# Capa de Gestión de Datos

- Persistencia – Bases de Datos - JDBC
- Persistencia - Ficheros
- Persistencia - Serialización

# Persistencia

- Las instancias y objetos de las clases sólo existen mientras se ejecuta el programa Java
- Persistencia: almacenamiento de los objetos en memoria secundaria (disco)
- Ficheros
  - No es conveniente si existen actualizaciones concurrentes
  - Java ofrece dos posibilidades:
    - Guardar los valores de los objetos en ficheros
    - Mecanismos de serialización
- SGBD
  - El SGBD y los programadores gestionan la concurrencia
  - JDBC
  - SGBD OO

# JDBC

- JDBC (Java DataBase Conectivity) es un API de java que sirve para:
  - Establecer conexiones con BDs
  - Enviar sentencias SQL a esas Bds
  - Procesar los resultados

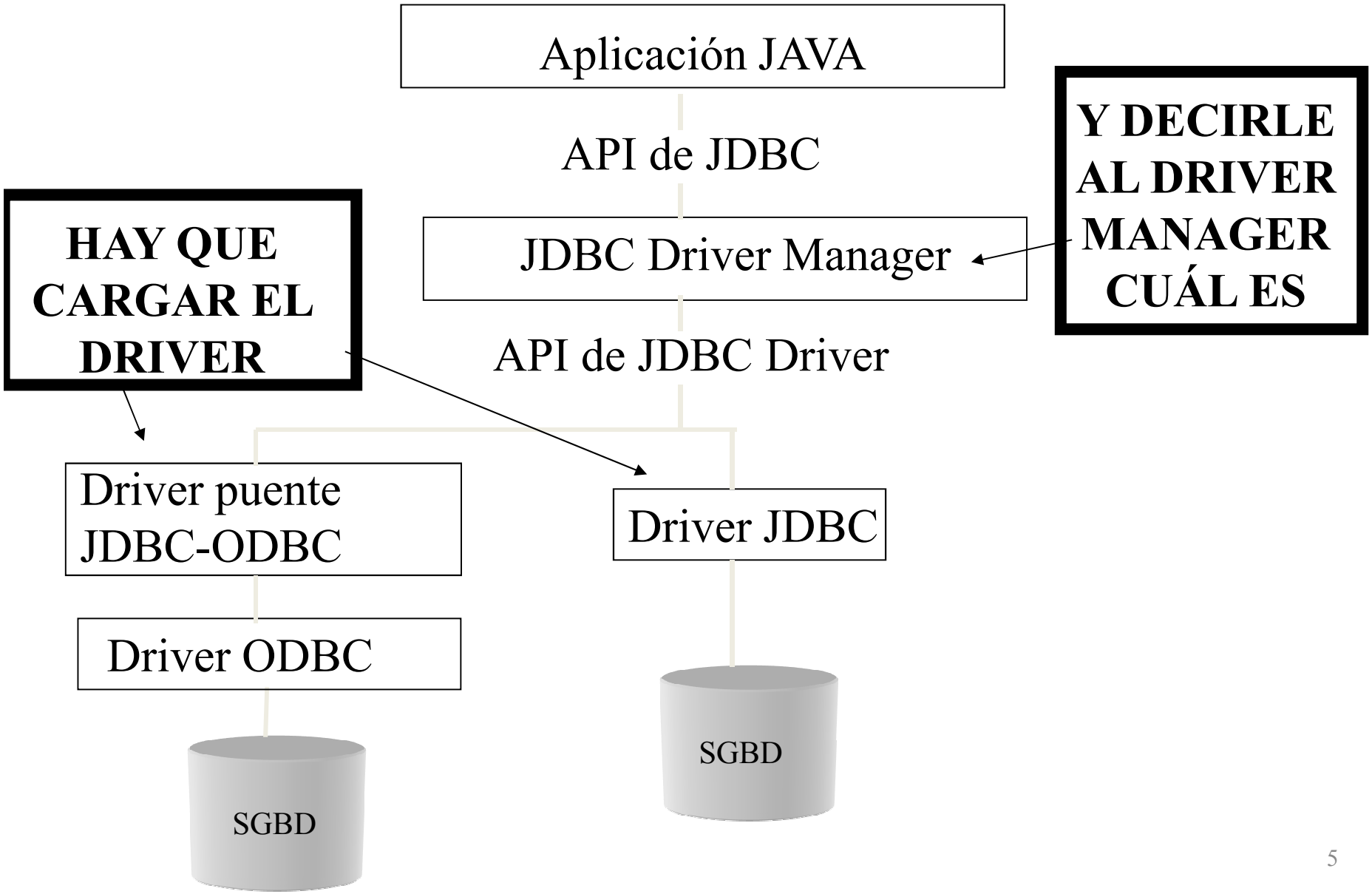
- Conjunto de clases y métodos que se encuentran en java.sql

- Los vendedores de SGBD deben proporcionar los controladores ('drivers') para JDBC, o la forma de "interpretar" llamadas JDBC
  - Hay muchos: http://industry.java.sun.com/products/jdbc/drivers

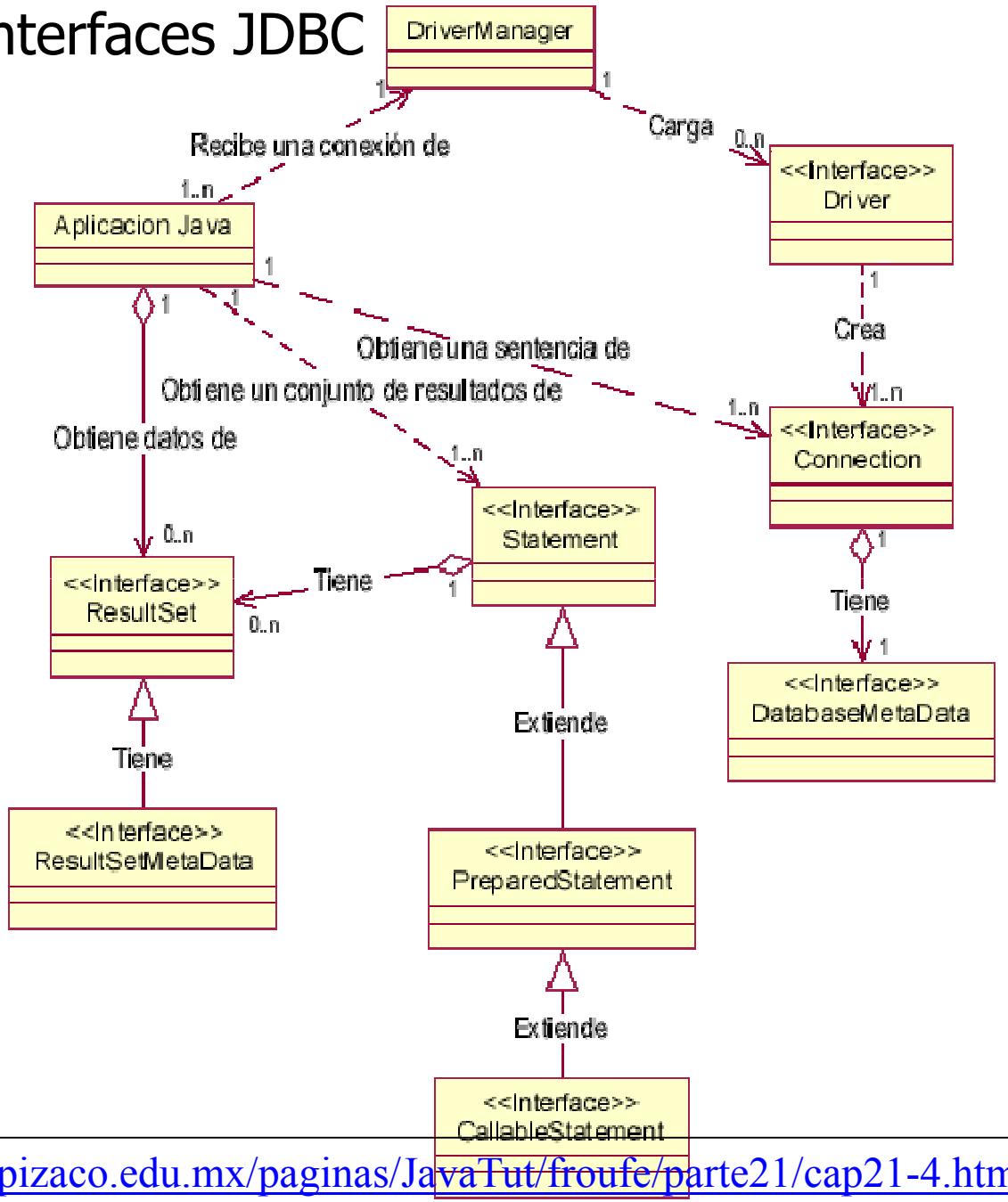
# JDBC y ODBC

- JDBC ofrece igual funcionalidad que ODBC (Open DataBase Connectivity) de Microsoft
- ODBC está escrito en C
- La gran mayoría de SGBD disponen de controladores ODBC
- JDK proporciona un puente JDBC-ODBC que permite convertir llamadas JDBC a ODBC y poder acceder así mediante JDBC a BDs que ya tienen un controlador ODBC
  - En caso de hacerlo así, hay que registrar la BD correspondiente para que pueda ser usada con ODBC

# Arquitectura JDBC



# Las Clases e Interfaces JDBC

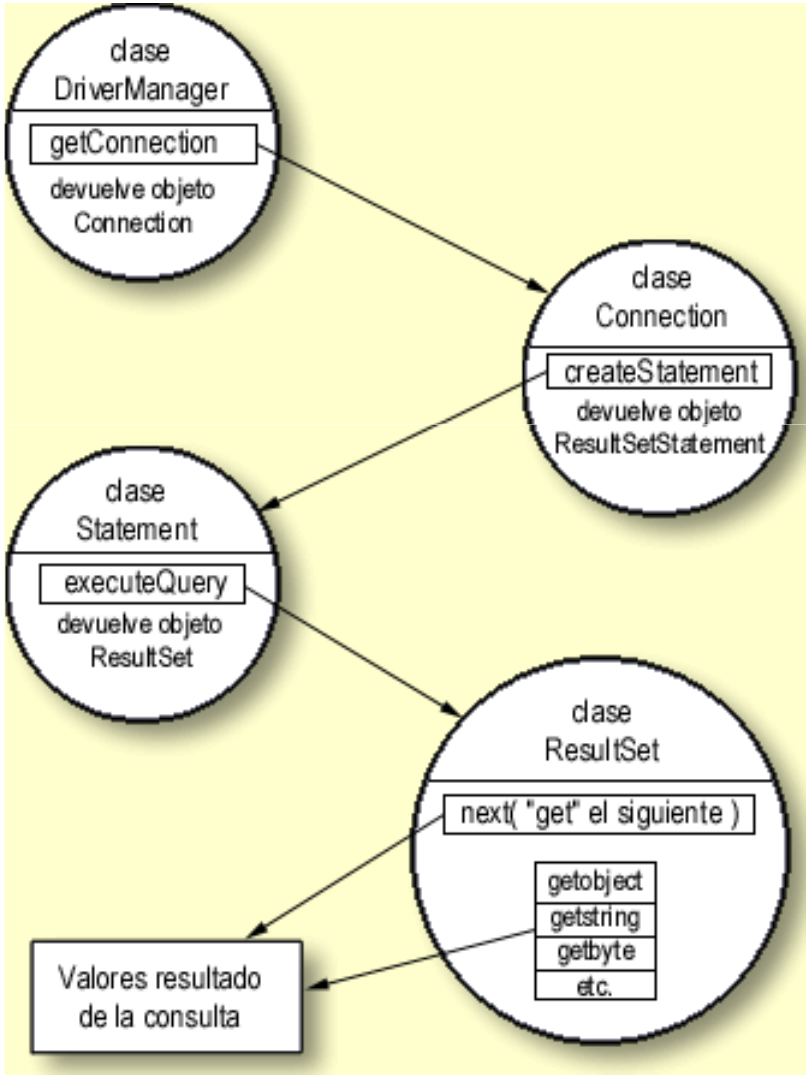



http://www.itapizaco.edu.mx/paginas/JavaTut/froufe/parte21/cap21-4.html

# Las clases e interfaces JDBC

| | |
|---|---|
| **Driver** | Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto |
| **DriverManager** | Permite gestionar todos los drivers instalados en el sistema |
| **DriverPropertyInfo** | Proporciona diversa información acerca de un driver |
| **Connection** | Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión a más de una base de datos |
| **DatabaseMetadata** | Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc. |
| **Statement** | Permite ejecutar sentencias SQL sin parámetros |
| **PreparedStatement** | Permite ejecutar sentencias SQL con parámetros de entrada |
| **CallableStatement** | Permite ejecutar sentencias SQL con parámetros de entrada y salida, típicamente procedimientos almacenados |
| **ResultSet** | Contiene las filas o registros obtenidos al ejecutar un SELECT |
| **ResultSetMetadata** | Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc. |

# Las clases e interfaces JDBC
# Pasos para realizar una consulta




**1. Cargar driver/controlador**
**Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );**

**2. Establecer conexión**
**Connection conexion = DriverManager.getConnection( "jdbc:odbc:Tutorial","","" );**

**3. Crear un Statement**
**Statement sentencia = conexion.createStatement();**

**4. Invocar a la base de datos: SQL**
**Resultset rs=sentencia.executeQuery(“SELECT ….”);**

**5. Procesar resultados**
**while (rs.next) {**
**rs.**
**}**

# 1. Cargar el Driver/Controlador

## Utilizando *Class.forName*

- Para cargar un controlador se puede usar el método forName() de la clase Class (carga clases Java)
  - **`Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`**
  - carga el puente JDBC-ODBC
  - **`Class.forName("org.gjt.mm.mysql.Driver");`**
  - carga el Driver JDBC para trabajar con el SGBD mySQL
- La clase java.lang.Class permite crear instancias de clases en tiempo de ejecución:

```
Object o = Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

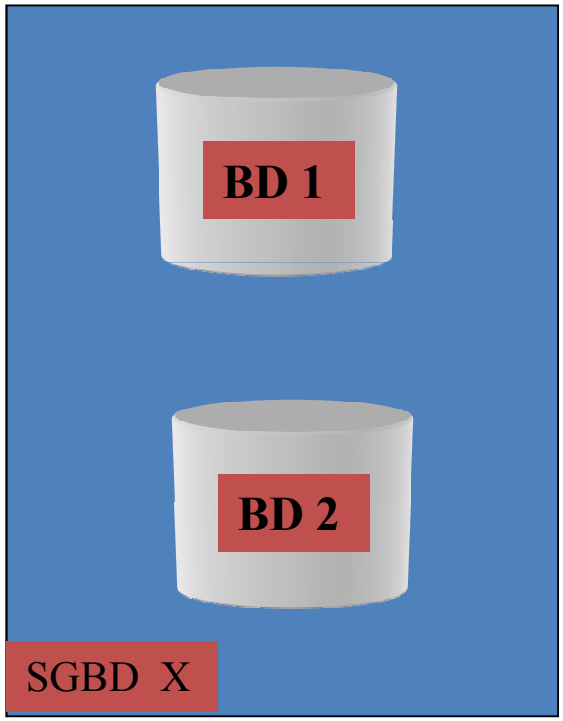
  - Crea una instancia de la clase org.gjt.mm.mm.mysql.Driver y la asigna al objeto o

# 1. Cargar el Driver/Controlador

## Utilizando la clase *DriverManager*

- Sirve para registrar los controladores o drivers
  - **`DriverManager.registerDriver(new sun.jdbc.odbc.JdbcOdbcDriver());`**
- No es necesario hacerlo si se ha cargado el Driver con Class.forName

# Ejemplo


**Ej: Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");**

# 2. Establecer una conexión

## El método *DriveManager.getConnection()*

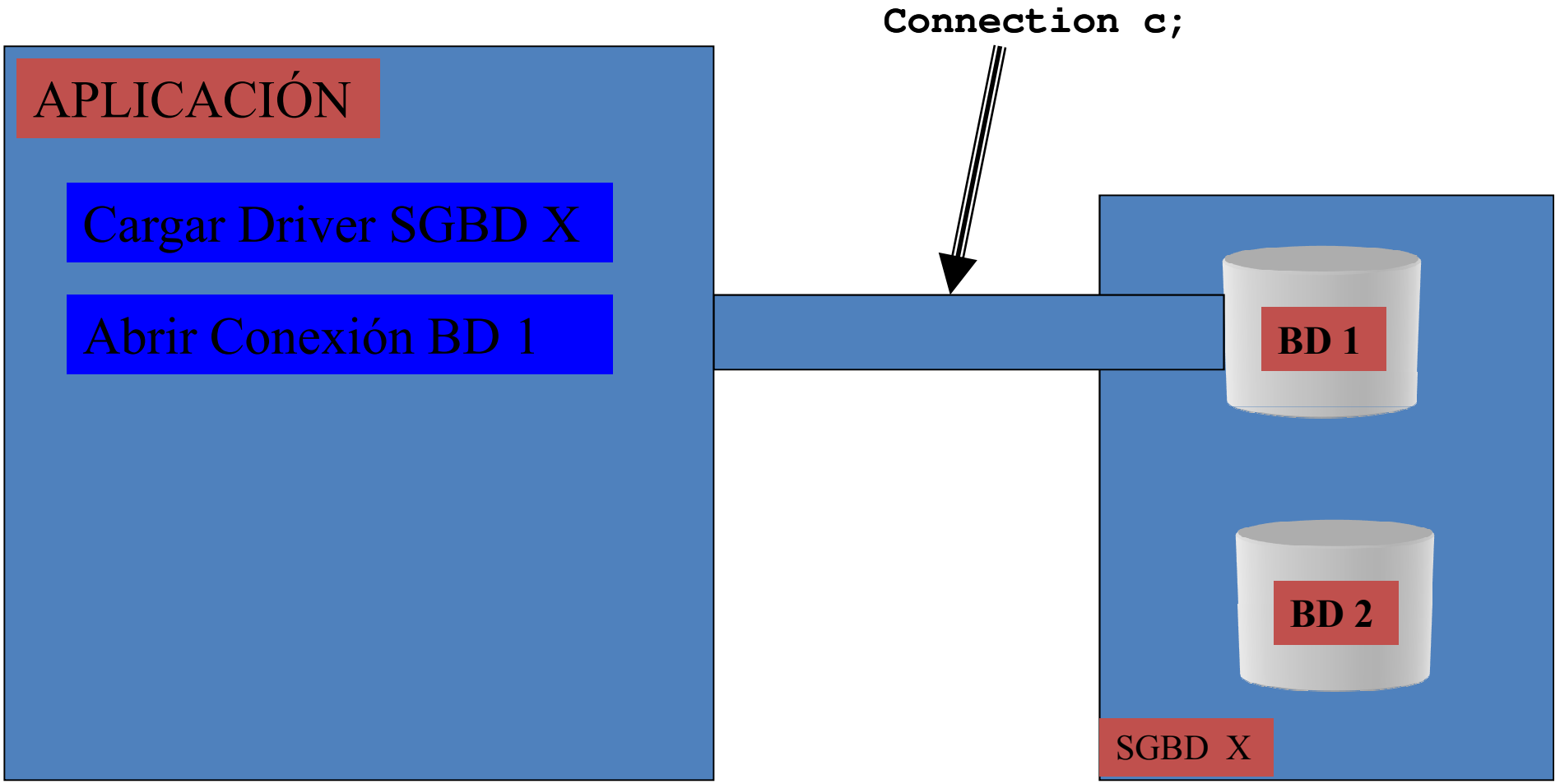
- La clase DriverManager es la encargada de establecer conexiones con las BDs
  - **`Connection c = DriverManager.getConnection(`**
    **`String dir, String usu, String clave)`**
  - **dir** identifica a la BD. Dependiendo del SGBD usado tendrá una forma u otra
  - **`jdbc:odbc//servidor:puerto/base de datos`**
  - **usu** y **clave** son nombre de usuario y clave, si hay
- Como resultado obtenemos una conexión con la Base de Datos a través de un objeto **Connection** (puede haber mas de una conexión)

# 2. Establecer una conexión
## Configuración de las transacciones

- Por defecto, toda sentencia SQL enviada a través de una conexión termina con un *commit* si tiene éxito.
- Si se quiere que varias sentencias SQL formen una transacción, se puede hacer así:

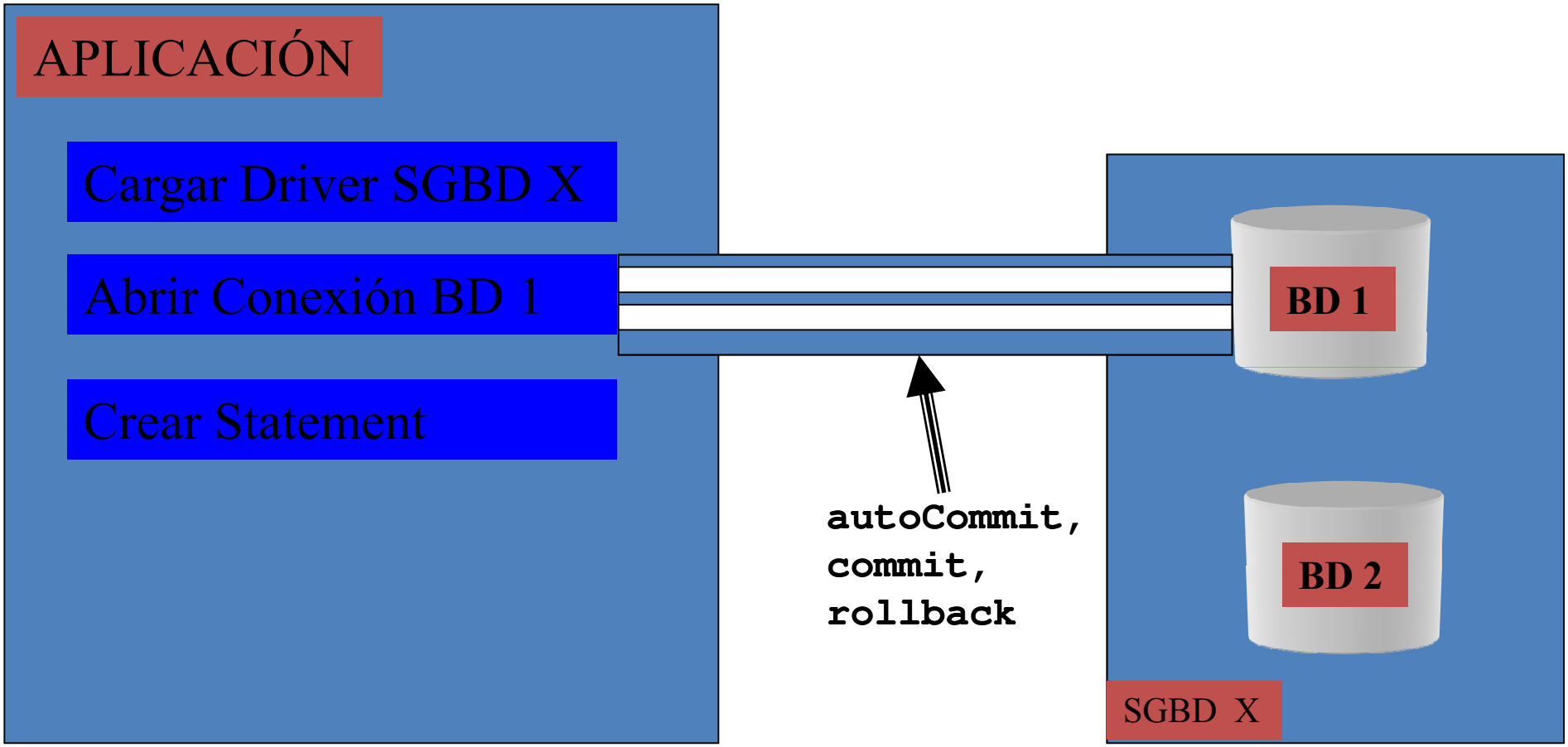
```
c.setAutoCommit(false);
// Enviar las sentencias SQL que forman
// la transacción
Ejecutar c.commit();
o bien c.rollback();
```

# Ejemplo




Ej: `Connection c = DriverManager.getConnection("jdbc:odbc:bd1");`

# Ejemplo


APLICACIÓN
Cargar Driver SGBD X
Abrir Conexión BD 1
Crear Statement
autoCommit,
commit,
rollback
BD 1
BD 2
SGBD X

# 3. Crear un Statement
## La interfaz *Statement*

- Una vez que tenemos una conexión abierta, se pueden lanzar sentencias SQL desde el programa Java.
- Para ello hay que crear objetos "Statement"
  - `// c contiene un objeto Connection`
  - `Statement s = c.createStatement();`
- Las conexiones hay que cerrarlas cuando no se necesitan (no queremos enviar más sentencias SQL a la BD) para liberar recursos

  `c.close();`

# 3. Crear un Statement
## La interfaz *Statement*

- Se utiliza para realizar invocaciones a la BD

  **s.executeQuery(String sql);**

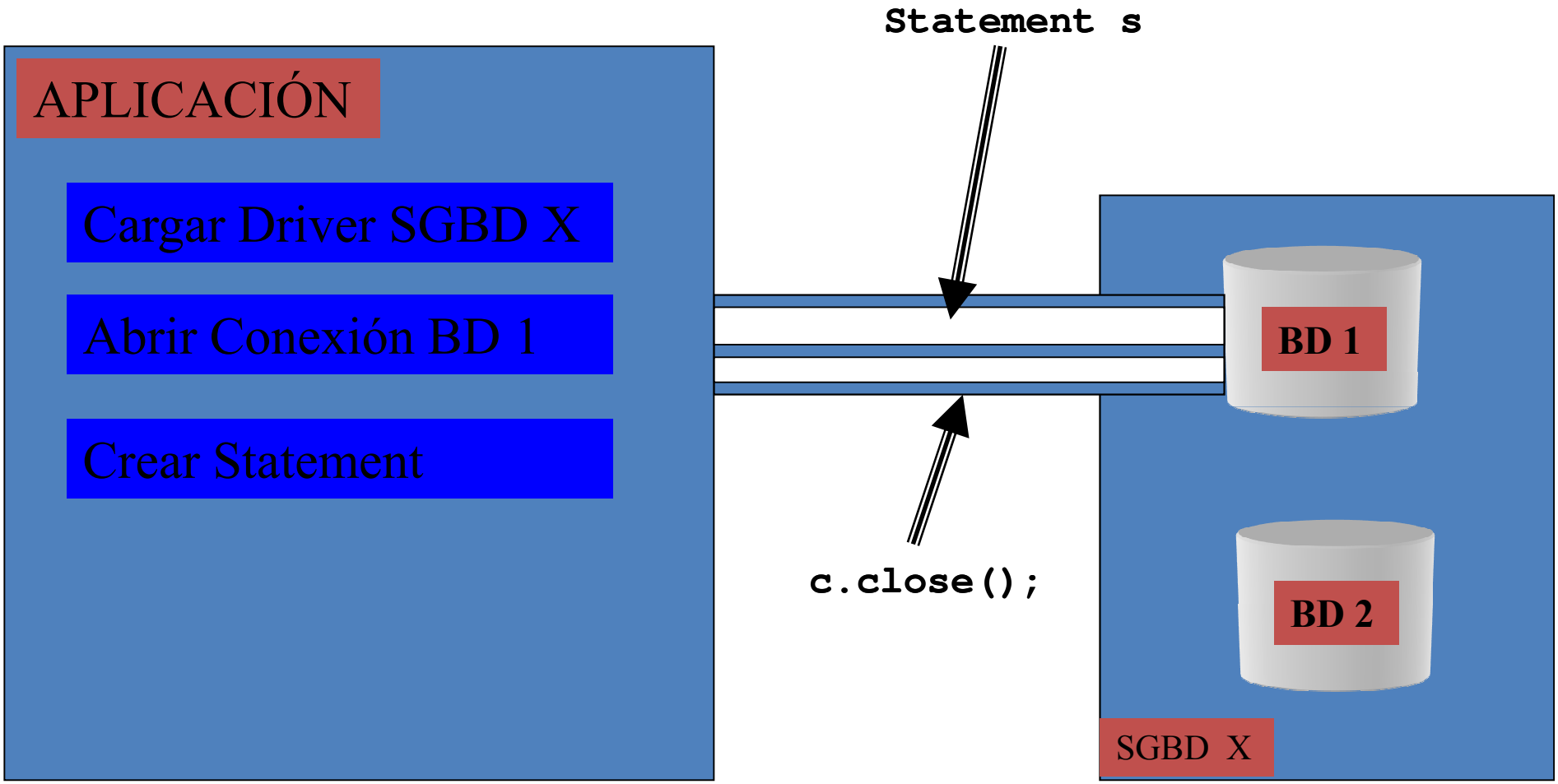
- Se puede limitar el número máximo de tuplas que queremos que nos devuelva el SGBD:

  **s.setMaxRows(maxTuplas);**

- Se puede limitar el número máximo de segundos que queremos que espere el Driver:

```
s.setQueryTimeout(maxSegundos);
```

# Ejemplo




Ej: `Statement s = c.createStatement();`

# 4. Invocar a la BD

Sirve para enviar sentencias SQL a la BD

**//s contiene un objeto Statementnt**

**i = s.executeUpdate(String sql);**

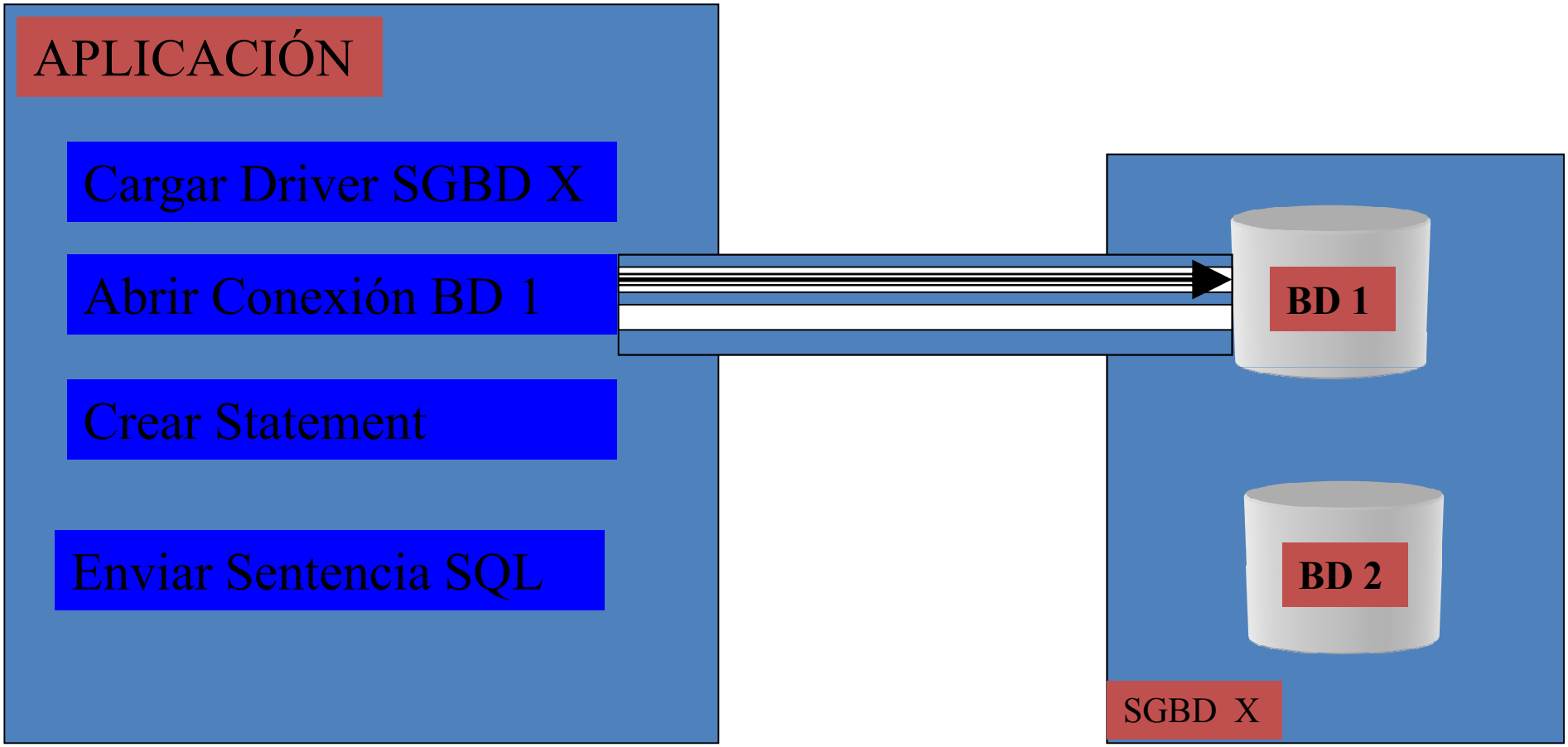
ejecuta una sentencia SQL INSERT, UPDATE o DELETE y devuelve el número de tuplas afectadas

**ResultSet r = s.executeQuery(String sql);**

ejecuta una sentencia SQL SELECT y devuelve el resultado en un objeto ResultSet

**NOTA: sólo puede haber un ResultSet "abierto" sobre un objeto "Statement"**
**CONCLUSIÓN: si queremos trabajar con dos preguntas SQL a la vez se necesita definir dos *Statement* diferentes**

# Ejemplo




Ej: `s.executeQuery(sentSQL);`

`s.executeUpdate(sentSQL);`

# 5. Procesar los resultados

## La interfaz *ResultSet*

- Sirve para trabajar con las respuestas a las preguntas SQL realizadas

```
boolean b = r.next();

// r es un objeto de tipo ResultSet
```

se posiciona en la siguiente tupla del resultado; devuelve true si hay o false si se ha llegado al final

- Existen métodos **get** que devuelven el valor de la tupla actual correspondiente a un atributo de la tabla respuesta.
  - El atributo se puede identificar por el nombre dado en la parte SELECT de la pregunta o por su posición en la misma
  - El tipo del atributo puede ser conocido o no

# 5. Procesar los resultados

## La interfaz *ResultSet*

```
  String s = r.getString(numColumna);
  String s = r.getString(nombreColumna);
  int i = r.getInt(numColumna);
  int i = r.getInt(nombreColumna);
  boolean b = r.getBoolean(numColumna);
  boolean b = r.getBoolean(nombreColumna);
// si conocemos el tipo del atributo

  Object o = r.getObject(numColumna);
  Object o = r.getObject(nombreColumna);
// si no conocemos el tipo del atributo
```

# 5. Procesar los resultados

## La interfaz *ResultSet*

```
ResultSet rs = s.executeQuery("SELECT * FROM BILLETES");
```

BILLETES : Tabla

| | NUMERO | ESTADO | NOMBRE |
|---|---|---|---|
| ▶ | 0 | OCUPADO | P. Salanueva |
| | 1 | LIBRE | |
| | 2 | LIBRE | |
| * | | | |

```
rs.next()
```

⟶ `true`

```
int i = rs.getInt(1);
```

⟶ `0`

```
String est = rs.getString(2);
```

⟶ `OCUPADO`

# 5. Procesar los resultados

## La interfaz *ResultSet*

```
ResultSet rs = s.executeQuery("SELECT * FROM BILLETES");
```

BILLETES : Tabla

| | NUMERO | ESTADO | NOMBRE |
|---|---|---|---|
| ▶ | 0 | OCUPADO | P. Salanueva |
| | 1 | LIBRE | |
| | 2 | LIBRE | |
| * | | | |

```
rs.next()
```

⟶ `true`

```
int i = rs.getInt(1);
```

⟶ `1`

```
String est = rs.getString(2);
```

⟶ `LIBRE`

# 5. Procesar los resultados

## La interfaz *ResultSet*

```
ResultSet rs = s.executeQuery("SELECT * FROM BILLETES");
```

BILLETES : Tabla

| | NUMERO | ESTADO | NOMBRE |
|---|---|---|---|
| ▶ | 0 | OCUPADO | P. Salanueva |
| | 1 | LIBRE | |
| | 2 | LIBRE | |
| * | | | |

```
rs.next()
```

⟶ `true`

```
int i = rs.getInt(1);
```

⟶ `2`

```
String est = rs.getString(2);
```

⟶ `LIBRE`

# 5. Procesar los resultados

## La interfaz *ResultSet*

```
ResultSet rs = s.executeQuery("SELECT * FROM BILLETES");
```

BILLETES : Tabla

| | NUMERO | ESTADO | NOMBRE |
|---|---|---|---|
| ▶ | 0 | OCUPADO | P. Salanueva |
| | 1 | LIBRE | |
| | 2 | LIBRE | |
| * | | | |

```
rs.next()
```

⟶ **false**

# Usando sentencias con parámetros

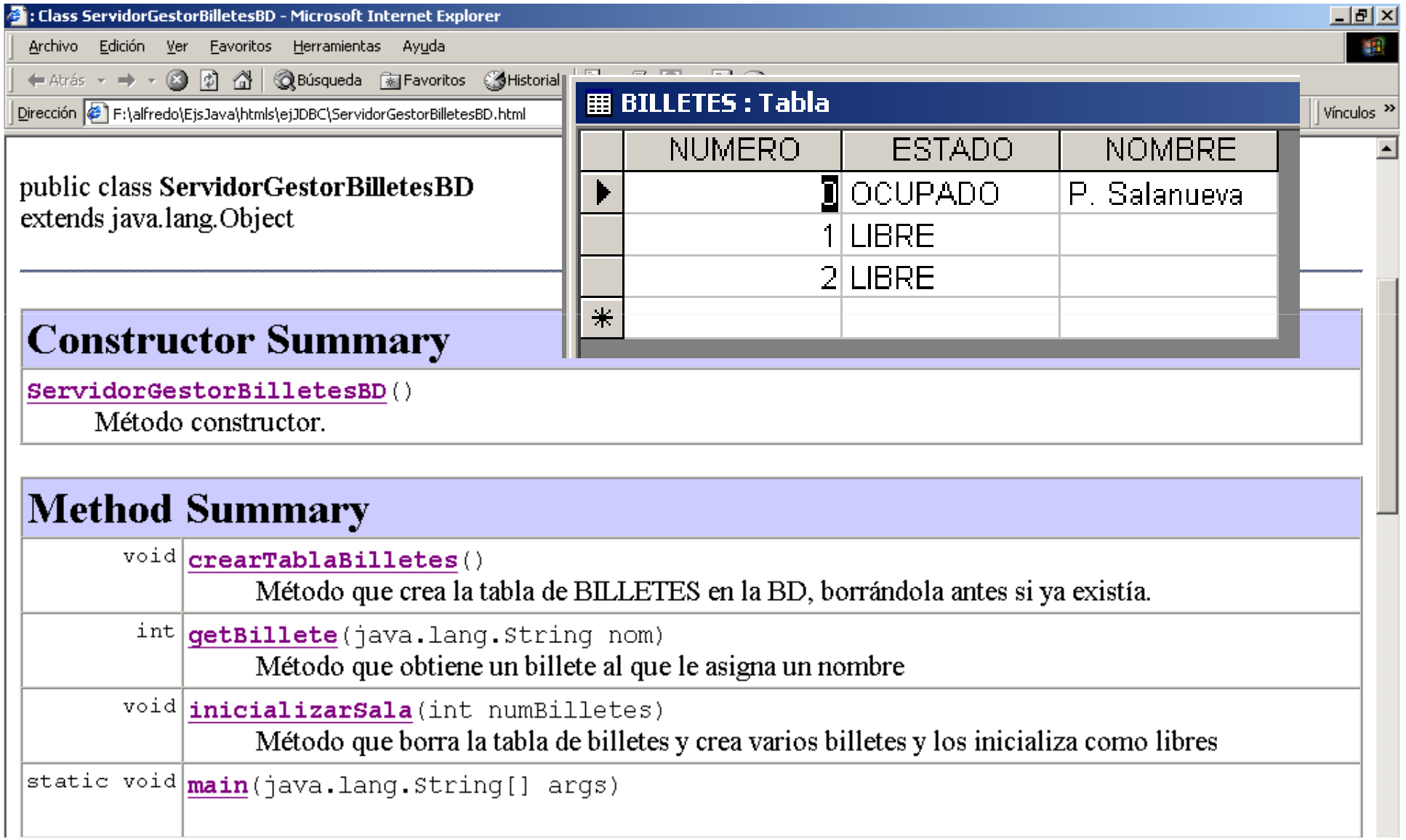
*El método prepareStatement()*

- Algunas sentencias SQL se repiten varias veces cambiando sólo algunos valores.
  - Se pueden parametrizar

```
Sea c un objeto de tipo Connection
PreparedStatement s = c.prepareStatement("SELECT
  NOMBRE FROM PERSONAS WHERE CIUDAD=? AND
  EDAD>?");
s.setString(1,"SS"); // Poner SS en 1er parámetro
s.setInt(2,25); // Poner 25 en 2º parámetro
ResultSet r = s.executeQuery(); // ...
// Se puede reejecutar s.setString(1,"Bi");..
```

# Ejemplo ServidorGestorBilletesBD

*La API*



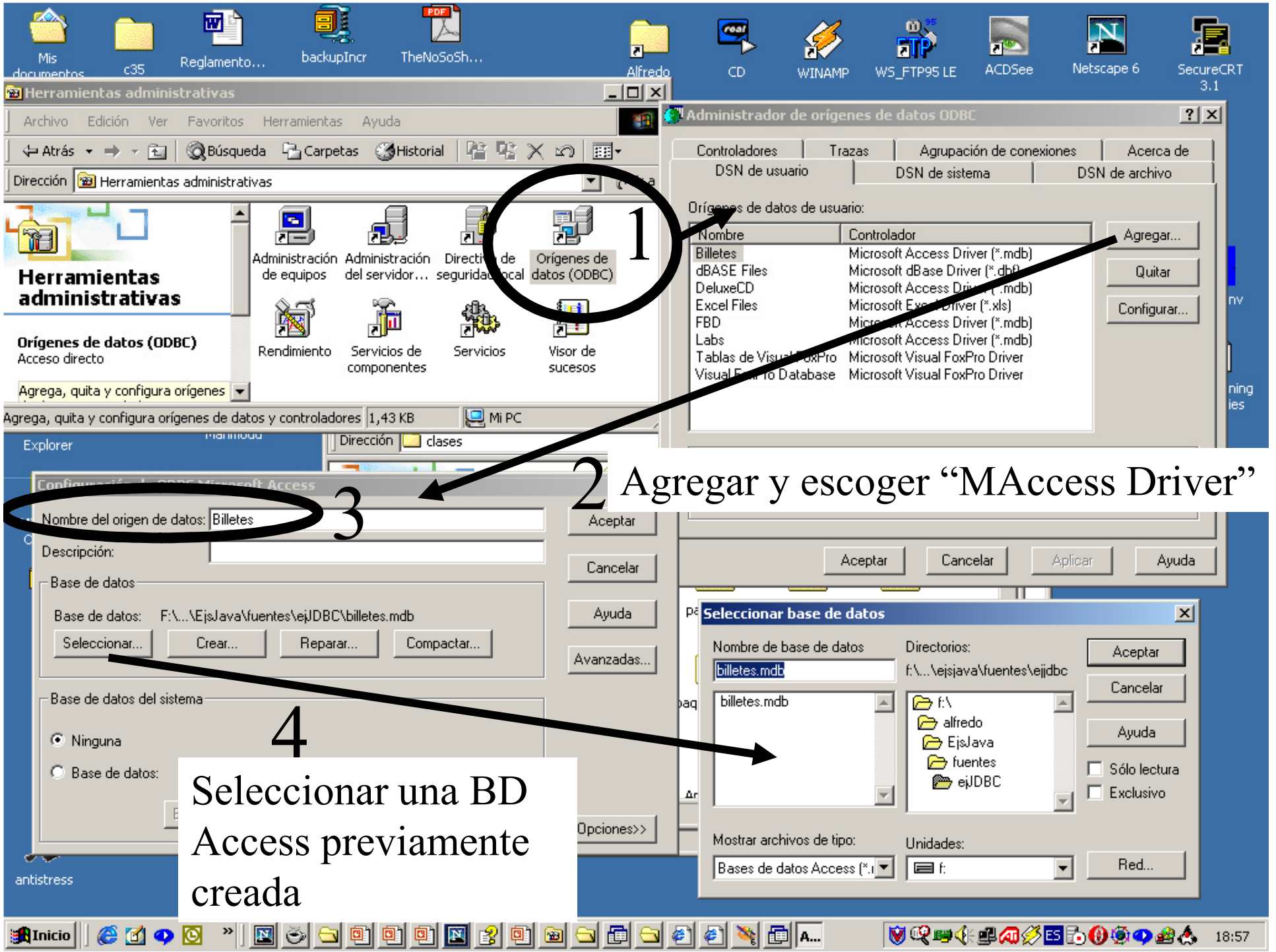
Ing

```java
// ServidorGestorBilletesBD.java
package ejJDBC;
import java.sql.*;

public class ServidorGestorBilletesBD
{
        private static  Connection conexion;
        private static  Statement sentencia;
/**
 * Método constructor. Carga el puente JDBC-ODBC, crea conexión
 * con la fuente de datos Billetes y crea la sentencia JDBC.
 * Además pone por defecto que debe hacerse commit cuando sea necesario.
*/
    public ServidorGestorBilletesBD() {
    ...       }
/**
 * Método que crea la tabla de BILLETES en la BD, borrándola antes si ya existía.
*/
    public void crearTablaBilletes() {
    ...        }
/**
 * Método que borra la tabla de billetes y crea varios billetes y los inicializa cor
 * @param numBilletes Número de billetes que crea
*/
    public void inicializarSala(int numBilletes) {
    ...             }
/**
 * Método que obtiene un billete al que le asigna un nombre
 * @param nom Nombre a asignar al billete
 * @return El nombre del billete, -1 si no hay billetes o -2 si hay otros problemas
*/
    public int getBillete(String nom) {
    ... }
}
```

# Ejemplo ServidorGestorBilletesBD

*El método constructor. Carga driver y conexión*

```
/**
* Método constructor. Carga el puente JDBC-ODBC,
* crea conexión con la fuente de datos Billetes
* y crea la sentencia JDBC. Además pone por defecto
* que debe hacerse commit cuando sea necesario.
*/
public ServidorGestorBilletesBD() {
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      conexion=DriverManager.getConnection("jdbc:odbc:Billetes");
      conexion.setAutoCommit(false);
      // Hay que hacer commit "manualmente"

      sentencia=conexion.createStatement();
    }
    catch(Exception e)
      { System.out.println("Se ha producido un error:"+e.toString());}
    }
```


Herramientas administrativas
Orígenes de datos (ODBC)
Acceso directo
Agrega, quita y configura orígenes de datos y controladores
Administrador de orígenes de datos ODBC
Controladores
Trazas
Agrupación de conexiones
Acerca de
DSN de usuario
DSN de sistema
DSN de archivo
Orígenes de datos de usuario:
Nombre
Controlador
Billetes
Microsoft Access Driver (*.mdb)
dBASE Files
Microsoft dBase Driver (*.dbf)
Excel Files
Microsoft Visual FoxPro Driver
Agregar...
Quitar
Configurar...
1
2 Agregar y escoger “MAccess Driver”
3
Nombre del origen de datos: Billetes
Descripción:
Base de datos
Base de datos: F:\...\EjsJava\fuentes\ejJDBC\billetes.mdb
Seleccionar...
Crear...
Reparar...
Compactar...
Base de datos del sistema
Ninguna
Base de datos:
Aceptar
Cancelar
Ayuda
Avanzadas...
Opciones>>
Seleccionar base de datos
Nombre de base de datos
billetes.mdb
Directorios:
f:\...\ejsjava\fuentes\ejjdbc
f:\
alfredo
EjsJava
fuentes
ejJDBC
Sólo lectura
Exclusivo
Mostrar archivos de tipo:
Bases de datos Access (*.
Unidades:
f:
Red...
4
Seleccionar una BD Access previamente creada
Inicio
18:57

# Ejemplo ServidorGestorBilletesBD

*Crea la tabla billetes*

```
/**
* Método que crea la tabla de BILLETES en la BD,
* borrándola antes si ya existía.
*/
public void crearTablaBilletes() {
      try{
       sentencia.executeUpdate("drop table BILLETES");
      } catch (Exception e)
         {System.out.println("La tabla BILLETES no existía");}
      try{
       sentencia.executeUpdate("create table BILLETES " +
                               "(NUMERO int, " +
                                "ESTADO varchar(8), " +
                                "NOMBRE varchar(25))");
      } catch (Exception e)
       { System.out.println("Se ha producido un error:"+e.toString());}
    }
```

# Ejemplo ServidorGestorBilletesBD

*Inicializa la tabla billetes*

```
/**
* Método que borra la tabla de billetes y crea varios billetes
* y los inicializa como libres
* @param numBilletes Número de billetes que crea
*/
public void inicializarSala(int numBilletes) {
 try{
  sentencia.executeUpdate("delete from billetes");
  for(int i=0;i<numBilletes;i++)
   sentencia.executeUpdate("insert into billetes (numero) values ("+i+")");
  sentencia.executeUpdate("update BILLETES set ESTADO='LIBRE'");
  conexion.commit();
 } catch (Exception e)
  { System.out.println("Se ha producido un error:"+e.toString());}
}
```

# Ejemplo ServidorGestorBilletesBD

*Inicializa la tabla billetes*

```
for(int i=0;i<numBilletes;i++)
 sentencia.executeUpdate("insert into billetes (numero) values ("+i+")");
```

EL CÓDIGO ANTERIOR PODRÍA SUSTITUIRSE POR EL SIGUIENTE, DONDE SE USA UNA LLAMADA A UNA SENTENCIA SQL PARAMETRIZADA

```
PreparedStatement s1=
  conexion.prepareStatement("insert into billetes (numero) values (?)");
for (int i=0;i<numBilletes;i++) {
  s1.setInt(1,i);
  s1.executeUpdate(); }
```

# Ejemplo ServidorGestorBilletesBD

*Lógica de negocio. getBillete(String nom)*

```
/**
* Método que obtiene un billete al que le asigna un nombre
* @param nom Nombre a asignar al billete
* @return El nombre del billete, -1 si no hay billetes o -2 si hay otros problemas
*/
public int getBillete(String nom) {
  String pregSQL = "SELECT NUMERO FROM BILLETES"+
   " WHERE ESTADO='LIBRE'";
  try{
    ResultSet rs = sentencia.executeQuery(pregSQL);
    if (rs.next()) {
      int num = rs.getInt("NUMERO");
      int act = sentencia.executeUpdate("UPDATE BILLETES"+
              " SET ESTADO='OCUPADO', NOMBRE = '"+nom+
              "' WHERE NUMERO="+num+" AND ESTADO='LIBRE'");
      conexion.commit();
      if (act>0) return num; // Núm. billete asignado
      return -2;
    } // Otro ya ha OCUPADO ese billete
    else return -1; } // No había ninguno libre
  catch (SQLException e)
    {System.out.println("Error: "+e.toString());}
  return -2; // Que prueben otra vez a llamar
```

# Procedimientos almacenados

Se pueden crear desde Java

```
private static Connection c;
    Statement s = c.createStatement();

    String sql = "create procedure MOSTRAR_USUARIO() " +
          "select * from prueba order by usuario desc;";


    s.executeUpdate(sql);


    s.close();
```

# Procedimientos almacenados

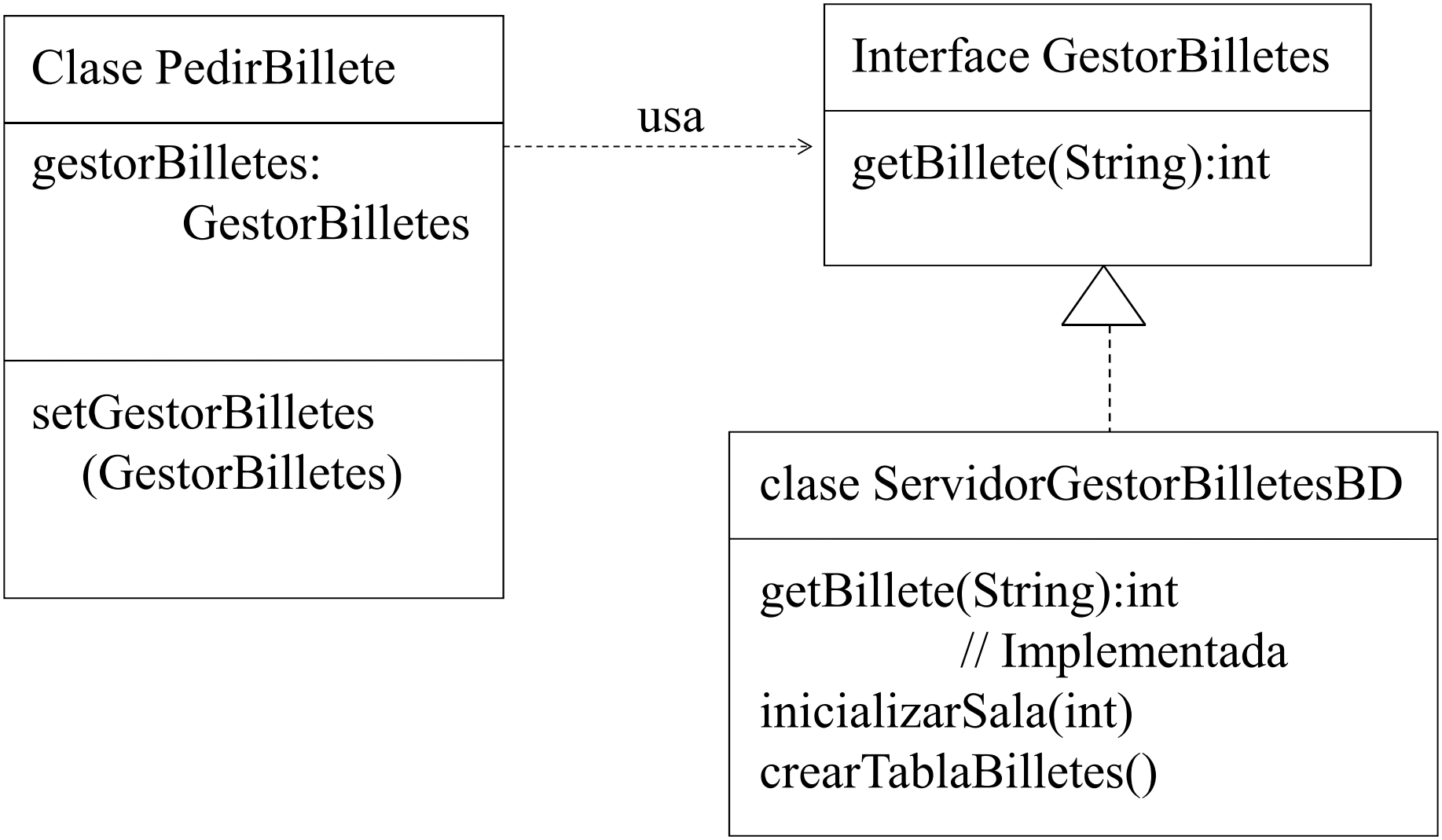
Se pueden usar desde Java

```
private static Connection c;
String sql= "{CALL MOSTRAR_USUARIOS()}";
CallableStatement s = (CallableStatement) c.prepareCall(sql);
ResultSet r = (ResultSet) s.executeQuery();
while (r.next()) {
   System.out.print(     r.getString("usuario"
                  + " "
                  + r.getString("contraseña"));
   System.out.println("");
}
s.close();
```

# Conexión con el nivel de presentación

- La clase ServidorGestorBilletes es una clase que contiene la LÓGICA DEL NEGOCIO
  - inicializarSala y getBillete
- Y que realiza llamadas al NIVEL DE DATOS
  - Son todas las sentencias JDBC
- Para realizar la conexión con el NIVEL DE PRESENTACIÓN, bastará con implementar la interfaz utilizada por el mismo y asignar un objeto de la lógica del negocio al objeto de presentación

# Conexión con el nivel de presentación



# Conexión con el nivel de presentación

```
public class ServidorGestorBilletesBD implements GestorBilletes
{
        ...
         public int getBillete(String nom) {
            ... }
}
```

SE AÑADE

```
ServidorGestorBilletesBD s = new ServidorGestorBilletesBD();
s.crearTablaBilletes();
s.inicializarSala(3);
PedirBillete p = new PedirBillete();
p.setGestorBilletes(s);
p.setVisible(true);
```

Y ASIGNAMOS ESTA
LÓGICA DEL NEGOCIO
AL OBJETO PRESENTACIÓN