

# **TEMAS SELECTOS DE BASES DE DATOS**

# **TEMAS SELECTOS DE ING. DE SOFTWARE**

## **FACULTAD DE INGENIERÍA**



- Parte 1:
  - Antecedentes: Java
- Parte 2:
  - Acceso a base de datos con JDBC
- Parte 3:
  - Hibernate
- Parte 4:
  - Aplicaciones Web



PARTE 2

# **7. CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC**



# Modelando casos de estudio ..

- Antes de iniciar con JDBC, se realizará el modelo de datos de un caso de estudio asociado a la administración de los datos de una empresa teatral. (ver práctica de modelado).
- Los ejemplos en esta sección se realizan empleando Oracle, sin embargo, pueden ser adaptados para cualquier manejador.
- Los siguientes puntos muestran los elementos básicos para realizar el modelado de datos:
- Elementos de un modelo de datos
  - **Entidades:** Sustantivos que representan a un objeto de la vida real que cuenta con al menos un atributo.
    - *Atributos: característica o propiedad de una entidad*
  - **Reglas de negocio:** Enunciados cortos, precisos y sin ambigüedades que definen la forma en que estas entidades interactúan y se relacionan con otras.
  - **Relaciones entre entidades.**
    - 1:1
    - 1:M
    - M:N



# Modelado de casos de estudio (cont.)

- Notaciones mas comunes:
  - Crow's foot
  - IDEF1X
- Conceptos importantes para modelar:
  - Nivel de dependencia:
    - *Relaciones identificativas.*
    - *Relaciones no identificativas*
  - Dependencia de existencia.
    - *Opcional*
    - *Obligatoria.*
  - Participación de la existencia de una entidad en una dependencia:
    - *Opcional*
    - *Obligatoria.*
  - Cardinalidad
  - Subtipos (Generalización y especialización)
  - Modelado de entidades cambiantes con el tiempo (históricos).

# Notación crow's foot..

- Relaciones no identificativas, dependencia de existencia obligatoria:



- Relaciones no identificativas, dependencia de existencia opcional



- Relaciones identificativas, dependencia de existencia obligatoria.



- Relaciones no identificativas, dependencia de existencia obligatoria:
- Relaciones no identificativas, dependencia de existencia opcional:



- Relaciones identificativas, dependencia de existencia obligatoria.



CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC

# PRÁCTICA 6. DESARROLLAR DIAGRAMA ER



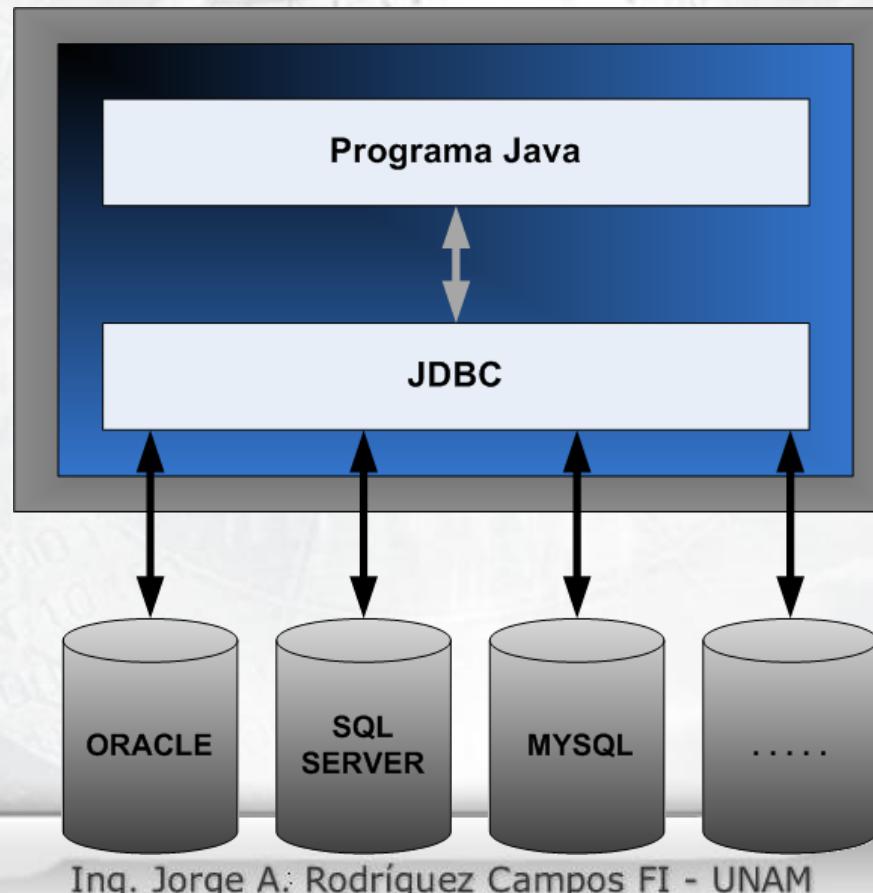
- Acceso a datos (data access) es el proceso de obtener información que es tomada de una fuente de información (datasource).
- La fuente de información no necesariamente tiene que ser relacional.
- Ejemplos de fuentes de datos:
  - Un servidor de base de datos como Oracle.
  - Un archivo de texto en la PC local.
  - Una hoja de calculo.



- Java Database Connectivity JDBC es el API estándar de Java para empleada para interactuar con una bases de datos.
- Provee un medio independiente de conexión entre Java y bases de datos ya sea basadas en SQL o algún otro medio tabular de información (datos relacionales).
- Proporciona una interfaz estándar tratando de ocultar las particularidades de cada manejador de bases de datos.



- El API esta integrada por un conjunto de interfaces y clases escritas en Java independientes del manejador.
- La gran mayoría de manejadores de bases de datos proporcionan una implementación de este API..



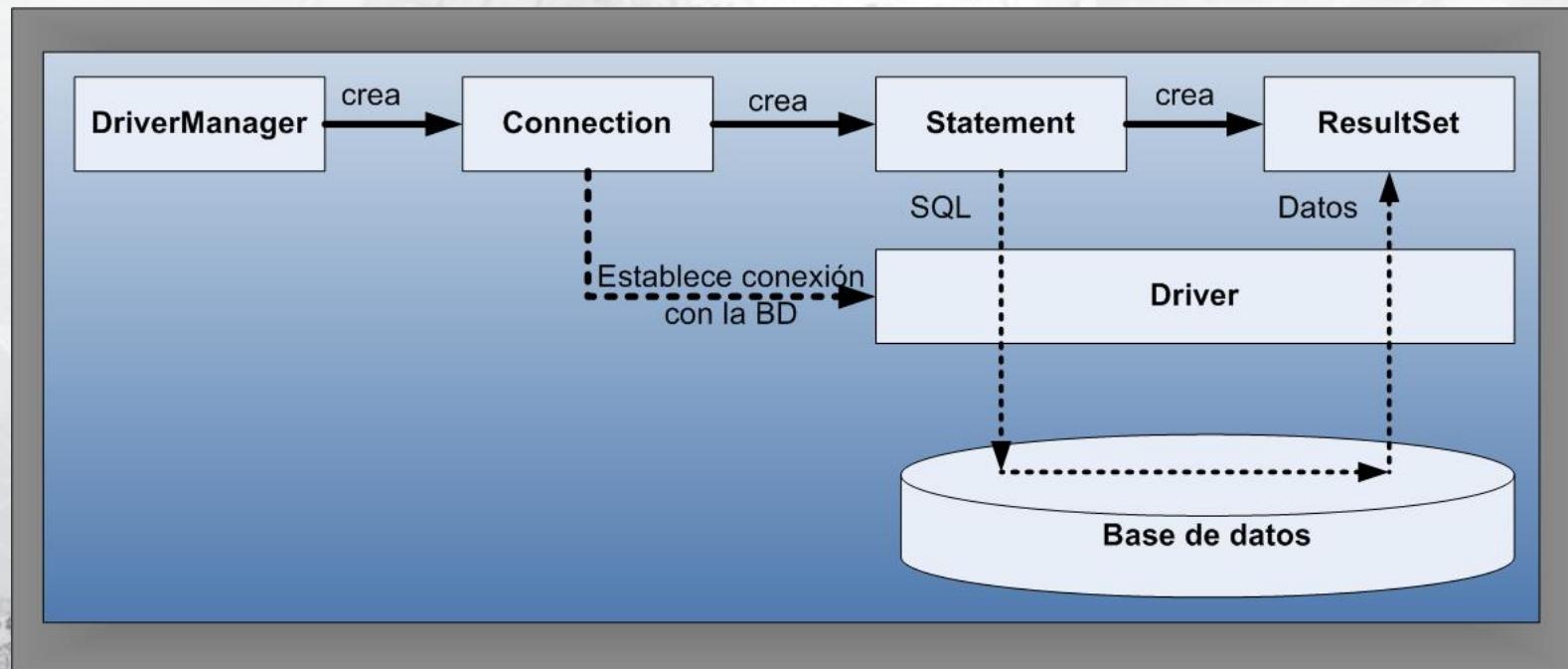
- JDBC 1.0 (JDK 1.1)
  - Primera versión de la especificación
  - Incluida en el JDK 1.1
  - Pensada principalmente desde el punto de vista del cliente.
- JDBC 2.0 (JDK 1.2)
  - Se agrega la interfaz DataSource como otro medio para realizar conexiones.
  - Pool de conexiones.
  - Transacciones distribuidas. Permite que una transacción abarque más de una BD.
  - Se agrega la interfaz RowSet.



- JDBC 3.0 (J2SDK 1.4)
  - Pool de statements para reutilizar sentencias.
  - Savepoints. Para hacer un rollback hasta un punto determinado.
  - Metadatos para un PreparedStatement
- JDBC 4.0 (Java 6)
  - Auto discovery de drivers
  - Se agrega soporte para el tipo de dato XML del estándar SQL:2003.
  - Soporte para encadenamiento de excepciones con SQLException



- La arquitectura JDBC está basada en una serie de clases e interfaces que en conjunto permiten a una aplicación conectarse a una base de datos, crear y ejecutar sentencias SQL , obtener y modificar información.



- De acuerdo a la implementación del driver se pueden clasificar en cuatro tipos:
  - JDBC-ODBC Bridge driver
  - Native API/partly Java
  - Pure Java Driver for Database Middleware
  - Direct-to-Database Pure Java Driver

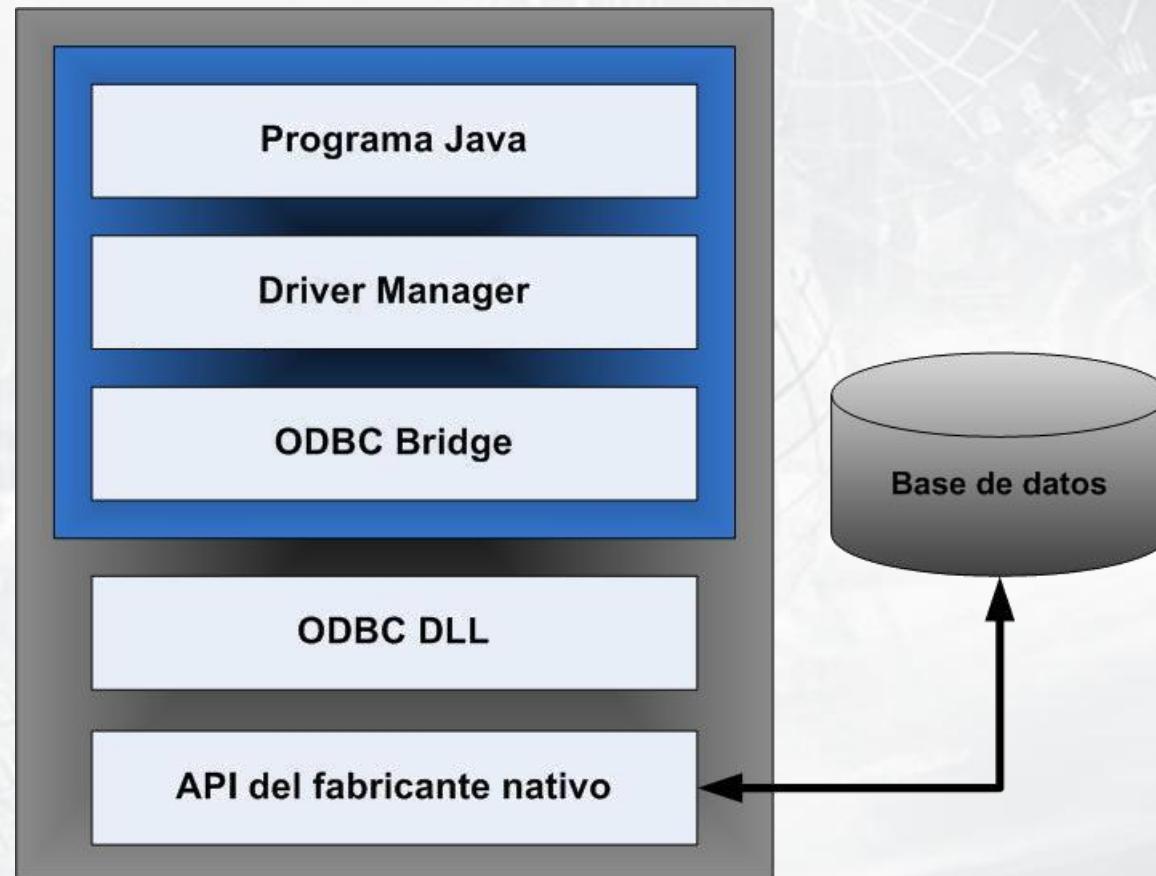


# Driver tipo 1: JDBC-ODBC Bridge driver

- Traduce operaciones JDBC en operaciones ODBC en lenguaje C.
- Las llamadas se pasan al driver ODBC apropiado.
- El driver implementa el API de JDBC para cualquier driver ODBC.
- Es usado en aquellos casos que no existe un driver para Java.
- Incluido en JDK.
- Trabaja solo en las plataformas donde se soporta ODBC.



- Arquitectura driver tipo 1



# Driver tipo 1, ventajas y desventajas

- Ventajas
  - Se incluye con el JDK en el paquete **sun.jdbc.odbc.JdbcOdbcDriver**
  - Trabaja con cualquier driver ODBC
  - Útil para clientes que ya cuentan con un ODBC instalado.
- Desventajas
  - Rendimiento
  - No recomendable para ambientes de producción.
  - Funcionalidad limitada
  - No adecuado para sistemas multi-hilos
  - Solo funciona para plataformas windows (donde existen los ODBC).
  - Puede corromper la JVM

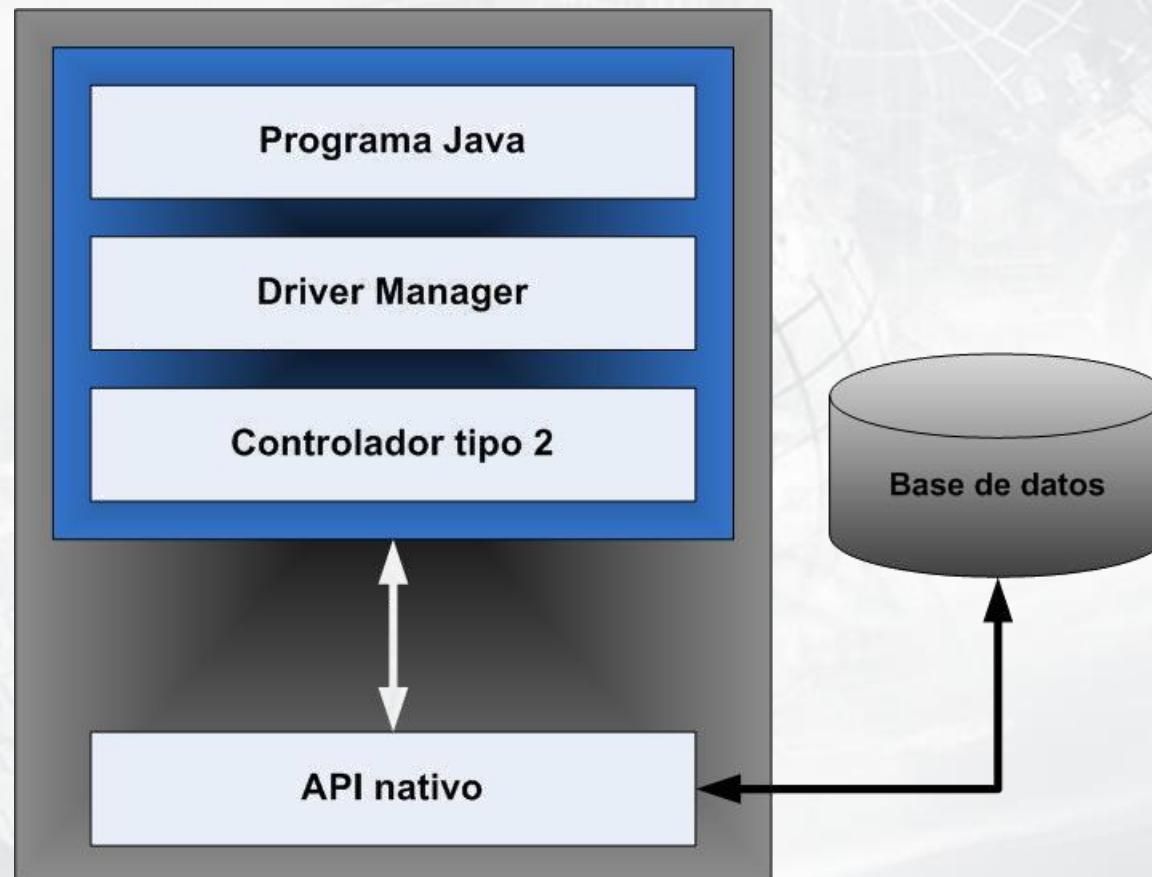


# Driver tipo 2: Native API/Partly Java Driver

- El driver reside en la capa del cliente junto con la aplicación
- Los archivos binarios específicos del manejador de base de datos deben estar instalados en el cliente.
- Se comunica directamente con la base de datos.
- Los drivers están escritos en una combinación Java, C, C++.



- Arquitectura driver tipo 2



# Driver tipo 2, ventajas y desventajas

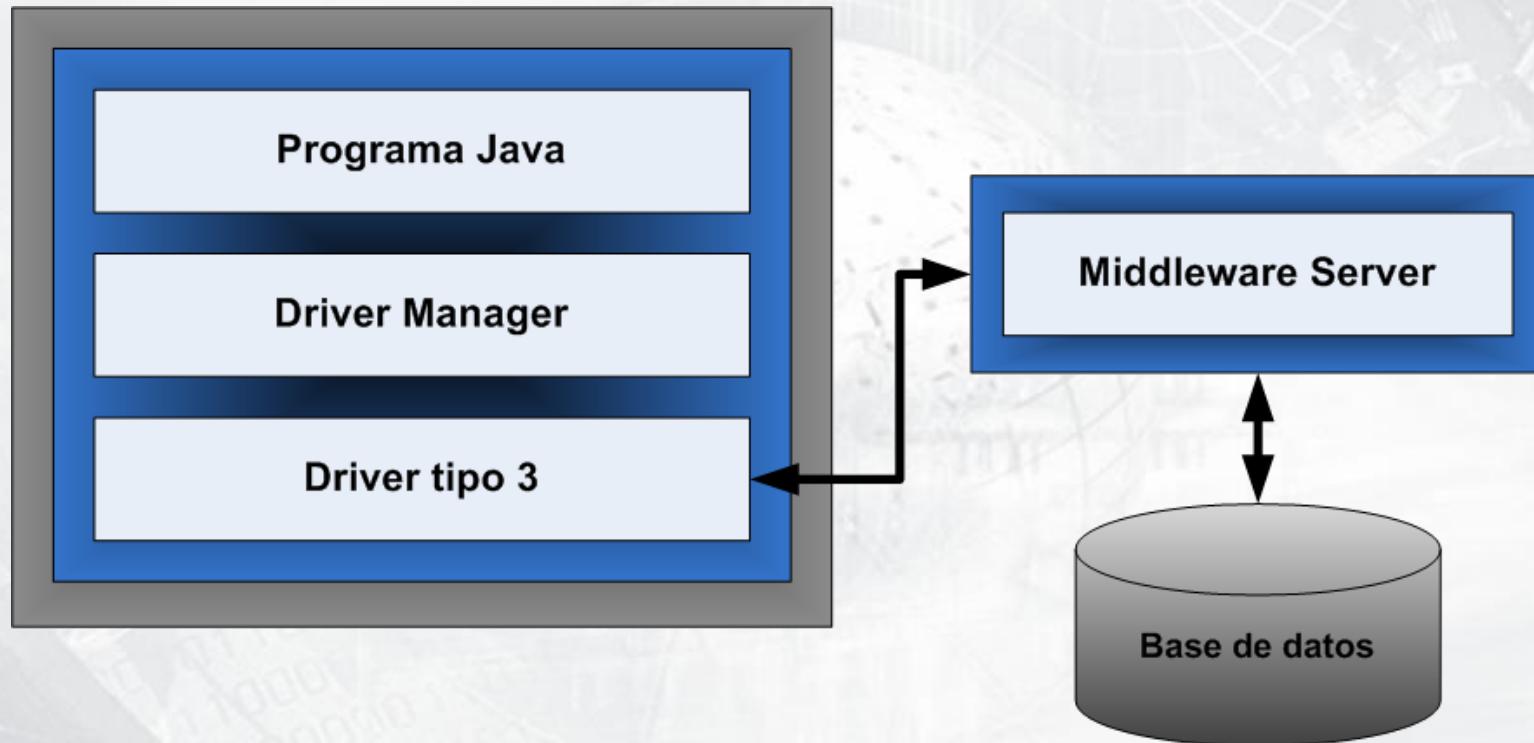
- Ventajas
  - Ofrece mejor rendimiento que el tipo 1 (puente JDBC-ODBC)
- Desventajas
  - Las librerías del fabricante deben ser instaladas en el cliente, son dependientes de la plataforma
  - Tienen menor rendimiento que los drivers 3 y 4.
  - Puede corromper la JVM



- Las instrucciones de JDBC son traducidas a un protocolo de red independiente del manejador de BD.
- Estas instrucciones se envían al servidor middleware (traductor de instrucciones).
- El middleware traduce las instrucciones para diversos manejadores de BD y le pasa la instrucción al manejador.
- El servidor middleware administra las conexiones, cache, balanceo de carga.etc.
- El servidor puede estar escrito en Java.



- Arquitectura driver tipo 3



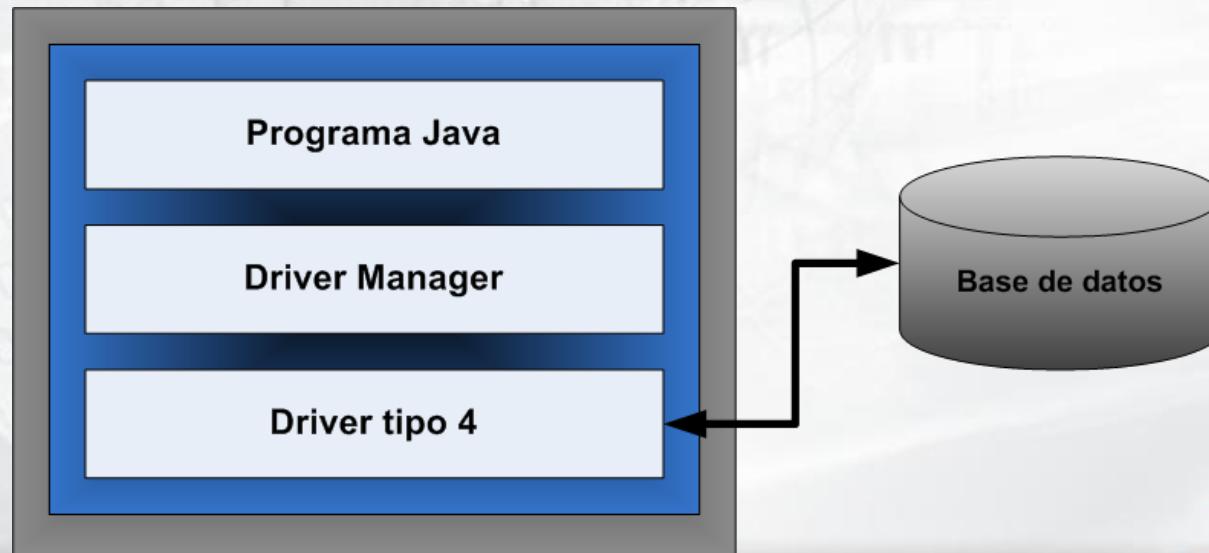
# Driver tipo 3, ventajas y desventajas

- Ventajas
  - No requiere instalar librerías específicas del manejador de base de datos en el cliente.
  - Ideal para redes basadas en intranet o internet.
  - El driver tipo 3 es muy ligero y rápido de cargar.
  - El driver ofrece servicios adicionales como cache, balanceo de cargas, administración avanzada, seguridad.
- Desventajas
  - Los drivers tipo 3 requieren código específico de la base de datos
  - El transporte de datos a través del server intermedio puede incrementar considerablemente los tiempos de respuesta.



# Driver tipo 4: Direct-to-Database Pure Java

- Comunica directamente con el manejador de la BD utilizando el protocolo nativo del manejador.
- Pueden estar escritos totalmente en Java
- Se requiere un driver diferente para cada manejador.



# Driver tipo 4 , ventajas y desventajas

- Ventajas
  - No se requiere traducir las instrucciones a ODBC ni emplear un servidor intermedio
  - Ofrece mayor rendimiento que los drivers 1 y 2
  - No requiere de librerías nativas en el cliente.
- Desventajas
  - Se requiere un driver diferente para cada manejador de bases de datos.



- Tanto JDBC como ODBC están basados en un estándar común: SQL X/Open CLI
- Este estándar establece el acceso a datos a través de la definición de las siguientes conceptos (interfaces):
  - Driver Manager
  - Driver
  - Connection
  - Statement
  - Metadata
  - Resultset



# Principales pasos a seguir para acceder a una BD

- Importar las clases necesarias
- Cargar el driver JDBC empleando la instrucción **Class.forName**
- Identificar la fuente de datos empleando URLs
- Obtener un objeto **Connection**
- Obtener un objeto **Statement**
- Ejecutar una sentencia SQL empleando **Statement**
- Obtener los datos del objeto **ResultSet**
- Cerrar el objeto **ResultSet**
- Cerrar el objeto **Statement**
- Cerrar el objeto **Connection**.



- Carga el controlador de base de datos. Maneja las conexiones entre la aplicación y el controlador.
- Cuando se carga el driver a través de **Class.forName**, este se registra así mismo empleando el método **DriverManager.registerDriver(Driver)**
- Dentro de **DriverManager** puede existir una lista de Drivers registrados.



- Ejemplo:

```
// Carga el driver JDBC
Class.forName("com.mysql.jdbc.Driver");
```

- En la instrucción anterior, al invocar al método **forName**, se realiza una búsqueda de la clase **com.mysql.jdbc.Driver** en el classpath y se genera una instancia (el jar del driver debe estar en el classpath).
  - Si no se encuentra dicha clase se lanza **ClassNotFoundException**.
- Al generarse dicha instancia, se invoca el método **registerDriver** en la clase **DriverManager** para registrar este driver dentro de la lista.

- Principales métodos.
  - Obteniendo una conexión con DriverManager:

```
String urlConexion = "jdbc:mysql://
127.0.0.1/assembly?user=root&password=root";
```

```
// Obtiene un objeto Connection
Connection conn =
  DriverManager.getConnection(urlConexion);
```

- DriverManager regresa la conexión con el primer Driver de su lista que corresponda con el valor de la variable **urlConexion**.
- ¿qué significa la variable **urlConexion**?

C DriverManager (from java.sql)	
	Operation
S	<a href="#">getLogWriter()</a>
S	<a href="#">setLogWriter(PrintWriter)</a>
S	<a href="#">getConnection(String, Properties)</a>
S	<a href="#">getConnection(String, String, String)</a>
S	<a href="#">getConnection(String)</a>
S	<a href="#">getDriver(String)</a>
S	<a href="#">registerDriver(Driver)</a>
S	<a href="#">deregisterDriver(Driver)</a>
S	<a href="#">getDrivers()</a>
S	<a href="#">setLoginTimeout(int)</a>
S	<a href="#">getLoginTimeout()</a>
S	<a href="#">setLogStream(PrintStream)</a>
S	<a href="#">getLogStream()</a>
S	<a href="#">println(String)</a>
S	<a href="#">getCallerClass(ClassLoader, String)</a>
S	<a href="#">loadInitialDrivers()</a>
S	<a href="#">getConnection(String, Properties, ClassLoader)</a>
A	<a href="#">initialize()</a>
C	<a href="#">DriverManager()</a>
S	<a href="#">getCallerClassLoader()</a>
A	<a href="#">&lt;clinit&gt;()</a>

- Un URL representa un recurso en algún servidor.
- En el caso de JDBC el url denota el driver y la localización de una fuente de datos.
- El formato que usa es:
  - **jdbc:<subprotocol>:<dataSourceIdentifier>**
- subprotocol
  - Normalmente es una cadena que identifica el nombre del manejador de la BD, mysql, oracle, etc, y que representa el subprotocolo de comunicación entre la BD y el driver.
- DataSource Identifier:
  - Es una cadena específica del driver que identifica el acceso a la fuente de datos. Normalmente en esta cadena se especifica el usuario, el password y el nombre de la BD. Para mysql por ejemplo, podría ser:
    - *localhost/curso*



- Oracle:
  - `jdbc:oracle:thin:[user/password]@[host][:port]:SID`
    - *SID: System Identifier (nombre de la instancia)*
  - Driver: `oracle.jdbc.driver.OracleDriver`
- PostgreSQL:
  - `jdbc:postgresql://host:port/database?user=userName&password=pass`
  - Driver: `org.postgresql.Driver`
- MySQL:
  - `jdbc:mysql://<servidor>[:<puerto>]/<instancia>?user=<user>&password=<password>`
  - `com.mysql.jdbc.Driver`



Connection  
(from java.sql)

## Attribute

- `SF TRANSACTION_NONE : int = 0;`
- `SF TRANSACTION_READ_UNCOMMITTED : int = 1;`
- `SF TRANSACTION_READ_COMMITTED : int = 2;`
- `SF TRANSACTION_REPEATABLE_READ : int = 4;`
- `SF TRANSACTION_SERIALIZABLE : int = 8;`

## Operation

- `createStatement()`
- `prepareStatement(String)`
- `prepareCall(String)`
- `nativeSQL(String)`
- `setAutoCommit(boolean)`
- `getAutoCommit()`
- `commit()`
- `rollback()`
- `close()`
- `isClosed()`
- `getMetaData()`
- `setReadOnly(boolean)`
- `isReadOnly()`
- `setCatalog(String)`
- `getCatalog()`
- `setTransactionIsolation(int)`
- `getTransactionIsolation()`

- `getWarnings()`
- `clearWarnings()`
- `createStatement(int, int)`
- `prepareStatement(String, int, int)`
- `prepareCall(String, int, int)`
- `getTypeMap()`
- `setTypeMap(Map<String, Class<?>>)`
- `setHoldability(int)`
- `getHoldability()`
- `setSavepoint()`
- `setSavepoint(String)`
- `rollback(Savepoint)`
- `releaseSavepoint(Savepoint)`
- `createStatement(int, int, int)`
- `prepareStatement(String, int, int, int)`
- `prepareCall(String, int, int, int)`
- `prepareStatement(String, int)`
- `prepareStatement(String, int[])`
- `prepareStatement(String, String[])`
- `createClob()`
- `createBlob()`
- `createNClob()`
- `createSQLXML()`
- `isValid(int)`
- `setClientInfo(String, String)`
- `setClientInfo(Properties)`
- `getClientInfo(String)`



# Preparando el ambiente de desarrollo..

- Pasos previos antes de iniciar.
  - Instalar manejador de base de datos, de preferencia Oracle.
  - Crear una instancia.
  - Crear un esquema y/o usuario llamado «cenat».
  - Obtener el driver jdbc para el manejador seleccionado. Por ejemplo, para Oracle, este se puede obtener de:  
<http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html>
  - Para la versión 11g de Oracle, se obtendrá el archivo **ojdbc6.jar**
  - Para MySQL: <http://dev.mysql.com/downloads/connector/j/>, el archivo se llama **mysql-connector-java-5.1.15.tar.gz** o **mysql-connector-java-5.1.15.zip**
  - Para PostgreSQL: <http://jdbc.postgresql.org/download.html> seleccionar el archivo **postgresql-8.4-702.jdbc3.jar** o el equivalente dependiendo la versión del manejador.

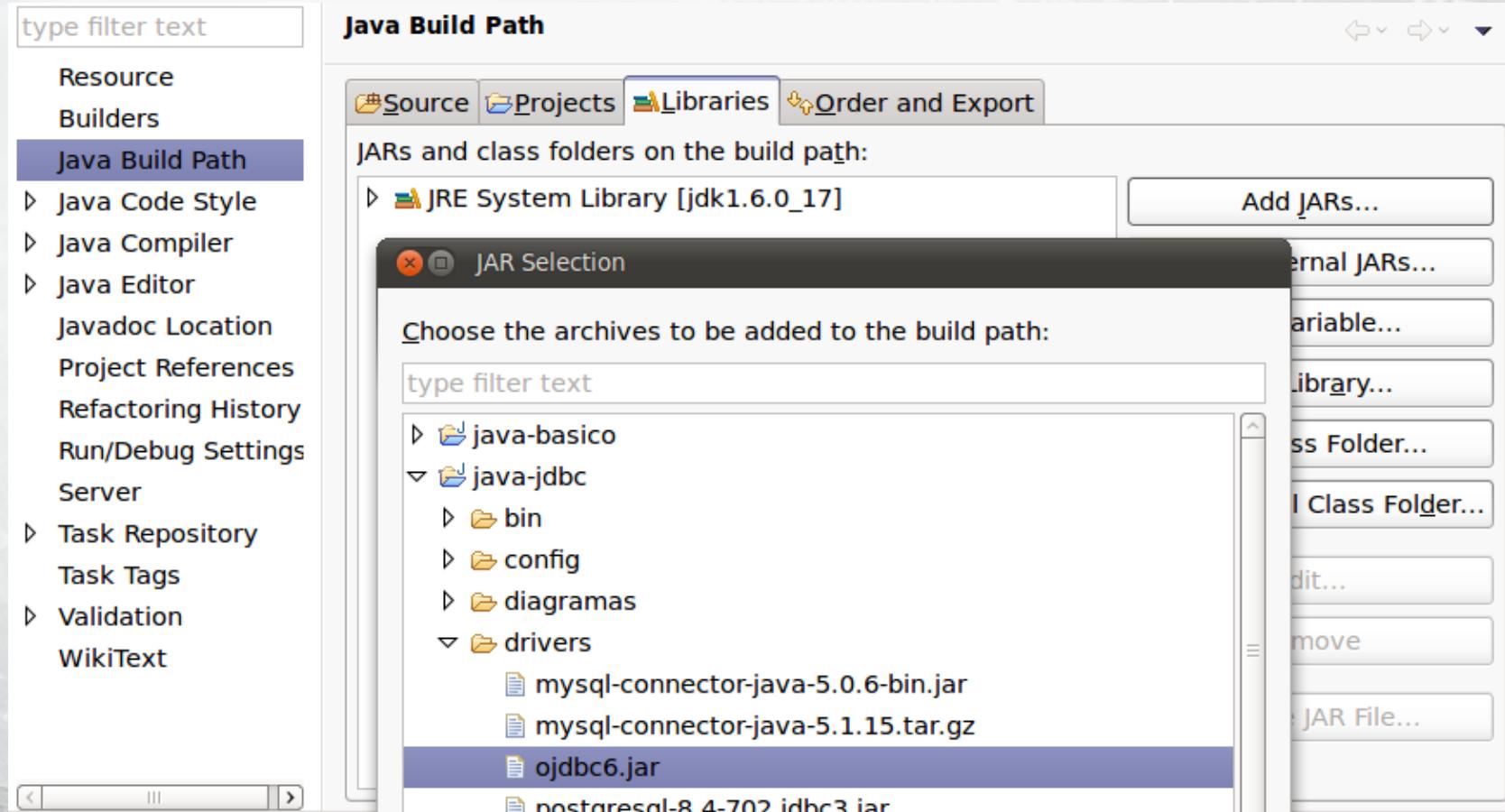


# Preparando el ambiente de desarrollo (cont.)

- Crear un proyecto en la IDE o reutilizar uno existente.
- Crear un directorio **drivers** dentro del proyecto.
- Copiar el driver al directorio drivers.
- Configurar el classpath de la IDE para incluir el driver en el classpath.
- En eclipse, seleccionar el proyecto, clic derecho-> properties. Del lado izquierdo seleccionar java build path, en la pestaña libraries agregar el driver haciendo clic en add Jars...
- Crear un programa en Java encargado de conectarse a la base de datos.



## – Configurando el driver..



# Programa básico en JDBC

```
public class ConexionJdbc {  
  
    public static void main(String[] args) {  
        Connection conn = null;  
        DatabaseMetaData metadata;  
  
        try {  
            // Carga el driver JDBC buscando la clase OracleDriver en el classpath  
            Class.forName("oracle.jdbc.OracleDriver");  
            // Identifica el data source  
            String urlConexion = "jdbc:oracle:thin:cenat/cenat@localhost:1521:ORA11DEV";  
            // Obtiene un objeto Connection  
            conn = DriverManager.getConnection(urlConexion);  
            System.out.println("conexion establecida con exito! " + conn);  
  
            System.out.println("Imprimiendo datos de la conexion");  
            metadata = conn.getMetaData();  
  
            System.out.println("url: " + metadata.getURL());  
            System.out.println("Manejador: " + metadata.getDatabaseProductName());  
            System.out.println("Version: " + metadata.getDatabaseProductVersion());  
            System.out.println("Driver: " + metadata.getDriverName());  
            System.out.println("Version: " + metadata.getDriverVersion());  
        }  
    }  
}
```

# Programa básico en JDBC (cont.)

```
catch (SQLException e) {
    System.err.println("Ocurrió un error, más detalle en la excepción ");
    e.printStackTrace();
    System.err.println("codigo: " + e.getSQLState());
} catch (ClassNotFoundException e) {
    System.err.println("No se encontró el driver JDBC");
    e.printStackTrace();
} finally {
    try {
        // Cierra el objeto ResultSet
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException e) {
        System.err.println("Error al liberar recurso");
    }
}
}
```

Importante!, nunca olvidar cerrar la conexión.

En JDBC 4.0, ya no es necesario la siguiente línea:

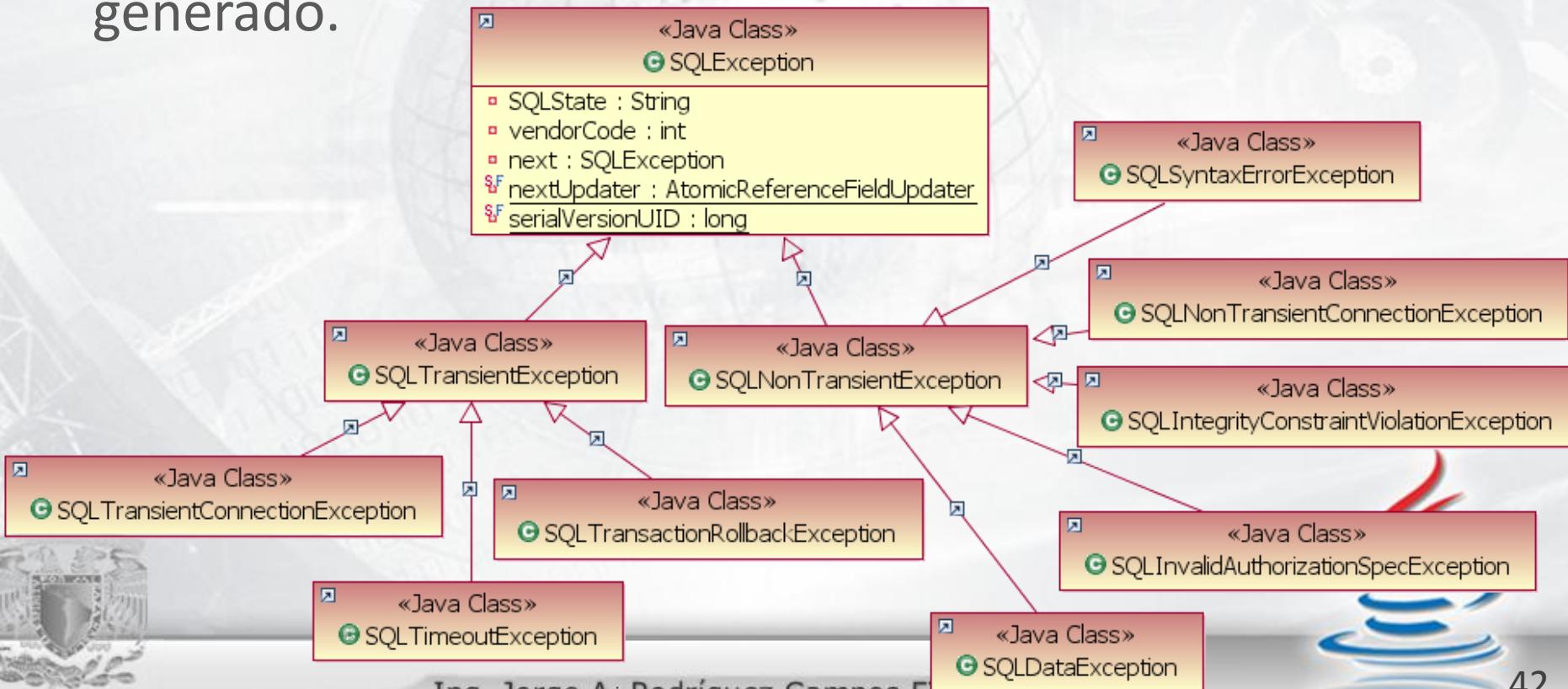
`Class.forName("oracle.jdbc.OracleDriver");`

Para determinar la clase, se verifica el archivo `java.sql.Driver` ubicado en el directorio `META-INF/services` dentro del archivo jar del driver.

- Comentar la línea, comentar el catch de `ClassNotFoundException`, y verificar que todo funcione correctamente.

# Manejo de errores en JDBC 4

- La clase **SQLException** es la clase base empleada para reportar errores al interactuar con la base de datos.
- A partir de JDBC 4, se define una nueva Jerarquía de excepciones que permite identificar el tipo de error generado.



- Clases que heredan de SQLTransientException
  - Se refieren a errores que pueden corregirse intentando nuevamente, o intentando tiempo después
  - Ejemplo: Error al intentar conectarse a la base de datos por algún problema de conectividad.
- Clases que heredan de SQLNonTransientException
  - Se refieren a errores que no se corrigen a pesar de que se realicen varios intentos.
  - Ejemplo: Error de sintaxis.



- Cadena que contiene un estado de error definido por el estándar X/Open SQL que se obtiene al invocar al método **getSQLState**
- La cadena esta formada por 5 caracteres, clase y subclase.
- Ejemplos:
  - 01 Éxito pero con advertencias:
    - 01002 *Error de desconexión*
  - 08 Excepción en la conexión
    - 08001 *El servidor rechazo la conexión*
    - 08S01 *Error de comunicaciones*



# Códigos de error particulares del RDBMS

- Es un valor entero particular al manejador de base de datos. Se obtiene haciendo una llamada al método **getErrorCode**.
- Su significado es completamente dependiente del manejador.
- Encadenamiento de excepciones:
  - Ocurre cuando hay mas de un objeto de excepción asociado con el error.
  - Se emplea una lista ligada para enlazarlas.
  - Para recorrer la lista se emplea el método **getNextException**



# Encadenamiento de excepciones, ejemplo:

```
catch (SQLException e) {  
    SQLException nextException;  
    System.err.println(  
        "Ocurrió un error, más detalle en la excepción ");  
    System.err.println("codigo de error: " + e.getSQLState());  
    System.err.println("Codigo del manejador: " + e.getErrorCode());  
    System.err.println("Mensaje: " + e.getMessage());  
    System.err.println("Encadenamiento de excepciones");  
    while ((nextException = e.getNextException()) != null) {  
        System.err.println("Excepcion encadenada: ");  
        System.err.println("Mensaje: " + nextException.getMessage());  
        System.err.println(  
            "codigo de error: " + nextException.getSQLState());  
        System.err.println("Codigo del manejador: "  
            + nextException.getLocalizedMessage());  
    }  
}
```



# Creando un ConnectionFactory

- El ejemplo visto anteriormente que obtiene la conexión a una base de datos, funciona correctamente, sin embargo tiene algunas desventajas:
  - Cada vez que se requiera elaborar un programa Java que realice una conexión, habrá que repetir el código para conectarse, generando código duplicado de forma innecesaria.
  - Si se cambia de manejador, habrá que recompilar la clase, ya que se especifica de forma explícita el url (String) que indica los datos del manejador a conectarse.
  - Hasta antes de JDBC 4, se tiene que indicar la clase que representa al Driver del RDMBS.
- Solución: Crear un ConnectionFactory reutilizable para cualquier manejador.



# Creando un ConnectionFactory, ejemplo:

- La siguiente clase permite obtener objetos tipo Connection con solo invocar la siguiente instrucción:
  - **Connection con = ConnectionFactory.getConnection();**
- El manejo de excepciones la realiza la propia clase.
- Para cerrar los recursos y la conexión, simplemente se invoca la siguiente instrucción:
  - **ConnectionFactory.close(con);**
- Los datos específicos del RDBMS (usuario, password, ip, etc.) se configuran de forma externa en un archivo de configuración, llamado archivo de propiedades (properties file).

# Código de la clase ConnectionFactory

```
public class ConnectionFactory {  
  
    // crea un objeto Properties que representa al archivo de configuraciones  
    private static final Properties properties = new Properties();  
  
    // Crea una instancia de esta clase a través del constructor privado (una sola  
    // instancia)  
    private static final ConnectionFactory cf = new ConnectionFactory();  
  
    //constructor privado  
    private ConnectionFactory() {  
  
        String driverClass;  
        System.out.println("Inicializando los datos de la conexión");  
        try {  
            //carga el contenido del archivo config/conexion.properties  
            properties.load(new FileReader(new File("config/conexion.properties")));  
            driverClass = properties.getProperty("connection.database.driver.class");  
            System.out.println("Cargando el driver.." + driverClass);  
            Class.forName(driverClass);  
        } catch (FileNotFoundException e) {  
            // transforma a RuntimeException ya que son errores irrecuperables.  
            throw new RuntimeException("Archivo de configuracion no encontrado", e);  
        } catch (IOException e) {  
            throw new RuntimeException("Error al leer el archivo de configuracion", e);  
        } catch (ClassNotFoundException e) {  
            throw new RuntimeException("Driver no encontrado, revise el classpath", e);  
        }  
    }  
}
```

Observar la ruta relativa.

Nombre de una propiedad, debe existir en el archivo de configuración.

```
/** Obtiene una conexion de la base de datos */
public static Connection getConnection() {
    String url = properties.getProperty("connection.database.url");
    System.out.println("Obteniendo conexion ");
    try {
        return DriverManager.getConnection(url);
    } catch (SQLException e) {
        throw new RuntimeException("Error al obtener la conexion de la base de datos", e);
    }
}

/** Cierra una conexion */
public static void close(Connection connection) {

    System.out.println("Cerrando conexion");
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (SQLException e) {
        System.out.println("Error al cerrar la conexion");
        e.printStackTrace();
    }
}
```



```
/* Cierra un Statement */
public static void close(Statement statement) {
    System.out.println("Cerrando statement");
    try {
        if (statement != null) {
            statement.close();
        }
    } catch (SQLException e) {
        System.out.println("Error al cerrar el statement");
        e.printStackTrace();
    }
}

/* Cierra un ResultSet */
public static void close(ResultSet rs) {

    System.out.println("Cerrando ResultSet");
    try {
        if (rs != null) {
            rs.close();
        }
    } catch (SQLException e) {
        System.out.println("Error al cerrar la conexion");
        e.printStackTrace();
    }
}
```



- En Java existe un estándar para crear archivos de configuraciones (archivos de propiedades).
- Cada línea del archivo representa una propiedad formada por una llave y un valor.
  - **nombre.llave=valor.propiedad**
- Por convención las palabras compuestas se separan por puntos.
- El símbolo # se emplea para comentar una línea.
- Por convención los archivos tienen extensión .properties
- Los archivos se leen proporcionando su ubicación absoluta o relativa.
  - Recordando: Una ruta relativa, es relativa al directorio donde se invoca a la JVM. En este caso, a la carpeta del proyecto.



# Archivo de propiedades para ConnectionFactory

```
#Nombre completo del driver para conexion a la bd  
connection.database.driver.class=oracle.jdbc.OracleDriver  
  
#URL de conexion a la base de datos  
connection.database.url=jdbc:oracle:thin:cenat/cenat@localhost:1521:ORA11DEV
```

- La cadena del lado izquierdo del operador **=**, representa la llave o nombre de la propiedad.
- La cadena del lado derecho del operador **=**, representa el valor de la propiedad.
- Para obtener el valor de una propiedad empleando una instancia de la clase **Properties**, se emplea el método **getProperty(key)** tomando como argumento en nombre de la propiedad.



- Se emplea para enviar sentencias SQL a la base de datos.
- El driver de cada manejador implementa a la interface de forma particular.
- **Statement** puede ser empleada para ejecutar sentencias DDL como sentencias DML.
- Los principales métodos empleados para enviar sentencias a la base de datos son:
  - **executeQuery** se emplea para ejecutar sentencias que regresan datos (sentencias select).
  - **executeUpdate** se emplea para ejecutar sentencias que no regresan datos (sentencias diferente a select).
  - **execute** se emplea para sentencias DDL.

# La interface Statement (cont.)

- Existen métodos para tener mayor control sobre las sentencias ejecutadas como:
  - Limitar el número máximo de registros regresados.
  - Limitar el tiempo máximo para esperar un resultado.
  - Cancelar una sentencia.
  - Métodos para realizar operaciones en batch
  - Obtención de las llaves primarias generadas por el manejador para columnas identity o auto increment (MySQL, etc.)



# Interface Statement, principales métodos.



Statement  
(from java.sql)

Attribute  
Operation

- `executeQuery(String)`
- `executeUpdate(String)`
- `close()`
- `getMaxFieldSize()`
- `setMaxFieldSize(int)`
- `getMaxRows()`
- `setMaxRows(int)`
- `setEscapeProcessing(boolean)`
- `getQueryTimeout()`
- `setQueryTimeout(int)`
- `cancel()`
- `getWarnings()`
- `clearWarnings()`
- `setCursorName(String)`
- `execute(String)`
- `getResultSet()`
- `getUpdateCount()`
- `getMoreResults()`
- `setFetchDirection(int)`

- `getFetchDirection()`
- `setFetchSize(int)`
- `getFetchSize()`
- `getResultSetConcurrency()`
- `getResultSetType()`
- `addBatch(String)`
- `clearBatch()`
- `executeBatch()`
- `getConnection()`
- `getMoreResults(int)`
- `getGeneratedKeys()`
- `executeUpdate(String, int)`
- `executeUpdate(String, int[])`
- `executeUpdate(String, String[])`
- `execute(String, int)`
- `execute(String, int[])`
- `execute(String, String[])`
- `getResultSetHoldability()`
- `isClosed()`
- `setPoolable(boolean)`
- `isPoolable()`

- Para el caso de estudio, generar un programa que realice las siguientes operaciones:
  - Crear la tabla STATUS\_OBRA
  - Poblar la tabla con los estados de cada obra teatral descritos en el caso de estudio.

STATUS_OBRA		
◆	STATUS_OBRA_ID (PK) NUMBER(2,0)	NOT NULL
◆	CLAVE	VARCHAR2(50) NOT NULL
◆	DESCRIPCION	VARCHAR2(255) NOT NULL



# Interface Statement, ejemplo (cont.).

```
public class EjemploStatementCreaStatus {  
  
    public static void main(String[] args) {  
        creaTablaStatus();  
        llenaTablaStatus();  
    }  
  
    private static void creaTablaStatus() {  
        Connection connection;  
        Statement statement = null;  
        StringBuilder sql;  
  
        sql = new StringBuilder("CREATE TABLE STATUS_OBRA( ");  
        sql.append("STATUS_OBRA_ID      NUMBER(2, 0)      NOT NULL," );  
        sql.append("CLAVE              VARCHAR2(50)      NOT NULL," );  
        sql.append("DESCRIPCION        VARCHAR2(255)     NOT NULL," );  
        sql.append("CONSTRAINT status_obra_pk PRIMARY KEY (STATUS_OBRA_ID))");  
  
        connection = ConnectionFactory.getConnection();  
        try {  
            statement = connection.createStatement();  
            System.out.println("Creando la tabla status_obra ");  
            statement.execute(sql.toString());  
        } catch (SQLException e) {  
            System.out.println("Error al crear la tabla: ");  
            e.printStackTrace();  
        } finally {  
            ConnectionFactory.close(connection);  
            ConnectionFactory.close(statement);  
        }  
    }  
}
```



# Interface Statement, ejemplo (cont.).

```
private static void llenaTablaStatus() {  
    Connection connection;  
    Statement statement = null;  
    String[] data;  
    int numAffectedRows;  
  
    data = new String[] {  
        "insert into status_obra (status_obra_id,clave,descripcion) values "  
        + "(1,'REGISTRADA','Status asignado a una obra nueva que se registra en la BD')",  
        "insert into status_obra (status_obra_id,clave,descripcion) values "  
        + "(2,'EN PREPARACION','Status asignado a una obra que se encuentra en ensayos')",  
        "insert into status_obra (status_obra_id,clave,descripcion) values "  
        + "(3,'LISTA PARA PUBLICAR','Status asignado a una obra que ha terminado ensayos')",  
        "insert into status_obra (status_obra_id,clave,descripcion) values "  
        + "(4,'EN CARTELERA','Status asignado a una obra que esta en exhibicion al publico')",  
        "insert into status_obra (status_obra_id,clave,descripcion) values "  
        + "(5,'CONCLUIDA','Status asignado a una obra al terminar su puesta en escena')", };  
  
    connection = ConnectionFactory.getConnection();  
  
    System.out.println("Poblando la tabla estatus.");
```

¿Qué desventajas tendrá este código SQL ?



# Interface Statement, ejemplo (cont.).

```
for (String sql : data) {  
    try {  
        statement = connection.createStatement();  
        numAffectedRows = statement.executeUpdate(sql);  
        if (numAffectedRows != 1) {  
            throw new SQLException("Error al ejecutar la sentencia " + sql  
                + ", numero de registros afectados incorrecto: " + numAffectedRows);  
        }  
        ConnectionFactory.close(statement);  
  
    } catch (SQLException e) {  
        System.out.println("Error al insertar el nuevo estatus: "+sql);  
        e.printStackTrace();  
    }  
    finally{  
        ConnectionFactory.close(connection);  
    }  
}  
}
```



## 7. CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC

### EJERCICIO 1.



- Creando el catálogo de géneros teatrales.
  1. Del caso de estudio CENAT, se cuenta con un catálogo de géneros teatrales representado por la siguiente tabla:

GENERO_TEATRAL		
GENERICO_ID (PK)	NUMBER(2,0)	NOT NULL
CLAVE	VARCHAR2(50)	NOT NULL
DESCRIPCION	VARCHAR2(500)	NOT NULL

2. El contenido del catálogo es:
  - a. *Tragedia: (1,TRAGEDIA,Genero teatral tragedia)*
  - b. *Drama(2,DRAMA,Genero teatral drama)*
  - c. *Comedia(3,COMEDIA,Genero teatral comedia)*
  - d. *Clásica (4,CLASICA,Genero teatral clasico)*
  - e. *Monologo(5,MONOLOGO, Obra representada por una sola persona)*



– Generar un programa que realice las siguientes acciones:

- *Empleando ConnectionFactory el programa deberá crear la tabla anterior, e insertar los datos asociados con los generos teatrales empleando Statement (similar al ejemplo visto anteriormente).*
- *Incluir en el ejercicio, el código fuente y la salida del programa.*



- ¿Qué características tienen en común las siguientes sentencias SQL ?
  - `insert into grupo(idgrupo,idsemestre,clave) values(1,1,'CT-01');`
  - `insert into grupo(idgrupo,idsemestre,clave) values(2,1,'CT-02');`
  - `insert into grupo(idgrupo,idsemestre,clave) values(3,1,'MB-01');`
- **PreparedStatement** extiende de **Statement**, permite realizar una optimización al realizar una sola vez el procesamiento de una sentencia SQL que será ejecutada n veces con datos diferentes: Pre-compilación.
- Procesamiento de una sentencia SQL
  - Análisis léxico
  - Análisis sintáctico
  - Análisis semántico.



- La Parametrización de una sentencia SQL se realiza a través de un placeholder representado por el signo de interrogación “?”
- Ejemplo:
  - **`insert into grupo(idgrupo,idsemestre,clave) values(?, ?, ?);`**
- Cada placeholder representa el dato que será enviado a la base de datos.
- Cada placeholder es indexado iniciando en 1,2,3.. n.
- Cuidado, los parámetros solo se pueden emplear para sustituir datos, no para sustituir elementos de la estructura o sintaxis de la sentencia (nombres de tablas, nombres de campos, etc.).

# Asignando valores a los parámetros

- Antes de ejecutar la sentencia SQL se deben proporcionar todos los valores de los parámetros o placeholders indicados por «?».
- Los placeholders solo pueden utilizarse para representar los datos a enviar.
- Para asociar los valores a cada placeholder se emplean los métodos:
  - **setXxx(parameterIndex, valor)**.
- Xxx Corresponde al tipo de dato a enviar a BD.
  - Ejemplos: **setInt**, **setString**, **setBoolean**.
- parameterIndex inicia en 1,2,3... , indica en número de placeholder a sustituir.
- Para determinar el tipo de dato correcto se emplea la siguiente tabla de equivalencias entre tipos de dato Java y tipos de dato SQL.
- Para obtener un PreparedStatement se emplea:
  - **PreparedStatement = con.prepareStatement(query)**



# Mapeo de tipos de datos SQL a Java

Tipo SQL	Tipo Java
CHAR	<b>String</b>
VARCHAR	<b>String</b>
LONGVARCHAR	<b>String</b>
NUMERIC	<b>java.math.BigDecimal</b>
DECIMAL	<b>java.math.BigDecimal</b>
BIT	<b>boolean</b>
TINYINT	<b>byte</b>
SMALLINT	<b>short</b>
INTEGER	<b>int</b>
BIGINT	<b>long</b>



# Mapeo de tipos de datos SQL a Java (cont.)

Tipo SQL	Tipo Java
REAL	<b>float</b>
FLOAT	<b>double</b>
DOUBLE	<b>double</b>
BINARY	<b>byte[]</b>
VARBINARY	<b>byte[]</b>
LONGVARBINARY	<b>byte[]</b>
DATE	<b>java.sql.Date</b>
TIME	<b>java.sql.Time</b>
TIMESTAMP	<b>java.sql.Timestamp</b>

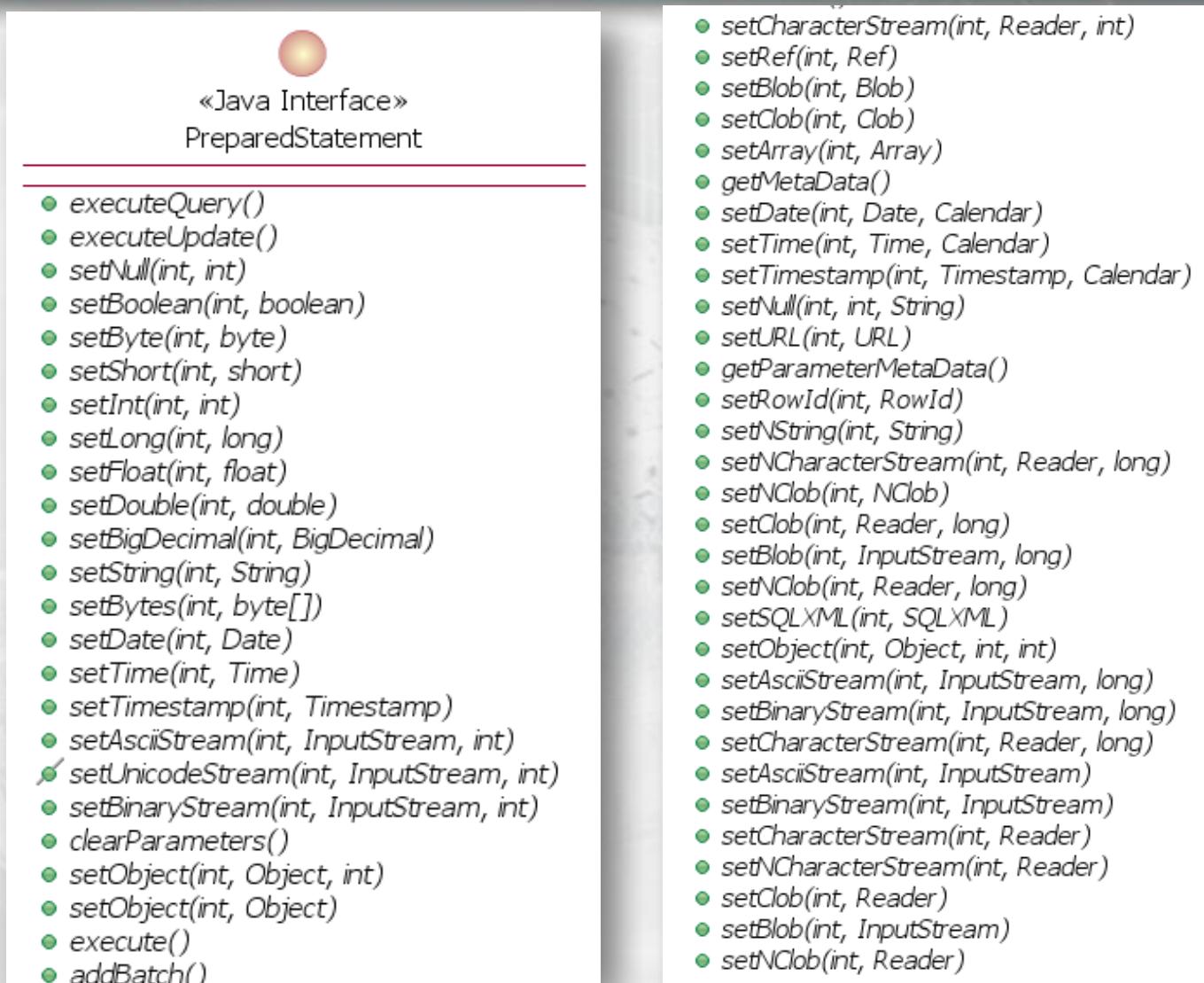


# Mapeo de tipos de datos SQL a Java (cont.)

Tipo SQL	Tipo Java
REAL	<code>float</code>
DOUBLE	<code>double</code>
VARBINARY, LONGVARBINARY	<code>byte[]</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>

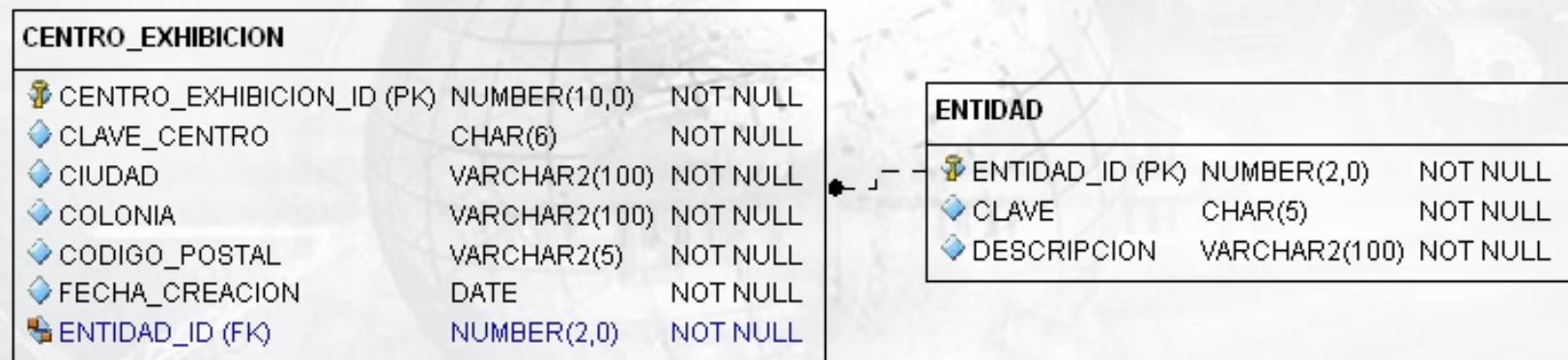


# Interface PreparedStatement, diagrama:



# PreparedStatement, ejemplo

- Crear un programa que inicialice el catálogo de entidades empleando un PreparedStatement.



# PreparedStatement, creando estados.

```
public class EjemploPreparedStatementCreaCatEntidad {  
  
    private static final String queryCreaEntidad = "insert into entidad "  
        + "(entidad_id,clave,descripcion) values (?,?,?,?)";  
  
    private static final String[] entidades = new String[] { "AGUASCALIENTES",  
        "BAJA CALIFORNIA", "BAJA CALIFORNIA SUR", "CAMPECHE", "CHIHUAHUA", "COAHUILA",  
        "COLIMA", "CHIAPAS", "CHIHUAHUA", "DURANGO", "ESTADO DE MÉXICO", "GUANAJUATO",  
        "GUERRERO", "HIDALGO", "JALISCO", "MICHOACÁN", "MORELOS", "NAYARIT", "NUEVO LEÓN",  
        "OAXACA", "PUEBLA", "QUERÉTARO", "QUINTANA ROO", "SAN LUIS POTOSÍ", "SINALOA",  
        "SONORA", "TABASCO", "TAMAULIPAS", "TLAXCALA", "VERACRUZ", "YUCATÁN", "ZACATECAS" };  
  
    private static final String[] claves = new String[] { "AGS", "BC", "BCS", "CAMP", "CHIS",  
        "CHIH", "COAH", "COL", "DF", "DGO", "GTO", "GRO", "HGO", "JAL", "MEX", "MICH", "MOR",  
        "NAY", "NL", "OAX", "PUE", "QRO", "QR", "SLP", "SIN", "SON", "TAB", "TAMPS", "TLX",  
        "VER", "YUC", "ZAC" };  
  
    public static void inicializaCatalogoEntidades() {  
  
        Connection con;  
        PreparedStatement ps = null;  
        int rows = 0;
```



```
con = ConnectionFactory.getConnection();
try {
    ps = con.prepareStatement(queryCreaEntidad);
    System.out.println("Insertando estados de la Rep. Mex.");
    for (int i = 0; i < entidades.length; i++) {
        // sustituye parametros
        ps.setInt(1, i + 1);
        ps.setString(2, claves[i]);
        ps.setString(3, entidades[i]);
        // ejecuta ps
        rows += ps.executeUpdate();
        ps.clearParameters();
    }
    if (rows != entidades.length) {
        throw new SQLException(
            "No se insertaron todos los registros esperados, solo: " + rows);
    }
} catch (SQLException e) {
    System.out.println("Error al insertar los estados de la Rep. Mex:");
    e.printStackTrace();
} finally {
    ConnectionFactory.close(ps);
    ConnectionFactory.close(con);
}

public static void main(String[] args) {
    inicializaCatalogoEntidades();
}
```

Observar que el objeto **ps** se reutiliza

En lugar de indicar el número de índice, también se puede indicar el nombre del campo:

```
ps.setInt("entidad_id", i+1);
ps.setString("clave", claves[i]);
```



# Manejo de valores nulos

- Para enviar valores nulos a la base de datos, se emplea el método **setNull** de **PreparedStatement**
- Aunque el dato a enviar sea nulo, se debe especificar el tipo de dato, empleando la clase **java.sql.Types**.
- Ejemplo: Suponer que el campo apellido\_materno puede ser nulo:

```
if(actor.getApellido == null){  
    ps.setNull("apellido_materno", Types.VARCHAR);  
}else{  
    ps.setString("apellido_materno",  
        actor.getApellidoMaterno());  
}
```

Tipo de dato  
especificado en la BD.

## «Java Class»

### Types

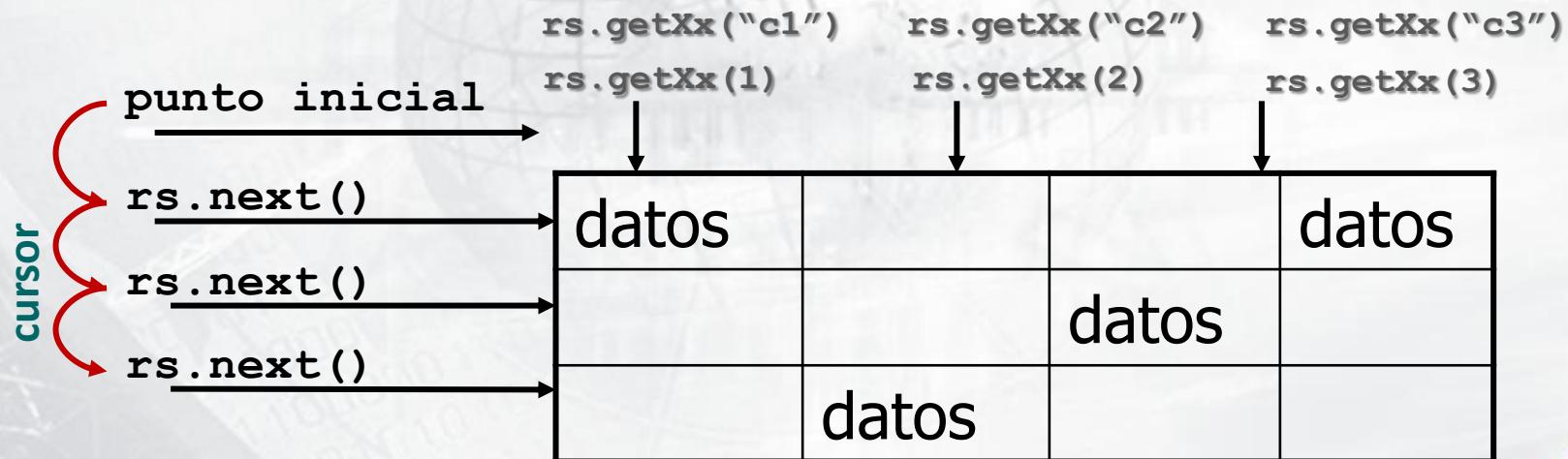
BIT : int
TINYINT : int
SMALLINT : int
INTEGER : int
BIGINT : int
FLOAT : int
REAL : int
DOUBLE : int
NUMERIC : int
DECIMAL : int
CHAR : int
VARCHAR : int
LONGVARCHAR : int
DATE : int
TIME : int
TIMESTAMP : int
BINARY : int
VARBINARY : int
LONGVARBINARY : int
NULL : int
OTHER : int
JAVA_OBJECT : int
DISTINCT : int
STRUCT : int
ARRAY : int
BLOB : int
CLOB : int
REF : int
DATALINK : int
BOOLEAN : int
ROWID : int
NCHAR : int
NVARCHAR : int
LONGNVARCHAR : int
NCLOB : int
SQLXML : int
Types ()

- Un objeto **ResultSet** encapsula los datos obtenidos como resultado de la ejecución de una sentencia SQL.
- Un ResultSet representa a una matriz de M renglones por N columnas que contiene los datos de una consulta.
- Cada matriz de datos contiene un puntero empleado para recorrer los renglones de la matriz.
- Antes de obtener los valores de las columnas, el puntero debe moverse al registro correspondiente.
- Para obtener el valor de una columna, se emplean los métodos **getXxx** recorriendo cada renglón o registro obtenido..
- Xxx al igual que con PreparedStatement representa el tipo de dato Java al cual se convertirá el dato recuperado de la BD.
- El método **next** mueve el cursor (puntero a un registro obtenido).



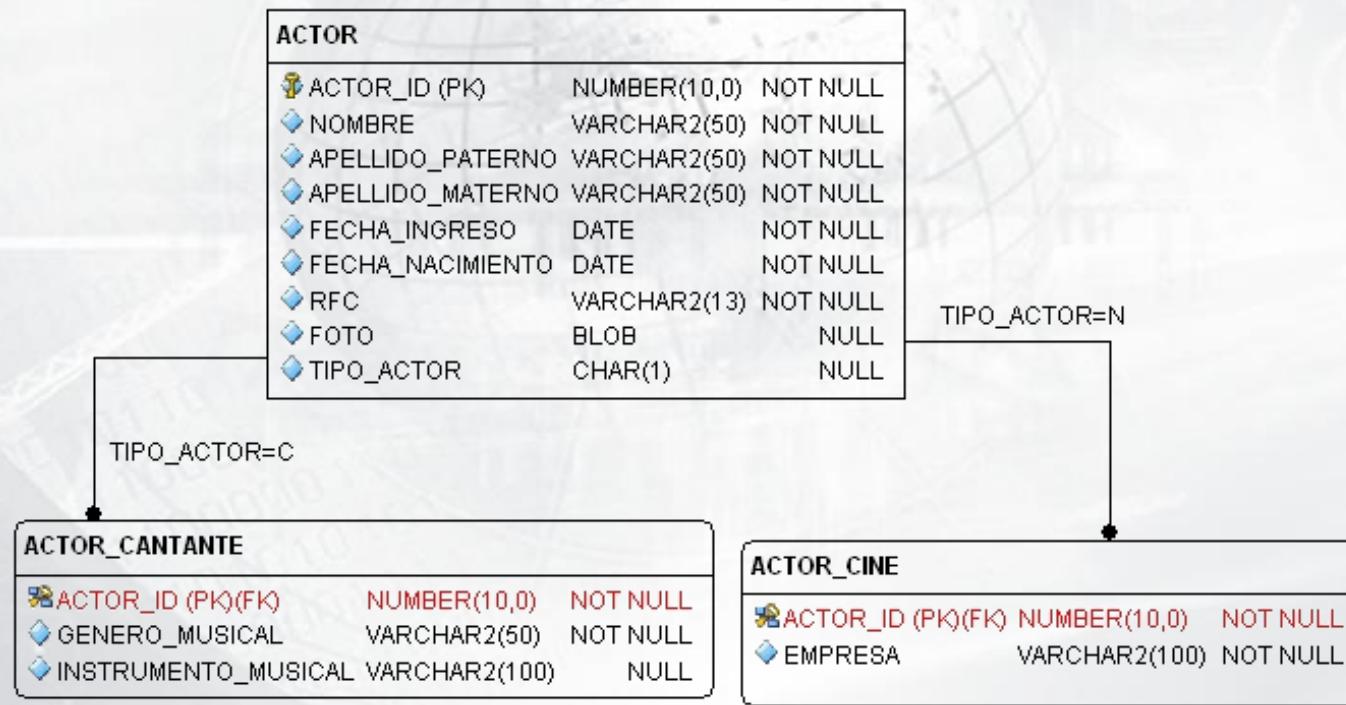
# La interface ResultSet (cont.)

- Típicamente un objeto ResultSet se recorre empleando un ciclo while.
- Dentro de cada iteración se extraen todos los datos del registro al que se está apuntando.



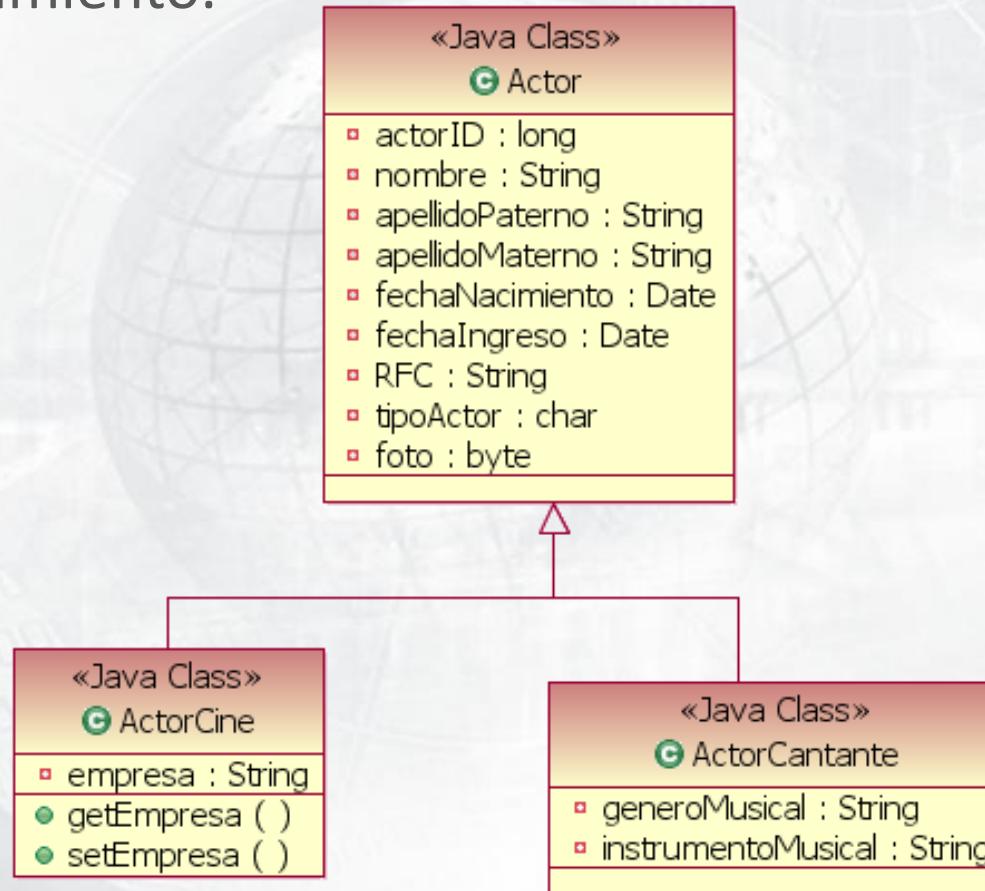
# PreparedStatement y ResultSet, ejemplo

- Crear un programa que registre un nuevo actor.
- Observe el siguiente diagrama, se emplea el concepto de subtipos para representar a los tipos de actores.



# ResultSet, ejemplo (cont.).

- Para realizar el programa, se generan las siguientes clases de negocio o entidades, que representan a un actor empleando encapsulamiento.



# ResultSet, ejemplo (cont.)

```
public class EjemploPreparedStatementCreaActor {  
  
    private static final String sqlSecuenciaActor = "select actor_seq.nextval from dual";  
  
    private static final String sqlCreaAutor = "insert into actor(actor_id,nombre,"  
        + " apellido_paterno,apellido_materno,fecha_ingreso,fecha_nacimiento,rfc,tipo_actor,"  
        + " foto) values(?,?,?,?,?,?)" ;  
  
    private static final String sqlCreaAutorCantante = "insert into actor_cantante (actor_id,"  
        + " genero_musical,instrumento_musical) values(?,?)";  
  
    private static final String sqlCreaAutorCine = "insert into actor_cine (actor_id,empresa) "  
        + " values(?,?)";  
  
    public static void creaActor(Actor actor) {  
  
        Connection conn;  
        PreparedStatement ps = null;  
        long actorID;  
        int numRows = 0;  
        int index = 1;  
  
        System.out.println("Creando a un nuevo actor");  
        conn = ConnectionFactory.getConnection();
```

Recomendación, crear todos los SQL al inicio de la clase. Observar el uso de secuencias.



# ResultSet, ejemplo (cont.)

```
try {  
    System.out.println("generando consecutivo para PK");  
    ps = conn.prepareStatement(sqlSecuenciaActor);  
    actorID = generaConsecutivo(ps);  
    actor.setActorID(actorID);  
    ps.close();  
    System.out.println("creando los datos generales del actor");  
    ps = conn.prepareStatement(sqlCreaAutor);  
    ps.setLong(index++, actor.getActorID());  
    ps.setString(index++, actor.getNombre());  
    ps.setString(index++, actor.getApellidoPaterno());  
    ps.setString(index++, actor.getApellidoMaterno());  
    ps.setTimestamp(index++, new Timestamp(actor.getFechaIngreso().getTime()));  
    ps.setTimestamp(index++, new Timestamp(actor.getFechaNacimiento().getTime()));  
    ps.setString(index++, actor.getRFC());  
    ps.setString(index++, String.valueOf(actor.getTipoActor()));  
    // suponer que la foto siempre es nula, esto despues se modificara.  
    ps.setNull(index++, Types.BLOB);  
    // se inserta en la tabla Actor.  
    numRows += ps.executeUpdate();  
}
```

Revisar el método  
**generaConsecutivo**  
en laminas posteriores.

Checar el uso de **index++**

**Timestamp** se emplea para  
enviar a la BD valores de  
fechas con resolución de hasta  
segundos o milisegundos.

# ResultSet, ejemplo (cont.)

```
// se inserta solo en uno de los subtipos.  
if (actor instanceof ActorCantante) {  
    System.out.println("Creando a un actor cantante");  
    numRows += creaAutorCantante(actor, conn);  
} else if (actor instanceof ActorCine) {  
    System.out.println("Creando a un autor de cine");  
    numRows += creaAutorCine(actor, conn);  
} else {  
    throw new RuntimeException("Tipo de actor invalido");  
}  
  
if (numRows != 2) {  
    throw new SQLException(  
        "No se registraron los datos esperados. Se esperan 2 registros");  
}  
  
System.out.println("Datos insertados correctamente.");  
} catch (SQLException e) {  
    e.printStackTrace();  
} finally {  
    ConnectionFactory.close(ps);  
    ConnectionFactory.close(conn);  
}
```

Dependiendo del tipo de  
objeto se invoca a  
**creaAutorCantante**  
o **creaAutorCine**



# ResultSet, ejemplo (cont.)

```
private static int creaAutorCine(Actor actor, Connection conn) throws SQLException {  
  
    PreparedStatement ps = null;  
    int numRows;  
    ps = conn.prepareStatement(sqlCreaAutorCine);  
    ps.setLong(1, actor.getActorID());  
    ps.setString(2, ((ActorCine) actor).getEmpresa());  
    // ejecuta sentencia en BD  
    numRows = ps.executeUpdate();  
    ps.close();  
    // la conexion no se cierra, esta se cierra en el metodo principal  
    return numRows;  
}  
  
private static int creaAutorCantante(Actor actor, Connection conn) throws SQLException {  
    PreparedStatement ps = null;  
    int numRows;  
    ActorCantante cantante;  
    ps = conn.prepareStatement(sqlCreaAutorCantante);  
    ps.setLong(1, actor.getActorID());  
    cantante = (ActorCantante) actor;  
    ps.setString(2, cantante.getGeneroMusical());  
    ps.setString(3, cantante.getInstrumentoMusical());  
    // ejecuta sentencia en BD  
    numRows = ps.executeUpdate();  
    ps.close();  
    // la conexion no se cierra, esta se cierra en el metodo principal  
    return numRows;  
}
```

Estos métodos insertan en las tablas subtipos dependiendo el tipo de dato.



# ResultSet, ejemplo (cont.)

```
private static long generaConsecutivo(PreparedStatement ps) throws SQLException {  
  
    ResultSet rs;  
  
    rs = ps.executeQuery();  
    // aqui no es necesario while, ya que se espera una matriz de 1x1, solo  
    // se invoca a rs.next una vez para mover el puntero.  
    if (!rs.next()) {  
        throw new SQLException("No fue posible generar el consecutivo, RS sin valores");  
    }  
    // ya que se recorrio el puntero, se lee el dato de la primer columna.  
    return rs.getLong(1);  
}  
  
public static void main(String[] args) throws ParseException {  
    ActorCantante cantante;  
  
    System.out.println("Creando a un nuevo Actor cantante");  
    cantante = creaCantanteFicticio();  
    creaActor(cantante);  
}
```

Observar el uso de **ResultSet** para la obtención de consecutivos empleando secuencias en Oracle.

El método **main** sustituye por ejemplo, la función de una página web en la que se capturarían los datos del actor .



# ResultSet, ejemplo (cont.)

```
private static ActorCantante creaCantanteFicticio() throws ParseException {
    ActorCantante cantante;
    SimpleDateFormat sdf;

    cantante = new ActorCantante();
    cantante.setNombre("Juan Alegria");
    cantante.setApellidoPaterno("Aguirre");
    cantante.setApellidoMaterno("Gutierrez");
    cantante.setGeneroMusical("Salsa");
    cantante.setInstrumentoMusical("Teclado");
    cantante.setRFC("GUAJ790203LX3");
    cantante.setTipoActor('C');
    // creando fechas con Java
    sdf = new SimpleDateFormat("dd/MM/yyyy");
    cantante.setFechaNacimiento(sdf.parse("03/02/1979"));
    cantante.setFechaIngreso(sdf.parse("22/04/2003"));
    return cantante;
}
```

Finalmente, este método se emplea para generar un Actor con datos ficticios, se emplea **SimpleDateFormat** para crear objetos tipo **Date**



# ResultSet, consultando datos..

- Generar un programa que obtenga los datos de todos los actores se hayan registrado a partir de febrero del 2003, y que sean cantantes.

```
public class EjemploResultSetConsultaActores {  
  
    private static final String querySelectActores = "select a.actor_id,a.nombre,"  
        + "a.apellido_paterno,a.apellido_materno,a.fecha_ingreso,a.fecha_nacimiento,"  
        + "a.rfc,a.tipo_actor,c.genero_musical,c.instrumento_musical from actor a,"  
        + "actor_cantante c where a.actor_id =c.actor_id and a.fecha_ingreso >= ? and "  
        + " a.tipo_actor = ?";  
  
    public static List<ActorCantante> getActores(Date fechaIngresoMin, char tipo) {  
  
        List<ActorCantante> lista;  
        PreparedStatement ps = null;  
        ResultSet rs = null;  
        Connection con;  
        ActorCantante actor;  
        int index;  
  
        con = ConnectionFactory.getConnection();  
        lista = new ArrayList<ActorCantante>();  
    }  
}
```

Observar que las condiciones del query se expresan con placeholders sin importar su tipo de dato,y sin usar ''

```
try {  
    ps = con.prepareStatement(querySelectActores);  
    ps.setDate(1, new java.sql.Date(fechaIngresoMin.getTime()));  
    ps.setString(2, String.valueOf(tipo));  
    // ejecuta la consulta  
    rs = ps.executeQuery();  
    while (rs.next()) {  
        actor = new ActorCantante();  
        index = 1;  
        actor.setActorID(rs.getLong(index++));  
        actor.setNombre(rs.getString(index++));  
        actor.setApellidoPaterno(rs.getString(index++));  
        actor.setApellidoMaterno(rs.getString(index++));  
        actor.setFechaIngreso(rs.getDate(index++));  
        actor.setFechaNacimiento(rs.getDate(index++));  
        actor.setRFC(rs.getString(index++));  
        actor.setTipoActor(rs.getString(index++).charAt(0));  
        actor.setGeneroMusical(rs.getString(index++));  
        actor.setInstrumentoMusical(rs.getString(index++));  
        lista.add(actor);  
    }  
} catch (SQLException e) {  
    System.out.println("Error al obtener los datos del actor.");  
    e.printStackTrace();  
} finally {  
    ConnectionFactory.close(ps);  
    ConnectionFactory.close(rs);  
    ConnectionFactory.close(con);  
}  
return lista;
```

Para pasar objetos tipo **Date** a la base de datos, se debe emplear un objeto **java.sql.Date**. En este ejemplo se convierte un objeto **java.util.Date** a un **java.sql.Date**.



# ResultSet, consultando datos (cont.)

```
public static void main(String[] args) {  
    List<ActorCantante> actores;  
    Calendar calendar;  
  
    System.out.println("Obteniendo a los cantantes que ingresaron a partir de 1/02/2003");  
    // fecha actual  
    calendar = Calendar.getInstance();  
    // hace reset de los elementos de la fecha  
    calendar.clear();  
    // establece los valores para año, mes y dia  
    calendar.set(Calendar.YEAR, 2003);  
    calendar.set(Calendar.MONTH, Calendar.FEBRUARY);  
    calendar.set(Calendar.DAY_OF_MONTH, 1);  
  
    actores = getActores(calendar.getTime(), 'C');  
  
    System.out.println("Resultados obtenidos");  
    System.out.println(actores);  
}
```

En este ejemplo, la clase Calendar se emplea para formar Objetos Date que representen un instante en el tiempo.



- A partir de JDBC 2.0 el cursor puede moverse en ambas direcciones (scrollable) o moverse a un renglón en particular:

**absolute**(int row)

**afterLast**()

**beforeFirst**()

**first**()

**last**()

**previous**()

**relative**(int rows)

- next()
- close()
- wasNull()
- getString(int)
- getBoolean(int)
- getByte(int)
- getShort(int)
- getInt(int)
- getLong(int)
- getFloat(int)
- getDouble(int)
- ~~getBigDecimal(int, int)~~
- getBytes(int)
- getDate(int)
- getTime(int)
- getTimestamp(int)
- getAsciiStream(int)
- ~~getUnicodeStream(int)~~
- getBinaryStream(int)
- getString(String)
- getBoolean(String)
- getByte(String)
- getShort(String)
- getInt(String)
- getLong(String)
- getFloat(String)
- getDouble(String)
- ~~getBigDecimal(String, int)~~
- getBytes(String)
- getDate(String)
- getTime(String)
- getTimestamp(String)

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT		
getByte	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getShort	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getLong	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getFloat	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getDouble	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getBigDecimal	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getBoolean	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getString	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
getBytes													X	X	X												
getDate											X	X	X			X	X										
getTime											X	X	X			X	X										
getTimestamp											X	X	X			X	X	X									
getAsciiStream											X	X	X	X	X												
getUnicodeStream											X	X	X	X	X												
getBinaryStream												X	X	X													
getClob																		X									
getBlob																			X								
getArray																				X							
getRef																					X						
getCharacterStream													X	X	X	X	X	X									
getObject	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	

Las X en mayúscula indican la combinación mas adecuada. Por ejemplo, **getInt**, se debe emplear para un tipo de dato **INTEGER**, aunque puede usarse para todos los demás tipos de datos marcados con «x» en minúscula.



# Recuperando el valor de la PK

- Algunos manejadores como MySQL no cuentan con soporte de secuencias para generar PKs.
- En su lugar utilizan columnas de tipo:
  - AUTO\_INCREMENT
  - IDENTITY
- En ambas estrategias, al insertar un registro en la base de datos, el manejador genera en automático el siguiente valor (consecutivo) de la columna marcada como autoincrement.
- De lo anterior se desprende la siguiente cuestión:
  - Si la base de datos genera la PK, ¿de que forma el programa Java puede recuperar el valor generado?



# Recuperando el valor de la PK (cont.)

- Definición de una tabla con una columna AUTO\_INCREMENT en MySQL.



```
CREATE TABLE profesor (
profesor_id INTEGER(12) UNSIGNED NOT NULL AUTO_INCREMENT,
nombre VARCHAR(45) NOT NULL,
rfc VARCHAR(15) NULL,
fecha_nacimiento DATE NOT NULL,
PRIMARY KEY(profesor_id) ) TYPE=InnoDB;
```

- El siguiente programa muestra un ejemplo que inserta en una tabla BANCO, marcada como AUTO\_INCREMENT.



# Recuperación del valor de la PK en MySQL

```
public class EjemploStatement {  
  
    public static void main(String[] args) {  
  
        Connection connection = null;  
        Statement statement = null;  
        ResultSet rs = null;  
        String[] queries;  
  
        try {  
            connection = ConnectionFactory.getConnection();  
            statement = connection.createStatement();  
            int result;  
            long id;  
            queries = new String[] {  
                "insert into BANCO(nombre,descripcion,activo) values "  
                + "('BANCOMER','bbVa BANCOMER',true)"};  
  
            for (String query : queries) {  
  
                result = statement.executeUpdate(query, Statement.RETURN_GENERATED_KEYS);  
                if (result <= 0) {  
                    throw new SQLException("No fue posible insertar el registro, " +  
                        "0 actualizaciones");  
                }  
            }  
        }  
    }  
}
```

Observar que en el query no se incluye el campo que esta marcado como AUTO\_INCREMENT (banco\_id)

Al momento de ejecutar el query, le indicamos al objeto **Statement** o **PreparedStatement** que obtenga los valores de las PK generadas.



# Recuperación del valor de la PK en MySQL

```
// obteniendo las llaves generadas
rs = statement.getGeneratedKeys();
// solo se invoca 1 vez al metodo next() por insertar 1 solo registro
if (rs.next()) {
    id = rs.getLong(1);
    System.out.println("Se inserto exitosamente el banco con Id: " + id);
} else {
    throw new SQLException("No fue posible obtener el valor generado de la PK");
}

System.out.println("query insertado con exito");
}

} catch (SQLException e) {
    e.printStackTrace();
} finally {
    System.out.println("cerrando recursos");
    ConnectionFactory.close(connection);
    ConnectionFactory.close(statement);
    ConnectionFactory.close(rs);
}
}
```

Se emplea un **ResultSet** para recuperar el valor de las PKs generadas.



## 7. CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC

### EJERCICIO 2.



- Creando centros teatrales.
  - Empleando **PreparedStatement**, generar un programa que inserte los datos de un centro teatral.
  - Los datos de varios centros teatrales se encuentran en un archivo de texto.
    - *Cada renglón del archivo deberá representar un registro a insertar.*
    - *Generar los valores de la PK empleando una secuencia.*
    - *El formato del archivo es libre (el archivo deberá tener como mínimo 3 registros).*
    - *Los datos que se deben registrar, se describen en la siguiente tabla:*

CENTRO_EXHIBICION		
◆ CENTRO_EXHIBICION_ID (PK)	NUMBER(10,0)	NOT NULL
◆ CLAVE_CENTRO	CHAR(6)	NOT NULL
◆ CIUDAD	VARCHAR2(100)	NOT NULL
◆ COLONIA	VARCHAR2(100)	NOT NULL
◆ CODIGO_POSTAL	VARCHAR2(5)	NOT NULL
◆ FECHA_CREACION	DATE	NOT NULL
◆ ENTIDAD_ID (FK)	NUMBER(2,0)	NOT NULL



- Empleando **PreparedStatement** y **ResultSet**, generar un programa que realice la consulta de los centros teatrales.
- El programa deberá imprimir los datos del centro.
- Para consultar los centros, el programa acepta como criterios de búsqueda:
  - *El id del centro y/o*
  - *La clave del centro*
  - *La fecha de creación*
- Si no se encuentran registros, se deberá mostrar un mensaje que indique esta condición.
- Incluir en el ejercicio el código y la salida de su ejecución.



- Cómo podemos almacenar datos binarios en una BD (fotos, audio, video, objetos Java, etc.. ) ?
  - **DATALINK**. (JDBC 3.0) Hace referencia a un archivo que se encuentra fuera de la base de datos, pero es administrada por dicha base de datos. Se Mapea a un objeto **java.net.URL**
  - **BLOB** (Binary Large Object) es una secuencia de bytes que habrán de almacenarse en una base de datos, representado por la interfaz **java.sql.Blob**
  - **CLOB** (Character Large Object) es una cadena de caracteres muy grande (mayor a un varchar), representado por la interfaz **java.sql.Clob**
  - **ARRAY** Mapeado a la interfaz **java.sql.Array**
  - A partir de la versión 1.5 de Java, se agregan las clases **SerialBlob**, **SerialClob**, **SerialDataLink**, empleada para generar objetos tipo **Blob**.



# Manejando datos binarios, ejemplo:

- Generar un programa que actualice la foto del actor creado en ejemplos anteriores (Juan Alegria Aguirre Gutierrez).

Actualiza la foto.

```
public class EjemploActualizaFoto {  
  
    private static final String queryActualizaFoto = "update actor set foto = ? "  
        + "where nombre = ? and apellido_paterno = ? and apellido_materno = ?";  
  
    public static void actualizaFoto(Actor actor) {  
        Connection con;  
        PreparedStatement ps;  
        SerialBlob serialBlob;  
        int resultado;  
  
        con = ConnectionFactory.getConnection();
```



# Manejando datos binarios (cont.)

```
try {  
    ps = con.prepareStatement(queryActualizaFoto);  
    serialBlob = new SerialBlob(actor.getFoto());  
    ps.setBlob(1, serialBlob.getBinaryStream());  
    ps.setString(2, actor.getNombre());  
    ps.setString(3, actor.getApellidoPaterno());  
    ps.setString(4, actor.getApellidoMaterno());  
    resultado = ps.executeUpdate();  
    if (resultado != 1) {  
        throw new SQLException("No se actualizo la foto correctamente.");  
    }  
    System.out.println("Foto actualizada con exito.");  
} catch (SQLException e) {  
    System.out.println("Error al realizar la actualizacion en la BD:");  
    e.printStackTrace();  
} finally{  
    ConnectionFactory.close(ps);  
    ConnectionFactory.close(con);  
}  
  
public static void main(String[] args) throws IOException  
{  
    Actor actor;  
    actor = new Actor();  
    actor.setNombre("Juan Alegria");  
    actor.setApellidoPaterno("Aguirre");  
    actor.setApellidoMaterno("Gutierrez");  
    actor.setFoto(FotoAleatoria.getFotoAleatoria());  
    System.out.println("Actualizando foto del actor: " + actor);  
    actualizaFoto(actor);  
}
```

Uso de **SerialBlob**

Obtiene una foto empleando la clase **FotoAleatoria** (ver sig. lamina.)

# Obteniendo fotos aleatorias.

- Este programa muestra una estrategia para obtener un archivo binario del directorio fotos, ubicado dentro de la carpeta del proyecto.

```
public class FotoAleatoria {  
  
    private static final String DIR_FOTOS = "fotos/";  
  
    public static byte[] getFotoAleatoria() throws IOException {  
  
        byte[] foto;  
        File[] arrayFoto;  
        File fileAleatorio;  
        BufferedInputStream buffer;  
  
        arrayFoto = new File(DIR_FOTOS).listFiles();  
  
        fileAleatorio = arrayFoto[(int) (Math.random() * arrayFoto.length)];  
        buffer = new BufferedInputStream(new FileInputStream(fileAleatorio));  
        foto = new byte[(int) fileAleatorio.length()];  
        buffer.read(foto);  
        buffer.close();  
        return foto;  
    }  
}
```



# Mostrando datos binarios: foto

- Para verificar que el programa anterior actualizó la foto del actor de forma adecuada, el siguiente programa obtiene la foto de la base de datos y la muestra en una ventana sencilla generada con swing.

```
public class EjemploMuestraFoto {  
  
    private static final String querySelectFoto = "select actor_id,nombre,apellido_paterno,"  
        + "apellido_materno,foto from actor where RFC = ?";  
  
    public static Actor getFotoActor(String RFC) {  
        Actor actor;  
        PreparedStatement ps = null;  
        ResultSet rs = null;  
        Connection con;  
        SerialBlob serialBlob;  
  
        con = ConnectionFactory.getConnection();
```



# Mostrando datos binarios (cont.)

```
try {  
    System.out.println("Obteniendo los datos del actor y su foto.");  
    ps = con.prepareStatement(querySelectFoto);  
    ps.setString(1, RFC);  
    rs = ps.executeQuery();  
    // se espera un solo registro  
    if (!rs.next()) {  
        System.out.println("no se encontraron registros");  
        return null;  
    }  
    actor = new Actor();  
    actor.setActorID(rs.getLong(1));  
    actor.setNombre(rs.getString(2));  
    actor.setApellidoPaterno(rs.getString(3));  
    actor.setApellidoMaterno(rs.getString(4));  
    serialBlob = new SerialBlob(rs.getBlob(5));  
    actor.setFoto(serialBlob.getBytes(1, (int) serialBlob.length()));  
    return actor;  
} catch (SQLException e) {  
    System.out.println("Error al consultar los datos de la BD.");  
    throw new RuntimeException(e);  
} finally {  
    ConnectionFactory.close(ps);  
    ConnectionFactory.close(rs);  
    ConnectionFactory.close(con);  
}
```

Obtención del la foto con **SerialBlob**, la obtención de los bytes se realiza con el método **getBytes** indicando la posición inicial (inicia con 1) e indicando la cantidad de bytes a leer.

# Mostrando datos binarios (cont.)

```
public static void main(String[] args) {  
    JFrame frame;  
    JPanel panel;  
    Actor actor;  
    String RFC = "GUAJ790203LX3";  
  
    System.out.println("Obteniendo la foto del actor con RFC: " + RFC);  
    actor = getFotoActor(RFC);  
    System.out.println("Foto obtenida, mostrando imagen");  
  
    panel = new PanelImagen(actor.getFoto());  
    frame = new JFrame("Foto del actor: " + actor.getNombre() + " "  
        + actor.getApellidoPaterno() + " " + actor.getApellidoMaterno());  
    frame.getContentPane().add(panel);  
    frame.setSize(450, 250);  
    frame.setVisible(true);  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}  
}
```

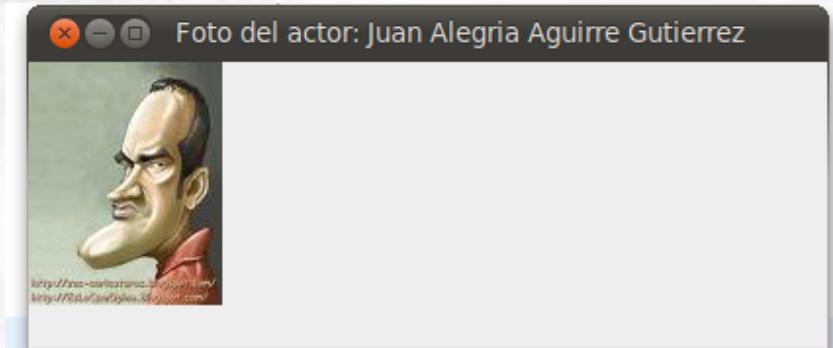
Esta clase muestra una ventana con swing para pintar la imagen recuperada de la base de datos. Se emplea la clase **PanelImagen** para pintar la imagen (ver sig. lámina).



# Mostrando datos binarios (cont.)

- Código de la clase **PanelImagen**, muestra la imagen empleando swing

```
public class PanelImagen extends JPanel {  
  
    BufferedImage imagen;  
  
    public PanelImagen(byte[] imagenBytes) {  
        try {  
            this.imagen = ImageIO.read(new ByteArrayInputStream(imagenBytes));  
        } catch (IOException ie) {  
            throw new RuntimeException(ie);  
        }  
    }  
  
    // pinta la imagen en el JPanel  
    @Override  
    public void paint(Graphics g) {  
        g.drawImage(imagen, 0, 0, null);  
    }  
}
```



## 7. CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC

### EJERCICIO 3.



1. Programar el ejemplo anterior, y verificar que todo funcione correctamente, verificar que se muestre la imagen en la ventana empleando swing.
2. Agregar una nueva funcionalidad que permita exportar la foto de un actor a un directorio llamado **export**, ubicado en el directorio raíz del proyecto Java (eclipse).
  1. Como dato de entrada, el programa deberá aceptar el identificador del actor.
  2. El programa deberá guardar el archivo jpg con la siguiente nomenclatura:
    - a. *actorID\_RFC.jpg*



## 7. CONSULTA Y MANIPULACIÓN DE DATOS CON JDBC

### EJERCICIO 4.



- Preguntas.
  1. Describa la diferencia entre un objeto **Statement** y un **PreparedStatement**
  2. ¿Cuál es la utilidad de la clase **java.sql.Types**?
  3. ¿En que casos se debe utilizar la clase **java.sql.Date**, y en que casos **java.sql.Timestamp**?
  4. Escriba un Programa que obtenga un objeto tipo Date y que represente a esta fecha: 18/03/2011
  5. Describa la forma en la que se recupera un arreglo de bytes de la base de datos empleando **SerialBlob**.