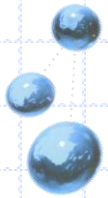


JDBC

Comunicándose con la Base de datos

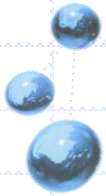
Profesores:

- Andrés Farías

fcfm

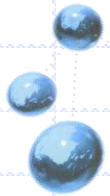
Entendiendo qué es y para qué sirve…

Introducción a JDBC



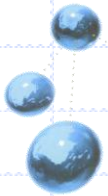
Introducción

- JDBC provee una librería para acceder a distintas bases relacionales y "normaliza" la mayor parte de las operaciones (las hace independientes de la base de datos utilizada y por tanto portables). Es parte de las distribuciones standard de Java
- JDBC no comprueba que las sentencias SQL son correctas, sencillamente las pasa a la base de datos



JDBC drivers

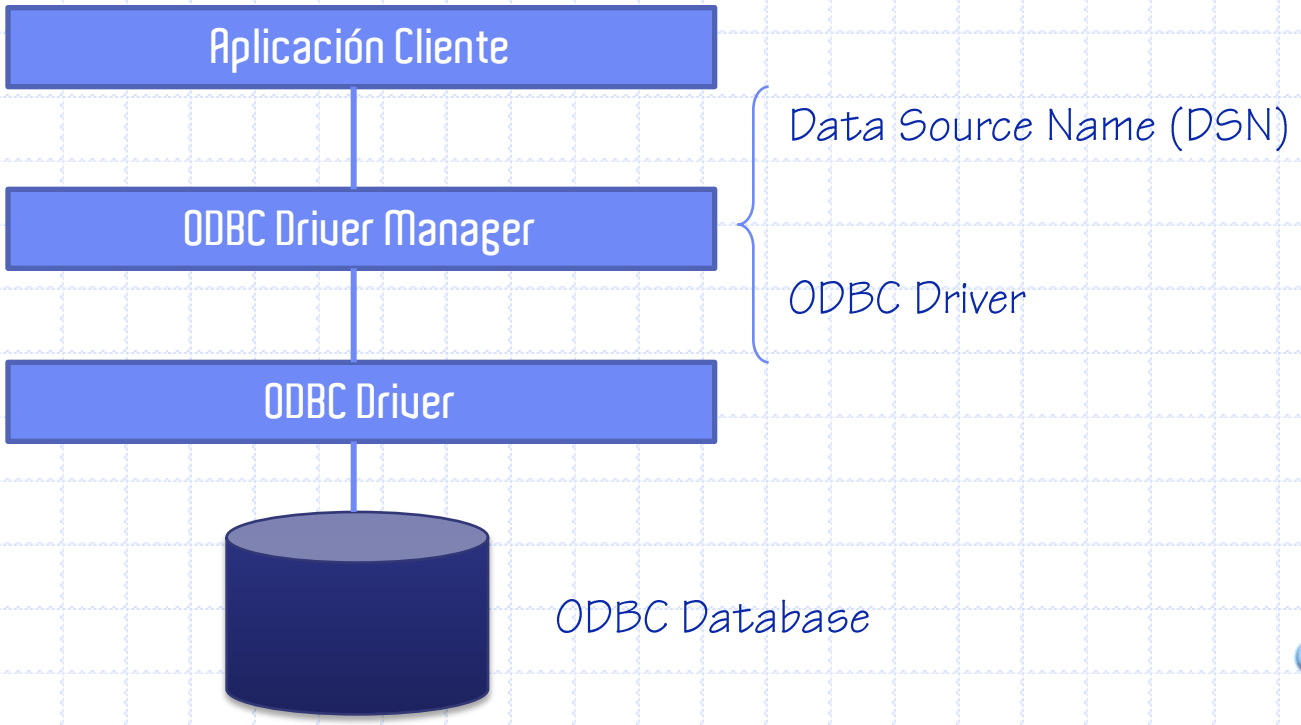
- JDBC consta de un
  - ✓ API genérica (escrita en Java)
  - ✓ Drivers específicos de cada base de datos que se encargan de las interacciones con la base de datos especificas de cada sistema que "habla" a la base con la base de datos.
- Tanto la API como los drivers se ejecutan en el cliente.



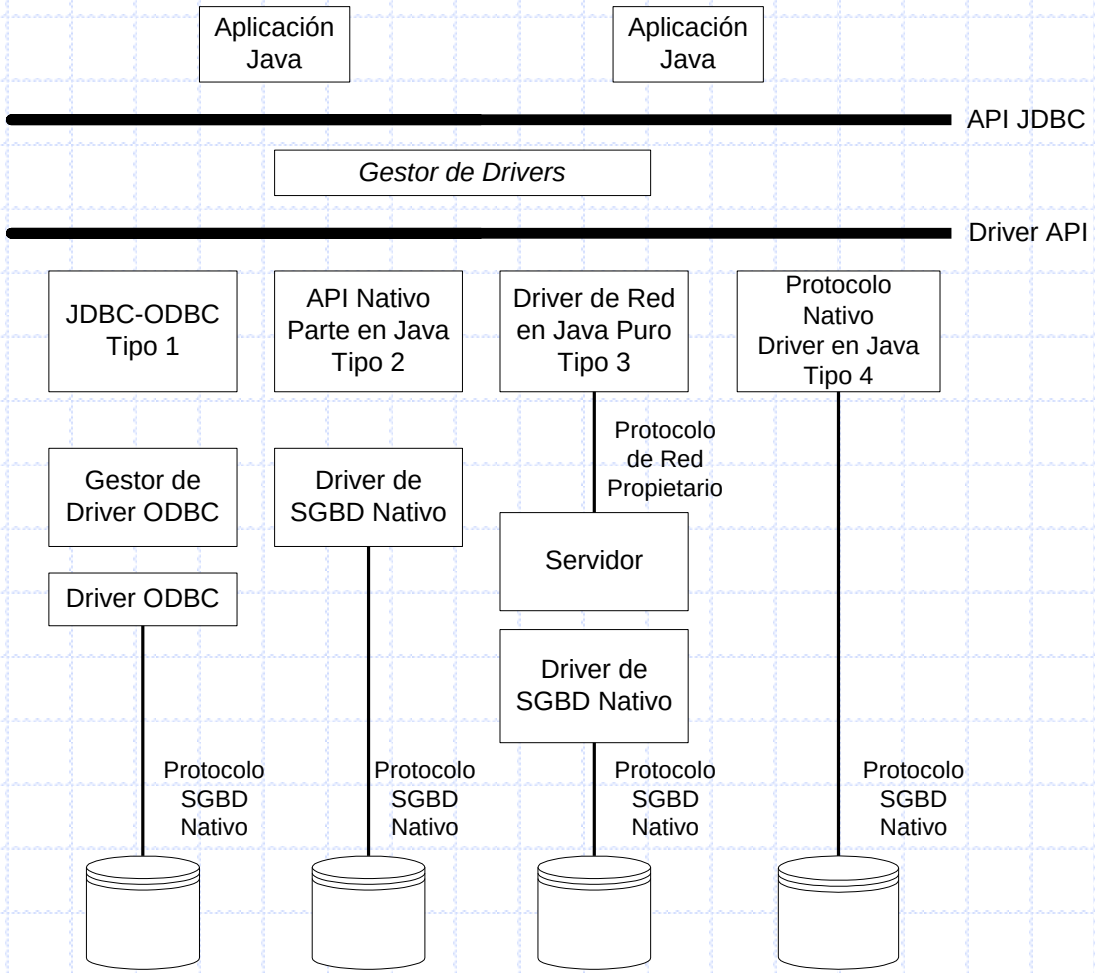
Conectividad ODBC

Open DataBase Connectivity

- Interface de aplicaciones (API) para acceder a datos en sistemas gestores de bases de datos utilizando SQL

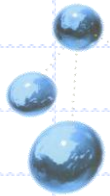


JDBC de Javasoft: arquitectura I



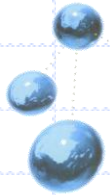
JDBC de Javasoft: arquitectura II

- Gestor de drivers es núcleo de arquitectura y comunica aplicaciones Java con cuatro tipos diferentes de drivers:
  - ✓ Puente ODBC-JDBC más el driver ODBC
    - Traduce JDBC en ODBC
    - No adecuado si parte cliente es anónima
      - Necesidad de registrar fuente ODBC en cada máquina
    - Uso: aplicaciones de dos capas en red local



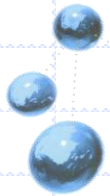
JDBC de Javasoft: arquitectura III

- ✓ API Nativo parte en Java:
  - Traduce JDBC en protocolo específico de BD: Oracle JDBC/OCI driver
  - Requieren algo de código nativo en cada máquina cliente en forma de un driver específico que se debe instalar
  - Uso: servidores RMI, servlets
- ✓ Driver de Red en Java Puro:
  - Utiliza protocolo comunicaciones publicado para la comunicación con un servidor remoto
  - Servidor se comunica con la BD utilizando ODBC o un driver nativo
  - Java puro: applets → se descarga lo mínimo
  - No requiere código instalado en máquina cliente



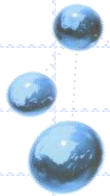
JDBC de Javasoft: arquitectura IV

- ✓ Protocolo nativo en Java Puro:
  - Implementan protocolo de BD de un suministrador
  - Específicos para BD concreta
  - No tienen necesidad de intermediarios
  - Alto rendimiento

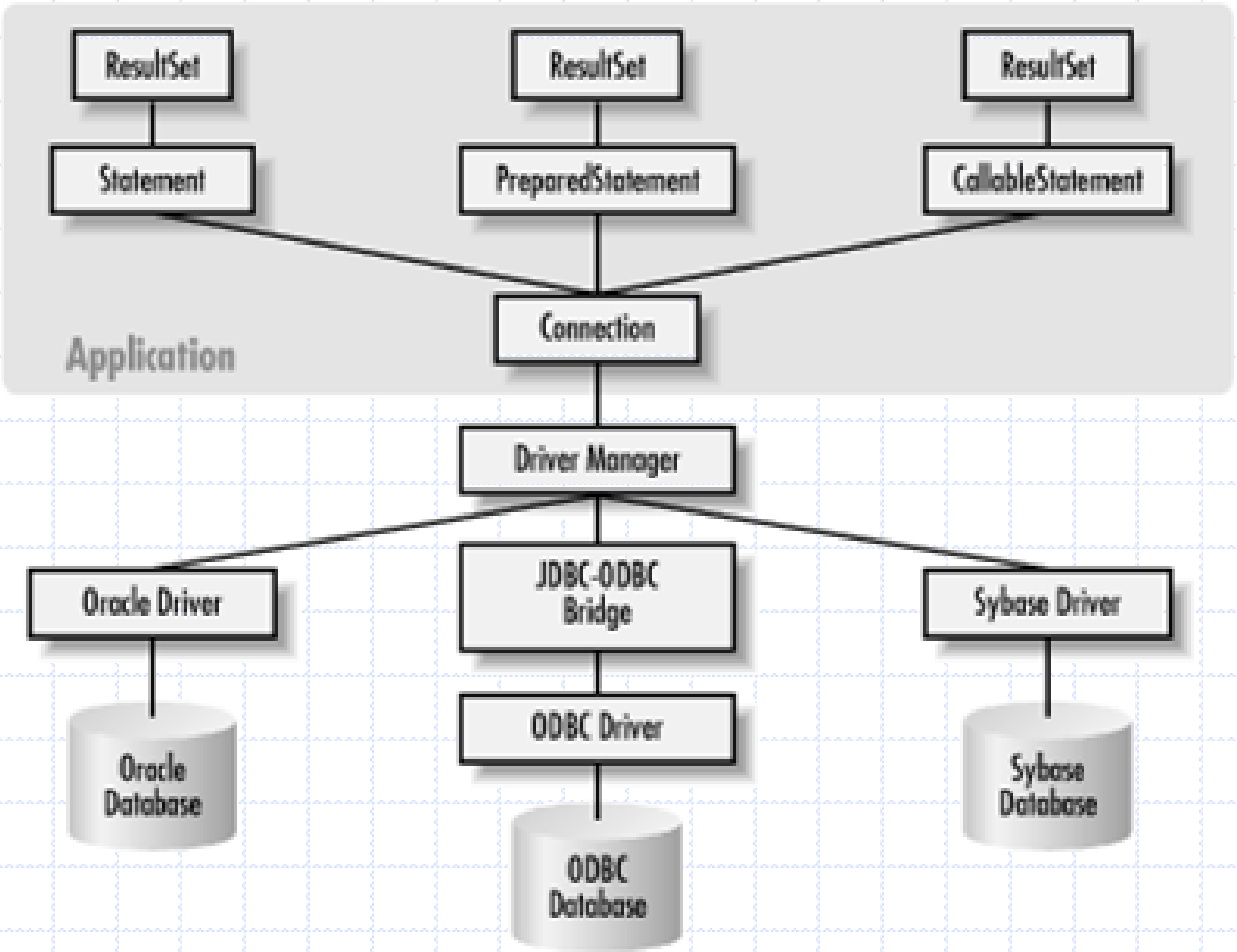


JDBC vs. ODBC

- ¿Por qué no usar ODBC desde Java?
  - ✓ Se puede usar ODBC desde Java
  - ✓ Puente JDBC-ODBC
- ¿Por qué se necesita JDBC?
  - ✓ ODBC no es apropiado para su uso directo desde Java porque usa una interface en C
  - ✓ Una traducción de la ODBC API en C a una API en Java no sería deseable
  - ✓ ODBC es duro de aprender
  - ✓ Una API en Java como JDBC es necesaria para conseguir una solución "puramente Java"
- JDBC API es una interface natural de Java

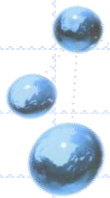


Controladores (Drivers) JDBC



Más informacion en:

- Sun JDBC web
  - ✓ http://java.sun.com/products/jdbc
- JDBC tutorial
  - ✓ http://java.sun.com/docs/book/tutorial/jdbc
- Lista de "drivers"
  - ✓ http://industry.java.sun.com/products/jdbc/drivers
- java.sql API
  - ✓ http://java.sun.com/j2se/1.4/docs/api/java/sql/package-summary.html

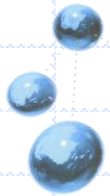


Preparándose para JDBC

JDBC 3.0 API

- La JDBC 3.0 API comprende 2 paquetes:
  - ✓ java.sql
  - ✓ javax.sql (añade capacidades de la parte servidor)

- Básicamente, los pasos a seguir son:
  - ✓ Registrar un driver:
    - Clase **`DriverManager`**
  - ✓ Establecer una conexión con la base de datos:
    - Interface **`Connection`**
  - ✓ Enviar sentencias SQL a la base de datos:
    - Interface **`Statement`**
  - ✓ Procesar los resultados de las consultas
    - Interface **`ResultSet`**



Cargar el driver

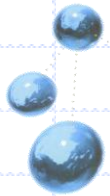
```
// cargar las clases relacionadas con sql
. . .
// carga el driver usando el metodo Class.forName()

try {
   Class.forName("org.postgresql.Driver");
} catch (ClassNotFoundException cnfe) {
    System.out.println("Couldn't find the driver!");
    System.out.println("Let's print a stack trace, and exit.");
    cnfe.printStackTrace();
    System.exit(1);
  }
   // Si el driver no esta disponible, el método forName()
   //emitira una excepcion del tipo  //ClassNotFoundException.
```

Cargar el driver II

- Cargar el driver requiere una única línea de código.
- La forma de cargar el driver mostrada en la transparencia anterior es la más común pero hace que el código no sea portable, si cambiamos de base de datos y la nueva elección no es postgreSQL.
- Otra posibilidad es especificar el driver necesario cuando se ejecuta el programa pasándolo mediante **`-Dargumento`**. Por ejemplo:

```
java -Djdbc.drivers=org.mysql.Driver myaplicacion
```



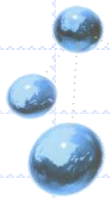
Definir la URL usada para establecer la conexión

- Una vez cargado el driver hay que solicitar una conexión a la base de datos, para ello se usa la clase **DriverManager** y una **URL** como el siguiente.

```
jdbc:postgresql:peliculas
```

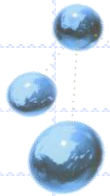
- Donde la sintaxis genérica es:

```
jdbc:<subprotocolo>:<subnombre>
jdbc:odbc:cliente
jdbc:msql://www.deusto.es:1112/catalogo
jdbc:mysql://localhost:3306/docman
```



Establecer la conexión

```
Connection connection;
connection =
  DriverManager.getConnection("jdbc:postgre
  sql:peliculas", "username","password");
```



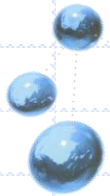
Opcionalmente conseguir información sobre la base

```
import java.sql.DatabaseMetaData
//cargar driver
//conectarse a la base
. . .
DatabaseMetaData dbMetaData = connection.getMetaData();

String productName = dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);

String productVersion = dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);

// Chequea si la tabla “actor” existe
ResultSet tables = dbMetaData.getTables(null, null, "actor", null);
if (tables.next()) {
     System.out.println("Table exists");
  }
```



Crear una Sentencia SQL y ejecutarla

- El siguiente paso requiere la creación de un objeto de la clase **Statement**. El se encargará de enviar la consulta en SQL a la base.

```
Statement statement = connection.createStatement();
//statement necesita una conexión abierta
```

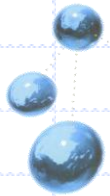
- Es necesaria una instancia activa de una conexión para crear el objeto **Statement**.
- JDBC devuelve los resultados en un objeto de la clase **ResultSet** .
- Finalmente, ejecutamos la consulta:

```
String query = "SELECT * FROM cia";
ResultSet resultSet = statement.executeQuery(query);
```

- El resultado de la consulta se almacena en resulset. No se debe escribir el **;** al final de la consulta.

Si se desea modificar la Base...

- Entonces se ejecuta:
  - ✓ **`executeUpdate(string);`** en lugar de **`executeQuery(string)`** (donde string contiene **`UPDATE`**, **`INSERT`** o **`DELETE`**);
  - ✓ **`SetQueryTimeout`** se puede usar para especificar cuanto se espera antes de abortar la transacción.



Statement se puede reutilizar

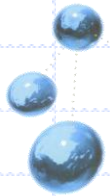
- No hace falta crear un objeto statement para cada consulta

```
statement.executeUpdate("INSERT INTO cia
   VALUES('Algeria','Africa',
   2381740.0,28539321.0,97100000000.0)");

statement.executeUpdate("INSERT INTO cia " +
                  "VALUES(American Samoa'," +
    "'Oceania',199.0,57366.0,128000000.0)");
```

Procesando los resultados

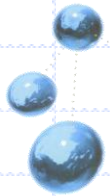
- El objeto **resultSet**, contiene 265 tuplas describiendo los distintos países. Para acceder a los distintos valores se leerán las tuplas una a una y accederemos a los atributos usando distintos métodos según el tipo de variable.
- El método **next** mueve el cursor a la siguiente tupla (sobre la que se opera) .
- Cada invocación al método **next** mueve el cursor una fila hacia delante



Procesar los resultados

```
//Los métodos usados no son los correctos para cia
while(resultSet.next()) {
 System.out.println(resultSet.getString(1) + " " +
                    resultSet.getDouble(2) + " " +
                    resultSet.getInt(3) + " " +
                    resultSet.getTimeStamp(4));
```

- La primera columna tiene índice 1 (no 0)
- **ResultSet** define varios métodos **getXxxx** que dado el número de columna devuelve el valor allí almacenado
- A partir de **JDBC 2.0**, es posible mover el cursor hacia atrás y/o incrementarlo en más de una unidad
  - ✓ `srs.absolute(4);`
  - ✓ `srs.relative(-3);`



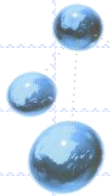
Procesar los Datos

- También se puede indexar usando atributos.

```
while(resultSet.next()) {
 System.out.println(resultSet.getString(1) + " " +
                    resultSet.getString(2) + " " +
                    resultSet.getInt(3)    + " " +
                    resultSet.getInt(4));
```

- Es equivalente a:

```
while(resultSet.next()) {
 System.out.println(resultSet.getString("nombre") + " " +
                    resultSet.getString("region") + " " +
                    resultSet.getInt   ("area")   + " " +
                    resultSet.getInt("poblacion"));
```

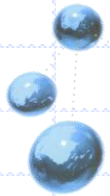


Correspondencia tipos SQL/Java

Tipo SQL	Tipo Java	Descripción
CHAR	string	Un carácter
VARCHAR	string	Cadena de caracteres de longitud variable
NUMERIC	java.math.BigDecimal	Valor numérico para cantidades económicas
BIT	boolean	Valor binario (0 o 1)
INTEGER	int	Entero de 32 bits con signo
REAL	float	Valor en punto flotante
DATE	java.sql.Date	Formato Fecha 'aaaa-mm-dd'
TIME	java.sql.Time	Formato Hora 'hh-mm-ss'

Cerrar la conexión

- `connection.close();`
- Abrir una conexión es caro así que debe posponerse el cierre de la conexión si se van a ejecutar más consultas a la misma base de datos.



Ejemplo Completo-I

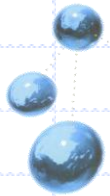
```
import java.sql.*;

public class TestDriver {
  public static void main(String[] Args) {
    try {
        //Constructor
        Class.forName("org.postgresql.Driver");}
        Connection connection =
          DriverManager.getConnection
         ("jdbc:postgresql:cia","roberto","pepino");
         Statement statement =
                  connection.createStatement();
         String consultaSQL = "SELECT 1+1";
```

Ejemplo Completo-II

```
ResulSet results = statement.executeQuery(consultaSQL);

int outp;
   while (results.next()) {
      outp = results.getInt();
      System.out.println("1+1 es: " + outp + "\n");
   }
}
cath(java.lang.Exception ex){
    System.out.println("Error: " + ex);
}
finally { connection.close(); } }}
```



Servlet con JDBC

```
public class RegistrationServlet extends HttpServlet {

  private String firstName = "";
  private String lastName = "";
  private String userName = "";
  private String password = "";

  public void init() {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      System.out.println("JDBC driver loaded");
    }
    catch (ClassNotFoundException e) {
      System.out.println(e.toString());
    }
  }

public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    sendPageHeader(response);
    firstName = request.getParameter("firstName");
    lastName = request.getParameter("lastName");
    userName = request.getParameter("userName");
    password = request.getParameter("password");
```

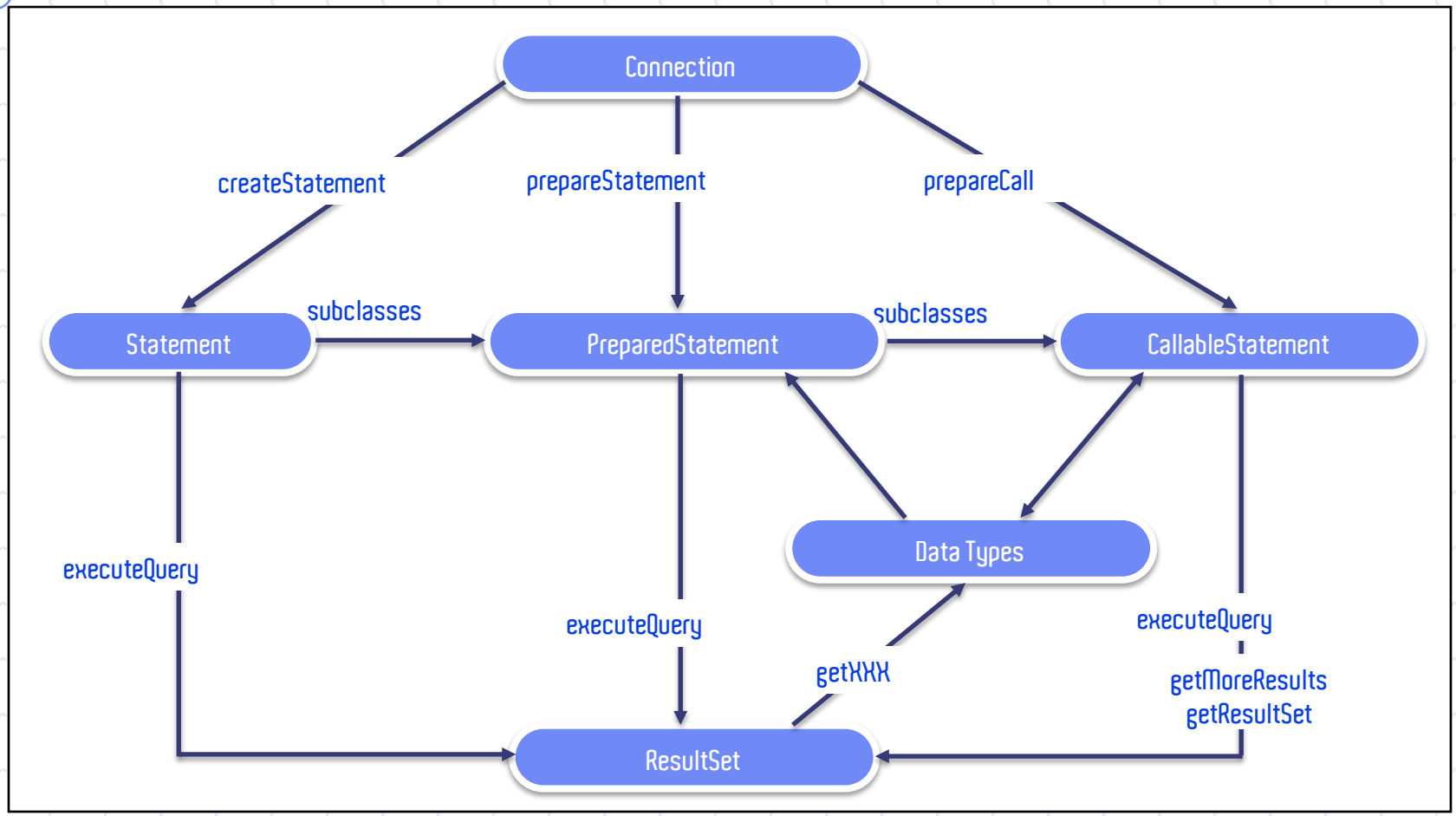
Servlet con JDBC

```
...
boolean error = false;
String message = null;
try {
  Connection con = DriverManager.getConnection("jdbc:odbc:UsersDB");
  System.out.println("got connection: " + password);

  Statement s = con.createStatement();

  String sql = "SELECT UserName FROM Users" +
    " WHERE userName='" + StringUtil.fixSqlFieldValue(userName) + "'";
  ResultSet rs = s.executeQuery(sql);
  System.out.println("Ejecutando select: ");
  if (rs.next()) {
    rs.close();
    message = "The user name <B>" + StringUtil.encodeHtmlTag(userName) +
      "</B> has been taken. Please select another name.";
    System.out.println(message);
    error = true;
  }
// ...
}
```

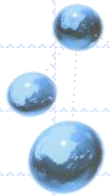
JDBC 3.0 API


Connection
createStatement
prepareStatement
prepareCall
Statement
subclasses
PreparedStatement
subclasses
CallableStatement
Data Types
executeQuery
executeQuery
executeQuery
getXXX
getMoreResults
getResultSet
ResultSet

Más sobre lo elemental

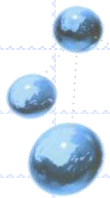
ResulSetMetaData

- Permite responder a preguntas como:
  - ✓ ¿Cuántas columnas hay en un ResultSet?
  - ✓ ¿Cuál es el nombre de una columna en particular?
  - ✓ ¿Qué tipo de datos contiene una columna en particular?
  - ✓ ¿Cuál es el tamaño máximo de una columna?



ResulSetMetaData: Métodos

- `getColumnCount`
  - ✓ número de columnas de la relación
- `getColumnDisplaySize`
  - ✓ máximo ancho de la columna especificada
- `getColumnName`
  - ✓ nombre de la columna
- `getColumnType`
  - ✓ tipo de datos (SQL) de la columna
- `isNullable`
  - ✓ ¿Esta columna puede contener NULLs?
  - ✓ Las posibles respuestas son `columnNoNulls`, `columnNullable`, `columnNullableUnknown`



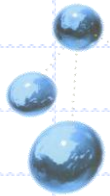
Ejemplo usando "Meta Data"

```
Connection connection=
 DriverManager.getConnection(url,username,password);

//Info sobre la base de datos
DatabaseMetaData dbMetaData = connection.getMetaData();

String productName = dbMetaData.getDatabaseProductName();
System.out.println("Database: " + ProductName);

String productVersion =
   dbMetaData.getDatabaseProductVersion();
```



Ejemplo usando "Meta Data" II

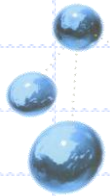
```
Statement statement = connection.createStatement();
String query = "SELECT * FROM cia";
ResultSet resultset = statement.executeQuery(query);

//Info sobre la tabla
ResultSetMetaData resultsMetaData =
   resultSet.getMetaData();
int columnCount = resultsMetaData.getColumnCount();

// El índice de las columnas empieza en 1
for (int i=1; i< columnCount ;i++)
  System.out.print( resultsMetaDAta.getColumnName(i) + " ");
```

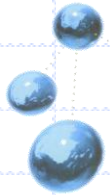
Statement

- A través de la clase **`Statement`**, se envían las consultas en SQL a la base
- Existen varios tipos de objetos "statement". Nos concentraremos en dos:
  - ✓ **`Statement`**: ejeccción de consultas sencillas.
  - ✓ **`PreparedStatement`**: ejecución de consultas precompiladas (parametrizadas).



PreparedStatement

- Si se van a ejecutar varias consultas similares el uso de consultas "parametrizadas" puede ser más eficiente
- Se crea la consulta, este se envía a la base de datos donde se optimiza antes de ser usada.
- Cuando se desee usar basta con remplazar los parámetros usando los métodos **setXxxx**.



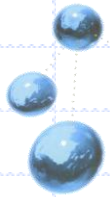
Prepared Statements

- Donde se ha usado **Statement** es generalmente posible usar **PreparedStatement** para hacer más eficientes las consultas
- Una instrucción con **PreparedStatement** va a ser, en la mayoría de los casos, traducida a una consulta SQL nativa de la base de datos en tiempo de compilación
- La otra ventaja es que es posible usar parámetros dentro de ella, pudiendo hacer más flexibles las consultas o hacer varias consultas distintas dentro de un ciclo cambiando el valor de algunas variables

```
PreparedStatement us = con.prepareSatatement("update alumnos
   set comuna = ?  where direccion like = ?);
   us.setString(1,'Vitacura')
   us.setString(2,'Hualtatas');
```

Prepared Statements: Ejemplo

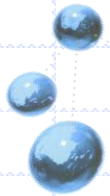
```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
                      "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast",
   "Espresso",  "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
   updateSales.setInt(1, salesForWeek[i]);
   updateSales.setString(2, coffees[i]);
   updateSales.executeUpdate();
  }
```



Transacciones

- Por defecto JDBC abre las conexiones en modo **autocommit**, esto es, cada comando se trata como una transacción
- Para controlar las transacciones se debe:
  - ✓ anular el modo **autocommit**

    ```
    connection.setAutoCommit(false);
    ```
  - ✓ llamar **commit** para registrar permanentemente los cambios
  - ✓ llamar a **rollback** en caso de error



Transacciones: ejemplo

```
Connection connection =
   DriverManager.getConnection(url,user,passwd);
connectio.setAutoCommit(false);
//setTransactionIsolation. NO commit explicito
try{
   statement.executeUpdate(…);
   statement.executeUpdate(…);
} catch (SQLExcepcione e){
    try{ connection.rollback();}
   catch(SQLExcepcion sqle) {  <ErrorMsg> }
  }
 try {
   connection.commit();
   connection.close();
  } catch (SQLException sqle) { <ErrorMsg> }
}
```

Preguntas?