

Programación en J2EE

Acceso a datos. JDBC

INDICE:

Programación en J2EE.....	1
Acceso a datos. JDBC	1
1 Objetivos	2
2 Arquitectura J2EE: Acceso a datos	3
2.1 Contexto.....	3
3 JDBC	5
3.1 Definición	5
3.2 Packages del API JDBC	5
4 Acceso a BBDD mediante JDBC.....	8
4.1 Proceso genérico	8
4.2 Ejemplo desde NetBeans	12
5 Accesos básicos 1 (Insert/ Update / Delete).....	16
5.1 Descripción del Proceso.....	16
5.2 Ejemplo desde Netbeans	17
6 Accesos básicos 2 (Select). Uso de ResultSets	21
6.1 Descripción del Proceso.....	21
6.2 Ejemplo desde NetBeans	23
7 Transacciones	27
7.1 Definición	27
7.2 Niveles de aislamiento	27
7.3 Ejemplo desde NetBeans	28
8 Pool de Base de Datos	30
8.1 Definición	30
8.2 DataSource.....	30
8.3 Ejemplo desde NetBeans	31
9 Ejercicios.....	36
9.1 Ejercicio 1 (Entregado por el profesor)	36
9.2 Ejercicio 2 (3 puntos).....	36
9.3 Ejercicio 3 (3 puntos).....	39
9.4 Ejercicio 4 (3 puntos).....	40
9.5 Ejercicio 5 (1 punto)	40

1 Objetivos

- Manejar una Base de datos a través del uso de Drivers JDBC.
- Abrir conexiones a la base de datos.
- Ejecutar sentencias SQL.
- Procesar los resultados obtenidos.
- Escribir programas que tengan acceso a la información almacenada
- Entender y usar transacciones
- Uso de un pool de conexiones
- Manejo de DataSources

Requeriremos de la base de datos que cumpla con la propiedad ACID. Esto es que permitan realizar transacciones seguras. ACID es un acrónimo de Atomicity, Consistency, Isolation and Durability: Atomicidad, Consistencia, Aislamiento y Durabilidad en español.

- **Atomicidad:** es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.
- **Consistencia:** es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper la reglas y directrices de integridad de la base de datos.
- **Aislamiento:** es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información nunca generará ningún tipo de error.
- **Durabilidad:** es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

3 JDBC

3.1 Definición

JDBC es el acrónimo de *Java Database Connectivity*, un **API** (Application programming interface) incluida en Java, que describe o define una librería estándar para **acceso a fuentes de datos**, principalmente orientado a Bases de Datos relacionales que usan SQL (Structured Query Language).

Consiste en un conjunto de clases e interfaces, escritas en Java, que ofrecen un completo API para la programación de bases de datos, por tanto, se trata de una solución cien por cien Java para el acceso a bases de datos..

JDBC no sólo provee un interfaz para acceso a motores de bases de datos, sino que también **define una arquitectura estándar**, para que los fabricantes puedan crear los drivers que permitan a las aplicaciones java el acceso a los datos.

Básicamente el API JDBC hace posible realizar las siguientes tareas:

- Establecer una conexión con una base de datos.
- Enviar sentencias SQL.
- Manipular los datos.
- Procesar los resultados de la ejecución de las sentencias.

Debido a que JDBC está escrito completamente en Java también posee la ventaja de ser independiente de la plataforma. No será necesario escribir un programa para cada tipo de base de datos, una misma aplicación escrita utilizando JDBC podrá manejar bases de datos Oracle, Sybase, o SQL Server.

Además podrá ejecutarse en cualquier sistema que posea una Máquina Virtual de Java, es decir, serán aplicaciones completamente independientes de la plataforma.

3.2 Packages del API JDBC

El API JDBC proporciona una interfaz de programación para acceso a datos de Base de Datos Relacionales desde el lenguaje de programación Java a través de los paquetes (packages):

- El paquete `java.sql` es el corazón de la API JDBC 2.0.
- El paquete `javax.sql` es la API de Extensión Estándar JDBC 2.0; y proporciona la funcionalidad de fuente de datos (objeto `DataSource`), y agrupación de conexiones (connection pooling).

3.2.1 Package JAVA.SQL

A continuación podemos ver el índice del paquete **JAVA.SQL** (núcleo central de JDBC) en el que se muestra todas las clases, interfaces y excepciones que se encuentran dentro de este paquete:

Interfaces:		
• CallableStatement	• SQLOutput	• Time
• Connection	• Array	• Timestamp
• DatabaseMetaData	• Blob	• Types
• Driver	• Clob	• SQLPermission
• PreparedStatement	• Ref	
• ResultSet	• Struct	
• ResultSetMetaData		
• Statement	Clases:	Excepciones:
• SQLData	• Date	• DataTruncation
• SQLInput	• DriverManager	• SQLException
	• DriverPropertyInfo	• SQLWarning
		• BatchUpdateException

De esta enumeración de interfaces, clases y excepciones los más importantes son:

- ***java.sql.DriverManager***: es la clase gestora de los drivers. Esta clase se encarga de cargar y seleccionar el driver adecuado para realizar la conexión con una base de datos determinada.
- ***java.sql.Connection***: representa una conexión con una base de datos.
- ***java.sql.Statement***: actúa como un contenedor para ejecutar sentencias SQL sobre una base de datos. Este interfaz tiene otros dos subtipos:
 - java.sql.PreparedStatement*** para la ejecución de sentencias SQL precompiladas a las que se le pueden pasar parámetros de entrada; y
 - java.sql.CallableStatement*** que permite ejecutar procedimientos almacenados de una base de datos.
- ***java.sql.ResultSet***: controla el acceso a los resultados de la ejecución de una consulta, es decir, de un objeto Statement, permite también la modificación de estos resultados.
- ***java.sql.SQLException***: para tratar las excepciones que se produzcan al manipular la base de datos, ya sea durante el proceso de conexión, desconexión u obtención y modificación de los datos.
- ***java.sql.ResultSetMetaData***: este interfaz ofrece información detallada relativa a un objeto ResultSet determinado.
- ***java.sql.DatabaseMetaData***: ofrece información detallada sobre la base de datos a la que nos encontramos conectados.

Más adelante estudiaremos el uso concreto de cada elemento.

3.2.2 Package JAVAX.SQL

Respecto al package **JAVAX.SQL** este permite

- Usar el interface DataSource como alternativa a *java.sql.DriverManager* para establecer la conexión con la Base de Datos
- Uso de pools de conexiones a Base de Datos
- Uso de transacciones distribuidas entre varias bases de datos
- Uso de Rowsets para trabajar por eventos contra la base de datos, o trabajar en modo desconectado.

Desde el punto de vista de la programación orientada a Web nos son interesantes las tres primeras posibilidades de javax.sql. En este tema abordaremos las dos primeras (uso de DataSource y pools), mientras que la tercera se deja para cursos más avanzados, ya que implicaría el uso de varias bases de datos trabajando en de forma distribuida y complicaría en exceso el contenido del curso.

4 Acceso a BBDD mediante JDBC

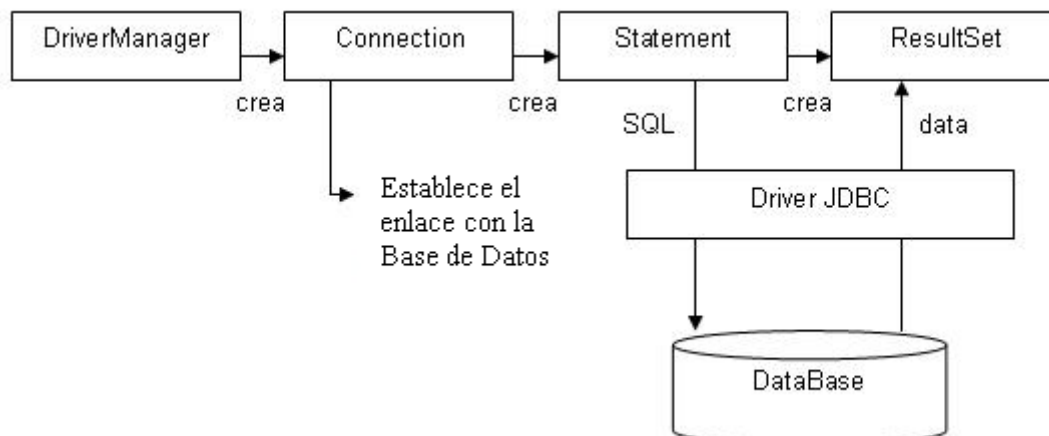
Hasta este punto hemos visto las diferentes partes del API JDBC. Veamos ahora como usar este API JDBC en la práctica usando el **driver** adecuado de Base de Datos. Los drivers nos permiten conectarnos con una base de datos determinada y lanzar **queries** para manipularla. En esta sección abordaremos el proceso de forma genérica, describiendo cada una de las fases. En los siguientes puntos se abordará el proceso de forma específica para las operaciones de manipulación de datos (insert / delete / update) y para las de lectura (select)

4.1 Proceso genérico

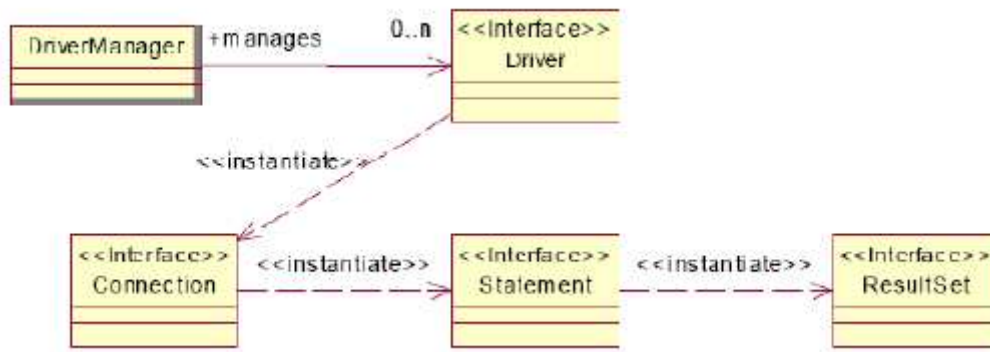
El proceso que se seguirá siempre a la hora de acceder a Base de Datos será

- 1) Establecer la conexión
Permite hacer uso del driver elegido para acceder a la Base de Datos
- 2) Ejecución SQL mediante Statements (sentencias)
Ejecutará las sentencias que queramos lanzar. Si son de tipo selección se devolverá un objeto de tipo ResultSet con las filas seleccionadas. Si son de actualización, borrado o inserción no devolverá ningún objeto de tipo ResultSet.
- 3) Obtener datos mediante ResultSet
En el caso de que la sentencia sea de selección se habrá devuelto un ResultSet que deberemos recorrer accediendo a cada campo de cada fila devuelta.
- 4) Liberar recursos
Es muy importante liberar los recursos en cuanto nos sea posible, dado que en las aplicaciones Web hay muchas peticiones simultáneas queriendo acceder a los mismos recursos

Para llevar a cabo este proceso las principales clases java que se usarán son DriverManager, Connection, Statement y ResultSet. Si los representamos esquemáticamente, junto a su función principal dentro del proceso:



Es de importancia hacer notar que cada instancia de cada clase se crea por medio de otra. Esto es, un objeto de tipo Connection se crea mediante la clase DriverManager, un objeto Statement se crea mediante un objeto de tipo Connection y un objeto ResultSet se crea mediante un objeto Statement.



Veamos con mayor detenimiento cada parte del proceso

4.1.1 Establecer la conexión

Lo primero que tenemos que hacer es establecer una conexión con el controlador (driver) de base de datos que queremos utilizar. Esto implica dos pasos:

- (A) cargar el driver
- (B) hacer la conexión.

4.1.1.1 Cargar el Driver (A)

La carga del driver la realizaremos usando el método `Class.forName(driver)`. Esta carga sólo implica una línea de código. Si, por ejemplo, queremos utilizar el driver `MySQL-Connector/J`, se cargaría la siguiente línea de código:

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

Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

Existe una alternativa (**`javax.sql.DataSource`**) que estudiaremos más adelante junto al uso de Pool de conexiones.

4.1.1.2 Hacer la conexión (B)

El objetivo es conseguir un objeto del tipo `java.sql.Connection` a través del método **`DriverManager.getConnection(String url)`**

Cuando este método es invocado, el `DriverManager` tratará de usar el driver especificado anteriormente para conectar a la base de datos especificada en la URL. Ese driver debería entender el “subprotocolo” que especifica la URL.

Esa URL tendrá la siguiente estructura de nombrado de bases de datos:

`jdbc:<subprotocol>:<subname>`

donde

- `jdbc`: parte fija
- `subprotocol`: mecanismo particular de acceso a base de datos.
- `subname`: dependerá del subprotocolo. JDBC recomienda seguir también la convención de nombrado URL: `//hostname:port/subsubname`.

Por ejemplo, si estamos utilizando JDBC para acceder a una base de datos MySQL llamada "**CientesDB**" nuestro URL podría ser **jdbc:mysql://localhost:3306/CientesDB**.

De este modo un ejemplo para establecer la conexión es:

```
Class.forName("com.mysql.jdbc.Driver");
String urlBBDD = "jdbc:mysql://localhost:3306/CientesDB ";
String login="root";
String passwd="root";
Connection con=(java.sql.DriverManager.getConnection(urlBBDD,login,passwd));
```

En una aplicación real tanto el nombre de la clase del driver, la URL (depende del driver) de acceso a Base de Datos, el nombre de usuario y la contraseña deberían ser configurables (Ej.: leerlos de un fichero de configuración).

4.1.2 Ejecución SQL mediante Statements

Una vez establecida la conexión, esta se usa para pasar sentencias SQL a la base de datos subyacente. JDBC no pone ninguna restricción sobre los tipos de sentencias que pueden enviarse, esto proporciona gran flexibilidad, permitiendo el uso de sentencias específicas de la base de datos, siempre que ésta las soporte.

JDBC suministra tres clases para el envío de sentencias SQL y tres métodos en la interfaz Connection para crear instancias de estas tres clases. Son los siguientes:

- **Statement** – creada por el método `createStatement`. El Objeto statement se usa para enviar sentencias SQL simples, sin parámetros.

El objeto *Statement* proporciona básicamente 3 métodos, *execute()*, *executeUpdate()* y *executeQuery()*, que actúan como conductores de información con la base de datos. La diferencia entre cada uno de estos métodos la mostramos en la siguiente tabla:

Método	Uso recomendado
<code>executeQuery()</code>	Se utiliza con sentencias SELECT y devuelve un <code>ResultSet</code>
<code>executeUpdate()</code>	Se utiliza con sentencias INSERT, UPDATE y DELETE, o bien, con sentencias DDL SQL.
<code>execute()</code>	Utilizado para cualquier sentencia DDL, DML o comando específico de la base de datos

- **PreparedStatement** – Hereda de `Statement`. Creada por el método `prepareStatement`. Un Objeto `PreparedStatement` se usa para ejecutar sentencias SQL que toman uno o más parámetros como argumentos de entrada (parámetros IN). Estos parámetros se especifican mediante signo de interrogación (?) en la sentencia. Se denominan sentencias SQL precompiladas.

Este objeto tiene una serie de métodos (`setXXX`) que fijan los valores de los parámetros IN, los cuales son enviados a la base de datos, cuando se procesa la sentencia SQL. Es más eficiente y rápida que su antecesor.

De este modo un ejemplo para lanzar una sentencia es:

```

PreparedStatement stmt;
stmt = con.prepareStatement("INSERT INTO cliente VALUES(?,?,?,?,?,?,?)");
stmt.setInt(1,"143765987");
stmt.setString(2,"Benito");
stmt.setString(3,"Perez");
stmt.setString(4,"Garrido");
stmt.setString(5,"beni");
stmt.setString(6,"ben1");
stmt.setFloat(7,1000);
stmt.setInt(8,0);

stmt.executeUpdate();

```

- **CallableStatement** - creado por el método `prepareCall`. Los Objetos `CallableStatement` se usan para ejecutar procedimientos almacenados SQL. Un objeto `CallableStatement` hereda métodos para el manejo de los parámetros IN de `PreparedStatement`, y añade métodos para el manejo de los parámetros OUT e INOUT.

Un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros de entrada, resultados, y cualquier combinación de parámetros de entrada/salida.

Nota: En este curso no abordaremos el uso de procedimientos almacenados desde las aplicaciones por falta de tiempo. Estos procedimientos son específicos de cada Base de Datos e implicaría tener que abordar aspectos que este curso no pretende abarcar. Estos conocimientos se impartirán en cursos más avanzados, dado que el uso de procedimientos almacenados incrementa de forma notable la velocidad de acceso y manipulación de datos.

4.1.3 Obtener datos mediante ResultSet

En un objeto *ResultSet* se encuentran los resultados de la ejecución de una sentencia SQL, por lo tanto, un objeto `ResultSet` contiene las filas que satisfacen las condiciones de una sentencia SQL, y ofrece el acceso a los datos de las filas a través de una serie de métodos `getXXX()` que permiten acceder a las columnas de la fila actual.

El método `next()` del interfaz *ResultSet* es utilizado para desplazarse a la siguiente fila del `ResultSet`, haciendo que la próxima fila sea la actual, además de este método de desplazamiento básico, según el tipo de *ResultSet* podremos realizar desplazamientos libres utilizando métodos como `last()`, `relative()` o `previous()`.

Un *ResultSet* mantiene un cursor que apunta a la fila actual de datos. El cursor se mueve hacia abajo cada vez que el método `next()` es lanzado. Inicialmente está posicionado antes de la primera fila. De esta forma, la primera llamada a `next()` situará el cursor en la primera fila, pasando a ser la fila actual.

Las filas del ResultSet son devueltas de arriba abajo, según se va desplazando el cursor, con las sucesivas llamadas al método next(). Un cursor es valido hasta que el objeto ResultSet o su objeto padre Statement, es cerrado.

Un ejemplo para recorrer un ResultSet es:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM cliente");
while (rs.next()) {
    String d = rs.getString("DNI ");
    String n = rs.getString ("Nombre");
    System.out.println(d + " " + n);
}
```

4.1.4 Liberar recursos

La programación en Web tiene la pega de que hay muchos accesos simultáneos a un mismo recurso: la base de datos. Es por ello que tenemos que liberar las conexiones tan pronto nos sea posible.

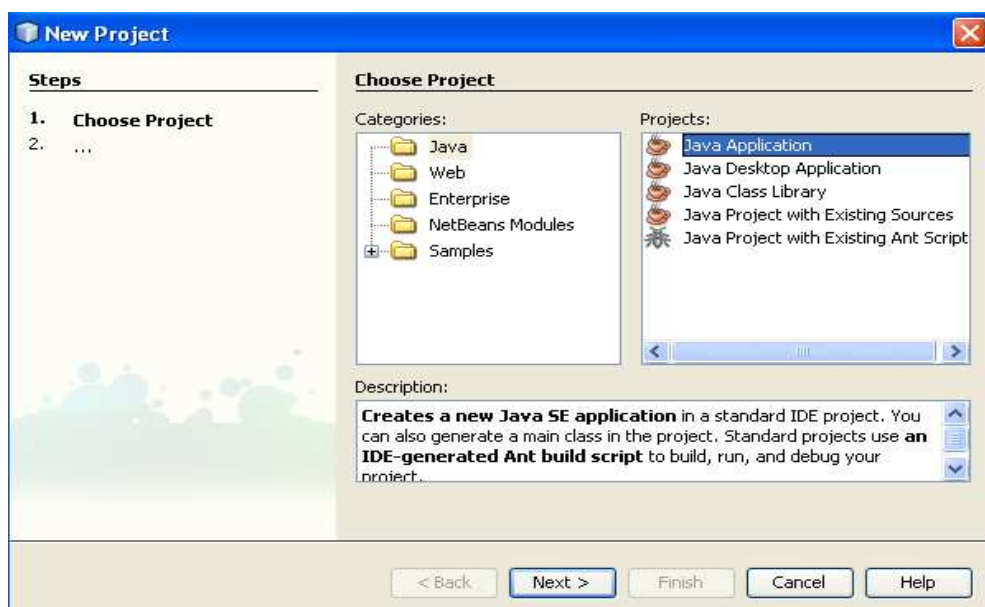
A pesar de que el cierre de una conexión provoca el cierre de todos sus Statements asociados y el cierre de un Statement provoca el cierre de todos sus ResultSet asociados se aconseja realizar el cierre ordenado nosotros mismos. Esto es, por ejemplo:

```
try {
    if (rs != null) rs.close(); //Cerramos el resulset
    if (stmt != null) stmt.close(); //Cerramos el Statement
    if (con != null) con.close(); //Cerramos la conexión
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

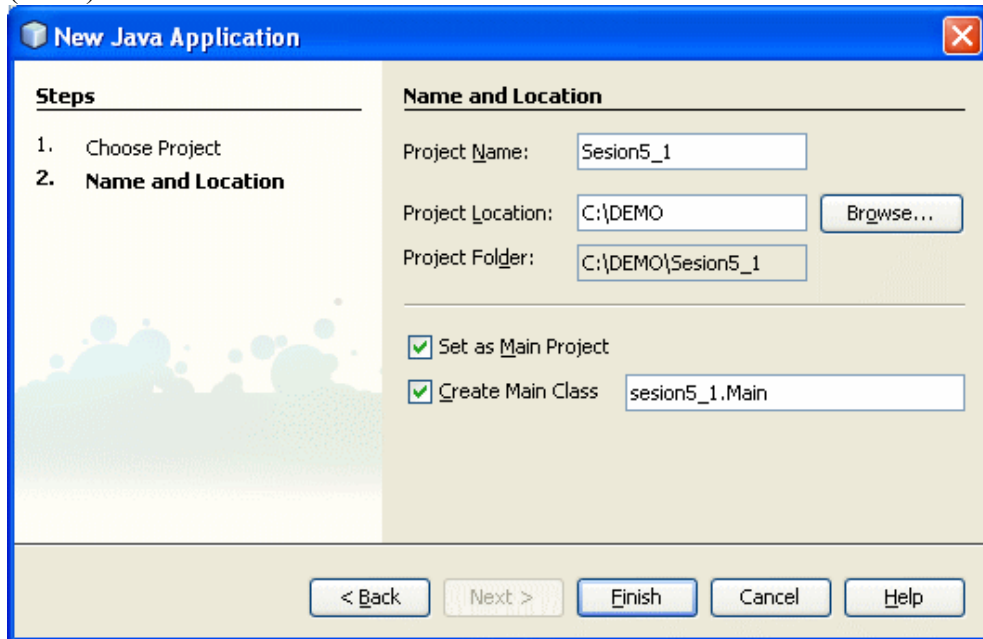
4.2 Ejemplo desde NetBeans

Para las pruebas sobre la base de datos usaremos un proyecto no web, es decir usaremos un aplicación que muestre su salida por consola. Para ello haremos:

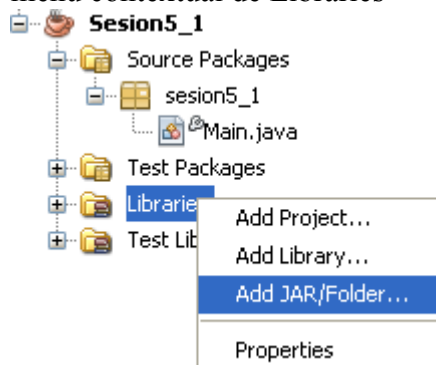
Crearemos un proyecto de tipo Java / Java Application



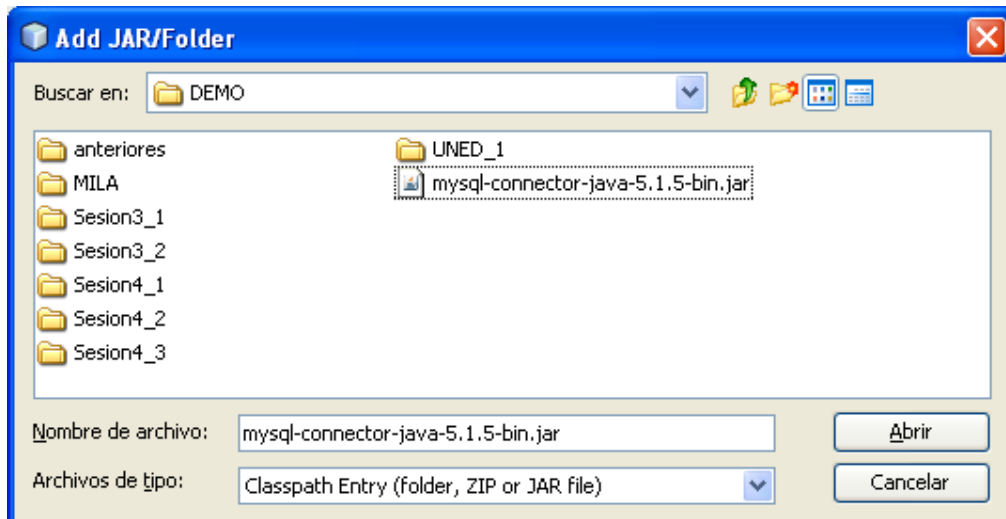
Llamaremos al proyecto sesion5_1 y dejaremos que cree por nosotros la clase principal (Main)



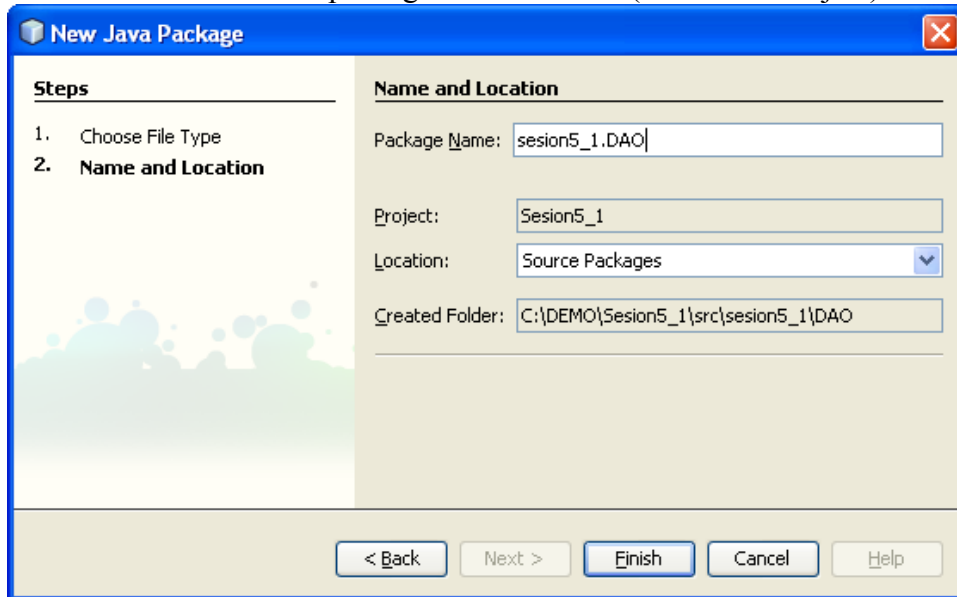
Añadiremos la librería mysql-connector al proyecto mediante la opción Add Jar del menú contextual de Libraries



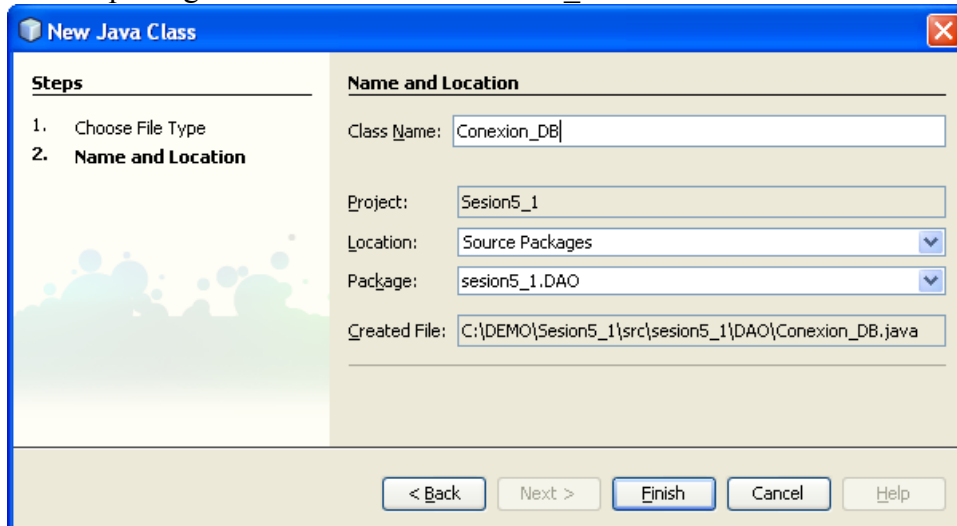
Habremos descargado de <http://www.mysql.com/products/connector/j/> el fichero comprimido que al desempaquetar veremos que contiene el jar **mysql-connector-java-5.1.X-bin.jar** (el último disponible). Seleccionamos ese jar, el cual corresponde al driver de MySQL



A continuación crearemos una clase llamada **Conexion_DB**, en la cual añadiremos unos métodos para abrir y cerrar la conexión a la Base de Datos ClientesDB ya creada. Esta clase la crearemos en un package llamado DAO (data access object). Esto es:



En este package creamos la clase Conexion_DB



Y esta clase tendrá el siguiente código

```
package DAO;
import java.sql.Connection;
import java.sql.SQLException;

public class Conexion_DB {
    public Connection AbrirConexion() throws Exception
    {
        Connection con=null; // instancia una conexión
        try {
            Class.forName("com.mysql.jdbc.Driver"); // Cargar el driver
            String urlOdbc = "jdbc:mysql://localhost:3306/ClientesDB";
            con=(java.sql.DriverManager.getConnection(urlOdbc,"root","root")); //crea conexión
            return con;
        } catch(Exception e){//SQLException y ClassNotFoundException
            e.printStackTrace();
            throw new Exception("Ha sido imposible establecer la conexion"+e.getMessage());
        }
    }
}
```

```

public void CerrarConexion(Connection con) throws Exception
{
    try {
        if (con != null) con.close();
    } catch (SQLException e) {
        e.printStackTrace();
        throw new Exception("Ha sido imposible cerrar la conexion"+e.getMessage());
    }
}
}

```

El cual testaremos a través de la clase principal, cuyo código será

```

package sesion5_1;
import java.sql.Connection;
import DAO.Conexion_DB;

public class Main {

    public static void main(String[] args) {
        try {
            Conexion_DB conexion_DB = new Conexion_DB();
            System.out.println("Abrir Conexión");
            Connection con = conexion_DB.AbrirConexion();
            System.out.println("Conexión abierta");

            System.out.println("Cerrar Conexión");
            conexion_DB.CerrarConexion(con);
            System.out.println("Conexión cerrada");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

Como se puede observar, una vez le demos a  la ejecución de este código únicamente establece y cierra la conexión con Base de Datos. La salida será:

```

Output - Sesión5_1 (run)
init:
deps-jar:
Compiling 2 source files to C:\DEMO\Sesion5_1\build\classes
compile:
run:
Abrir Conexión
Conexión abierta
Cerrar Conexión
Conexión cerrada
BUILD SUCCESSFUL (total time: 2 seconds)

```

De este modo, tendremos una primera clase base (Conexion_DB) para el resto de los ejemplos.

5 Accesos básicos 1 (Insert/ Update / Delete)

En este punto y el siguiente veremos **casos concretos de proceso genérico** de acceso a datos descrito anteriormente.

5.1 Descripción del Proceso

Uno de los procesos habituales, a la hora de realizar mantenimiento sobre una tabla, es el de la actualización, borrado e inserción de datos. En este punto explicaremos como realizar el acceso. Dado que no se devuelven filas de la Base de Datos no utilizaremos la clase ResultSet

Veamos un ejemplo de actualización de datos. Para este menester, SQL nos proporciona la sentencia **UPDATE**.

```
UPDATE tabla SET campo1=valor1, campo2=valor2 WHERE
campo=condicion
```

Se actualizarán todas las filas que cumplan la condición (WHERE campo=condicion). Y la actualización consistirá en asignar nuevos valores a ciertas columnas (SET campo=valor).

Para poder ejecutar esta sentencia de UPDATE mediante JDBC, es necesario realizar los pasos explicados anteriormente:

- establecer la conexión

```
// carga el driver de la bbdd.
String sDriver = "com.mysql.jdbc.Driver";
Class.forName(sDriver).newInstance();

// abrir conexión con la bbdd.
String sURL = "jdbc:mysql://localhost:3306/ClientesDB";
con = DriverManager.getConnection(sURL,"root","password");
```

- ejecución SQL mediante Statements: Una vez tenemos la conexión, preparamos la sentencia. Esto lo hacemos apoyándonos en la clase PreparedStatement. PreparedStatement se utiliza cuando se va a realizar una sustitución de alguno de los valores de la condición, sino, se podría utilizar Statement directamente para ejecutar la sentencia.

```
PreparedStatement stmt;
stmt = con.prepareStatement("UPDATE Usuarios SET Apel='Lopez' WHERE
DNI=?");
stmt.setString(1,"143765987");
```

Solo nos queda ejecutar la actualización. Para ello hay que ejecutar el método executeUpdate() del PreparedStatement

```
int retorno = stmt.executeUpdate();
```


Dicho método devolverá el número de filas que se han actualizado. Será un valor entero desde 0 al número de filas actualizadas.

- Liberación de recursos:

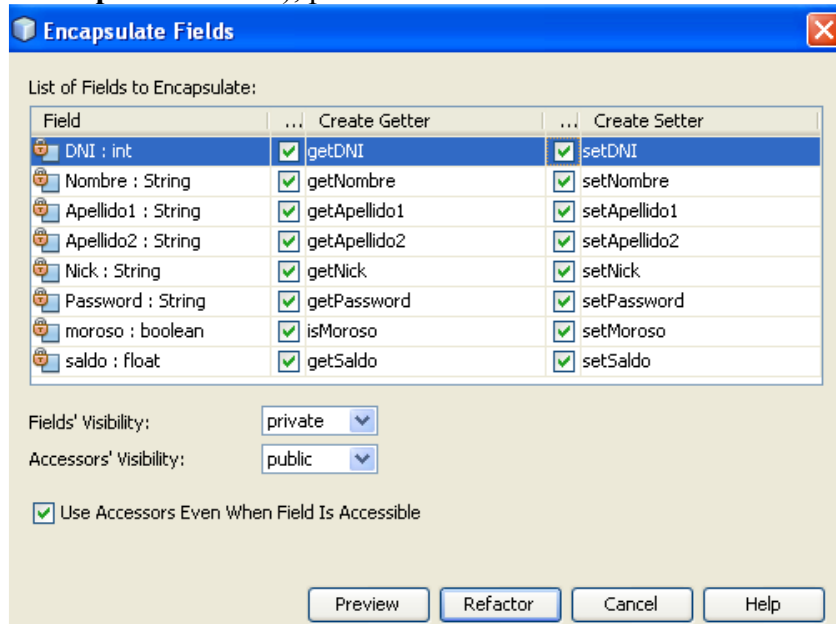
```
try {
    if (rs != null) rs.close(); //Cerramos el resulset
    if (stmt != null) stmt.close(); //Cerramos el Statement
    if (con != null) con.close(); //Cerramos la conexión
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

En el caso que en el proceso, desde la conexión a la ejecución de la sentencia, ocurriese un error, se produciría una excepción **SQLException**. Es por ello que no nos queda más remedio que capturar dicha excepción y ejecutar todo el código entre la sentencia try-catch.

5.2 Ejemplo desde Netbeans

Vamos a ir completando el proyecto Sesión5_1 creado anteriormente con clases que permitan actualizar / insertar y borrar datos de nuestra base de datos de ejemplo.

Primeramente crearemos un JavaBean llamado **Cliente** dentro de un package llamado Entidad, tal y como se ha hecho en sesiones anteriores (**recordemos el uso de Refactor / Encapsulate Fields**), para almacenar la información de cliente en un objeto.



Esta clase no es necesaria para realizar una actualización de datos, pero permitirá tener un código sencillo y fácilmente mantenible. Su código será:

```

package Entidad;

public class Cliente {
    private int DNI;private String Nombre;private String Apellido1;
    private String Apellido2;private String Nick;private String Password;
    private boolean moroso=false;
    private float saldo=0;

    public int getDNI() { return DNI; }
    public void setDNI(int DNI) { this.DNI = DNI; }

    public String getNombre() { return Nombre; }
    public void setNombre(String Nombre) { this.Nombre = Nombre; }

    public String getApellido1() { return Apellido1; }
    public void setApellido1(String Apellido1) {this.Apellido1 = Apellido1; }

    public String getApellido2() {return Apellido2; }
    public void setApellido2(String Apellido2) { this.Apellido2 = Apellido2; }

    public String getNick() { return Nick; }
    public void setNick(String Nick) {this.Nick = Nick; }

    public String getPassword() { return Password; }
    public void setPassword(String Password) {this.Password = Password; }

    public boolean isMoroso() {return moroso;}
    public void setMoroso(boolean moroso) {this.moroso = moroso;}

    public float getSaldo() {return saldo;}
    public void setSaldo(float saldo) {this.saldo = saldo;}
}

```

Nótese en el código anterior su relación con la tabla Cliente de la Base de Datos

Cliente
DNI: INTEGER
Nombre: VARCHAR(50)
Ape1: VARCHAR(50)
Ape2: VARCHAR(50)
Nick: VARCHAR(10)
Passwd: VARCHAR(10)
Saldo: FLOAT
Moroso: INTEGER

De este modo, vamos a realizar actualizaciones sobre la tabla cliente. Es por ello que añadiremos dentro del paquete DAO una clase llamada **ClienteDAO** que será la encargada de realizar todas las operaciones de manipulación de datos contra la tabla Cliente. Añadiremos en esta clase los métodos para modificar, insertar y borrar clientes. Su código será:

```
package DAO;
import Entidad.Cliente;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class ClienteDAO {
    public void actualiza(Connection con, Cliente cliente) throws Exception
    {
        PreparedStatement stmt=null;
        try {
            stmt = con.prepareStatement("UPDATE cliente SET Nombre=?,Ape1=?, "+
                "Ape2=?,Nick=?,Passwd=?,Saldo=?,Moroso=? WHERE DNI=?");
            stmt.setString(1,cliente.getNombre());
            stmt.setString(2,cliente.getApellido1());
            stmt.setString(3,cliente.getApellido2());
            stmt.setString(4,cliente.getNick());
            stmt.setString(5,cliente.getPassword());
            stmt.setFloat(6,cliente.getSaldo());
            if (cliente.isMoroso()) stmt.setInt(7,1); else stmt.setInt(7,0);
            stmt.setInt(8,cliente.getDNI());
            stmt.executeUpdate();
        } catch (SQLException ex) {
            ex.printStackTrace();
            throw new Exception("Ha habido un problema al actualizar cliente "+ex.getMessage());
        } finally
        {
            if (stmt != null) stmt.close();//Cerramos el Statement
        }
    }
    public void elimina(Connection con,Cliente cliente) throws Exception
    {
        PreparedStatement stmt=null;
        try {
            stmt = con.prepareStatement("DELETE FROM cliente WHERE DNI=?");
            stmt.setInt(1,cliente.getDNI());
            stmt.executeUpdate();
        } catch (SQLException ex) {
            ex.printStackTrace();
            throw new Exception("Ha habido un problema al insertar cliente "+ex.getMessage());
        } finally
        {
            if (stmt != null) stmt.close();//Cerramos el Statement
        }
    }
    public void inserta(Connection con,Cliente cliente) throws Exception
    {
        PreparedStatement stmt=null;
        try {
            stmt = con.prepareStatement("INSERT INTO Cliente (DNI, Nombre, Ape1, Ape2, Nick, " +
                "Passwd, Saldo, Moroso) VALUES(?, ?, ?, ?, ?, ?, ?, ?)");
            stmt.setInt(1,cliente.getDNI());
            stmt.setString(2,cliente.getNombre());
            stmt.setString(3,cliente.getApellido1());
            stmt.setString(4,cliente.getApellido2());
            stmt.setString(5,cliente.getNick());
            stmt.setString(6,cliente.getPassword());
            stmt.setFloat(7,cliente.getSaldo());
            if (cliente.isMoroso()) stmt.setInt(8,1); else stmt.setInt(8,0);

            stmt.executeUpdate();
        } catch (SQLException ex) {
            ex.printStackTrace();
            throw new Exception("Ha habido un problema al insertar cliente "+ex.getMessage());
        } finally
        {
            if (stmt != null) stmt.close();//Cerramos el Statement
        }
    }
}
```

Por último, modificaremos la clase Main (pondremos entre comentarios el código anterior) para que pase a utilizar esta clase ClienteDAO para hacer, por ejemplo:

- Insertar un nuevo cliente con DNI a 8977, Nombre a Alejandro, primer apellido a Martinez, Nick a alex, Passwd a ale1, un saldo de 1000 y no moroso
- Modificaremos el nombre del cliente insertado antes cuyo DNI es 8977 a Juan
- Borrar el cliente cuyo DNI es 25

Esto es:

```
package sesion5_1;
import DAO.ClienteDAO;
import java.sql.Connection;
import DAO.Conexion_DB;
import Entidad.Cliente;

public class Main {
    public static void main(String[] args) throws Exception {
        Conexion_DB _conexion_DB = new Conexion_DB();
        Connection _con = null;
        try
        {
            _con = _conexion_DB.AbrirConexion();// Abrimos la conexión
            ClienteDAO _clienteDAO=new ClienteDAO();
            // Insertar nuevo cliente
            Cliente _cliente1=new Cliente();
            _cliente1.setDNI(8977);
            _cliente1.setNombre("Alejandro");
            _cliente1.setApellido1("Martinez");
            _cliente1.setNick("alex");
            _cliente1.setPassword("ale1");
            _cliente1.setSaldo(1000);
            _cliente1.setMoroso(false);
            _clienteDAO.inserta(_con, _cliente1);
            System.out.println("Nuevo cliente insertado");
            // Actualizar anterior cliente
            _cliente1.setNombre("Juan");
            _clienteDAO.actualiza(_con, _cliente1);
            System.out.println("Cliente actualizado");
            // Eliminar cliente cuyo DNI es 25
            Cliente _cliente3=new Cliente();
            _cliente3.setDNI(25);
            _clienteDAO.elimina(_con, _cliente3);
            System.out.println("Cliente eliminado");
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally
        {
            _conexion_DB.CerrarConexion(_con);// Abrimos la conexión
        }
    }
}
```

Lo que dejará la tabla de cliente como se muestra

MySQL Query Browser - Connection: root@localhost:3306 / ClientesDB									
File Edit View Query Script Tools Window MySQL Enterprise Help									
<div> <div>Go back</div> <div>Next</div> <div>Refresh</div> </div> <div>SELECT * FROM clientesdb.cliente c;</div>									
Resultset 1									
	DNI	Nombre	Ape1	Ape2	Nick	Passwd	Saldo	Moroso	
▶	14	Francisco	Lopez	Sanchis	franlo	fran1	1000	0	
	15	Pedro	Martinez	NULL	peter	pet1	1000	0	
	33	Marcos	Heredia	Buendia	marcs	mar1	1000	0	
	8977	Juan	Martinez	NULL	alex	ale1	1000	0	
	367809	Guillermo	Toro	Fuentes	guille	gui1	1000	0	
	456781	Julia	Sanches	Guarner	juli	jul1	1000	0	
	31678901	Ana	Bermejo	Beltan	anaber	ana1	1000	0	

6 Accesos básicos 2 (Select). Uso de ResultSets

6.1 Descripción del Proceso

Para poder ejecutar sentencias de tipo SELECT mediante JDBC, es necesario realizar los pasos explicados anteriormente en el proceso genérico:

- establecer la conexión (exactamente igual que en la actualización)

```
// carga el driver de la bbdd.
String sDriver = "com.mysql.jdbc.Driver";
Class.forName(sDriver).newInstance();

// abrir conexión con la bbdd.
String sURL = "jdbc:mysql://localhost:3306/CientesDB";
con = DriverManager.getConnection(sURL,"root","password");
```

- ejecución SQL mediante Statements

```
PreparedStatement stmt;
stmt = con.prepareStatement("SELECT * FROM cliente WHERE DNI=?");
stmt.setString(1,"143765987");
ResultSet rs = stmt.executeQuery();
```

Es idéntico al anterior pero en vez de devolver un entero devuelve un ResultSet con las filas obtenidas de la base de Datos

- Recorrer el Resultset

Utilizaremos el método *next()* del interfaz *ResultSet* para desplazarnos al registro siguiente dentro de un *ResultSet*. El método *next()* devuelve un valor booleano (tipo *boolean* de Java), true si el registro siguiente existe y false si hemos llegado al final del objeto *ResultSet*, es decir, no hay más registros.

Los métodos *getXXX(Idcolumna)* del interfaz *ResultSet* ofrecen los medios para recuperar los valores de las columnas (campos) de la fila (registro) actual del *ResultSet*. Donde:

- **XXX**

Es el tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, si la primera columna de cada fila almacena un valor del tipo VARCHAR de SQL. El método para recuperar un valor VARCHAR es getString. Si la segunda columna de cada fila almacena un valor del tipo FLOAT de SQL, y el método para recuperar valores de ese tipo es getFloat.

La correspondencia entre tipos Java y tipos SQL es

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.Timestamp	TIMESTAMP
java.sql.Date	DATE
java.sql.Time	TIME

- **Idcolumnna**

Para designar una columna podemos utilizar su nombre o bien su número de orden en la fila. Por ejemplo si la segunda columna de un objeto rs de la clase ResultSet se llama "título" y almacena datos de tipo String, se podrá recuperar su valor de las formas que muestra el siguiente fragmento de código:

```
//rs es un objeto de tipo ResultSet
String valor=rs.getString(2);
String valor=rs.getString("titulo");
```

Por ejemplo, el siguiente código accede a los valores almacenados en la fila actual de rs e imprime una línea con el DNI seguido por tres espacios y el nombre. Cada vez que se llama al método next, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en rs.

```
while (rs.next()) {
    String d = rs.getString("DNI ");
    String n = rs. getString ("Nombre");
    System.out.println(d + "   " + n);
}
```

- Liberar los recursos (exactamente igual que en la actualización)

```
try {
    if (rs != null) rs.close(); //Cerramos el resulset
    if (stmt != null) stmt.close(); //Cerramos el Statement
    if (con != null) con.close(); //Cerramos la conexión
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

Al igual que antes, desde la conexión a la ejecución de la sentencia, puede producirse una excepción **SQLException**. Es por ello que no nos queda más remedio que capturar dicha excepción y ejecutar todo el código entre la sentencia try-catch.

6.2 Ejemplo desde NetBeans

Sigamos completando el proyecto Sesión5_1 creado anteriormente. En este caso queremos ser capaces de recoger un cliente de la Base de Datos a partir de su DNI, pero también queremos ser capaces de recoger listados de clientes. Para esto seguiremos usando el JavaBean llamado Cliente que hemos creado con anterioridad.

En este ejemplo queremos añadir a ClienteDAO la posibilidad de:

- seleccionar un cliente por su Nick (el cual es único)
- seleccionar un cliente por DNI
- Obtener un listado de cliente cuyo DNI comience por una letra
- Obtener el cliente que más haya gastado

De este modo los nuevos métodos a añadir a ClienteDAO que serán:

```
private void obtenClienteFila(ResultSet rs, Cliente cliente) throws SQLException
{
    cliente.setDNI(rs.getInt("DNI"));
    cliente.setNombre(rs.getString("Nombre"));
    cliente.setApellido1(rs.getString("Ape1"));
    cliente.setApellido2(rs.getString("Ape2"));
    cliente.setNick(rs.getString("Nick"));
    cliente.setPassword(rs.getString("Passwd"));
    cliente.setSaldo(rs.getFloat("Saldo"));
    if (rs.getInt("Moroso")==0) cliente.setMoroso(false);
    else cliente.setMoroso(true);
}

public Cliente findByDNI(Connection con, Cliente cliente) throws Exception
{
    Cliente _cliente=null;
    PreparedStatement stmt=null;
    ResultSet rs=null;
    try {
        stmt = con.prepareStatement("SELECT * FROM Cliente WHERE DNI=?");
        stmt.setInt(1, cliente.getDNI());
        rs =stmt.executeQuery();
        while (rs.next()) {
            _cliente=new Cliente();
            obtenClienteFila(rs, _cliente);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
        throw new Exception("Ha habido un problema al buscar el cliente por DNI "+ex.getMessage());
    } finally
    {
        if (rs != null) rs.close(); //Cerramos el resulset
        if (stmt != null) stmt.close(); //Cerramos el Statement
    }
    return _cliente;
}

public Cliente findByNick(Connection con, Cliente cliente) throws Exception
{
    Cliente _cliente=null;
    PreparedStatement stmt=null;
    ResultSet rs=null;
    try {
        stmt = con.prepareStatement("SELECT * FROM Cliente WHERE Nick=?");
        stmt.setString(1, cliente.getNick());
        rs =stmt.executeQuery();
        while (rs.next()) {
            _cliente=new Cliente();
            obtenClienteFila(rs, _cliente);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
        throw new Exception("Ha habido un problema al buscar el cliente por Nick "+ex.getMessage());
    } finally
    {
        if (rs != null) rs.close(); //Cerramos el resulset
        if (stmt != null) stmt.close(); //Cerramos el Statement
    }
    return _cliente;
}
```

Los dos primeros, solo varían en la SELECT. Además creamos un método privado, donde ponemos el código común de obtener un cliente de una fila de la Base de Datos. En el tercer método la SELECT devuelve más de una fila. Observen como se introduce un parámetro de tipo LIKE en la sentencia SQL

```
public List<Cliente> findByNumberDNISearch(Connection con,int numero) throws Exception
{
    List<Cliente> _listaClientes=new ArrayList();
    PreparedStatement stmt=null;
    ResultSet rs=null;
    try {
        stmt = con.prepareStatement("SELECT * FROM Cliente WHERE DNI like ?");
        stmt.setString(1,numero+"%");
        rs =stmt.executeQuery();
        Cliente _cliente=null;
        while (rs.next()) {
            _cliente=new Cliente();
            obtenClienteFila(rs,_cliente);
            _listaClientes.add(_cliente);
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
        throw new Exception("Ha habido un problema al buscar el cliente por Nick "+ex.getMessage());
    } finally
    {
        if (rs != null) rs.close(); //Cerramos el resulset
        if (stmt != null) stmt.close();//Cerramos el Statement
    }
    return _listaClientes;
}
```

En el cuarto método muestra como combinar la potencia de SQL con la potencia de Java, dado que la SELECT no devuelve directamente el cliente que más ha gastado, sino una lista de clientes y los gastos que han realizado de la cual debemos elegir el cliente con mayor gasto. Una vez tengamos esta cliente volveremos a hacer una SELECT para recuperar sus datos. Esto es:

```
public Cliente findByMayorGasto(Connection con) throws Exception
{
    Cliente _cliente=null;
    PreparedStatement stmt=null;
    ResultSet rs=null;
    try {
        stmt = con.prepareStatement("SELECT Cliente_DNI AS DNI,(SUM(Precio*Numero)) AS GASTO "+
        "FROM articulo_factura af, articulo a,factura f "+
        "WHERE af.Articulo_idArticulo=a.idArticulo "+
        "AND af.Factura_idFactura=f.idFactura "+
        "GROUP BY(Cliente_DNI)");

        rs =stmt.executeQuery();
        float _gastoAnterior=0;
        while (rs.next()) {
            float gasto=rs.getFloat("GASTO");
            if (gasto>_gastoAnterior)
            {
                _cliente=new Cliente();
                _cliente.setDNI(rs.getInt("DNI"));
                _gastoAnterior=gasto;
            }
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
        throw new Exception("Ha habido un problema al buscar el cliente por Nick "+ex.getMessage());
    } finally
    {
        if (rs != null) rs.close(); //Cerramos el resulset
        if (stmt != null) stmt.close();//Cerramos el Statement
    }
    if (_cliente!=null) _cliente=findByDNI(con,_cliente);
    return _cliente;
}
```


Por último, modificaremos la clase Main(pondremos entre comentarios el código anterior) para que pase a utilizar esta clase ClienteDAO para hacer, por ejemplo:

- Obtener el cliente cuyo DNI es 8977
- Obtener el password del cliente cuyo Nick es marcs
- Obtener todos los clientes cuyo DNI comience por 3
- Obtener el cliente que haya gastado más

Esto es (se ha añadido un método muestraCliente para mostrar los datos de cliente por pantalla):

```
package sesion5_1;
import DAO.ClienteDAO;
import java.sql.Connection;
import DAO.Conexion_DB;
import Entidad.Cliente;
import Entidad.Factura;
import java.util.List;

public class Main {
    private static void muestraCliente(Cliente _cliente1) {
        System.out.println("Cliente DNI->" + _cliente1.getDNI());
        System.out.println("Cliente nombre->" + _cliente1.getNombre());
        System.out.println("Cliente Primer apellido->" + _cliente1.getApellido1());
        System.out.println("Cliente Segundo apellido->" + _cliente1.getApellido2());
        System.out.println("Cliente Nick->" + _cliente1.getNick());
        System.out.println("Cliente Password->" + _cliente1.getPassword());
        System.out.println("Cliente Saldo->" + _cliente1.getSaldo());
        System.out.println("Cliente Moroso->" + _cliente1.isMoroso());
    }

    public static void main(String[] args) throws Exception {
        Conexion_DB _conexion_DB = new Conexion_DB();
        Connection _con = _conexion_DB.AbrirConexion(); // Abrimos la conexión
        ClienteDAO _clienteDAO = new ClienteDAO();
        System.out.println("CLIENTE CUYO DNI ES 8977");
        Cliente _cliente1 = new Cliente();
        _cliente1.setDNI(8977);
        _cliente1 = _clienteDAO.findByDNI(_con, _cliente1);
        muestraCliente(_cliente1);

        System.out.println("CLIENTE CUYO NICK ES marcs");
        Cliente _cliente2 = new Cliente();
        _cliente2.setNick("marcs");
        _cliente2 = _clienteDAO.findByNick(_con, _cliente2);
        muestraCliente(_cliente2);

        List<Cliente> _clientes1 = _clienteDAO.findByNumberDNISStart(_con, 3);
        for (int i=0; i<_clientes1.size(); i++)
        {
            System.out.println("CLIENTE CUYO DNI COMIENZA CON 3");
            muestraCliente(_clientes1.get(i));
        }
        System.out.println("CLIENTE CON MAYOR GASTO");
        Cliente _cliente3 = _clienteDAO.findByMayorGasto(_con);
        muestraCliente(_cliente3);

        _conexion_DB.CerrarConexion(_con);
    }
}
```

Lo que mostrará por salida de consola al pulsar será  lo siguiente:

```

Output - Sesión5_1 (run)
init:
deps-jar:
Compiling 1 source file to C:\DEMO\Sesion5_1\build\classes
compile:
run:
CLIENTE CUYO DNI ES 8977
Cliente DNI->8977
Cliente nombre->Juan
Cliente Primer apellido->Martinez
Cliente Segundo apellido->null
Cliente Nick->alex
Cliente Password->alel
Cliente Saldo->1000.0
Cliente Moroso->>false
CLIENTE CUYO NICK ES marcs
Cliente DNI->33
Cliente nombre->Marcos
Cliente Primer apellido->Heredia
Cliente Segundo apellido->Buendia
Cliente Nick->marcs
Cliente Password->marl
Cliente Saldo->1000.0
Cliente Moroso->>false
CLIENTE CUYO DNI COMIENZA CON 3
Cliente DNI->33
Cliente nombre->Marcos
Cliente Primer apellido->Heredia
Cliente Segundo apellido->Buendia
Cliente Nick->marcs
Cliente Password->marl
Cliente Saldo->1000.0
Cliente Moroso->>false
CLIENTE CUYO DNI COMIENZA CON 3
Cliente DNI->367809
Cliente nombre->Guillermo
Cliente Primer apellido->Toro
Cliente Segundo apellido->Fuentes
Cliente Nick->guille
Cliente Password->guil
Cliente Saldo->1000.0
Cliente Moroso->>false
CLIENTE CUYO DNI COMIENZA CON 3
Cliente DNI->31678901
Cliente nombre->Ana
Cliente Primer apellido->Bermejo
Cliente Segundo apellido->Beltan
Cliente Nick->anaber
Cliente Password->anal
Cliente Saldo->1000.0
Cliente Moroso->>false
CLIENTE CON MAYOR GASTO
Cliente DNI->33
Cliente nombre->Marcos
Cliente Primer apellido->Heredia
Cliente Segundo apellido->Buendia
Cliente Nick->marcs
Cliente Password->marl
Cliente Saldo->1000.0
Cliente Moroso->>false
BUILD SUCCESSFUL (total time: 3 seconds)

```

7 Transacciones

7.1 Definición

A veces tenemos la necesidad de que varias sentencias SQL se ejecuten como un todo, y si falla alguna, hay que deshacer los cambios efectuados (transferencias entre cuentas bancarias, almacenar los datos de una factura).

El uso de transacciones se controla mediante métodos del objeto **Connection**.

Como ya se ha dicho, **Connection** representa una conexión a una Base de datos dada, luego representa el lugar adecuado para el manejo de transacciones, dado que estas afectan a todas las sentencias ejecutadas sobre una conexión a la base de datos.

Por defecto, una conexión funciona en modo autocommit, es decir, cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción, que sólo afecta a dicha sentencia. Si queremos agrupar varias sentencias SQL en una misma transacción utilizaremos el método `setAutoCommit(false)`, indicará que una sentencia no se ejecute automáticamente.

Una vez que se ha deshabilitado el modo auto-commit, las sentencias que ejecutemos no modificarán la base de datos hasta que no llamemos al método `commit()` (si todo va bien) o `rollback()` (si falla) del interfaz *Connection*. Todas las sentencias que se han ejecutado previamente a la llamada del método `commit()` se incluirán dentro de una misma transacción y se llevarán a cabo sobre la base de datos como un todo.

Por ejemplo:

```
try{
    conexion.setAutoCommit(false);

    Statement sentenciaResta=conexion.createStatement();
    sentenciaResta.executeUpdate(...)
    Statement sentenciaSuma=conexion.createStatement();
    sentenciaSuma.executeUpdate(...)

    conexion.commit();
} catch(SQLException ex){
    conexion.rollback();
    System.out.println("La transacción a fallado.");
} finally{
    //Cerraremos la conexión
}
```

7.2 Niveles de aislamiento

Es posible también especificar el nivel de aislamiento de una transacción, mediante `setTransactionIsolation()`. Los niveles de aislamiento se representan mediante las constantes que se muestran en la lista siguiente, en la cual se explica muy básicamente el efecto de cada nivel de aislamiento.

- **TRANSACTION_NONE**

No se pueden utilizar transacciones.

- **TRANSACTION_READ_UNCOMMITTED**

Desde esta transacción se pueden llegar a ver registros que han sido modificados por otra transacción, pero no guardados, por lo que podemos llegar a trabajar con valores que nunca llegan a guardarse realmente.

- **TRANSACTION_READ_COMMITTED**

Se ven solo las modificaciones ya guardadas hechas por otras transacciones.

- **TRANSACTION_REPEATABLE_READ**

Si se leyó un registro, y otra transacción lo modifica, guardándolo, y lo volvemos a leer, seguiremos viendo la información que había cuando lo leímos por primera vez. Esto proporciona un nivel de consistencia mayor que los niveles de aislamiento anteriores.

- **TRANSACTION_SERIALIZABLE**

Se verán todos los registros tal y como estaban antes de comenzar la transacción, no importa las modificaciones que otras transacciones hagan, ni que lo hayamos leído antes o no. Si se añadió algún nuevo registro, tampoco se verá.

Por ejemplo:

```
conexion.setAutoCommit(false);  
conexion.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

7.3 Ejemplo desde NetBeans

Ampliamos nuestro ejemplo anterior para que se usen transacciones. Tal y como hemos creado la clases del package DAO, estas no saben si están dentro de una transacción o no. Es decir será la clase que invoque a las diferentes operaciones de bases de datos quien indicará si están dentro de una transacción o no.

Imaginemos un ejemplo donde intentaremos eliminar todos los clientes cuyo DNI comience por 1, pero introduciremos una excepción y observaremos como no se realiza ningún borrado. Únicamente necesitamos modificar la clase Main (pondremos entre comentarios el código anterior) que crea las conexiones a base de datos y llama a las clases de DAO. Esto es:

```

public static void main(String[] args) throws Exception {
    Connection _con=null;
    Conexion_DB _conexion_DB = new Conexion_DB();
    try {
        _con = _conexion_DB.AbrirConexion();// Abrimos la conexión
        _con.setAutoCommit(false);
        _con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        ClienteDAO _clienteDAO=new ClienteDAO();
        List<Cliente> _clientes=_clienteDAO.findByNumberDNISStart(_con, 1);
        for (int i=0;i<_clientes.size();i++)
        {
            System.out.println("CLIENTE CUYO DNI COMIENZA CON 1");
            _clienteDAO.elimina(_con, _clientes.get(i));
            System.out.println("Cliente cuyo DNI es"+_clientes.get(i).getDNI()+" será eliminado");
        }
        //if (true) throw new Exception("Error para evitar el borrado de los clientes");
        _con.commit();
    } catch (Exception ex) {
        System.out.println("Excepcion->" + ex.getMessage());
        if (_con!=null) _con.rollback();
    } finally
    {
        if (_con!=null) _conexion_DB.CerrarConexion(_con); //Cerramos la conexión
    }
}

```

Una vez lo ejecutemos mediante  veremos por pantalla

Output - Sesión5_1 (run)

```


init:
deps-jar:
Compiling 1 source file to C:\DEMO\Sesion5_1\build\classes
compile:
run:
CLIENTE CUYO DNI COMIENZA CON 1
Cliente cuyo DNI es14 será eliminado
CLIENTE CUYO DNI COMIENZA CON 1
Cliente cuyo DNI es15 será eliminado
Excepcion->Error para evitar el borrado de los clientes
BUILD SUCCESSFUL (total time: 1 second)

```

Y si visualizamos el contenido de la tabla, veremos que no se ha borrado ningún cliente

Si ahora comentáramos la línea que lanza la excepción, esta es

```
//if (true) throw new Exception("Error para evitar el borrado de los clientes");
```

y volvemos a ejecutar mediante  veremos que no ha quedado en la tabla ningún cliente cuyo DNI comience por 1

MySQL Query Browser - Connection: root@localhost:3306 / ClientesDB

FileEditViewQueryScriptToolsWindowMySQL EnterpriseHelp

Go back

Next

Refresh

SELECT * FROM clientesdb.cliente c;

Resultset 1

	DNI	Nombre	Ape1	Ape2	Nick	Passwd	Saldo	Moroso
▶	33	Marcos	Heredia	Buendia	marcs	mar1	1000	0
	8977	Juan	Martinez	NULL	alex	ale1	1000	0
	367809	Guillermo	Toro	Fuentes	guille	gui1	1000	0
	456781	Julia	Sanches	Guarner	juli	jul1	1000	0
	31678901	Ana	Bermejo	Beltan	anaber	ana1	1000	0

8 Pool de Base de Datos

8.1 Definición

Se denomina **connection pool** (agrupamiento de conexiones) al manejo de una colección de conexiones abiertas a una base de datos de manera que puedan ser reutilizadas al realizar múltiples consultas o actualizaciones.

Cada vez que un programa cliente necesita comunicarse con una base de datos, establece una conexión. Estas conexiones, para aplicaciones Web se abren y cierran lo antes posible para ocupar la base de datos el menor tiempo posible.

El problema es que realizar conexiones de Base de Datos es un proceso que implica el uso de recursos y penaliza el sistema. Por esta razón es recomendable reutilizar las conexiones una vez establecidas. La reutilización de estas conexiones establecidas ("latentes") se lleva a cabo colocándolas en un grupo ("pool") para que cualquier programa o recurso del sistema pueda adquirirla, **sin incurrir en las penalidades de abrir una conexión desde una etapa inicial**.

Lo explicado hasta ahora implica que se realiza una conexión a la base de datos en cada petición. Este sería un esquema de **conexión-operación-desconexión**. Esta forma de trabajar es perfectamente válida, pero resulta **ineficiente** ante aplicaciones con gran número de usuarios, ya que se están desperdiciando ciclos de ejecución en cada conexión y desconexión.

Cuando una petición requiere una conexión la solicita al conjunto de conexiones disponibles (desocupadas, "idle") y cuando no la va a usar la devuelve al pool. De esta forma, nos ahorramos el consumo de tiempo de conexión y desconexión.

8.2 DataSource

Cargar el driver y obtener conexiones con **DriverManager.getConnection** requiere conocer:

- el nombre de la clase del driver
- la cadena de conexión. (URL)
- el usuario y password de acceso a la Base de datos

A no ser que leamos los anteriores parámetros de un fichero de configuración, la creación de conexiones no será portable. Precisamente el DataSource permite configurar todos estos datos fuera de la aplicación.

Esencialmente un objeto **DataSource** representa una fuente de datos en una aplicación Java. Encapsula la información de la conexión a una base de datos y del driver JDBC dentro de un objeto simple y estandarizado.

La interfaz **javax.sql.DataSource** es la alternativa recomendada a **DriverManager.getConnection**

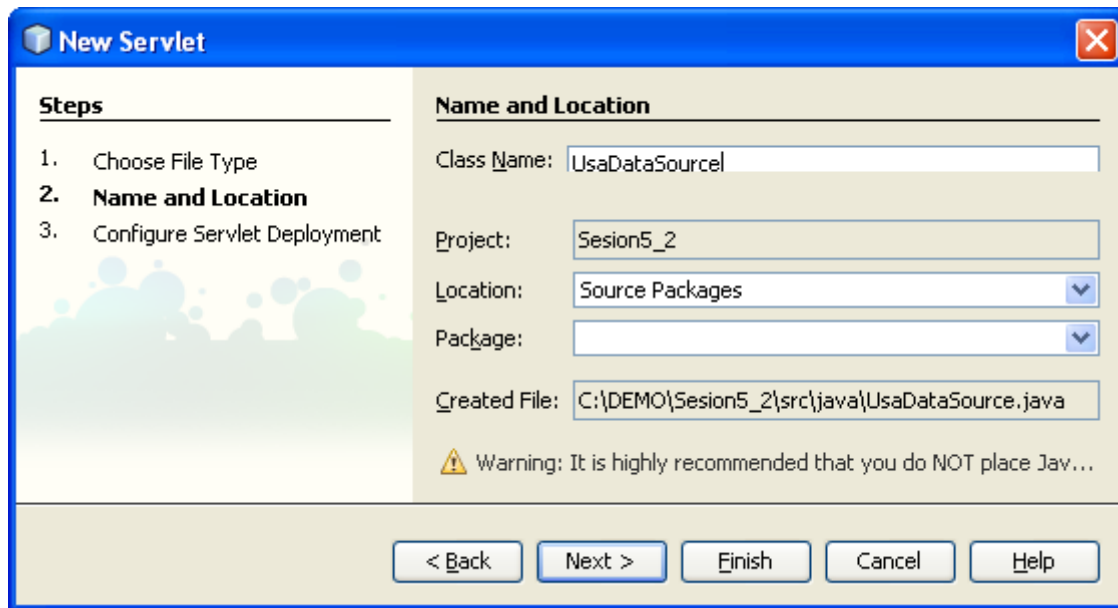
No hay que escribir código para registrar el driver, dado que se registra en el servidor y mediante JNDI (Java Naming and Directory Interface) se obtiene una referencia a un objeto **DataSource**

Este DataSource actúa como una factoría de conexiones, es decir, permite bien crear nuevas conexiones, bien acceder al grupo de conexiones del pool.

8.3 Ejemplo desde NetBeans

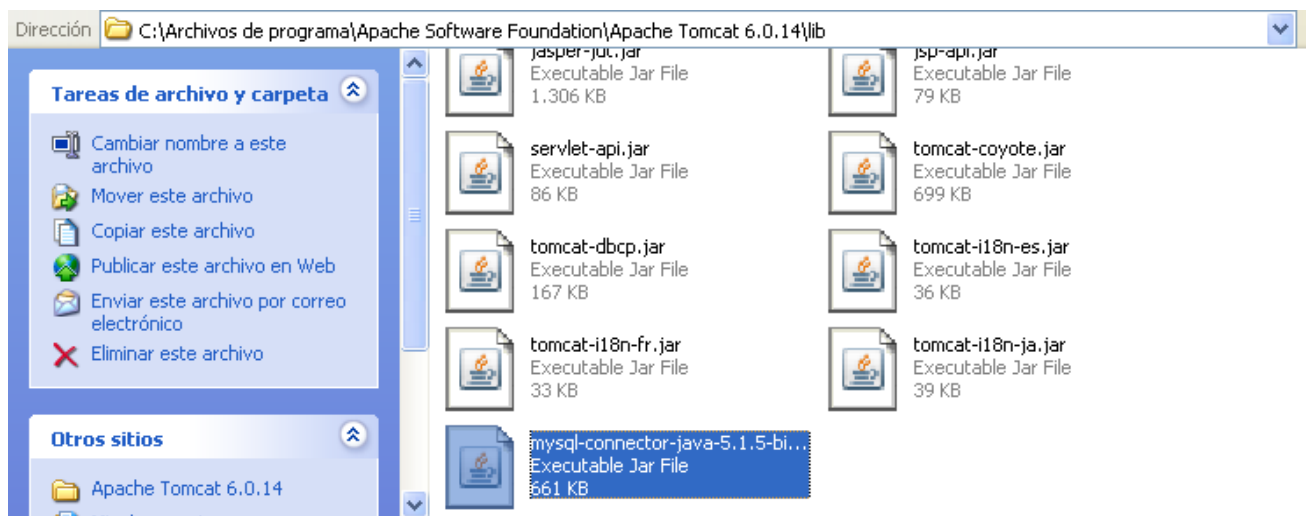
La mayor parte de servidores de aplicaciones (Tomcat, WebSphere, etc.) incluyen clases que implementan un pool de conexiones. En el caso de MySQL y Tomcat, este pool es implementado mediante las clases DataBase Common Pooling (DBCP) de Tomcat.

Para poder usar el Datasource y el Pool de conexiones crearemos un Servlet llamado UsaDataSource, dentro de un **nuevo proyecto de tipo Web** llamado **Sesion5_2**.

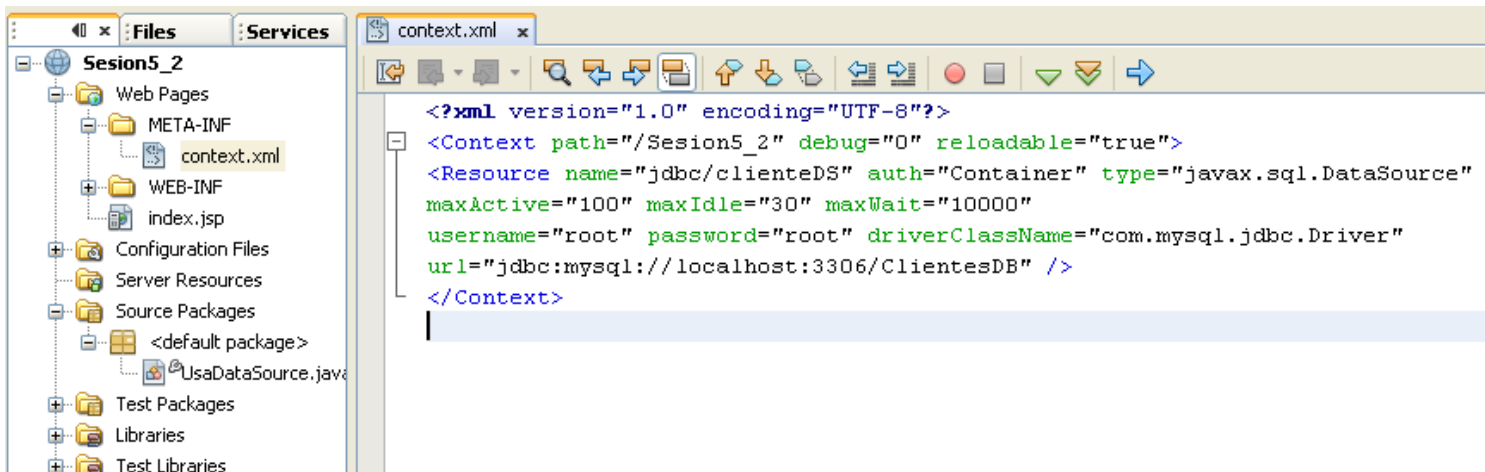


Una vez creado los pasos a seguir son:

1- Hemos de **añadir a Tomcat** el fichero jar con nuestro **driver de mysql**. Para ello lo añadiremos en la carpeta /lib de la instalación de Tomcat que viene incrustado con NetBeans. Esto permite a Tomcat cargar el driver para crear las conexiones. Esto es:



2- Editamos el **fichero context.xml** que se encuentra dentro de Web Pages/META-INF y añadiremos la siguiente información:



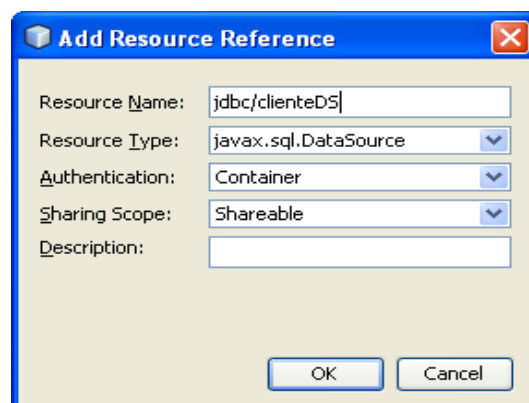
donde el significado de los parámetros es:

Parámetro	Descripción
driverClassName	Nombre completo del driver
url	Cadena de conexión JDBC a la base de datos
username	Nombre del usuario MySQL
password	Contraseña del usuario MySQL
maxActive	Número máximo de conexiones activas que pueden existir al mismo tiempo
maxIdle	Número máximo de conexiones que pueden estar inactivas al mismo tiempo
maxWait	Número máximo de milisegundos para esperar a una conexión disponible.

3- Accedemos al **fichero web.xml** que se encuentra dentro de Web Pages/WEB-INF y añadiremos un Resource a través del botón Add de la pestaña References



en el que indicaremos el nombre del DataSource



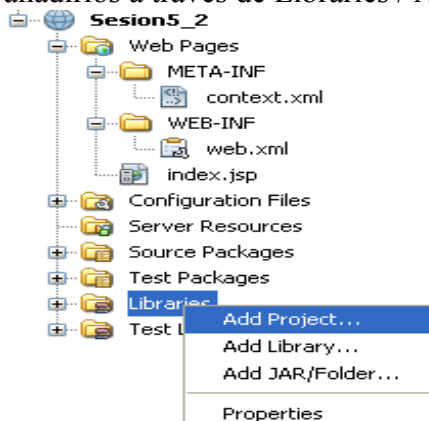
4- Una vez hemos definido el contexto del proyecto a través de web.xml y context.xml definiremos un nuevo método llamado AbrirConexionDS en la clase Conexion_DB del proyecto Sesión5_1 con el código para abrir una conexión usando el DataSource definido en el contexto. Esto es:

```
public Connection AbrirConexionDS() throws Exception
{
    Connection con=null; // instancia una conexión
    try {
        Context ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/clienteDS");
        con = ds.getConnection();
        return con;
    } catch(Exception e){//SQLException y ClassNotFoundException
        e.printStackTrace();
        throw new Exception("Ha sido imposible establecer la conexión desde DataSource"+e.getMessage());
    }
}
```

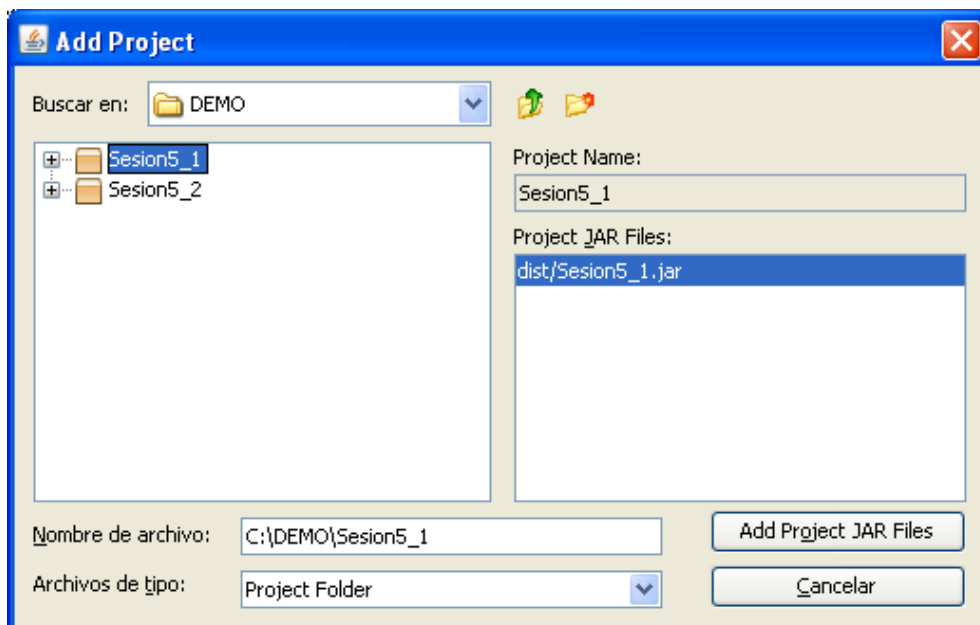
Además de añadir los imports necesarios para que compile. Estos son:

```
import javax.sql.DataSource;
import javax.naming.Context;
import javax.naming.InitialContext;
```

Para que nuestro Servlet pueda usar las clases del proyecto Sesión5_1, hemos de añadirlos a través de Libraries / AddProject



donde seleccionaremos el proyecto Sesión5_1 de nuestra carpeta DEMO



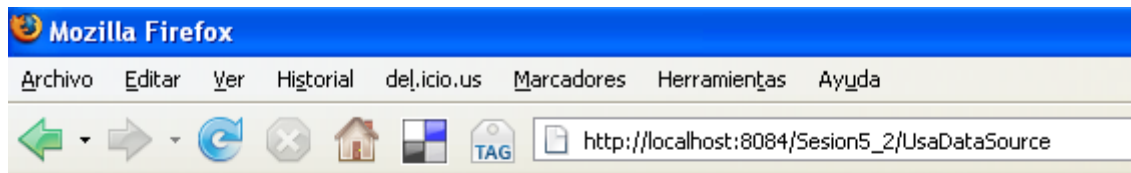
De este modo, ya podemos usar las clases definidas a lo largo de los ejemplos desde nuestro Servlet. Por ejemplo, este sería el código para listar los clientes cuyo DNI comienza por 3 por pantalla:

```
import DAO.ClienteDAO;
import DAO.Conexion_DB;
import Entidad.Cliente;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.Connection;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;

public class UsaDataSource extends HttpServlet {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            Conexion_DB conexionDB=new Conexion_DB();
            Connection _con=conexionDB.AbrirConexionDS();
            ClienteDAO _clienteDAO=new ClienteDAO();
            List<Cliente> _clientes1=_clienteDAO.findByNumberDNISStart(_con, 3);
            out.println("CLIENTES CUYO DNI COMIENZA CON 3");
            out.println("<HR><BR>");
            Cliente _cliente1=null;
            for (int i=0;i<_clientes1.size();i++)
            {
                _cliente1=_clientes1.get(i);
                out.println("Cliente DNI->"+_cliente1.getDNI()+"<BR>");
                out.println("Cliente nombre->"+_cliente1.getNombre()+"<BR>");
                out.println("Cliente Primer apellido->"+_cliente1.getApellido1()+"<BR>");
                out.println("Cliente Segundo apellido->"+_cliente1.getApellido2()+"<BR>");
                out.println("Cliente Nick->"+_cliente1.getNick()+"<BR>");
                out.println("Cliente Password->"+_cliente1.getPassword()+"<BR>");
                out.println("Cliente Saldo->"+_cliente1.getSaldo()+"<BR>");
                out.println("Cliente Moroso->"+_cliente1.isMoroso()+"<BR>");
                out.println("<HR><BR>");
            }
            conexionDB.CerrarConexion(_con); //Cerramos la conexión */
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            out.close();
        }
    }
}
```

HttpServlet methods. Click on the + sign on the left to edit the code.

Y lo que se mostraría al ejecutarlo mediante  y acceder a la URL correcta:



CLIENTES CUYO DNI COMIENZA CON 3

Cliente DNI->33
Cliente nombre->Marcos
Cliente Primer apellido->Heredia
Cliente Segundo apellido->Buendia
Cliente Nick->marcs
Cliente Password->mar1
Cliente Saldo->1000.0
Cliente Moroso->>false

Cliente DNI->367809
Cliente nombre->Guillermo
Cliente Primer apellido->Toro
Cliente Segundo apellido->Fuentes
Cliente Nick->guille
Cliente Password->gui1
Cliente Saldo->1000.0
Cliente Moroso->>false

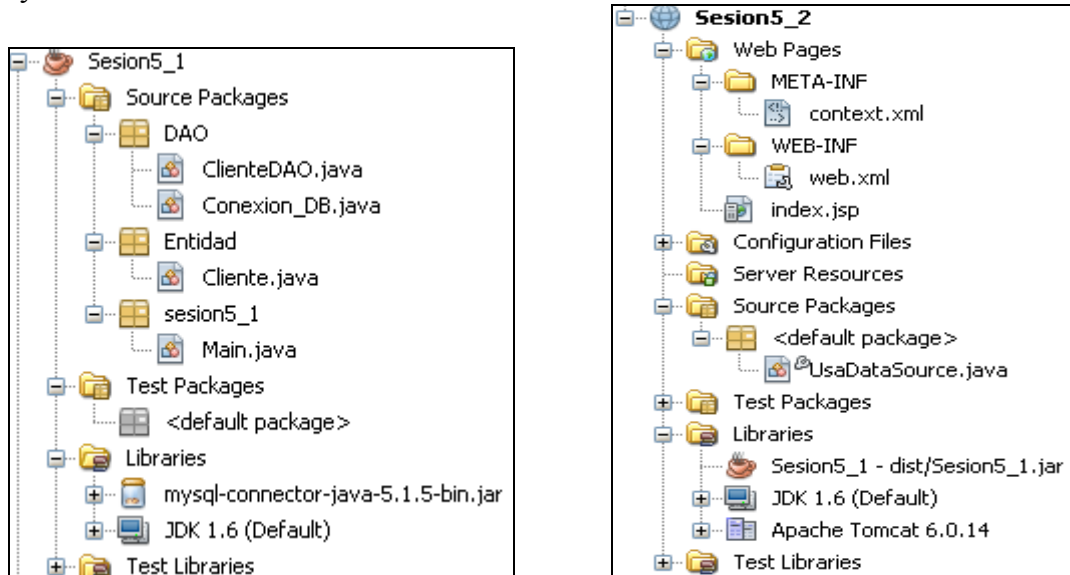
Cliente DNI->31678901
Cliente nombre->Ana
Cliente Primer apellido->Bermejo
Cliente Segundo apellido->Beltan
Cliente Nick->anaber
Cliente Password->ana1
Cliente Saldo->1000.0
Cliente Moroso->>false

9 Ejercicios

Realiza los siguientes ejercicios y envíalos en un comprimido con tu nombre y apellidos al profesor. (Ej.: vperezcabello_sesion5.zip).

9.1 Ejercicio 1 (Entregado por el profesor)

Realiza las aplicaciones ejemplos Sesión5_1 y Sesión5_2 explicadas a lo largo de la sesión. Para que te sirva de guía estos son los elementos que deben tener estos proyectos:



NOTA: Este ejercicio es entregado por el profesor, de modo que el alumno tenga disponible el código indicado en la sesión.

Realiza los siguientes ejercicios ampliando Sesión5_1 y Sesión5_2

9.2 Ejercicio 2 (3 puntos)

Crea las siguientes clases, atributos y métodos en

Package Entidad

- **Factura**->para almacenar los datos de las facturas
- **Articulo**->para almacenar los datos de los artículos

Package DAO

- **FacturaDAO**
 - o Método creaFactura->que genera nuevas facturas

```
public Factura creaFactura(Connection con, Factura factura, Cliente cliente) throws Exception
```

- o Método addArticulo->que añade artículos a la factura

```
public void addArticulo(Connection con, Factura factura, Articulo articulo, int numero) throws Exception
```

- **ArticuloDAO**

- o Método updateStock->que modifica el stock de ese articulo

```
public Articulo updateStock(Connection con, Articulo articulo) throws Exception
```

- o Método findById->que trae los datos de un articulo de la Base de Datos

```
public Articulo findById(Connection con, Articulo articulo) throws Exception
```

Además se necesita añadir un método en **ClienteDAO**:

- o Método `updateSaldo`->que modifica el saldo restante de ese cliente

```
public void updateSaldo(Connection con, Cliente cliente) throws Exception
```

Crea una nueva clase llamada **Main2** con un método llamado `pruebaEjercicio2`

```
private static void pruebaEjercicio2() throws Exception
```

que será llamado desde el método `main` que añadiremos a esa clase

```
public static void main(String[] args) throws Exception {
    pruebaEjercicio2();
}
```

y que tendrá como **parte de su definición** (a falta de abrir y cerrar conexión, y de controlar las transacciones) el siguiente código, el cual permite probar que funciona el añadir una compra al cliente con DNI 367809 de dos artículos con `IdArticulo` a 1:

```
ClienteDAO clienteDAO=new ClienteDAO();
Cliente _cliente=new Cliente();//Crea un objeto Cliente con el DNI que se solicita
_cliente.setDNI(367809);
_cliente=clienteDAO.findByDNI(_con, _cliente);

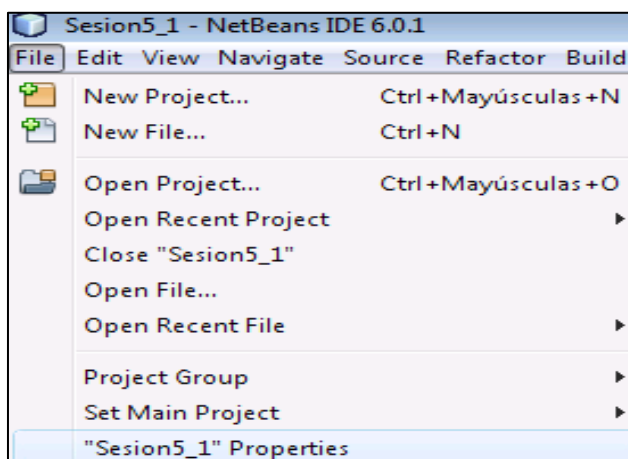
ArticuloDAO articuloDAO=new ArticuloDAO();//Obtiene los datos del Articulo
Articulo articulo=new Articulo();
articulo.setId(1);
articulo=articuloDAO.findById(_con, articulo);

FacturaDAO facturaDAO=new FacturaDAO();
Factura factura=new Factura();

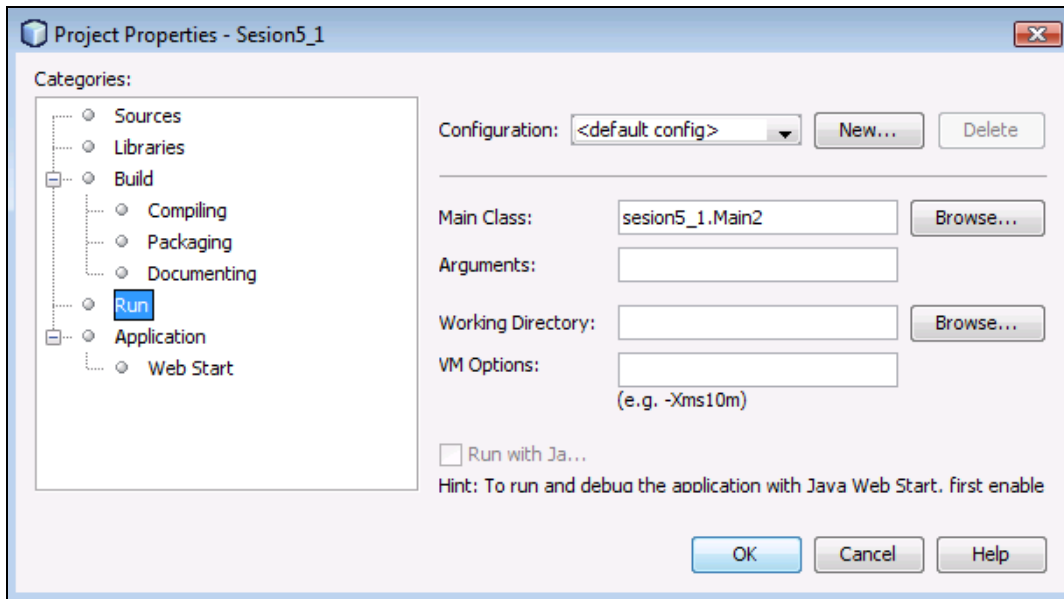
Date ahora = new Date();//Crea la fecha del dia actual
SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
int fecha=Integer.parseInt(sdf.format(ahora));

factura.setFecha(fecha);
factura=facturaDAO.creaFactura(_con, factura,_cliente);//Crea una nueva factura
facturaDAO.addArticulo(_con, factura, articulo, 2);//Inserta la linea de factura
articulo.setStock(articulo.getStock()-2);
articuloDAO.updateStock(_con, articulo);//Actualiza el stock de articulo
_cliente.setSaldo(_cliente.getSaldo()-(articulo.getPrecio()*2));
clienteDAO.updateSaldo(_con, _cliente);
```

Para que el proyecto use esa clase `Main2` como principal hemos de seleccionarla a través del menú `File/Sesion5_1 Properties`



Seleccionaremos el apartado Run y mediante pulsar Browse seleccionaremos la clase creada como **Main class**



Esto debe generar en la base de datos:

- una nueva fila en la tabla Factura
- una nueva fila en la tabla Articulo_Factura
- una modificación del stock de la fila del IdArticulo=1 en la tabla Articulo
- una modificación del saldo del cliente

Recuerda usar transacciones para que no queden datos inconsistentes en las tablas

Nota: En la tabla factura se ha codificado la fecha como un entero. Esto es un truco que se usa en programación para evitar así incompatibilidades ante migraciones de Bases de Datos. Es decir, una fecha como **20080315** se referirá al **15 de Marzo del 2008**

La gran ventaja de esta codificación es que permite comparar fechas de manera directa, viendo que número es mayor, y solo se ha de modificar su aspecto de cara a presentarlo al usuario.

Nota2: En el método creaFactura necesitareis obtener el id que se ha autogenerado en la tabla. Esto se hace mediante el siguiente código:

```
PreparedStatement stmt;
stmt = con.prepareStatement("INSERT INTO .....",Statement.RETURN_GENERATED_KEYS);
.....
stmt.executeUpdate(); //ejecuta el INSERT
ResultSet rs = stmt.getGeneratedKeys (); //recupera un resultset cuyo
rs.next(); // contenido es la Primary Key
int key = rs.getInt(1); //autogenerada en la tabla
factura.setId(key);
```

9.3 Ejercicio 3 (3 puntos)

Crea en ClienteDAO un método que devuelva un cliente con sus facturas a partir del DNI. Esto es:

```
public Cliente getClienteFacturas(Connection con, Cliente cliente) throws Exception
```

Para esto deberá añadir al JavaBean Cliente un atributo que será un listado (java.util.List) de objetos Factura (del package Entidad)

```
private List<Factura> Facturas=new ArrayList();
```

además de sus métodos **accesores**

Para probarlo, actuaremos de forma similar al anterior ejercicio creando la clase Main3 y definiendo pruebaEjercicio3 con el siguiente código

```
private static void pruebaEjercicio3() throws Exception {
    Connection _con=null;
    Conexion_DB _conexion_DB = new Conexion_DB();
    try {
        _con = _conexion_DB.AbrirConexion();// Abrimos la conexión
        ClienteDAO clienteDAO=new ClienteDAO();
        Cliente _cliente=new Cliente();//Crea un objeto Cliente con el DNI que se solicita
        _cliente.setDNI(456781);
        _cliente=clienteDAO.getClienteFacturas(_con, _cliente);//Llama a este método para recoger las facturas
        for (int i=0;i<_cliente.getFacturas().size();i++)
        {
            Factura factura=_cliente.getFacturas().get(i);
            System.out.println("Factura encontrada con Id="+factura.getId()+" y Fecha "+factura.getFecha());
        }
    } catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("Excepcion->" + ex.getMessage());
    } finally
    {
        if (_con!=null) _conexion_DB.CerrarConexion(_con); //Cerramos la conexión
    }
}
```

que será llamado desde el método main de esa clase Main3

```
public static void main(String[] args) throws Exception {
    pruebaEjercicio3();
}
```

y mostrará en consola

```
Output - Sesión5_1 (run)
compile:
run:
Factura encontrada con Id=2 y Fecha 20080311
Factura encontrada con Id=3 y Fecha 20080324
BUILD SUCCESSFUL (total time: 1 second)
```

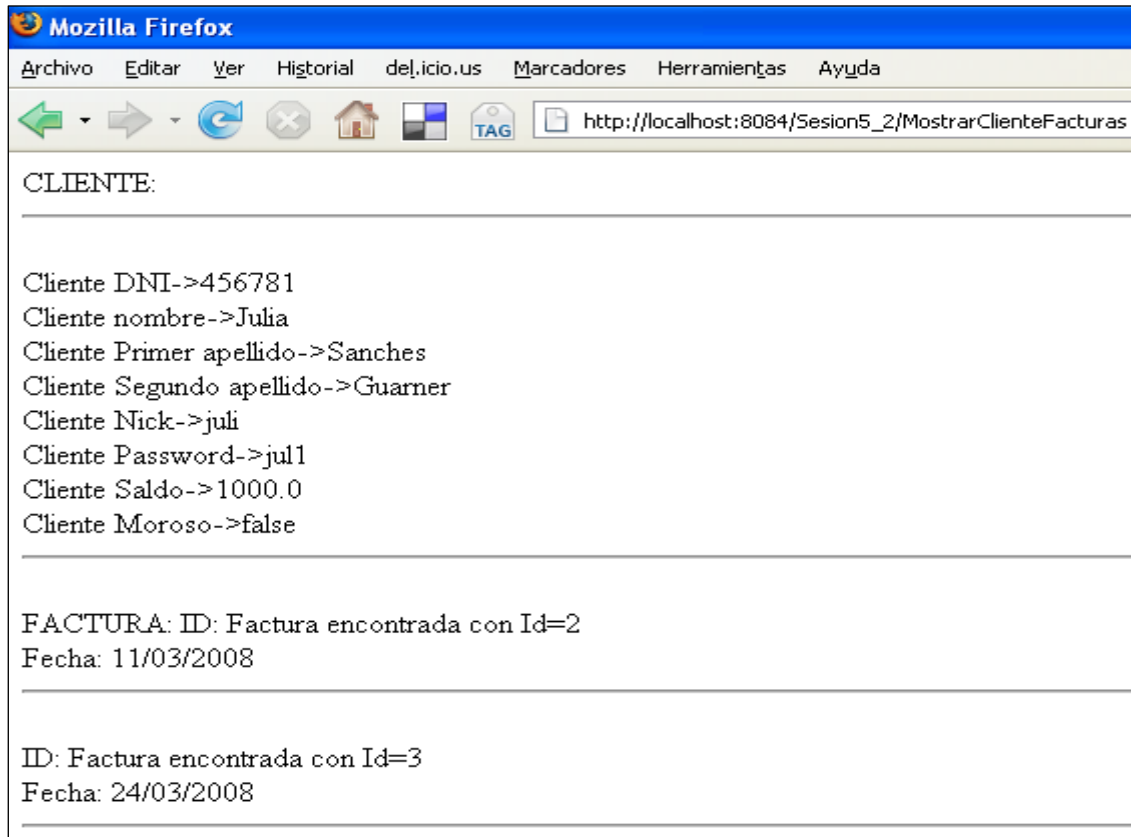
Nota: En este ejercicio se puede observar como las clases del package Entidad se relacionan entre sí. Otra posible relación sería entre Factura y Artículo, dado que una Factura tiene un listado de artículos.

9.4 Ejercicio 4 (3 puntos)

Cree un Servlet en **Sesion5_2** llamado **MostrarClienteFacturas** que muestre por pantalla los datos del cliente con DNI= 456781 y sus facturas.

Para ello usará el método **getClientefacturas** del Ejercicio3 y el método **findByDNI**. Ambos métodos son de ClienteDAO

La salida debe ser lo más parecido a



Nota: Para facilitar la tarea, se indica el código necesario para formatear la fecha

```
String year= (""+factura.getFecha()).substring(0, 4);
String mes= (""+factura.getFecha()).substring(4, 6);
String dia= (""+factura.getFecha()).substring(6, 8);
out.println(dia+"/"+mes+"/"+year);
```

9.5 Ejercicio 5 (1 punto)

Modifique el ejercicio 4 para que use un DataSource llamado **clientesDSAlt** que deberá **añadir al ya existente**.

Para esto añadirá un método en la clase **Conexion_DB** llamado

```
public Connection AbrirConexionDSAlt() throws Exception
```

Nota: Dentro del fichero xml, solo habrá un elemento **Context**, con tantos elementos **Resource** como deseemos