

Bases de Datos 3

- Docentes: Federico Gómez, Diego Siri
- Carreras: Licenciatura & Ingeniería en Informática
- Año de la carrera: 3º

Capítulo 2:

Acceso a BD desde

Lenguajes de

Programación

Introducción:

En este capítulo hablaremos de los mecanismos utilizados para acceder a **Bases de Datos** relacionales desde **Lenguajes de Programación**.

En cursos anteriores se estudiaron conceptos teóricos sobre las Bases de Datos y se vieron mecanismos prácticos para trabajar sobre ellas interactuando **directamente** con un DBMS. En este curso veremos estrategias para trabajar con Bases de Datos desde **aplicaciones**, que las utilizan como mecanismo de **persistencia**.

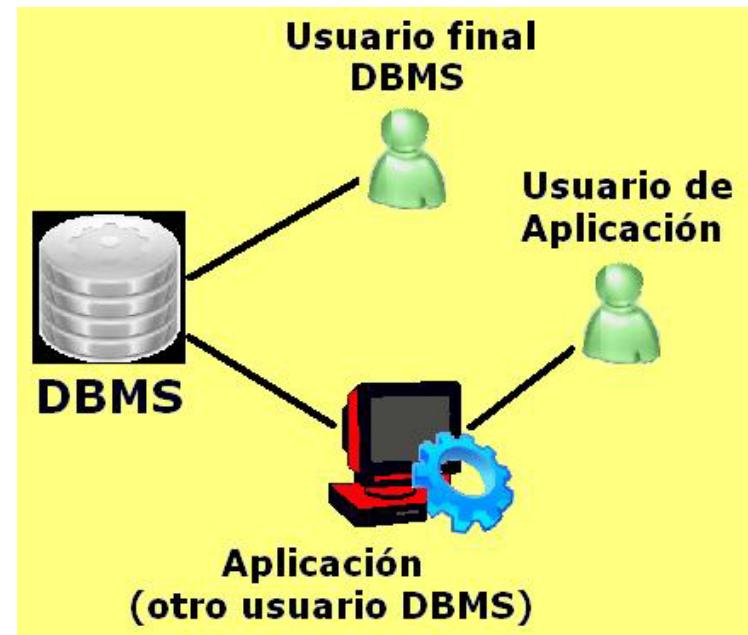
A efectos de poder comprender dichas estrategias, primeramente es necesario comprender los mecanismos que los lenguajes de programación utilizan para establecer conexiones con los DBMS, acceder a sus bases de datos, ejecutar consultas sobre ellas, y procesar sus resultados.

Conexión con DBMS desde Lenguajes de Programación:

Así como los usuarios finales pueden conectarse con un DBMS, los lenguajes de programación también pueden hacerlo. Para el DBMS la conexión realizada por un usuario final es **idéntica** a la conexión realizada desde un lenguaje de programación.

Cuando la aplicación se conecta con el DBMS, se la considera **un usuario más** del DBMS y puede realizar tareas sobre él como cualquier usuario final.

A su vez, los usuarios finales de la aplicación son irrelevantes para el DBMS. La aplicación en sí misma es la usuaria del DBMS.



Conexión con DBMS desde Lenguajes de Programación (cont.)

El establecimiento de una conexión entre la aplicación y un DBMS está dado por el uso de un **Driver**. Se trata de un componente que está implementado en el lenguaje de programación de la aplicación y que es capaz de conectarse con el DBMS en cuestión.



Algunos lenguajes ya traen por defecto **drivers** implementados para ciertos DBMS. En otros, han de ser incluidos por fuera de sus librerías básicas.

Los fabricantes de los DBMS suelen proveer **drivers** para distintos lenguajes. También hay implementados por terceros.

Conexión con DBMS desde Lenguajes de Programación (cont.)

ODBC (Open Database Connectivity) es una **API** disponible en diversos Sistemas Operativos para facilitar la conexión entre DBMS y lenguajes de programación. El uso de ODBC es una alternativa a la interacción directa entre lenguaje y DBMS. Tener un **driver** que conecte la aplicación con ODBC basta para establecer la conexión, independientemente de cuál sea el DBMS utilizado.

El lenguaje de programación y el DBMS elegidos para utilizar en este curso son **Java** y **MySQL** respectivamente (ver instructivo para obtención del **driver** correspondiente en el sitio web del curso). No obstante, en algunas actividades se podrá usar **C++** o **SQL Server**.

Dado que **Java** es un Lenguaje **Orientado a Objetos**, muchos de los conceptos a estudiar en este curso son también aplicables a cualquier otro Lenguaje OO (**C++**, **C#**, *Delphi*, *Python*, *PHP 5*, etc.).

Acceso a BD desde Java – JDBC:

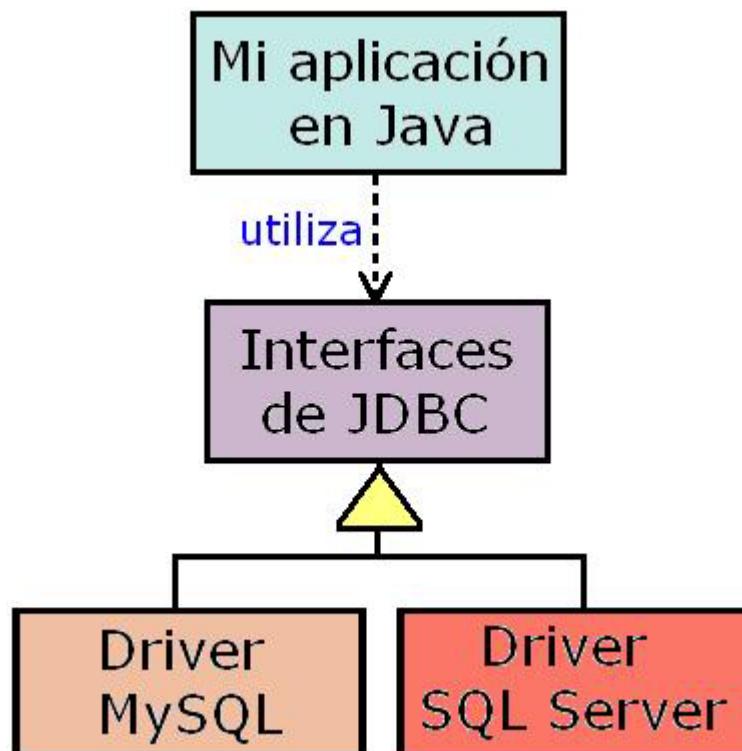
JDBC (Java Database Connectivity) es una **API de Java** encargada de definir el acceso a bases de datos desde este lenguaje. Consiste en un **componente** de Java que posee distintas clases e interfaces encargadas de dicho propósito. Dicho componente está constituido por los **packages** `java.sql` y `javax.sql`.

La idea de **JDBC** es introducir una capa de abstracción entre **Java** y el DBMS a utilizar. Las distintas interfaces de JDBC definen los métodos que cualquier **driver** debe implementar. De este modo, se uniformiza la forma de comunicación entre **Java** y cualquier DBMS.

Cualquiera que programe un **driver** para el DBMS (fabricante o terceros) lo hará implementando los métodos de las interfaces de **JDBC**, logrando así que el código fuente de nuestra aplicación se independice de la implementación concreta del driver.

Acceso a BD desde Java – JDBC (continuación):

Nuestra aplicación desarrollada en **Java** invocará a los métodos de las interfaces de **JDBC** que sean necesarios para establecer la conexión con el DBMS y trabajar sobre BD albergadas en él.



En forma interna, se ejecutará la implementación correspondiente al **driver** concreto con el que estemos trabajando (puede ser de un DBMS concreto como de ODBC) sin que quede a la vista en nuestro código.

Previamente debemos asegurarnos de incorporar a nuestras librerías la implementación del driver (si es que no viene incorporado por defecto).

Acceso a BD desde Java – JDBC (continuación):

Presentamos a continuación algunas de las interfaces de **JDBC** más relevantes (definidas en el paquete `java.sql`):

- **Driver**: Representa el corazón del *driver*. Es quien se encarga de establecer físicamente la conexión con el DBMS (u ODBC).
- **Connection**: Una vez establecida la conexión con el DBMS, es quien establece una conexión con una BD albergada por él.
- **Statement**: Una vez establecida la conexión con una BD, es quien permite ejecutar consultas sin parámetros sobre la BD.
- **PreparedStatement**: Una vez establecida la conexión con una BD, permite ejecutar consultas con parámetros sobre la BD.
- **ResultSet**: Una vez ejecutada una consulta sobre la BD, es quien devuelve la lista de registros obtenidos por la consulta.

Carga de Driver – JDBC:

Para establecer la conexión con el DBMS, lo primero a realizar es incorporar la implementación del **driver** a nuestras librerías (si es que no viene incorporado por defecto). Luego, desde el código de la aplicación, se lo carga dinámicamente en tiempo de ejecución:

```
String driver = "com.mysql.jdbc.Driver";  
Class.forName(driver);
```

El método **Class.forName** realiza la carga dinámica de una clase en tiempo de ejecución. Recibe como parámetro el nombre de la clase a cargar (en este caso la clase que implementa la interface **Driver**). Tras cargada, queda en memoria para su posterior uso.

Observación: Este paso no es necesario cuando se trata de un driver que ya está incorporado por defecto, o cuando ya ha sido cargado mediante algún otro mecanismo.

Conexión con Base de Datos – JDBC:

Una vez cargado el driver, el siguiente paso es solicitarle la creación de una conexión con la BD a utilizar:

```
String url = "jdbc:mysql://localhost:3306/MiBD";  
Connection con = DriverManager.getConnection  
    (url, user, password);
```

La **url** es la ruta a la BD relativa al origen de datos en uso (DBMS u ODBC). Usualmente es de la forma **jdbc:origenDatos:rutaBD**.

El método **getConnection** necesita también el **usuario** y la **password** del origen de datos. En caso de que no sea posible establecer la conexión, lanza una **SQLException**.

Observación: Si **no** se realizó explícitamente la carga del driver, **getConnection** igualmente intentará localizar un driver adecuado entre aquellos que hayan sido cargados por defecto.

Realización de consultas – JDBC:

Una vez establecida la conexión, es posible ejecutar consultas sobre la BD. Existen dos tipos de consultas posibles:

- **Sin parámetros**: Se ejecutan típicamente usando **Statement**.
- **Con parámetros**: Necesitan uno o mas parámetros. Se ejecutan típicamente usando **PreparedStatement**.

En ambos casos se habla de consultas en el sentido más amplio. En realidad se trata de cualquier sentencia **SQL** válida (**Create**, **Drop**, **Select**, **Insert**, **Update**, **Delete**, etc.).

Las consultas hechas con **Statement** son ejecutadas directamente en el DBMS, mientras que las hechas con **PreparedStatement** son precompiladas antes de su ejecución. **Nota:** Las consultas **sin** parámetros también pueden ejecutarse con PreparedStatement.

Consultas sin parámetros – JDBC:

Veamos un ejemplo de una consulta **sin** parámetros:

```
01. String query;  
02. query = "SELECT cedula,nombre FROM Personas";  
03. Statement stmt = con.createStatement();  
04. ResultSet rs = stmt.executeQuery(query);  
05. while (rs.next()) {  
06.     int cedula = rs.getInt("cedula");  
07.     String nombre = rs.getString("nombre");  
08.     System.out.println(cedula + "-" + nombre);  
09. }  
10. rs.close();  
11. stmt.close();  
12. con.close();
```

Consultas sin parámetros – JDBC (continuación):

- **Líneas 1 y 2:** Se crea un String con el texto de la consulta.
- **Línea 3:** Sigue la conexión (creada previamente) la creación de un **Statement** para ejecutar la consulta anterior.
- **Línea 4:** Sigue el Statement la ejecución de la consulta. Le pasa el texto de la misma por parámetro al momento de la ejecución. Devuelve un **ResultSet** con el resultado de la misma. Este objeto contiene los (cero o mas) registros devueltos.
- **Líneas 5 a 9:** Recorre el ResultSet registro a registro. Funciona como un **iterador** de registros. En cada registro, obtiene el valor de cada columna pasando su nombre (o número). Al hacerlo usa un método compatible con el **tipo de datos** de la columna.
- **Líneas 10 a 12:** Cierra el ResultSet, luego el Statement y por último la Connection (en orden **inverso** a como fueron abiertos)¹⁴

Consultas con parámetros – JDBC:

Veamos un ejemplo de una consulta con parámetros:

```
01. String insert = "INSERT INTO Personas " +
02.                 "(cedula, nombre, edad) " +
03.                 "VALUES (?,?,?);"

04. PreparedStatement pstmt =
        con.prepareStatement(insert);

05. pstmt.setInt (1, 1234567);
06. pstmt.setString (2, "Venancio Carusso");
07. pstmt.setInt (3, 65);

08. int r = pstmt.executeUpdate();
09. System.out.print("Cant. Lineas Afectadas" + r);

10. pstmt.close();
11. con.close();
```

Consultas con parámetros – JDBC (continuación):

- **Líneas 1 a 3:** Se crea un String con el texto de la consulta. La ubicación que tendrá cada parámetro se indica con **?**.
- **Línea 4:** Sigue la conexión (creada previamente) la creación de un **PreparedStatement** para ejecutar la consulta anterior. Se le pasa el texto de la consulta en ese mismo instante.
- **Líneas 5 a 7:** Se pasan valores a los parámetros. A cada uno se lo identifica mediante su **posición** (1, 2, 3) y se le asigna el valor mediante un método compatible con su **tipo de datos**.
- **Líneas 8 y 9:** Sigue al PreparedStatement la ejecución de la consulta. Por ser una consulta de **modificación**, devuelve únicamente la cantidad de filas afectadas por la modificación.
- **Líneas 10 y 11:** Cierra el PreparedStatement y por último la Connection (en orden **inverso** a como fueron abiertos).

Realización de consultas – JDBC (continuación):

Observaciones:

- Los métodos **executeQuery** y **executeUpdate** están presentes tanto en **Statement** como en **PreparedStatement**. En el primero se les pasa el texto de la consulta por parámetro. En el segundo no se hace, pues ya fue hecho al crear el PreparedStatement.
- **executeQuery** se utiliza sólo en consultas de selección (**Select**), mientras que **executeUpdate** se utiliza en cualquier consulta que altere el estado de la BD (**Create**, **Drop**, **Insert**, etc.).
- **ResultSet** posee métodos de la forma **getXXX** para devolver los valores (de tipos compatibles con el tipo **XXX**) de las columnas devueltas por una consulta. De igual modo, **PreparedStatement** posee métodos de la forma **setXXX** para setear parámetros.

Compatibilidad de Tipos entre SQL y Java:

| Tipo SQL | Tipo Java | Métodos |
|-------------------|---------------|------------------------------------|
| CHAR, VARCHAR | String | getString (...), setString (...) |
| BIT | boolean | getBoolean (...), setBoolean (...) |
| SMALLINT | short | getShort (...), setShort (...) |
| INTEGER | int | getInt (...), setInt (...) |
| BIGINT | long | getLong (...), setLong (...) |
| REAL | float | getFloat (...), setFloat (...) |
| FLOAT, DOUBLE | double | getDouble (...), setDouble (...) |
| BINARY, VARBINARY | byte [] | getBytes (...), setBytes (...) |
| DATE | java.sql.Date | getDate (...), setDate (...) |
| TIME | java.sql.Time | getTime (...), setTime (...) |

Realización de consultas – JDBC (continuación):

- **Todo** método de **Statement**, **PreparedStatement** y **ResultSet** lanzará una **SQLException** en caso de ocurrir cualquier error vinculado con la BD durante su ejecución.
- Los objetos utilizados al realizar cualquier consulta sobre la BD (**Connection**, **Statement**, **PreparedStatement**, **ResultSet**) han de ser siempre **cerrados** luego de utilizados, y en orden **inverso** a como fueron abiertos. Esto es tanto para **optimizar** recursos como para **asegurar** que cualquier cambio en la BD sea efectivamente realizado tras la ejecución.
- Por defecto, cada **Connection** funciona de modo tal que cada consulta es ejecutada en **su propia transacción**. Para poder realizar **varias** consultas en una misma transacción, será necesario realizar **Manejo de Transacciones**.

Repaso de Transacciones:

Recordemos que una **transacción** es una secuencia de accesos a una BD que se ejecuta como una **unidad**. Es posible de ejecutarse en forma **concurrente** con otros accesos a la BD. Debería cumplir las propiedades **ACID** (**A**tomicity, **C**onsistency, **I**solation, **D**urability).

- **Atomicity**: La transacción debe ejecutarse en forma atómica (se ejecutan todos sus pasos o no se ejecuta ninguno).
- **Consistency**: La transacción debe partir de la BD en un estado consistente y dejarla en otro estado consistente al finalizar.
- **Isolation**: La transacción debe ejecutarse en forma aislada. Los cambios sobre la BD realizados por la transacción no deben ser vistos por otras transacciones hasta que sean confirmados.
- **Durability**: Los cambios realizados por la transacción deben ser permanentes en la BD una vez confirmados.

Repaso de Transacciones (continuación):

Toda **transacción** puede ser finalizada de dos maneras:

- **commit** (todos los cambios realizados son confirmados)
- **rollback** (ningún cambio realizado es confirmado).

Dependiendo del nivel de control de concurrencia que se aplique al ejecutar la transacción, pueden ocurrir los siguientes **problemas**:

- **Lectura sucia**: La transacción lee un dato no commiteado por otra transacción que luego aborta (dato inexistente).
- **Lectura no repetible**: La transacción realiza dos lecturas de un mismo dato que otra transacción modificó entre ambas lecturas.
- **Lectura fantasma**: La transacción realiza dos lecturas de un mismo conjunto de registros, cuya cantidad fue modificada por otra transacción entre ambas lecturas.

Repaso de Transacciones (continuación):

Hay 5 niveles de concurrencia que pueden aplicarse sobre una BD:

- **TRANSACTION_NONE**: No hay control alguno de concurrencia.
- **TRANSACTION_READ_UNCOMMITTED**: Permite tanto lectura sucia como lectura no repetible y lectura fantasma.
- **TRANSACTION_READ_COMMITTED**: Permite tanto lectura no repetible como lectura fantasma.
- **TRANSACTION_REPEATABLE_READ**: Sólo permite lectura fantasma.
- **TRANSACTION_SERIALIZABLE**: Nivel más bajo de concurrencia, elimina todos los problemas de transaccionalidad.

No todos los DBMS soportan los cinco niveles, incluso no todos los **drivers** para un mismo DBMS los manejan. Tanto **MySQL** como el driver elegido en este curso soportan los cinco niveles.

Manejo de Transacciones – JDBC:

Como se vio anteriormente, cada **Connection** funciona de modo tal que cada consulta se ejecuta en su propia transacción. Los pasos para ejecutar más de una consulta en una misma transacción son los siguientes:

1. Des-habilitar el modo **auto-commit** en la **Connection**.
2. Ejecutar cada una de las consultas de la transacción, cada una de ellas utilizando **Statement** o **PreparedStatement** (según corresponda).
3. Terminar la transacción ejecutando **commit** si todos los pasos fueron exitosos, o **rollback** en caso contrario.

Veremos un ejemplo de una transacción en la cual se desea dar de alta conjuntamente a un **perro** junto con su **dueño** en la base de datos correspondiente a un certamen de mascotas.

Manejo de Transacciones – JDBC (continuación):

```
01. String dueño = "INSERT INTO Personas " +
02.           "(cedula, nombre, edad) VALUES (?,?,?)";
03. String perro = "INSERT INTO Mascotas " +
04.           "(nombre, raza, cedDueño) VALUES (?,?,?)";
05. try {
06.     con = DriverManager.getConnection(url,usr,pwd);
07.     con.setTransactionIsolation
08.             (Connection.TRANSACTION_SERIALIZABLE);
09.     pstmt1 = con.prepareStatement(dueño);
10.     pstmt2 = con.prepareStatement(perro);
11.     pstmt1.setInt(1,1234567);
12.     pstmt1.setString(2, "Venancio Carusso");
13.     pstmt1.setInt(3, 65);
```

Manejo de Transacciones – JDBC (continuación):

```
14. pstmt1.executeUpdate();
15. pstmt1.close();
16. pstmt2.setString(1, "Tobi");
17. pstmt2.setString(2, "Ovejero");
18. pstmt2.setInt(3, 1234567);
19. pstmt2.executeUpdate();
20. pstmt2.close();
21. con.commit();
22. }
23. catch (SQLException e){
24.     con.rollback();
25. }
26. finally {
27.     con.close();
28. }
```

Manejo de Transacciones – JDBC (continuación):

- **Líneas 1 a 4:** Se crean los Strings con los textos de las dos consultas correspondientes a las inserciones a realizar.
- **Líneas 6 a 8:** Se crea la **Connection**, se le indica el nivel de transaccionalidad y se le desactiva el modo **auto-commit**.
- **Líneas 9 a 20:** Se crean los **PreparedStatement** para ambas consultas, se les pasan los parámetros respectivos, se ejecutan, y se cierran ambos PreparedStatement.
- **Línea 21:** Si ambas consultas se ejecutaron exitosamente, se finaliza la transacción ejecutando **commit** sobre la conexión.
- **Línea 24:** Si alguna de las dos consultas falló, se finaliza la transacción ejecutando **rollback** sobre la conexión.
- **Línea 27:** En cualquier caso, se cierra la conexión.

Invocación de Stored Procedures – JDBC:

Ya hemos visto cómo ejecutar consultas y cómo usar transacciones mediante **JDBC**. Veremos ahora cómo invocar **Stored Procedures** (procedimientos almacenados) desde Java mediante **JDBC**.

Recordemos que un **SP** (*stored procedure*) es un subprograma que se encuentra almacenado en la BD y que ejecuta sentencias sobre dicha BD en el DBMS. Combina sentencias SQL con instrucciones propias del fabricante del DBMS. En consecuencia, los SP no son **ANSI**, sino que están escritos en un lenguaje propietario del DBMS.

No todos los DBMS trabajan con SP, así como no todos los **drivers** soportan su ejecución. La invocación de SP desde JDBC es posible siempre y cuando tanto el **DBMS** como el **driver** con los que se esté trabajando la soporten.

Invocación de Stored Procedures – JDBC (continuación):

Para invocar SP desde Java se usa la interface **CallableStatement** del paquete **java.sql**. La misma permite generar la invocación al SP, así como pasarle parámetros (tanto de entrada como de salida) y procesar los resultados de la ejecución del SP.

Dado que los SP son escritos en un lenguaje propietario del DBMS, su invocación también se realiza usando dicho lenguaje propietario. Por ejemplo, la invocación de un SP llamado **MiProc** se realizaría de las siguientes formas en **SQL Server** y en **MySQL**:

- **EXEC MiProc 'Juana', @resu OUTPUT** (SQL Server)
- **CALL MiProc ('Juana', @resu)** (MySQL)

Con el fin de evitar que el lenguaje propietario quede a la vista en el código, **JDBC** define una sintaxis **genérica** para invocación de SP que luego es traducida por el **driver** al lenguaje propietario.

Sintaxis para invocación de Stored Procedures – JDBC:

Sintaxis **genérica** para invocación de SP desde Java:

- SP sin parámetros: {`call proc_name`}
- SP con parámetros: {`call proc_name(?,...,?)`}

La invocación del SP desde Java será escrita usando esta sintaxis genérica. Antes de ejecutar el SP en la BD, el **driver** se encargará de precompilar y traducir esta invocación al lenguaje propietario del DBMS. Los parámetros (tanto de entrada como de salida) se indican usando **?** como en **PreparedStatement**.

Algunos DBMS soportan SP que permiten devolver un resultado. En tal caso, la sintaxis genérica para su invocación es la siguiente:

- SP sin parámetros: {`? = call proc_name`}
- SP con parámetros: {`? = call proc_name(?,...,?)`}

Pasaje de Parámetros en Stored Procedures – JDBC:

Los SP pueden tener tanto parámetros de entrada como de salida. Los primeros se setean usando métodos de la forma **setXXX** (como en **PreparedStatement**). Los segundos tienen que ser registrados antes de ejecutar el SP. Luego de la ejecución, sus valores pueden ser obtenidos usando métodos **getXXX** (como en **ResultSet**).

Ejemplo: Parámetro **1** de **MiProc** es de entrada y el **2** de salida.

```
String sp = "{call MiProc(?,?)}";
CallableStatement cstmt = con.prepareCall(sp);
cstmt.registerOutParameter(2, java.sql.Types.BIT);
cstmt.setString(1,"Juana");
/* aquí se realiza la ejecución del SP */
boolean resu = cstmt.getBoolean(2);
cstmt.close();
```

Ejecución de Stored Procedures – JDBC:

Veremos ahora cómo hacer para **ejecutar** un SP desde Java. Hay **tres** maneras posibles de ejecutar el SP, las cuales dependen del **tipo** y la **cantidad** de sentencias SQL que el SP execute.

| Tipo de sentencias del SP | Ejecución desde Java |
|--|--|
| Únicamente <u>una</u> sentencia Select (devuelve solamente un ResultSet) | Se realiza mediante el método executeQuery de CallableStatement |
| Únicamente <u>una</u> sentencia de alteración de la BD (Create , Insert , Update , etc.) | Se realiza mediante el método executeUpdate de CallableStatement |
| <u>Una o más</u> sentencias Select y/o <u>una o más</u> sentencias de alteración de la BD | Se realiza mediante el método execute de CallableStatement |

Ejecución de Stored Procedures – JDBC (continuación):

Cuando el tipo de SP que va a ejecutar se conoce de antemano, se puede usar el cuadro anterior para determinar cómo ejecutarlo. Si esto **no** se conoce de antemano, obsérvese que la ejecución con el método **execute** de **CallableStatement** es **siempre** posible.

La ejecución mediante el método **execute** es más **trabajosa** ya que requiere **consultar** (luego de la ejecución) qué tipo de resultados y qué cantidad fueron obtenidos de la ejecución. En las ejecuciones con **executeQuery** y **executeUpdate** esto no es necesario.

Presentamos a continuación un ejemplo de SP de cada tipo:

1. SP que dada una raza, lista todas las mascotas de dicha raza
2. SP que dada una edad, suma 1 a todos los dueños con esa edad
3. SP que dadas dos cédulas, transfiere las mascotas del primer al segundo dueño, chequeando **previamente** que ambos existan

32

Ejecución de Stored Procedures – JDBC (continuación):

1. SP que dada una raza, lista todas las mascotas de dicha raza:

```
01. String sp = "{call ListarPorRaza(?)}";  
02. CallableStatement cstmt = con.prepareCall(sp);  
03. cstmt.setString(1, "Caniche");  
04. ResultSet rs = cstmt.executeQuery();  
05. while (rs.next())  
06. { /* se procesa el listado */ }  
07. rs.close();  
08. cstmt.close();
```

Observación: Este SP recibe solamente un parámetro de entrada, no posee parámetros de salida e internamente ejecuta una única sentencia **Select** (devuelve solamente un **ResultSet**).

Ejecución de Stored Procedures – JDBC (continuación):

1. SP que dada una raza, lista todas las mascotas de dicha raza:
 - **Líneas 1 y 2:** Sigue la ejecución de la clase anterior. Se solicita a la conexión (creada antes) la creación de un **CallableStatement** para ejecutar el SP anterior. Se le pasa la sintaxis genérica de la invocación en ese instante.
 - **Línea 3:** Se le pasa el valor correspondiente al parámetro de entrada. Se lo identifica mediante su **posición** (1) y se le asigna el valor mediante un método compatible con su **tipo de datos**.
 - **Línea 4:** Sigue la ejecución de la clase anterior. Se solicita al CallableStatement la ejecución del SP. Se asume que solamente se retorna un **ResultSet**. De no ser así, puede ocurrir una **SQLException**.
 - **Líneas 5 a 8:** Procesa el ResultSet y luego lo cierra junto con el CallableStatement (en orden **inverso** a como fueron abiertos).

Ejecución de Stored Procedures – JDBC (continuación):

2. SP que dada una edad, suma 1 a todos los dueños con esa edad:

```
01. String sp = "{call CumpleAños(?)}";  
02. CallableStatement cstmt = con.prepareCall(sp);  
03. cstmt.registerOutParameter  
        (1, java.sql.Types.INTEGER);  
04. cstmt.setInt(1, 65);  
05. int cantFilasAfectadas = cstmt.executeUpdate();  
06. int edadResultante = cstmt.getInt(1);  
07. cstmt.close();
```

Observación: Este SP recibe un solo parámetro, que es a la vez un parámetro de entrada y también de salida (edad antes y después de sumar 1). Internamente ejecuta una única sentencia **Update**.

Ejecución de Stored Procedures – JDBC (continuación):

2. SP que dada una edad, suma 1 a todos los dueños con esa edad:

- **Líneas 1 y 2:** Sigue la ejecución de la conexión (creada antes) la creación de un **CallableStatement** para ejecutar el SP anterior. Se le pasa la sintaxis genérica de la invocación en ese instante.
- **Líneas 3 y 4:** Se registra el parámetro como parámetro de salida de tipo **INTEGER** (tipo de **SQL**) y luego se le asigna el valor como parámetro de entrada.
- **Línea 5:** Sigue la ejecución del CallableStatement la ejecución del SP. Se asume que solamente ejecuta una actualización sobre la BD, obteniendo la cantidad de filas afectadas. De no ser así, puede ocurrir una **SQLException**.
- **Líneas 6 y 7:** Obtiene la edad resultante de la ejecución y luego cierra el CallableStatement.

Ejecución de Stored Procedures – JDBC (continuación):

3. SP que dadas dos cédulas, transfiere las mascotas del primer al segundo dueño, chequeando **previamente** que ambos existan:

```
01. String sp = "{call TransferirMascotas(?,?)}";  
02. CallableStatement cstmt = con.prepareCall(sp);  
03. cstmt.setInt(1, 1234567);  
04. cstmt.setInt(2, 3444555);  
05. boolean isResultSet = cstmt.execute();
```

Observación: Este SP recibe **dos** parámetros de entrada y **ninguno** de salida. Internamente ejecuta una sentencia **Select** que verifica la existencia de ambos dueños. Si efectivamente ambos existen, luego ejecuta un comando **Update**. Nótese que su ejecución depende del resultado del **Select**, por lo que podría **no** llegar a ejecutarse.

Ejecución de Stored Procedures – JDBC (continuación):

```
06. if (isResultSet) { /* se espera un ResultSet */  
07.     ResultSet rs = cstmt.getResultSet();  
08.     while (rs.next())  
09.     { /* rs vino con 0, 1 o 2 dueños */ }  
10.    rs.close();  
11.    isResultSet = cstmt.getMoreResults();  
12.    if (!isResultSet) {  
13.        /* se espera una actualización */  
14.        int cantFilasAfec = cstmt.getUpdateCount();  
15.        if (cantFilasAfec == -1)  
16.            System.out.print("No se hizo update");  
17.    }  
18. }  
19. cstmt.close();
```

Ejecución de Stored Procedures – JDBC (continuación):

- **Líneas 1 a 4:** Sigue la ejecución de la conexión (creada antes) la creación de un **CallableStatement** para ejecutar el SP y le asigna el valor de los parámetros de entrada.
- **Línea 5:** Sigue la ejecución del CallableStatement la ejecución del SP, mediante el método **execute**, el cual devuelve un valor booleano que indica si el primer resultado es o no un **ResultSet**.
- **Líneas 6 a 10:** En caso de ser efectivamente de un ResultSet (no se esperaba otra cosa), procesa los dueños devueltos en él.
- **Líneas 11 a 18:** Pregunta al CallableStatement si hubo un 2do resultado. En caso afirmativo, le pregunta si se trata de una actualización. Si devolvió **-1**, significa que no se ejecutó. En otro caso, significa que la actualización efectivamente se ejecutó.
- **Línea 19:** En cualquier caso, se cierra el CallableStatement.

Ejecución de Stored Procedures – JDBC (continuación):

Observaciones:

- El método **execute** de **CallableStatement** devuelve un valor booleano que indica si lo devuelto por la ejecución es o no un **ResultSet**. En caso de no serlo, puede ser una actualización o puede no haber resultado.
- El método **getMoreResults** sirve para ir obteniendo los distintos resultados de la ejecución hasta que no queden más. Los métodos **getResultSet** y **getUpdateCount** sirven para irlos procesando a medida que se obtienen.
- Conforme más complejo sea el SP, más complejo se vuelve el procesamiento de sus resultados. Será necesario utilizar una cuidadosa combinación de los métodos anteriores a efectos de procesarlos de manera correcta.

Obtención de Metadatos – JDBC:

En el contexto de una **base de datos**, se denomina **metadatos** a toda información relativa a las **características** y **organización** de la base de datos.

Los **metadatos** **no** se refieren a datos almacenados en las tablas de la base sino que constituyen información sobre su **estructura**; qué tablas posee, cuáles son sus campos, de qué tipos son, etc.

En ocasiones, interesa consultar información sobre los metadatos de la base desde la aplicación a desarrollar. Por ejemplo desde una aplicación que le permita al usuario escribir sus propias consultas, donde la aplicación **no** conoce de antemano las tablas a utilizar.

JDBC provee dos interfaces en el paquete **java.sql** que permiten al desarrollador obtener metadatos de las bases albergadas por el DBMS, ellas son **DataBaseMetaData** y **ResultSetMetaData**.

Obtención de Metadatos – JDBC (continuación):

Ambas interfaces ofrecen métodos que permiten consultar una amplia variedad de metadatos, como por ejemplo:

- Esquemas (bases de datos) albergados por el DBMS.
- Sentencias SQL soportadas por el DBMS.
- Niveles de transaccionalidad soportados por el DBMS.
- Tablas que componen una base de datos determinada.
- Características de las distintas tablas de una base.
- Permisos disponibles sobre las distintas tablas de una base.
- Stored Procedures disponibles para una base.
- Características de las columnas devueltas por una consulta.
- Etc.

Referencia: Consultar la API de **JDBC** para ver una descripción detallada de los métodos provistos por **DataBaseMetaData** y por **ResultSetMetaData**.