Acceso a bases de datos relacionales mediante Java/JDBC

Wladimiro Díaz

12 de febrero de 2014

Introducción

¿Qué es *JDBC*?

- *JDBC* es una API de Java de conexión a bases de datos que proporciona acceso a virtualmente cualquier fuente de datos tabulares a través de una aplicación Java.
- Además de proporcionar acceso a un amplio rango de bases de datos SQL, también permite acceder a otras fuentes de datos tales como hojas de cálculo o ficheros simples.
- *JDBC* se centra en la ejecución de sentencias SQL y en la recuperación de los resultados.
- Se basa en un estándar internacional para el acceso a bases de datos SQL (*X/Open SQL Call Level Interface*).
  - Se trata del mismo estándar sobre el que se basa la interface ODBC de Microsoft.
- *JDBC* proporciona una interface simple y consistente para trabajar con bases de datos relacionales.

¿Qué es *JDBC*?

- La mayor fuerza de *JDBC* reside en que se diseñó para trabajar exactamente igual con cualquier base de datos relacional.
- Proporciona una interface uniforme que se apoya sobre una gran variedad de diferentes módulos de conexión a bases de datos.
  - No es necesario escribir programas diferentes para acceder a una base de datos Sybase, SQL Server, Oracle, Postgresql, Mysql, etc...
- La funcionalidad básica de *JDBC* es:
  - Establecer una conexión con la base de datos o cualquier otra fuente de datos tabulares.
  - Enviar comandos SQL a la base de datos.
  - Procesar los resultados.

¿Cómo funciona *JDBC*?

- `java.sql.DriverManager`. Se encarga de la carga y localización de los *drivers*.
- `java.sql.Driver`. Se encarga de comprobar que las conexiones son válidas.
- `java.sql.Connection`. Actúa como un contenedor para ejecutar sentencias SQL a través de una conexión dada.
- `java.sql.ResultSet`. Controla el acceso a los resultados de un `statement` a través de una estructura que puede ser recorrida utilizando un *cursor* del cual se extraen los datos.

Los elementos de *JDBC*

El *DriverManager*

- Proporciona los servicios básicos para gestionar los *drivers* de *JDBC*.
- Un programa pueden hacer referencia a un *driver* concreto usando la llamada *Class.forName()*.
- Un ejemplo:

```
1    /**
2       La siguiente sentencia carga la clase DriverManager
3    */
4    Class.forName("com.mysql.jdbc.Driver").newInstance();
```

- Un objeto `Connection` representa una conexión con la base de datos.
- Una sesión incluye las sentencias SQL que se ejecutan sobre esa conexión y los resultados que se devuelven.
- Una aplicación puede tener una o muchas conexiones con una única base de datos o con varias bases de datos diferentes.
- La forma estándar de establecer una conexión con la base de datos es mediante una llamada al método `getConnection(URL)`.

- El parámetro `URL` (*Universal Resource Locator*) es una forma flexible de identificar la base de datos.
- La sintáxis estándar de una `URL` de *JDBC* es:

```
jdbc:<subprotocolo>:<subnombre>
```

  - `jdbc`. En este caso, el protocolo es siempre `jdbc`.
  - `<subprotocolo>`. El nombre del *driver* o mecanismo de conexión.
  - `<subnombre>`: Un identificador único de la base de datos.

- Un ejemplo:

```
1    Class.forName("com.mysql.jdbc.Driver").newInstance();
2    String url = "jdbc:mysql://divm2.uv.es/" + database;
3    connection = DriverManager.getConnection(url,
4      username, password);
```

SQL *Statement*

- Una vez que se ha establecido la conexión, lo normal es pasar sentencias SQL a la base de datos.
- No existen restricciones acerca del tipo o naturaleza de los comandos que se pueden enviar al SGBD.
- *JDBC* proporciona tres clases para enviar sentencias SQL a la base de datos:
  - `Statement`. Se utiliza para enviar sentencias SQL simples. Un `statement` se crea mediante el método `createStatement()`.
  - `PreparedStatement`. Se trata de un `statement` precompilado que puede utilizarse para ejecutar una sentencia múltiples veces con mejor rendimiento.
  - `CallableStatement`. Se utiliza para invocar procedimientos almacenados.

- Los objetos `statement` se utilizan para ejecutar sentencias SQL estáticas y obtener los resultados producidos.
- `statement` define tres métodos para ejecutar sentencias SQL dependiendo del tipo de resultado que se obtienen:
  - `executeUpdate(String sql)`. Ejecuta sentencias `INSERT`, `UPDATE` o `DELETE` que devuelven el número de filas afectadas o cero. También se utiliza para comandos DDL tales como `CREATE TABLE` que no devuelven nada.
  - `executeQuery(String sql)`. Ejecuta una sentencia SQL que devuelve un único `ResultSet`.
  - `execute(String sql)`. Ejecutan una sentencia SQL que puede devolver múltiples resultados.

Ejemplo 1:

```
1   Class.forName("com.mysql.jdbc.Driver").newInstance();
2   String  url  = "jdbc:mysql://localhost/"  + username;
3   connection = DriverManager.getConnection(url,
4     username, password);
5
6   statement = connection.createStatement();
7   String  tableSQL = "CREATE TABLE empleados ( "
8     + "idEmpleado INTEGER   NOT NULL,"
9     + "nombre      VARCHAR(128) NOT NULL,"
10    + "apellidos    VARCHAR(128) NOT NULL,"
11    + "antiguedad SMALLINT   NULL)";
12  statement.executeUpdate(tableSQL);
```

Ejemplo 2:

```
1   Class.forName("com.mysql.jdbc.Driver").newInstance();
2   String url = "jdbc:mysql://localhost/" + username;
3   connection = DriverManager.getConnection(url,
4     username, password);
5
6   statement = connection.createStatement();
7
8   String insertSQL = "INSERT INTO Empleados "
9     + "(idEmpleado,nombre,apellidos) "
10    + "VALUES (23457213,'Jose Antonio','Martinez Lopez')";
11  statement.execute(insertSQL);
```

Ejemplo 3:

```
1   Class.forName("com.mysql.jdbc.Driver").newInstance();
2   String url = "jdbc:mysql://localhost/" + username;
3   connection = DriverManager.getConnection(url,
4     username, password);
5
6   statement = connection.createStatement();
7   String selectSQL = "SELECT * FROM Empleados";
8   resultSet = statement.executeQuery(selectSQL);
9
10  while (resultSet.next()) {
11    System.out.print("[" + resultSet.getInt("idEmpleado") +
12        "] ");
13    System.out.print(resultSet.getString("nombre") + " ");
14    System.out.print(resultSet.getString("apellidos"));
15    System.out.println(". Antiguedad: " +
16      resultSet.getByte("antiguedad"));
17  }
```

Actualizaciones por lotes

- Una actualización por lotes (*batch update*) es un conjunto de sentencias que se envían a la base de datos para ser procesadas como un lote (*batch*).
- Este proceso puede ser más eficiente que procesar los comandos SQL por separado.
- En un lote pueden coexistir sentencias de actualización, inserción y borrado de filas.
- También puede contener sentencias DLL, tales como `CREATE TABLE` o `DROP TABLE`.
- Importante:

> En un lote no se pueden procesar sentencias que devuelvan un `ResultSet` (como es una sentencia `SELECT`). Sólo se pueden utilizar sentencias que devuelven un número de filas afectadas o cero.

Actualizaciones por lotes

- Para manejar las actualizaciones por lotes, disponemos de los comandos:
  - `AddBatch`. Añade una sentencia SQL al lote.
  - `clearBatch`. Vacila la lista de sentencias.
  - `executeBatch`. Ejecuta todas las sentencias contenidas en la lista *batch*.
- Un ejemplo:

```
1   connection.setAutoCommit(false);
2   statement = connection.createStatement();
3   statement.addBatch("INSERT INTO Empleados "
4     + "(idEmpleado,nombre,apellidos,antiguedad) "
5     + "VALUES (85438592,'Mike','Jagger',40)");
6   statement.addBatch("INSERT INTO Empleados "
7     + "(idEmpleado,nombre,apellidos) "
8     + "VALUES (23827181,'David','Bisbal')");
9   int [] updateCounts = statement.executeBatch();
10  connection.commit();
11  connection.setAutoCommit(false);
```

Actualizaciones por lotes

- El SGBD ejecuta los comandos del lote en el orden en que son introducidos en la lista.
- Devuelve un vector de enteros, que representan el resultado de los comandos ejecutados con éxito.
- Si alguna de las sentencias del lote no se ejecuta correctamente, se genera una excepción `BatchUpdateException`.
- Importante:

> Siempre es necesario deshabilitar el modo auto-commit durante la actualización por lotes, de modo que si se produce algún error, se puede gestionar correctamente.

- Ejemplo:

```
1    catch (BatchUpdateException e) {
2      System.err.print("Contador de sentencias: ");
3      int [] updateCounts = e.getUpdateCounts();
4      for (int i = 0; i < updateCounts.length; i++)
5        System.err.println(updateCounts[i]);
6    }
```

- Un `ResultSet` es el resultado de una consulta SQL.
- Consiste en todas las filas que satisfacen la condición de la consulta.
- Los datos en un `ResultSet` se organizan como una tabla.
- Por ejemplo,
  - La consulta

```
1  SELECT idEmpleado,nombre,apellidos
2  FROM Empleados
3  WHERE apellidos LIKE '%orre%';
```

  - produce los siguiente resultados:

idEmpleado	*nombre*	*apellidos*
34832719	Emilio	Torres Guadiana
39281274	Patricia	Latorre Rúiz

- Un objeto `ResultSet` mantiene un *cursor* que apunta a la fila de datos accesible a través de los métodos `getter` de `ResultSet`.
- El método `next()` hace que el cursor avance a la siguiente línea.
- Inicialmente, el *cursor* se encuentra posicionado **delante** de la primera columna, por lo que es necesario llamar al método `next()` para posicionar el *cursor* en la primera fila.
- El método `next()` devuelve el valor booleano **true** si está disponible una nueva fila, por lo que es fácil y elegante recorrer un `ResultSet` mediante un bucle **while**.
- Los datos contenidos en el `ResultSet` son accedidos mediante métodos *getter* con el nombre de la columna que contiene los datos.
- Los nombres de columna empleados en el *getter* no distinguen entre mayúsculas y minúsculas.
- Dentro de una fila, las columnas pueden accederse en cualquier orden.

Un ejemplo:

```
1   try {
2       Class.forName("com.mysql.jdbc.Driver").newInstance();
3       String url = "jdbc:mysql://localhost/" + username;
4       connection = DriverManager.getConnection(url,
5         username, password);
6       statement = connection.createStatement();
7       String selectSQL = "SELECT * FROM Empleados";
8       resultSet = statement.executeQuery(selectSQL);
9
10      while (resultSet.next()) {
11        System.out.print("[" + resultSet.getInt("idEmpleado") + "] ");
12        System.out.print(resultSet.getString("nombre") + " ");
13        System.out.print(resultSet.getString("apellidos"));
14        System.out.println(". Antiguedad: "
15          + resultSet.getByte("antiguedad"));
16      }
17  }
```

- Los *getters* tienen dos variantes: una que referencia la columna por nombre y otra que utiliza el nombre de columna.
- En detalle:

Tipo de dato	*Método*
BigDecimal	getBigDecimanl(String columnName, int scale)
boolean	getBoolean(String columnName)
byte	getByte(String columnName)
byte[]	getBytes(String columnName)
double	getDouble(String columnName)
float	getFloat(String columnName)
int	getInt(String columnName)
java.sql.Date	getDate(String columnName)
java.sql.Time	getTime(String columnName)
java.sql.Timestamp	getTimestamp(String columnName)
long	getLong(String columnName)
short	getShort(String columnName)
String	getString(String columnName)

- Un `ResultSet` *scrollable* tiene la capacidad de mover el *cursor* en cualquier dirección, obtener la posición en que se encuentra y posicionarlo en una fila particular.
- El tipo de `ResultSet` que devuelve un objeto `Statement` se define cuando se crea el `Statement` con la llamada al método `Connection.createStatement`.
- Existen dos formas de este último método:
  - `createStatement()`
  - `CreateStatement(int rsType, int rsConcurrency)`
- La segunda variante permite crear un `ResultSet` *scrollable* y/o *updateable*.

- El argumento `rsType` define el tipo de `ResultSet` y puede tomar los valores:
  - `TYPE_FORWARD_ONLY`. No *scrollable*, como los que hemos visto.
  - `TYPE_SCROLL_INSENSITIVE`. *Scrollable*, pero no refleja los cambios producidos en el `ResultSet` mientras está abierto.
  - `TYPE_SCROLL_SENSITIVE`. Además de *Scrollable*, refleja los cambios producidos en la base de datos.
- El segundo argumento permite decidir el modo de acceso:
  - `CONCUR_READ_ONLY`. El `ResultSet` es de sólo lectura.
  - `CONCUR_UPDATABLE`. Los valores del `ResultSet` se pueden modificar y estos cambios se reflejan en la base de datos (*updateable*).

Un ejemplo:

```
1   statement = connection.createStatement(
2     ResultSet.TYPE_SCROLL_INSENSITIVE,
3     ResultSet.CONCUR_READ_ONLY);
4   String selectSQL = "SELECT * FROM Empleados";
5   resultSet = statement.executeQuery(selectSQL);
6   resultSetMetaData = resultSet.getMetaData();
7
8   for (int i = 1; i <= resultSetMetaData.getColumnCount(); i++) {
9     System.out.print(resultSetMetaData.getColumnLabel(i) + "\t");
10  }
11  System.out.print("\n");
12
13  while (resultSet.next()) {}
14
15  while (resultSet.previous()) {
16    System.out.print("["+resultSet.getInt("idEmpleado")+"]\t");
17    System.out.print(resultSet.getString("nombre")+"\t");
18    System.out.print(resultSet.getString("apellidos")+"\t");
19    System.out.println(resultSet.getByte("antiguedad"));
20  }
```

Control del cursor

- Avance y retroceso: `next()` y `previous()`.
- Posicionamiento del cursor en una fila concreta:
  - `first()`
  - `last()`
  - `beforeFirst()`
  - `afterLast()`
  - `absolute(int rowNumber)`
  - `relative(int rowNumber)`
- Obtener la posición del cursor:
  - `isFirst()`
  - `isLast()`
  - `isBeforeFirst()`
  - `isAfterLast()`
  - `getRow()`

- Los cambios realizados en los datos de un `ResultSet` *updateable* se propagan a la base de datos.
- Es posible insertar una nueva fila, borrar una fila existente o modificar el valor de una o varias columnas de una fila.
- Solicitar un `ResultSet` *updatable* no garantiza que lo obtengamos: la disponibilidad depende del *driver* utilizado
- Conviene comprobarlo con el método `getConcurrency()`:

```
1   statement = connection.createStatement(
2     ResultSet.TYPE_SCROLL_INSENSITIVE,
3     ResultSet.CONCUR_UPDATABLE);
4   String selectSQL = "...";
5   resultSet = statement.executeQuery(selectSQL);
6
7   if (resultSet.getConcurrency() == ResultSet.CONCUR_UPDATABLE)
8     System.out.println("UPDATABLE");
9   else
10    System.out.println("READ_ONLY");
```

Actualizando un `ResultSet`

- Para apreciar lo simple y elegante que es el uso de un `ResultSet` actualizable, veamos cómo haríamos para cambiar los apellidos del empleado com *id* 34832719 a “Torre Guadiana”:

```
1  UPDATE Empleados
2     SET apellidos = "Torre Guadiana"
3   WHERE idEmpleado = 34832719;
```

- Utilizando un `UpdatableResultSet`, lo único que hay que hacer es:
  - Posicionar el cursor en la fila deseada.
  - Cambiar el valor de la columna utilizando un método de actualización específico del tipo de valor.
  - Por ejemplo:

```
1  resultSet.updateString("apellidos", "Torre Guadiana");
```

Actualizando un `ResultSet`

- Ya que los cambios realizados en un `ResultSet` sólo afectan a la fila actual, es imprescindible mover previamente el cursor a la fila correcta.
- La mayoría de los métodos `update` utilizan dos parámetros: el nombre de la columna a actualizar y el nuevo valor de la columna.
- La columna se puede referenciar bien por su nombre, bien por el número.
- Existe un método especial, `updateNull`, utilizado para poner el valor de una columna en `NULL`.
- Después de actualizar una fila, es necesario invocar al método `updateRow()` para hacer permanentes los cambios, antes de mover de nuevo el cursor.
- Es posible cancelar los cambios en cualquier momento invocando al método `cancelRowUpdates()`.
- Esto no funciona si previamente se ha invocado el método `updateRow()`.

Actualizando un `ResultSet`

- En detalle:

Tipo de dato	*Método*
BigDecimal	updateBigDecimanl(String columnName, BigDecimal x)
boolean	updateBoolean(String columnName, boolean x)
byte	updateByte(String columnName, byte x)
byte[]	updateBytes(String columnName, byte[] x)
double	updateDouble(String columnName, double x)
float	updateFloat(String columnName, float x)
int	updateInt(String columnName, int x)
Date	updateDate(String columnName, Date x)
Time	updateTime(String columnName, Time x)
Timestamp	updateTimestamp(String columnName, Timestamp x)
long	updateLong(String columnName, long x)
short	updateShort(String columnName, short x)
String	updateString(String columnName, String x)
NULL	updateNull(String columnName)

Insertando una fila nueva

- Los objetos de tipo `ResultSet` cuentan con una fila especial conocida como la *insert row*.
- Se trata de una fila especializada del buffer que se utiliza para añadir nuevas filas.
- Para insertar una nueva fila, basta con seguir los siguientes pasos:
  - Mover el *cursor* a la *insert row* llamando al método `moveToInsertRow()`.
  - Asignar un valor para cada columna en la fila utilizando el método *updater* adecuado al tipo de dato.
  - Invocamos al método `insertRow()` para insertar la nueva fila en el `ResultSet` y además en la base de datos.
- Si no se asignan valores a todas las columnas de la fila, pueden suceder varias cosas:
  - Se utilizará el valor por omisión de la columna si existe.
  - Si la columna acepta un valor `NULL`, se utilizará ese valor.
  - En cualquier otro caso, se generará una `SQLException`.

Insertando una fila nueva

- Importante: si se mueve el *cursor* antes de invocar al método `insertRow()`, se perderán los valores añadidos en la nueva fila.
- Una vez finalizada la inserción, es necesario mover el *cursor* de nuevo al `ResultSet`.
- Para ello podemos:
  - utilizar los métodos que posicionan el *cursor* en una fila específica: `first`, `last`, `beforeFirst`, `afterLast` y `absolute`.
  - Los métodos `previous` y `relative`, ya que el `ResultSet` mantiene un registro a la fila actual cuando se posicione en la *insert row*.
  - También se puede utilizar el método especial `moveToCurrentRow()`, que se puede invocar sólo cuando nos encontramos en la *insert row*.
    - Este método devuelve el *cursor* a la fila que era previamente la fila actual.

Borrando una fila

- Borrar una fila es muy sencillo:
  - Primero movemos el cursor a la fila que queremos borrar.
  - Invocamos al método `deleteRow()`.
- Cuidado: el comportamiento puede diferir entre diferentes implementaciones del *driver* de JDBC.

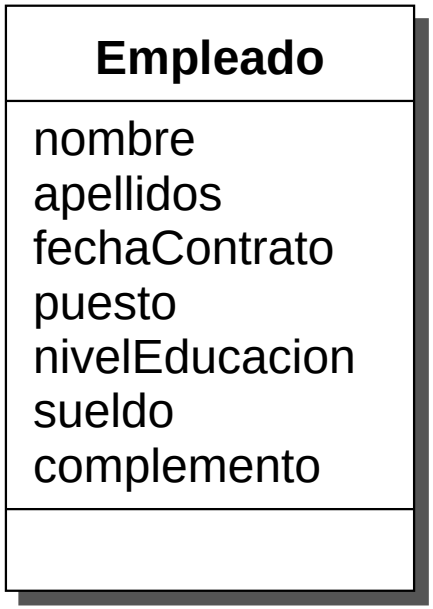
Por último:

> Cuando se realizan cambio en un `ResultSet`, los cambios no son necesariamente visibles...

Ejemplo

La clase *Empleado*

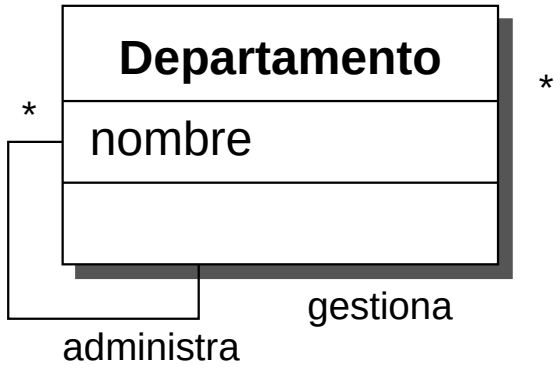
- El diagrama de clases:



- Los métodos:
  - Mostrar la tabla.

La clase *Departamento*

- El diagrama de clases:



- Los métodos:
  - Mostrar la tabla.

Bloque principal de código:

```
1   public void execute(){
2     Connection connection = null;
3     Statement statement = null;
4     try {
5       Class.forName("com.mysql.jdbc.Driver").newInstance();
6       String url = "jdbc:mysql://localhost/tasi";
7       connection =
8           DriverManager.getConnection(url,username,password);
9       statement = connection.createStatement();
10        ...
11    }
12      catch (Exception e) {
13        ...
14      }
15    }
16  }
```

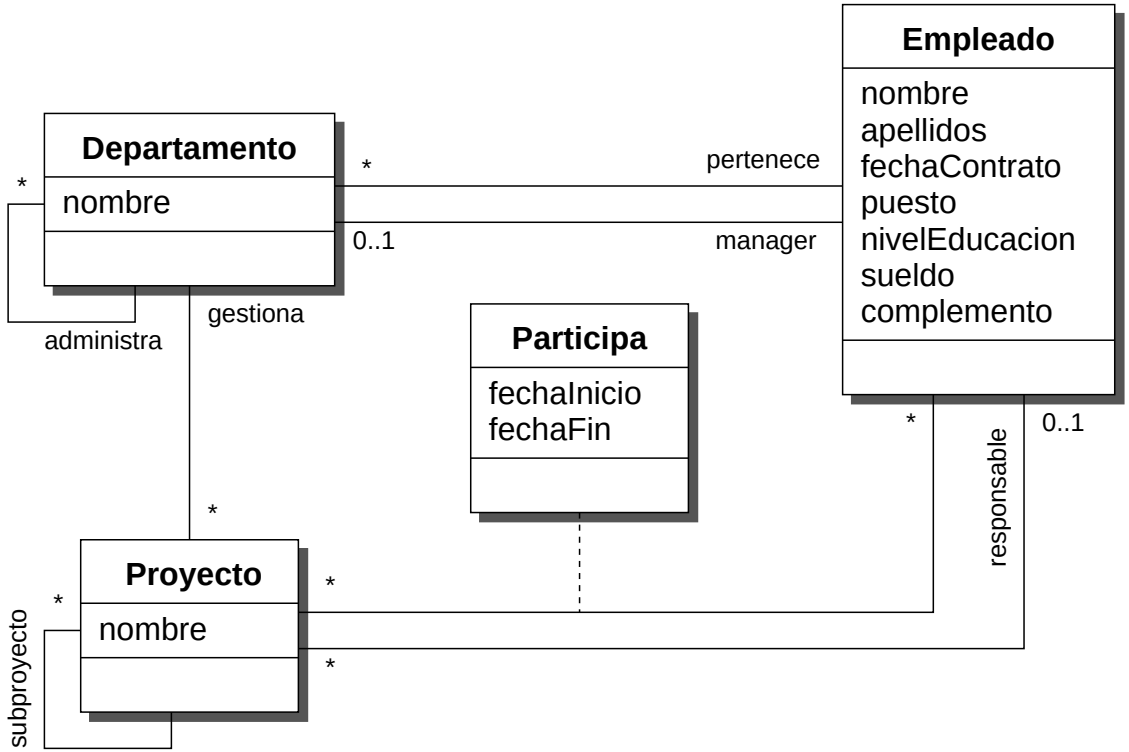
Ejercicio rápido...

- A partir del código de ejemplo, escriba la clase Java *DisplayDepartamentos* que muestre todos los departamentos de la empresa.

Ejercicio 2

El problema completo

- Diagrama de clases:



- Crear las clases java para:
  - Encontrar empleados buscando por apellidos.
  - Insertar proyectos y dedicaciones.
  - Mostrar la dedicación de cada *Empleado* en los proyectos que tiene asignados.