Java e Integración con Bases de Datos

JDBC (Java DataBase Connectivity)

●La API **JDBC** provee acceso a bases de datos relacionales (RDBMS), así como a fuentes de datos tabulares (planillas de cálculo). El único requerimiento es que exista el driver JDBC apropiado.

●La API **JDBC** provee una interface de programación única, que independiza a las aplicaciones del motor de base de datos usado. Incluye ***manejo de conexiones a base de datos, ejecución de sentencias SQL, store procedures, soporte de transacciones, etc.***

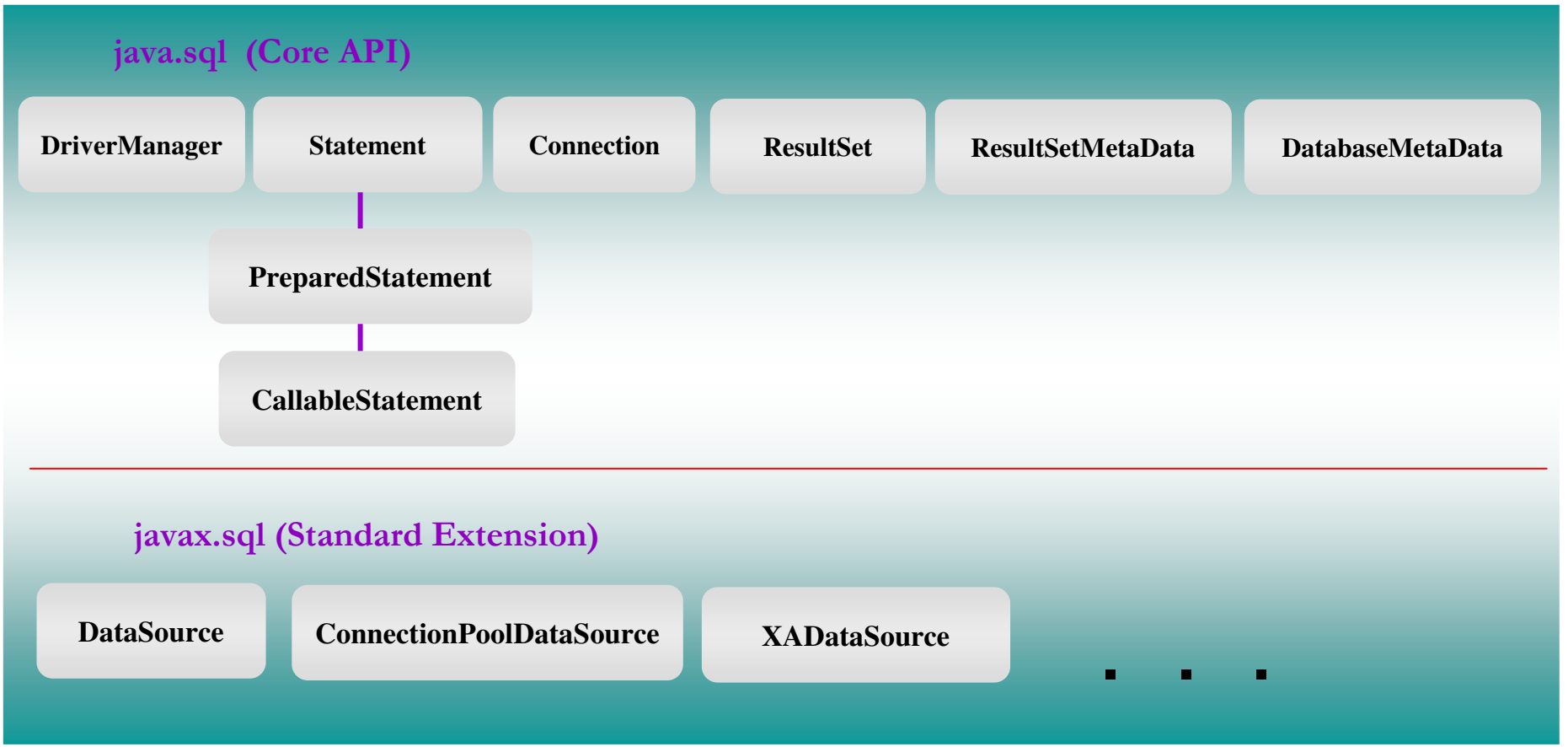
●**JDBC** define un conjunto de interfaces que el proveedor de base de datos implementa, en un código llamado ***driver***.

●El **driver JDBC** es usado por la JVM para traducir las invocaciones JDBC genéricas en invocaciones que la base de datos propietaria entiende. Los drivers son clases Java que se cargan en ejecución.

La API JDBC

Las clases e interfaces de la API **Java Database Connectivity** - **JDBC** están en los paquetes java.sql y javax.sql

En estos paquetes se encuentran definidos métodos que permiten: **conectarse a una BD**, **recuperar información acerca de la BD**, **realizar consultas SQL, ejecutar Stored Procedures y trabajar con los resultados.**



Estableciendo una Conexión
URL JDBC

Una base de datos en JDBC es identificada por una URL (Uniform Resource Locator). La sintaxis recomendada para la URL de JDBC es la siguiente:

jdbc:<subprotocolo>:<subnombre>

subprotocolo: nombra a un mecanismo particular de conectividad a una base, que puede ser soportado por uno o más drivers.

subnombre: dependen del subprotocolo, pero en general, responde a una de las siguientes sintaxis:

```
database
//host/database
//host:port/database
```

Ejemplos

"jdbc:odbc:empleadosDB" El origen de datos ODBC es **empleadosDB**, debe haberse definido en el cliente

"jdbc:mysql://localhost:3306/cursoJ2EE" Una URL para mysql con el driver **Connector/J**

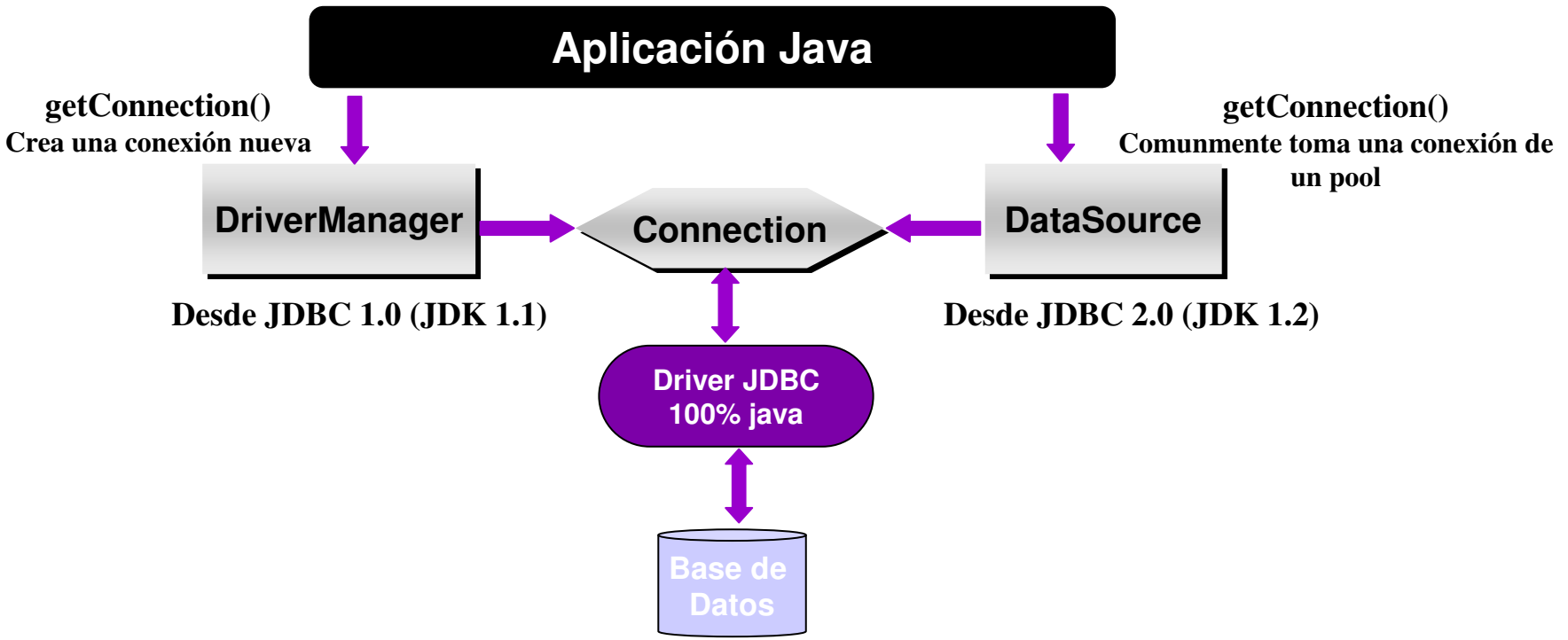
"jdbc:db2://server:50000/EMPLE" Permite conectarse a la base de IBM de nombre EMPLE

"jdbc:oracle:thin:@esales:1521:orcl" URL para un driver tipo 4 de Oracle

"jdbc:postgresql:stock" Los valores por defecto para postgreSQL son: localhost y port=5432.

Estableciendo una conexión

Objeto java.sql.Connection



La API JDBC envía comandos SQL desde la aplicación a través de un objeto ***connection***, usando el driver específico que provee acceso a la base de datos. Si la conexión no se puede establecer, se dispara una excepción SQL.

Estableciendo una Conexión
java.sql.DriverManager

DriverManager

- Es una clase que fue introducida en el original JDBC 1.0. Cuando una aplicación intenta conectarse por primera vez a una fuente de datos, especificando la URL, **DriverManager** cargará automáticamente cualquier driver JDBC, encontrado en el CLASSPATH. Cualquier driver anterior al JDBC 4.0, debe ser cargado explícitamente por la aplicación.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
miConexion = DriverManager.getConnection("jdbc:odbc:empleadosDB", uusuario, clave);

Class.forName("com.ibm.db2.jcc.DB2Driver");
miConexion = DriverManager.getConnection("jdbc:db2://server:50000/BASEEMPL");
```

- Es una clase (no una interface) que viene con la API, con lo cual, un proveedor no puede optimizarla.
- Es una clase que **permite devolver una conexión** a la base de datos.
- Dispone de 3 métodos de clase de nombre **getConnection(url,…)**, que requieren como mínimo la URL JDBC (la aplicación necesita conocer el nombre de la DB, ubicación, etc.)
- La clase **DriverManager** internamente mantiene drivers JDBC y dada una URL JDBC retorna una conexión usando el driver apropiado.

Estableciendo una Conexión
javax.sql.DataSource

La interface DataSource

- Esta interface fue introducida en **JDBC 2.0** y representa una fuente de datos particular.

- La interface **DataSource** es implementada por los proveedores de DB (o Drivers). Permite elegir las mejores técnicas para lograr un acceso óptimo a la base y definir que atributos son necesarios para crear las conexiones.

- Hay 3 tipos de implementaciones:

  (1) Implementación básica: produce un objeto Connection estándar (igual a la obtenida con DriverManger) .

  (2) Implementación de un pool de objetos Connection: produce un obJeto Connection que puede participar del *pooling* de conexiones.

  (3) Implementación para transacción distribuída: produces un objeto Connection que puede ser usado para transacciones distribuidas y casi siempre participa de un pool de conexiones.

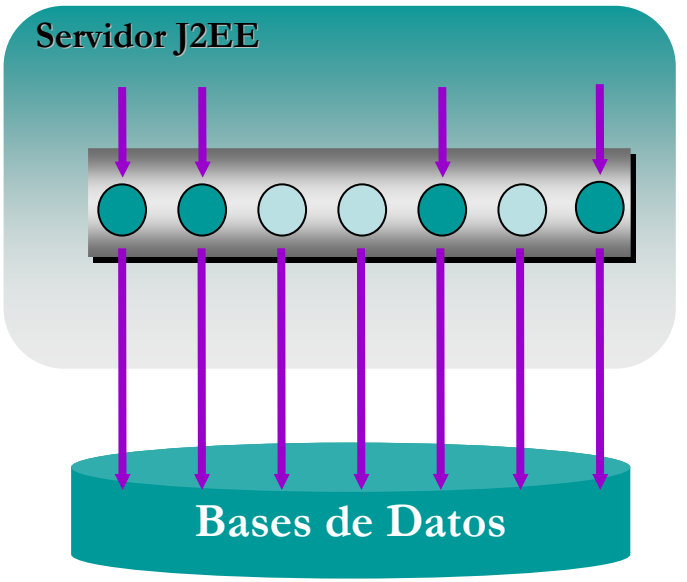
- Es **el mecanismo preferido para obtener una conexión** porque permite que los detalles acerca de los datos subyacentes, se mantengan transparentes para la aplicación. Para crear una conexión usando **DataSource no se necesita información sobre la base, el servidor, el usuario, la clave, etc**.

Estableciendo una Conexión
Pool de Conexiones

La solución más eficiente y escalable para establecer una conexión con la DB, consiste en mantener un **Pool de Conexiones**

Hay 2 alternativas:

- Se usa **DataSource** si el Servidor Web lo soporta.
- El programador podría implementar un pool de conexiones usando **DriverManager**. La funcionalidad básica del pool es obtener una conexión libre y liberarla. Se podría programar para que exista una única instancia del pool.



Las componentes J2EE, como Servlets, JavaServer Pages, y Enterprise Java Beans (EJB), a menudo requieren acceso a bases de datos relacionales y usar la API JDBC para su acceso. Sin embargo, cuando las componentes J2EE usan la JDBC API, el contenedor maneja sus transacciones y el gerenciamiento de los objetos DataSoruces, como veremos a continuación.

La API JDBC
Estableciendo una conexión con DriverManager

La clase **DriverManager** dispone de 3 métodos de clase que permiten establecer una conexión con una fuente de datos.

- **getConnection**(String **url**)
- **getConnection**(String **url**, String **usr**, String **pwd**)
- **getConnection**(String **url**, Properties **info**)

La **url** (**jdbc:<subprotocolo>:<subnombre>**), permite identificar una fuente de datos.

Solo necesario para las versiones anteriores a JDBC 4.0.

Se crea la conexión, se utiliza y se cierra

Se manejan 2 excepciones: una controla si se encuentra el **driver** y otra verifica si se estableció la conexión

```
. . .
Connection con;
 try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 con=DriverManager.getConnection("jdbc:odbc:empleadosDB");
  Statement st = con.createStatement();
       . . .
  st.close();
  con.close();
 } catch (ClassNotFoundException e) {
       System.out.println("no se encontró el driver");
 } catch (SQLException e) {
       System.out.println("no se pudo conectar a la BD");
}
```

Nombre de la clase del Driver

Nombre del Origen de Datos ODBC

URL JDBC

La API JDBC
Estableciendo una conexión con DataSource

DataSources y JNDI

El Java Naming Directory Interface (JNDI) es una API java estándar que provee acceso a un directorio. Un directorio es una ubicación centralizada que provee rápido acceso a un recurso usado por una aplicación java. JNDI permite buscar un objeto por su nombre (**`String`**).

Los objetos DataSource combinados con JNDI, son la combinación ideal para obtener una conexión. Para nuestro propósito, el uso de JNDI es muy directo:

- Se obtiene una instancia del contexto JNDI.
- Se usa ese contexto para encontrar un objeto: la base de datos.

```
InitialContext ctx = new InitialContex();
DataSource ds = (DataSource) ctx.lookup("jdbc/curso");
```

- El **`DataSource`**, representa a una base de datos. La información necesaria para crear las conexiones como el nombre de la base, del servidor, el driver, el port, etc., son propiedades de este objeto que se configuran en el servidor J2EE y son transparentes para la aplicación. Los mecanismos para configurar el DataSource son dependientes del contenedor J2EE.
- Un objeto que implementa la interface DataSource, típicamente será registrado con un servicio de nombres basado en JNDI. Esta funcionalidad es transparente para el programador.

La API JDBC
Estableciendo una conexión con DataSource

La interface **DataSource** provee 2 métodos para obtener una conexión:

- **getConnection**()
- **getConnection**(String **usr**, String **pwd**)

La **usr** y la **pas**, no son obligatorias

Lo único que necesita saber la aplicación es el nombre bajo el cual el objeto **DataSource** es almacenado en el servicio de directorios **JNDI**.

```
. . .
 try {
   InitialContext ctx=new InitialContex();
   DataSource ds=(DataSource) ctx.lookup("jdbc/curso");
   con = ds.getConnection();
   Statement st = con.createStatement();
   ResultSet rs= st.executeQuery("Select * from Usuarios");
   while (rs.next()) {
    System.out.println(rs.getString(1)+"-"+rs.getString(2));
   }
   rs.close(); st.close();
   con.close();  //en gral. devuelve la conexión al pool
 } catch( javax.naming.NamingException e){
     System.out.println("Error de Nombre"+e.getMessage());
 } catch (javax.sql.SQLException e) {
     System.out.println("Error de SQL"+e.getMessage());
 }
. . .
```

Se crea la conexión, se utiliza y se cierra

Se deben manejar 2 excepciones: una para controlar si se encuentra el **datasoruce** y otra para verificar si se realizó la conexión.

La API JDBC

La interface Connection

Las aplicaciones usan la interface Connection para especificar atributos de transacciones y para crear objetos **Statement**, **PreparedStatement** o **CallableStatement**. Estos objetos son usados para ejecutar sentencias SQL y recuperar resultados. Esta interface provee los siguientes métodos:

(permite definir si es ***READ_ONLY*** *(por defecto)* o ***UPDATABLE***)

```
Statement createStatement() throws SQLException
Statement createStatement(int resultSetType, int resultSetConcurrency)
```

(permite definir si es ***FORWARD***

```
throws  SQLException
```

(por defecto) o en ambas direcciones)

```
PreparedStatement preparedStatement(String sql) throws SQLException
PreparedStatement preparedStatement(String sql, int resultSetType,
                                    int resultSetConcurrency)
                                    throws SQLException
```

Sentencia SQL a ser ejecutada

```
CallableStatement prepareCall(String sql) throws SQLException
CallableStatement prepareCall(String sql,int resultSetType,
                              int resultSetConcurrency)
                              throws SQLException
```

Store Procedure a ejecutarse

La API JDBC
Objetos java.sql.Statement

Un objeto **Statement** se crea con el método **createStatement()** y se ejecuta con **executeUpdate()** o **executeQuery()** dependiendo del tipo de sentencias SQL:

Este objeto puede ser recorrido y actualizado.

ResulSet executeQuery(String sql) throws SQLException

Se usa para ejecutar la sentencia SQL: **SELECT**

cantidad de filas que se insertaron, borraron o actualizaron (0 si no hay actualización)

int executeUpdate(String sql) throws SQLException

Se usa para ejecutar sentencias SQL: **CREATE TABLE, INSERT, UPDATE y DELETE**

Creación de un objeto Statement

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

Ejecución de un objeto Statement

```
ResultSet resul=sent.executeQuery("select nombre,edad from empleados");
int res = sent.executeUpdate("insert into empelados values('Juan', 56)");
```

La API JDBC

Objetos java.sql.PreparedStatement

- Un objeto **`PreparedStatement`**, es un tipo de sentencia sql que se precompila, y puede ser utilizada repetidas veces sin recompilar -> **mejora la performance**.
- A diferencia de las sentencias tradicionales cuando se crean requieren de la sentencia SQL como argumento del constructor. Esta sentencia es enviada al motor de la base de datos para su compilación y cuando se ejecuta, no se recompila.
- Como la sentencia puede ejecutarse repetidas veces, no se especifican los parámetros en la creación, sino que se usamos **?** para indicar dondo iran los parámetros y se parametrizan a través de los métodos set<datatype>().

Creación de una sentencia preparada

Parámetros de la sentencia SELECT

```
PreparedStatement  p_sent  =
miConexion.prepareStatement("SELECT * FROM Empledos WHERE edad >? AND Sexo=?")
```

Configuración de los parámetros de la sentencia y ejecución

```
p_sent.clearParameters();
p_sent.setInt(1, 55);
p_sent.setChar(2, "F");
ResultSet resul = p_sent.executeQuery();
```

Opcional, para limpiar cualquier parámetro seteado anteriormente

Setea los parámetros de la sentencia pre-compilada

La API JDBC

Objetos java.sql.CallableStatement

- Un objeto **`CallableStatement`** provee una manera para llamar a stored procedures (SP) para cualquier DBMSs. Los **SPs** son programas almacenados y que ejecutan en el propio motor de la Base de Datos. Típicamente se escriben en el lenguaje propio de la base de datos, aunque es posible hacerlo en **Java**.
- Los SP's se parametrizan a través de los métodos set<datatype>() de la misma manera que las sentencias preparadas.

Creación y ejecución de un procedimiento almacenado (SP)

Sin parámetros:

```
CallableStatement miSP=miConexion.prepareCall("call SP_CONSULTA");
ResultSet resul = miSP.executeQuery();
```

Con parámetros:

```
CallableStatement miSP =miConexion.prepareCall("call SP_CONSULTA[(?,?)]");
miSP.setString(1, "Argentino");
miSP.setFloat(2, "12,56f");
ResultSet resul = miSP.executeQuery();
```

La API JDBC
Recuperación de resultados

El resultado de un **`executeQuery()`**, es devuelto en un objeto **`ResultSet`**. Este objeto contiene un *cursor* que puede manipularse para hacer referencia a una fila particular del **`ResultSet`**. Inicialmente se ubica en la posición anterior a la primera fila. El método **`next()`** avanza una fila.

Recorrer el ResultSet

```
boolean next()  throws SQLException
boolean previous() throws SQLException
boolean first()  throws SQLException
boolean last()  throws SQLException
boolean absolute(int pos) throws SQLException
```

Devuelven **true** si el cursor está en una fila válida y **false** en caso contrario

Devuelve **true** si el cursor está en una fila válida y **false** si pos es <1 o mayor que la cantidad de filas.

Recuperar y Actualizar campos del ResultSet

Los campos de cada fila del **`ResultSet`** puede obtenerse mediante su nombre o posición.

El método a usar depende del tipo de dato almacenado:

```
String getString(int indiceColum) throws SQLException
String getString(String nombreCol) throws SQLException
int getInt(int indiceCol) throws SQLException
int getInt(String nombreCol) throws SQLException
void updateString(int indiceColum, String y) throws SQLException
```

Si el el ResultSet es actualizable, se puede invocar el método *updateRow()*.

La API JDBC

Ejecución de *queries* y recuperación de resultados

```
String insertaEmple ="INSERT INTO Empleados VALUES('Gomez','Juan
                                                    Martin','10301', -100, )";
 try {
     Statement sent = miConexion.createStatement();
     // Si res = 1 pudo insertar
      int res = sent.executeUpdate(insertaEmple);
      sent.close();
      miConexion.close();
 }  catch (SQLException e1) { }
```

Válido también para executeQuery (selects) sobre PreparedStatements y CallableStatements

```
try {
   . . .
   Statement sent = miConexion.createStatement();
   ResultSet resul = sent.executeQuery("SELECT * FROM Empleados WHERE
                                                          Edad>55");
    // Si entra al while obtuvo al menos una fila
    while (resul.next()){
       out.println(resul.getString( "APELLIDO")+","+resul.getString("NOMBRES"));
    }
     miConexion.close();
} catch (SQLException e1) { }
```

El Patrón DAO

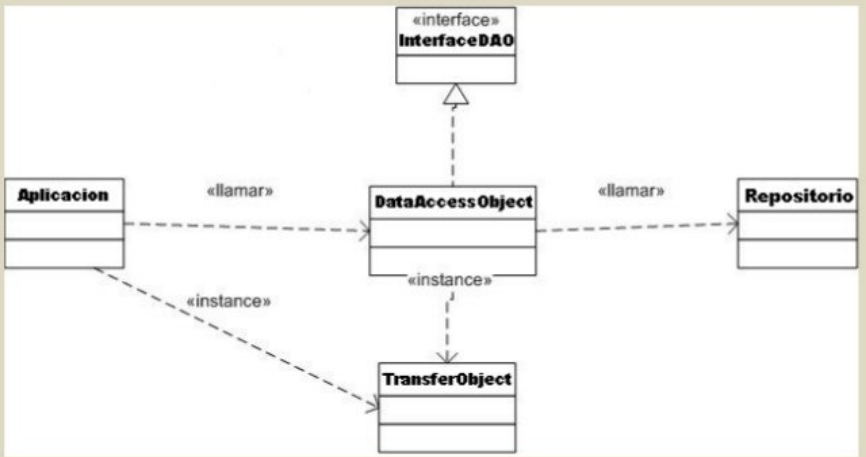
(Data Access Object)

El Patrón DAO
(Data Access Object)

- Es bastante normal hacer aplicaciones que almacenan y recogen datos de una base de datos. Suele ser habitual, también, querer hacer nuestra aplicación lo más independiente posible de una base de datos concreta y de cómo se accede a los datos.
- Nuestra aplicación debe conseguir los datos o ser capaz de guardarlos en algún sitio, pero no tiene por qué saber de dónde los está sacando o dónde se guardan.
- Hay una forma de hacer esto que ha resultado bastante eficiente en el mundo Java y de aplicaciones web, pero que es aplicable a cualquier tipo de aplicación que deba recoger datos de algún sitio y almacenarlos. Es lo que se conoce como patrón **DAO (Data Access Object)**.
- Las aplicaciones pueden utilizar el API JDBC para acceder a los datos de una base de datos relacional. Este API permite una forma estándar de acceder y manipular datos en una base de datos relacional.
- El API JDBC permite a las aplicaciones Java utilizar sentencias SQL, que son el método estándar para acceder a tablas y vistas. La idea del patrón **DAO (Data Access Object)** es ocultar la fuente de datos y la complejidad del uso de JDBC a la capa de presentación o de negocio.

El Patrón DAO
(Data Access Object)

- Un **DAO** define la relación entre la lógica de presentación y empresa por una parte y por otra los datos. El **DAO** tiene un interfaz común, sea cual sea el modo y fuente de acceso a datos.



- Resulta útil que el **DAO** implemente una interfaz. De esta forma los objetos de la aplicación tienen una forma unificada de acceder a los **DAO**.
- El **DAO** accede a la fuente de datos y la encapsula para los objetos de la aplicación. Entendiendo que oculta tanto la fuente como el modo (JDBC) de acceder a ella.
- El **TransferObject** encapsula una unidad de información de la fuente de datos. El ejemplo sencillo es entenderlo como un "bean de tabla", es decir, como una representación de una tabla de la base de datos, por lo que representamos las columnas de la tabla como atributos del **TransferObject**. El **DAO** crea un **TransferObject** (o una colección de ellos) como consecuencia de una transacción contra la fuente de datos. Por ejemplo, una consulta sobre Personas debe crear tantos objetos (**TransferObject**) de la clase Persona como registros de la consulta; el **DAO** devolverá la colección de **TrasnferObject** de la clase Persona al objeto de la aplicación. También puede ocurrir que el objeto de la aplicación mande un **TransferObject** para parametrizar una consulta o actualización de datos por parte del **DAO**.

El Patrón DAO
(Data Access Object)

- La idea de este patrón es sencilla. En primer lugar, debemos hacer una interface que contenga los métodos necesarios para obtener y almacenar, por ejemplo Personas. Esta interface no debe tener nada que la relacione con una base de datos ni cualquier dato específico del medio de almacenamiento que vayamos a usar.

Por ejemplo:

```
public interface IPersonaDAO {
   public List<Persona> getPersonas();
   public Persona getPersonaPorNombre(String nombre);
   ...
   public void guardarPersona(Persona persona);
   public void modificarPersona(Persona persona);
   public void borrarPersonaPorNombre(Persona persona);
   ...
}
```

```
public class Persona {

   private long id;
   private String nombre;
   private String apellido;
   private int edad;

   //getters y setters de los atributos
   ...
}
```

- Con esto deberíamos construir nuestra aplicación, usando la clase ***Persona*** y usando la interface ***IPersonaDAO*** para obtener y modificar Personas.

El Patrón DAO
(Data Access Object)

- Veamos de que manera se implementa la interface mencionada anteriormente:

```
public class PersonaDAO implements IPersonaDAO{
    ...
    public void guardarPersona(Persona persona){

                Connection con = getConexion();
                String orden = "INSERT INTO PERSONA VALUES (" +"'" + persona.getNombre + "' ") +
                                                        ", " + "'" + persona.getApellido + "' ") +
                                                        ", " + "'" + persona.getEdad + "' ") + ")";
                Statement sentencia = con.createStatement();
                sentencia.executeUpdate(orden);
                sentencia.close();

    }
    ...
}
```

El Patrón DAO
(Data Access Object)

Conclusión

- **Permite la Transparencia**
  Los objetos de negocio pueden utilizar la fuente de datos sin conocer los detalles específicos de su implementación. El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.
- **Centraliza Todos los Accesos a Datos en un Capa Independiente**
  Como todas las operaciones de acceso a los datos se ha delegado en los DAOs, esto se puede ver como una capa que aísla el resto de la aplicación de la implementación de acceso a los datos. Esta centralización hace que la aplicación sea más sencilla de mantener y de manejar.
- **Permite una Migración más Fácil**
  Una capa de DAOs hace más fácil que una aplicación pueda migrar a una implementación de base de datos diferente. Los objetos de negocio no conocen la implementación de datos subyacente, la migración implica cambios sólo en la capa de los DAOs.
- **Reduce la Complejidad del Código de los Objetos de Negocio**
  Como los DAOs manejan todas las complejidades del acceso a los datos, se simplifica el código de los objetos de negocio y de otros clientes que utilizan los DAOs. Todo el código relacionado con la implementación (como las sentencias SQL) están dentro del DAO y no en el objeto de negocio. Esto mejora la lectura del código y la productividad del desarrollo.

El Patrón DAO
(Data Access Object)

- ¿De que manera podemos crear objetos DAOs?
- Una solución a esto podría ser una factoría de objetos DAO, que se responsabiliza de instanciar el DAO adecuado (en nuestro ejemplo PersonaDAO)

```
public class FactoriaDAO{
   ...
   public static IPersonaDAO getPersonaDAO() {

        return new PersonaDAO();
   }
   ...
}
```

```
{
   ...
   IPersonaDAO dao = FactoriaDAO.getPersonaDAO();  // Obtengo el DAO de la factoria
   dao.guardarPersona(persona);                    // Guardo una persona en la base de datos
   ...
}
```