El ABC de JDBC

Copyright

Para cualquier duda, consulta, insulto o tirón de orejas sobre este tutorial dirigirse a abraham@javahispano.org.

Índice

1 Sobre este documento

Este documento pretende ser una introducción al API JDBC. No pretende en ningún momento explicar qué es una base de datos (BD) o como instalarla, o ser un tutorial a cerca del el lenguaje estándar de consultas SQL (Standard Query Language), aunque sí se presuponen en el lector conocimientos básicos de SQL.

Les recomiendo a aquellos lectores que quieran más información a cerca de qué es una base de datos el tutorial de Alberto Molpeceres *Bases de Datos* http://www.javahispano.org/tutorials.item.action?id=24. Es un excelente tutorial, si bien a veces es demasiado técnico. Para aquellos que no tengan paciencia, o no estén interesados en meterse tan a fondo en la materia, la lectura de los dos primeros capítulos del tutorial es suficiente para tener unos conocimientos básicos sobre bases de datos.

A aquellos que no dominen el lenguaje SQL les recomiendo el primer apartado del capítulo 4 del anterior tutorial, que les dará una base más que suficiente para comprender este tutorial.

Finalmente a aquellos que deseen instalar una base de datos para hacer pruebas con el API JDBC en la primera parte del artículo de Roberto Casas, *Cómo instalar Apache+Tomcat 3.x con soporte MySQL*, http://www.javahispano.org/articles.article.action?id=18 se explica como instalar la base de datos MySQL (http://www.mysql.com). Esta base de datos es Open Source, por lo que os la podéis descargar desde si sitio web, existiendo distribuciones tanto para Windows como para Linux.

2 ¿Qué es el API JDBC?

El API JDBC está definido por el JSR 56 (Java Specification Request). En este JSR Sun Microsystems junto con otros miembros del JCP (Java Comunnity Process http://jcp.org) han definido la interfaz que permite al desarrollador de aplicaciones Java interactuar con una gran variedad de bases de datos relacionales, y por otro la interfaz que deben respetar los distintos desarrolladores de manejadores JDBC. Como ya explicaremos más adelante un manejador es un driver que traduce los comandos estándar del API JDBC al formato propietario de una base de datos.

Este tutorial está dedicado a las características más básicas de la interfaz del desarrollador del API JDBC. Esto es, la parte del API que un desarrollador Java, dado un manejador para una determinada base de datos, necesita para interactuar con la base de datos. Esta parte del API permite al desarrollador realizar tres tareas:

1. Establecer una conexión con una base de datos
2. Enviar una consulta SQL a la base de datos
3. Procesar los resultados de la consulta.

El API JDBC del desarrollador de aplicaciones consta de dos partes, por un lado está el paquete *java.sql*, que contiene las clases e interfaces que permiten acceder a la funcionalidad básica del API JDBC. Este paquete forma parte de la edición estándar de la plataforma Java (J2SE), desde la versión 1.1 de ésta. Este tutorial se centra en la funcionalidad de este paquete.

Por otro lado están las clases e interfaces del paquete javax.sql, que forma parte de la edición empresarial de plataforma Java (J2EE), en el cual se incluye funcionalidad avanzada del API JDBC. Esta funcionalidad no se cubre en este tutorial.

3 Los manejadores

Los manejadores, también llamados drivers, son un conjunto de clases que implementan las clases e interfaces del API JDBC necesarias para que una aplicación Java pueda conectarse con una BD. Cuando los desarrolladores de una BD desean que esta pueda ser accesible mediante JDBC éstos deben implementar un manejador para esa base de datos; la misión del manejador será traducir comandos estándar del API JDBC al protocolo nativo de esa base de datos.

Cada base de datos emplea un protocolo diferente de comunicación, protocolos que normalmente son propietarios. El uso de un manejador, una capa intermedia entre el código del desarrollador y la base de datos, permite independizar el código Java que accede a la BD del sistema de BD concreto a la que estamos accediendo, ya que en nuestro código Java emplearemos comandos estándar, y estos comandos serán traducidos por el manejador a comandos propietarios de cada sistema de BD concreto. Si queremos cambiar el sistema de BD que empleamos lo único que deberemos hacer es reemplazar el antiguo manejador por el nuevo, y seremos capaces de conectarnos la nueva BD.

Para garantizar que un manejador respeta el API JDBC existe un conjunto de pruebas que debe pasar el manejador para poderse considerar "JDBC Compliant™", estas pruebas (disponibles en http://java.sun.com/products/jdbc/download.html) nos garantizan que un manejador ha sido desarrollado siguiendo el API JDBC.

En http://servlet.java.sun.com/products/jdbc/drivers podéis encontrar manejadores para prácticamente cualquier base de datos.

5.1 Tipos de manejadores

Hay 4 tipos de manejadores JDBC, que difieren en si usan o no tecnología Java Pura, en su rendimiento y en la flexibilidad para cambiar de base de datos. Veamos cuales son:

Puente JDBC-ODBC (tipo1)

ODBC es un API estándar semejante a JDBC, que permite que lenguajes como C++ accedan de un modo estándar a distintos sistemas de BD. Un manejador tipo puente JDBC-ODBC delega todo el trabajo sobre un manejador ODBC, que es quien realmente se comunica con la BD. El puente JDBC-ODBC permite la conexión desde Java a BD que no proveen manejadores JDBC. Fue muy útil cuando se creó el API JDBC, ya que muchas BD no disponían de manejadores JDBC, pero si de manejadores ODBC. Empelado el puente JDBC-ODBC podía accederse a estas bases de datos empleando el API JDBC .

Este tipo de manejador tiene dos desventajas: por un lado depende de código nativo, ya que el manejador ODBC no ha sido desarrollado en Java. Esto compromete la portabilidad de nuestro desarrollo. Por otro lado al emplear este tipo de manejador nuestra aplicación llama al gestor de manejadores JDBC, quien a su vez llama al manejador JDBC (puente JDBC-ODBC), quien llama al manejador ODBC, que es el que finalmente llama a la base de datos. Hay muchos puntos donde potencialmente puede producirse un fallo, lo cual hace que estos manejadores muchas veces sean bastante inestables.

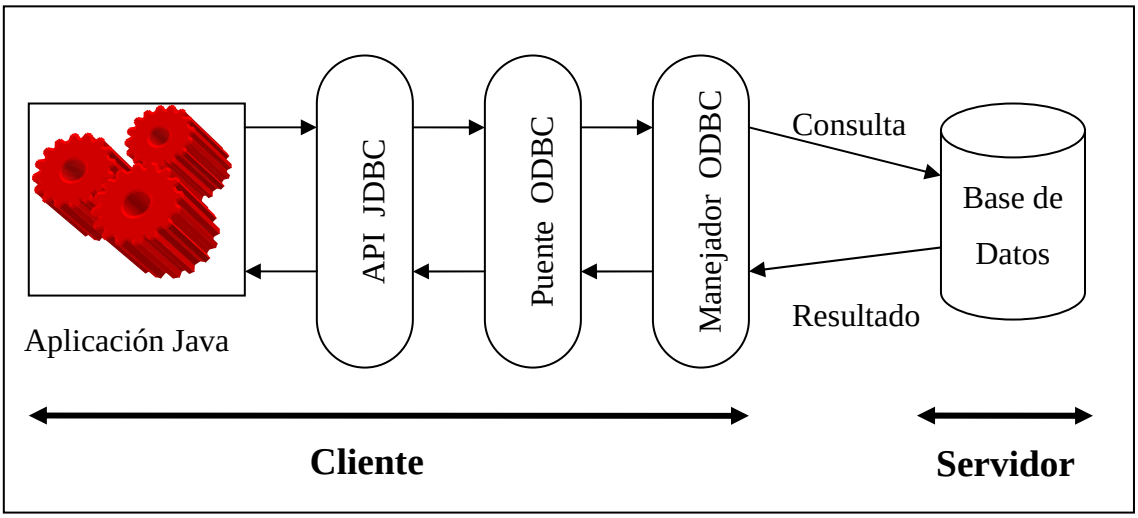


Figura 1 Puente JDBC-ODBC

El puente forma parte del jdk de SUN, esta en el paquete sun.jdbc.odbc. Inicialmente este tipo de manejador fue bastante útil por aliviar la carencia d emanejadores JDBC.

Hoy en día es el menos recomendado por limitar la portabilidad de la aplicación, al requerir código nativo, y por emplear tantas capas para la comunicación con la base de datos, lo que a veces hace que incrementa la inestabilidad de estos manejadores.

Manejador de API nativo (tipo2)

Se basa en una librería escrita en código nativo para acceder a la base de datos. El manejador traduce las llamadas JDBC a llamadas al código de la librería nativa, siendo el código nativo el que se comunica con las bases de datos. La librería nativa es proporcionada por los desarrolladores de la BD.

Estos manejadores son más eficientes y tienen menos puntos de fallo que el puente JDBC-ODBC ya que hay menos capas entre el código de la aplicación y la base de datos. Sin embargo siguen teniendo el problema de pérdida de portabilidad por emplear código nativo.

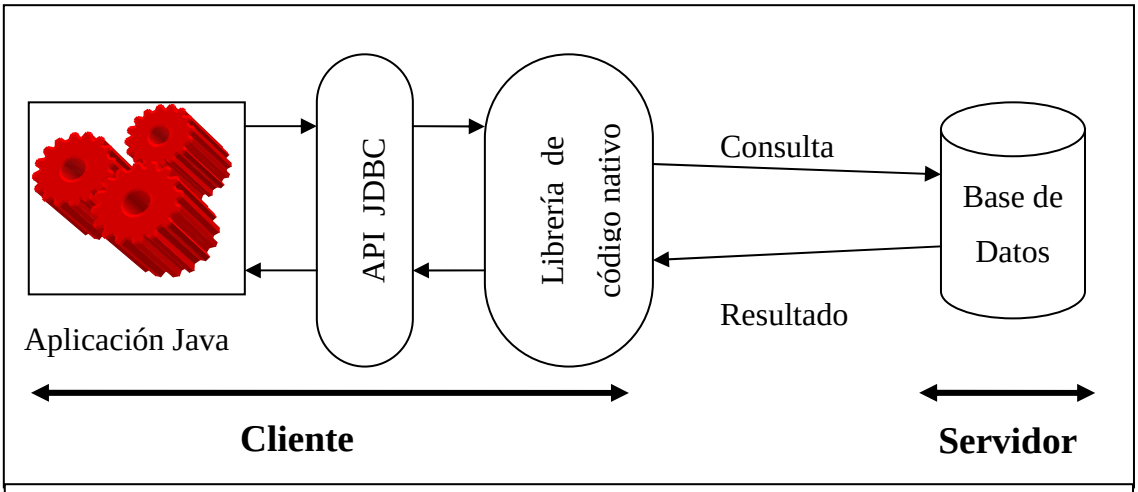


Figura 2 Manejador de API nativo

Manejador de JDBC-Net (tipo3)

El manejador se comunica con un servidor intermedio que se encuentra entre el cliente y la base de datos. El servidor intermediario se encarga de traducir las al API JDBC al protocolo específico de la base de datos.

No se requiere ningún tipo de código nativo en el cliente, por lo que la portabilidad de la aplicación está garantizada: el manejador es tecnología 100% Java.. Además si el intermediario es capaz de traducir las llamadas JDBC a protocolos específicos de diferentes sistemas de BD podremos emplear un mismo manejador para comunicarnos con diferentes bases de datos. Esto lo convierte en el manejador más flexible de todos. No obstante se trata de un driver complejo, ya que requiere la presencia de un midelware, una capa intermedia entre el cliente y la ase de datos, por lo que no es muy común emplearlo en arquitecturas simples, sino más bien en arquitecturas sofisticadas donde muchas veces entran en juego varias bases de datos distintas.

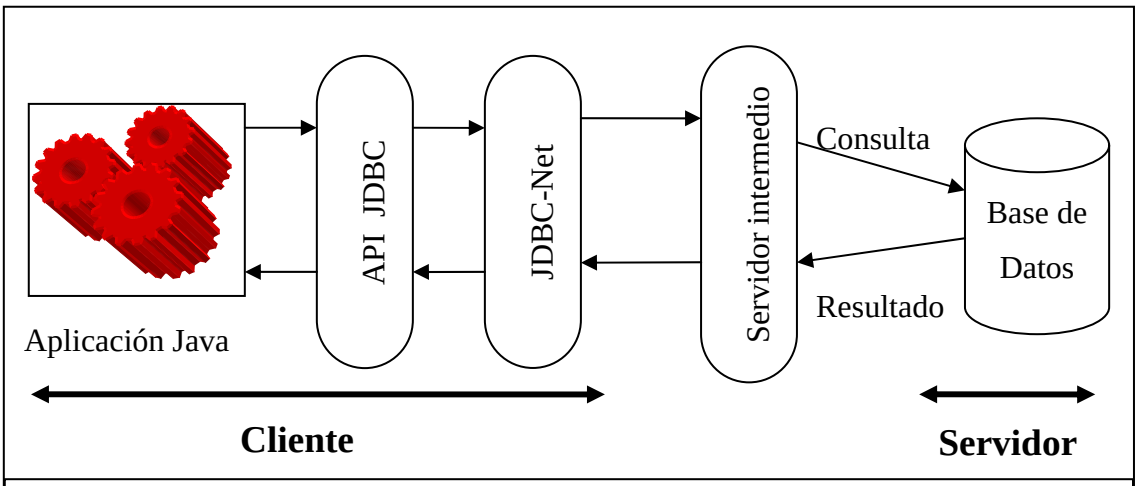


Figura 3 Manejador JDBC-Net

Manejador de Protocolo Nativo (tipo4)

El manejador traduce directamente las llamadas al API JDBC al protocolo nativo de la base de datos. Es el manejador que tiene mejor rendimiento, pero está más ligado a la base de datos que empleemos que el manejador tipo JDBC-Net, donde el uso del servidor intermedio nos da una gran flexibilidad a la hora de cambiar de base de datos. Este tipo de manejadores tambien emplea tecnología 100% Java.

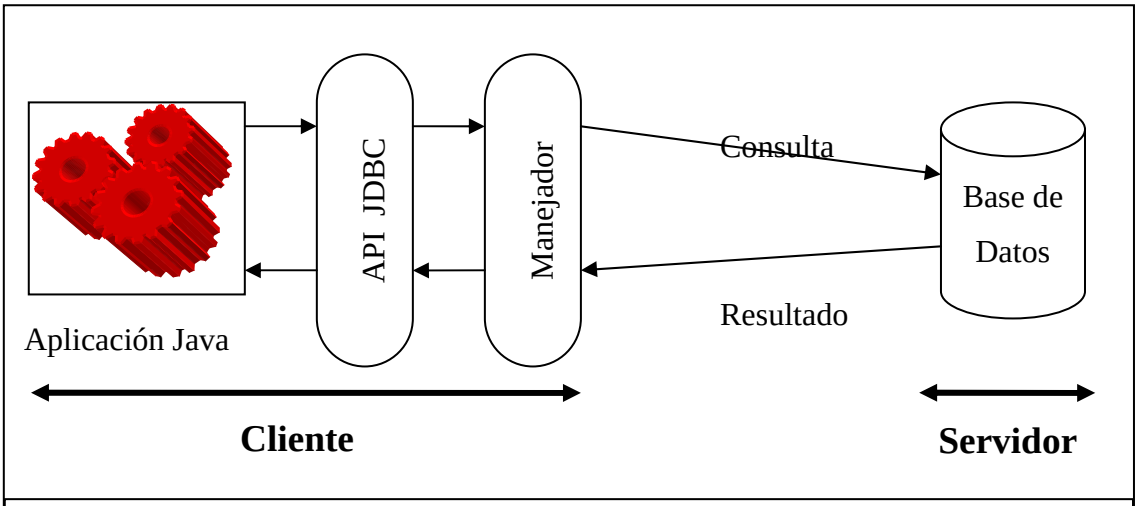


Figura 4 Manejador de Protocolo Nativo

4 El API del desarrollador de aplicaciones, paquete java.sql

En esta sección introduciremos las principales clases e interfaces, del paquete java.sql, que un desarrollador de aplicaciones Java debe emplear para interactuar con una base de datos.

5.1 Clase DriverManager

Como su nombre indica esta clase es la gestora de los diversos drivers (manejadores) que haya en nuestra aplicación. Es posible que sea necesario en una misma aplicación tener varios manejadores para una misma base de datos, acceder a varias bases de datos que emplean distintos manejadores, o bien ambas situaciones a la vez. De ahí el interés de contar con este gestor de manejadores.

Los métodos que más nos interesan de esta clase son *static Connection getConnection(String url)* y *static Connection getConnection(String url, String user, String password)*. Estos métodos intentan establecer conexión con la base de datos que le indiquemos en el campo url empleando para ellos todos los manejadores que hemos registrado. La diferencia entre un método y el otro, obviamente, es que en uno sólo se especifica la base de datos a la que nos queremos conectar, mientras que en el otro se indica también el nombre de usuario de la base de datos y su password. Más adelante explicaremos que es el objeto *Connection* que devuelven estos métodos.

Lo que hace la clase DriverManager para intentar establecer conexión con la base de datos es invocar al método connect de la interface *Driver*, interface que como veremos deben implementar todos los manejadores. Realiza esta operación con todos los manejadores que tiene registrados, si el manejador devuelve null significa que no se ha podido conectar con la base de datos, y el gestor intenta de nuevo conectarse con otro manejador. Si consigue conectarse con un manejador no sigue intentándolo con el resto

de los manejadores registrados, y si no consigue establecer la conexión con ningún manejador lanza una excepción tipo *SQLException*.

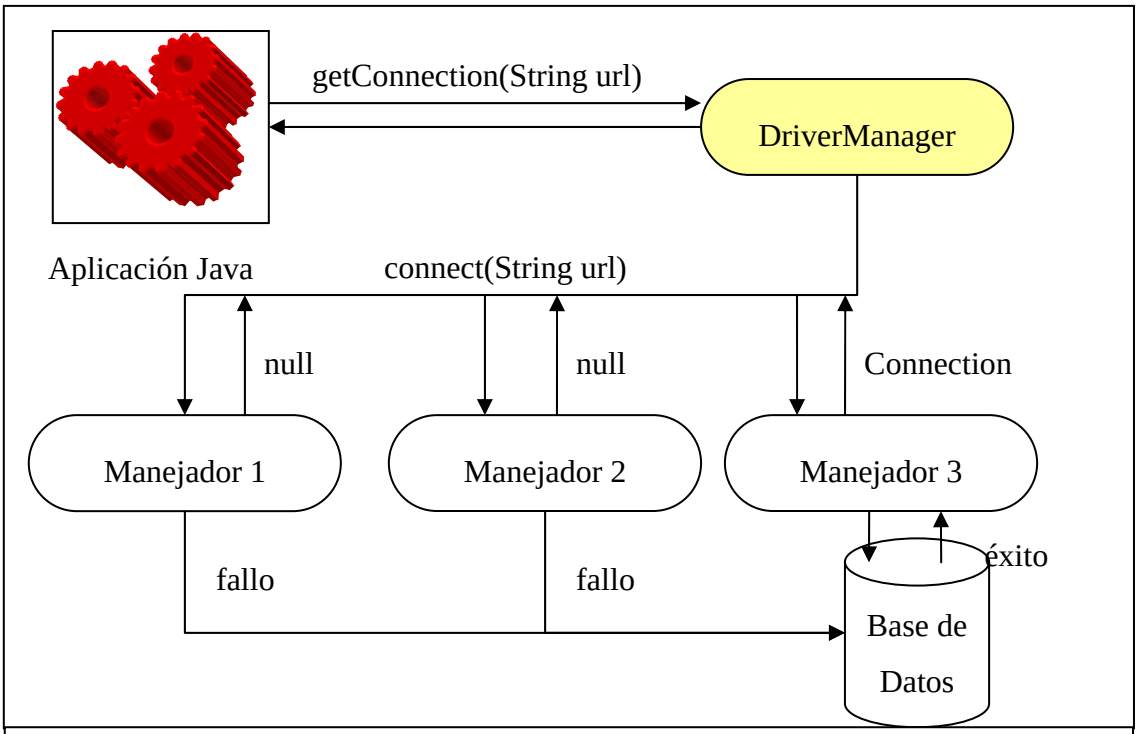


Figura 5Proceso de conexión a la base de datos

Sintaxis de los URL de JDBC

Los URL (Uniform Resource Locator) de JDBC identifican una base de datos, y un protocolo de conexión a esta, de un modo unívoco. Toda URL de JDBC consta siempre de tres partes:

protocolo:subprotocolo:subnombre.

Un ejemplo de URL de conexión podría ser: “jdbc:odbc:usuarios”. Veamos qué es cada una de las partes del URL:

1. Protocolo de la conexión. Siempre es jdbc.
2. Subprotocolo de la conexión. Identifica el tipo de mecanismo de conexión que emplearemos para acceder a la base de datos. Es imprescindible especificarlo para que el *DriverManager* pueda saber que tipo de manejador debe emplear para crear la conexión. En el URL ejemplo el subprotocolo empleado para la conexión a la base de datos será ODBC, por lo que el *DriverManager* intentará establecer la conexión con todos los manejadores registrados que sean de tipo 1, puente JDBC-ODBC.
3. Subnombre, identifica la base de datos con la que queremos establecer una conexión.

En el ejemplo el subnombre es “clientes”; será el sistema ODBC quien proporcione, a partir de ese nombre, el resto de la información necesaria para establecer la conexión con la base de datos. Sin embargo si empleamos otro protocolo puede ser necesario incluir la dirección de red del servidor como subnombre. Así por ejemplo para conectar con una base de datos remota la sintaxis de la URL es:

protocolo:subprotocolo://servidor:puerto/subnombre

Así por ejemplo la URL: jdbc:bdnet://javahispano.org:4040/usuarios Emplearía el protocolo jdbc y un manejador de tipo JDBC-Net para conectarse a la hipotética base de datos “usuarios” a través del puerto 4040 de la máquina javahispano.org.

Registro de un manejador

Registrar un manejador no es más que cargar en memoria una clase que implementa el interfaz *Driver*, clase proporcionada por los desarrolladores de la base de datos. Existen tres formas de registrar un Driver:

1. Empleando el cargador de clases. Para ello empleamos, al igual que para cargar cualquier otra clase, el método *forName(String clase)* de la clase *Class*.

```
Class.forName("com.mysql.Driver");
```

2. Instanciando la clase que implementa el interfaz Driver. De este modo tendremos una referencia al Driver, pudiendo establecer la conexión directamente con él sin necesidad de emplear la clase DriverManager. La sintaxis es la misma que para instaciar cualquier otra clase:

Driver driverPraMySQL = new org.gjt.mm.mysql.Driver();

3. Definir la propiedad jdbc.drivers. Podemos hacerlo de dos modos diferentes:
    a. Definiendo la propiedad jdbc.drivers en nuestra máquina virtual. La sintaxis es jdbc.drivers=nombreManejador1: nombreManejador2:…

jdbc.drivers=gjt.mm.mysql.Driver:oracle.jdbc.driver.OracleDriver;

    b. Emplear la opción –D al invocar a la máquina virtual. Para ellos debemos arrancar nuestra aplicación con el comando java -Djdbc.drivers= nombreManejador1: nombreManejador2:…

java -Djdbc.drivers=gjt.mm.mysql.Driver:oracle.jdbc.driver.OracleDriver;

5.2 Interfaz Driver

Aunque es vital para establecer la conexión con la base de datos en la práctica el desarrollador de aplicaciones puede olvidarse de esta interfaz. Es responsabilidad del desarrollador del manejador proveer una clase que la implemente esta interfaz, y será responsabilidad del *DriverManager* llamar al método *connect* de los drivers registrados para intentar establecer la conexión con la base de datos.

No obstante es posible crear un objeto tipo *Driver*, instanciando la clase desarrollada por el desarrollador del manejador, e invocar directamente a su método *connect*, obteniendo una conexión con la base de datos sin necesidad de emplear la clase *DriverManager*. De este modo tenemos más control sobre el Driver que estamos empleando en la conexión, cuando hemos registrado más de un driver, ya que no nos arriesgamos a que se establezca conexión con la BD con un driver distinto al que nosotros deseamos usar.

5.3 Interfaz Connection

Representa una conexión con la base de datos. Permite crear objetos que representan consultas que se ejecutarán en la base de datos, y permite acceder a información sobre la base de datos y las posibilidades del manejador JDBC.

En esta tabla recogemos los principales métodos de la interfaz *Connection*. Sobre muchos de ellos volveremos a hablar más adelante.

Métodos de la Interfaz Connection	
`void close()`	Libera lo los recursos de esta conexión.
`void commit()`	Hace permanentes los cambios que se realizaron desde el ultimo commit o rollback.
`Statement createStatement()`	Crea un objeto de tipo Statement.
`bolean getAutoCommit()`	Indica si está en modo auto-commit.
`DatabaseMetaData getMetaData()`	Devuelve un objeto tipo `DatabaseMetaData` con meta información a cerca de la base de datos contra la cual se ha abierto esta conexión..
`CallableStatement`	Crea un objeto `CallableStatement` para ejecutar

`prepareCall(String sql)`	procedimientos almacenados SQL en la BD.
`PreparedStatement prepareStatement (String sql)`	Crea un objeto `PreparedStatement` para ejecutar consultas SQL parametrizadas en la BD.
`void rollback()`	Deshace todos los cambios realizados desde la última vez que se ejecutó comit o rollback
`void setAutoCommit (boolean autoCommit)`	Indica el modo de commit en el cual se opera.

5.4 Ejecución de instrucciones: Satment, PreparedStatment y CallableStatment

Interface Statement

Esta interfaz permite enviar instrucciones SQL a la base de datos. Podemos obtener un objeto que implemente esta interfaz a partir del método *`Statement createStatement()`* de la interfaz *`Connection`*. Para enviar una consulta tipo SELECT se emplea el método *execteQuery(String sql)*. Este método devuelve un objeto tipo *Resulset*. Para enviar una instrucción tipo DELETE, UPDATE, INSERT o una instrucción DDL (Data Definition Language) se emplea *executeUpdate(String sql)*. Mediante el método execute(String sql) podemos ejecutar cualquiera de los comandos anteriores.

Interfaz PreparedStatment

Representa una instrucción SQL preparada, esto es, una instrucción SQL precompilada cuya ejecución es mucho más eficiente que ejecutar repetidas veces una misma instrucción SQL. Cuando vayamos a ejecutar varias veces las mismas instrucciones debemos emplear esta interfaz.

Podemos obtener un objeto que implemente esta interfaz a partir del método `PreparedStatement createPreparedStatement(String sql)` de la interfaz `Connection`. La interfaz *PreparedStatment* extiende a *Statment* añadiendo una serie de métodos *setXXX(int indice, XXX valor)* que permiten, para cada ejecución de la instrucción, asignar un valor a los parámetros de la instrucción SQL precompilada. El primer valor, siempre un entero, es el índice del parámetro al cual le vamos a asignar un valor, empezando a contar en 1, y "valor" es el valor que le asignamos a ese parámetro.

La interfaz CallableStatment

Permite ejecutar instrucciones no SQL en la base de datos, como por ejemplo procedimientos almacenados. Extiende a la interfaz *PreparedSatatmen*. Podemos obtener un objeto que implemente esta interfaz a partir del método `CallableStatement prepareCall(String sql)`de la interfaz `Connection`.

5.5 Interfaz ResulSet

Esta interfaz representa un conjunto de datos que son el resultado de una consulta SQL. La clase posee una serie de métodos *XXX getXXX(int columna)* y *XXX getXXX(String columna)* que permiten acceder a los resultados de la consulta (para la sintaxis concreta de estos métodos acudir al javadoc de la interfaz ResulSet). Para acceder a los distintos registros empleamos un cursor, que inicialmente apunta justo antes de la primera fila. Para desplazar el cursor empleamos el método next().

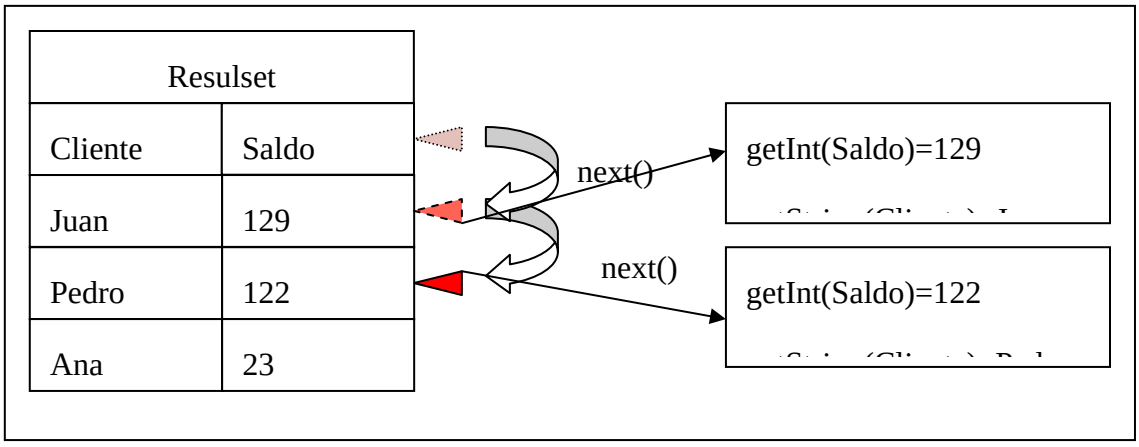


Figura 6 Funcionamiento del Resulset

4 Una aplicación JDBC básica

Toda aplicación que acceda a una base de datos empleando el API JDBC debe realizar una serie de pasos:

1. Establecer una conexión con la base de datos.
2. Ejecutar una o varias instrucciones SQL
3. Si las intrusiones devuelven datos (SELECT) debemos procesarlos.
4. Liberar los recursos de la conexión.

En este apartado mostraremos cómo abordar cada uno de estos pasos mediante el análisis de fragmentos del código HolaMundoJDBC.java, que se distribuye junto con este tutorial.

5.1 Conectándose a la base de datos

En un primer lugar debemos de registrar el o los manejadores que queramos emplear para acceder a la base de datos. Para ello debemos asegurarnos que el archivo .jar que contiene los drivers de nuestra base de datos está incluido en el CLASSPATH que emplea nuestra máquina virtual. También debemos sustituir en el código HolaMundoJDBC.java el valor de la variable *String driver* por un *String* que referencia al manejador de la base de datos que vas a usar.

Para registrar el manejador podemos emplear cualquiera de los procedimientos recogidos en la sección 0. En este código se ha empleado el cargador de clases: *Class.forName(String driver).*

Para establecer la conexión emplearemos el método *getConection* de la clase *DriverManager*. Debes de sustituir antes el valor de la variable *String url* por la URL de tu base de datos. En la documentación del driver debe haber información a cerca de que

URL debes usar. También debes proporcionar, en caso de ser necesario, el nombre de un usuario de la base de datos y su password.

Aquí tienes el fragmento de código del ejemplo HolaMundoJDBC.java que se encarga de realizar estas tareas.

```
//Reemplaza esta variable con el driver de tu base de datos
    String driver = "Tu Driver";
//Remplaza esta variable con la url de tu base de datos
    String url = "La URL de tu base de datos";
    Connection conexion = null;
    try {
      //Registramos el manejador
      Class.forName(driver);
      //Establecemos una conexión con la base de datos
      conexion = DriverManager.getConnection(url, "user",
"");
…
```

5.2 Ejecutar las instrucciones SQL

Para ejecutar las instrucciones SQL debemos crear un objeto tipo *Statment*, *PreparedStatment* o *CallableStatment*, dependiendo de si queremos ejecutar SQL, SQL precompilado o instrucciones no-SQL. Para ello empleamos los métodos definidos con tal propósito en la interface *Connection* (ver sección 5.3).

En nuestro caso será un objeto tipo *Statment*: *statment = conexion.createStatement()*. A continuación ejecutamos las instrucciones SQL, empleando los métodos *executeUpdate(String sql)* para instrucciones tipo UPDATE, DELETE, INSERT o instrucciones DDL. Para las intrusiones de tipo SELECT empleamos el método *Resulset executeQuery(String sql).*

En este caso primero se ejecuta una instrucción DDL, en la que se define una nueva tabla en la base de datos, a continuación creamos una serie de registros en la tabla, y finalmente realizamos una consulta en la cual pedimos el campo NOMBRE de todos los registros cuyo campo NIVEL_PARTICIPACION sea superior a 0.5.

```
  //Creamos un objeto Statment que nos permitirá ejecutar
instrucciones SQL
      statment = conexion.createStatement();
 //Ejecutamos una instrucción de DDL para crear una tabla
      statment.executeUpdate("create table usuarios" +
                             "(NOMBRE varchar(25), " +
                             "LOGIN varchar(15)," +
                             "EDAD int, " +
                             "NIVEL_PARTICIPACION float)");

      //Insertamos un grupo de registros en la tabla
      statment.executeUpdate(
      "insert into usuarios values('Pepe','pepe',23, 0.64)");
…
      //Ejecutamos una consulta
      Resulset resulset = statment.executeQuery(
          "select NOMBRE from usuarios "+
          "where NIVEL_PARTICIPACION > 0.5");
…
```

En la Figura 7 mostramos la tabla que crean las sentencias SQL de este código.

	NOMBRE	LOGIN	EDAD	NIVEL_PARTICIPACION
1	Pepe	pepe	23	0,64
2	Juan	juan	38	0,23
3	Antonio	anton	28	0,82
4	Maria	mar	22	0,84
5	Natalia	Nati	35	0,33
6	Paco	paco	42	0,22

select NOMBRE from usuarios
where NIVEL_PARTICIPACION > 0.5

	NOMBRE
1	Pepe
2	Antonio
3	Maria

Figura 7 Tabla generada por el SQL del código HolaMundoJDBC.java. A la Derecha representamos el ResultSet obtenido por la sentencia SELECT de ese mismo código.

5.3 Procesar los resultados

Tras realizar una consulta tipo *SELECT* obtenemos como resultado un conjunto de registros, a los cuales podemos acceder mediante la interface *Resulset*. Esta provee una serie de métodos getXXX(columna) que permiten acceder a cada campo de cada registro devuelto como respuesta a nuestra consulta.

Es importante conocer a qué tipo Java se corresponde cada tipo SQL para saber mediante qué método debemos acceder al contenido de cada columna. En la Figura 8 mostramos la equivalencia entre los tipos de dato Java y los tipos de datos SQL.

En nuestro código la única operación que se realiza sobre los resultados es imprimirlos por pantalla junto con un pequeño mensaje de agradecimiento. En un código real podrían realizarse operaciones de todo tipo, inclusive operaciones que supongan modificar registros en la base de datos, o ejecutar nuevamente consultas contra esta y volver a procesar sus resultados.

Tipo Java	Tipo SQL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
java.math.BigDecimal	NUMERIC
String	VARCHAR o LONGVARCHAR
byte[]	VARBINARY o LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Figura 8 Correspondencia entre tipos de dato Java y tipos de dato SQL

Debemos tener en cuenta siempre que cada objeto *Resulset* está ligado a su conexión, por lo que si tenemos un objeto *Resulset* y ejecutamos una sentencia tipo SELECT los resultados de la anterior consulta se perderán, si no hemos hecho una copia de ellos.

```
//Mientras queden resultados por procesar
//Recuerda que next() desplaza el cursor del Resulset una
//posición hacia delante cada vez que se invoca.
      while (resulset.next()) {
//Accedemos a los resgistros resultado mediante los métodos
//getXXXX Indicando el nombre de la columna
System.out.println(resulset.getString("NOMBRE")+
            " gracias por ser un usuario tan activo");
      }
```

5.4 Liberar los recursos de la conexión

Una vez que hayamos terminado las operaciones que queríamos realizar en la base de datos debemos liberar todos los recursos que estaba consumiendo la conexión. Aunque estos recursos son automáticamente liberados por el GarbageCollector cundo sus correspondientes objetos son eliminados, es buena practica liberarlos manualmente, para que esta liberación se produzca en cuanto dejan de ser útiles.

Los recursos que debemos liberar son el *Resultset*, el *Statment* y la propia conexión. Para liberarlos debemos invocar al método *close()* de sus correspondieres objetos. Invocar al método *close()* de *Statment* automáticamente libera los recursos de su *ResultSet*. Aquí tenemos el código de HolaMundoJDBC que libera los recursos.

```
    //No nos olvidemos de liberar todos los recursos que
hemos empleado una vez
    //que no necesitemos la conexión: debemos liberar tanto
los recursos
    //consumidos por el Resulset, por los Statment empleados
y las propias
    //Conexiones
    try {
        resulset.close();
        statment.close();
        conexion.close();
    }
```

5 Gestión de excepciones JDBC

En el ejemplo HolaMundoJDBC.java tanto el código para establecer la conexión, como el código que accede a la base de datos están dentro de bloques try-catch. Todos los métodos definidos en las interfaces *Connection*, *Statment* y *ResultSet*, así como el método *getConnection()* de la clase *DriverManager* lanzan excepciones de tipo *java.sql.SQLException*, por lo que todo el código que de un u otro modo acceda a la base de datos o procese los resultados de un acceso ha de estar dentro de una clausula try/chatch, o bien ha de estar dentro de un método que lance excepciones de tipo *SQLException*.

5.1 La Clase SQLException

La excepción *SQLException* tiene algunas características que la diferencian del resto de las excepciones. Más que una excepción propiamente dicha, es una lista de excepciones encadenadas. Muchos y muy distintos problemas pueden suceder en el servidor, y no sería fácil almacenar toda la información correspondiente a todos los problemas en una única excepción. Por ello la clase *SQLException* ha sido diseñada como una cadena de Excepciones, permitiendo almacenar información de todos los errores surgidos en la operación. Esta clase tiene un método, *getNextException()*, que nos devuelve la siguiente excepción de la cadena , o null si no hay más excepciones encadenadas.

El código que mostramos a continuación permite imprimir un código de error del fabricante de la base de datos, el mensaje de error, y la traza de todas las excepciones encadenadas en una *SQLException*

```
    catch (SQLException e) {
      do{
        System.out.println(e.getErrorCode());
        System.out.println(e);
```

```
            e.printStackTrace();
            e = e.getNextException();
          }while (e != null);
        }
```

Las principales causas de que se produzcan excepciones SQL son:

1. Se pierde la conexión con la base de datos. Muchas cosas pueden ir mal cuando la base de datos es remota y se accede a ella a través de la red.
2. Se envía un comando SQL no válido; por ejemplo, tecleamos “wehere” en vez de “where”. En este sentido resulta muy útil probar el SQL empleando alguna interfaz gráfica de nuestra base de datos antes de empotrarlo en el código Java.
3. Se hace referencia a una columna o tabla que no existen
4. Se emplea una función que no está soportada por la base de datos.

5.2 La clase SQLWarning

Es una subclase de *SQLException*. Un *SQLWarning* se genera cuando se produce un error no fatal en la base de datos. Se encadenan igual que las excepciones SQL, pudiendo acceder al siguiente *SQLWarning* de la cadena mediante el método *getNextWarning()*.

A diferencia que las excepciones los warning no se lanzan, cuando queramos acceder a ellos deberemos emplear el método *getWarnings()*, que se haya definido en las interfaces *Connection*, *Statment* y *Resultset*.

6 Metainformación

El API JDBC nos permite acceder a metainformación sobre la base de datos y sobre el *ResultSet* generado por una consulta, es decir, a información a cerca de las características (no del contenido) de la base de datos y del *Resultset*, información como el nombre de la base de datos, la versión, el número y nombre de las columnas del *Resultset*…

La metainformación de la base de datos está representada por un objeto que implementa la interfaz *java.sql.DatabaseMetaData*, objeto al cual nos da acceso el método *getMetaData()* de la interface *Connection*. En *DatabaseMetaData* hay, por ejemplo métodos como *getDriverName()*, que nos devuelve el nombre del driver que estamos empleando, *getDriverVersion()*, que nos devuelve la versión del driver, *getDatabaseProductName()*, el nombre de la base de datos, *getURL()*, la URL de esta base de datos…

La metainformación del *ResultSet* está representada por un objeto que implementa la interfaz *ResultSetMetaData*, accesible mediante el método *getMataData()* de la interfaz ResultSet. En esta interfaz tenemos métodos para, por ejemplo, averiguar el número de columnas del *ResultSet* *getColumnCount()*, el nombre de cada columna, *getColumnName(int columna)*, saber si podemos escribir en una columna, *isWritable(int columna)*, clase Java correspondiente a una columna, *getColumnClassName(int columna)*…

Junto con este tutorial se proporciona un código llamado HolaMundoMetaData.java, en el que se accede a una tabla llamada usuarios (supondremos que es la tabla que creó en la base de datos el código HolaMundoJDBC), y se recupera toda la información de esa tabla y le pasa el *ResultSe*t semejante al método imprimeConsulta(ResultSet), cuyo código se puede ver bajo estas líneas (en el se han omitido algunas líneas que nada tienen que ver con el procesamiento del *ResultSet*, sino con la presentación en pantalla). En este código se accede a la metainformación del *ResultSet* para averiguar el número y nombre de las columnas, y se imprime una tabla en la consola con el resultado.

```
  public static void imprimeConsulta(ResultSet resultSet) {
    try {
      //Obtenemos la Metainformación de ResulSet
      ResultSetMetaData resultSetMetaData =
resultSet.getMetaData();
      //A partir de la metainformación obtenemos el némero de
columnas
      int numeroDeColumnas =
resultSetMetaData.getColumnCount();
      //Para todas las columnas
      for (int i = 1; i <= numeroDeColumnas; i++) {
        //Averguamos cual es el nombre de cada columna
        String nombreDeLaColumna =
resultSetMetaData.getColumnName(i);
        stringBuffer.append(nombreDeLaColumna + "\t");
      }
      //Ahora recorremos el resulset igual que en el primer
ejemplo
      while (resultSet.next()) {
        //Accedesmoa a todas las columnas del registro
        for (int i = 1; i <= numeroDeColumnas; i++) {
        //Acedesmos a todas como String, aunque emplenado la
metainformación
        //sería posible averiguar que tipo de dato es
          stringBuffer.append(resultSet.getString(i) + "\t");
        }
      }
```

```
        System.out.println(stringBuffer.toString());
    }
```

En la Figura 9 presentamos una captura de pantalla de la consola cuando se ejecuta este código y se le pasa como argumento el *ResulSet* generado por la instrucción "select * from usuarios", siendo la tabla usuarios la tabla que se muestra en la Figura 7. En el código original hay más sentencias que generan la parte de la presentación que se muestra en la imagen. En http://www.javahispano.org/codesnippets.item.action?id=366233401 podéis encontrar un código semejante al método que imprime el resultado de la consulta en consola, pero que en vez de imprimir en consola rellana un JTable, con la información del ResultSet.

```
|NOMBRE |LOGIN  |EDAD   |NIVEL_PARTICIPACION
-----------------------------------------
|Pepe   |pepe   |23     |0.64
-----------------------------------------
|Juan   |juan   |38     |0.23
-----------------------------------------
|Antonio        |anton  |28     |0.82
-----------------------------------------
|Maria  |mar    |22     |0.84
-----------------------------------------
|Natalia        |Nati   |35     |0.33
-----------------------------------------
|Paco   |paco   |42     |0.22
-----------------------------------------
```

Figura 9 Salida de pantalla del método imprimeConsulta

7 Transacciones

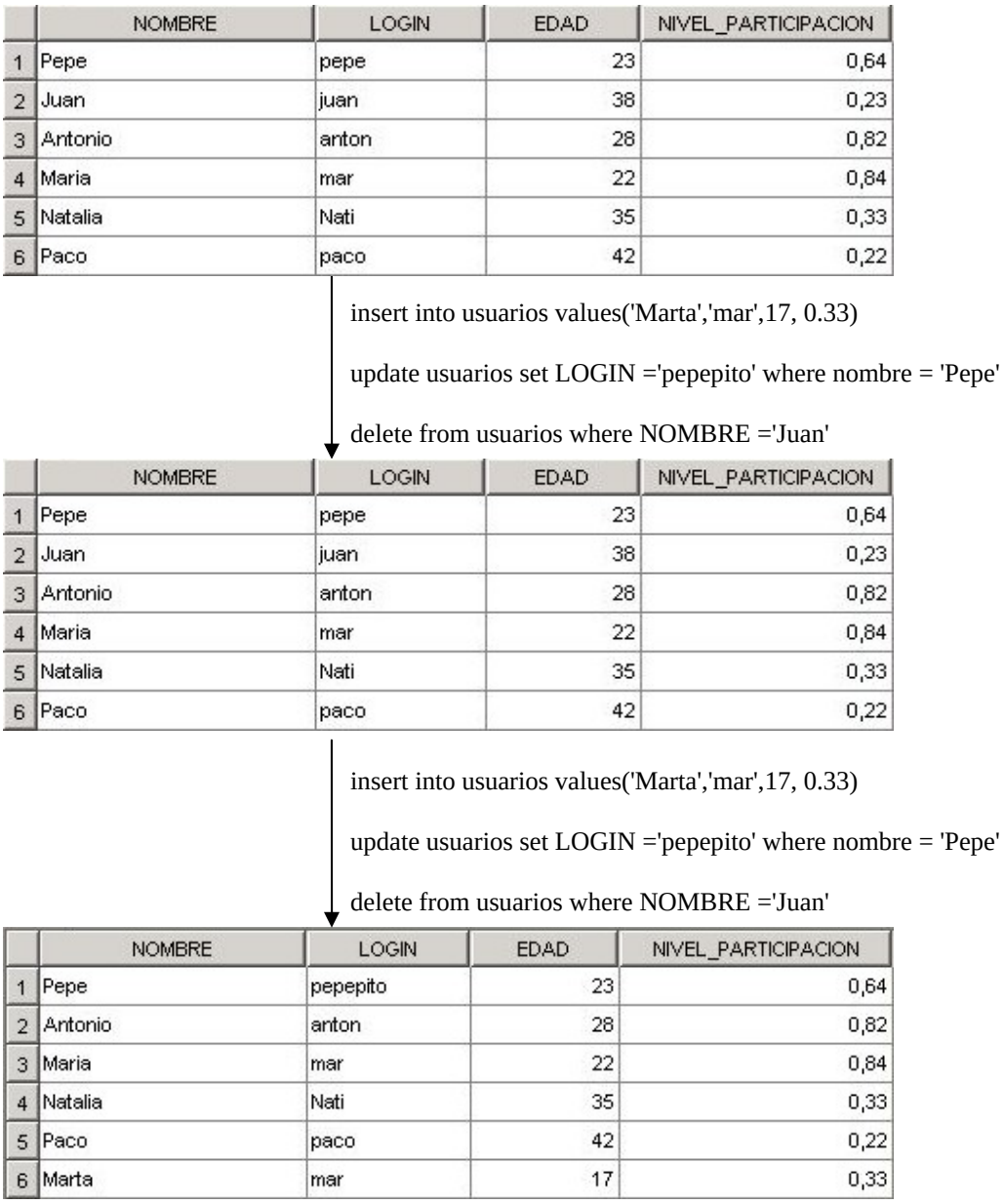
Una transacción es un conjunto de operaciones que deben de realizarse de un modo atómico sobre la base de datos, esto es, o bien se realizan todas ellas correctamente, o bien no se realiza ninguna. Las transacciones son importantes para mantener la consistencia de la información en la base de datos. Imaginemos que tenemos un registro en la tabla “usuario”, la clave primaria de este registro es referenciada por un registro en una tabla llamada pedidos. Borrar a ese usuario implica borrar los pedidos que haya hecho ese usuario; en este caso podemos hacerlo con dos instrucciones DELETE, una sobre el registro del usuario y la otra sobre el pedido. Si la primera instrucción tiene éxito, y la segunda no, en la base de datos tendremos un pedido que referencia a un usuario que no existe. Para evitar este problema podemos agrupar las dos operaciones de borrado en una misma transacción; de ese modo o eliminamos ambos registros o no eliminados ninguno. En cualquier caso la información de la base de datos será consistente.

Para emplear transacciones con el API JDBC debemos invocar el método *setAutocommit(false)* de la interfaz *Connection*. Por defecto la interfaz *Connection* ejecuta un commit, es decir, una confirmación de los cambios realizados en la base de datos, cada vez que ejecutamos una sentencia. Este método nos permite activar y desactivar este comportamiento.

A continuación ejecutamos las operaciones que deseemos agrupar en una única transacción, y finalmente, si queremos confirmar los cambios, invocamos el método *commit()*, del interfaz *Statment*. En ese momento los cambios realizados se harán permanentes en la base de datos y se liberará cualquier tipo de bloqueo que haya realizado esta conexión.

Si queremos cancelar los cambios invocamos al método *rollback()*, también de la interfaz *Statment*. Este método deshace todos los cambios hechos en la base de datos desde el último commit o rollback y libera cualquier tipo de bloqueo que haya realizado esta conexión. Cuando se gestionan las excepciones del código transaccional, siempre se suele poner un rollback en el catch, de ese modo si hay un error en la ejecución de alguna de las operaciones de la transacción deshacemos los cambios de las que se ejecutaron con éxito.

Junto con el tutorial se distribuye un código con nombre HolaMundoTransacciones.java, en el cual se ejecutan varias operaciones sobre la tabla de la Figura 7, operaciones que se muestra en la figura Figura 10. Primero se ejecuta un método llamado cancelaOperaciones(), que ejecuta una inserción, un borrado y una actualización sobre la tabla. Finalmente se llama a *rollback()*, y se imprime la tabla por consola. No ha cambios en ella, ya que los cancelamos al invocar a *rollback()*.



	NOMBRE	LOGIN	EDAD	NIVEL_PARTICIPACION
1	Pepe	pepe	23	0,64
2	Juan	juan	38	0,23
3	Antonio	anton	28	0,82
4	Maria	mar	22	0,84
5	Natalia	Nati	35	0,33
6	Paco	paco	42	0,22

	NOMBRE	LOGIN	EDAD	NIVEL_PARTICIPACION
1	Pepe	pepe	23	0,64
2	Juan	juan	38	0,23
3	Antonio	anton	28	0,82
4	Maria	mar	22	0,84
5	Natalia	Nati	35	0,33
6	Paco	paco	42	0,22

	NOMBRE	LOGIN	EDAD	NIVEL_PARTICIPACION
1	Pepe	pepepito	23	0,64
2	Antonio	anton	28	0,82
3	Maria	mar	22	0,84
4	Natalia	Nati	35	0,33
5	Paco	paco	42	0,22
6	Marta	mar	17	0,33

Figura 10 Ejemplo de transacciones. Muestra las operaciones que se hacen en el código HolaMunodoTransacciones.java y sus efectos sobre la base de datos.

A continuación se llama al método ejecutaOperaciones(), que realiza las mismas operaciones sobre la tabla, pero esta vez llama a *commit()*, con lo que sí se hacen permanentes los cambios en la base de datos. La tabla se vuelve a imprimir por consola y vemos como efectivamente se han realizado los cambios.

Observad que en el código, antes de ejecutar ninguna operación, se ha desactivado el modo autocommit de la conexión mediante el método *setAutocommit(false).* Si no hubiésemos hecho esto aunque llamásemos a *rollback()* los cambios efectuados por el método cancelaOperaciones() se habrían hecho permanentes.

A continuación mostramos los fragmentos del código de HolaMundoTransacciones.java relacionados con el control de las transacciones.

```
public static void cancelaOperaciones(Statement statment) {
    try {
   //Ejecutamos varias operaciones sobre la base de datos.
      statment.executeUpdate(
          "insert into usuarios values('Marta','mar',17,
0.33)");
      statment.executeUpdate(
          "update usuarios SET LOGIN ='pepepito' where nombre
= 'Pepe'");
      statment.executeUpdate(
          "delete from usuarios where NOMBRE ='Juan'");
      //Obtenemos el Objeto Conection de este Statment y
cancelamos los cambios
      statment.getConnection().rollback();
…
  }
  public static void ejecutaOperaciones(Statement statment) {
    try {
```

```
   //Ejecutamos varias operaciones sobre la base de datos.
      statment.executeUpdate(
          "insert into usuarios values('Marta','mar',17,
0.33)");
      statment.executeUpdate(
          "update usuarios SET LOGIN ='pepepito' where nombre
= 'Pepe'");
      statment.executeUpdate(
          "delete from usuarios where NOMBRE ='Juan'");
      //Obtenemos el Objeto Conection de este Statment y
cancelamos los cambios
      statment.getConnection().commit();
…
  }

  public static void main(String args[]) {
…
      //Que no ejecute commit automáticamente
      conexion.setAutoCommit(false);
      //Creamos un objeto Statment que nos permitirá ejecutar
instrucciones SQL
      statment = conexion.createStatement();
      resulset = statment.executeQuery("select * from
usuarios");
      imprimeConsulta(resulset);
      cancelaOperaciones(statment);
      resulset = statment.executeQuery("select * from
usuarios");
      imprimeConsulta(resulset);
```

```
      ejecutaOperaciones(statment);
      resulset = statment.executeQuery("select * from
usuarios");
      imprimeConsulta(resulset);
```

5.1 Control de la concurrencia

Al emplear transacciones para acceder a la base de datos pueden surgir una serie de problemas si hay varias conexiones abiertas al mismo tiempo contra la base de datos (la base se está accediendo de modo concurrente). Es importante conocer y entender cuales son esos problemas, así como conocer que niveles de aislamiento transaccional nos proporciona el API JDBC para evitarlos.

Los problemas que pueden surgir son:

1. Lecturas sucias: una fila de la base de datos que ha sido modificada por una transacción y antes de que esta transacción ejecute un commit (si es que finalmente llega a ejecutarse) esa fila es leída por otra transacción.
2. Lecturas no repetibles: una transacción lee una fila, otra transacción modifica esta fila y la primera transacción vuelve a leer la fila, pero esta vez obtiene un resultado diferente de la primera lectura.
3. Lecturas fantasmas: una transacción selecciona todos los registros que satisfacen una determinada condición, otra transacción inserta un nuevo registro que satisface esa condición. La primera transacción vuelve a seleccionar todos los registros que satisfaces la misma condición, pero esta vez obtiene un registro adicional, el que insertó la segunda transacción.

5.2 Control del nivel de aislamiento transaccional

El API JDBC nos permite controlar el nivel de aislamiento transaccional, esto es, cuales de los tres problemas recogidos en el apartado anterior vamos a tolerar en nuestra conexión y cuales no. Para ello empleamos el método *setTransactionIsolation(in nivel)* de la interfaz *Connection*. El entero que se le pasa es una variable estática y final definida dentro de la interfaz *Connection*. Los posibles valores son:

1. Connection.TRANSACTION_NONE: Indica que la conexión no soporta transacciones. Podemos obtener el nivel de aislamiento de una conexión mediante el método *getTransactionIsolation(),* si su valor es éste significa que la conexión no soporta transacciones.
2. Connection.TRANSACTION_READ_UNCOMMITED: Permite que sucedan lecturas sucias, fantasmas y no repetibles. Si una transacción finalmente ejecuta un *rollback()* otras transacciones pueden haber leído información incorrecta.
3. Connection.TRANSACTION_READ_COMMITED: Permite que sucedan lecturas fantasmas y no repetibles, pero no lecturas sucias.
4. Connection.TRANSACTION_REPETABLE_READ: Sólo permite que sucedan lecturas fantasmas, las lecturas sucias y no repetibles no sucederán.
5. Connection.TRANSACTION_SERIALIZABLE: Evita lecturas fantasmas, sucias y no repetibles.

Podría parecer que lo ideal es emplear siempre el máximo aislamiento transaccional, sin embargo esto disminuye notablemente el rendimiento de la base de datos. En general el rendimiento de los accesos a la base de datos es inversamente proporcional al nivel de asilamiento transaccional que empleemos, por lo que debemos estudiar detenidamente que nivel trasnacional nos podemos permitir en nuestra aplicación.

8 Conclusiones

En este tutorial hemos introducido los conceptos más básicos del API JDBC. Nos quedan otros puntos más avanzados sobre este API. Entre ellos estrían las nuevas características de JDBC 2.0, tales como cursores desplazables y actualizables, y el soporte de los nuevos tipos de datos que incorpora SQL3. Para continuar deberíamos abordar las extensiones del API JDBC incluidas en el paquete *javax.sql*, paquete que sólo forma parte de J2EEvhasta la versión 1.4.0 de J2SE, versión que ya lo incluye. Este paquete contiene una serie de clases que son útiles para el acceso a bases de datos desde el servidor. Entre ellas destacan el soporte para JNDI, el empleo de Pool de conexiones, el soporte para transacciones distribuidas (transacciones que involucran operaciones que se realizan sobre dos o más bases de datos diferentes), y los *RowSet* (objetos que encapsulan un conjunto de filas obtenidas como resultado de una consulta a una base de datos y que dan soporte al modelo de componentes *JavaBeans*).

También aconsejo al lector interesado en trabajar con bases de datos empleando el lenguaje de programación Java que estudie la opción de emplear un motor de persistencia. Estos constituyen una solución para persistir nuestros datos (almacenarlos en un soporte físico perdurable del cual los podamos recuperar, como una base de datos) que muchas veces resulta más simple y rápida que el empleo del API JDBC de modo directo. En estos motores de persistencia se describe en un fichero (muchas veces un fichero XML) a que tabla de la base de datos se asocia cada uno de los campos de nuestro objeto. Una vez realizado ese mapeo el almacenar y recuperar información de la base de datos es prácticamente transparente para el programador. Recomiendo al lector interesado en profundizar en el tema el tutorial de Héctor Suárez González, *Manual Hibernate*, http://www.javahispano.org/articles.article.action?id=82.

.