

# BASES DE DATOS JDBC



ESCUELA DE INGENIERÍA INFORMÁTICA

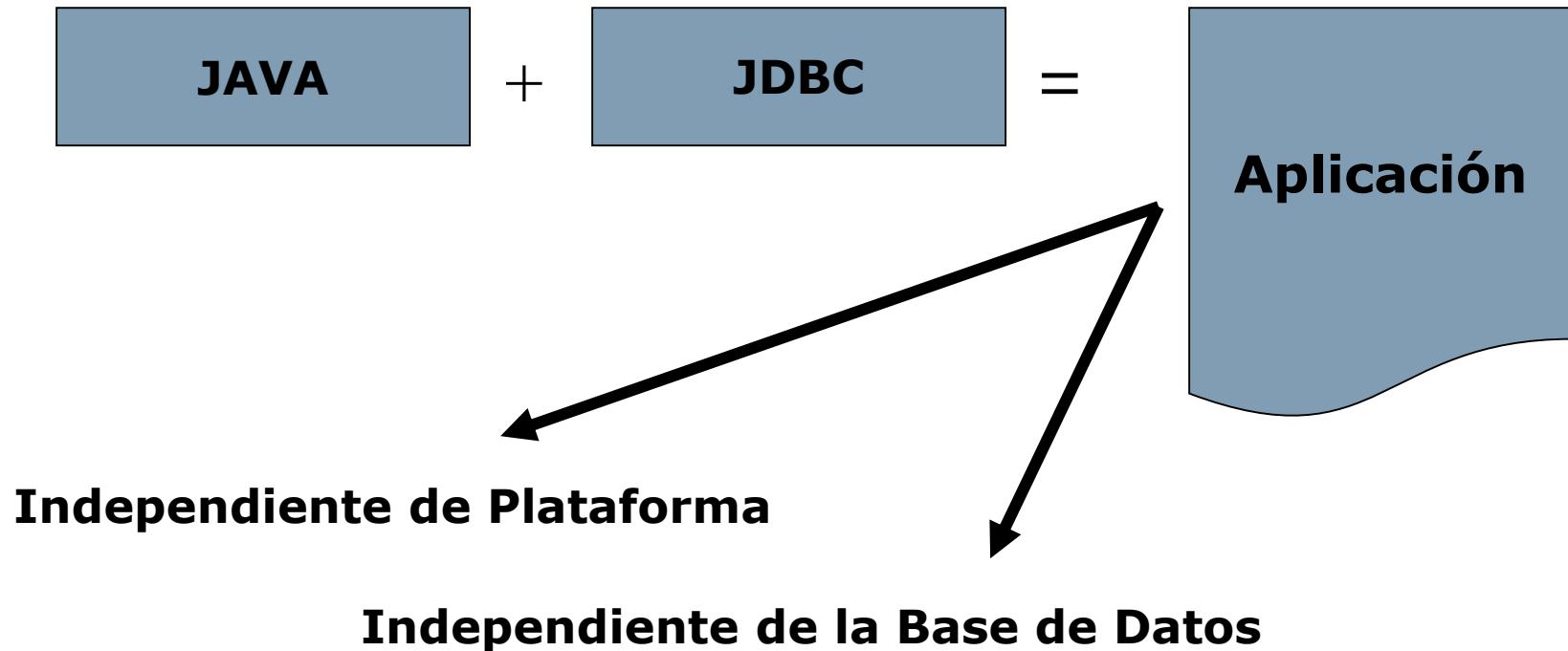
# Contenidos

- ⌘ 1. Introducción
- ⌘ 2. Conceptos básicos de programación JDBC
- ⌘ 3. Resumen de clases e interfaces empleados
- ⌘ 4. Transacciones
- ⌘ 5. Procedimientos Almacenados
- ⌘ 6. Excepciones

# ¿Qué es JDBC?

- ✎ JDBC es un conjunto de clases para la ejecución de sentencias SQL.
- ✎ Ha sido desarrollado conjuntamente por JavaSoft, Sybase, Informix e IBM entre otros.
- ✎ Permite manipular cualquier base de datos SQL. No es necesario un programa para manipular Oracle, otro Sybase, etc. Un mismo programa puede manipular cualquier base de datos.

# Principal Ventaja de JDBC



# Versiones

## ☞ JDBC 1.0

- Incluido en el JDK 1.1

## ☞ JDBC 2.0 (1998)

- Incluido en el JDK 1.2

## ☞ JDBC 3.0 (Febrero, 2002)

- Incluido en el Java 2 Standard Edition (J2SE), version 1.4

## ☞ JDBC 4.0 (Diciembre, 2006)

- Incluido en el Java Standar Edition (JSE) 6.0

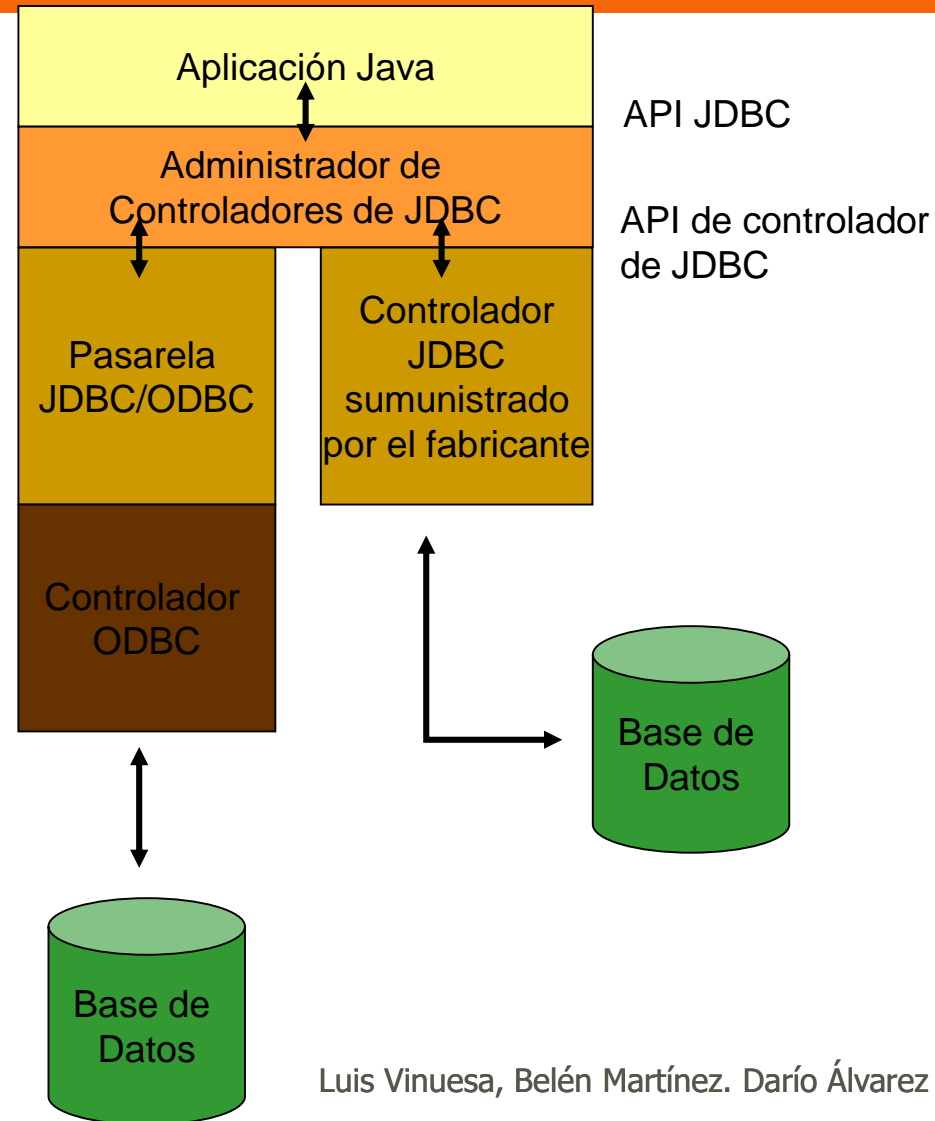
# Esquema Comunicación JDBC

## JDBC está compuesto de dos capas:

- API JDBC
- API del Administrador de controladores

## Los programadores tratan con la capa Java /JDBC

- Los programas escritos según la API JDBC deberían hablar con el administrador de controladores JDBC, el cual, por orden, podría usar aquellos controladores que estuvieran disponibles en el momento de conectar con la base de datos actual



# Contenidos

∞ 1. Introducción



∞ 2. Conceptos básicos de programación JDBC

∞ 3. Resumen de clases e interfaces empleados

∞ 4. Transacciones

∞ 5. Procedimientos Almacenados

∞ 6. Excepciones

# Programación JDBC – Conceptos Básicos

- ⌘ 1) Establecimiento de la conexión
- ⌘ 2) Ejecución de Comandos SQL
  - executeQuery
  - ResultSet
  - executeUpdate
- ⌘ 3) Sentencias predefinidas



# Establecimiento de la conexión

## 1) Conocer el nombre de las clases de los controladores

Es necesario conocer el nombre de las clases de los controladores JDBC utilizados por el fabricante

- ORACLE: `oracle.jdbc.OracleDriver` (anteriormente `oracle.jdbc.driver.OracleDriver`)
- HSQLDB: `org.hsqldb.jdbc.JDBCDriver`

# Establecimiento de la conexión

## (II)

## 2. Encontrar la librería donde se encuentra el controlador

- En el caso de Oracle 11 la librería es **ojdbc6.jar**
- En el caso de HSQLDB la librería es **hsqldb.jar**

∞ Hay que indicar la ruta completa a ese controlador en la ruta de clases. Para ello, tres mecanismos

- Lanzar los programas de bases de datos con el argumento de línea de comandos `-classpath`
- Modificar la variable de entorno `classpath`
- Copiar la librería de base de datos en el directorio `jre/lib/ext`.

# Establecimiento de la conexión

## (III)

### 3. Registrar el driver

- Antes de que el administrador de controladores pueda activar un driver, éste tiene que estar registrado. Una posibilidad para registrarlo es cargando su clase con el método *forName* o bien mediante el método *registerDriver* de la clase *DriverManager*

#### Ejemplo 1:

- `Class.forName("oracle.jdbc.OracleDriver");`
- `Class.forName("org.hsqldb.jdbc.JDBCDriver");`

#### Ejemplo 2:

- `DriverManager.registerDriver(new oracle.jdbc.OracleDriver())`
- `DriverManager.registerDriver(new org.hsqldb.jdbc.JDBCDriver())`

# Hacer la con

- Cuando se conecta con una base de datos se debe especificar el origen de los datos (y a veces otros parámetros)
- Para especificar el origen de los datos JDBC emplea una sintaxis similar a una URL **jdbc:nombre subprotocolo:otros elementos**
- Ejemplos:
  - jdbc:db2:MIBASEDATOS
  - jdbc:cloudscape:MIBASEDATOS;crcreate=true;
  - jdbc:odbc:MIBASEDATOS

## CONEXIÓN ORACLE:

- ✎ Una vez registrados los controladores, la conexión a la base de datos se abre de la siguiente forma:

Protocolo   Vendedor   Driver   Servidor   Puerto   Nombre bd

↑   ↑   ↑   ↑   ↑   ↑

```
String url = "jdbc:oracle:thin:@156.35.94.98:1521:DESA19";  
String username = "Usuario";  
String password = "Contraseña";  
Connection conn = DriverManager.getConnection(url, username, password);
```


- ✎ El administrador de controladores buscará uno que use el protocolo especificado por la URL de la base de datos, recorriendo todos los controladores registrados en él.

# Hacer la conexión

## CONEXIÓN HSQldb:

```
String url = "jdbc:hsqldb:hsqldb://localhost/labdb" ;
String username = "SA";
String password = "";
Connection conn = DriverManager.getConnection(url,username, password);
```

Nombre bd



# Ejecución de comandos SQL - executeQuery

- ✧ Para poder ejecutar un comando SQL en primer lugar es necesario crear un objeto `Statement`

```
Statement stat = conn.createStatement();
```

- ✧ Especificar la sentencia SQL a ejecutar:

```
String sentencia = " Select * from libros ";
```

- ✧ Ejecutar la instrucción

```
stat.executeQuery(sentencia);
```

- ✧ El método `executeQuery` es para lanzar instrucciones `Select` del lenguaje SQL

# Ejecución de comandos SQL - Resultados

- ✧ Cuando se ejecuta una consulta SQL, lo importante es el resultado.
- ✧ El objeto `executeQuery` devuelve un objeto de tipo `ResultSet` que se puede emplear para procesar las filas del resultado

```
ResultSet rs = stat.executeQuery ("Select * from libros");
```

- ✧ Para analizar el conjunto de resultados:

```
while (rs.next()) {  
    String isbn = rs.getString(1);  
    float precio = rs.getDouble("Precio"); }  
  
...
```

# Ejecución de comandos SQL - executeUpdate

## ∞ Crear el objeto **Statement**

```
Statement stat = conn.createStatement();
```

## ∞ Especificar la sentencia SQL a ejecutar:

```
String sentencia = "Update libros set price = price*1.25";
```

## ∞ Ejecutar la instrucción

```
stat.executeUpdate(sentencia);
```

## ∞ El método `executeUpdate` es para lanzar instrucciones SQL como INSERT, DELETE, UPDATE, CREATE TABLE, DROP TABLE; etc.



# Ejecución de sentencias predefinidas

McGraw-Hill  
Pearson  
O'Reilly

☞ Supongamos el siguiente ejemplo:

```
SELECT Libros.precio, Libros.titulo  
FROM Libros, Editoriales  
WHERE Libros.codEditorial = Editoriales.codEditorial and  
      Editoriales.Nombre=
```

- ☞ En lugar de construir una sentencia cada vez que el usuario lanza una consulta como ésta podemos prepararla como una variable y usarla muchas veces, rellenando previamente dicha variable con una cadena diferentes
- ☞ Esta técnica nos ofrece una ganancia en el rendimiento

# Ejecución de sentencias predefinidas (II)

- 1) Especificar la sentencia SQL identificando las variables de servidor con ?

```
String consulta1 = "SELECT Libros.precio, Libros.titulo "+  
    " FROM Libros, Editoriales "+  
    " WHERE Libros.codEditorial = Editoriales.CodEditorial and " +  
    " Editoriales.Nombre= ?";
```

- 2) Crear el PreparedStatement

```
PreparedStatement psConsulta= conn.prepareStatement(consulta1);
```

- 3) Enlazar cada variable con su correspondiente valor a través del **método set**.

```
psConsulta.setString(1,"McGrawHill");
```


- 4) Ejecutar la consulta

```
ResultSet rs = psConsulta.executeQuery();
```

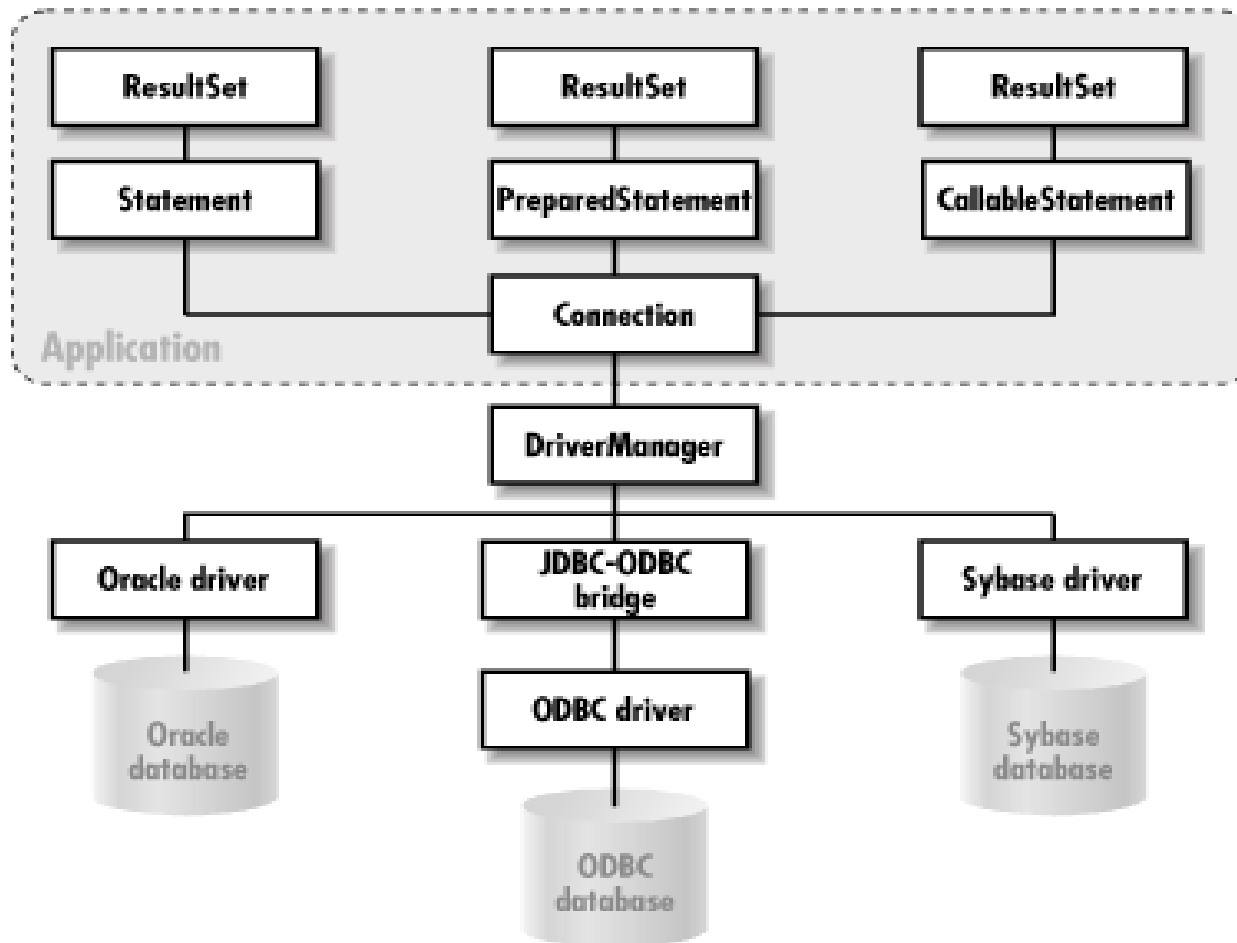
# JDBC- Ejemplo

```
String url = "jdbc:oracle:thin:@156.35.81.6:1521:DESA";  
Connection con = DriverManager.getConnection(url,"login","password");  
Statement st = con.createStatement();  
ResultSet rs = st.executeQuery("Select campo1,campo2,campo3 from Tabla1");  
while (rs.next())  
    String s = rs.getString("campo1");  
    int i     = rs.getInt("campo2");  
    float f   = rs.getFloat("campo3");  
}  
rs.close();  
st.close();  
con.close();
```

# Contenidos

- ✎ 1. Introducción
- ✎ 2. Conceptos básicos de programación JDBC
-  ✎ 3. Resumen de clases e interfaces empleados
- ✎ 4. Transacciones
- ✎ 5. Procedimientos Almacenados
- ✎ 6. Excepciones

# JDBC - Resumen de clases e interfaces principales



# JDBC - Resumen de clases e interfaces principales (II)

Las clases e interfaces que permiten trabajar y acceder a cualquier base de datos SQL se encuentran en el paquete `java.sql`.

Clases : `DriverManager`

Interfaces

- `Connection`
- `Statement`
- `PreparedStatement`
- `CallableStatement`
- `ResultSet`
- Otros: `Driver`, `DatabaseMetaData`, `ResultSetMetaData`
- ...

# Interface Driver

- ✧ El interface `Driver` especifica los métodos que todo driver JDBC debe implementar.
- ✧ Los Driver son los que permiten que una misma aplicación trabaje con distintos tipos de bases de datos (Oracle, Informix, Sybase, Access, etc.)
- ✧ Su misión es la de transformar las sentencias SQL emitidas por el programa al lenguaje nativo de la base de datos.
- ✧ Será necesario un Driver por cada tipo de base de datos que se desee utilizar
- ✧ No se suele trabajar directamente con esta clase.

# Clase DriverManager

- ✧ Agrupa los drivers y es el encargado de entregar el Driver adecuado para cada base de datos.
- ✧ El `DriverManager` conoce los drivers disponibles mediante el chequeo de la propiedad `sql.drivers`, que contiene la lista de drivers que se desea cargar.
- ✧ No obstante, también pueden incorporarse nuevos drivers una vez que el `DriverManager` se ha inicializado.



# Interface Connection (I)

- ✧ Representa una sesión abierta con una base de datos.
- ✧ Proporciona un contexto en el que emitir sentencias SQL y obtener resultados.
- ✧ Una aplicación puede tener varias conexiones a una misma base de datos y/o varias conexiones a distintas bases de datos.

# Interface Connection (II)

☞ Una conexión permite:

- Obtener `Statements` para ejecutar sentencias SQL
- Establecer el modo de las transacciones
- Obtener información sobre la base de datos (`DatabaseMetaData`)

# Interface Connection (III)

- ∞ La forma de obtener una conexión es:

`DriverManager.getConnection(url)`

- ∞ El `DriverManager` busca entre sus drivers áquel que sepa manejar el tipo de base de datos que se le indica.
- ∞ Si se encuentra un Driver adecuado el `DriverManager` le pide que cree una conexión a dicha base de datos y la entrega como valor de retorno. La aplicación no suele trabajar directamente con el Driver.
- ∞ De esta forma las aplicaciones trabajan con `Connections` sin preocuparse del tipo de base de datos con la que trabajan.

# Interface Statement

- ✧ Las instancias de `Statement` permiten la ejecución de sentencias SQL y la obtención de los resultados.
  - Se crean con `Connection.createStatement()` ;
- ✧ Hay dos formas principales de lanzar las sentencias SQL:
  - `executeQuery`, para sentencias que producen tuplas como resultado. Ej. `Select`.
  - `executeUpdate`, para sentencias que devuelven un entero o no devuelven nada. Ej. `Insert`, `Delete`, `Update`, `Create`, `Drop`, ...
- ✧ Una vez que el resultado haya sido procesado se debe cerrar el `Statement` (`close()`) .

# Interface ResultSet

- ☞ Ofrece métodos para manipular el resultado de la consulta.
- ☞ Mantiene un cursor a la fila actual. Cada vez que se invoca el método `next` se pasa a la siguiente fila.
- ☞ Se obtienen los valores de la fila actual utilizando las distintas versiones del método `getXXXX(<columna>)` pudiendo indicar la columna mediante su nombre o mediante su posición.
- ☞ El método `getMetaData` nos devuelve una instancia de la clase `ResultSetMetaData` con la cual podemos obtener información sobre el `ResultSet` y sus columnas.

# Interface PreparedStatement (I)

- ☞ Permite lanzar comandos SQL precompilados.
- ☞ En sentencias que van a ser ejecutadas múltiples veces se obtiene un aumento de rendimiento al tener la consulta ya analizada y optimizada.
- ☞ La sentencia SQL contiene uno o más parámetros (indicados mediante ?) que podrán ser modificados en distintas ejecuciones de la sentencia.

```
PreparedStatement ps = con.prepareStatement("DELETE  
FROM Table1 WHERE campo=?");
```

# Interface PreparedStatement



- ⌘ Antes de que la sentencia SQL pueda ser ejecutada todos los parámetros deberán tener asignado algún valor.
- ⌘ Para asignar el valor se utiliza el método `set<tipo>()`, siendo el tipo compatible con el parámetro.
- ⌘ Este método `set<tipo>()` lleva dos argumentos:  

```
ps.setInt(2, 23)
```


  - La posición del parámetro a asignar dentro de la sentencia
  - El valor a asignar al parámetro.
- ⌘ Finalmente se lanza la sentencia: `ps.executeUpdate()` o bien `ps.executeQuery()`

# JDBC- Ejemplo

```
public static void main (String[] args){
try{
    String url = "jdbc:oracle:thin:@156.35.81.6:1521:DESA";
    Connection con = DriverManager.getConnection(url,"login","password");
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("Select campo1,campo2 from Tabla1");
    while (rs.next()){
        System.out.print(rs.getString("campo1"))
        System.out.print(rs.getInt("campo2"));
        System.out.println(" ");
    }
    rs.close();
    st.close();
    con.close();
}
catch (java.lang.Exception ex){
    ex.printStackTrace();
}
} Bases de Datos
```



# Contenidos

- ⌘ 1. Introducción
- ⌘ 2. Conceptos básicos de programación JDBC
- ⌘ 3. Resumen de clases e interfaces empleados
-  ⌘ 4. Transacciones
- ⌘ 5. Procedimientos Almacenados
- ⌘ 6. Excepciones

# Transacciones

- ✧ Una transacción es un conjunto de sentencias que se deben completar o anular en su totalidad.
- ✧ Cuando se inicia una transacción la forma de hacerla definitiva es mediante el método `commit`.
- ✧ Si se desean deshacer todas las sentencias de la transacción hay que invocar el método `rollback`.
- ✧ Las conexiones están por defecto en modo `autoCommit`, es decir, las sentencias se auto-confirman llamando a `commit` cuando finalizan (por lo tanto, cada sentencia es una transacción).

# Transacciones (II)

- El modo `autoCommit` se activa/desactiva con el método `setAutoCommit`.
- Si el modo `autoCommit` está desactivado las transacciones no finalizarán hasta que se llame a `commit` o `rollback` explícitamente.
- A su vez dichos métodos inician una nueva transacción, por lo que una transacción serán todas aquellas sentencias ejecutadas entre dos `commit` y/o `rollback` consecutivos.

```
conn.setAutoCommit(false);  
  
Statement stat = conn.createStatement();  
stat.executeUpdate(commando1);  
stat.executeUpdate(comando2);  
...  
conn.commit(); o bien conn.rollback();
```

# Transacciones (III)

- ✎ Cuando un proceso está realizando una transacción (no se sabe si la confirmará o no) y otro intenta acceder a filas afectadas, será el programador el encargado de decidir como debe comportarse la conexión.
- ✎ Método `setTransactionIsolation (<tipo>)`.
  - `READ_UNCOMMITTED`
  - `READ_COMMITTED`
  - `REPEATABLE_READ`
  - `SERIALIZABLE`

# Contenidos

- 1. Introducción
- 2. Conceptos básicos de programación JDBC
- 3. Resumen de clases e interfaces empleados
- 4. Transacciones
- 5. Procedimientos Almacenados
- 6. Excepciones



# Procedimientos almacenados

- ✎ Un procedimiento almacenado es un grupo de enunciados SQL que forman una unidad lógica y hacen una tarea concreta
- ✎ Se emplean para encapsular un conjunto de operaciones o consultas que se van a ejecutar sobre un servidor de base de datos
- ✎ Pueden ser ejecutados y compilados con diferentes parámetros y resultados, y pueden tener cualquier combinación de parámetros de entrada, de salida y de entrada/salida
- ✎ La mayoría de los SGBD tienen soporte. Mucha variación en cuanto a sintaxis y posibilidades.

# Invocación procedimiento almacenado

- Es posible invocar un procedimiento almacenado desde una aplicación escrita en Java.
- Para ello hay que crear un objeto `CallableStatement`, que contiene la llamada al procedimiento almacenado y no el procedimiento mismo

```
CallableStatement cs = con.prepareCall("{call MUESTRA_EDITORIALES}");  
cs.execute();
```

Si el procedimiento contiene más de un enunciado SQL, se puede producir más de un conjunto de resultados, o más de un update, etc. En este caso donde hay múltiples resultados se debe invocar el método `execute`

- Un `callableStatement` puede devolver
  - `ResultSet`
  - Valores discretos en parámetros OUT e INOUT

# Invocación procedimiento almacenado (II)

## Invocación de procedimiento con parámetros de entrada (IN)

- Se tratan igual que en los PreparedStatement

```
CallableStatement cs = con.prepareCall ("{call MUESTRA_EDITORIALES (?,?) }");  
cs.setString(1,"xyz");  
cs.setString(2,"ab");  
cs.execute();
```



# Invocación procedimiento almacenado (III)

## Invocación de procedimiento con parámetros de salida (OUT)

### Los parámetros OUT

- Se referencian también por posición
- Deben ser registrados antes de la ejecución (`registerOutParameter`)

```
CallableStatement cs = con.prepareCall ("{call CALCULA_MEDIA (?) }");  
cs.registerOutParameter(1, java.sql.Types.FLOAT);  
cs.execute();  
float media = cs.getFloat(1);
```

# Invocación procedimiento almacenado (IV)

## Invocación de procedimiento con parámetros de salida (INOUT)

### Los parámetros INOUT

- Se referencian también por posición
- Combina los métodos anteriores:
  - Método `set<TIPO> (<pos>, <valor>)`
  - `registerOutParameter (...)`
  - `Execute [<modo>] ()`
  - Método `get<TIPO> (<pos>)`

### Ejemplo

```
CallableStatement cs = con.prepareCall ("{call ACTUALIZA_PRECIOS (?) }");
cs.setByte(1, (byte)25);
cs.registerOutParameter(1, java.sql.Types.TINYINT);
cs.execute();
byte x = cs.getBytes(1);
```

# Invocación de una función

## Invocación de una función en HSQLDB

∞ Ejemplo de invocación a una función con un parámetro de entrada: “ Invocación a una función que devuelve la media para un determinado alumno cuyo dni es pasado como parámetro a la misma “

```
CallableStatement cs = con.prepareCall ("{call CALCULA_MEDIA (?)}");
cs.setString(1,mdni);
if (cs.execute()){
    ResultSet rs = cs.getResultSet();
    rs.next();
    float media = cs.getFloat(1);
    System.out.println("La media es : "+ media);
}
cs.close();
```

**IMPORTANTE:** La invocación de funciones se realiza de forma diferente en ORACLE y HSQLDB

# Invocación de una función (II)

## Invocación de una función en ORACLE

- ✎ Ejemplo de invocación a una función con un parámetro de entrada : " Invocación a una función que devuelve la media para un determinado alumno cuyo dni es pasado como parámetro a la misma "

```
CallableStatement cs = con.prepareCall (" {? = call CALCULA_MEDIA (?) }");  
cs.registerOutParameter(1, java.sql.Types.FLOAT);  
cs.setString(2, mdni);  
cs.execute();  
float media = cs.getFloat(1);  
System.out.println("La media es : "+ media);
```

# Contenidos

- ✎ 1. Introducción
- ✎ 2. Conceptos básicos de programación JDBC
- ✎ 3. Resumen de clases e interfaces empleados
- ✎ 4. Transacciones
- ✎ 5. Procedimientos Almacenados
- ✎ 6. Excepciones



# Excepciones y avisos de error

- ✧ Las condiciones de excepción de SQL se convierten en excepciones `SQLException` de Java
- ✧ Las excepciones `SQLException` se encadenan juntas y se puede acceder a las siguientes mediante el método `getNextException`
- ✧ Los objetos `Connection`, `Statement` y `ResultSet` tienen disponibles las condiciones de aviso de error (*warning*) de SQL mediante el método `getWarnings`.
- ✧ Estos avisos de error se encadenan juntos y se puede acceder a los siguientes mediante el método `getNextWarning`

# Bloques try ...catch

☞ Java requiere que cuando un método lanza una excepción exista algún mecanismo que lo maneje:

```
try{
...
} catch (SQLException ex){
    System.err.println("SQLException:"+ex.getMessage());
}
try{
    Class.forName("myDriverClassName");
} catch (java.lang.ClassNotFoundException e){
    System.err.print("ClassNotFoundException : ");
    System.err.println(e.getMessage());
}
```

# SQLException

## ☞ Principales métodos:

- `String getSQLState()`, identifica el error de acuerdo a X/Open
- `int getErrorCode()`, obtiene el código de excepción específico del fabricante
- `String getMessage()`, obtiene una cadena que describe la excepción
- `SQLException getNextException()`, obtiene la excepción encadenada de ésta

`try{...`

```
} catch (SQLException ex) {  
    System.out.println(" SQLException recogida: ");  
    while (ex!=null){  
        System.out.println("Mensaje: "+ex.getMessage());  
        System.out.println("SQLState: "+ex.getSQLState());  
        System.out.println("ErrorCode: "+ex.getErrorCode());  
        ex=ex.getNextException();  
        System.out.println(" ");  
    }  
}
```



# Warnings

- ✧ Los objetos `SQLWarning` son una subclase de `SQLException` que tratan los avisos en el acceso a las bases de datos
- ✧ A diferencia de las excepciones los *warnings* no interrumpen la ejecución de una aplicación
- ✧ Los *warnings* pueden ser reportados sobre objetos `Connection`, `Statement` (tanto `PreparedStatement` como `CallableStatement`) o un `ResultSet`.
  - Cada una de las clases tiene un método `getWarnings` que debe ser invocado con el objetivo de ver el primer *warning* producido
  - Si `getWarnings` devuelve un *warning* se puede invocar al método `getNextWarning` para recoger cualquier *warning* adicional que se haya producido

# Warnings - Ejemplo

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("Select * from libros");
While (rs.next()){
    ...
    SQLWarning warning =stmt.getWarnings();
    if (warning!=null){
        System.out.println("Mensaje: "warning.getMessage());
        System.out.println("SQLState: "+warning.getSQLState());
        System.out.println("ErrorCode: "+warning.getErrorCode());
        warning=warning.getNextWarning();
        System.out.println(" ");
    }

    SQLWarning war =rs.getWarnings();
    if (war!=null){
        System.out.println("Mensaje: "war.getMessage());
        System.out.println("SQLState: "+war.getSQLState());
        System.out.println("ErrorCode: "+war.getErrorCode());
        war=war.getNextWarning();
        System.out.println(" ");
    }
}
```