por ejemplo: una persona ya fue ingresada dentro de la BBDD (duplicado).

Vemos como englobar las transacciones SELECT(Consultar), INSERT, UPDATE y DELETE con excepciones. Le sacamos el throws genérico al método y trabajamos un poco mas las posibles excepciones.

Si hay un throws al método debería ser sobre excepciones de negocio (objetos).

**CLASE\_09\_EJEMPLO\_2**

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

**Manejo de Transacciones – 1ra parte**

Comenzamos diciendo que el método execute() que vemos en los 4 metodos estáticos ejecuta la transaccion SQL en forma inmediata.

Esto puede no ser tan apropiado por ejemplo en el caso de un pago bancario. Que sucedería si en el momento de efectuar un pago hay algun problema de conexión y el mismo no termina de concretarse.

Bueno para solucionar esto existe un método llamado rollback(), que deja los datos en su estadio inicial. Es decir, antes de ejecutar la transacción.

Entonces si quisiéramos trabajar con transacciones de manera programática lo primero que hay que hacer es poner el autocommit en false. Esto hace que los cambios no se autoComiten en la base de datos. Es decir no se ejecuten si o si.

Entonces dentro del **try** y al inicio del método después de la conexión ponemos

**Clase personaDAO**

con.setAutoCommit(false)

//sentencias DML hacia la BDD

//luego de finalizadas todas las sentencias y el comando execute()

//se procede a comitear

con.commit();

Que pasa si ocurre un error=>

Colocamos el rollback dentro del catch para que cuando se maneje la excepción se produzca el rollback de la transaccion.

En el ejercicio del siguiente codigo lo que hacemos es

1. Creamos un campo **auxiliar** en la tabla personas desde el **Workbench**.
2. A ese campo lo creamos como **varchar(4)** de forma tal que solo permita ingresar hasta 4 caracteres.
3. En **PersonaDAO** vamos poniendo try … catch con los correspondientes rollback.
4. Ejecutamos el **insertar**(p) desde el **main** en **Java** y vemos que ocurre.

CLASE\_09\_EJEMPLO\_3

**Codigo Netbeans:**



**Codigo Eclipse:**

**Transaccion**

Una transacción en SQL es una colección de sentencias DML que forman una unidad lógica de trabajo o procesamiento, con propiedades bien definidas.

JDBC permite que las declaraciones de SQL sean agrupadas juntas en una sola transacción. De esta manera, es posible asegurar la atomicidad y consistencia de datos.

**setAutoCommit()**

El método setAutoCommit es el encargado de establecer si se trabajara agrupando varias sentencias en una transacción, o por el contrario cada sentencia SQL sera una transacción independiente

Para trabajar con varias sentencias SQL y ejecutarlas como una transacción es necesario establecer el auto-commit en false:

unaConexion.setAutoCommit(false) ;

Si no es necesario trabajar con transacciones, será necesario establecer el auto-commit en true:

unaConexion.setAutoCommit(true); //por defecto

**Commit()**

Si no se llama al método commit(), los cambios no serán reflejados en la base de datos aun cuando se hayan ejecutados una o más sentencias SQL.

La forma de invocarlo es la siguiente:

unaConexion.commit() ;

**rollback()**

Vuelve atrás todos los cambios realizados sobre los datos desde el último commit() realizado.

El método rollback() tambien pertenece a la conexion, y la forma de invocarlo es la siguiente: unaConexion.rollback();

**Ejercicio Integrador**

Algunas consideraciones:

1. La declaración del PreparedStatement y de la conexion la sacamos fuera del try en null para poder ser tratadas luego en el metodo.
2. El PreparedStatement se puede guardar como un atributo de la clase para ganar **eficiencia**. Otro beneficio del PreparedStatement es evitar errores de escritura de la sentencia DML. El uso de consultas parametrizadas mejora el rendimiento entre un 20 y un 30 % a nivel de base de datos.

PRUEBAS:

Probamos si el producto existe. Si existe, nos avisa. Y sino existe lo inserta.



1. Ejercicio Integrador con Collections:

**Codigo: Clase09\_Ejercicio\_Integrador.**

Agregamos el método getProductos() en PersonaDAO.

Objetivo:

//todos los productos que tengo en la base de datos deberia

//de transformarlos a objetos Producto y devolver toda la lista

//de productos



Agregamos el método getProductos(), haciendo algunas modificaciones al método getProducto() previamente creado. Y desde el main insertamos un registro y corremos el nuevo método creado:

Producto p1 = new Producto(420,"Cafe Indu");

ProductoDAO.inserta(p1);

System.out.println(ProductoDAO.getProductos());//aplicamos el metodo

**Codigo Netbeans:**



**Codigo Eclipse:**

****