ACCESO A DATOS

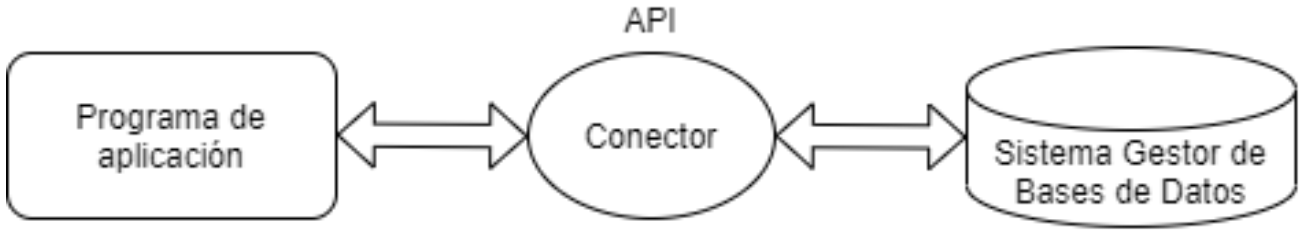
UNIDAD 03 – BBDD RELACIONALES

INDICE

3.1 – Conectores para BBDD

3.1 – CONECTORES PARA BBDD

- Los SGBD de distintos tipos tienen sus propios lenguajes especializados para operar con datos.
- Los programas de aplicación se escriben con lenguajes de programación de propósito general.
- Para que los programas puedan interactuar con los SGBD necesitan APIs llamadas conectores.

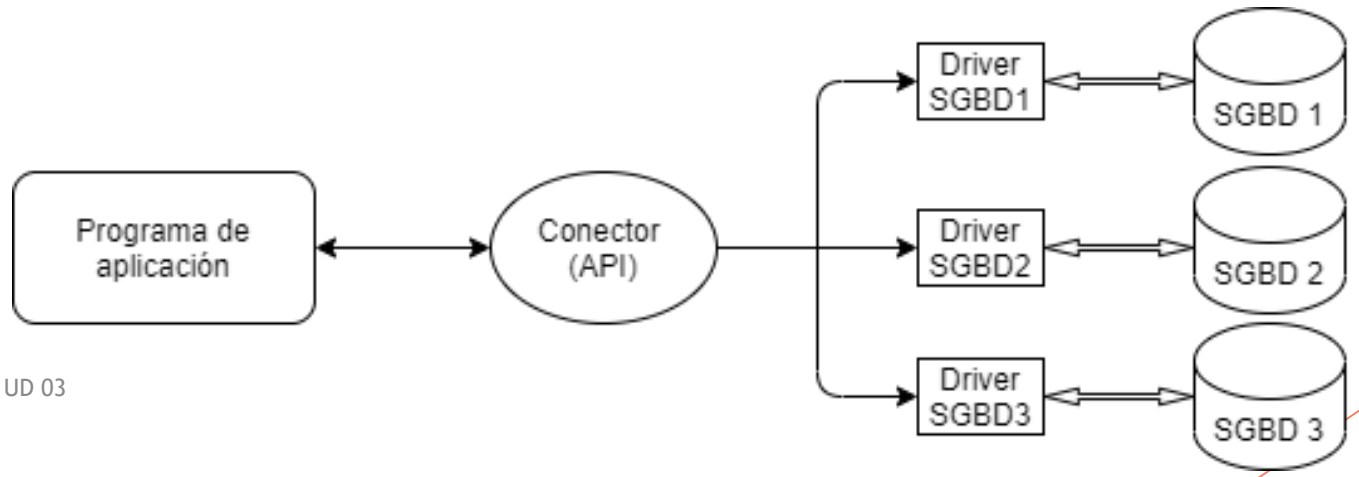


3.1 – CONECTORES PARA BBDD

- Los SGBD más utilizados con diferencia son relacionales.
- Para trabajar con ellos se utiliza SQL.
- Existen muchas BBDD relacionales y cada una tiene su propia versión de SQL y su propia versión de interfaz a bajo nivel.
- Los *drivers* permiten desarrollar una arquitectura genérica en la que el conector tiene una interfaz común. Los *drivers* se ocupan de las particularidades de las distintas BBDD.

3.1 – CONECTORES PARA BBDD

- De esta manera, el conector no es un simple API, sino una arquitectura, porque especifica unas interfaces que los distintos drivers tienen que implementar para acceder a las BBDD particulares.



3.1 – CONECTORES PARA BBDD

- La primera arquitectura de conectores que surgió fue ODBC (Open DataBase Connectivity) desarrollada por Microsoft para Windows a principios de los 90.
- Sucesoras de ODBC han sido OLE-DB y ADO (ActiveX Data Objects).
- A finales de los 90 surge JDBC como equivalente de ODBC para Java, el cual es muy similar.
- Existen drivers para ellas, no solo para BBDD relacionales.
- Todas las BBDD importantes proporcionan drivers de ODBC y JDBC.
- Se dispone de un driver JDBC para ODBC por si no existiera driver específico de JDBC pero sí de ODBC.

3.1 – CONECTORES PARA BBDD

- La principal ventaja del uso de drivers es la independencia de la BD.
- A cambio se tiene una mayor complejidad, y en algunos casos, menor rendimiento.
- En cualquier caso, el uso de un conector para una BBDD relacional es igual en lo esencial, y conforme al estándar X/Open SQL CLI.

3.2 – Consultas a BBDD mediante conectores

3.2 – CONSULTAS A BBDD CON CONECTORES

- Los conectores permiten todo tipo de operaciones pero, por ahora, nos centraremos en las consultas.
- La cuestión fundamental con conectores es la correspondencia entre las estructuras de datos utilizadas para el almacenamiento en las base de datos y las estructuras de datos de las que dispone el lenguaje de programación.
- En la BBDD relacionales, la estructura fundamental para el almacenamiento es la tabla.
- Cada tabla tiene filas con un tipo de dato.
- Una consulta SQL devuelve un conjunto de filas (*recordset*).
- Los conectores permiten recuperar esos resultados fila a fila mediante un iterador o cursor.

3.2 – CONSULTAS A BBDD CON CONECTORES

- Ejemplo de consulta con conectores en Java y con clases JDBC, aunque en esencia es igual en cualquier lenguaje.
- El objeto de tipo *ResultSet* actúa como iterador sobre los resultados de la consulta, los cuales pueden ser accedidos por su posición (como en el ejemplo) o por su nombre.
- Según el tipo de BBDD habrán diferentes operaciones, pero siempre se tratará de: abrir conexión, realizar consulta y utilizar iterador/cursor para obtener los datos.

```
Connection c = getConnection(datos de conexión)
Statement s = c.createStatement();
ResultSet rs = s.executeQuery("SELECT...");
while(rs.next()){//En rs están disponibles más resultados de la consuta
    String dato1 = rs.getString(1); // obtener String de primera columna
    int dato2 = rs.getInt(2); // ontener int de segunda columna
}
s.close();
c.close();
```

3.3 – Desfase Objeto-relacional

3.3 – DESFASE OBJETO-RELACIONAL

- Hacer al contrario, es decir almacenar los resultados de variables de memoria en una BBDD relacional, puede ser más complicado. **Especialmente cuando se trabaja con POO y objetos complejos**.
- Una colección de objetos tiene estructura de grafo, y no es sencillo almacenar la información en tablas con filas y columnas.
- Al conjunto de dificultades se le conoce como desfase objeto-relacional.
- Para la persistencia de objetos complejos hay 2 posibilidades:
  - BBDD de objetos
  - Técnicas de correspondencia objeto-relacional (ORM)

3.4 – JDBC

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- JDBC está basada en drivers y su API está disponible en el paquete java.sql.
- Los drivers de JDBC proporcionan clase que implementan las interfaces de la API JDBC para una base de datos en particular.
- Si no existiera driver JDBC pero sí ODBC para una base de datos, se podría utilizar el driver JDBC para el conector ODBC.
- Sitio web de Oracle con información general, una breve introducción y tutoriales de JDBC.
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- **Operaciones básicas con JDBC**
- Se usará los conocimientos de SQL tanto de DML (data manipulation language) como de DDL (data definition language).
- DML – Diferenciamos entre consulta y modificación
  - SELECT → *executeQuery()* devuelve *Resultset*
  - UPDATE, DELETE, INSERT → *executeUpdate* devuelve nº filas afectadas.
- DDL
  - *execute()*

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- Operaciones básicas con JDBC

Comandos JDBC	Funcionalidad
Class.*forName*(*nombre del driver*); // No necesario desde JDBC 4.0 (Java SE 6)	Cargar driver
Connection c = DriverManager.*getConnection*(datos de conexión);	Crear conexión
Statement s = c.*createStatement*();	Crear sentencia

ResultSet rs = s.*executeQuery*(*consulta*);
while(rs.*next*()) {
…
}
rs.*close*();
	int res = s.*executeUpdate*(*sent.DML*);
	
	(o bien)
	
	boolean res = s.*execute*(*sent.DDL*);
		Ejecutar sentencia. Si consulta, obtener resultados fila a fila y cerrar lista de resultados

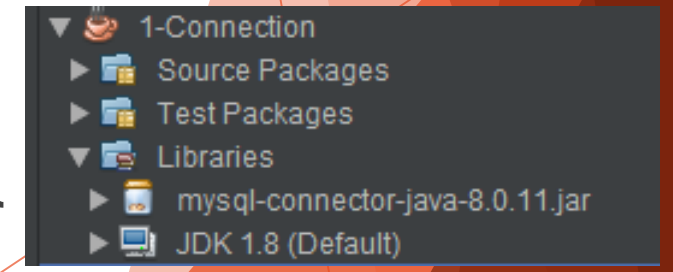
s.*close*();	Cerrar sentencia
c.*close*();	Cerrar conexión

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- Apertura y cierre de conexiones
- Los drivers JDBC están disponibles en ficheros tipo jar.
- Se puede establecer una conexión mediante la clases *DriverManager*, con el método *getConnection(String URL_conexión)*.
- La *URL_Conexión* contiene los datos necesarios para establecer la conexión con la BD.
- Este método carga automáticamente en memoria las clases para los drivers de JDBC de versión 4.0 o posteriores disponibles.

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- Después selecciona el apropiado para la BD indicada en la URL de conexión y le pasa esta para que establezca la conexión.
- Si consigue establecer la conexión devuelve como resultado una *Connection* y se encargará de todas las operaciones realizadas con ella.
- El driver para MySQL 8.0, por ejemplo lo proporciona la clase com.mysql.cj.jdbc.Driver.
- Para añadir un driver de JDBC en un IDE como NetBeans o Eclipse, se puede añadir el fichero jar al proyecto.



3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- El siguiente programa abre una conexión a una BD MySQL y luego la cierra.
- Para la conexión hace falta indicar el servidor, la bd y la autenticación.
- Para abrir y cerrar los recursos se utiliza un bloque *try* con recursos. Esto evita tener que cerrar explícitamente recursos con *close()* incluso aunque hubiera una excepción.

```java
package jdbc_connection;

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

public class JDBC_Connection {

  public static void muestraErrorSQL(SQLException e) {
    System.err.println("SQL ERROR mensaje: " + e.getMessage());
    System.err.println("SQL Estado: " + e.getSQLState());
    System.err.println("SQL código específico: " + e.getErrorCode());
  }

  public static void main(String[] args) {

    String basedatos = "...";
    String host = "localhost";
    String port = "3306";
    String parAdic = "...";
    String urlConnection = "jdbc:mysql://" + host + ":" + port + "/" + basedatos + parAdic;
    String user = "...";
    String pwd = "...";

    //Class.forName("com.mysql.jdbc.Driver");     // No necesario desde SE 6.0
    //Class.forName("com.mysql.cj.jdbc.Driver"); // para MySQL 8.0, no necesario
    try (Connection c = DriverManager.getConnection(urlConnection, user, pwd)) {
      System.out.println("Conexión realizada.");
    } catch (SQLException e) {
        muestraErrorSQL(e);
    } catch (Exception e) {
      e.printStackTrace(System.err);
    }
  }
}
```

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- La interfaz *Statement* se utiliza para ejecutar cualquier tipo de sentencia SQL. Se obtiene con el método *getStatement()* de *Connection*.

Método	Funcionalidad
ResultSet *executeQuery*(String sql)	Ejecuta una consulta (sentencia SELECT de SQL) y devuelve un ResultSet que permite acceder a sus resultados.
ResultSet *getResultSet()*	Obtiene el conjunto de resultados de una consulta SELECT y otros tipos como SPs
int *executeUpdate*(String sql)	Operaciones que modifican los contenidos de la BD. INSERT, UPDATE y DELETE. Devuelve el número de filas afectadas.
boolean *execute*(String sql);	Se puede utilizar para ejecutar cualquier tipo de consulta. Es el método que hay que utilizar preferiblemente para sentencias de DDL tales como CREATE, ALTER, DROP. El valor devuelto depende de la sentencia y sus resultados. Si se trata de una sentencia DDL, devuelve false.
void *close*();	Cierra el Statement

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- **Ejecución de sentencias DDL.**
- DDL se pueden ejecutar con el método *execute()*.
- No es necesario llamada a *close()* ni de *Connection* ni de *Statement*, porque se crean en la parte de inicialización de recursos del bloque *try*.

```
  // Se omite declaración de variables para los datos de conexión
try (
        Connection c = DriverManager.getConnection(urlConnection, user, pwd);
        Statement s = c.createStatement()) {
  s.execute("CREATE TABLE CLIENTES (DNI CHAR(9) NOT NULL,"
          + " APELLIDOS VARCHAR(32) NOT NULL, CP CHAR(5),"
          + " PRIMARY KEY(DNI));");

} catch (SQLException e) {
  muestraErrorSQL(e);
} catch (Exception e) {
  e.printStackTrace(System.err);
}
```

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- **Ejecución de sentencias para modificar contenidos de la BD**
- Se pueden ejecutar con el método *executeUpdate()*.
- Pueden ser sentencias INSERT, UPDATE o DELETE.
- Devolverán el número de filas afectadas.

```
// Se omite declaración de variables para los datos de conexión
    try (
            Connection c = DriverManager.getConnection(urlConnection, user, pwd);
            Statement s = c.createStatement()) {

      int nFil = s.executeUpdate("INSERT INTO CLIENTES (DNI,APELLIDOS,CP) VALUES "
              + "('78901234X','NADALES','44126'),"
              + "('89012345E','HOJAS', null),"
              + "('56789012B','SAMPER','29730'),"
              + "('09876543K','LAMIQUIZ', null);");

      System.out.println(nFil + " Filas insertadas.");

    } catch (SQLException e) {
      muestraErrorSQL(e);
    } catch (Exception e) {
      e.printStackTrace(System.err);
    }
```

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

ACTIVIDADES PROPUESTAS.

a) Haz un programa que haga los cambios necesarios para que los contenidos de la tabla CLIENTES sean los siguientes: (‘78901234X’, ‘NADALES’, ‘44126’), (‘89012345E’, ‘ROJAS’, null), (‘56789012B’, ‘SAMPER’, ‘29730’), partiendo de los contenidos de la tabla resultantes de la ejecución del ejemplo anterior. El programa debe utilizar sentencias UPDATE y DELETE.

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- **Ejecución de consultas y manejo de *ResultSet***
- Sentencias SELECT ejecutadas con *executeQuery()* que devolverán un *ResultSet* con los resultados.
- *ResultSet* contiene un conjunto de filas y mantiene internamente un cursor a la fila actual.
- Hay métodos de *ResultSet* para acceder a los datos tanto por posición como por nombre de columna.
- El recordSet se recorre avanzando al siguiente resultado con el método *next()*.

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- Métodos de ResultSet para consultar contenidos

Método	Funcionalidad
boolean *next*();	El cursor del *ResultSet* puede apuntar a cualquier fila suya (o antes de la primera fila o después de la última fila). Inicialmente apunta antes de la primera fila. El método mueve el cursor a la siguiente posición y devuelve *true*, a menos que esté en la última fila o tras ella, en cuyo caso devuelve *false*.
getXXX(int) getXXX(String)	Obtiene el contenido de la columna especificada de la fila actual. Se puede especificar una fila por su posición o por su nombre. Hay distintas funciones para distintos tipos: *getInt(), getString(), getDate(),* etc.
void *close*();	Cierra el *Statement*. Se debería hacer siempre.

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

- **Ejemplo que muestra los datos de todos los clientes en la tabla.**
- No es necesario llamada a *close()* ni del *Statement* ni del *ResultSet* porque se crean en la parte de inicialización de recursos del bloque *try*.

```
// Se omite declaración de variables para los datos de conexión
    try (
            Connection c = DriverManager.getConnection(urlConnection, user, pwd);
            Statement s = c.createStatement();
            ResultSet rs = s.executeQuery("SELECT * FROM CLIENTES")) {

      int i=1;
      while (rs.next()) {
        System.out.println("[" + (i++) + "]");
        System.out.println("DNI: " + rs.getString("DNI"));
        System.out.println("Apellidos: " + rs.getString("APELLIDOS"));
        System.out.println("CP: " + rs.getString("CP"));
      }

    } catch (SQLException e) {
      muestraErrorSQL(e);
    } catch (Exception e) {
      e.printStackTrace(System.err);
    }
```

3.4 – JAVA DATABASE CONNECTIVITY (JDBC)

ACTIVIDADES PROPUESTAS.

a) El código postal (columna CP) está definido con tipo CHAR(5) en la tabla CLIENTES, pero es siempre un número entero. ¿Se podría utilizar getInt() en lugar de getString() para recuperar su valor? Cambia el programa y verifica tu hipótesis, o justifica los resultados si no son los que esperabas.

3.5 – Sentencias preparadas, transacciones y claves autogeneradas

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Sentencias preparadas (SP)**
- Hasta ahora las sentencias SQL era fijas, pero necesitaremos poder ejecutarlas con variables.
- Se podría crear una sentencia en un *String* y asignarles variables de programación, pero esto plantea problemas de ***Seguridad*** (ataques de inyección SQL) y ***Rendimiento*** (análisis y plan de ejecución cada vez que cambie sólo una variable).
- Las sentencias preparadas evitan estos problemas introduciendo marcadores (placeholders) donde se introducirán valores en el momento de ejecutar la sentencia.
- Las sentencias preparadas se proporcionan al servidor de BBDD solo una vez, y este la prepara y precompila, pasándole un valor para cada marcador cada vez que se ejecuta la consulta.

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Sentencias preparadas (SP)**
- Se utiliza *PreparedStatement* del método *getPreparedStatement* de *Connection*.
- A este método se le pasa como parámetro la sentencia SQL y se utiliza el carácter "*?*" como marcador.

Método	Funcionalidad
ResultSet *executeQuery()* int *executeUpdate()* boolean *execute()*	Estos tres métodos no tienen como parámetro la consulta, esta se le pasó al constructor.
setXXX(int pos, YYYY valor)	Se utilizan para asignar un valor a un placeholder determinado, dado por su posición, siendo 1 la posición del primero. **Los hay con distintos nombres, dependiendo del tipo**.
*setNull(*int pos, int tipoSQL*)*	Asigna valor NULL a una columna. Debe indicarse el tipo. Los posibles tipos están definidos en la clase *java.sql.Types*

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Ejemplo donde se inserta en una tabla CLIENTES1 utilizando sentencias preparadas (SP)**
- Para la inserción de los últimos registros, se ha utilizado una variable que se incrementando. Esto evita problemas y agiliza si se quieren añadir más registros.
- La tabla CLIENTES1 se puede crear en MySQL con : **CREATE TABLE CLIENTES1 (DNI CHAR(9) NOT NULL, APELLIDOS VARCHAR(32) NOT NULL, CP INTEGER, PRIMARY KEY(DNI));**

```
// Se omite declaración de variables para los datos de conexión
try (
        Connection c = DriverManager.getConnection(urlConnection, user, pwd);
        PreparedStatement sInsert = c.prepareStatement("INSERT INTO "
                + "CLIENTES1(DNI,APELLIDOS,CP) VALUES (?,?,?);")) {

  sInsert.setString(1, "78901234X");
  sInsert.setString(2, "NADALES");
  sInsert.setInt(3, 44126);

  sInsert.executeUpdate();

  int i=1;
  sInsert.setString(i++, "89012345E");
  sInsert.setString(i++, "ROJAS");
  sInsert.setNull(i++, Types.INTEGER);

  sInsert.executeUpdate();

  sInsert.setString(i=1, "56789012B");
  sInsert.setString(i++, "SAMPER");
  sInsert.setInt(i++, 29730);

  sInsert.executeUpdate();

} catch (SQLException e) {
  muestraErrorSQL(e);
} catch (Exception e) {
  e.printStackTrace(System.err);
}
```

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

ACTIVIDADES PROPUESTAS.

a) Escribe un programa que muestre los datos de varios clientes, o todos, de la tabla CLIENTES1. El programa debe utilizar una sentencia preparada para la consulta **SELECT * FROM CLIENTES1 WHERE DNI=?** Debe realizarse una consulta para cada cliente, especificando su DNI, y obtener los datos del *ResultSet* resultante, que solo tendrá una fila, al ser el acceso por clave primaria.

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Transacciones**
- Recordemos: un conjunto de operaciones que se ejecutan conjuntamente como un todo, y de manera aislada.
- Se ejecutan completamente o la BD queda como antes de la ejecución.
- Características ACID (atomic, consisten, isolated, durable).
- Unos SGBD, como Oracle, las tienen habilitadas por defecto (se necesita COMMIT para confirmar fin de transacción) y otras, como MySQL, no.
- Una transacción en pseudocódigo sería:

```
INICIA TRANSACCION
Operación 1
Operación 2
…
COMMIT
```

3.5 – SP, TRANSACCIONES Y CLAVES AUTOGENERADAS

- Se puede abortar una transacción con ROLLBACK.
- Si falla la sentencia COMMIT (inusual) también se descartan todos los cambios.
- **Métodos de *Connection* relacionados con Transacciones**

Método	Funcionalidad
void *setAutoCommit*(boolean autoCommit)	Con *autoCommit=false* inicia una transacción. En este sentido, equivale a **START TRANSACTION**
void commit()	Equivale a una sentencia **COMMIT** de **SQL**. Si después de utilizar este método se quiere ejecutar más sentencias pero no en una transacción, se puede hacer *setAutoCommit(true)*.
void rollback()	Descarta todos los cambios realizados por la transacción actual. Se hará normalmente en respuesta a cualquier excepción *SQLException*, para confirmar que se descartan los cambios.

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Ejemplo donde se ejecutan varias sentencias como una transacción.**
- En esta ocasión, no se puede agrupar la creación de la conexión con la de la sentencia preparada en el mismo bloque de inicialización de recursos de un bloque try, porque estos no están disponibles en el bloque catch correspondiente, y entonces la conexión no estaría disponible para hacer *c.rollback()*.
- Además, hay que hacer *c.rollback()* en un bloque *try … catch*, porque puede lanzar una *SQLException*.

```
// Se omite declaración de variables para los datos de conexión
try(
        Connection c = DriverManager.getConnection(urlConnection, user, pwd)) {
    try (
            PreparedStatement sInsert = c.prepareStatement("INSERT INTO "
                    + "CLIENTES1(DNI,APELLIDOS,CP) VALUES (?,?,?);")) {

      c.setAutoCommit(false);

      int i = 0;
      sInsert.setString(++i, "54320198V");
      sInsert.setString(++i, "CARVAJAL");
      sInsert.setString(++i, "10109");
      sInsert.executeUpdate();

      sInsert.setString(i = 1, "76543210S");
      sInsert.setString(++i, "MARQUEZ");
      sInsert.setString(++i, "46987");
      sInsert.executeUpdate();

      sInsert.setString(i = 1, "90123456A");
      sInsert.setString(++i, "MOLINA");
      sInsert.setString(++i, "35153");
      sInsert.executeUpdate();

      c.commit();

    } catch (SQLException e) {
      muestraErrorSQL(e);
      try {
        c.rollback();
      } catch (Exception er) {
        System.err.println("ERROR haciendo ROLLBACK");
        er.printStackTrace(System.err);
      }
    }
} catch (Exception e) {
    System.err.println("ERROR de conexión");
    e.printStackTrace(System.err);
}
```

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

ACTIVIDADES PROPUESTAS.

a) Comprueba las diferencias entre el programa anterior, en el que se agrupan todas la inserciones de registros con transacciones, y otro programa igual pero sin transacciones. Primero hay que hacer una copia del programa y eliminar el código que gestiona las transacciones (setAutoCommit, commit y rollback). Después se trata de probar ambos programas con un mismo conjunto de datos inicial en la tabla CLIENTES1, para comprobar la manera distinta en que se comportan. El conjunto de datos inicial y el programa podrían ser tales que las dos primeras inserciones se realizaran sin problemas, pero la última no, por haber ya en la tabla un cliente con ese DNI. El programa con transacciones no insertaría ningún registro. El programa sin transacciones insertará registros hasta que se produjera un error. La creación del conjunto de datos inicial debe automatizarse. Se puede hacer mediante una secuencia de sentencias SQL, que se pueden guardar en un fichero de texto, y ejecutar con el intérprete de SQL, o mediante un programa. En esas secuencias podrían borrarse todos los contenidos de la tabla con una sentencia DELETE y añadirse varias filas con sentencias INSERT.

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Claves autogeneradas**
- Es una práctica habitual crear una tabla donde la clave primaria sea una columna numérica y se deje al SGBD que proporcione un nuevo valor para cada registro que se inserta (Ejemplo Facturas)
- El mecanismo es básicamente igual en todas las BBDD con pequeños cambios. Históricamente Oracle tenía particularidades que a partir de la versión 12 se han eliminado.
- Veamos ejemplos de definición de claves autogeneradas o autoincrementales.

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

MySQL	Oracle 12 o porterior con IDENTITY
CREATE TABLE FACTURAS *(* *NUM_FACTURA INTEGER AUTO_INCREMENT NOT NULL,* *DNI_CLIENTE CHAR(9) NOT NULL,* *PRIMARY KEY(NUM_FACTURA),* *FOREIGN KEY FK_FACT_DNI_CLIENTES (DNI_CLIENTE)* *REFERENCES CLIENTES (DNI)* *);*	*CREATE TABLE FACTURAS (* *NUM_FACTURA INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY, DNI_CLIENTE CHAR(9) NOT NULL,* *CONSTRAINT PK_FACTURAS PRIMARY KEY(NUM_FACTURA),* *CONSTRAINT FK_FACT_DNI_CLIENTES FOREIGN KEY (DNI_CLIENTE) REFERENCES CLIENTES (DNI)* *);*
CREATE TABLE LINEAS_FACTURA *(* *NUM_FACTURA INTEGER NOT NULL, LINEA_FACTURA SMALLINT NOT NULL, CONCEPTO VARCHAR(32) NOT NULL, CANTIDAD SMALLINT NOT NULL,* *PRIMARY KEY(NUM_FACTURA, LINEA_FACTURA),* *FOREIGN KEY FK_LINEAFACT_NUMFACTURA (NUM_FACTURA) REFERENCES FACTURAS (NUM_FACTURA)* *);*	***CREATE TABLE LINEAS_FACTURA*** *(* *NUM_FACTURA INTEGER NOT NULL, LINEA_FACTURA SHORTINTEGER NOT NULL, CONCEPTO VARCHAR(32) NOT NULL, CANTIDAD SHORTINTEGER NOT NULL,* *CONSTRAINT PK_LINEAS_FACTURA* *PRIMARY KEY(NUM_FACTURA, LINEA_FACTURA),* *CONSTRAINT FK_LINEAFACT_NUM_FACTURA* *FOREIGN KEY (NUM_FACTURA) REFERENCES FACTURAS (NUM_FACTURA)* *);*

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- Para recuperar con el valor asignado a la clave autogenerada, usaremos métodos de *Statement* y de *Connection*

Proviene de	Método	Funcionalidad
Statement	int *executeUpdate*(String sql, int autoGeneratedKeys)	autoGeneratedKeys puede ser: Statement.RETURN_GENERATED_KEYS (*) o Statement.NO_GENERATED_KEYS
Statement	*ResultSet getGeneratedKeys()*	Devuelve un *ResultSet* con los valores de las claves autogeneradas. También para *PreparedStatement*
Connection	*PreparedStatement prepareStatement (String sql, int autoGeneratedKeys)*	(*)Prepara una sentencia para poder recuperar valores para claves autogeneradas, hay que indicar el valor *PreparedStatement.RETURN_GENERATED_KEYS* para *autoGeneratedKeys.*

3.5 – SENTENCIAS PREPARADAS, TRANSACCIONES Y CLAVES AUTOGENERADAS

- **Ejemplo donde se crea una factura con varias líneas.**
- Una vez insertada una nueva factura en FACTURAS inserta sus líneas en LINEAS_FACTURAS, e indica como valor para el número de la factura el valor de la clave autogenerada que se acaba de crear.
- La creación de la factura y sus líneas se realiza conjuntamente como una transacción.

```
// Se omite declaración de variables para los datos de conexión
try (
        Connection c = DriverManager.getConnection(urlConnection, user, pwd)) {
    try (
        PreparedStatement sInsertFact = c.prepareStatement("INSERT INTO "
            + "FACTURAS(DNI_CLIENTE) VALUES (?)",
                PreparedStatement.RETURN_GENERATED_KEYS);
        PreparedStatement sInsertLineaFact = c.prepareStatement("INSERT INTO "
            + "LINEAS_FACTURA(NUM_FACTURA,LINEA_FACTURA,CONCEPTO,CANTIDAD) "
                + "VALUES (?,?,?,?);");) {

    c.setAutoCommit(false);

    int i = 1;
    sInsertFact.setString(i++, "78901234X");
    sInsertFact.executeUpdate();
    ResultSet rs=sInsertFact.getGeneratedKeys();
    rs.next();
    int numFact=rs.getInt(1);

    int lineaFact = 1;
    i = 1;
    sInsertLineaFact.setInt(i++, numFact);
    sInsertLineaFact.setInt(i++, lineaFact++);
    sInsertLineaFact.setString(i++, "PICO");
    sInsertLineaFact.setInt(i++, 15);
    sInsertLineaFact.executeUpdate();

    i = 1;
    sInsertLineaFact.setInt(i++, numFact);
    sInsertLineaFact.setInt(i++, lineaFact++);
    sInsertLineaFact.setString(i++, "PALA");
    sInsertLineaFact.setInt(i++, 25);
    sInsertLineaFact.executeUpdate();
    c.commit();

} catch (SQLException e) {
```

3.6 – Llamadas a procedimientos y funciones almacenadas

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

- **El estándar SQL incluye extensiones prodedurales del lenguaje (If, bucles, …)**
- Los **procedimientos** y **funciones almacenados** son bloques de código escritos en este leguaje que pueden tener parámetros de entrada, salida o entrada-salida.
- En el caso de las funciones pueden devolver un valor.
- Son análogos a los procedimientos y funciones de programación y pueden ser invocados tanto desde el intérprete del SGBD como desde JDBC.
- JDBD proporciona una única interfaz: *CallableStatement*.

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

- El **procedimiento** que se va a utilizar se puede crear de la siguiente manera en el intérprete de SQL.
- Al procedimiento se le pasa un DNI y devuelve un conjunto de filas con DNI y APELLIDOS de los clientes por orden alfabético de APELLIDOS, hasta el valor de APELLIDOS para el cliente con el DNI proporcionado. Al valor pasado en el parámetro de entrada y salida *inout_long* se le suma la longitud del valor de APELLIDOS para el cliente con el DNI pasado en *in _dni*.

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

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

- La secuencia de pasos en JDBC es igual:
  - Obtener un CallableStatement con el método prepareCall(String patrón_llamada) de Connection.

Tipo	Con parámetros	Sin parámetros
Procedimiento	{ call procedimiento(?, ?, …) }	{ call procedimiento}
Función	{ ? = call función(?, ?, …) }	{ call función }

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

- Métodos de *CallableStatement* para uso de procedimientos y funciones almacenados.

Método	Funcionalidad
setXXX(int pos, YYYY valor)	De manera similar a *PreparedStatement*, se utiliza para asignar un valor de diversos tipos a un parámetro determinado, identificado por su posición, siendo 1 el primero.
void *registerOutParameter*(int pos, int sqlType)	Registra un parámetro de salida para poder obtener un valor devuelto por él.
getXXX(int pos,)	Obtiene el valor de un parámetro de salida.
ResultSet *getResultSet*()	Obtiene el *ResultSet* resultado del procedimiento o función.

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

```
// Se omite declaración de variables para los datos de conexión
    try (
            Connection c = DriverManager.getConnection(urlConnection, user, pwd);
            CallableStatement s = c.prepareCall("{call listado_parcial_clientes(?,?)}")) {

        s.setString(1,"78901234X");
        s.setInt(2,0);
        s.registerOutParameter(2,java.sql.Types.INTEGER);

        s.execute();

        ResultSet rs = s.getResultSet();

        int inout_long = s.getInt(2);
        System.out.println("=> inout_long: "+inout_long);
        int nCli = 0;
        while (rs.next()){
            System.out.println("[" + (++nCli) + "[");
            System.out.println("DNI: " + rs.getString("DNI"));
            System.out.println("Apellidos: " + rs.getString("APELLIDOS"));
        }

      } catch (SQLException e) {
        muestraErrorSQL(e);
      } catch (Exception e) {
        e.printStackTrace(System.err);
      }
  }

}
```

- Ejemplo de programa que llama al procedimiento almacenado anterior.

3.6 – PROCEDIMIENTOS Y FUNCIONES ALMACENADAS

ACTIVIDADES PROPUESTAS.

a) Crea una función almacenada con MySQL a la que se le pase el DNI de un cliente y que devuelva sus apellidos. Crea un programa en Java que realice una llamada a ella utilizando JDBC y escriba el valor devuelto por la función. Será necesario consultar documentación o investigar cómo crear funciones almacenadas en la base de datos utilizada, pero se hará de manera muy similar a un procedimiento almacenado.

RESUMEN

- ✓ Un conector es una API que permite acceder, desde programas de aplicación, a los datos almacenados en SGBD.
- ✓ Cada BBDD es distinta. Los conectores suelen ser para un tipo particular de BBDD, pero tienen características comunes. Proporcionan métodos de conexión a una BD, permiten ejecutar sentencias en el lenguaje de la BD, proporcionan iteradores para acceder a los resultados de las consultas. Si la BD soporta transacciones, también proporcionará métodos para gestionarlas.
- ✓ La API JDBC de Java permite trabajar con BBDD Relacionales. Es más una arquitectura que una API, pues proporciona una interfaz común para los programas de aplicaciones y el acceso a BBDD y requiere driver específico para las BBDD en particular. El driver hace posible la conexión a una BD particular usando sus propios protocolos de red e interfaces de bajo nivel y proporciona la implementación para esa base de datos de todas las interfaces recogidas en la especificación JDBC.

RESUMEN

- ✓ JDBC permite ejecutar cualquier sentencia SQL mediante *execute*, *executeUpdate* y *executeQuery* y para las consultas obtener los resultados en un *ResultSet*.
- ✓ En consultas con variables, deben usarse sentencias preparadas o *PreparedStatement* por seguridad (evitar ataques de inyección SQL). También por eficiencia.
- ✓ JDBC permite el uso de transacciones de forma sencilla utilizando métodos estándares de JDBC e independiente de la BD.
- ✓ JDBC permite invocar procedimientos y funciones almacenados de manera independiente de la BD mediante el uso de métodos estándares de JDBC.

Acceso a Datos

FIN DE LA UNIDAD

GRACIAS