Capítulo 8

Bases de datos

8.1. Resultados de aprendizaje

8. Utiliza Bases de Datos Orientadas a Objetos, analizando sus características y aplicando técnicas para mantener la persistencia de la información.

a) Se han identificado las características de las Bases de Datos Orientadas a Objetos.

b) Se ha analizado su aplicación en el desarrollo de aplicaciones mediante lenguajes orientados a objetos.

c) Se han instalado sistemas gestores de bases de datos orientados a objetos.

d) Se han clasificado y analizado los distintos métodos soportados por los sistemas gestores para la gestión de la información almacenada.

e) Se han creado bases de datos y las estructuras necesarias para el almacenamiento de objetos.

f) Se han programado aplicaciones que almacenen objetos en las bases de datos creadas.

g) Se han realizado programas para recuperar, actualizar y eliminar objetos de las bases de datos.

h) Se han realizado programas para almacenar y gestionar tipos de datos estructurados, compuestos y relacionados.

9. Gestiona información almacenada en bases de datos relacionales manteniendo la integridad y consistencia de los datos.

a) Se han identificado las características y métodos de acceso a sistemas gestores de bases de datos relacionales.

b) Se han programado conexiones con bases de datos.

c) Se ha escrito código para almacenar información en bases de datos.

d) Se han creado programas para recuperar y mostrar información almacenada en bases de datos.

e) Se han efectuado borrados y modificaciones sobre la información almacenada.

f) Se han creado aplicaciones que ejecuten consultas sobre bases de datos.

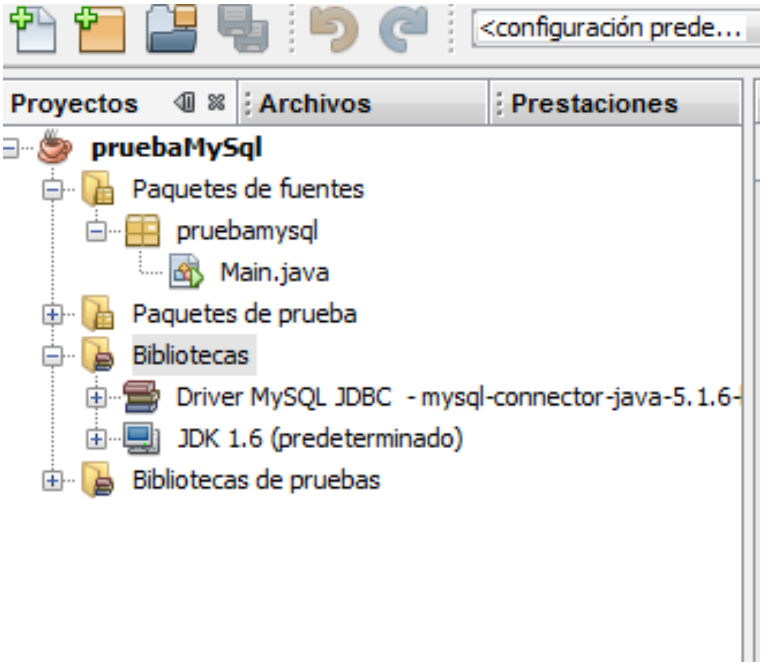
g) Se han creado aplicaciones para posibilitar la gestión de información presente en bases de datos relacionales.

8.2. Bases de datos relacionales

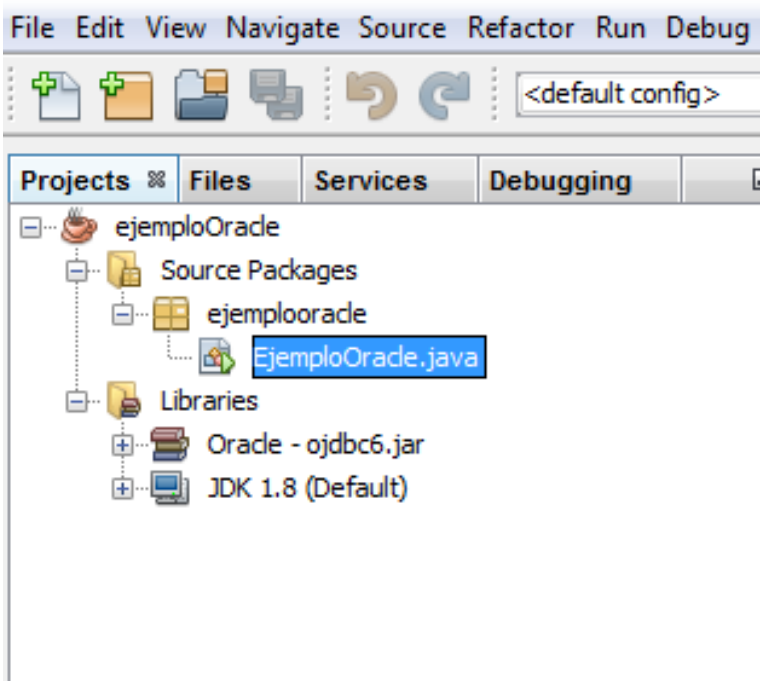
Para poder trabajar con una base de datos relacional en nuestro proyecto, la primera operación que debemos realizar es la de añadir al mismo la biblioteca o librería que permite la comunicación entre Java y el sistema gestor de la base de datos.

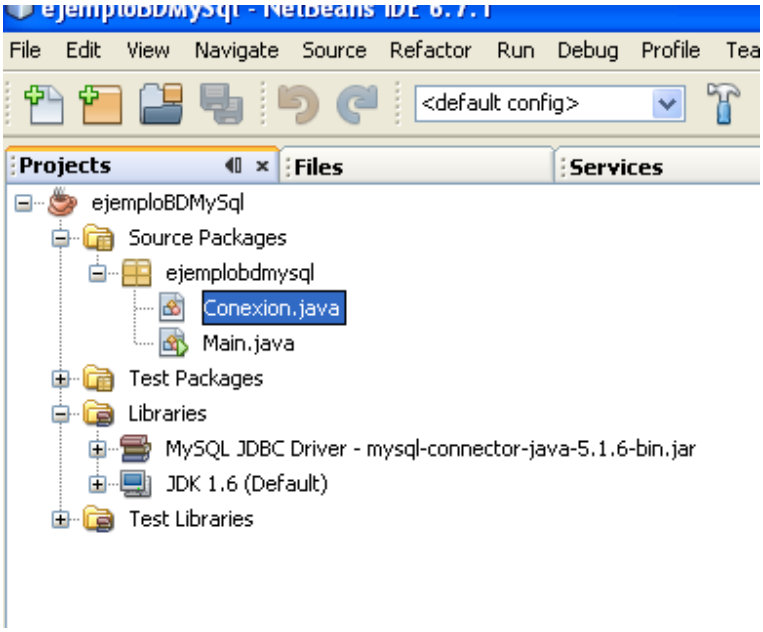
En informática, una biblioteca (del inglés library) es un conjunto de implementaciones funcionales, codificadas en un lenguaje de programación, que ofrece una interfaz bien definida para la funcionalidad que se invoca.

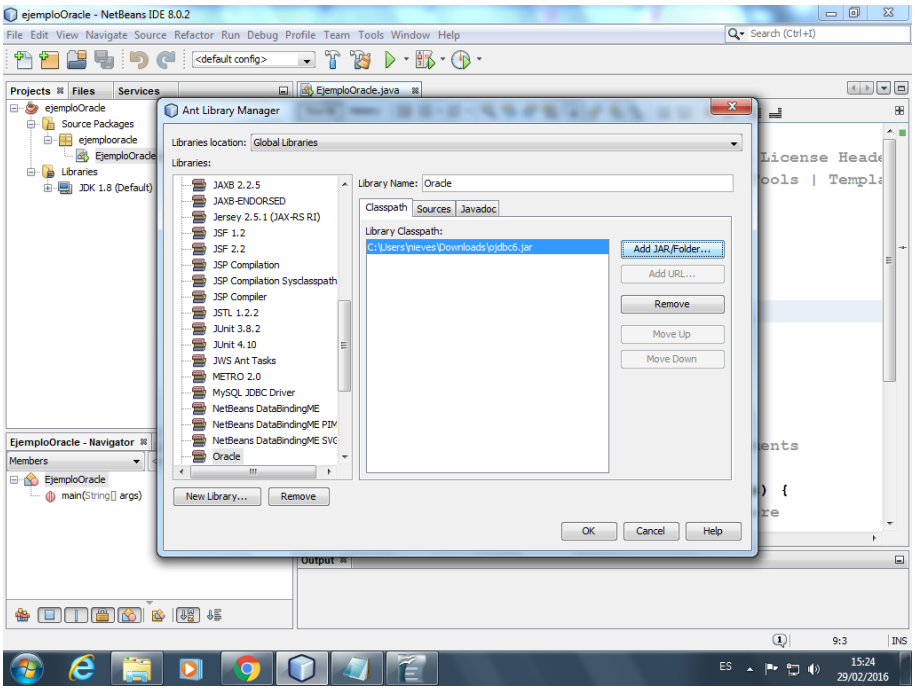
La librería puede encontrarse ya en el entorno de Netbeans o tal vez tengamos que crearla. En el caso de Oracle habrá que crearla a partir del .jar descargado de la plataforma oficial (http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html). En el caso de mysql ya existe.

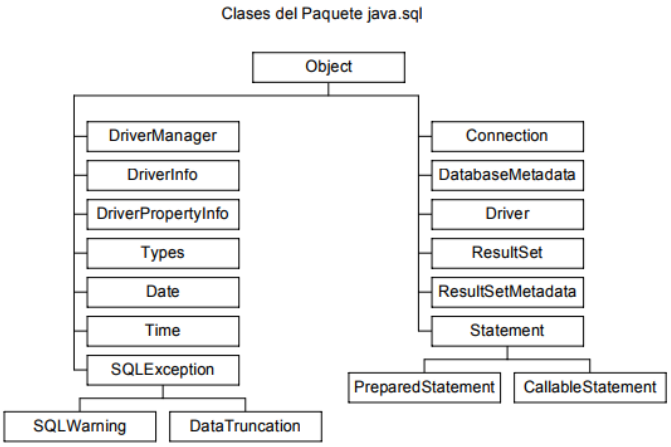


A continuación, en el **paquete java.sql**, encontraremos todas las clases necesarias para trabajar con el tipo de base de datos relacional elegido.


File Edit View Navigate Source Refactor Run Debug
<default config>
Projects
Files
Services
Debugging
ejemploOracle
Source Packages
ejemplooracle
EjemploOracle.java
Libraries
Oracle - ojdbc6.jar
JDK 1.8 (Default)


File Edit View Navigate Source Refactor Run Debug Profile
<default config>
Projects
Files
Services
ejemploBDMySql
Source Packages
ejemplobdmysql
Conexion.java
Main.java
Test Packages
Libraries
MySQL JDBC Driver - mysql-connector-java-5.1.6-bin.jar
JDK 1.6 (Default)
Test Libraries


ejemploOracle - NetBeans IDE 8.0.2
Ant Library Manager
Libraries location: Global Libraries
Libraries:
JAXB 2.2.5
JAXB-ENDORSED
Jersey 2.5.1 (JAX-RS RI)
JSF 1.2
JSF 2.2
JSP Compilation
JSP Compilation Sysclasspath
JSP Compiler
JSTL 1.2.2
JUnit 3.8.2
JUnit 4.10
JWS Ant Tasks
METRO 2.0
MySQL JDBC Driver
NetBeans DataBindingME
Oracle
Library Name: Oracle
Classpath
Sources
Javadoc
Library Classpath:
C:\Users\nieves\Downloads\ojdbc6.jar
Add JAR/Folder...
Add URL...
Remove
Move Up
Move Down
New Library...
Remove
OK
Cancel
Help


Clases del Paquete java.sql
Object
DriverManager
DriverInfo
DriverPropertyInfo
Types
Date
Time
SQLException
SQLWarning
DataTruncation
Connection
DatabaseMetadata
Driver
ResultSet
ResultSetMetadata
Statement
PreparedStatement
CallableStatement

8.3. Ejemplo

```
1  package ejemplooracle;
2
3  import java.sql.*;
4  public class EjemploOracle {
5
6
7      public static void main(String[] args) {
8
9          try{
10
11             // Abrir conexión con la bd
12             Class.forName("oracle.jdbc.OracleDriver");
13
14             String login="nieves";
15             String password= "nieves";
16             String url = "jdbc:oracle:thin:@SrvOracle:1521:orcl";
17             Connection con =
                   DriverManager.getConnection(url,login,password);
18
19
20             // insert
21             Statement sentencia = con.createStatement();
22             sentencia.executeUpdate("insert into empleados
                   values('13','Pepe')");
23             sentencia.executeUpdate("insert into empleados
                   values('14','Juan')");
24
25
26             // select
27             ResultSet resultado = sentencia.executeQuery("select *
                   from empleados");
28             String mensaje="";
29             while(resultado.next())
30             {
31                 mensaje += resultado.getString("dni");
32                 mensaje += resultado.getString("nombre");
33                 mensaje +="\n";
34             }
35             javax.swing.JOptionPane.showMessageDialog(null, mensaje);
36             con.close();
37         }
38         catch(Exception e){
39             javax.swing.JOptionPane.showMessageDialog(null,"Problemas
                   "+e.getMessage());
40
41         }
42     }
43
44 }
```

8.4. Pasos a seguir para trabajar con una base de datos relacional

8.4.1. Cargar el driver del tipo de base de datos a la que queremos acceder

```
1  try{
2          //Registramos el driver
3          Class.forName("oracle.jdbc.OracleDriver");
4     }
5     catch(ClassNotFoundException e)
6     {
7
8     }
```

Si el driver no está disponible, la ejecución del método forName lanza una excepción de tipo ClassNotFound

Para una base de datos Microsoft Access el driver es **sun.jdbc.odbc.JdbcOdbcDriver.**

Para una base de datos MySql el driver es **com.mysql.jdbc.Driver**

También se puede de la seguiente manera:

```
1  DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
```

8.4.2. Definir la dirección (url) de la base de datos

Una vez cargado el driver ya se sabe el tipo de base de datos, ahora hay que solicitar una conexión a la base de datos concreta, para ello se usa la clase DriverManager y una URL.

```
1  String url = "jdbc:oracle:thin:@SrvOracle:1521:orcl";
2
3  // Nombre de host: SrvOracle. Es el nombre del servidor o maquina
       donde está alojada nuestra Base de Datos.
4
5  // Puerto: 1521. Identifica el puerto TCP utilizado en la
       comunicación. El 1521 es el puerto por defecto en Oracle.
6
7  // SID: orcl. Representa le nombre de la instancia de la base de
       datos para la que estemos preparando la conexión.
8
9
10
11 String url="jdbc:mysql://localhost:3307/"+"nombreBD";
```

8.4.3. Establecer la conexión

Utilizando los drivers disponibles la clase java.sql.DriverManager se establece la conexión con la base de datos, es decir, se abre la vía de comunicación. Normalmente, el único método que un programador deberá utilizar de la clase DriverManager es el método getConnection().

```
1     String login="nieves";
2     String password= "nieves";
3     String url = "jdbc:oracle:thin:@SrvOracle:1521:orcl";
4
5     Connection con = DriverManager.getConnection(url,login,password);
6
7
8     Connection con = DriverManager.getConnection(url,"root","usbw");
9
10
11    Connection con =
          DriverManager.getConnection("jdbc:oracle:thin:@server224:1521:orcl","proyecto","proyec
```

8.4.4. Ejecutar sentencias SQL

Existen tres clases para representar una sentencia SQL: **Statement**, **PreparedStatement** y **CallableStatement**.

PreparedStatement hereda del interfaz Statement y CallableStatement del interfaz PreparedStatement.

Cada tipo de sentencia está especializada para enviar un tipo de sentencia SQL. Un objeto **Statement** es utilizado para ejecutar una sentencia SQL simple, es decir, sin parámetros; un objeto **PreparedStatement** es utilizado para ejecutar sentencias SQL precompiladas con o sin parámetros de entrada; y un objeto **CallableStatement** es utilizado para ejecutar una llamada a un procedimiento almacenado de una base de datos, que puede tener parámetros de entrada, de salida y entrada/salida.

Posteriormente, hay un método executeQuery o executeUpdate que nos permitirá ejecutar la consulta (select) o la actualización de datos (insert, update, delete).

El resultado de una consulta es un objeto de la clase ResultSet.

```
1
2  Statement sentencia=con.createStatement();
3
4  ResultSet resultado = sentencia.executeQuery("select * from
       tAlumnos;");
```

Más ejemplos de instrucciones SQL:

```
try{

        //Inserción de valores
        Statement sentencia=con.createStatement();
        sentencia.executeUpdate("INSERT INTO datosPersonales
            VALUES(4,'Pepe',40,'c/Cuchilleria')");

        System.out.println("Fila insertada");
        con.close();
    }
    catch(Exception e)
    {
          System.out.println("Se ha producido el siguiente error" +
              e.getMessage());
    }
```

```
try{
         //Actualización de valores
        Statement sentencia=con.createStatement();
        sentencia.executeUpdate("UPDATE datosPersonales SET EDAD =
            41 WHERE Id = 4");

        con.close();
    }
    catch(Exception e)
    {
         System.out.println("Se ha producido el siguiente error" +
              e.getMessage());
    }
```

```
try{
         //Eliminación de valores
        Statement sentencia=con.createStatement();
        sentencia.executeUpdate("DELETE FROM datosPersonales WHERE
            Id = 4");

        con.close();
    }
    catch(Exception e)
    {
        System.out.println("Se ha producido el siguiente error" +
            e.getMessage());
    }
```

```
try{
         //Sentencias que manipulan la estructura de la base de
             datos

        Statement sentencia=con.createStatement();
        sentencia.executeUpdate("CREATE TABLE CAFES(NOMBRE
            STRING,CANTIDAD INTEGER,PRECIO FLOAT,TOTAL INTEGER)");

      //sentencia .executeUpdate("DROP TABLE CAFES CASCADE");
      //sentencia.executeUpdate("ALTER TABLE CAFES ADD Referencia
          STRING");
```

```
9         //sentencia.execute("ALTER TABLE CAFES DROP Referencia
              CASCADE");
10
11           con.close();
12           } catch(Exception e){
13               System.out.println("Se ha producido el siguiente error"
                     + e.getMessage());
14           }
```

La interfaz **PreparedStatement** extiende la interfaz Statement. Se utiliza para enviar sentencias sql precompiladas con uno ó mas parámetros. Por ejemplo , SELECT * FROM datosPersonales WHERE Nombre = a.getNombre();

Para trabajar con este tipo de sentencias, en primer lugar se crea un objeto de tipo PreparedStatement especificando la plantilla que indica los lugares donde irán los parámetros (?).

```
1 String plantilla="SELECT * FROM datosPersonales WHERE Nombre = ?"
```

Se precompila la sentencia:

```
1   PreparedStatement ps=conexión.prepareStatement(plantilla);
```

Los parámetros se especifican utilizando los métodos setXXX (,) indicando el número de parámetro y el dato a insertar en la sentencia.

```
1 ps.setString(1, "Elena Sanchez");
```

Hay métodos setXXX(,), para todos los tipos de datos.

- setString(int posicion,String x)
- setBoolean(int posicion,boolean x)
- setInt(int posicion,int x)

La sentencia SQL y los parámetros se envían a la base de datos cuando se llama al método execute que corresponda (executeQuery o executeUpdate)

ResultSet rs=ps.executeQuery();

Este tipo de operaciones también se puede hacer utilizando la clase Statement.

```
1  public void insertData(String table_name, String ID, String name,
       String lastname, String age, String gender) {
2          try {
3              String Query = "INSERT INTO " + table_name + " VALUES("
4                      + "\"" + ID + "\", "
5                      + "\"" + name + "\", "
6                      + "\"" + lastname + "\", "
7                      + "\"" + age + "\", "
```

```
8                       + "\"" + gender + "\")";
9               Statement st = Conexion.createStatement();
10              st.executeUpdate(Query);
11              JOptionPane.showMessageDialog(null, "Datos almacenados
                    de forma exitosa");
12          } catch (SQLException ex) {
13              JOptionPane.showMessageDialog(null, "Error en el
                    almacenamiento de datos");
14          }
15      }
```

8.4.5. Procesar los resultados de una consulta

En un objeto de tipo **ResultSet** se encuentran los resultados de la ejecución de una consulta SQL. Contiene las filas que satisfacen las condiciones de la consulta 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 método 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 a abajo según se va desplazando el cursor con las sucesivas llamadas al método next(). Un cursor es válido hasta que el objeto ResultSet o su objeto padre Statement es cerrado.

```
1
2       ResultSet resultado = sentencia.executeQuery("select * from
            tAlumnos;");
3       resultado.next();
4       System.out.println(resultado.getInt("codigo"));
5       System.out.println(resultado.getString("nombre"));
6       resultado.next();
7       System.out.println(resultado.getInt("codigo"));
8       System.out.println(resultado.getString("nombre"));
9       conn.close();
```

```
1           System.out.println("NOMBRE");
2           int col=resultadoSelect.findColumn("nombre");
3           boolean seguir=resultadoSelect.next();
4
```

```
5          //Mientras queden registros .
6          while (seguir){
7            System.out.println(resultadoSelect.getString(col));
8            seguir=resultadoSelect.next();
9          }
```

```
1
2       ResultSet resultado = sentencia.executeQuery("select * from
            tAlumnos;");
3       resultado.next();
4       System.out.println(resultado.getInt("codigo"));
5       System.out.println(resultado.getString("nombre"));
6       resultado.next();
7       System.out.println(resultado.getInt("codigo"));
8       System.out.println(resultado.getString("nombre"));
9       conn.close();
```

8.4.6. Cerrar la conexión

El último paso consiste en la liberación de los recursos utilizados.

```
1    //Liberamos recursos
2
3    conexion.close();
```

8.4.7. Callable Statement

La interfaz CallableStatements nos permite ejecutar procedimientos almacenados e interactuar con el resultado de una manera rápida y sencilla posicionandose como una buena opción a la hora de trabajar con procedimientos almacenados

```
1  CREATE PROCEDURE demoSp(IN inputParam VARCHAR(255), INOUT inOutParam INT)
2  BEGIN
3      DECLARE z INT;
4      SET z = inOutParam + 1;
5      SET inOutParam = z;
6
7      SELECT inputParam;
8
9      SELECT CONCAT('zyxw', inputParam);
10 END//
```

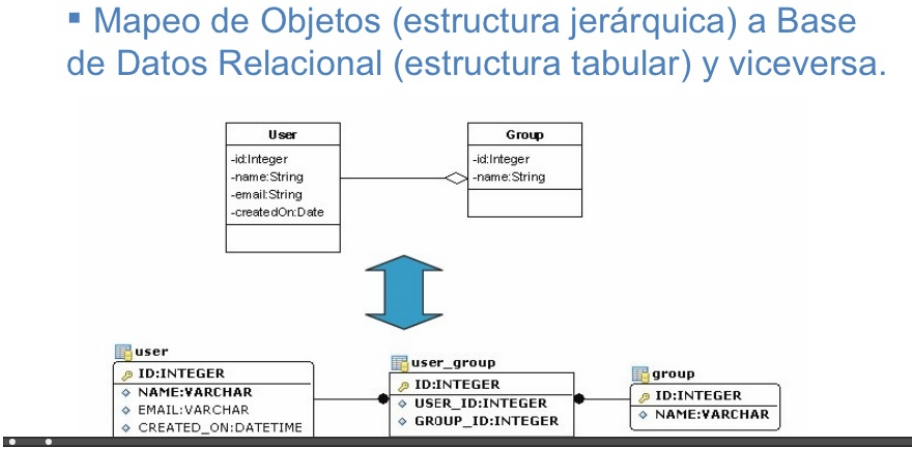
```
1
2  CallableStatement cStmt = conn.prepareCall("{call demoSp(?, ?)}");
3
4  // Parámetros de entrada
5  cStmt.setString(1, "abcdefg");
6
7
```

```
8  // Parámetros de salida
9
10 cStmt.registerOutParameter(2, Types.INTEGER);
11
12 // Para establecer el valor de los parametros, también se puede utilizar
       el nombre
13 // cStmt.setString("inputParameter", "abcdefg");
14
15 // cStmt.setInt("inOutParam", 1);
16
17 // Ejecución
18
19 cStmt.execute();
20
21 // Procesar resultado
22
23       final ResultSet rs = cStmt.getResultSet();
24
25       while (rs.next()) {
26           // process result set
27             ...
28       }
29
30
31 int outputValue = cStmt.getInt(1); // Por posición
32
33 outputValue = cStmt.getInt("inOutParam"); // Por nombre
```

8.5. Serialización: Persistencia de datos en java

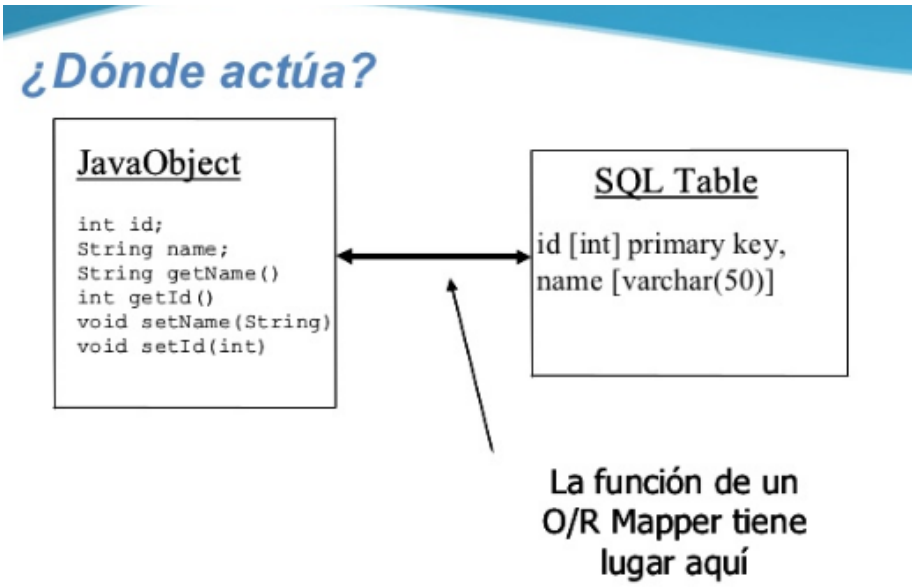
Cuando abordamos el desarrollo de una aplicación en Java, uno de los primeros requerimientos que debemos resolver es la **integración con una base de datos** para guardar, actualizar y recuperar la información que utiliza nuestra aplicación (como ya hemos visto).

Se llama ***persistencia*** de los objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento. La persistencia en base de datos relacionales se suele implementar mediante el desarrollo de funcionalidad específica utilizando la tecnología JDBC (Java Database Connectivity) o mediante **frameworks que automatizan el proceso a partir de mapeos** (conocidos como Object Relational Mapping, ORM).



Los frameworks ORM son librerías que facilitan la vinculación de los objetos de nuestras aplicaciones con la base de datos.

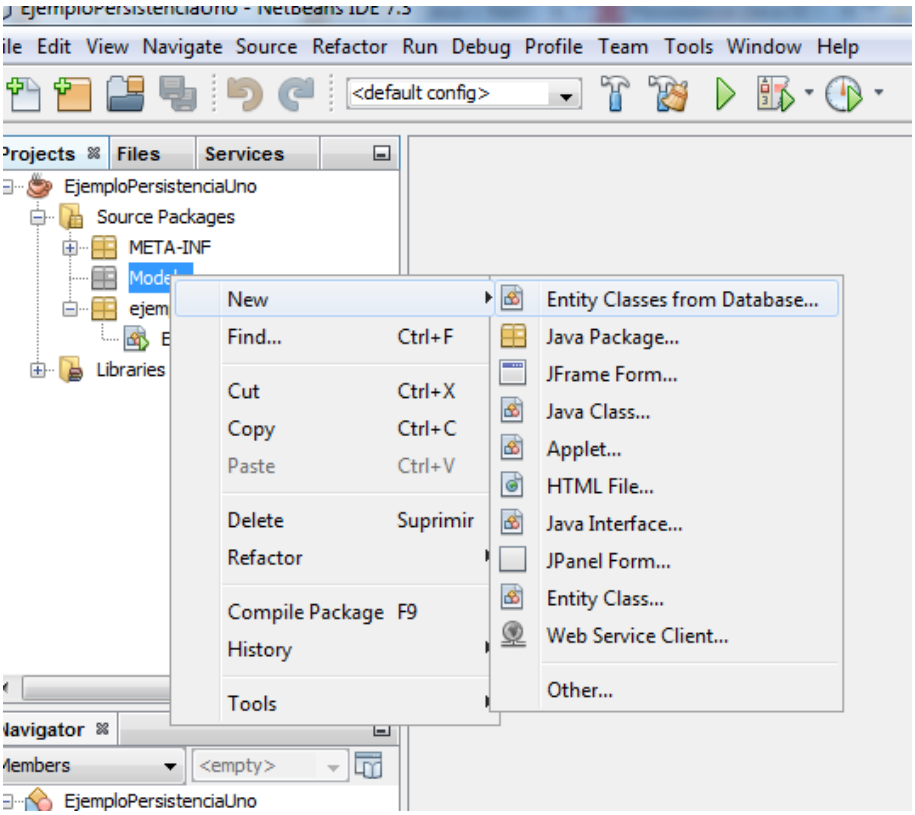
Las aplicaciones, sobretodo en lo referente a la parte que modela el negocio y que hay que persistir, se desarrollan usando programación orientada a objetos (herencia, polimorfismo, ...) Las bases de datos se organizan en estructuras relacionales. Los frameworks ORM facilitan la unión entre estos dos mundos.

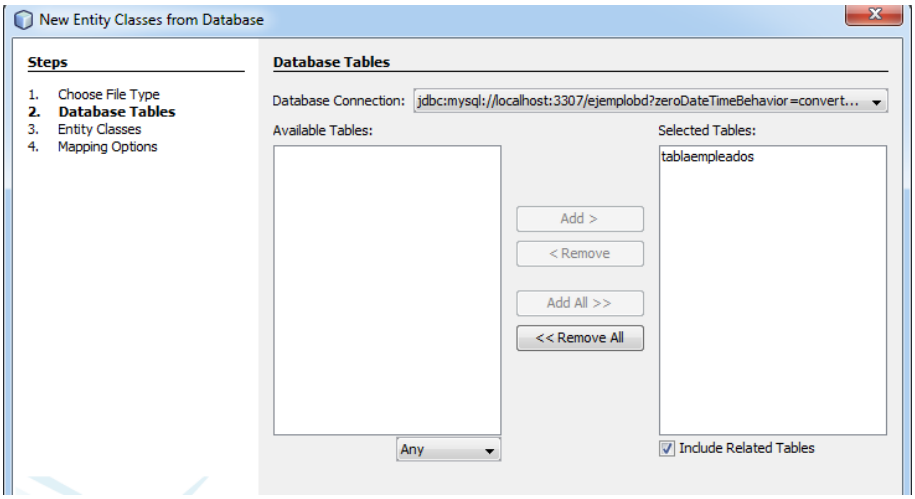


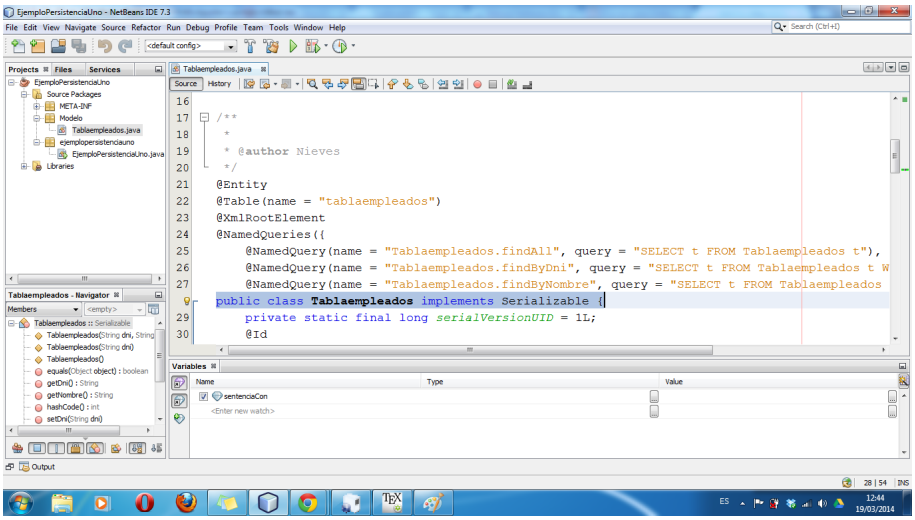
8.5.1. JPA

JPA (Java Persistence API) nos permite establecer una correlación entre una base de datos relacional y un sistema orientado a objetos. ... JPA establece una interface común que es implementada por un proveedor de persistencia de nuestra eleccion (TopLink, EclipseLink, Hibernate, entre otros).

El primer paso para trabajar con JPA es crear las **clases (entidades)** a partir de las tablas de la base de datos.


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
Files
Services
EjemploPersistenciaUno
Source Packages
META-INF
Libraries
New
Find... Ctrl+F
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Suprimir
Refactor
Compile Package F9
History
Tools
Entity Classes from Database...
Java Package...
JFrame Form...
Java Class...
Applet...
HTML File...
Java Interface...
JPanel Form...
Entity Class...
Web Service Client...
Other...
Navigator
Members
<empty>
EjemploPersistenciaUno


New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Database Connection: jdbc:mysql://localhost:3307/ejemplobd?zeroDateTimeBehavior=convert...
Available Tables:
Selected Tables:
tablaempleados
Add >
< Remove
Add All >>
<< Remove All
Any
Include Related Tables



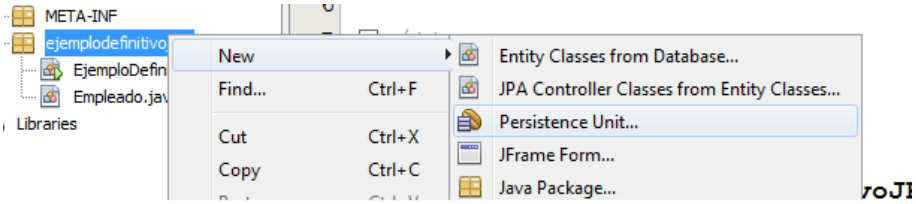
Una **entidad** es un objeto de dominio de persistencia. Normalmente, una entidad representa una tabla en el modelo de datos relacional y cada instancia de esta entidad corresponde a un registro en esa tabla.

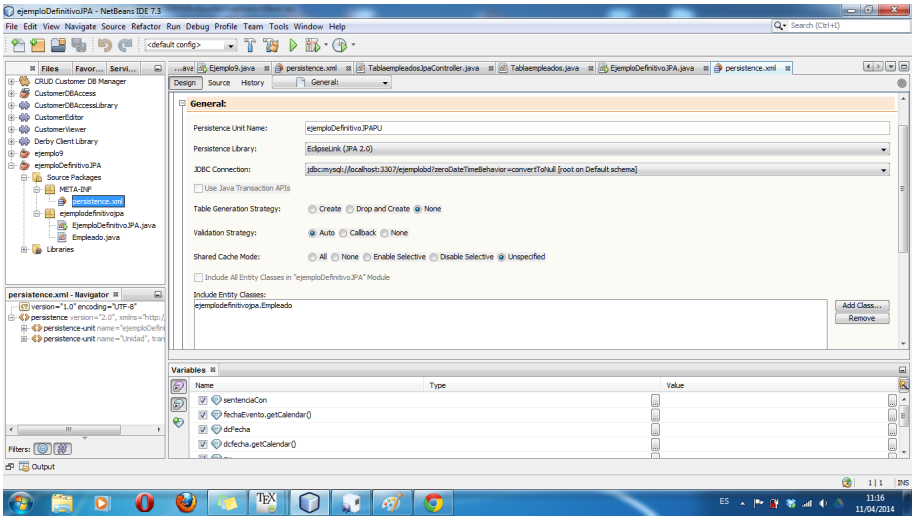
Las entidades se configuran a base de **anotaciones**. Estas anotaciones representan (tablas, columnas, relaciones, multipliciades, etc ...).

```
1  @Entity
2  @Table(name = "tablaempleados")
3  @XmlRootElement
4  @NamedQueries({
5      @NamedQuery(name = "Tablaempleados.findAll", query = "SELECT t FROM
           Tablaempleados t"),
6      @NamedQuery(name = "Tablaempleados.findByDni", query = "SELECT t FROM
           Tablaempleados t WHERE t.dni = :dni"),
7      @NamedQuery(name = "Tablaempleados.findByNombre", query = "SELECT t
           FROM Tablaempleados t WHERE t.nombre = :nombre")})
8
9  public class Tablaempleados implements Serializable {
10     private static final long serialVersionUID = 1L;
11     @Id
12     @Basic(optional = false)
13     @Column(name = "dni")
14     private String dni;
15     @Basic(optional = false)
16     @Column(name = "nombre")
17     private String nombre;
18
19     public Tablaempleados() {
20     }
21
22     public Tablaempleados(String dni) {
23         this.dni = dni;
24     }
25
```

```
26      public Tablaempleados(String dni, String nombre) {
27          this.dni = dni;
28          this.nombre = nombre;
29      }
30
31      public String getDni() {
32          return dni;
33      }
34
35      public void setDni(String dni) {
36          this.dni = dni;
37      }
38
39      public String getNombre() {
40          return nombre;
41      }
42
43      public void setNombre(String nombre) {
44          this.nombre = nombre;
45      }
46
47      @Override
48      public int hashCode() {
49          int hash = 0;
50          hash += (dni != null ? dni.hashCode() : 0);
51          return hash;
52      }
53
54      @Override
55      public boolean equals(Object object) {
56          // TODO: Warning - this method won't work in the case the id
                fields are not set
57          if (!(object instanceof Tablaempleados)) {
58              return false;
59          }
60          Tablaempleados other = (Tablaempleados) object;
61          if ((this.dni == null && other.dni != null) || (this.dni != null
                && !this.dni.equals(other.dni))) {
62              return false;
63          }
64          return true;
65      }
66
67      @Override
68      public String toString() {
69          return "Modelo.Tablaempleados[ dni=" + dni + " ]";
70      }
71
72  }
```

Tras crear entidades, hay que crear un **Unidad de persistencia** (GenericoBD).

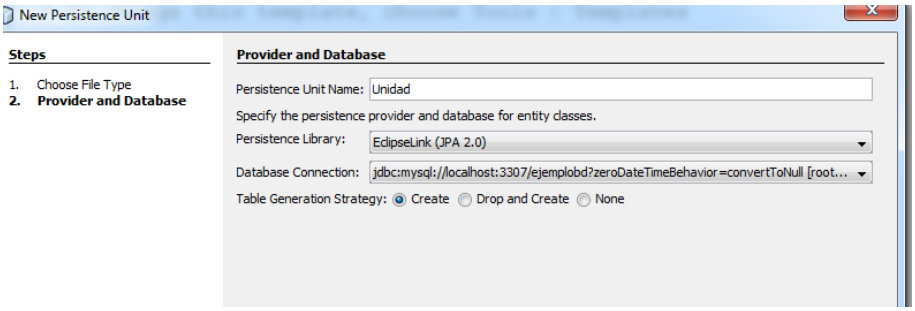




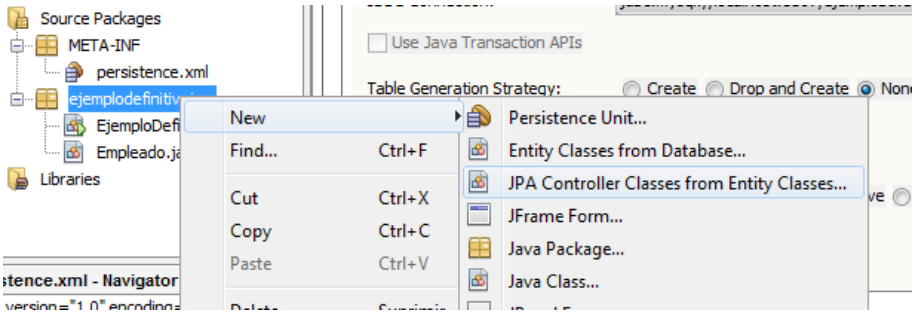
La unidad de persistencia define un conjunto de todas las entidades (clases) que son gestionadas por la instancia del EntityManager en una aplicación. Este conjunto de clases de entidad representa los datos contenidos en una única base de datos.

Las unidades de persistencia se definen en el fichero de configuración persistence.xml.

```
1    <persistence>
2      <persistence-unit name="OrderManagement">
3        <description>...</description>
4        <jta-data-source>jdbc/MyOrderDB</jta-data-source>
5        <jar-file>MyOrderApp.jar</jar-file>
6        <class>com.widgets.Order</class>
7        <class>com.widgets.Customer</class>
8      </persistence-unit>
9    </persistence>
```



Por último, nos falta el **controlador** de cada una de las entidades. Este controlador contendrá los métodos necesarios para trabajar con la entidad (select, insert, delete y updates).(EmpleadosBD)




```
1  package ejemplodefinitivojpa;
2
3  import ejemplodefinitivojpa.exceptions.NonexistentEntityException;
4  import ejemplodefinitivojpa.exceptions.PreexistingEntityException;
5  import java.io.Serializable;
6  import java.util.List;
7  import javax.persistence.EntityManager;
8  import javax.persistence.EntityManagerFactory;
9  import javax.persistence.Query;
10 import javax.persistence.EntityNotFoundException;
11 import javax.persistence.criteria.CriteriaQuery;
12 import javax.persistence.criteria.Root;
13
14 public class EmpleadoJpaController implements Serializable {
15
16     public EmpleadoJpaController(EntityManagerFactory emf) {
17         this.emf = emf;
18     }
19     private EntityManagerFactory emf = null;
20
21     public EntityManager getEntityManager() {
22         return emf.createEntityManager();
23     }
24
```

```
25      public void create(Empleado empleado) throws
            PreexistingEntityException, Exception {
26          EntityManager em = null;
27          try {
28              em = getEntityManager();
29              em.getTransaction().begin();
30              em.persist(empleado);
31              em.getTransaction().commit();
32          } catch (Exception ex) {
33              if (findEmpleado(empleado.getDni()) != null) {
34                  throw new PreexistingEntityException("Empleado " +
                        empleado + " already exists.", ex);
35              }
36              throw ex;
37          } finally {
38              if (em != null) {
39                  em.close();
40              }
41          }
42      }
43
44      public void edit(Empleado empleado) throws
            NonexistentEntityException, Exception {
45          EntityManager em = null;
46          try {
47              em = getEntityManager();
48              em.getTransaction().begin();
49              empleado = em.merge(empleado);
50              em.getTransaction().commit();
51          } catch (Exception ex) {
52              String msg = ex.getLocalizedMessage();
53              if (msg == null || msg.length() == 0) {
54                  String id = empleado.getDni();
55                  if (findEmpleado(id) == null) {
56                      throw new NonexistentEntityException("The empleado
                            with id " + id + " no longer exists.");
57                  }
58              }
59              throw ex;
60          } finally {
61              if (em != null) {
62                  em.close();
63              }
64          }
65      }
66
67      public void destroy(String id) throws NonexistentEntityException {
68          EntityManager em = null;
69          try {
70              em = getEntityManager();
71              em.getTransaction().begin();
72              Empleado empleado;
73              try {
74                  empleado = em.getReference(Empleado.class, id);
75                  empleado.getDni();
76              } catch (EntityNotFoundException enfe) {
77                  throw new NonexistentEntityException("The empleado with
                        id " + id + " no longer exists.", enfe);
78              }
```

```
79              em.remove(empleado);
80              em.getTransaction().commit();
81          } finally {
82              if (em != null) {
83                  em.close();
84              }
85          }
86      }
87
88      public List<Empleado> findEmpleadoEntities() {
89          return findEmpleadoEntities(true, -1, -1);
90      }
91
92      public List<Empleado> findEmpleadoEntities(int maxResults, int
            firstResult) {
93          return findEmpleadoEntities(false, maxResults, firstResult);
94      }
95
96      private List<Empleado> findEmpleadoEntities(boolean all, int
            maxResults, int firstResult) {
97          EntityManager em = getEntityManager();
98          try {
99              CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
100             cq.select(cq.from(Empleado.class));
101             Query q = em.createQuery(cq);
102             if (!all) {
103                 q.setMaxResults(maxResults);
104                 q.setFirstResult(firstResult);
105             }
106             return q.getResultList();
107         } finally {
108             em.close();
109         }
110     }
111
112     public Empleado findEmpleado(String id) {
113         EntityManager em = getEntityManager();
114         try {
115             return em.find(Empleado.class, id);
116         } finally {
117             em.close();
118         }
119     }
120
121     public int getEmpleadoCount() {
122         EntityManager em = getEntityManager();
123         try {
124             CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
125             Root<Empleado> rt = cq.from(Empleado.class);
126             cq.select(em.getCriteriaBuilder().count(rt));
127             Query q = em.createQuery(cq);
128             return ((Long) q.getSingleResult()).intValue();
129         } finally {
130             em.close();
131         }
132     }
133
134 }
```

Creada la estructura jpa la utilizaremos.

```
1  // Creamos el controlador de la entidad relacionandolo con la unidad de
       persistencia
2          EmpleadoJpaController c;
3          c = new
               EmpleadoJpaController(Persistence.createEntityManagerFactory("ejemplodefinitivojpapu"));
4
5          // Creamos objeto de tipo entidad
6          Empleado e = new Empleado();
7          e.setDni("2111111");
8          e.setNombre("Pepe");
9
10         // Realizamos operaciones.
11         try{
12             c.create(e);
13             c.destroy(3);
14             int numeroEmpleados = c.getEmpleadoCount();
15         }
16         catch(Exception e){}
```

8.6. Bases de datos orientadas a objetos

En una base de datos orientada a objetos, la información se representa mediante objetos como los empleados en la programación orientada a objetos. Cuando se integra las características de una base de datos con las de un lenguaje de programación orientado a objetos, el resultado es un **sistema gestor de base de datos orientada a objetos** (ODBMS, object database management system). Los ODBMS usan exactamente el mismo modelo que los lenguajes, es decir, UML.

DB4O es un motor de base de datos orientada a objetos. Sus siglas se corresponden con la expresión "DataBase 4 (for) Objects", que a su vez es el nombre de la compañía que lo desarrolla: db4objects, Inc.

Las claves innovadoras de este producto son su alto rendimiento (sobre todo en modo embebido) y el modelo de desarrollo que proporciona a las aplicaciones para su capa de acceso a datos, el cual propugna un abandono completo del paradigma relacional de las bases de datos tradicionales.

La mayor clave del éxito que está teniendo este motor de base de datos frente a otros competidores que han desarrollado tecnologías similares, es que se ha optado por un modelo de licenciamiento idéntico al utilizado por empresas como MySQL: licencia dual GPL/comercial. Es decir, si se quiere desarrollar software libre con esta librería, su uso no conlleva ningún coste por licencia; sin embargo si se desea aplicar a un software privativo, se aplica otro modelo de licenciamiento concreto.

Una vez descargado tenemos que crear la librería y posteriormente añadirla al proyecto.

http://www.java2s.com/Code/Jar/c/Downloadcomdb4ojar.htm

Para poder trabajar con una base de datos de este tipo debemos seguir los siguientes pasos:

- 1.- Importar las clases del paquete **com.db4o**.

  com.db4o.* contiene métodos estáticos que nos permiten abrir y cerrar la base de datos, conectarnos a un servidor y configurar la base de datos.

  com.db40.ObjectContainer es la interface más importante ya que representa nuestra base de datos. Cada objectContainer mantiene sus propias referencias a objetos almacenados e instanciados.

- 2.- **Abrir** la base de datos.

```
1 ObjectContainer db=Db4o.openFile(''databaseName'');
2
3 db = Db4oEmbedded.openFile("mibd");
4
5 db = com.db4o.cs.Db4oClientServer.openClient(null, port, null, null);
```

  El método openFile abre la base de datos si existe o la crea y abre si no existe. Devuelve un objeto de tipo ObjectContainer que representa la base de datos.

  Db4o nos permite embeber la base de datos, conexiones cliente a servidor y conexiones distribuidas entre servidores.

```
1 ObjectContainer db =
      Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
      DB4OFILENAME);
2
3 EmbeddedConfiguration configuracion =
      Db4oEmbedded.newConfiguration();
4  configuracion.common().objectClass(???.class).updateDepth(100);
5  bd = Db4oEmbedded.openFile(configuracion, "miBd.yap");
```

- 3.- **Cerrar** la base de datos o contenedor de objetos.

  Método close.

```
1 db.close();
```

- 4.- **Guardar** un objeto

  Método set o método store. En las ultimas versiones el método set aparece tachado.

```
1 // storeFirstPilot
2 Pilot pilot1=new Pilot("Michael Schumacher",100);
3 db.store(pilot1);
```

- 5- **Consultas**

  DB4o nos ofrece distintos tipos de consultas.

El resultado de una consulta es un ObjectSet.

Para movernos dentro de un objectset tenemos el método next. Para saber si hay más o no, disponemos del método hasNext.

1. El método get o **queryByExample** permite consultas por ejemplo o prototipo.

```
// Devuelve todos los pilotos con cero puntos
Pilot proto=new Pilot(null,0);
ObjectSet result=db.queryByExample(proto);


// Devuelve todos los objetos de tipo piloto
ObjectSet result=db.queryByExample(Pilot.class);
// También se logra lo mismo con un objeto vacío de tipo Pilot

// Recorrer un ObjectSet (Conjunto de objetos devuelto por la
    consulta
while(result.hasNext()) {
      Pilot objeto = (Pilot) result.next();
      .................................
}

// Consulta por nombre de piloto
Pilot proto=new Pilot("Michael Schumacher",0);
ObjectSet result=db.queryByExample(proto);
```

2. **Native Queries**
   Una segunda forma de crear consultas consiste en crear un predicado o expresión y posteriormente implementar el método match perteneciente a la clase predicate. Por último se utiliza el método query para ejecutar la consulta.

```
List pilots = db.query(new Predicate(){public boolean
    match(Pilot pilot){
                  return pilot.getPoints() > 99
                  && pilot.getPoints() < 199 ||
                  pilot.getName().equals(''Rubens
                      Barrichelo''''·);
                }
          });


List pilots = db.query(new Predicate() {
            public boolean match(Pilot pilot) {
              return pilot.getPoints() == 100;
}
```

   En este caso el tipo de datos devuelto por la consulta es List (similar a ArrayList), que dispone además de los constructores y los métodos addItem() y getItem() de los siguientes métodos:

   - **int countItems()** Devuelve el número de opciones de la Lista.
   - **String getItem(int)** Devuelve la cadena mostrada por la opción del índice especificado.

- **void addItem(String, int)** Añade la opción especificada en el índice especificado.
- **void replaceItem(String, int)** Reemplaza la opción el índice especificado.
- **void clear(), void delItem(int), y void delItems(int, int)** Borran una o más opciones de la lista. El método clear() vacía la lista. El método delItem() borra la opción especificada de la lista. El método delItems() borra las opciones que existan entre los índices especificados (ambos incluidos).
- **int getSelectedIndex()** Devuelve el índice de la opción selecciona en la lista. Devuelve -1 si no se ha seleccionado ninguna opción o si se ha seleccionado más de una.
- **int[] getSelectedIndexes()** Devuelve los índices de las opciones seleccionadas.
- **String getSelectedItem()** Igual getSelectedIndex(), pero devuelve el string con el texto de la opción en lugar de su índice. Develve null si no se ha seleccionado ninguna opción o si se ha seleccionado más de una.
- **String[] getSelectedItems()** Igual getSelectedIndexes(), pero devuelve los strings con el texto de las opciones seleccionadas en lugar de sus índices.
- **boolean isSelected(int)** Devuelve true si la opción con el índice especificado está seleccionada.
- **int getRows()** Devuelve el número de línes visibles en la lista.

3. **SODA Query**
   Son consultas de nodo dinámicas de bajo nivel que permiten directamente recorrer la jerarquia de clases.

```
1  //Consulta de todos los pilotos
2  Query query = db.query();
3  query.constrain(Pilot.class);
4  ObjectSet result = query.execute();
5
6  //Consulta por nombre de piloto
7  Query query = db.query();
8  query.constrain(Pilot.class);
9  query.descend(''name'').constrain(''MS'');
10 ObjectSet result = query.execute();
11
12 //Consulta por nombre y puntos
13 Query query = db.query();
14 query.constrain(Pilot.class);
15 query.descend(''name'').constrain(''MS'');
16 query.descend(''points'').greater(60);
17 ObjectSet result = query.execute();
18
19 Query query=db.query();
```

```
20  query.constrain(Pilot.class);
21  Query pointQuery=query.descend("points");
22  query.descend("name").constrain("Rubens Barrichello")
23  .or(pointQuery.constrain(new Integer(99)).greater()
24  .and(pointQuery.constrain(new Integer(199)).smaller()));
25  ObjectSet result=query.execute();
```

- 6.- **Modificaciones**

  Para modificar un objeto, primero lo recuperamos a través de una consulta, posteriormente lo modificamos y, por último, lo grabaremos (set o store).

```
1
2  ObjectSet result=db.queryByExample(new Pilot("Michael
       Schumacher",0));
3  Pilot found=(Pilot)result.next();
4
5  found.addPoints(11);
6
7  db.store(found);
```

- 7.- **Borrados**

  Para borrar un objeto de la base de datos se utiliza un método delete.

```
1  ObjectSet result=db.queryByExample(new Pilot("Rubens
       Barrichello",0));
2
3  Pilot found=(Pilot)result.next();
4
5  db.delete(found);
```

8.6.1. Objetos estructurados

No tenemos por qué almacenar todos los objetos subordinados si no solo el objeto de mayor nivel.

Si queremos actualizar un objeto subordinado salvando un objeto superior en la base de datos debemos configurar la base de datos para actualización de profundidad con **cascadeOnUpdate**() antes de abrirla. Por defecto la profundidad es 1.

De forma análoga a la actualización el borrado recursivo debe ser configurado mediante **.cascadeOnDelete()** antes de la apertura. Con esto conseguimos borrar los objetos subordinados borrando únicamente el objeto de mayor nivel.

También hay que tener en cuenta el tema de la profundidad cuando trabajamos con arrays y colecciones de datos.

```
1  EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
2
3  config.common().objectClass("com.db4o.f1.chapter2.Car").cascadeOnUpdate(true);
4
5  ObjectContainer db = Db4oEmbedded.openFile(config, DB4OFILENAME);
6
7
8  EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
9
10 config.common().objectClass("com.db4o.f1.chapter2.Car").cascadeOnDelete(true);
11
12 ObjectContainer db = Db4oEmbedded.openFile(config, DB4OFILENAME);
```