

# Persistencia de Datos

- 1 ¿qué es Persistencia?
- 2 ¿Dónde se ubica la capa de persistencia?
- 3 Tipos de persistencia con JAVA

## (1) Serialización

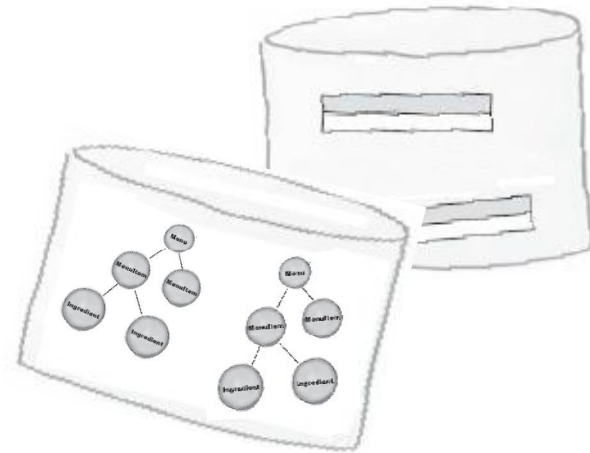
## (2) JDBC & SQL

Tipos de Drivers

La API JDBC

- Estableciendo una conexión: DriverManager & DataSource
- Sentencias SQL: objetos **Statement**, **PreparedStatement** y **CallableStatement**
- Soporte de Transacciones
- Manejo de Excepciones

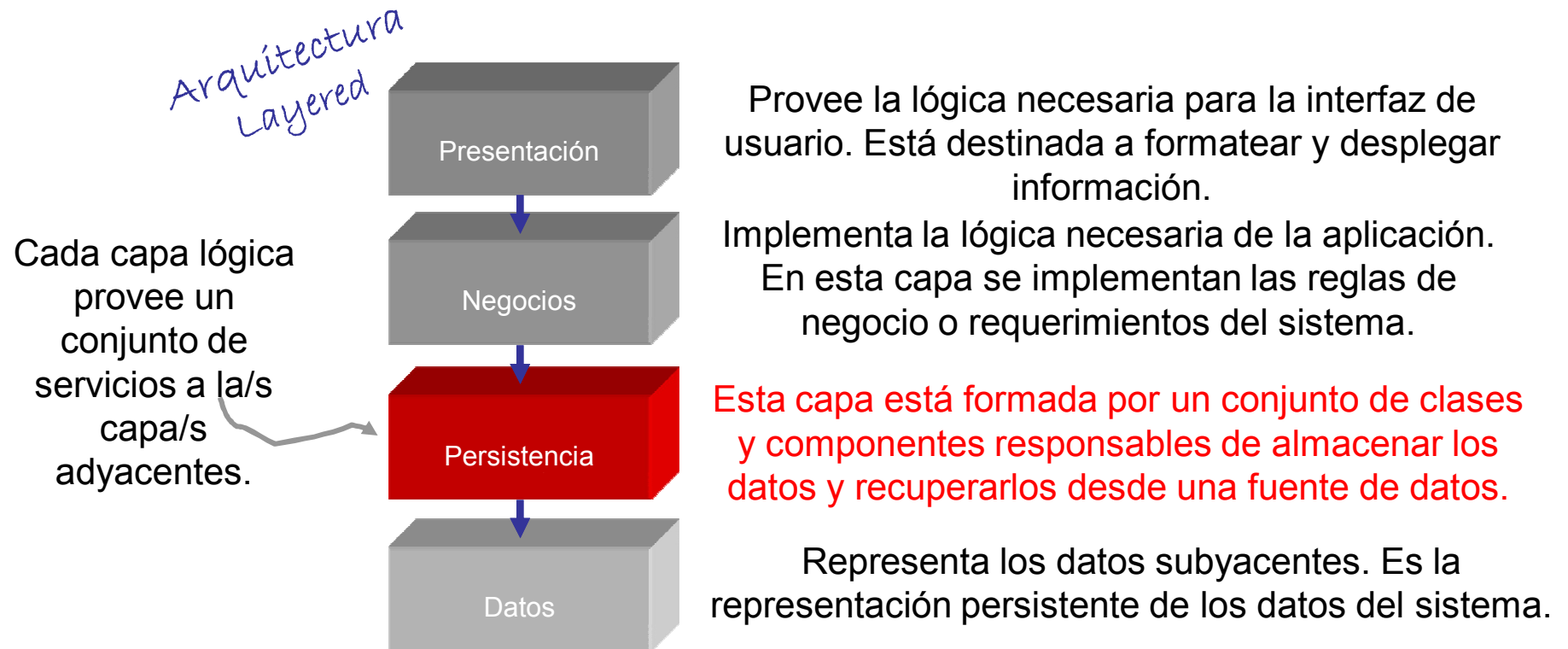
## (3) Object Relational Mapping. JPA & Hibernate



# Persistencia

## ¿dónde se ubica la capa de persistencia?

La mayoría de las aplicaciones pueden ser organizadas para que soporten un conjunto de capas lógicas o layers. Una arquitectura de alto nivel para una aplicación empresarial, usa estas capas:



**Persistencia** es el almacenamiento de datos desde la memoria (donde trabaja un programa) a un repositorio permanente. En aplicaciones Orientadas a Objetos, la persistencia le permite a un objeto, sobrevivir a la aplicación que lo creó. El estado de los objetos puede almacenarse en disco, y un objeto con el mismo estado, puede ser re-creado en el futuro.

# Persistencia

## Alternativas para la capa de Persistencia

En los sistemas orientados a objetos, los objetos pueden hacerse persistentes, de diferentes maneras. La elección del método de persistencia, es una parte importante del diseño de una aplicación. Cuando hablamos de persistencia en java, generalmente hablamos de almacenar datos en una Base de Datos usando SQL, sin embargo existen otros mecanismos a saber:

- **Serialización:** la serialización de objetos es el proceso de salvar el estado de un objeto a una secuencia de bytes (a un archivo .ser o para transmitir sobre la red). Esos *streams de bytes* pueden luego usarse para reconstruir el objeto original. Se lo utiliza para transferir estado de sesiones en clusters de servidores J2EE, y no para aplicaciones de escritorio.
- **JDBC y SQL:** JDBC es una interface de programación, que permite independizar 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. Son aconsejables cuando se trabaja con datos completamente tabulares o cuando se tiene que reutilizar *store procedures*.
- **ORM (object/relacional mapping):** es la persistencia automatizada y transparente de objetos pertenecientes a una aplicación java a tablas en una base de datos relacional, usando *metadata* que describen el mapeo entre los objetos y la base. ORM trabaja transformando datos desde una representación a otra.

# Persistencia

## (1) Serialización clásica

La **serialización** de objetos es el proceso de salvar el estado de un objeto a una secuencia de bytes, que pueden ser almacenados en un disco local o transmitirse sobre la red. Esos *streams* de bytes pueden luego usarse para reconstruir el objeto original.

```
public class SerializacionTest
{
    public static void main(String[] args) throws IOException {
        Persona p=new Persona();
        . . .
        try {
            FileOutputStream out = new FileOutputStream("persona.ser");
            ObjectOutputStream oos = new ObjectOutputStream(out);
            oos.writeObject(p);
            oos.flush();
        } catch (Exception e) {
            System.out.println("Problema serializando: " + e);
        }
        Persona p1 = null;
        try {
            FileInputStream in = new FileInputStream("persona.ser");
            ObjectInputStream ois = new ObjectInputStream(in);
            p = (Persona) ois.readObject();
        } catch (Exception e) {
            System.out.println("Problema serializando: " + e);
        }
    }
}
```

La clase del objeto a serializar, debe implementar la interface **java.io.Serializable** y se serializan todos aquellos campos no declarados *transient*

La serialización es un mecanismo de persistencia simple, el cual permite que un programa **lea/escriba** objetos desde/a un stream de bytes.

# Persistencia

## (2) SQL & JDBC

Antes de JPA/Hibernate, los programadores usaban directamente SQL y JDBC para lograr persistencia de sus datos.

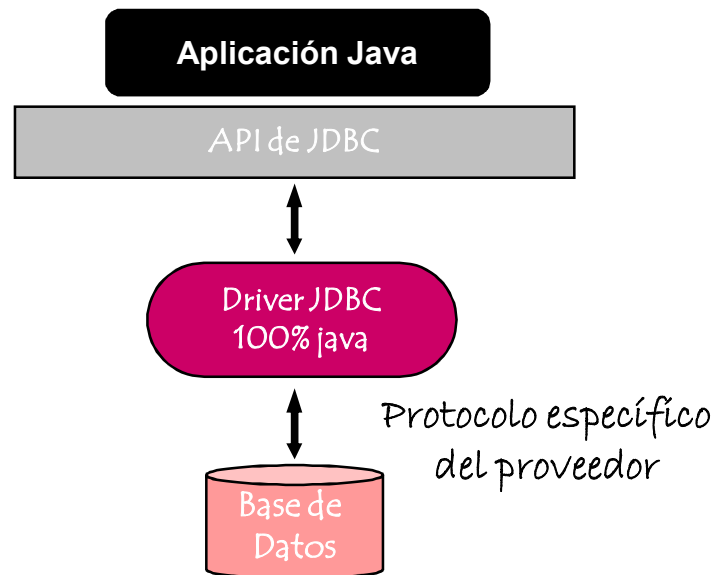
- La API JDBC provee un medio para acceder a una amplia variedad de fuentes de datos: DBMS, DBMSs orientadas a objetos, planillas de cálculo, etc. El único requerimiento es que exista el driver JDBC apropiado.
- La API JDBC provee una interface de programación única, que independiza a las aplicaciones 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 un proveedor de base de datos implementa como una pieza de código llamada driver. Un driver JDBC traduce las invocaciones JDBC genéricas en invocaciones específicas de una DB.
- La API JDBC es una parte integral de la plataforma Java. La especificación de JDBC 4.2 esta disponible en la JSR 221 y es parte de Java SE 8.

Año	Versión	JSR (Java Specification Request )	JDK
1997	JDBC 1.0	--	JDK 1.1
1999	JDBC 2.0	--	JDK 1.2
2001	JDBC 3.0	JSR 54	JDK 1.4
Mar, 2014	JDBC 4.2	JSR 221	Java SE 8

# JDBC – Tipos de Drivers

Existen diferentes tipos de drivers (tipo 1 al tipo 4), pero se recomienda utilizar el driver de tipo 4 porque:

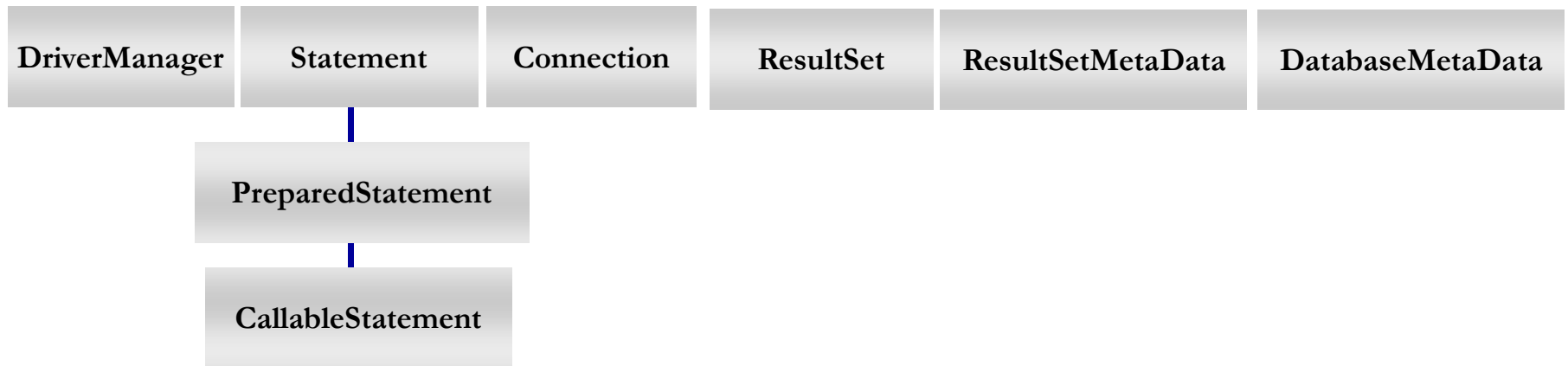
- Es un driver **Java Puro** que habla directamente con la base de datos.
- No requiere de ninguna librería adicional ni de la instalación de un *middleware*, como en el caso de los otros tipos.
- La mayoría de los fabricantes de Base de Datos proveen drivers JDBC de tipo 4 para sus Bases de Datos.



# La API JDBC

Las clases e interfaces de la API 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.

## java.sql (Core API)



---

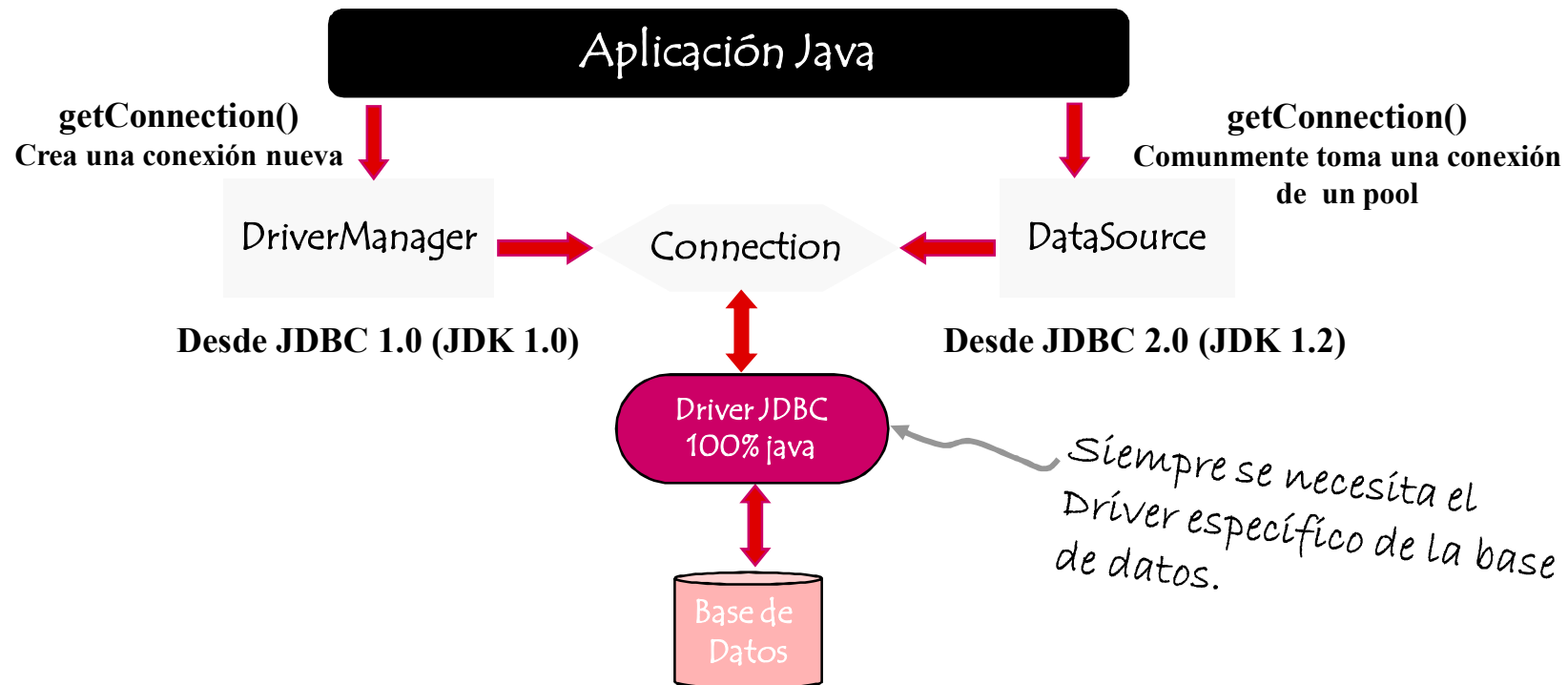
## javax.sql (Standard Extension)



# Persistencia

## SQL & JDBC

Las clases e interfaces de la API JDBC están en los paquetes **java.sql** y **javax.sql**. En estos paquetes se encuentran definidos métodos que permiten: conectarse a una base de datos, recuperar información relacionada con la base de datos, realizar consultas SQL, ejecutar Stored Procedures y trabajar con los resultados.



Una vez obtenido un objeto **Connection**, se pueden enviar comandos SQL desde la aplicación a la base de datos. Si la conexión no se puede establecer, se dispara una excepción SQL.



# Estableciendo una Conexión

## URL JDBC

Una base de datos en JDBC es identificada por una URL. 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

## DriverManager - DataSource

### La clase 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 y a partir de ahí intenta establecer una conexión.

```
miConexion = DriverManager.getConnection("jdbc:odbc:empleadosDB", usr, contra);  
  
miConexion = DriverManager.getConnection("jdbc:db2://server:50000/BASEEMPL");
```

- Es una clase que viene con la API, con lo cual, un proveedor no puede optimizarla. Mantiene internamente drivers JDBC y dada una URL JDBC retorna una conexión usando el driver apropiado.

### 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 (Drivers). Permite elegir las mejores técnicas para lograr un acceso óptimo a la base de datos y definir qué atributos son necesarios para crear las 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.

# 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.

```
...
Connection con;
try {
    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");
}
```

Se crea la conexión, se utiliza y se cierra

Se deben manejar 2 excepciones: una para controlar si el es encontrado el **driver** y otra para verificar si se estableció la conexión.

Nombre del Origen de Datos ODBC

URL JDBC

# La API JDBC

## Estableciendo una conexión con DataSource

Un objeto **DataSource**, representa 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.

### DataSources y JNDI

El Java Naming Directory Interface (JNDI) es una API java estándar que provee acceso a un servicio de directorio. Un servicio de 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 más JNDI, son la combinación ideal para obtener una conexión. Para nuestro propósito, el uso de JNDI es muy directo:

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

```
Context ctx = (Context)(new InitialContext().lookup("java:comp/env/"));  
DataSource ds = (DataSource) ctx.lookup("jdbc/MySQLDS");  
  
ds =(DataSource)new InitialContext().lookup("java:comp/env/jdbc/quecomemos");
```

Más información sobre Tomcat y JNDI:

<https://tomcat.apache.org/tomcat-8.0-doc/jndi-datasource-examples-howto.html>

# La API JDBC

## Configuración del Datasource

La configuración puede hacerse desde los entornos de desarrollo, visualmente o manualmente creando un archivo `context.xml` en la carpeta **META-INF** del **WebContent** (o en `src/main/webapp/resources` si se usa MAVEN) y modificando el archivo `web.xml` de la siguiente manera:

### **META-INF/context.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<Context debug="5" reloadable="true" crossContext="true">
  <Resource name="jdbc/MySQLDS" auth="Container" type="javax.sql.DataSource"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="root" password="lapass"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/unlp?autoReconnect=true"/>
</Context>
```

### **WEB-INF/web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
  javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>pruebaDataSource</display-name>
  <resource-ref>
    <res-ref-name>jdbc/MySQLDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

# La API JDBC

## Estableciendo una conexión con DataSource

La interface DataSource provee 2 métodos para obtener una conexión:

- `getConnection()`
- `getConnection(String usr, String pas)`

```
...  
try {  
    Context ctx =  
        (Context) (new InitialContext().lookup("java:comp/env/"));  
    DataSource ds = (DataSource) ctx.lookup("jdbc/MySQLDS");  
    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());  
}  
...  
}
```

Implementa la interface Context y provee el punto arranque para la resolución de nombres

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

Se crea la conexión, se utiliza y se cierra

Se deben manejar excepciones para controlar si se encuentra el **datasource** y 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)` throws `SQLException`

(permite definir si es **FORWARD** (por defecto) o en ambas direcciones)

Por ejemplo: `ResultSet.TYPE_FORWARD_ONLY`

`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 puede ejecutarse con `executeUpdate()` o `executeQuery()`.

<code>ResultSet executeQuery(String sql)</code>	Ejecuta la sentencia SQL que retorna un simple objeto <code>ResultSet</code> . Para sentencias <b>SELECT</b>
<code>int executeUpdate(String sql)</code>	Ejecuta la sentencia SQL que retorna un <code>int</code> . Para sentencias que crean base de datos, tablas, etc. o actualizan base de datos.

### Creación de un objeto `Statement`

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

### Ejecución de sentencias SQL

```
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 donde irán los parámetros y el método `set<datatype>()` para setear los valores.

### Creación de una sentencia preparada

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

Parámetros de la sentencia SELECT



### Configuración de los parámetros de la sentencia y ejecución

```
p_sent.clearParameters(); ← Opcional, para limpiar cualquier parámetro seteado anteriormente  
p_sent.setInt(1,55); ←  
p_sent.setChar(2,"F"); ← Seta los parámetros de la sentencia pre-compilada
```

```
ResultSet resul = p_sent.executeQuery();
```

# La API JDBC

## Objetos `java.sql.CallableStatement`

- Un objeto `CallableStatement` provee una manera para llamar a stored procedures (SP) para cualquier DBMSs. Los SP son programas almacenados 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 Store Procedures 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` pueden 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) { }
```

```
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) {
}
```

# La API JDBC

## Recuperación de parámetros de salida de SP

En el caso de Store Procedures que manejen parámetros, los mismos pueden ser IN, OUT o IN/OUT. Si el parámetro es de salida, entonces se lo debe indicar usando el método `registerOutParameter()` antes de que la sentencia sea ejecutada y los métodos `getXXX()` pueden ser usando para recuperar esos parámetros de salida (igual que como se lo hace con las preparedStatements).

Definiendo parámetros de entrada y de entrada/salida

```
CallableStatement miSP = miConexion.prepareCall("call getTestData(?,?)");
miSP.setByte(1, "12");
miSP.registerOutParameter(1, java.sql.Types.TINYINT);
miSP.registerOutParameter(2, java.sql.Types.DECIMAL,2);
ResultSet resul = miSP.executeQuery();
```

  
IN/OUT    OUT

```
// en el caso de los parámetros OUT se recuperan con métodos resul.getXXX()
byte x = miSP.getBytes(1);
Java.math.BigDecimal n = miSP.getBigDecimal(2);
```

# La API JDBC

## Soporte de transacciones

El objeto de tipo `java.sql.Connection` soporta el manejo de transacciones SQL. Una transacción SQL es un conjunto de sentencias SQL que deben ser ejecutadas como una unidad atómica. Para que la transacción sea exitosa cada sentencia debe serlo.

Cuando se crea un objeto `Connection`, automáticamente es configurado para que la ejecución de cada sentencia actualice en la base de datos (*commit* implícito). En este caso no se pueda deshacer dicha acción (no soporta *rollback*).

Con el objeto `Connection` se puede controlar cuando las sentencias SQL son efectivizadas (*committed*). Los siguientes métodos son provistos:

- `void setAutoCommit(boolean autoCommit)`: configura si las sentencias SQL son automáticamente *committed* o no. Si es `true` cuando una sentencia se ejecuta se efectiviza en la DB. Si es `false`, las sentencias no son *committed* hasta que no se ejecute el método `commit()`.
- `boolean getAutoCommit()`: devuelve el modo de la conexión, `true` si efectiviza cada sentencia o `false` si requiere el `commit()` explícito para actualizar.
- `void commit()`: es usado para efectivizar un conjunto de sentencias SQL.
- `void rollback()`: este método deshace todas las sentencias SQL ejecutadas después del último `commit()`.

# La API JDBC

## Soporte de transacciones

La Base de Datos es responsable de guardar las sentencias ejecutadas y deshacer todas las sentencias SQL ejecutadas después del último `commit()`, cuando encuentra el método `rollback()`.

```
try {  
    ...  
    miConexion.setAutoCommit(false);  
    Statement s = miConexion.createStatement();  
    int tran1 = s.executeUpdate("INSERT INTO TCuentas VALUES ('1001', -100)");  
    int tran2 = s.executeUpdate("INSERT INTO TCuentas VALUES ('1002', 100)");  
    if ((tran1>0) && (tran2>0)) {  
        miConexion.commit();  
    } else {  
        miConexion.rollback();  
    } catch (SQLException e1) {  
        ...  
    }  
}
```

Permite tener el control de qué y cuándo confirmar operaciones sobre la conexión.

```
try {  
    java.sql.Statement statement = miConexion.createStatement();  
    java.sql.ResultSet results = statement.executeQuery(  
        miConexion.commit();  
        miConexion.close();  
    } catch (Throwable t) {  
        miConexion.rollback();  
        miConexion.close();  
    }  
}
```

# La API JDBC

## Manejo de excepciones

Un objeto JDBC que encuentra un error serio (falla la conexión a la base, sentencias SQL mal formadas, falta de privilegios, etc.) dispara una excepción **SQLException**.

La clase **SQLException** extiende **java.lang.Exception** y define 3 métodos adicionales útiles:

- **getNextException()**: permite recorrer una cadena de errores sql.
- **getSQLState()**: devuelve un código de error SQL según ANSI-92.
- **getErrorCode()**: retorna un código de error específico del proveedor.

```
try {  
    // código de acceso a la base de datos  
} catch (SQLException e) {  
    while (e!=null){  
        System.out.println("SQL Exception:"+e.getMessage());  
        System.out.println("Error SQL ANSI-92:"+e.getSQLState());  
        System.out.println("Código de error del Proveedor:"+e.getErrorCode());  
        e = e.getNextException();  
    }  
}
```

Hasta las versiones JDBC 3.0, para todo tipo de excepción se disparaba **SQLException**.



# La API JDBC

## Manejo de excepciones

JDBC 4 ha mejorado el manejo de excepciones, incorporando:


- Clasificación de **SQLException**: no transitorias, representan fallas provocadas por alguna condición que debe ser resuelta antes de reintentar y transitorias, representan fallas que sin intervención podrían ser exitosas en otro intento.

SQLException

```
+----> SQLNonTransientException
| +----> SQLDataException
| +----> SQLFeatureNotSupportedException
| +----> SQLIntegrityConstraintViolationException
| +----> SQLInvalidAuthorizationException
| +----> SQLNonTransientConnectionException
| +----> SQLSyntaxErrorException
+----> SQLTransientException
| +----> SQLTimeoutException
| +----> SQLTransactionRollbackException
| +----> SQLTransientConnectionException
```

```
try {
    s.execute("Hola Mundo!!! ");
} catch (SQLSyntaxErrorException ex) {
    System.out.println("Problema con la Sintaxis SQL");
}
s.execute("create table testTable(id int,name varchar(8))");
try {
    s.execute("insert into testTable values (1,'Juan Moreno')");
} catch (SQLDataException ex) {
    System.out.println("Problema permanente con los datos de entrada");
}
```

- La clase **SQLException** implementa la **interface Iterable**, por lo que soporta la estructura de control for each (disponible a partir del J2SE 5.0) para recorrer las excepciones.

```
try {
    // código de acceso a la base de datos
} catch (SQLException e) {
    for (Throwable t: e) 
        System.out.println("Error SQL"+ t);
}
```