Acceso a Datos

Este libro sigue fielmente los contenidos expuestos en el módulo de Acceso a datos perteneciente al título de Técnico Superior en Desarrollo de Aplicaciones Multiplataforma (Real Decreto 450/2010, de 16 de abril, publicado en BOE con fecha 20 de mayo de 2010).

Está dividido en 6 capítulos que se corresponden con los contenidos del título. El libro tiene una orientación práctica; se exponen los contenidos teóricos seguidos de ejemplos y actividades diseñadas para facilitar la comprensión de los mismos. Se parte de los conocimientos previos que el alumno ha adquirido en el primer curso en los módulos profesionales de Bases de datos, Programación (Java), Lenguajes de marcas y sistemas de gestión de información y Entornos de desarrollo.

En el primer capítulo se trata el manejo de ficheros (de bytes, de texto, aleatorios y XML) utilizando el lenguaje JAVA. En el segundo se estudian los protocolos ODBC y JDBC para acceso a bases de datos SQL. Se utilizarán conectores para acceder a diferentes bases de datos SQL (SQLite, Apache Derby, HSQLDB, H2, MySQL, Oracle,...) y orientados a objetos (Db4o). Se realiza una introducción en el patrón de diseño Modelo-Vista-Controlador (MVC) y su uso para el acceso a bases de datos. En el tercer capítulo se utilizará Hibernate como herramientas de mapeo Objeto-Relacional (ORM) para acceder a Oracle y MySQL.

En el cuarto capítulo se utilizará el lenguaje PL/SQL en Oracle y JAVA en NeoDatis para acceder a los objetos almacenados en la base de datos. En el capítulo quinto se utilizará eXist como base de datos nativa XML; se realizarán consultas a documentos y colecciones XML utilizando los lenguajes XPath y XQuery y APIs de JAVA. En el capítulo seis se estudiarán los JavaBeans y se desarrollarán componentes para acceder a bases de datos.

El material disponible para la realización de los ejercicios y actividades se puede descargar de:
http://www.elaltozano.es/ACCESO_A_DATOS/

rceta
grupo editorial

www.garceta.es

ISBN 978-84-1545-228-7

Alicia Ramos Martín
Mª Jesús Ramos Martín

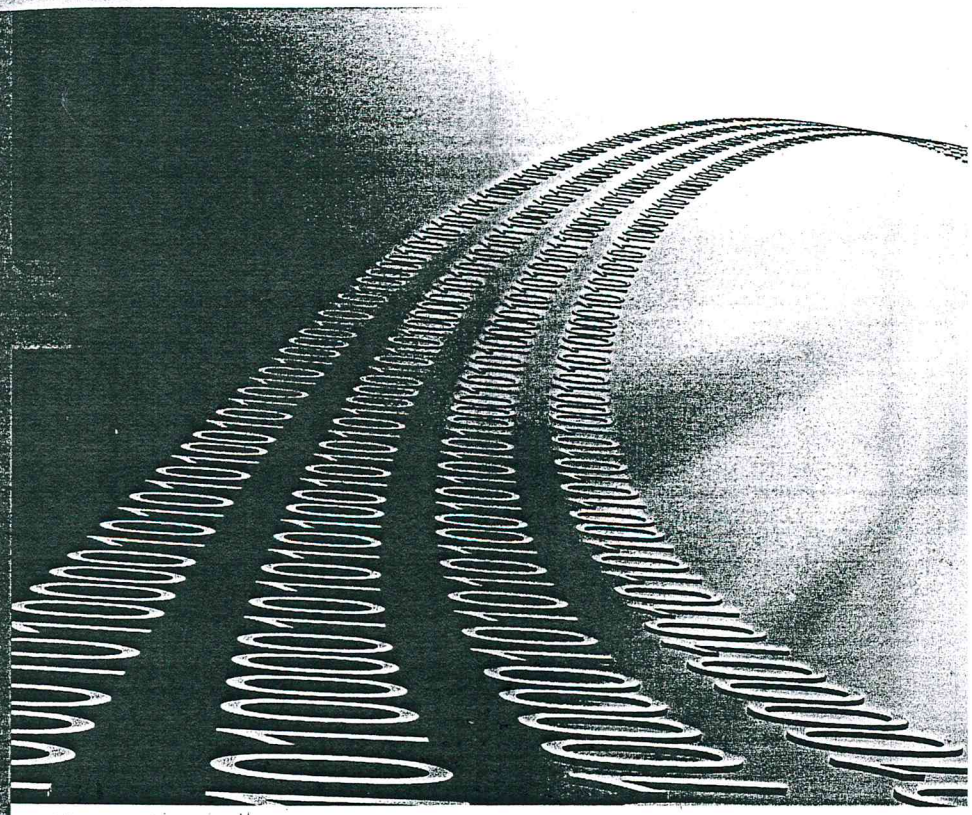
Técnico Superior en Desarrollo de Aplicaciones Multiplataforma

Acceso a Datos

Garceta
grupo editorial

Garceta
grupo editorial

Técnico Superior en Desarrollo de Aplicaciones Multiplataforma



Acceso a Datos

Alicia Ramos Martín
Mª Jesús Ramos Martín

R

Acceso a Datos

Alicia Ramos Martín
Mª Jesús Ramos Martín

Acceso a Datos

Técnico Superior en **Desarrollo de Aplicaciones Multiplataforma**

Garceta
grupo editorial

Acceso a Datos

Alicia Ramos Martín
Mª Jesús Ramos Martín

ISBN: 978-84-1545-228-7
IBERGARCETA PUBLICACIONES, S.L., Madrid 2012

Edición: 1.ª
Impresión: 1.ª
N.º de páginas: 308
Formato: 20 x 26 cm

Reservados los derechos para todos los países de lengua española. De conformidad con lo dispuesto en el artículo 270 y siguientes del código penal vigente, podrán ser castigados con penas de multa y privación de libertad quienes reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica fijada en cualquier tipo de soporte sin la preceptiva autorización. Ninguna parte de esta publicación, incluido el diseño de la cubierta, puede ser reproducida, almacenada o trasmitida de ninguna forma, ni por ningún medio, sea éste electrónico, químico, mecánico, electro-óptico, grabación, fotocopia o cualquier otro, sin la previa autorización escrita por parte de la editorial.

Diríjase a CEDRO (Centro Español de Derechos Reprográficos), www.cedro.org, si necesita fotocopiar o escanear algún fragmento de esta obra.

COPYRIGHT © 2012 IBERGARCETA PUBLICACIONES, S.L.
info@garceta.es

Acceso a Datos

© Alicia Ramos Martín, Mª Jesús Ramos Martín

1.ª edición, 1.ª impresión
OI: 238/2013
ISBN: 978-84-1545-228-7
Deposito Legal: M-26245-2012
Imagen de cubierta: © yellowj_Fotolia.com

Impresión: 1ª

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Nota sobre enlaces a páginas web ajenas: Este libro puede incluir referencias a sitios web gestionados por terceros y ajenos a IBERGARCETA PUBLICACIONES, S.L., que se incluyen sólo con finalidad informativa. IBERGARCETA PUBLICACIONES, S.L., no asume ningún tipo de responsabilidad por los daños y perjuicios derivados del uso de los datos personales que pueda hacer un tercero encargado del mantenimiento de las páginas web ajenas a IBERGARCETA PUBLICACIONES, S.L., y del funcionamiento, accesibilidad y mantenimiento de los sitios web no gestionados por IBERGARCETA PUBLICACIONES, S.L., directamente. Las referencias se proporcionan en el estado en que se encuentran en el momento de publicación sin garantías, expresas o implícitas, sobre la información que se proporcione en ellas.

ÍNDICE GENERAL

UNIDAD 1 MANEJO DE FICHEROS 1

1.1 INTRODUCCIÓN 2

1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS 2

1.3 FLUJOS O STREAMS. TIPOS 5
1.3.1 Flujos de bytes (Byte streams) 6
1.3.2 Flujos de caracteres (Character streams) 7

1.4 FORMAS DE ACCESO A UN FICHERO 8

1.5 OPERACIONES SOBRE FICHEROS 8
1.5.1 Operaciones sobre ficheros secuenciales 9
1.5.2 Operaciones sobre ficheros aleatorios 10

1.6 CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS 11
1.6.1 Ficheros de texto 11
1.6.2 Ficheros binarios 15
1.6.3 Ficheros de acceso aleatorio 20

1.7 TRABAJO CON FICHEROS XML 24
1.7.1 Acceso a ficheros XML con DOM 25
1.7.2 Acceso a ficheros XML con SAX 30
1.7.3 Serialización de objetos a XML 34
1.7.4 Conversión de ficheros XML a otro formato 37

1.8 EXCEPCIONES: DETECCIÓN Y TRATAMIENTO 39
1.8.1 Capturar excepciones 40
1.8.2 Especificar excepciones 42

1.9 EJERCICIOS 45

UNIDAD 2 MANEJO DE CONECTORES 47

2.1 INTRODUCCIÓN 48

2.2 EL DESFASE OBJETO-RELACIONAL 48

2.3 BASES DE DATOS EMBEBIDAS 48
2.3.1 SQLite 49
2.3.2 Apache Derby 50
2.3.3 HSQLDB 53

2.3.4 H2 55
2.3.5 Db4o 58
2.3.6 Otras 62

2.4 PROTOCOLOS DE ACCESO A BASES DE DATOS 66

2.5 ACCESO A DATOS MEDIANTE ODBC 66

2.6 ACCESO A DATOS MEDIANTE JDBC 69
2.6.1 Arquitecturas JDBC 70
2.6.2 Tipos de drivers 71
2.6.3 Cómo funciona JDBC 71
2.6.4 Acceso a datos mediante el Puente JDBC-ODBC 76

2.7 ESTABLECIMIENTO DE CONEXIONES 77

2.8 EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS 83
2.8.1 ResultSetMetaData 86

2.9 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS 88
2.9.1 Sentencias preparadas 91

2.10 EJECUCIÓN DE PROCEDIMIENTOS 93

2.11 GESTIÓN DE ERRORES 97

2.12 PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS 98

2.13 EJERCICIOS 109

UNIDAD 3 HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL (ORM) 113

3.1 INTRODUCCIÓN 114

3.2 CONCEPTO DE MAPEO OBJETO-RELACIONAL 114

3.3 HERRAMIENTAS ORM. CARACTERÍSTICAS 114

3.4 ARQUITECTURA HIBERNATE 116

3.5 INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE 117
3.5.1 Instalación del plugin 119
3.5.2 Configuración del driver MySQL 120
3.5.3 Configuración de Hibernate 122
3.5.4 Generar las clases de la base de datos 127
3.5.5 Primera consulta en HQL 130
3.5.6 Empezando a programar con Hibernate en Eclipse 132

3.6 ESTRUCTURA DE LOS FICHEROS DE MAPEO 139

3.7 CLASES PERSISTENTES 142

3.8 SESIONES Y OBJETOS HIBERNATE 144
3.8.1 Transacciones 145
3.8.2 Estados de un Objeto Hibernate 145
3.8.3 Carga de Objetos 146
3.8.4 Almacenamiento, modificación y borrado de objetos 148

3.9 CONSULTAS 148
3.9.1 Parámetros en las consultas 151
3.9.2 Consultas sobre clases no asociadas 152
3.9.3 Funciones de grupo en las consultas 152
3.9.4 Objetos devueltos por las consultas 153

3.10 INSERT, UPDATE y DELETE 155

3.11 RESUMEN DEL LENGUAJE HQL 157
3.11.1 Asociaciones y uniones (joins) 159

3.12 EJEMPLOS RESUELTOS 162

3.13 EJERCICIOS 170

UNIDAD 4 BASES DE DATOS OBJETO-RELACIONALES Y ORIENTADAS A OBJETOS 171

4.1 INTRODUCCIÓN 172

4.2 BASES DE DATOS OBJETO-RELACIONALES 172
4.2.1 Características 172
4.2.2 Tipos de objetos 173
4.2.3 Tablas de objetos 178
4.2.4 Tipos colección 179
4.2.5 Referencias 184
4.2.6 Herencia de tipos 185
4.2.7 Ejemplo de modelo relacional y objeto-relacional 187

4.3 BASES DE DATOS ORIENTADAS A OBJETOS 191
4.3.1 Características de las bases de datos OO 191
4.3.2 El estándar ODMG 193
4.3.3 El lenguaje de consultas OQL 195

4.4 EJEMPLO DE BDOO 199
4.4.1 Consultas sencillas 203
4.4.2 Consultas más complejas 205
4.4.3 Modo cliente/servidor de la BD 208

4.5 EJERCICIOS 211

UNIDAD 5 BASES DE DATOS XML 215

5.1 BASES DE DATOS NATIVAS XML 216

5.2 BASE DE DATOS EXIST 218
5.2.1 Instalación de eXist 219
5.2.2 Primeros pasos con eXist 220
5.2.3 El cliente de administración de eXist 222

5.3 LENGUAJES DE CONSULTAS XPATH Y XQUERY 224
5.3.1 Expresiones XPath 224
5.3.2 Nodos atributos XPath 231
5.3.3 Axis XPath 235
5.3.4 Consultas XQuery 237
5.3.5 Operadores y funciones más comunes en XQuery 240
5.3.6 Consultas complejas con XQuery 242

5.4 ACCESO A EXIST DESDE JAVA 251
5.4.1 La API XML:DB para bases de datos XML 251
5.4.2 La API XQJ (XQUERY) 256
5.4.3 Tratamiento de excepciones 260

5.5 EJERCICIOS 264

UNIDAD 6 PROGRAMACIÓN DE COMPONENTES DE ACCESO A DATOS 267

6.1 INTRODUCCIÓN 268

6.2 CONCEPTO DE COMPONENTE 268
6.2.1 Características 268
6.2.2 Ventajas e inconvenientes 269

6.3 JAVABEANS 270
6.3.1 Propiedades y atributos 271
6.3.2 Eventos 274
6.3.3 Persistencia del componente 276

6.4 HERRAMIENTAS PARA EL DESARROLLO DEL COMPONENTE 276
6.4.1 Crear JavaBeans con NetBeans 277

6.5 EMPAQUETADO DE COMPONENTES 283

6.6 USANDO JAVABEANS PARA ACCEDER A BASES DE DATOS 284
6.6.1 Usando un componente para acceder a una BDOO 285
6.6.2 Uso de un componente para acceder a bases de datos SQL 290

6.7 EJERCICIOS 300

UNIDAD 1

MANEJO DE FICHEROS

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Utilizar clases para la gestión de ficheros y directorios.
- Valorar las ventajas y los inconvenientes de las distintas formas de acceso.
- Utilizar las operaciones básicas para acceder a ficheros de acceso secuencial y aleatorio.
- Utilizar clases para almacenar y recuperar información almacenada en un fichero XML.
- Utilizar clases para convertir a otro formato información contenida en un fichero XML.
- Gestionar excepciones.

CONTENIDOS

1.1 INTRODUCCIÓN
1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS
1.3 FLUJOS O STREAMS. TIPOS
1.3.1 Flujos de bytes (Byte streams)
1.3.2 Flujos de caracteres (Character streams)
1.4 FORMAS DE ACCESO A UN FICHERO
1.5 OPERACIONES SOBRE FICHEROS
1.5.1 Operaciones sobre ficheros secuenciales
1.5.2 Operaciones sobre ficheros aleatorios
1.6 CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS
1.6.1 Ficheros de texto
1.6.2 Ficheros binarios
1.6.3 Ficheros de acceso aleatorio
1.7 TRABAJO CON FICHEROS XML
1.7.1 Acceso a ficheros XML con DOM
1.7.2 Acceso a ficheros XML con SAX
1.7.3 Serialización de objetos a XML
1.7.4 Conversión de ficheros XML a otro formato
1.8 EXCEPCIONES: DETECCIÓN Y TRATAMIENTO
1.8.1 Capturar excepciones.
1.8.2 Especificar excepciones
1.9 EJERCICIOS

1.1 INTRODUCCIÓN

Un **fichero** o **archivo** es un conjunto de bits almacenado en un dispositivo, como por ejemplo un disco duro. La ventaja de utilizar ficheros es que los datos que guardamos permanecen en el dispositivo aun cuando apaguemos el ordenador, es decir, no son volátiles. Los ficheros tienen un nombre y se ubican en directorios o carpetas, el nombre debe ser único en ese directorio; es decir, no puede haber dos ficheros con el mismo nombre en el mismo directorio. Por convención cuentan con diferentes extensiones que por lo general suelen ser de 3 letras (PDF, DOC, GIF, ...) y nos permiten saber el tipo de archivo.

Un fichero está formado por un conjunto de registros o líneas y cada registro por un conjunto de campos relacionados, por ejemplo un fichero de empleados puede contener datos de los empleados de una empresa, un fichero de texto puede contener líneas de texto, correspondientes a líneas impresas en una hoja de papel. La manera en que se agrupan los datos en el fichero depende completamente de la persona que lo diseñe.

En este tema aprenderemos a utilizar los ficheros con el lenguaje Java.

1.2 CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS

Antes de ver las clases que leen y escriben datos en ficheros, vamos a manejar la clase **File.** Esta clase proporciona un conjunto de utilidades relacionadas con los ficheros que nos van a proporcionar información acerca de los mismos, su nombre, sus atributos, los directorios, etc. Puede representar el nombre de un fichero particular o los nombres de un conjunto de ficheros de un directorio, también se puede usar para crear un nuevo directorio o una trayectoria de directorios completa si esta no existe. Para crear un objeto **File**, se puede utilizar cualquiera de los tres constructores siguientes:

File f1 = new File ("C:\\directorio\\uni1\\ejemplo1.txt")

- *File(String directorioyfichero):* en Linux: new File("/directorio/fichero.txt"); en plataformas Microsoft Windows: new File("C:\\directorio\\fichero.txt");
- *File(String directorio, String nombrefichero):* new File("directorio", "fichero.txt");
- *File(File directorio, String fichero):* new File(new File("directorio"), "fichero.txt")

File f1 = new File("C:\\ejercicios\\uni1", "ejemplo1.txt") ← los dos parámetros pueden ser una variable

En Linux se utiliza como prefijo de una ruta absoluta "/". En Microsoft Windows, el prefijo de un nombre de ruta consiste en la letra de la unidad seguida de ":" y, posiblemente, seguida por "\\" si la ruta es absoluta.

Ejemplos de uso de la clase **File** donde se muestran diversas formas para declarar un fichero:

```
File fichero1 = new File( "C:\\EJERCICIOS\\UNI1\\ejemplo1.txt");//Windows
File fichero1 = new File( "/home/ejercicios/uni1/ejemplo1.txt");//Linux

String directorio= "C:/EJERCICIOS/UNI1";
File fichero2 = new File(directorio, "ejemplo2.txt");

File direc = new File(directorio);
File fichero3 = new File(direc, "ejemplo3.txt");
```

C:\users\dam211\workspace\Tema1

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método **list()** que devuelve un array de Strings con los nombres de los ficheros y directorios contenidos en el directorio asociado al objeto **File.** Para indicar que estamos en el directorio actual creamos un objeto **File** y le pasamos el parámetro ".":

```
import java.io.*;
public class VerDir {
 public static void main(String[] args) {
   System.out.println("Ficheros en el directorio actual:");
   File f = new File(".");
   String[] archivos = f.list();
   for (int i = 0; i < archivos.length; i++) {
     System.out.println(archivos[i]);
   }
 }
}
```

Los argumentos se ponen en Run-Run configurations-Argumentos
Dentro del programa se referencian con args[0], args[1] ..

La siguiente declaración mostraría la lista de ficheros del directorio *d:\db:*

```
File f = new File("d:\\db");
```

Con la siguiente declaración se mostraría la lista de ficheros del directorio introducido desde la línea de comandos al ejecutar el programa:

```
String dir=args[0];
System.out.println("Archivos en el directorio " +dir);
File f = new File(dir);
```

Actividad 1: Realiza un programa Java que muestre los ficheros de un directorio. El nombre del directorio se pasará al programa desde la línea de comandos al ejecutarlo. C:\\

Algunos métodos importantes del objeto **File** son:

- *getName():* devuelve el nombre del fichero o directorio.
- *getPath():* devuelve el camino relativo.
- *getAbsolutePath():* devuelve el camino absoluto del fichero/directorio.
- *canRead():* devuelve true si el fichero se puede leer.
- *canWrite():* devuelve true si el fichero se puede escribir.
- *length():* nos devuelve el tamaño del fichero en bytes.
- *createNewFile():* crea un nuevo fichero, vacío, asociado a **File** si y solo si no existe un fichero con dicho nombre.
- *delete():* borra el fichero o directorio asociado al **File.**
- *exists():* devuelve true si el fichero/directorio existe.
- *getParent():* devuelve el nombre del directorio padre, o *null* si no existe.
- *isDirectory():* devuelve true si el objeto **File** corresponde a un directorio.

- *isFile():* devuelve true si el objeto **File** corresponde a un fichero normal.
- *mkdir():* crea un directorio con el nombre indicado en la creación del objeto **File.**
- *renameTo(File nuevonombre):* renombra el fichero.

El siguiente ejemplo muestra información del fichero *VerInf.java*:

```
import java.io.*;
public class VerInf {
 public static void main(String[] args) {
  System.out.println("INFORMACIÓN SOBRE EL FICHERO:");
  File f = new File("VerInf.java");
  if(f.exists()){
        System.out.println("Nombre del fichero : "+f.getName());
        System.out.println("Ruta               : "+f.getPath());
        System.out.println("Ruta absoluta      : "+f.getAbsolutePath());
        System.out.println("Se puede escribir  : "+f.canRead());
        System.out.println("Se puede leer      : "+f.canWrite());
        System.out.println("Tamaño             : "+f.length());
        System.out.println("Es un directorio   : "+f.isDirectory());
        System.out.println("Es un fichero      : "+f.isFile());
   }
}
}
```

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero  : VerInf.java
Ruta                : VerInf.java
Ruta absoluta       : D:\uni1\VerInf.java
Se puede escribir   : true
Se puede leer       : true
Tamaño              : 794
Es un directorio    : false
Es un fichero       : true
```

El siguiente ejemplo crea un directorio (de nombre NUEVODIR) en el directorio actual, a continuación crea dos ficheros vacíos en dicho directorio y uno de ellos lo renombra. En este caso para crear los ficheros se definen 2 parámetros en el objeto File: *File(File directorio, String nombrefich),* en el primero indicamos el directorio donde se creará el fichero y en el segundo indicamos el nombre del fichero:

```
import java.io.*;
public class CrearDir {
 public static void main(String[] args) {
   File d = new File("NUEVODIR"); //directorio que creo a partir del actual
   File f1 = new File(d,"FICHERO1.TXT");
   File f2 = new File(d,"FICHERO2.TXT");

   d.mkdir();//CREAR DIRECTORIO

   try {
    if (f1.createNewFile())
```



```
        System.out.println("FICHERO1 creado correctamente...");
    else
        System.out.println("No se ha podido crear FICHERO1...");
    if (f2.createNewFile())
        System.out.println("FICHERO2 creado correctamente...");
    else
        System.out.println("No se ha podido crear FICHERO2...");
   } catch (IOException ioe) {ioe.printStackTrace();}

   f1.renameTo(new File(d,"FICHERO1NUEVO"));//renombro FICHERO1

   try {
        File f3 = new File("NUEVODIR/FICHERO3.TXT");
      f3.createNewFile();//crea FICHERO3 en NUEVODIR
   } catch (IOException ioe) {ioe.printStackTrace();}
 }
}
```

Para borrar un fichero o un directorio usamos el método *delete(),* en el ejemplo anterior no podemos borrar el directorio creado porque contiene ficheros, antes habría que eliminarlos. Para borrar *FICHERO2* escribimos:

```
if(f2.delete())
    System.out.println("Fichero borrado...");
else
   System.out.println("No se ha podido borrar el fichero...");
```

El método **createNewFile()** puede lanzar la excepción *IOException,* por ello se utiliza el bloque **try-catch.**

1.3 FLUJOS O STREAMS. TIPOS

El sistema de entrada/salida en Java presenta una gran cantidad de clases que se implementan en el paquete **java.io.** Usa la abstracción del flujo **(stream)** para tratar la comunicación de información entre una fuente y un destino; dicha información puede estar en un fichero en el disco duro, en la memoria, en algún lugar de la red, e incluso, en otro programa. Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, igualmente si necesita enviar información abrirá un stream y se escribirá la información en serie. La vinculación de este stream al dispositivo físico la hace el sistema de entrada y salida de Java. Se definen dos tipos de flujos:

- **Flujos de bytes (8 bits):** realizan operaciones de entradas y salidas de bytes y su uso está orientado a la lectura/escritura de datos binarios. Todas las clases de flujos de bytes descienden de las clases **InputStream** y **OutputStream,** cada una de estas clases tienen varias subclases que controlan las diferencias ente los distintos dispositivos de entrada/salida que se pueden utilizar.
- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases **Reader** y **Writer.** La razón de ser de estas clases es la internacionalización; la antigua jerarquía de flujos de E/S solo soporta flujos de 8 bits, no manejando caracteres Unicode de 16 bits que se utilizaba con fines de internacionalización.

1.3.1 FLUJOS DE BYTES (BYTE STREAMS)

La clase **InputStream** representa las clases que producen entradas de distintas fuentes, estas fuentes pueden ser: un array de bytes, un objeto String, un fichero, una "tubería" (se ponen los elementos en un extremo y salen por el otro), una secuencia de otros flujos, otras fuentes como una conexión a Internet, etc. Los tipos de **InputStream** se resumen en la siguiente tabla:

CLASE	FUNCIÓN
`ByteArrayInputStream`	Permite usar un espacio de almacenamiento intermedio de memoria
`StringBufferInputStream`	Convierte un String en un InputStream
`FileInputStream`	Para leer información de un fichero
`PipedInputStream`	Implementa el concepto de "tubería"
`FilterInputStream`	Proporciona funcionalidad útil a otras clases InputStream
`SequenceInputStream`	Convierte dos o más objetos InputStream en un InputStream único

Los tipos de **OutputStream** incluyen las clases que deciden donde irá la salida: a un array de bytes, un fichero o una "tubería". Se resumen en la siguiente tabla:

CLASE	FUNCIÓN
`ByteArrayOutputStream`	Crea un espacio de almacenamiento intermedio en memoria. Todos los datos que se envían al flujo se ubican en este espacio
`FileOutputStream`	Para enviar información a un fichero
`PipedOutputStream`	Cualquier información que se desee escribir aquí acaba automáticamente como entrada del PipedInputStream asociado. Implementa el concepto de "tubería"
`FilterOutputStream`	Proporciona funcionalidad útil a otras clases OutputStream

La Figura 1.1 muestra la jerarquía de clases para lectura y escritura de flujos de bytes.

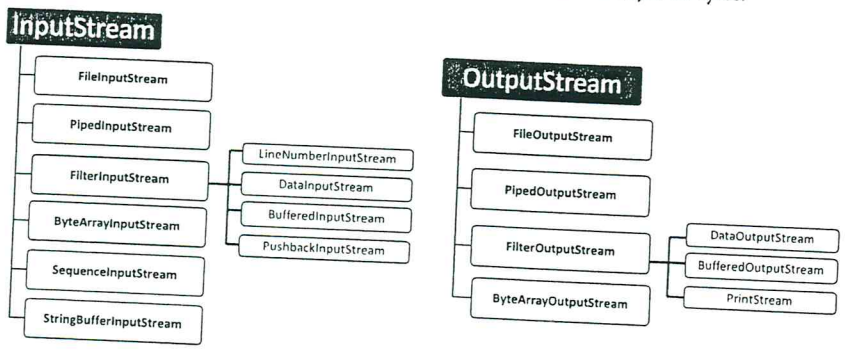


Figura 1. 1. Jerarquia de clases para lectura y escritura de bytes.

Dentro de los flujos de bytes están las clases **FileInputStream** y **FileOutputStream** que manipulan los flujos de bytes provenientes o dirigidos hacia ficheros en disco y se estudiarán en los siguientes apartados.

1.3.2 FLUJOS DE CARACTERES (CHARACTER STREAMS)

Las clases **Reader** y **Writer** manejan flujos de caracteres Unicode. Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto hay clases "puente" (es decir, convierte los streams de bytes a streams de caracteres): **InputStreamReader** que convierte un **InputStream** en un **Reader** (lee bytes y los convierte en caracteres) y **OutputStreamReader** que convierte un **OutputStream** en un **Writer.**

La siguiente tabla muestra la correspondencia entre las clases de flujos de bytes y de caracteres:

CLASES DE FLUJOS DE BYTES	CLASE CORRESPONDIENTE DE FLUJO DE CARACTERES
`InputStream`	`Reader`, convertidor `InputStreamReader`
`OutputStream`	`Writer`, convertidor `OutputStreamReader`
`FileInputStream`	`FileReader`
`FileOutputStream`	`FileWriter`
`StringBufferInputStream`	`StringReader`
(sin clase correspondiente)	`StringWriter`
`ByteArrayInputStream`	`CharArrayReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`PipedInputStream`	`PipedReader`
`PipedOutputStream`	`PipedWriter`

La Figura 1.2 muestra la jerarquía de clases para lectura y escritura de flujos de caracteres.

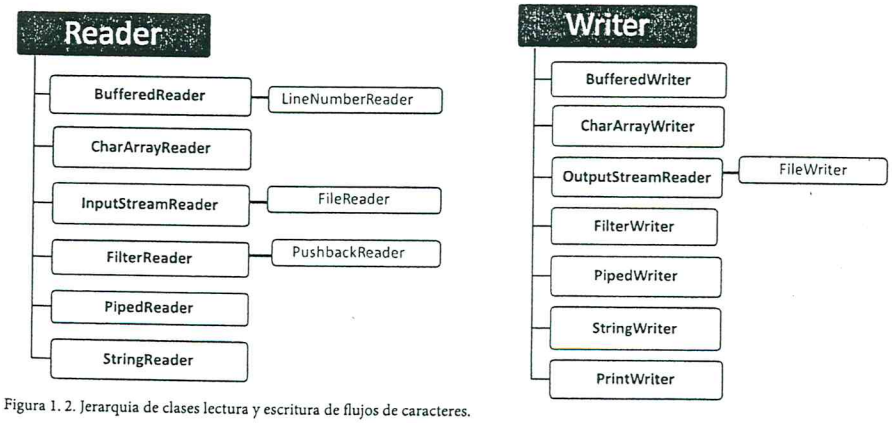


Figura 1. 2. Jerarquia de clases lectura y escritura de flujos de caracteres.

Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **FileReader** y **FileWriter.**
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres: **CharArrayReader** y **CharArrayWriter.**
- Para buferización de datos: **BufferedReader** y **BufferedWriter,** se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que utilizan un buffer intermedio entre la memoria y el stream.

1.4 FORMAS DE ACCESO A UN FICHERO

Hay dos formas de acceso a la información almacenada en un fichero: acceso secuencial y acceso directo o aleatorio:

- **Acceso secuencial:** los datos o registros se leen y se escriben en orden, del mismo modo que se hace en una antigua cinta de vídeo. Si se quiere acceder a un dato o un registro que está hacia la mitad del fichero es necesario leer antes todos los anteriores. La escritura de datos se hará a partir del último dato escrito, no es posible hacer inserciones entre los datos que ya hay escritos.
- **Acceso directo o aleatorio:** permite acceder directamente a un dato o registro sin necesidad de leer los anteriores y se puede acceder a la información en cualquier orden. Los datos están almacenados en registros de tamaño conocido, nos podemos mover de un registro a otro de forma aleatoria para leerlos o modificarlos.

En Java el acceso secuencial más común en ficheros puede ser binario o a caracteres. Para el acceso binario: se usan las clases **FileInputStream** y **FileOutputStream;** para el acceso a caracteres (texto) se usan las clases **FileReader** y **FileWriter.** En el acceso aleatorio se utiliza la clase **RandomAccessFile.**

1.5 OPERACIONES SOBRE FICHEROS

Las operaciones básicas que se realizan sobre cualquier fichero independientemente de la forma de acceso al mismo son las siguientes:

- **Creación del fichero.** El fichero se crea en el disco con un nombre que después se debe utilizar para acceder a él. La creación es un proceso que se realiza una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, la primera operación que tiene que realizar es la apertura del mismo. El programa utilizará algún método para identificar el fichero con el que quiere trabajar, por ejemplo asignar a una variable el descriptor del fichero.
- **Cierre del fichero.** El fichero se debe cerrar cuando el programa no lo vaya a utilizar. Normalmente suele ser la última instrucción del pograma.
- **Lectura de los datos del fichero.** Este proceso consiste en transferir información del fichero a la memoria principal, normalmente a través de alguna variable o variables de nuestro programa, en las que se depositarán los datos extraidos del fichero.

- **Escritura de datos en el fichero.** En este caso el proceso consiste en transferir información de la memoria (por medio de las variables del programa) al fichero.

Normalmente las operaciones típicas que se realizan sobre un fichero una vez abierto son las siguientes:

- **Altas:** consiste en añadir un nuevo registro al fichero.
- **Bajas:** consiste en eliminar del fichero un registro ya existente. La eliminación puede ser lógica, cambiando el valor de algún campo del registro que usemos para controlar dicha situación; o física, eliminando físicamente el registro del fichero. El borrado físico consiste muchas veces en reescribir de nuevo el fichero en otro fichero sin los datos que se desean eliminar y luego renombarlo al fichero original.
- **Modificaciones:** consiste en cambiar parte del contenido de un registro. Antes de realizar la modificación será necesario localizar el registro a modificar dentro del fichero; y una vez localizado se realizan los cambios y se reescribe el registro.
- **Consultas:** consiste en buscar en el fichero un registro determinado.

1.5.1 OPERACIONES SOBRE FICHEROS SECUENCIALES

En los ficheros secuenciales los registros se insertan en orden cronológico, es decir, un registro se inserta a continuación del último insertado. Si hay que añadir nuevos registros estos se añaden a partir del final del fichero.

Veamos como se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro es necesario empezar la lectura desde el primer registro, y continuar leyendo secuencialmente hasta localizar el registro buscado. Por ejemplo, si el registro a buscar es el 90 dentro del fichero, será necesario leer secuencialmente los 89 que le preceden.
- **Altas:** en un fichero secuencial las altas se realizan al final del último registro insertado, es decir, solo se permite añadir datos al final del fichero.
- **Bajas:** para dar de baja un registro de un fichero es necesario leer todos los registros uno a uno y escribirlos en un fichero auxiliar, salvo el que deseamos dar de baja. Una vez reescritos hemos de borrar el fichero inicial y renombrar el fichero auxiliar dándole el nombre del fichero original.
- **Modificaciones:** consiste en localizar el registro a modificar, efectuar la modificación y reescribir el fichero inicial en otro fichero auxiliar que incluya el registro modificado. El proceso es similar a las bajas.

Los ficheros secuenciales se usan típicamente en aplicaciones de proceso por lotes, como por ejemplo en el respaldo de los datos o backup, y son óptimos en dichas aplicaciones si se procesan todos los registros. La ventaja de estos ficheros es la rápida capacidad de acceso al siguiente registro, es decir, son rápidos cuando se accede a los registros de forma secuencial; y que aprovechan mejor la utilización del espacio. También son sencillos de usar y aplicar.

La desventaja es que no se puede acceder directamente a un registro determinado, hay que leer antes todos los anteriores; es decir, no soporta acceso aleatorio. Otra desventaja es el proceso de actualización, la mayoría de los ficheros secuenciales no pueden ser actualizados, habrá que reescribirlos totalmente. Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los ficheros secuenciales ofrecen un rendimiento pobre.

1.5.2 OPERACIONES SOBRE FICHEROS ALEATORIOS

Las operaciones en ficheros aleatorios son las vistas anteriormente pero teniendo en cuenta que para acceder a un registro hay que localizar la posición o dirección donde se encuentra. Los ficheros de acceso aleatorio en disco manipulan direcciones relativas en lugar de direcciones absolutas (número de pista y número de sector en el disco), lo que hace al programa independiente de la dirección absoluta del fichero en el disco.

Normalmente para posicionarnos en un registro es necesario aplicar una función de conversión que usualmente tiene que ver con el tamaño del registro y con la clave del mismo (la clave es el campo o campos que identifica de forma unívoca a un registro). Por ejemplo, disponemos de un fichero de empleados con tres campos: identificador, apellido y salario. Usamos el identificador como campo clave del mismo, y le damos el valor 1 para el primer empleado, 2 para el segundo empleado y así sucesivamente; entonces para localizar al empleado con identificador X necesitamos acceder a la posición tamaño*(X-1) para acceder a los datos de dicho empleado.

Puede ocurrir que al aplicar la función al campo clave, nos devuelva una posición ocupada por otro registro, en ese caso, habría que buscar una nueva posición libre en el fichero para ubicar dicho registro o utilizar una zona de excedentes dentro del mismo para ir ubicando estos registros.

Veamos cómo se realizan las operaciones típicas:

- **Consultas:** para consultar un determinado registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y leer el registro ubicado en esa posición. Habría que comprobar si el registro buscado está en esta posición, si no está se buscaría en la zona de excedentes.
- **Altas:** para insertar un registro necesitamos saber su clave, aplicar la función de conversión a la clave para obtener la dirección y escribir el registro en la posición devuelta. Si la posición está ocupada por otro registro, en ese caso, el registro se insertaría en la zona de excedentes.
- **Bajas:** las bajas suelen realizarse de forma lógica, es decir, se suele utilizar un campo del registro a modo de switch que tenga el valor 1 cuando el registro exista y le damos el valor 0 para darle de baja, físicamente el registro no desaparece del disco. Habría que localizar el registro a dar de baja a partir de su campo clave y reescribir en este campo el valor 0.
- **Modificaciones:** para modificar un registro hay que localizarlo, necesitamos saber su clave para aplicar la función de conversión y así obtener la dirección, modificar los datos que nos interesen y reescribir el registro en esa posición.

Una de las principales ventajas de los ficheros aleatorios es el rápido acceso a una posición determinada para leer o escribir un registro. El gran inconveniente es establecer la relación entre la posición que ocupa el registro y su contenido; ya que a veces al aplicar la función de conversión para obtener la posición se obtienen posiciones ocupadas y hay que recurrir a la zona de excedentes. Otro inconveniente es que se puede desaprovechar parte del espacio destinado al fichero, ya que se pueden producir huecos (posiciones no ocupadas) entre un registro y otro.

1.6 CLASES PARA GESTIÓN DE FLUJOS DE DATOS DESDE/HACIA FICHEROS

En Java podemos utilizar dos tipos de ficheros: de texto o binarios; y el acceso a los mismos se puede realizar de forma secuencial o aleatoria. Los ficheros de texto están compuestos de caracteres legibles, mientras que los binarios pueden almacenar cualquier tipo de dato *(int, float, boolean,...)*.

1.6.1 FICHEROS DE TEXTO

Los ficheros de texto, los que normalmente se generan con un editor, almacenan caracteres alfanuméricos en un formato estándar (ASCII, UNICODE, UTF8, etc.). Para trabajar con ellos usaremos las clases **FileReader** para leer caracteres y **FileWriter** para escribir los caracteres en el fichero. Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno, debemos hacerlo dentro de un manejador de excepciones **try-catch.** Al usar la clase **FileReader** se puede generar la excepción *FileNotFoundException* (porque el nombre del fichero no exista o no sea válido) y al usar la clase **FileWriter** la excepción *IOException* (el disco está lleno o protegido contra escritura).

Los métodos que proporciona la clase **FileReader** para lectura son los siguientes, estos métodos devuelven el número de caracteres leídos o -1 si se ha llegado al final del fichero:

- *int read():* lee un carácter y lo devuelve.
- *int read(char[] buf):* lee hasta *buf.length* caracteres de datos de una matriz de caracteres *(buf)*. Los caracteres leídos del fichero se van almacenando en *buf*.
- *int read(char[] buf, int desplazamiento, int n):* lee hasta n caracteres de datos de la matriz *buf* comenzando por *buf[desplazamiento]* y devuelve el número leído de caracteres.

En un programa Java para crear o abrir un fichero se invoca a la clase **File** y a continuación, se crea el flujo de entrada hacia el fichero con la clase **FileReader.** Después se realizan las operaciones de lectura o escritura y cuando terminemos de usarlo lo cerraremos mediante el método *close()*. El siguiente ejemplo lee cada uno de los caracteres del fichero de texto de nombre *LeerFichTexto.java* (localizado en la carpeta *C:\EJERCICIOS\UNI1)* y los muestra en pantalla, los métodos *read()* pueden lanzar la excepción **IOException,** por ello en *main()* se ha añadido *throws IOException* ya que no se incluye el manejador **try-catch:**

```
import java.io.*;

public class LeerFichTexto {
```

```
    public static void main(String[] args) throws IOException {
      //declarar fichero
      File fichero = new File("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
      FileReader fic = new FileReader(fichero); //crear el flujo de entrada
      int i;
      while ((i = fic.read()) != -1) //se va leyendo un carácter
        System.out.println((char) i);
      fic.close(); //cerrar fichero
    }
  }
```

En el ejemplo, la expresión *((char) i)* convierte el valor entero recuperado por el método *read()* a carácter, es decir, hacemos un *cast* a *char*. Se llega al final del fichero cuando el método *read()* devuelve -1.

Para ir leyendo de 20 en 20 caracteres escribimos:

```
char b[]= new char[20];
while ((i = fic.read(b)) != -1)
        System.out.println(b);
```

Actividad 2: Crea un fichero de texto con algún editor de textos y después realiza un programa Java que visualice su contenido. Cambia el programa Java para que el nombre del fichero se acepte al ejecutar desde la línea de comandos.

FileWriter f = new FileWriter(fichero);

Los métodos que proporciona la clase **FileWriter** para escritura son:

- *void write(int c):* escribe un carácter. char i = "A"; f.write(i);
- *void write(char[] buf):* escribe un array de caracteres. char[] array = ('H','O','L','A'); f.write(array);
- *void write(char[] buf, int desplazamiento , int n):* escribe n caracteres de datos en la matriz *buf* y comenzando por *buf[desplazamiento]*.
- *void write(String str):* escribe una cadena de caracteres. String cadena = "Adios"; f.write(cadena);
- *append(char c):* añade un carácter a un fichero. char c = 'z'; f.append(c);

cerrar flujo: f.close();

Estos métodos también pueden lanzar la excepción **IOException.** Igual que antes declaramos el fichero mediante la clase **File** y a continuación se crea el flujo de salida hacia el fichero con la clase **FileWriter.** El siguiente ejemplo escribe caracteres en un fichero de nombre *FichTexto.txt* (si no existe lo crea). Los caracteres se escriben uno a uno y se obtienen de un String:

```
import java.io.*;
public class EscribirFichTexto {
  public static void main(String[] args) throws IOException {
    File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichTexto.txt");
    FileWriter fic = new FileWriter(fichero); //crear el flujo de salida
    String cadena ="Esto es una prueba con FileWriter";
    char[] cad = cadena.toCharArray();//convierte un String en array de caract
```



```
    for(int i=0; i<cad.length; i++)
        fic.write(cad[i]);  //se va escribiendo un carácter

    fic.append('*'); //añado al final un *
    fic.close();     //cerrar fichero
  }
}
```

En vez de escribir los caracteres uno a uno, también podemos escribir todo el array: *fic.write(cad);*

El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se irán grabando una a continuación de la otra sin saltos de línea:

```
String prov[] = {"Albacete","Avila","Badajoz","Cáceres","Huelva","Jaén",
          "Madrid","Segovia","Soria","Toledo","Valladolid","Zamora"};

for(int i=0; i<prov.length; i++)
        fic.write(prov[i]);
```

Hay que tener en cuenta que si el fichero existe cuando vayamos a escribir caracteres sobre él, todo lo que tenia almacenado anteriormente se borrará. Si queremos añadir caracteres al final, usaremos la clase **FileWriter** de la siguiente manera, colocando en el segundo parámetro del constructor el valor *true:*

```
FileWriter fic = new FileWriter(fichero, true);
```

FileReader no contiene métodos que nos permitan leer líneas completas, pero **BufferedReader** sí; dispone del método *readLine()* que lee una línea del fichero y la devuelve, o devuelve *null* si no hay nada que leer o llegamos al final del fichero. También dispone del método *read()* para leer un carácter. Para construir un **BufferedReader** necesitamos la clase **FileReader:**

```
BufferedReader fichero = new BufferedReader(new FileReader(NombreFichero));
```

El siguiente ejemplo lee el fichero *LeerFichTexto.java* línea por línea y las va visualizando en pantalla, en este caso las instrucciones se han agrupado dentro de un bloque **try-catch:**

```
import java.io.*;
public class LeerFichTextoBuf {
  public static void main(String[] args) {
  try{
      BufferedReader fichero = new BufferedReader(
                        new FileReader("LeerFichTexto.java"));
      String linea;
      while((linea = fichero.readLine())!=null) //lee una línea del fichero
        System.out.println(linea);

      fichero.close();
    }
    catch (FileNotFoundException fn ){
            System.out.println("No se encuentra el fichero");}
```

prueba.txt

```
        catch (IOException io) {
                System.out.println("Error de E/S ");}
    }
  }
```

La clase **BufferedWriter** también deriva de la clase **Writer.** Esta clase añade un buffer para realizar una escritura eficiente de caracteres. Para construir un **BufferedWriter** necesitamos la clase **FileWriter:**

```
BufferedWriter fichero = new BufferedWriter(new FileWriter(NombreFichero));
```

El siguiente ejemplo escribe 10 filas de caracteres en un fichero de texto y después de escribir cada fila salta una línea con el método *newLine():*

```
import java.io.*;
public class EscribirFichTextoBuf {
  public static void main(String[] args) {
  try{
      BufferedWriter fichero = new BufferedWriter
        (new FileWriter("FichTexto.txt"));
      for (int i=1; i<11; i++){
        fichero.write("Fila numero: "+i); //escribe una línea
        fichero.newLine(); //escribe un salto de línea
      }
      fichero.close();
      }
    catch (FileNotFoundException fn ){
            System.out.println("No se encuentra el fichero");}
    catch (IOException io) {
            System.out.println("Error de E/S ");}
  }
}
```

La clase **PrintWriter,** que también deriva de **Writer,** posee los métodos *print(String)* y *println(String)* (idénticos a los de *System.out*) para escribir en un fichero. Ambos reciben un *String* y lo imprimen en un fichero, el segundo método salta de línea. Para construir un **PrintWriter** necesitamos la clase **FileWriter:**

```
PrintWriter  fichero = new PrintWriter (new FileWriter(NombreFichero));
```

El ejemplo anterior usando la clase **PrintWriter** y el método *println()* quedaría:

```
PrintWriter  fichero = new PrintWriter (new FileWriter("FichTexto.txt"));
for (int i=1; i<11; i++)
   fichero.println("Fila numero: "+i);
fichero.close();
```


1.6.2 FICHEROS BINARIOS

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario como ocurría con los ficheros de texto. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que nos permiten trabajar con ficheros son **FileInputStream** (para entrada) y **FileOutputStream** (para salida), estas trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Los métodos que proporciona la clase **FileInputStream** para lectura son similares a los vistos para la clase **FileReader,** estos métodos devuelven el número de bytes leídos o -1 si se ha llegado al final del fichero:

- *int read():* lee un byte y lo devuelve.
- *int read(byte[] b):* lee hasta *b.length* bytes de datos de una matriz de bytes.
- *int read(byte[] b, int desplazamiento , int n):* lee hasta *n* bytes de la matriz *b* comenzando por *b[desplazamiento]* y devuelve el número leído de bytes.

Los métodos que proporciona la clase **FileOutputStream** para escritura son:

- *void write(int b):* escribe un byte.
- *void write(byte[] b):* escribe *b.length* bytes.
- *void write(byte[] b, int desplazamiento , int n):* escribe *n* bytes a partir de la matriz de bytes de entrada y comenzando por *b[desplazamiento].*

El siguiente ejemplo escribe bytes en un fichero y después los visualiza:

```
import java.io.*;
public class EscribirFichBytes {
  public static void main(String[] args) throws IOException {
   File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichBytes.dat");
   //crea flujo de salida hacia el fichero
   FileOutputStream fileout = new FileOutputStream(fichero);
   //crea flujo de entrada
   FileInputStream filein = new FileInputStream(fichero);
   int i;

   for (i=1; i<100; i++)
       fileout.write(i); //escribe datos en el flujo de salida
   fileout.close(); //cerrar stream de salida

   //visualizar los datos del fichero
   while ((i = filein.read()) != -1) //lee datos del flujo de entrada
        System.out.println(i);
   filein.close();  //cerrar stream de entrada
  }
}
```

Para añadir bytes al final del fichero usaremos **FileOutputStream** de la siguiente manera, colocando en el segundo parámetro del constructor el valor *true:*

```
FileOutputStream fileout = new FileOutputStream(fichero, true);
```

Para leer y escribir datos de tipos primitivos: *int, float, long,* etc. usaremos las clases **DataInputStream** y **DataOutputStream.** Estas clases además de los métodos *read()* y *write()* vistos anteriormente proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina. Algunos de los métodos se muestran en la siguiente tabla:

MÉTODOS PARA LECTURA	MÉTODOS PARA ESCRITURA
boolean readBoolean();	void writeBoolean(boolean v);
byte readByte();	void writeByte(int v);
int readUnsignedByte();	void writeBytes(String s);
int readUnsignedShort();	void writeShort(int v);
short readShort();	void writeChars(String s);
char readChar();	void writeChar(int v);
int readInt();	void writeInt(int v);
long readLong();	void writeLong(long v);
float readFloat();	void writeFloat(float v);
double readDouble();	void writeDouble(double v);
String readUTF();	void writeUTF(String str);

(UTF, *Unicode Transmition Format,* se utiliza para transmitir los caracteres de un String de 16 bits codificada en 8 bits.)

Para abrir un objeto **DataInputStream,** se utilizan los mismos métodos que para **FileInputStream,** ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
FileInputStream filein = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(filein);
```

Para abrir un objeto **DataOutputStream,** se utilizan los mismos métodos que para **FileOutputStream,** ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileout);
```

El siguiente ejemplo inserta datos en el fichero *FichData.dat,* los datos los toma de dos arrays, uno contiene los nombres de una serie de personas y el otro sus edades, recorremos los arrays y vamos escribiendo en el fichero el nombre *(dataOS.writeUTF(nombres[i]);)* y la edad *(dataOS.writeInt(edades[i]);):*

```
import java.io.*;
public class EscribirFichData {
  public static void main(String[] args) throws IOException {

    File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
    FileOutputStream fileout = new FileOutputStream(fichero);
```



```
    DataOutputStream dataOS = new DataOutputStream(fileout);

    String nombres[] = {"Ana","Luis Miguel","Alicia","Pedro","Manuel","Andrés",
                        "Julio","Antonio","María Jesús"};

    int edades[] = {14,15,13,15,16,12,16,14,13};

    for (int i=0;i<edades.length; i++){
       dataOS.writeUTF(nombres[i]); //inserta nombre
       dataOS.writeInt(edades[i]);  //inserta edad
    }
    dataOS.close();  //cerrar stream
  }
}
```

El siguiente ejemplo visualiza los datos grabados anteriormente en el fichero, se deben recuperar en el mismo orden en el que se insertaron, es decir, primero obtenemos el nombre y luego la edad:

```
import java.io.*;
public class LeerFichData {
  public static void main(String[] args) throws IOException {
    File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichData.dat");
    FileInputStream filein = new FileInputStream(fichero);
    DataInputStream dataIS = new DataInputStream(filein);

    String n;
    int e;

    try {
     while (true) {
          n = dataIS.readUTF(); //recupera el nombre
          e = dataIS.readInt(); //recupera la edad
          System.out.println("Nombre: " + n + ", edad: " + e);
     }
     }catch (EOFException eo) {}

    dataIS.close();  //cerrar stream
  }
}
```

Se obtiene la siguiente salida:

```
D:\uni1>javac LeerFichData.java

D:\uni1>java LeerFichData
Nombre: Ana, edad: 14
Nombre: Luis Miguel, edad: 15
Nombre: Alicia, edad: 13
Nombre: Pedro, edad: 15
Nombre: Manuel, edad: 16
Nombre: AndrÚs, edad: 12
Nombre: Julio, edad: 16
Nombre: Antonio, edad: 14
Nombre: MarÝa Jes s, edad: 13
```

OBJETOS SERIALIZABLES

Hemos visto cómo se guardan los tipos de datos primitivos en un fichero, pero por ejemplo, si tenemos un objeto de tipo empleado con varios atributos (el nombre, la dirección, el salario, el departamento, el oficio, etc.) y queremos guardarlo en un fichero, tendríamos que guardar cada atributo que forma parte del objeto por separado, esto se vuelve engorroso si tenemos gran cantidad de objetos. Por ello Java nos permite guardar objetos en ficheros binarios; para poder hacerlo, el objeto tiene que implementar la interfaz **Serializable** que dispone de una serie de métodos con los que podremos guardar y leer objetos en ficheros binarios. Los más importantes a utilizar son:

- *void readObject(java.io.ObjectInputStream stream) throws IOException, ClassNotFound Exception:* para leer un objeto.
- *void writeObject(ObjectOutputStream stream) throws IOException:* para escribir un objeto.

La serialización de objetos de Java permite tomar cualquier objeto que implemente la interfaz **Serializable** y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer y escribir objetos serializables a un stream se utilizan las clases Java **ObjectInputStream** y **ObjectOutputStream** respectivamente. A continuación se muestra la clase *Persona* que implementa la interfaz **Serializable** y, que utilizaremos para escribir y leer objetos en un fichero binario. La clase tiene dos atributos: el nombre y la edad, y los métodos *get* para obtener el valor del atributo y *set* para darle valor:

```
import java.io.Serializable;
public class Persona implements Serializable{
    private String nombre;
    private int edad;

    public Persona(String nombre,int edad)   {
      this.nombre=nombre;
      this.edad=edad;
    }
    public Persona() {
      this.nombre=null;
    }
    public void setNombre(String nom){nombre=nom;}
    public void setEdad(int ed){edad=ed;}

    public String getNombre(){return nombre;} //devuelve nombre
    public int getEdad(){return edad;}        //devuelve edad
}//fin Persona
```

El siguiente ejemplo escribe objetos *Persona* en un fichero. Necesitamos crear un flujo de salida a disco con **FileOutputStream** y a continuación se crea el flujo de salida **ObjectOutputStream,** que es el que procesa los datos y se ha de vincular al fichero de **FileOutputStream:**


```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
ObjectOutputStream dataOS = new ObjectOutputStream(fileout);
```

El método *writeObject()* escribe los objetos al flujo de salida y los guarda en un fichero en disco:

```
dataOS.writeObject(persona);
```

El código es el siguiente:

```
import java.io.*;

public class EscribirFichObject {
  public static void main(String[] args) throws IOException {
   Persona persona;//defino variable persona
   //declara el fichero
   File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
   FileOutputStream fileout =
                    new FileOutputStream(fichero);//crea el flujo de salida
    //conecta el flujo de bytes al flujo de datos
   ObjectOutputStream dataOS = new ObjectOutputStream(fileout);

   String nombres[] = {"Ana","Luis Miguel","Alicia","Pedro","Manuel","Andrés",
                       "Julio","Antonio","María Jesús"};

   int edades[] = {14,15,13,15,16,12,16,14,13};

   for (int i=0;i<edades.length; i++){ //recorro los arrays
     persona= new Persona(nombres[i],edades[i]); //creo la persona
     dataOS.writeObject(persona); //escribo la persona en el fichero
   }
   dataOS.close();  //cerrar stream de salida
   }
}
```

Para leer objetos *Persona* del fichero necesitamos el flujo de entrada a disco **FileInputStream** y a continuación crear el flujo de entrada **ObjectInputStream** que es el que procesa los datos y se ha de vincular al fichero de **FileInputStream:**

```
File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
FileInputStream filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);
```

El método *readObject()* lee los objetos del flujo de entrada, puede lanzar la excepción *ClassNotFoundException*:

```
persona= (Persona) dataIS.readObject();
```

El código es el siguiente:

```
import java.io.*;
```

```
public class LeerFichObject {
  public static void main(String[] args) throws IOException,
                                                ClassNotFoundException{
    Persona persona;   //defino la variable persona
    File fichero = new File("C:\\EJERCICIOS\\UNI1\\FichPersona.dat");
    FileInputStream filein = new FileInputStream(fichero);//crea el flujo de entrada
    //conecta el flujo de bytes al flujo de datos
    ObjectInputStream dataIS = new ObjectInputStream(filein);

    try {
     while (true) { //lectura del fichero
       persona= (Persona) dataIS.readObject(); //leer una Persona
       System.out.println("Nombre: " + persona.getNombre() +
                          ", edad: " + persona.getEdad());
     }
     }catch (EOFException eo) {}

    dataIS.close();  //cerrar stream de entrada
  }
}
```

ObjectOutputStream puede darnos algunos problemas, por ejemplo, si escribimos datos en el fichero y lo cerramos. Después volvemos a abrirlo para añadir datos *(FileOutputStream(fichero,true))*, entonces se escribe una nueva cabecera al final de los objetos introducidos anteriormente y después se van añadiendo el resto de datos. Esto origina el problema que al leer el fichero se produzca la excepción *StreamCorruptedException*.

1.6.3 FICHEROS DE ACCESO ALEATORIO

Hasta ahora todas las operaciones que hemos realizado sobre los ficheros se realizaban de forma secuencial. Se empezaba la lectura en el primer byte o el primer carácter o el primer objeto y seguidamente, se leían los siguientes uno a continuación de otro hasta llegar al fin del fichero. Igualmente cuando escribíamos los datos en el fichero se iban escribiendo a continuación de la última información escrita. Java dispone de la clase **RandomAccessFile** que dispone de métodos para acceder al contenido de un fichero binario de forma aleatoria (no secuencial) y para posicionarnos en una posición concreta del mismo. Esta clase no es parte de la jerarqía **InputStream/OutputStream,** ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Disponemos de dos posibilidades para crear el fichero de acceso aleatorio:

- Escribiendo el nombre del fichero: *fichero = new* ***RandomAccessFile****(String nombre,String modoAcceso);*
- Con un objeto **File**: *fichero = new* ***RandomAccessFile****(File fich, String modoAcceso);*

El argumento *modoAcceso* puede ser: **"r"** para solo lectura o **"rw"** para lectura y escritura. Una vez abierto el fichero pueden usarse los métodos *read()* y *write()* de las clases **DataInputStream** y **DataOutputStream** (vistos anteriormente). La clase **RandomAccessFile** maneja un *puntero* que

indica la posición actual en el fichero. Cuando en el fichero se crea el *puntero* se coloca en 0, apuntando al principio del mismo. Las sucesivas llamadas a los métodos *read()* y *write()* ajustan el *puntero* según la cantidad de bytes leídos o escritos.

Los métodos más importantes son:

- *long getFilePointer():* devuelve la posición actual del *puntero* del fichero.
- *void seek(long posicion):* coloca el *puntero* del fichero en una posición determinada desde el comienzo del mismo.
- *long length():* devuelve el tamaño del fichero en bytes. La posición *length()* marca el final del fichero.
- *int skipBytes(int desplazamiento):* desplaza el *puntero* desde la posición actual el número de bytes indicados en desplazamiento.

El ejemplo que se muestra a continuación inserta datos de empleados en un fichero aleatorio. Los datos a insertar: el apellido, departamento y salario, se obtienen de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial (no se usará el método *seek()*). Por cada empleado también se insertará un identificador que coincidirá con el índice +1 con el que se recorren los arrays. La longitud del registro de cada empleado es la misma (36 bytes) y los tipos que se insertan y su tamaño en bytes es el siguiente:

- Se inserta en primer lugar un entero, que es el identificador, ocupa 4 bytes.
- A continuación una cadena de 10 caracteres, es el apellido, cada carácter Unicode ocupa 2 bytes, por tanto el apellido ocupa 20 bytes.
- Un tipo entero que es el departamento, ocupa 4 bytes.
- Un tipo *Double* que es el salario, ocupa 8 bytes.

Tamaño de otros tipos: *short* (2 bytes), *byte* (1 byte), *long* (8 bytes), *boolean* (1bit), *float* (4 bytes), etc .

El fichero se abre en modo **"rw"** para lectura y escritura. El código es el siguiente:

```
import java.io.*;
public class EscribirFichAleatorio {
  public static void main(String[] args) throws IOException {
    File fichero = new File("C:\\EJERCICIOS\\UNI1\\AleatorioEmple.dat");
    //declara el fichero de acceso aleatorio
    RandomAccessFile file = new RandomAccessFile(fichero, "rw");
    //arrays con los datos
    String apellido[] = {"FERNANDEZ","GIL","LOPEZ","RAMOS",
                         "SEVILLA","CASILLA", "REY"};//apellidos
    int dep[] = {10, 20, 10, 10, 30, 30, 20};        //departamentos
    Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
                      2200.0, 1435.87, 2000.0};//salarios

    StringBuffer buffer = null;//buffer para almacenar apellido
    int n=apellido.length;//numero de elementos del array
```

```
    for (int i=0;i<n; i++){ //recorro los arrays
     file.writeInt(i+1); //uso i+1 para identificar empleado
     buffer = new StringBuffer( apellido[i] );
     buffer.setLength(10); //10 caracteres para el apellido
     file.writeChars(buffer.toString());//insertar apellido
     file.writeInt(dep[i]);        //insertar departamento
     file.writeDouble(salario[i]);//insertar salario
     }
     file.close();  //cerrar fichero
     }
  }
```

El siguiente ejemplo toma el fichero anterior y visualiza todos los registros. El posicionamiento para empezar a recorrer los registros empieza en 0, para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento:

```
import java.io.*;
public class LeerFichAleatorio {
  public static void main(String[] args) throws IOException {
   File fichero = new File("C:\\EJERCICIOS\\UNI1\\AleatorioEmple.dat");
   //declara el fichero de acceso aleatorio
   RandomAccessFile file = new RandomAccessFile(fichero, "r");
   //
   int  id, dep, posicion;
   Double salario;
   char apellido[] = new char[10], aux;

   posicion=0;  //para situarnos al principio

   for(;;){  //recorro el fichero
    file.seek(posicion); //nos posicionamos en posicion
    id=file.readInt();   // obtengo id de empleado
    for (int i = 0; i < apellido.length; i++) {
         aux = file.readChar();//recorro uno a uno los caracteres del apellido
         apellido[i] = aux;    //los voy guardando en el array
    }
    String apellidoS= new String(apellido);//convierto a String el array
    dep=file.readInt();//obtengo dep
    salario=file.readDouble();  //obtengo salario

    System.out.println("ID: " + id + ", Apellido: "+  apellidoS +
                       ", Departamento: "+dep + ", Salario: " + salario);
    posicion= posicion + 36; // me posiciono para el sig empleado
                             //Cada empleado ocupa 36 bytes (4+20+4+8)
    //Si he recorrido todos los bytes salgo del for
    if (file.getFilePointer()==file.length())break;

   }//fin bucle for
   file.close();  //cerrar fichero
   }
}
```

La ejecución muestra la siguiente salida:


```
ID: 1, Apellido: FERNANDEZ , Departamento: 10, Salario: 1000.45
ID: 2, Apellido: GIL       , Departamento: 20, Salario: 2400.6
ID: 3, Apellido: LOPEZ     , Departamento: 10, Salario: 3000.0
ID: 4, Apellido: RAMOS     , Departamento: 10, Salario: 1500.56
ID: 5, Apellido: SEVILLA   , Departamento: 30, Salario: 2200.0
ID: 6, Apellido: CASILLA   , Departamento: 30, Salario: 1435.87
ID: 7, Apellido: REY       , Departamento: 20, Salario: 2000.0
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero, conociendo su identificador podemos acceder a la posición que ocupa dentro del mismo y obtener sus datos. Por ejemplo, supongamos que se desean obtener los datos del empleado con identificador 5, para calcular la posición hemos de tener en cuenta los bytes que ocupa cada registro (36):

```
int identificador = 5;
posicion = (identificador - 1 ) * 36; //calculo donde empieza el registro
if(posicion >= file.length())
    System.out.println("ID: " + registro + ", NO EXISTE EMPLEADO...");
 else{
    file.seek(posicion);//nos posicionamos
    id=file.readInt(); // obtengo id de empleado
    //obtener resto de los datos, como en el ejemplo anterior
 }
```

Actividad 3: Crea un programa Java que reciba un identificador de empleado desde la línea de comandos y visualice sus datos. Si el empleado no existe debe visualizar mensaje indicándolo.

Para añadir registros a partir del último insertado, hemos de posicionar el puntero del fichero al final del mismo:

```
long posicion= file.length() ;
file.seek(posicion);
```

Para insertar un nuevo registro aplicamos la función al identificador para calcular la posición. El siguiente ejemplo inserta un empleado con identificador 20, se ha de calcular la posición donde irá el registro dentro del fichero *(identificador -1) * 36 bytes:*

```
StringBuffer buffer = null; //bufer para almacenar apellido
String apellido="GONZALEZ"; //apellido a insertar
Double salario=1230.87; //salario
int id=20;   //id del empleado
int dep=10;  //dep del empleado

long posicion = (id -1 ) * 36; //calculamos la posición

file.seek(posicion); //nos posicionamos
file.writeInt(id); //se escribe id
buffer = new StringBuffer( apellido);
```

```
buffer.setLength(10); //10 caracteres para el apellido
file.writeChars(buffer.toString());//insertar apellido
file.writeInt(dep);        //insertar departamento
file.writeDouble(salario);//insertar salario

file.close();  //cerrar fichero
```

Para modificar un registro determinado, accedemos a su posición y efectuamos las modificaciones. El fichero debe abrirse en modo "**rw**". Por ejemplo para cambiar el departamento y salario del empleado con identificador 4 escribo:

```
int registro = 4  ;//id a modificar

long posicion = (registro -1 ) * 36; //(4+20+4+8)
posicion=posicion+4+20; //sumo el tamaño de ID+apellido
file.seek(posicion); //nos posicionamos
file.writeInt(40);   //modif departamento
file.writeDouble(4000.87);//modif salario
```

Actividad 4: Crea un programa Java que reciba desde la línea de comandos un identificador de empleado y un importe. Se debe sumar al salario del empleado el importe tecleado. El programa debe visualizar el apellido, el salario antiguo y el nuevo. Si el identificador no existe se visualizará un mensaje indicándolo.

1.7 TRABAJO CON FICHEROS XML

XML *(eXtensible Markup Language- Lenguaje de Etiquetado Extensible)* es un metalenguaje, es decir, un lenguaje para la definición de lenguajes de marcado. Nos permite jerarquizar y estructurar la información así como describir los contenidos dentro del propio documento. Los ficheros XML son ficheros de texto escritos en lenguaje XML donde la información está organizada de forma secuencial y en orden jerárquico. Existen una serie de marcas especiales como son los símbolos menor que, < y mayor que , > que se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre y puede tener 0 o más atributos. Un fichero XML sencillo tiene la siguiente estructura:

```
<?xml version="1.0"?>
<Empleados>
    <empleado>
        <id>1</id>
        <apellido>FERNANDEZ</apellido>
        <dep>10</dep>
        <salario>1000.45</salario>
    </empleado>
    <empleado>
        <id>2</id>
        <apellido>GIL</apellido>
        <dep>20</dep>
        <salario>2400.6</salario>
```



```
    </empleado>
    <empleado>
        <id>3</id>
        <apellido>LOPEZ</apellido>
        <dep>10</dep>
        <salario>3000.0</salario>
    </empleado>
</Empleados>
```

Los ficheros XML se pueden utilizar para proporcionar datos a una base de datos o para almacenar copias de partes del contenido de la base de datos. También se utilizan para escribir ficheros de configuración de programas o en el protocolo SOAP (*Simple Object Access Protocol*), para ejecutar comandos en servidores remotos; la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o *parser*. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: **DOM:** *Modelo de Objetos de Documento* y **SAX:** *API Simple para XML*. Son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc. Utilizan dos enfoques muy diferentes:

- **DOM:** un procesador XML que utilice este planteamiento almacena toda la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen. Tiene su origen en el *W3C*. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.

- **SAX:** un procesador que utilice este planteamiento lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta, etcétera) en función de los resultados de la lectura. Cada evento invoca a un método definido por el programador. Este tipo de procesamiento prácticamente no consume memoria, pero por otra parte, impide tener una visión global del documento por el que navegar.

1.7.1 ACCESO A FICHEROS XML CON DOM

Para poder trabajar con **DOM** en Java necesitamos las clases e interfaces que componen el paquete *org.w3c.dom* (contenido en el JSDK) y el paquete *javax.xml.parsers* del API estándar de Java que proporciona un par de clases abstractas que toda implementación **DOM** para Java debe extender. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos (fichero, *InputStream*, etc.). Contiene dos clases fundamentales: *DocumentBuilderFactory* y *DocumentBuilder*.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol **DOM**. Para eso usaremos el paquete *javax.xml.transform* que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos **DOM** entre otros.

Los programas Java que utilicen **DOM** necesitan estas interfaces (no se exponen todas, solo algunas de las que usaremos en los ejemplos):

- **Document.** Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element.** Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- **Node.** Representa a cualquier nodo del documento.
- **NodeList.** Contiene una lista con los nodos hijos de un nodo.
- **Attr.** Permite acceder a los atributos de un nodo.
- **Text.** Son los datos carácter de un elemento.
- **CharacterData.** Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
- **DocumentType.** Proporciona información contenida en la etiqueta <!DOCTYPE>.

A continuación vamos a crear un fichero XML, a partir del fichero aleatorio de empleados creado en el epígrade anterior. Lo primero que hemos de hacer es importar los paquetes necesarios:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;
```

A continuación creamos una instancia de *DocumentBuilderFactory* para construir el parser (se debe encerrar entre **try-cath** porque se puede producir la excepción *ParserConfigurationException:*

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try{
  DocumentBuilder builder = factory.newDocumentBuilder();
  ........
```

Creamos un documento vacío de nombre *document* con el nodo raíz de nombre *Empleados* y asignamos la versión del XML:

```
DOMImplementation implementation = builder.getDOMImplementation();
Document document = implementation.createDocument(null, "Empleados", null);
document.setXmlVersion("1.0"); // asignamos la version de nuestro XML
```

El siguiente paso sería recorrer el fichero con los datos de los empleados y por cada registro crear un nodo empleado con 4 hijos *(id, apellido, dep y salario)*. Cada nodo hijo tendrá su valor (por ejemplo: *1, FERNANDEZ, 10, 1000.45*). Para crear un elemento usamos el método *createElement(String)* llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo *<empleado>* al documento:

```
Element raiz = document.createElement("empleado"); // creamos el nodo empleado
document.getDocumentElement().appendChild(raiz);//lo pegamos a la raiz del doc
```


A continuación se añaden los hijos de ese nodo (raiz), estos se añaden en la función *CrearElemento():*

```
CrearElemento("id",Integer.toString(id),   raiz, document);//añadir ID
CrearElemento("apellido",apellidoS.trim(), raiz, document);//añadir APELLIDO
CrearElemento("dep",Integer.toString(dep), raiz, document);//añadir DEP
CrearElemento("salario",Double.toString(salario), raiz, document);//Añadir SALARIO
```

Como se puede ver la función recibe el nombre del nodo hijo *(id, apellido, dep o salario)* y sus textos o valores que tienen que estar en formato String (*1, FERNANDEZ, 10, 1000.45*), el nodo al que se va a añadir *(raiz)* y el documento *(document)*. Para crear el nodo hijo (<id> o <apellido> o <dep> o <salario>) se escribe:

```
Element elem = document.createElement(datoEmple); //creamos un hijo
```

Para añadir su valor o su texto se usa el método *createTextNode(String):*

```
Text text = document.createTextNode(valor); //damos valor
```

A continuación se añade el nodo hijo a la raíz *(empleado)* y su texto o valor al nodo hijo:

```
raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
elem.appendChild(text); //pegamos el valor
```

Al final se generaría algo similar a esto por cada empleado:

```
<empleado><id>1</id><apellido>FERNANDEZ</apellido><dep>10</dep><salario>1000.45<
/salario></empleado>
```

La función es la siguiente:

```
static void  CrearElemento(String datoEmple, String valor,
                           Element raiz, Document document ){
        Element elem = document.createElement(datoEmple); //creamos hijo
        Text text = document.createTextNode(valor); //damos valor
        raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
        elem.appendChild(text); //pegamos el valor
 }
```

En los últimos pasos se crea la fuente XML a partir del documento:

```
Source source = new DOMSource(document);
```

Se crea el resultado en el fichero Empleados.xml:

```
Result result = new StreamResult(new java.io.File("Empleados.xml")); //fich XML
```

Se obtiene un *TransfomerFactory:*

```
Transformer transformer = TransformerFactory.newInstance().newTransformer();
```

Se realiza la transformación del documento a fichero.

```
transformer.transform(source, result);
```

Para mostrar el documento por pantalla, podemos especificar como resultado el canal de salida *System.out:*

```
Result console= new StreamResult(System.out);
transformer.transform(source, console);
```

El código completo es el siguiente:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class CrearEmpleadoXml {
 public static void main(String argv[]) throws IOException{
   File fichero = new File("C:\\EJERCICIOS\\UNI1\\AleatorioEmple.dat");
   RandomAccessFile file = new RandomAccessFile(fichero, "r");

   int  id, dep, posicion=0; //para situarnos al principio del fichero
   Double salario;
   char apellido[] = new char[10], aux;

   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

   try{
     DocumentBuilder builder = factory.newDocumentBuilder();
     DOMImplementation implementation = builder.getDOMImplementation();
     Document document = implementation.createDocument(
                                    null, "Empleados", null);
     document.setXmlVersion("1.0"); // asignamos la version de nuestro XML

     for(;;) {
       file.seek(posicion); //nos posicionamos
       id=file.readInt(); // obtengo id de empleado
       for (int i = 0; i < apellido.length; i++) {
         aux = file.readChar();//recorro uno a uno los caracteres del apellido
         apellido[i] = aux;     //los voy guardando en el array
       }
       String apellidoS= new String(apellido);//convierto a String el array
       dep=file.readInt();//obtengo dep
       salario=file.readDouble();  //obtengo salario

       if(id>0) { //id validos a partir de 1
       Element raiz = document.createElement("empleado"); //nodo empleado
```



```
       document.getDocumentElement().appendChild(raiz);
       CrearElemento("id",Integer.toString(id), raiz, document); //añadir ID
       CrearElemento("apellido",apellidoS.trim(), raiz, document); //Apellido
       CrearElemento("dep",Integer.toString(dep), raiz, document); //añadir DEP
       CrearElemento("salario",Double.toString(salario), raiz, document); //SAL
       }
       posicion= posicion + 36; // me posiciono para el sig empleado
       if (file.getFilePointer() == file.length())   break;

     }//fin del for que recorre el fichero

     Source source = new DOMSource(document);
     Result result = new StreamResult(new java.io.File("Empleados.xml"));
     Transformer transformer =
                 TransformerFactory.newInstance().newTransformer();
     transformer.transform(source, result);

     }catch(Exception e){System.err.println("Error: "+e);}
     file.close();  //cerrar fichero
   }//fin de main

   //Inserción de los datos del empleado
   static void  CrearElemento(String datoEmple, String valor,
                              Element raiz, Document document){
       Element elem = document.createElement(datoEmple); //creamos hijo
       Text text = document.createTextNode(valor); //damos valor
       raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
       elem.appendChild(text); //pegamos el valor
   }
}//fin de la clase
```

Para leer un documento XML, creamos una instancia de *DocumentBuilderFactory* para construir el parser y cargamos el documento con el método *parse():*

```
Document document = builder.parse(new File("Empleados.xml"));
```

Obtenemos la lista de nodos con nombre *empleado* de todo el documento:

```
NodeList empleados = document.getElementsByTagName("empleado");
```

Se realiza un bucle para recorrer esta lista de nodos. Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función *getNodo().* El código es el siguiente:

```
import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class LecturaEmpleadoXml {
 public static void main(String argv[]) {

   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

```
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(new File("Empleados.xml"));
        document.getDocumentElement().normalize();

        System.out.println("Elemento raíz: " +
                document.getDocumentElement().getNodeName());
        //crea una lista con todos los nodos empleado
        NodeList empleados = document.getElementsByTagName("empleado");
        //recorrer la lista
        for (int i = 0; i < empleados.getLength(); i ++) {
          Node emple = empleados.item(i); //obtener un nodo
          if (emple.getNodeType() == Node.ELEMENT_NODE) {//tipo de nodo
              Element elemento = (Element) emple; //obtener los elementos del nodo
              System.out.println("ID: " + getNodo("id", elemento));
              System.out.println("Apellido: " + getNodo("apellido", elemento));
              System.out.println("Departamento: " + getNodo("dep", elemento));
              System.out.println("Salario: " + getNodo("salario", elemento));
          }
        }
    } catch (Exception e) {e.printStackTrace();}
  }//fin de main

  //obtener la información de un nodo
  private static String getNodo(String etiqueta, Element elem)
  {
    NodeList nodo= elem.getElementsByTagName(etiqueta).item(0).getChildNodes();
    Node valornodo = (Node) nodo.item(0);
    return valornodo.getNodeValue();//devuelve el valor del nodo
  }
}//fin de la clase
```

API DOM: http://docs.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/package-tree.html

Actividad 5: A partir del fichero de objetos Persona utilizado anteriormente, crea un documento XML usando **DOM**.

1.7.2 ACCESO A FICHEROS XML CON SAX

SAX *(API Simple para XML)* es un conjunto de clases e interfaces que ofrecen una herramienta muy útil para el procesamiento de documentos XML. Permite analizar los documentos de forma secuencial (es decir, no carga todo el fichero en memoria como hace **DOM**), esto implica poco consumo de memoria aunque los documentos sean de gran tamaño, en contraposición, impide tener una visión global del documento que se va a analizar. **SAX** es más complejo de programar que **DOM**, es un API totalmente escrita en Java e incluida dentro del JRE que nos permite crear nuestro propio parser de XML.

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin del documento *(startDocument()* y *endDocument())*, la

etiqueta de inicio y fin de un elemento *(startElement()* y *endElement())*, los caracteres entre etiquetas *(characters())*, etc.:

DOCUMENTO XML (ALUMNOS.XML)	MÉTODOS ASOCIADOS A EVENTOS DEL DOCUMENTO
`<?xml version="1.0"?>`	`startDocument()`
`<listadealumnos>`	`startElement()`
`<alumno>`	`startElement()`
`<nombre>`	`startElement()`
`Juan`	`characters()`
`</nombre>`	`endElement()`
`<edad>`	`startElement()`
`19`	`characters()`
`</edad>`	`endElement()`
`</alumno>`	`endElement()`
`<alumno>`	`startElement()`
`<nombre>`	`startElement()`
`Maria`	`characters()`
`</nombre>`	`endElement()`
`<edad>`	`startElement()`
`20`	`characters()`
`</edad>`	`endElement()`
`</alumno>`	`endElement()`
`</listadealumnos>`	`endElement()`
	`endDocument()`

Vamos a construir un ejemplo sencillo en Java que muestra los pasos básicos necesarios para hacer que se puedan tratar los eventos. En primer lugar se incluyen las clases e interfaces de **SAX:**

```
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
```

Se crea un objeto procesador de XML, es decir un *XMLReader*, durante la creación de este objeto se puede producir una excepción *(SAXException)* que es necesario capturar (se incluye en el método *main()):*

```
XMLReader  procesadorXML = XMLReaderFactory.createXMLReader();
```

A continuación, hay que indicar al *XMLReader* qué objetos poseen los métodos que tratarán los eventos. Estos objetos serán normalmente implementaciones de las siguientes interfaces:

- **ContentHandler:** recibe las notificaciones de los eventos que ocurren en el documento.
- **DTDHandler:** recoge eventos relacionados con la DTD.
- **ErrorHandler:** define métodos de tratamientos de errores.
- **EntityResolver:** sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
- **DefaultHandler:** clase que provee una implementación por defecto para todos sus métodos, el

programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML. En el ejemplo la clase se llama *GestionContenido* y se tratan solo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada, encuentra datos carácter (*startDocument(), endDocument(), startElement(), endElement(),characters()*):

- *startDocument:* se produce al comenzar el procesado del documento XML.
- *endDocument:* se produce al finalizar el procesado del documento XML.
- *startElement:* se produce al comenzar el procesado de una etiqueta XML. Es aquí donde se leen los atributos de las etiquetas.
- *endElement:* se produce al finalizar el procesado de una etiqueta XML.
- *characters:* se produce al encontrar una cadena de texto.

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos *XMLReader: setContentHandler(), setDTDHandler(), setEntityResolver()* y *setErrorHandler();* cada uno trata un tipo de evento y está asociado con una interfaz determinada. En el ejemplo usaremos *setContentHandler()* para tratar los eventos que ocurren en el documento:

```
GestionContenido gestor= new GestionContenido();
procesadorXML.setContentHandler(gestor);
```

A continuación se define el fichero XML que se va a leer mediante un objeto *InputSource:*

```
InputSource fileXML = new InputSource("alumnos.xml");
```

Por último, se procesa el documento XML mediante el método *parse()* del objeto XMLReader, le pasamos un objeto *InputSource:*

```
procesadorXML.parse(fileXML);
```

El ejemplo completo se muestra a continuación:

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

public class PruebaSax1 {
    public static void main(String[] args)
                throws FileNotFoundException, IOException, SAXException{

     XMLReader  procesadorXML = XMLReaderFactory.createXMLReader();
     GestionContenido gestor= new GestionContenido();
     procesadorXML.setContentHandler(gestor);
```



```
     InputSource fileXML = new InputSource("alumnos.xml");
     procesadorXML.parse(fileXML);
    }
}//fin PruebaSax1

class GestionContenido extends DefaultHandler {
        public GestionContenido() {
            super();
        }

        public void startDocument() {
            System.out.println("Comienzo del Documento XML");
        }

        public void endDocument() {
            System.out.println("Final del Documento XML");
        }

        public void startElement(String uri, String nombre,
                  String nombreC, Attributes atts) {
            System.out.println("\tPrincipio Elemento: " + nombre);
        }
        public void endElement(String uri, String nombre, String nombreC) {
            System.out.println("\tFin Elemento: " + nombre);
        }

        public void characters(char[] ch, int inicio, int longitud)
                                        throws SAXException {
              String car=new String(ch, inicio, longitud);
              car = car.replaceAll("[\t\n]","");//quitar saltos de línea
              System.out.println ("\tCaracteres: " + car);
        }

}//fin GestionContenido
```

En el resultado de ejecutar el programa con el fichero *alumnos.xml* se puede observar cómo el orden de ocurrencia de los eventos está relacionado con la estructura del documento:

```
Comienzo del Documento XML
        Principio Elemento: listadealumnos
        Caracteres:
        Principio Elemento: alumno
        Caracteres:
        Principio Elemento: nombre
        Caracteres:   Juan
        Fin Elemento: nombre
        Caracteres:
        Principio Elemento: edad
        Caracteres:   19
        Fin Elemento: edad
        Caracteres:
        Fin Elemento: alumno
        Caracteres:
        Principio Elemento: alumno
```

```
        Caracteres:
        Principio Elemento: nombre
        Caracteres:  Maria
        Fin Elemento: nombre
        Caracteres:
        Principio Elemento: edad
        Caracteres:  20
        Fin Elemento: edad
        Caracteres:
        Fin Elemento: alumno
        Caracteres:
        Fin Elemento: listadealumnos
Final del Documento XML
```

API SAX: http://www.saxproject.org/apidoc/org/xml/sax/package-tree.html

Actividad 6: Utiliza **SAX** para visualizar el contenido del fichero Empleados.xml creado anteriormente.

1.7.3 SERIALIZACIÓN DE OBJETOS A XML

A continuación vamos a ver cómo se pueden serializar de forma sencilla objetos Java a XML y viceversa; utilizaremos para ello la librería **XStream.** Para poder utilizarla hemos de descargarnos los JAR desde el sitio web: http://xstream.codehaus.org/download.html, para el ejemplo se ha descargado el fichero *Binary distribution (xstream-distribution-1.4.2-bin.zip)* que hemos de descomprimir y buscar el *JAR xstream-1.4.2.jar* que está en la carpeta *lib* que es el que usaremos para el ejemplo. También necesitamos el fichero *kxml2-2.3.0.jar* que se puede descargar desde el apartado *Optional Dependencies.* Una vez que tenemos los dos ficheros, los definimos en el CLASSPATH, por ejemplo, supongamos que tenemos los JAR en la carpeta *D:\uni1\xstream:*

```
SET CLASSPATH=.;D:\uni1\xstream\kxml2-2.3.0.jar;D:\uni1\xstream\xstream-1.4.2.jar
```

Partimos del fichero *FichPersona.dat* que utilizamos en epígrafes anteriores y contiene objetos *Persona.* Crearemos una lista de objetos *Persona* y la convertiremos en un fichero de datos XML. Necesitaremos la clase *Persona* (ya definida) y la clase *ListaPersonas* en la que se define una lista de objetos *Persona* que pasaremos al fichero XML:

```
import java.util.ArrayList;
import java.util.List;
public class ListaPersonas {
    private List<Persona> lista = new ArrayList<Persona>();

    public ListaPersonas(){ }

    public void add(Persona per) {
            lista.add(per);
```



```
    }
    public List<Persona> getListaPersonas() {
            return lista;
    }
}
```

El proceso consistirá en recorrer el fichero *FichPersona.dat* para crear una lista de personas que después se insertarán en el fichero *Personas.xml,* el código Java es el siguiente (fichero *EscribirPersonas.java):*

```
import java.io.*;
import com.thoughtworks.xstream.XStream;

public class EscribirPersonas {
  public static void main(String[] args) throws IOException,
                                                ClassNotFoundException {
    File fichero = new File("FichPersona.dat");
    FileInputStream filein = new FileInputStream(fichero);//flujo de entrada
     //conecta el flujo de bytes al flujo de datos
    ObjectInputStream dataIS = new ObjectInputStream(filein);
    System.out.println("Comienza el proceso de creación del fichero a XML ...");

    //Creamos un objeto Lista de Personas
    ListaPersonas listaper = new ListaPersonas();

    try {
      while (true) { //lectura del fichero
        Persona persona= (Persona) dataIS.readObject(); //leer una Persona
        listaper.add(persona); //añadir persona a la lista
      }
    }catch (EOFException eo) {}
    dataIS.close();  //cerrar stream de entrada

    try {
          XStream xstream = new XStream();
          //cambiar de nombre a las etiquetas XML
          xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
          xstream.alias("DatosPersona", Persona.class);
          //quitar etiqueta lista (atributo de la clase ListaPersonas)
          xstream.addImplicitCollection(ListaPersonas.class, "lista");
          //Insertar los objetos en el XML
          xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
          System.out.println("Creado fichero XML....");

     }catch (Exception e)
       {e.printStackTrace();}
  } // fin main
} //fin EscribirPersonas
```

El fichero generado tiene el siguiente aspecto:

```
<ListaPersonasMunicipio>
  <DatosPersona>
    <nombre>Ana</nombre>
    <edad>14</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Luis Miguel</nombre>
    <edad>15</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Alicia</nombre>
    <edad>13</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Pedro</nombre>
    <edad>15</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Manuel</nombre>
    <edad>16</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Andrés</nombre>
    <edad>12</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Julio</nombre>
    <edad>16</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>Antonio</nombre>
    <edad>14</edad>
  </DatosPersona>
  <DatosPersona>
    <nombre>María Jesús</nombre>
    <edad>13</edad>
  </DatosPersona>
</ListaPersonasMunicipio>
```

En primer lugar para utilizar **XStream,** simplemente creamos una instancia de la clase **XStream:**

```
XStream xstream = new XStream();
```

En general las etiquetas XML se corresponden con el nombre de los atributos de la clase, pero se pueden cambiar usando el método *alias()*. En el ejemplo se ha dado un alias a la clase *ListaPersonas*, en el XML aparecerá con el nombre *ListaPersonasMunicipio*.

```
xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
```

También se ha dado un alias a la clase *Persona*, en el XML aparecerá con el nombre *DatosPersona:*

```
xstream.alias("DatosPersona", Persona.class);
```

Para que no aparezca el atributo *lista* de la clase *ListaPersonas* en el XML generado se ha utilizado el método *addImplicitCollection():*

```
xstream.addImplicitCollection(ListaPersonas.class, "lista");
```

Por último, para generar el fichero *Personas.xml* a partir de la lista de objetos se utiliza el método *toXML(objeto, OutputStream):*

```
xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
```

El proceso para realizar la lectura del fichero XML generado es el siguiente:

```
import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
```



```
import java.util.List;
import com.thoughtworks.xstream.XStream;

public class LeerPersonas {
  public static void main(String[] args) throws IOException {

    XStream xstream = new XStream();

    xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
    xstream.alias("DatosPersona", Persona.class);
    xstream.addImplicitCollection(ListaPersonas.class, "lista");

    ListaPersonas listadoTodas = (ListaPersonas)
         xstream.fromXML(new FileInputStream("Personas.xml"));
    System.out.println("Numero de Personas: " +
         listadoTodas.getListaPersonas().size());

    List<Persona> listaPersonas = new ArrayList<Persona>();
    listaPersonas = listadoTodas.getListaPersonas();

    Iterator iterador = listaPersonas.listIterator(); //iterador los elementos
    while( iterador.hasNext() ) {
        Persona p = (Persona) iterador.next(); //Obtengo el elemento contenido
        System.out.println("Nombre: " + p.getNombre() +
                           ", edad: " + p.getEdad());
    }
    System.out.println("Fin de listado .....");
  } //fin main
}//fin LeerPersonas
```

Se deben utilizar los métodos *alias()* y *addImplicitCollection()* para leer el XML ya que se usaron para hacer la escritura del mismo. Para obtener el objeto con la lista de personas o lo que es lo mismo, para deserializar el objeto, utilizamos el método *fromXML(InputStream):*

```
ListaPersonas listadoTodas = (ListaPersonas)
     xstream.fromXML(new FileInputStream("Personas.xml"));
```

API XStream: http://xstream.codehaus.org/javadoc/com/thoughtworks/xstream/XStream.html

1.7.4 CONVERSIÓN DE FICHEROS XML A OTRO FORMATO

XSL *(Extensible Stylesheet Language)* es toda una familia de recomendaciones del *World Wide Web Consortium* (http://www.w3.org/Style/XSL/) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo **XSL** describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja, puede contener elementos de reglas que representan a las reglas de construcción y elementos de reglas de estilo que representan a las reglas de mezcla de estilos. En el siguiente ejemplo, vamos a ver cómo a partir de un fichero XML que contiene datos y otro **XSL** que contiene la presentación de esos datos se puede generar un fichero HTML usando el lenguaje Java. Los ficheros son los siguientes:

ALUMNOS.XML	ALUMNOSPLANTILLA.XSL
<?xml version="1.0"?> <listadealumnos> <alumno> <nombre>Juan</nombre> <edad>19</edad> </alumno> <alumno> <nombre>Maria</nombre> <edad>20</edad> </alumno> </listadealumnos	<?xml version="1.0" encoding='ISO-8859-1'?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:template match='/'> <html><xsl:apply-templates /></html> </xsl:template> <xsl:template match='listadealumnos'> <head><title>LISTADO DE ALUMNOS</title></head> <body> <h1>LISTA DE ALUMNOS</h1> <table border='1'> <tr><th>Nombre</th><th>Edad</th></tr> <xsl:apply-templates select='alumno' /> </table> </body> </xsl:template> <xsl:template match='alumno'> <tr><xsl:apply-templates /></tr> </xsl:template> <xsl:template match='nombre|edad'> <td><xsl:apply-templates /></td> </xsl:template> </xsl:stylesheet>

Para realizar la transformación se necesita obtener un objeto *Transformer* que se obtiene creando una instancia de *TransformerFactory* y aplicando el método *newTransformer* a la fuente **XSL** que vamos a utilizar para aplicar la transformación del fichero de datos XML, o lo que es lo mismo, para aplicar la hoja de estilos **XSL** al fichero XML:

```
Transformer transformer =
            TransformerFactory.newInstance().newTransformer(estilos);
```

La transformación se consigue llamando al método *transform()*, pasándole los datos (el fichero XML) y el stream de salida (el fichero HTML):

```
transformer.transform(datos, result);
```

El código es el siguiente:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class convertidor {
 public static void main(String argv[]) throws IOException{
  String hojaEstilo = "alumnosPlantilla.xsl";
  String datosAlumnos = "alumnos.xml";
  File pagHTML = new File("mipagina.html");
  FileOutputStream os = new FileOutputStream(pagHTML); //crear fichero HTML
```



```
  Source estilos =new StreamSource(hojaEstilo); //fuente XSL
  Source datos =new StreamSource(datosAlumnos); //fuente XML
  Result result = new StreamResult(os);         //resultado de la transformación

  try{
   Transformer transformer =
              TransformerFactory.newInstance().newTransformer(estilos);
   transformer.transform(datos, result);     //obtiene el HTML
  }
  catch(Exception e){System.err.println("Error: "+e);}

  os.close();  //cerrar fichero
 }//de main
}//de la clase
```

El fichero HTML generado se muestra en la Figura 1.3.

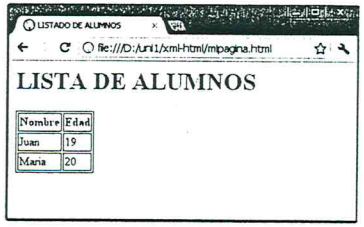


Figura 1.3. Fichero HTML generado.

API de Java: http://docs.oracle.com/javase/6/docs/api/index.html

1.8 EXCEPCIONES: DETECCIÓN Y TRATAMIENTO

Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Cuando no es capturada por el programa, se captura por el gestor de excepciones por defecto que retorna un mensaje y detiene el programa. La ejecución del siguiente programa produce una excepción y visualiza un mensaje indicando el error:

```
public class ejemploExcepcion {
public static void main(String[] args) {
    int nume=10, denom=0, cociente;
    cociente=nume/denom;
    System.out.println("Resultado:" +cociente);
 }//
}//

D:\unil>java ejemploExcepcion
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ejemploExcepcion.main(ejemploExcepcion.java:4)
```

Cuando dicho error ocurre dentro de un método Java, el método crea un objeto **Exception** y lo maneja fuera, en el sistema de ejecución. El manejo de excepciones en Java está diseñado pensando en situaciones en las que el método que detecta un error no es capaz de manejarlo, un método así *lanzará una excepción.*

Las excepciones en Java son objetos de clases derivadas de la clase base **Exception** que a su vez es una clase derivada de la clase base **Throwable.**

1.8.1 CAPTURAR EXCEPCIONES

Para capturar una excepción se utiliza el bloque **try-catch.** Se encierra en un bloque **try** el código que puede generar una excepción, este bloque va seguido por uno o más bloques **catch.** Cada bloque **catch** especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones. Después del último bloque **catch** puede aparecer un bloque **finally** (opcional) que siempre se ejecuta haya ocurrido o no la excepción; se utiliza el bloque **finally** para cerrar ficheros o liberar otros recursos del sistema después de que ocurra una excepción:

```
try {

    //Código que puede generar excepciones

} catch(excepcion1 e1) {

    //manejo de la excepcion1

} catch(excepcion2 e2) {

    //manejo de la excepcion2

}

//etc .........

finally {

    // Se ejecuta después de try o catch

}
```

El siguiente ejemplo, muestra la captura de 3 tipos de excepciones que se pueden producir. Cuando se encuentra el primer error se produce un salto al bloque **catch** que maneja dicho error; en este caso, al encontrar la sentencia de asignación *arraynum[10]=20;* se lanza la excepción *ArrayIndexOutOfBoundsException* (ya que el array está definido para 4 elementos y se da un valor al elemento de la posición 10) donde se ejecutan las instrucciones indicadas en el bloque, las sentencias situadas debajo de la que causó el error dentro del bloque **try** no se ejecutarán:

```
public class ejemploExcepciones {
public static void main(String[] args) {
    String cad1="20", cad2="0", mensaje;
    int nume, denom, cociente;
```



```
    int[] arraynum = new int[4];
    try {
        //código que puede producir errores
        arraynum[10]=20;                    //sentencia que produce la excepción
        nume=Integer.parseInt(cad1);        //no se ejecuta
        denom=Integer.parseInt(cad2);       //no se ejecuta
        cociente=nume/denom;                //no se ejecuta
        mensaje=String.valueOf(cociente);//no se ejecuta
    } catch(NumberFormatException ex){
        mensaje="Caracteres no numéricos";
    } catch(ArithmeticException ex){
        mensaje="Division por cero";
    } catch (ArrayIndexOutOfBoundsException ex) {
        mensaje="Fuera de rango en el array";
    } finally {
        System.out.println("SE EJECUTA SIEMPRE");
    }
    System.out.println(mensaje); //si se ejecuta
 }//fin de main
}//fin de la clase
```

El programa muestra la siguiente salida:

```
D:\uni1>java ejemploExcepciones
SE EJECUTA SIEMPRE
Fuera de rango en el array
```

Para capturar cualquier excepción utilizamos la clase base **Exception.** Si se usa habrá que ponerla al final de la lista de manejadores para evitar que los manejadores que vienen después queden ignorados, por ejemplo, el siguiente código maneja varias excepciones, si se produce alguna para la que no se ha definido manejador será capturada por **Exception:**

```
try {
        //código que puede producir errores
    } catch(NumberFormatException ex){
        //tratamiento excepción
    } catch(ArithmeticException ex){
        //tratamiento excepción
    } catch (ArrayIndexOutOfBoundsException ex) {
        //tratamiento excepción
    } catch (Exception ex) {
        //tratamiento si se produce cualquier otra excepción
    } finally {
        //se ejecuta haya o no excepción
    }
```

Para obtener más información sobre la excepción se puede llamar a los métodos de la clase base **Throwable,** algunos son:

- *String getMessage():* devuelve la cadena de error del objeto.
- *String getLocalizedMessage():* crea una descripción local de este objeto.

- *String toString():* devuelve una breve descripción del objeto.
- *void printStackTrace(), printStackTrace(PrintStream) o printStackTrace(PrintWriter):* imprime el objeto y la traza de pila de llamadas lanzada.

Por ejemplo, el siguiente bloque **try-catch:**

```
try {
      arraynum[10]=20;
      nume=Integer.parseInt(cad1);
      denom=Integer.parseInt(cad2);
      cociente=nume/denom;
      mensaje=String.valueOf(cociente);
} catch(Exception ex){
     System.err.println("toString            => "+ ex.toString());
     System.err.println("getMessage          => "+ ex.getMessage());
     System.err.println("getLocalizedMessage=> "+ ex.getLocalizedMessage());
     ex.printStackTrace();
} finally {
     System.out.println("SE EJECUTA SIEMPRE");
}
```

Muestra la siguiente salida al ejecutarse:

```
toString            => java.lang.ArrayIndexOutOfBoundsException: 10
getMessage          => 10
getLocalizedMessage=> 10
java.lang.ArrayIndexOutOfBoundsException: 10
        at ejemploExcepciones2.main(ejemploExcepciones2.java:8)
SE EJECUTA SIEMPRE
```

Una sentencia **try** puede estar dentro de un bloque de otra sentencia **try.** Si la sentencia **try** interna no tiene un manejador **catch,** se busca el manejador en las sentencias **try** más externas.

1.8.2 ESPECIFICAR EXCEPCIONES

Para especificar excepciones utilizamos la palabra clave **throws,** seguida de la lista de todos los tipos de excepciones potenciales; si un método decide no gestionar una excepción (mediante **try-catch**), debe especificar que puede lanzar esa excepción. El siguiente ejemplo indica que el método *main()* puede lanzar las excepciones *IOException* y *ClassNotFoundException:*

```
public static void main(String[] args) throws IOException,
                                              ClassNotFoundException {
```

Aquellos métodos que pueden lanzar excepciones, deben saber cuáles son esas excepciones en su declaración. Una forma típica de saberlo es compilando el programa. Por ejemplo, si al programa *EscribirPersonas.java* (visto en el epígrafe anterior) le quitamos la cláusula **throws** al método *main(),* al compilarlo aparecerán errores:

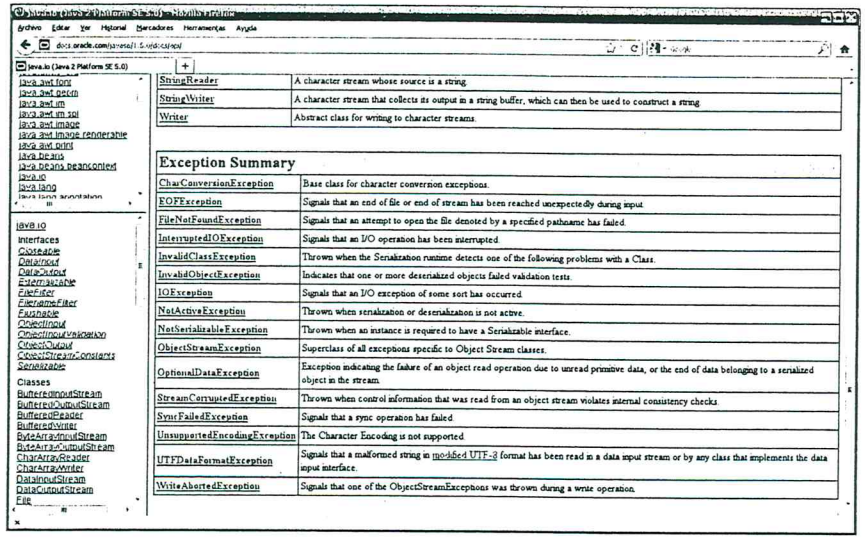

```
EscribirPersonas.java:7: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown  FileInputStream filein = new
FileInputStream(fichero);//crea el flujo de entrada
EscribirPersonas.java:9:unreported exception java.io.IOException; must be caught
or declared to be thrownObjectInputStream dataIS = new ObjectInputStream(filein);
  ......
EscribirPersonas.java:17: unreported exception java.lang.ClassNotFoundException;
must be caught or declared to be thrown   Persona persona= (Persona) dataIS.
readObject(); //leer una Persona
```

Indica que en la línea 7 se produce una excepción que no se ha declarado *(FileNotFoundException)* esta excepción debe ser capturada mediante un bloque **try-catch** o declarada para ser lanzada mediante **throws.** También se producen errores de excepciones no declaradas en las líneas 9 y 17.

El ejemplo anterior *(EscribirPersonas.java)* manejando las excepciones dentro de bloques **try-catch** quedaría:

```
public class EscribirPersonas2 {
  public static void main(String[] args)  {
    File fichero = new File("FichPersona.dat");
    FileInputStream filein;
    try {
      filein = new FileInputStream(fichero);//crea el flujo de entrada
      ObjectInputStream dataIS = new ObjectInputStream(filein);
      System.out.println("Comienza el proceso de creación del fichero a XML");
      ListaPersonas listaper = new ListaPersonas();

      try {
          while (true) { //lectura del fichero
            Persona persona= (Persona) dataIS.readObject(); //leer una Persona
            listaper.add(persona); //añadir persona a la lista
          }//del while
       } catch (EOFException eo) {//fin de fichero bo hago nada
       } catch (ClassNotFoundException  cn) {
            cn.printStackTrace();
      }//final bloque try interno

      dataIS.close();  //cerrar stream de entrada

      XStream xstream = new XStream();
      xstream.alias("ListaPersonasMunicipio", ListaPersonas.class);
      xstream.alias("DatosPersona", Persona.class);
      xstream.addImplicitCollection(ListaPersonas.class, "lista");
      xstream.toXML(listaper, new FileOutputStream("Personas.xml"));
      System.out.println("Creado fichero XML....");

    } catch (IOException io) {
        io.printStackTrace();
    } catch (Exception e)  {
          e.printStackTrace();
    }//final bloque try mas externo
  } // fin main
} //fin EscribirPersonas
```

Podemos consultar la API de Java http://docs.oracle.com/javase/6/docs/api/ para ver las excepciones que se pueden producir en los diferentes paquetes. La Figura 1.4 muestra las excepciones que pueden producir los métodos del paquete **java.io.**



StringReader	A character stream whose source is a string.
StringWriter	A character stream that collects its output in a string buffer, which can then be used to construct a string.
Writer	Abstract class for writing to character streams.

Exception Summary

CharConversionException	Base class for character conversion exceptions.
EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input.
FileNotFoundException	Signals that an attempt to open the file denoted by a specified pathname has failed.
InterruptedIOException	Signals that an I/O operation has been interrupted.
InvalidClassException	Thrown when the Serialization runtime detects one of the following problems with a Class.
InvalidObjectException	Indicates that one or more deserialized objects failed validation tests.
IOException	Signals that an I/O exception of some sort has occurred.
NotActiveException	Thrown when serialization or deserialization is not active.
NotSerializableException	Thrown when an instance is required to have a Serializable interface.
ObjectStreamException	Superclass of all exceptions specific to Object Stream classes.
OptionalDataException	Exception indicating the failure of an object read operation due to unread primitive data, or the end of data belonging to a serialized object in the stream.
StreamCorruptedException	Thrown when control information that was read from an object stream violates internal consistency checks.
SyncFailedException	Signals that a sync operation has failed.
UnsupportedEncodingException	The Character Encoding is not supported.
UTFDataFormatException	Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by any class that implements the data input interface.
WriteAbortedException	Signals that one of the ObjectStreamExceptions was thrown during a write operation.

Figura 1.4. Excepciones paquete java.io.

1.9 EJERCICIOS

❶ Realiza un programa Java que cree un fichero binario para guardar datos de departamentos, dale el nombre *Departamentos.dat*. Introduce varios departamentos. Los datos por cada departamento son:

Número de departamento: entero.

Nombre: String.

Localidad: String.

❷ Realiza un programa Java que te permita modificar los datos de un departamento. El programa recibe desde la línea de comandos el número de departamento a modificar, el nuevo nombre de departamento y la nueva localidad. Si el departamento no existe visualiza un mensaje indicándolo. Visualiza también los datos antiguos del departamento y los nuevos datos.

❸ Realiza un programa Java que te permita eliminar un departamento. El programa recibe desde la línea de comandos el número de departamento a eliminar. Si el departamento no existe visualiza un mensaje indicándolo. Visualiza también el número total de departamentos que existen en el fichero.

ejercicio 4 - pagina 45

❹ Realiza un programa que copie dos ficheros. El nombre de los dos ficheros, origen y destino, se introducen desde la línea de comandos al ejecutar el programa.

Nuevo.txt
copia.txt

Pasar como argumentos

❺ Cuáles de las siguientes afirmaciones sobre **SAX** y **DOM** son correctas:

a) Los procesadores **SAX y DOM** son independientes del lenguaje de programación y existen versiones particulares para Java, VisualBasic, C, etc.

b) Mediante **SAX** se carga todo el fichero XML en memoria y se lee de forma secuencial produciendo una secuencia de eventos.

c) Mediante **DOM** se almacena toda la estructura del documento XML en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales.

d) **SAX** es más complejo de programar que **DOM.**

e) El tipo de procesamiento de **SAX** necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.

f) El tipo de procesamiento de **DOM** impide tener una visión global del documento por el que navegar.

❻ A partir de los datos del fichero *Departamentos.dat* creado anteriormente crea un fichero XML usando **DOM.**

❼ A partir de los datos del fichero *Departamentos.dat* creado anteriormente crea un fichero XML usando la librería **XStream.**

❽ Crea una plantilla XSL para dar una presentación al fichero XML generado por el ejercicio anterior y realiza un programa Java para transformarlo en HTML.

❾ Señala la respuesta correcta sobre las excepciones:

a) Una sentencia **try** no puede estar dentro de un bloque de otra sentencia **try.**

b) Un bloque **try** va seguido por un único bloque **catch.**

c) Cada bloque **catch** especifica el tipo de excepción que puede atrapar y contiene un manejador de excepciones.

d) El bloque **finally** se ejecuta si no ocurre la excepción.

UNIDAD 2

MANEJO DE CONECTORES

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Instalar y utilizar bases de datos embebidas.
- Utilizar conectores para acceder a bases de datos.
- Establecer conexiones a bases de datos.
- Desarrollar aplicaciones para acceder a los datos de la base de datos.
- Ejecutar procedimientos de bases de datos.
- Definir modelos para comunicar con la base de datos con el patrón Modelo-Vista-Controlador.

CONTENIDOS

2.1 INTRODUCCIÓN
2.2 EL DESFASE OBJETO-RELACIONAL
2.3 BASES DE DATOS EMBEBIDAS
2.3.1 SQLite
2.3.2 Apache Derby
2.3.3 HSQLDB
2.3.4 H2
2.3.5 Db4o
2.3.6 Otras
2.4 PROTOCOLOS DE ACCESO A BASES DE DATOS
2.5 ACCESO A DATOS MEDIANTE ODBC
2.6 ACCESO A DATOS MEDIANTE JDBC
2.6.1 Arquitecturas JDBC
2.6.2 Tipos de drivers
2.6.3 Cómo funciona JDBC
2.6.4 Acceso a datos mediante el Puente JDBC-ODBC
2.7 ESTABLECIMIENTO DE CONEXIONES
2.8 EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS
2.8.1 ResultSetMetaData
2.9 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS
2.9.1. Sentencias preparadas
2.10 EJECUCIÓN DE PROCEDIMIENTOS
2.11 GESTIÓN DE ERRORES
2.12 PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS
2.13 EJERCICIOS

2.1 INTRODUCCIÓN

En general el término de acceso a datos significa el proceso de recuperación o manipulación de datos extraídos de un origen de datos local o remoto. Los orígenes de datos no tienen por qué ser relacionales y pueden provenir de muchas fuentes distintas. Algunos de los orígenes de datos con los que podemos encontrarnos son: una base de datos relacional remota en un servidor, una base de datos relacional local, una hoja de cálculo, un fichero de texto en nuestro ordenador, un servicio de información online, etc.

En esta unidad nos centraremos en los orígenes de datos relacionales, aprenderemos a realizar programas Java para acceder a una base de datos relacional. Para ello necesitaremos los **conectores**, que no son más que el software que se necesita para realizar las conexiones desde nuestro programa Java con una base de datos relacional.

2.2 EL DESFASE OBJETO-RELACIONAL

Actualmente las bases de datos orientadas a objetos están ganando cada vez más aceptación frente a las bases de datos relacionales, ya que solucionan las necesidades de aplicaciones más sofisticadas que requieren el tratamiento de elementos más complejos: Un ejemplo de estas son las aplicaciones para diseño y fabricación en ingeniería, los sistemas de información geográfica (GIS), experimentos científicos, aplicaciones multimedia, etc. Dentro de estas nuevas aplicaciones se definen las orientadas a objetos (OO) y en general el *Paradigma de Programación Orientada a Objetos* (POO) cuyos elementos complejos (nombrados anteriormente) son los Objetos.

En este sentido, las bases de datos relacionales no están diseñadas para almacenar estos objetos ya que existe un desfase entre las construcciones típicas que proporciona el modelo de datos relacional y las proporcionadas por los ambientes de programación basados en objetos; es decir, al guardar los datos de un programa bajo el enfoque orientado a objetos se incrementa la complejidad del programa dando lugar a más código y más esfuerzo de programación debido a la diferencia de esquemas entre los elementos a almacenar (objetos) y las características del repositorio de la base de datos (tablas). A esto es a lo que se denomina **desfase objeto-relacional** (o *desajuste de la impedancia*) y se refiere a los problemas que ocurren debido a las diferencias entre el modelo de datos de la base de datos y el del lenguaje de programación orientado a objetos.

Sin embargo, el paradigma relacional y el paradigma orientado a objetos pueden ser "amigos". Cada vez que los objetos deben extraerse o almacenarse en una base de datos relacional se requiere un mapeo desde las estructuras provistas en el modelo de datos a las provistas por el entorno de programación. Este tema se trata más ampliamente en la Unidad 3.

2.3 BASES DE DATOS EMBEBIDAS

Cuando desarrollamos pequeñas aplicaciones en las que no vamos a almacenar grandes cantidades de información no es necesario que utilicemos un sistema gestor de base de datos como Oracle

o MySQL. En su lugar podemos utilizar una base de datos embebida donde el motor esté incrustado en la aplicación y sea exclusivo para ella. La base de datos se inicia cuando se ejecuta la aplicación y termina cuando se cierra la aplicación.

Por lo general, este tipo de bases de datos vienen del movimiento Open Source, aunque también hay algunas de origen propietario. Veamos algunas de ellas.

2.3.1 SQLITE

SQLite es un sistema gestor de base de datos multiplataforma escrito en C que proporciona un motor muy ligero. Las bases de datos se guardan en forma de ficheros por lo que es fácil trasladar la base de datos con la aplicación que la usa. Cuenta con una utilidad que nos permitirá ejecutar comandos SQL contra una base de datos SQLite en modo consola. Es un proyecto de dominio público.

La biblioteca implementa la mayor parte del estándar SQL-92, incluyendo transacciones de base de datos atómicas, consistencia de base de datos, aislamiento y durabilidad, triggers (o disparadores) y la mayor parte de las consultas complejas. Los programas que utilizan la funcionalidad de SQLite lo hacen a través de llamadas simples a subrutinas y funciones. SQLite se puede utilizar desde programas en C/C++, PHP, Visual Basic, Perl, Delphi, Java, etc.

Su instalación es sencilla. Desde la página http://www.sqlite.org/download.html se puede descargar. Para sistemas Windows podemos descargar el fichero ZIP *sqlite-shell-win32-x86-3070900.zip*, al descomprimirlo obtenemos un único fichero ejecutable *(sqlite3.exe)*. Al ejecutarlo desde la línea de comandos escribimos el nombre del fichero que contendrá la base de datos, si el fichero no existe se creará, si existe cargará la base de datos. El siguiente ejemplo crea la base de datos *ejemplo.db* (en la carpeta *D:\db\SQlite*), todas las tablas que creemos en esta sesión se almacenarán en este fichero, para finalizar se escribe el comando *.quit:*

```
D:\>sqlite3 D:\db\SQlite\ejemplo.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> BEGIN TRANSACTION;
sqlite> CREATE TABLE departamentos (
   ...>  dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
   ...>  dnombre  VARCHAR(15),
   ...>  loc      VARCHAR(15)
   ...> );
sqlite>
sqlite>
sqlite> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
sqlite> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
sqlite> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
sqlite> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
sqlite> COMMIT;
sqlite> .quit
```

Para instalarlo desde Linux escribimos desde la línea de comandos: `$sudo apt-get install sqlite3`. Y para ejecutarlo escribimos lo siguiente para crear la base de datos en la carpeta */home/usuario/db/SQLite:*

```
$ sqlite3 /home/usuario/db/SQLite/ejemplo.db
```

Actividad 1: Crea las tablas EMPLEADOS y DEPARTAMENTOS en SQLite e inserta filas en ellas. La descripción de las tablas es la siguiente:

DEPARTAMENTOS:	EMPLEADOS:
DEPT_NO numérico, clave primaria DNOMBRE VARCHAR(15), LOC VARCHAR(15)	EMP_NO numérico, clave primaria APELLIDO VARCHAR(10), OFICIO VARCHAR(10), DIR numérico, FECHA_ALT DATE, SALARIO numérico, COMISION numérico, DEPT_NO numérico, clave ajena, referencia a DEPARTAMENTOS

2.3.2 APACHE DERBY

Apache Derby, es una base de datos relacional de código abierto, implementado en su totalidad en Java que forma parte del *Apache DB subproject* y está disponible bajo la licencia Apache, versión 2.0. Algunas ventajas de esta base de datos son: su tamaño reducido, está basada en Java y soporta los estándares SQL, ofrece un controlador integrado JDBC que permite incrustar Derby en cualquier solución basada en Java, soporta el tradicional paradigma cliente-servidor utilizando el *Derby Network Server*, es fácil de instalar, implementar y utilizar.

Para realizar la instalación descargamos la última versión desde la página web: http://db.apache.org/derby/derby_downloads.html. En Windows descargamos el fichero: *db-derby-10.8.2.2-bin.zip*, y lo descomprimimos por ejemplo en *D:\db-derby-10.8.2.2-bin*. A partir de ahora, para poder utilizar Derby en nuestros programas Java, solo será necesario tener accesible la librería *derby.jar* en el CLASSPATH de nuestro programa.

Ejemplos para configuar la variable CLASSPATH:
Windows:

```
C:\> SET DERBY_INSTALL=D:\db-derby-10.8.2.2-bin
C:\> SET CLASSPATH=DERBY_INSTALL\lib\derby.jar;DERBY_INSTALL\lib\derbytools.jar;.
```

Linux:

```
$ export DERBY_INSTALL=/opt/db-derby-10.8.2.2-bin
$ export CLASSPATH=$DERBY_INSTALL/lib/derby.jar:$DERBY_INSTALL/lib/derbytools.jar:.
```

Para visualizar la variable CLASSPATH:
Windows:

```
C:\> echo %CLASSPATH%
```

Linux:

```
$ $CLASSPATH
```


Apache Derby trae una serie de ficheros .BAT que nos permitirán ejecutar por consola órdenes para crear nuestras bases de datos y ejecutar sentencias DDL y DML. El fichero es **IJ.BAT** y se encuentra en la carpeta bin *(D:\db-derby-10.8.2.2-bin\bin)*. Desde la línea de comandos del DOS nos dirigimos a dicha carpeta y ejecutamos el fichero **IJ.BAT:**

```
D:\db-derby-10.8.2.2-bin\bin>ij
```

Para crear una base de datos de nombre *ejemplo* en la carpeta *D:\db\Derby* escribimos desde el indicador **ij>** la siguiente orden:

```
connect 'jdbc:derby:D:\db\Derby\ejemplo;create=true';
```

Donde:

- *connect*, es el comando para establecer la conexión.
- *jdbc:derby*, es el protocolo JDBC especificado por DERBY.
- *ejemplo* es el nombre de la base de datos que voy a crear (se crea una carpeta con dicho nombre y dentro una serie de ficheros).
- *create=true*, atributo usado para crear la base de datos.

Para salir de la línea de comandos de IJ, escribimos `exit;`

El siguiente ejemplo muestra la creación de la base de datos *ejemplo*, la creación de la tabla DEPARTAMENTOS, la inserción de filas en la tabla y la ejecución del script *Empleados.sql* que crea la tabla EMPLEADOS e inserta filas en ella. Al final se ejecuta la orden *exit* para salir:

```
D:\db-derby-10.8.2.2-bin\bin>ij
Versi¾n ij 10.8
ij> connect 'jdbc:derby:D:\db\Derby\ejemplo;create=true';
ij> CREATE TABLE departamentos (
> dept_no  INT NOT NULL PRIMARY KEY,
> dnombre  VARCHAR(15),
> loc      VARCHAR(15)
> );
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO departamentos VALUES (10,'CONTABILIDAD','SEVILLA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (20,'INVESTIGACIÓN','MADRID');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (30,'VENTAS','BARCELONA');
1 fila insertada/actualizada/suprimida
ij> INSERT INTO departamentos VALUES (40,'PRODUCCIÓN','BILBAO');
1 fila insertada/actualizada/suprimida
ij> commit;
ij> run 'Empleados.sql';
ij> CREATE TABLE empleados (
 emp_no    INT  NOT NULL PRIMARY KEY,
 apellido  VARCHAR(10),
 oficio    VARCHAR(10),
 dir       INT,
```

```
 fecha_alt DATE,
 salario   FLOAT,
 comision  FLOAT,
 dept_no   INT NOT NULL REFERENCES departamentos(dept_no)
);
0 filas insertadas/actualizadas/suprimidas
ij> INSERT INTO empleados VALUES (7369,'SANCHEZ','EMPLEADO',7902,'1990-12-
17',1040,NULL,20);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7499,'ARROYO','VENDEDOR',7698,'1990-02-20',
1500,390,30);
1 fila insertada/actualizada/suprimida
ij> exit;
```

Para volver a usar la base de datos escribiríamos la siguiente orden desde la línea de comandos de IJ: *connect 'jdbc:derby:D:\db\Derby\ejemplo';*.

Para realizar la instalación en Linux (Ubuntu) seguimos los siguientes pasos. En primer lugar es necesario instalar los paquetes JDK y JAVADB de Java (si no los tenemos instalados) ejecutando la siguiente orden desde la línea de comandos:

```
sudo apt-get install sun-java6-jdk sun-java6-plugin sun-java6-javadb
```

En algunas versiones de Linux (Ubuntu) puede no encontrar el paquete debido a un cambio de licencias de Oracle, en ese caso antes hay que añadir un nuevo repositorio y después actualizar el sistema:

```
sudo add-apt-repository ppa:ferramroberto/java
sudo apt-get update
```

A continuación descargamos la versión para Linux y la descomprimimos en la carpeta /opt, en este caso se ha descargado la versión: *db-derby-10.8.2.2-bin.tar.gz*. Se creará la carpeta */opt/db-derby-10.8.2.2-bin*. Configuramos la variable DERBY_INSTALL y DERBY_HOME con el nombre de carpeta donde se ha descargado, ejecutando desde la línea de comandos:

```
$ export DERBY_INSTALL=/opt/db-derby-10.8.2.2-bin
$ export DERBY_HOME=/opt/db-derby-10.8.2.2-bin
```

Para usar Derby necesitamos incluir en el CLASSPATH los ficheros JAR: *derby.jar* y *derbytools.jar*, escribimos las siguientes órdenes:

```
$ export CLASSPATH=$DERBY_INSTALL/lib/derby.jar:$DERBY_INSTALL/lib/derbytools.jar:.
```

A continuación nos dirigimos a la carpeta donde está instalado Derby y ejecutamos *setEmbeddedCP*:

```
$ cd $DERBY_INSTALL/bin
$ ./setEmbeddedCP
```


Desde aquí ya podemos utilizar la utilidad IJ para crear nuestra base de datos escribiendo desde la línea de comandos: `java org.apache.derby.tools.ij`. Lo primero que se visualiza es la versión. El siguiente ejemplo muestra la creación de la base de datos *ejemplo* en la carpeta */home/usuario/db/Derby/*, una vez creada finalizamos la conexión ejecutando la orden *exit*:

```
$ java org.apache.derby.tools.ij
Versión ij 10.8
ij> connect 'jdbc:derby:/home/usuario/db/Derby/ejemplo;create=true';
Mon Jan 02 21:37:24 CET 2012 Thread[main,5,main] java.io.FileNotFoundException: derby.
log (Permission denied)
----------------------------------------------------------------
Mon Jan 02 21:37:25 CET 2012:
 Arrancando Derby versión The Apache Software Foundation - Apache Derby - 10.8.2.2 -
(1181258): instancia a816c00e-0134-a024-0bfb-000000be33d0
en el directorio de base de datos /home/usuario/db/Derby/ejemplo  con el cargador de
clases sun.misc.Launcher$AppClassLoader@7d772e

java.vendor=Sun Microsystems Inc.
java.runtime.version=1.6.0_21-b06
user.dir=/opt/db-derby-10.8.2.2-bin/bin
derby.system.home=null

Cargador de clases de base de datos iniciado - derby.database.classpath=''
ij> exit;
----------------------------------------------------------------
Mon Jan 02 21:40:55 CET 2012: cerrando motor de Derby
----------------------------------------------------------------
Mon Jan 02 21:40:57 CET 2012:
Cerrando la instancia a816c00e-0134-a024-0bfb-000000be33d0 en el directorio de
base de datos /home/usuario/db/Derby/ejemplo con cargador de clases sun.misc.
Launcher$AppClassLoader@7d772e
----------------------------------------------------------------
$
```

Para volver a usar la base de datos escribiríamos la siguiente orden desde la línea de comandos de IJ: *connect 'jdbc:derby:/home/usuario/db/Derby/ejemplo';*. El resto de comandos SQL se usan igual que se usaron en el ejemplo para Windows.

Actividad 2: Crea las tablas EMPLEADOS y DEPARTAMENTOS en Apache Derby e inserta filas en ellas.

2.3.3 HSQLDB

HSQLDB (*Hyperthreaded Structured Query Language Database*) es un sistema gestor de bases de datos relacional escrito en Java. La suite ofimática OpenOffice lo incluye desde su versión 2.0 para dar soporte a la aplicación *Base*. Es compatible con SQL ANSI-92 y SQL: 2008. Puede mantener la base de datos en memoria o en ficheros en disco. Permite integridad referencial (claves foráneas),

procedimientos almacenados en Java, disparadores y tablas en disco de hasta 8GB. Se distribuye con licencia BSD (*Berkeley Software Distribution*) que es una licencia muy cercana al dominio público.

Desde la web http://sourceforge.net/projects/hsqldb/files/ podemos descargarnos la última versión. En Windows se ha descargado el fichero *hsqldb-2.2.6.zip*. Lo descomprimimos por ejemplo en la unidad D, se creará una carpeta con el nombre del fichero ZIP (*D:\hsqldb-2.2.6\hsqldb*), dentro de esa carpeta hay una con el nombre **hsqldb**, la llevamos a la unidad D para que nos quede instalado en *D:\hsqldb*. A continuación creamos una carpeta en *D:\hsqldb\data* para guardar los datos de la base de datos que vamos a crear, la llamamos *ejemplo*, nos debe quedar: *D:\hsqldb\data\ejemplo*.

Abrimos la línea de comandos del DOS y nos dirigimos a la carpeta *D:\hsqldb\bin*, ejecutamos el fichero BAT de nombre *runUtil* con el parámetro *DatabaseManager*, para conectarnos a la base de datos y que se ejecute la interfaz gráfica de este sistema gestor de base datos:

```
D:\hsqldb\bin>runUtil DatabaseManager
```

Se abre una ventana desde la que tenemos que configurar la conexión, escribimos en el campo *Setting Name* un nombre para la conexión, de la lista *Type* seleccionamos la opción *HSQL Database Engine Standalone*, para que la base de datos la tome de un fichero si existe y si no existe la cree; y en la casilla de *URL*, escribimos el nombre de la carpeta donde se almacenará la base de datos y el de la base de datos: *ejemplo/ejemplo*. Pulsamos el botón *OK*, véase Figura 2.1.

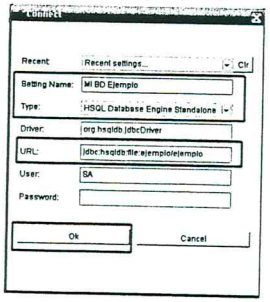


Figura 2.1. Configurar conexión en HSQLDB.

A continuación se abre una nueva ventana desde la que podemos ejecutar comandos DDL y DML para crear y manipular objetos de nuestra base de datos, véase Figura 2.2. Para ejecutar una sentencia SQL pulsamos el botón *Execute*. Desde la opción de menú *View->Refresh Tree* podemos actualizar el árbol de objetos.

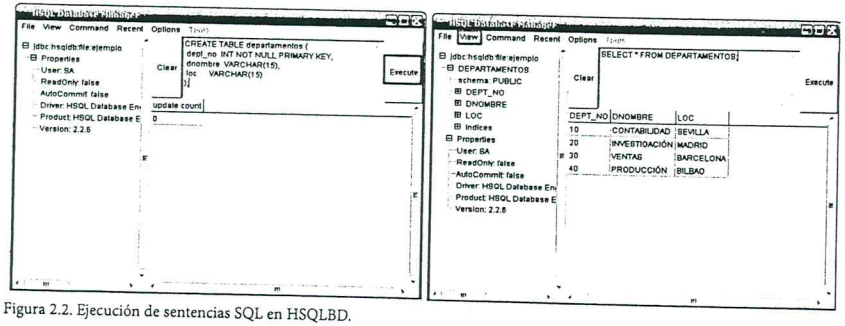


Figura 2.2. Ejecución de sentencias SQL en HSQLBD.

Para instalarlo en Ubuntu primero guardamos el fichero *hsqldb-2.2.6.zip* en la carpeta */opt*. Después desde la línea de comandos nos vamos a dicha carpeta y lo descomprimimos ejecutando la siguiente orden:

```
$ sudo unzip hsqldb-2.2.6.zip
```

Se creará en *opt* una carpeta con nombre *hsqldb-2.2.6*, dentro hay otra carpeta que se llama *hsqldb*, la movemos a */opt*. Al final nos debe quedar */opt/hsqldb*. A continuación escribo desde la línea de comandos las siguientes órdenes para establecer el PATH con las librerías de hsqldb en la variable CLASSPATH:

```
$ CLASSPATH="$CLASSPATH:/opt/hsqldb/lib/hsqldb.jar"
$ export CLASSPATH
```

Creamos en la carpeta *home/usuario/db/Hsqldb* la carpeta *ejemplo* (nos debe quedar */home/usuario/db/Hsqldb/ejemplo*) para almacenar todos los datos de la base de datos *ejemplo* que crearemos a continuación. Seguidamente ejecutamos desde la línea de comandos la siguiente orden:

```
$ java org.hsqldb.util.DatabaseManager
```

Se abre la ventana de conexión, similar a la mostrada en la Figura 2.1. Rellenamos los campos como se hizo anteriormente. En el campo URL escribimos la carpeta donde se almacenará la base de datos y su nombre, nos debe quedar: *jdbc:hsqldb:file:/home/usuario/db/Hsqldb/ejemplo*. El resto de operaciones son similares.

Actividad 3: Crea las tablas EMPLEADOS y DEPARTAMENTOS en HSQLDB e inserta filas en ellas.

2.3.4 H2

H2 es un sistema gestor de base de datos relacional programado integramente en Java. Está disponible como software de código libre bajo la *Licencia Pública de Mozilla* o la *Eclipse Public License*. Desde la web http://sourceforge.net/projects/hsqldb/files/ podemos descargarnos la última versión. Para probarla descargamos la versión ZIP para todas las plataformas *h2-2011-11-26.zip*. En Windows lo descomprimimos por ejemplo en la unidad D se creará una carpeta con el nombre *D:/h2*. Desde la línea de comandos de Windows nos dirigimos a la carpeta *D:\h2\bin* y ejecutamos el fichero **h2.bat** para arrancar la consola:

```
D:\h2\bin>h2
```

Se abre el navegador web con la consola de administración de H2, véase Figura 2.3. Escribimos un nombre para la configuración de la base de datos, en el campo URL JDBC escribimos la URL para la conexión a nuestra base de datos: *jdbc:h2:D:/db/H2/ejemplo/ejemplo* (si las carpetas no existen las crea), pulsamos el botón *Guardar* para que guarde la configuración y a continuación el botón

Conectar. Cada vez que queramos conectarnos a nuestra base de datos usaremos el nombre dado a la configuración. Podemos pulsar el botón *Probar la conexión* antes de conectar para ver si todo ha ido bien, entonces se mostrará el mensaje: *Prueba correcta.* Se creará la carpeta *ejemplo* en *H2* y dentro los ficheros de nuestra base de datos.

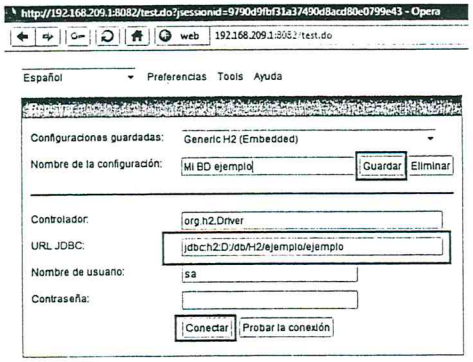


Figura 2.3. Establecer conexión en H2.

Una vez conectados se visualiza una nueva pantalla, Figura 2.4, desde la que podremos realizar las operaciones sobre la base de datos. Se muestran los comandos más importantes y un script de ejemplo. Desde la zona de instrucciones SQL podremos escribir las sentencias para crear tablas, insertar filas, etc., véase Figura 2.5.

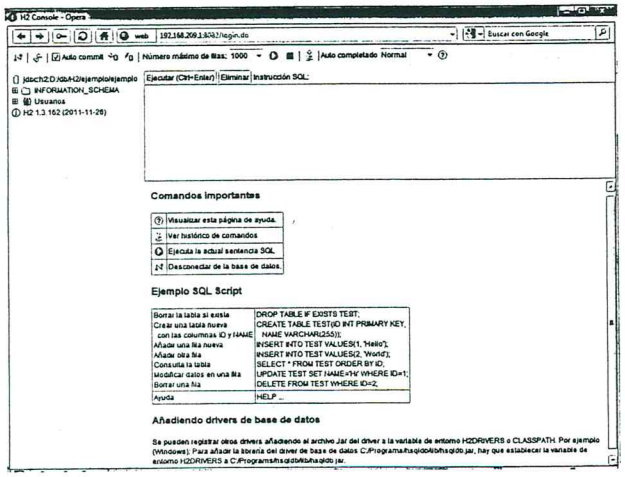


Figura 2.4. Pantalla de manejo de la base de datos en H2.

Los botones Ejecutar (Ctrl+Enter) y nos permitirán ejecutar las sentencia SQL que escribamos en el área de instrucción SQL. El botón *Desconectar* nos lleva a la pantalla inicial donde elegimos la conexión.

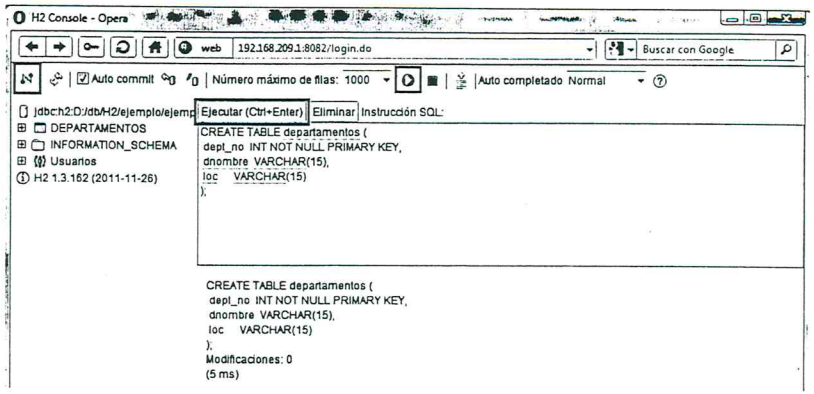


Figura 2.5. Ejecución de sentencias en H2.

Desde el enlace *Preferencias* de la ventana inicial se pueden configurar diversos aspectos como: los clientes permitidos (locales/remotos), conexión segura (uso de SSL), puerto del servidor Web o notificar las sesiones activas. La opción *Tools* presenta una serie de herramientas que se pueden utilizar sobre la base de datos: backup, restaurar base de datos, ejecutar scripts, convertir la base de datos en un script, encriptación, etc.

En Linux Ubuntu extraemos el fichero *h2-2011-11-26.zip* en la carpeta *opt* de esta manera tendremos */opt/h2*. Nos dirigimos a la carpeta */opt/h2/bin* y ejecutamos el fichero **h2.sh** para arrancar la consola (también se puede ejecutar desde el entorno gráfico), desde la línea de comandos escribiríamos:

```
/opt/h2/bin$ ./h2.sh
```

Si se visualiza el mensaje de permiso denegado ejecutamos las siguientes órdenes para dar permiso de ejecución y después para ejecutar el fichero **h2.sh**:

```
/opt/h2/bin$ sudo chmod +x h2.sh
/opt/h2/bin$ ./h2.sh
```

El resto de pasos son similares a los vistos anteriormente, salvo la escritura de la URL. En el campo URL JDBC podemos escribir lo siguiente *jdbc:h2:/home/usuario/db/H2/ejemplo/ejemplo*, para que nos guarde la base de datos en la carpeta */home/usuario/db/H2/ejemplo*, se crearían las carpetas, si no existen, y dentro los ficheros para la base de datos de nombre *ejemplo*.

Actividad 4: Crea las tablas EMPLEADOS y DEPARTAMENTOS en H2 e inserta filas en ellas.

2.3.5 DB4O

Db4o (*DataBase 4 (for) Objects*) es un motor de base de datos orientado a objetos. Se puede utilizar de forma embebida o en aplicaciones cliente-servidor. Está disponible para entornos Java y .Net. Dispone de licencia dual GPL/comercial. Proporciona algunas características interesantes:

- Se evita el problema del desfase objeto-relacional.
- No existe un lenguaje SQL para la manipulación de datos, en su lugar existen métodos delegados.
- Se instala añadiendo un único fichero de librería (JAR para Java o DLL para .NET).
- Se crea un único fichero de base de datos con la extensión .YAP.

Para utilizar Db4o nos dirigimos a la web http://www.db4o.com/, descargamos la última versión y la descomprimimos. Para el ejemplo se ha descargado la versión *db4o-8.0-java.zip*. Al descomprimirla hay una carpeta de nombre *lib* donde se encuentra los JAR (*db4o-.jar*) que tenemos que agregar a nuestro proyecto para utilizar el motor de la base de datos.

Desde Eclipse podemos crear una librería con todos los JAR para ello seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Paths-> Add Libreries.** Se visualiza una ventana desde la que hemos de elegir la opción **User Library** y pulsar el botón *Next*. A continuación se visualiza una nueva ventana desde la que pulsamos el botón *User Libraries*, a continuación pulsamos el botón *New* para dar un nombre a la librería, por ejemplo *Db4oLib*. Seguidamente pulsamos el botón *Add JARs*, localizamos todos los *db4o-.jar* de la carpeta *lib* y pulsamos el botón *Abrir*, véase Figura 2.6. A continuación *OK* y para finalizar pulsamos *Finish*. Aunque no es necesario añadir todos los JAR, bastaría con añadir el JAR *db4o-8.0.224.15975-core-java5.jar*.

La librería quedará integrada en nuestro proyecto Eclipse. La instalación en Linux es similar. Si no usamos entorno gráfico para nuestros programas Java hemos de incluir en la variable CLASSPATH los JAR necesarios.

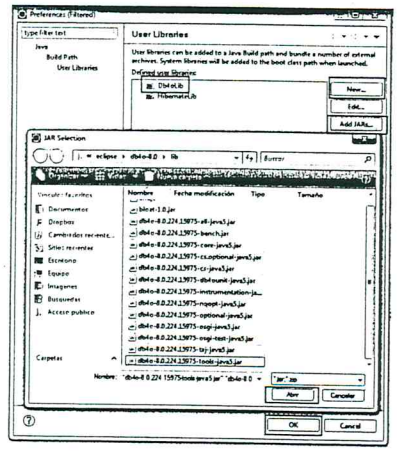


Figura 2.6. Añadir librerías Db4o en Eclipse.

El siguiente ejemplo muestra un programa Java que crea crea una base de datos y almacena objetos Persona en ella. Se ha definido la clase **Persona.java** formada por los atributos *nombre* y *ciudad* y los métodos *get* y *set* para obtener y almacenar los valores de un registro Persona:

```
public class Persona {
  private String nombre;
  private String ciudad;

  public Persona(String nombre,String ciudad)   {
  this.nombre=nombre;
  this.ciudad=ciudad;
  }
  public Persona() {
  this.nombre=null;
  this.ciudad=null;
  }
  public String getNombre(){return nombre;}
  public void setNombre(String nom){nombre=nom;}

  public String getCiudad(){return ciudad;      }
  public void setCiudad(String dir){ciudad=dir;}

}//fin Persona
```

Desde Eclipse para generar automáticamente los *getters* y los *setters* de los atributos pulsamos con el botón derecho del ratón en el código de la clase, seleccionamos la opción *Source* y a continuación *Generate Getters and Setters*, véase Figura 2.7. A continuación hemos de seleccionar los atributos y pulsar el botón *OK*.

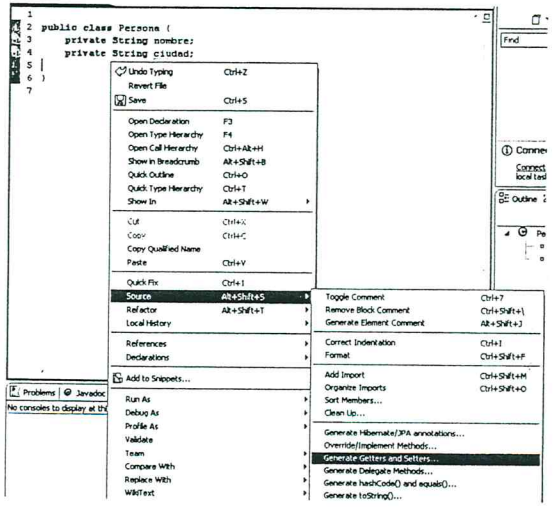


Figura 2.7. Generar *Getters* y *Setters*.

El siguiente código muestra la clase **Main.java** que crea la base de datos (si no existe) e inserta objetos Persona en ella:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Main {
     final static String BDPer = "D:/eclipse/bd_db4o/DBPersonas.yap";

     public static void main(String[] args) {
             ObjectContainer db= Db4oEmbedded.openFile
                                 (Db4oEmbedded.newConfiguration(),BDPer);

             // Creamos Personas
             Persona p1 = new Persona("Juan", "Guadalajara");
             Persona p2 = new Persona("Ana", "Madrid");
             Persona p3 = new Persona("Luis", "Granada");
             Persona p4 = new Persona("Pedro", "Asturias");

             //Almacenar objetos Persona en la base de datos
             db.store(p1);
             db.store(p2);
             db.store(p3);
             db.store(p4);

             db.close();          //cerrar base de datos

     }//fin de main
}//fin de la clase Main
```

Para realizar cualquier acción, ya sea insertar, modificar o realizar consultas debemos manipular una instancia de ***ObjectContainer*** donde se define el fichero de base de datos, en el ejemplo el fichero se llama *DBPersonas.yap* y se almacena en la variable *BDPer* donde será necesario incluir el trayecto donde se encuentra el fichero. Algunos de los métodos más importantes son:

- Para abrir la base de datos llamamos al método ***openFile( )***:

```
ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),BDPer);
```

- Para cerrarla llamamos a ***close()***:

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

- Para almacenar objetos utilizamos ***store()***:

```
db.store(p1);
```

- Para recuperar objetos podemos utilizar la interfaz Query-By-Example ***queryByExample()***. El siguiente ejemplo muestra todos los objetos Persona existentes en la base de datos, si no existe ninguno el método **size()** aplicado al objeto **ObjectSet** devuelve 0:

```
Persona per = new Persona(null,null);
ObjectSet<Persona> result = db.queryByExample(per);
if(result.size() == 0) System.out.println("No existen Registros de Personas..");
       else{
```



```
            System.out.println("Número de registros: "+result.size());

            while(result.hasNext()){
               Persona p = result.next();
               System.out.println("Nombre:" + p.getNombre()+
                                  ", Ciudad:" + p.getCiudad());
            }
       }
       db.close();    //cerrar base de datos
```

El siguiente ejemplo obtendría los objetos Persona cuyo nombre es Juan:

```
Persona per = new Persona("Juan",null);
ObjectSet<Persona> result = db.queryByExample(per);
```

El siguiente ejemplo obtendría los objetos Persona cuya ciudad es Guadalajara:

```
Persona per = new Persona (null,"Guadalajara");
ObjectSet<Persona> result = db.queryByExample(per);
```

- Para modificar un objeto primero hay que localizarlo y después se modifica con ***store()***. El siguiente ejemplo modifica la ciudad de Juan a Toledo y luego visualiza sus datos:

```
ObjectSet<Persona> result = db.queryByExample(new Persona("Juan",null));
if(result.size() == 0) System.out.println("No existe Juan…");
else{
          Persona existe = (Persona) result.next();
          existe.setCiudad("Toledo");
          db.store(existe); //ciudad modificada
          //consultar los datos
          result = db.queryByExample(new Persona("Juan",null));
          existe = (Persona) result.next();
          System.out.println("Nombre:" +existe.getNombre()+
                        " Nueva Ciudad: "+ existe.getCiudad());
   }
```

- Para eliminar objetos utilizamos ***delete()***, antes será necesario localizar el objeto a eliminar. El siguiente ejemplo elimina todos los objetos cuyo nombre sea Juan:

```
ObjectSet<Persona> result = db.queryByExample(new Persona("Juan",null));
if(result.size() == 0) System.out.println("No existe Juan…");
else {
   Persona existe = (Persona) result.next();
   System.out.println("Registros a borrar: "+ result.size());
   if (result.size()>1) {  //varios objetos con nombre Juan
          while(result.hasNext()){
              Persona p = result.next();
              db.delete(p);
              System.out.println("Borrado...." );
          }
   } else     db.delete(existe);     //solo hay un objeto con nombre Juan
}
```

En Linux cambiaríamos la localización del fichero, por ejemplo:

```
final static String BDPer = "/home/usuario/Java/Db4o/DBPersonas.yap";
```

Si queremos ejecutar desde la línea de comandos los programas Java tendríamos que modificar la variable CLASSPATH. En el siguiente ejemplo los ficheros Java están en la carpeta */home/usuario/Java/Db4o*, dentro está la carpeta *lib* que contiene los JAR de Db4o, añadiremos al CLASSPATH el fichero *db4o-8.0.224.15975-core-java5.jar*, después se compilarían los ficheros y al final se ejecutaría Main:

```
/Java/Db4o$ export CLASSPATH=$CLASSPATH:/home/usuario/Java/Db4o/lib/db4o-8.0.224.15975-
core-java5.jar
/Java/Db4o$ javac Persona.java
/Java/Db4o$ javac Main.java
/Java/Db4o$ java Main
```

En la Unidad 4 se profundizará más acerca de las bases de datos orientadas a objetos.

Actividad 5: Crea una base de datos Db4o de nombre EMPLEDEP.YAP e inserta objetos EMPLEADOS y DEPARTAMENTOS en ella. Después obtén todos los objetos empleado de un departamento concreto. Visualiza también el nombre de dicho departamento.

2.3.6 OTRAS

Existen más sistemas de bases de datos embebidos tanto en software libre como en sistemas propietarios. Algunos ejemplos son:

- **Firebird** que se deriva del código fuente de *InterBase 6.0*, de *Borland*. Es un sistema gestor de bases de datos relacional de código abierto que no tiene licencias duales, por lo que es totalmente libre y se puede usar tanto en aplicaciones comerciales como de código abierto. Se presenta en tres versiones de servidor: *SuperServer, Classic* y *Embedded*. La edición embebida (*Embedded*) es un completo servidor Firebird empacado en unos cuantos ficheros. Es fácil distribuir aplicaciones, puesto que no requiere instalación, ideal para crear catálogos en CDROM, versiones mono usuario, de evaluación o portátiles de las aplicaciones. Algunas de sus características son:
  - Completo soporte para procedimientos almacenados y disparadores.
  - Integridad referencial.
  - Bajo consumo de recursos.
  - Lenguaje interno para procedimientos almacenados y disparadores (PSQL) .
  - Soporte para funciones externas.
  - Poca o ninguna necesidad de administradores especializados.
  - Múltiples formas de acceder a la base de datos: nativo/API, drivers dbExpress, ODBC, OLEDB, proveedor .Net, driver JDBC nativo tipo 4, módulo Python, PHP, Perl, etc.
  - Etc.

- **SQL Server Mobile** (*Microsoft SQL Server 2005 Mobile Edition*): Es una base de datos compacta y con una gran variedad de funciones diseñada para admitir una amplia lista de dispositivos inteligentes y Tablet PC. Es un producto de Microsoft que incluye varias características de las bases de datos relacionales a la vez que ocupa poco espacio. Algunas características son:
  - Un motor de base de datos compacto y un sólido optimizador de consultas.
  - Compatibilidad con la réplica de mezcla y el acceso a datos remotos.
  - Integración con Microsoft SQL Server 2005.
  - Las herramientas de administración son Microsoft SQL Server Management Studio y SQL Server Management Studio Express.
  - Integración con Microsoft Visual Studio 2005.
  - Acceso a datos remotos y réplica de mezcla para sincronizar datos.
  - Microsoft Proveedor de datos .NET Framework y .NET Compact Framework para SQL Server Compact Edition (System.Data.SqlServerCe).
  - Compatibilidad con Microsoft ADO.NET y el proveedor de OLE DB para SQL Server Compact Edition.
  - Un subconjunto de sintaxis SQL.
  - Se implementa como una base de datos incrustada en equipos de escritorio, dispositivos móviles y Tablet PC.
  - Compatibilidad con la tecnología de implementación ClickOnce.

- **Oracle Embedded**: bajo este nombre Oracle ofrece un conjunto de soluciones al desarrollo software que aporta ciertas ventajas como la instalación "silenciosa" (se instala solo sin tener que hacer clic), la configuración automática o el funcionamiento desatendido y menores costes de implementación, licencia, hardware y administración. La tecnología Embedded de Oracle ofrece elementos especializados y una gran libertad de elección según el tipo de necesidades. La oferta de bases de datos permite elegir entre tres elementos:
  - ***Oracle Database 11g***: para aplicaciones empresariales.
  - ***Oracle TimesTen In-Memory Database:*** para aplicaciones en tiempo real que necesitan un tiempo de respuesta de microsegundos. *TimesTen* es una base de datos relacional que se ejecuta en memoria y que gracias a ello ofrece unos altísimos tiempos de respuesta y una caché de datos en tiempo real. Está especialmente diseñada para entornos de misión crítica.
  - ***Oracle Berkeley DB:*** proporciona a los desarrolladores una base de datos embebida permitiendo varias opciones de almacenamiento: clave/valor, SQL, XML / XQuery o Java Object para su modelo de datos. Entre sus características cabe destacar que se puede ejecutar en disco o en memoria y proporciona un alto rendimiento. Algunas características de Oracle Berkeley DB son:
    - Rendimiento extraordinario, elimina los gastos de la comunicación interprocesos y SQL.
    - Administración cero, toda la administración se realiza a través de una API, se integra por completo en la aplicación y es invisible para los usuarios finales.
    - En dispositivos móviles, el tamaño de las librerías es de menos de 1MB y el tiempo de ejecución de los requisitos de memoria dinámica es de solo unos pocos KB.
    - Bajo coste total de propiedad, al ser embebida se reducen los gastos de implementación, licencias y hardware, y los costes de administración.

El siguiente ejemplo en Java muestra el uso de una base de datos Berkeley para almacenar pares clave/valor. Para poder ejecutarlo hemos de agregar el fichero *je-5.0.34.jar* a nuestro proyecto Eclipse o a nuestro CLASSPATH, este se puede conseguir descargando la versión ZIP de *Berkeley DB Java Edition 5.0.34* (*je-5.0.34.zip*) desde la URL de Oracle: http://www.oracle.com/technetwork/database/berkeleydb/downloads/index.html. El fichero se encuentra en la carpeta *lib*. Primero será necesario crear un entorno de base de datos y a continuación la base de datos. En el ejemplo la base de datos se llama NUMEROS y se almacena en la carpeta *D:/je-5.0.34/ejemplo*:

```
import java.io.File;
import java.io.UnsupportedEncodingException;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.Transaction;

public class Main {
 public static void main(String[] args) throws UnsupportedEncodingException {

     String envDir = "D:/je-5.0.34/ejemplo";//carpeta donde se almacenará la BD

     // Creamos un entorno de base de datos transaccional
     EnvironmentConfig envConfig = new EnvironmentConfig();
     envConfig.setAllowCreate(true);
     envConfig.setTransactional(true);
     Environment env = new Environment(new File(envDir), envConfig);

    //Crear una base de datos transaccional para ese entorno
     Transaction tx = env.beginTransaction(null, null);
     DatabaseConfig dbConfig = new DatabaseConfig();
     dbConfig.setTransactional(true);
     dbConfig.setAllowCreate(true);
     dbConfig.setSortedDuplicates(true);
     Database myDb = env.openDatabase(null, "NUMEROS", dbConfig);

     // DatabaseEntry representa los pares clave valor de cada registro
     DatabaseEntry clave =new DatabaseEntry();
     DatabaseEntry valor =new DatabaseEntry();

     //introducimos pares clave/valor
     clave = new DatabaseEntry("1".getBytes("UTF-8"));
     valor = new DatabaseEntry("uno".getBytes("UTF-8"));
     myDb.put(tx, clave, valor);

     clave = new DatabaseEntry("2".getBytes("UTF-8"));
     valor = new DatabaseEntry("dos".getBytes("UTF-8"));
     myDb.put(tx, clave, valor);
```



```
     clave = new DatabaseEntry("3".getBytes("UTF-8"));
     valor = new DatabaseEntry("tres".getBytes("UTF-8"));
     myDb.put(tx, clave, valor);
     tx.commit();

     String laclave;
     String elvalor;
     //Cursor para recorrer todos los pares Clave/Valor
     Cursor cursor = myDb.openCursor(null, null);

     while (cursor.getNext(clave, valor, LockMode.DEFAULT) ==
              OperationStatus.SUCCESS) {
         laclave = new String(clave.getData(), "UTF-8");
         elvalor = new String(valor.getData(), "UTF-8");
         System.out.println("Clave = " + laclave +  " Valor = " + elvalor);
     }

     cursor.close();    //cerramos cursor
     myDb.close();      //cerramos BD
     env.close();       //cerramos entorno

    }//fin de main

}//fin de la clase
```

La ejecución del programa visualiza la siguiente información:

```
Clave = 1 Valor = uno
Clave = 2 Valor = dos
Clave = 3 Valor = tres
```

Para probarlo en Linux cambiaríamos la carpeta donde se almacenará la base de datos (que tiene que existir), por ejemplo:

```
String envDir = "/home/usuario/db/Berkeley/ejemplo";
```

Después desde la línea de comandos definimos en el CLASSPATH donde se encuentra el fichero JAR, por ejemplo, si está en la carpeta */home/usuario/Java/Berkeley* escribimos:

```
$ export CLASSPATH=$CLASSPATH:/home/usuario/Java/Berkeley/je-5.0.34.jar
```

Compilamos y ejecutamos (suponemos que nuestro programa Java está en la misma carpeta):

```
Java/Berkeley$ javac Main.java
Java/Berkeley$ java Main
Clave = 1 Valor = uno
Clave = 2 Valor = dos
Clave = 3 Valor = tres
Java/Berkeley$
```

2.4 PROTOCOLOS DE ACCESO A BASES DE DATOS

En tecnologías de base de datos podemos encontrarnos con dos normas de conexión a una base de datos SQL:

- **ODBC** (*Open Database Connectivity*): define una API (*Application Program Interface-Interfaz para Programas de Aplicación*) que pueden usar las aplicaciones para abrir una conexión con una base de datos, enviar consultas, actualizaciones y obtener los resultados. Las aplicaciones pueden usar esta API para conectarse a cualquier servidor de base de datos compatible con ODBC.
- **JDBC** (*Java Database Connectivity - Conectividad de base de Datos con Java*) define una API que pueden usar los programas Java para conectarse a los servidores de bases de datos relacionales.

Hay muchos orígenes de datos que no son bases de datos relacionales, algunos puede que ni si quiera sean bases de datos, tal es el caso de los ficheros planos y los almacenes de correo electrónico. **OLE-DB** (*Object Linking and Embedding for Databases-Enlace e incrustación de objetos para bases de datos*) de Microsoft es una API de C++ con objetivos parecidos a los de ODBC pero para orígenes de datos que no son bases de datos. OLE-DB proporciona estructuras para la conexión con orígenes de datos, ejecución de comandos y devolución de resultados en forma de conjunto de filas. Sin embargo, se diferencia de ODBC en algunos aspectos. Los programas OLE-DB pueden negociar con los orígenes de datos para averiguar las interfaces que soportan. En ODBC los comandos siempre están en SQL, en OLE-DB pueden estar en cualquier lenguaje soportado por el origen de datos, puede que algunos orígenes soporten SQL o un subconjunto limitado de SQL y otros ofrezcan el acceso a los datos de los ficheros planos sin ninguna capacidad de consulta.

La API **ADO** (*Active Data Objects – Objetos activos de datos*) creada por Microsoft ofrece una interfaz sencilla de utilizar con la funcionalidad OLE-DB, que puede llamarse desde los lenguajes de guiones como VBScript y JScript.

2.5 ACCESO A DATOS MEDIANTE ODBC

ODBC es un estándar de acceso a bases de datos desarrollado por *Microsoft Corporation* con el objetivo de posibilitar el acceso a cualquier dato desde cualquier aplicación, sin importar que sistema gestor de bases de datos almacene los datos. Cada sistema de base de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API ODBC el código de la biblioteca se comunica con el servidor para realizar la acción solicitada y obtener los resultados.

Los pasos para usar ODBC son:

1. El primer paso es configurar la interfaz ODBC, para ello el programa asigna en primer lugar un entorno SQL con la función *SQLAllocHandle()*, después un manejador (o handle) para la conexión a la base de datos basada en el entorno anterior, el propósito de este manejador es traducir las consultas de datos de la aplicación en comandos que el sistema de base de datos entienda. ODBC define varios tipos de manejadores:

   - SQLHENV: define el entorno de acceso a los datos.
   - SQLHDBC: identifica el estado y configuración de la conexión (driver y origen de datos).
   - SQLHSTMT: declaración SQL y cualquier conjunto de resultados asociados.
   - SQLHDESC: recolección de metadatos utilizados para describir una sentencia SQL.
2. Una vez reservados los manejadores el programa abre la conexión a la base de datos usando *SQLDriverConnect()* o *SQLConnect()*.
3. Una vez realizada la conexión el programa puede enviar órdenes SQL a la base de datos usando *SQLExecDirect()*.
4. Al final de la sesión el programa se desconecta de la base de datos y libera la conexión y los manejadores del entorno SQL.

La API ODBC usa una interface escrita en el lenguaje C y no es apropiada para su uso directo desde Java. Las llamadas desde Java a código C nativo tienen un número de inconvenientes en la seguridad, implementación, robustez y portabilidad de las aplicaciones. ODBC es duro de aprender. Mezcla características elementales con otras más avanzadas y tiene complejas opciones incluso para las consultas más simples.

Algunas funciones importantes son:

- *SQLAllocHandle, SQLDriverConnect*: necesarias para establecer la conexión con la base de datos.
- *SQLAllocStmt, SQLExecDirect*: ejecutan sentencias SQL sobre la base de datos.
- *SQLFetch, SQLGetData, SQLNumResultCols, SQLRowCount*: obtienen datos de la consulta SQL, como los resultados de una consulta, número de filas o de columnas.
- *SQLDisconnect, SQLFreeHandle*: operaciones de limpieza de memoria y desconexión a la base de datos.

El siguiente programa C accede mediante ODBC a una base de datos MySQL llamada *ejemplo* que tiene creadas las tablas EMPLEADOS y DEPARTAMENTOS; el usuario y la clave del mismo tienen el mismo nombre de la base de datos. Para poder acceder necesitamos crear un origen de datos ODBC. Desde Linux (Ubuntu) hemos de instalar los paquetes *unixodbc*, *unixodbc-dev* y *libmyodbc* y crear el origen de datos, por ejemplo ***Mysql-odbc***. Al instalarse los paquetes la librería *libmyodbc.so* se instala en */usr/lib/odbc*:

```
# apt-get install unixodbc unixodbc-dev libmyodbc
```

Editamos el fichero *odbcinst.ini* de la carpeta */etc/*, si no existe lo creamos y escribimos las siguientes líneas donde indicamos dónde se encuentra el **driver** ODBC:

```
# gedit /etc/odbcinst.ini
[Mysql]
Description     = Mysql Connector 3.51
Driver          = /usr/lib/odbc/libmyodbc.so
UsageCount      = 1
CPTimeout       =
CPReuse         =
```

Editamos el fichero *odbc.ini* de la carpeta */etc/*, si no existe lo creamos y escribimos las siguientes líneas para crear el origen de datos ODBC de nombre ***Mysql-odbc*** en el que definimos el driver, la base de datos y el usuario con su clave para acceder a dicha base de datos:

```
# gedit /etc/odbc.ini
[ODBC Data Sources]
Mysql-odbc    = MyODBC 3.51 Driver DSN

[Mysql-odbc]
Driver        = Mysql
Description   = Base de datos MySQL ejemplo
SERVER        = 127.0.0.1
PORT          = 3306
USER          = ejemplo
Password      = ejemplo
Database      = ejemplo
OPTION        = 3
SOCKET        =
```

El código del programa C es el siguiente, necesitamos las librerías *<sql.h>* y *<sqlext.h>*:

```
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

int main() {
  SQLHENV env;
  SQLHDBC dbc;
  SQLHSTMT stmt;
  SQLRETURN ret;

//Definir el entorno
  SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
  SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void *) SQL_OV_ODBC3, 0);
  SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

//conectarse al origen de datos Mysql-odbc
  ret = SQLDriverConnect(dbc, NULL, "DSN=Mysql-odbc;", SQL_NTS,
          NULL, 0, NULL, SQL_DRIVER_COMPLETE);

  char *consulta ="SELECT * FROM departamentos";
  SQLSMALLINT nCols = 0;
  SQLINTEGER nFilas = 0;
  SQLINTEGER nIdicator = 0;
  SQLCHAR buf[1024] = {0};

  if (SQL_SUCCEEDED(ret)) {
      printf("Conectado\n");
      SQLAllocStmt(dbc,&stmt );
      ret =SQLExecDirect( stmt,consulta, SQL_NTS );
      SQLNumResultCols( stmt, &nCols );
      SQLRowCount( stmt, &nRows );
      printf("* Número de Columnas: %u\n", nCols );
```



```
      printf("* Número de Filas: %u \n", nFilas );

      while( SQL_SUCCEEDED( ret = SQLFetch( stmt ) ) )
      {
       printf("\n");
       for( int i=1; i <= nCols; ++i )
       {
         ret = SQLGetData( stmt,i,SQL_C_CHAR,buf,1024,&nIdicator );
         if( SQL_SUCCEEDED( ret ) )
         {
            printf("* Columna %s", buf );
         }
       }
      } // while
      printf("\n");
      SQLFreeHandle( SQL_HANDLE_STMT, stmt );
      SQLDisconnect( dbc );

  } else { printf("NO HA SIDO POSIBLE LA CONEXIÓN\n"); }
return 1;
} // fin main
```

Lo compilamos con el compilador **gcc** cargando la librería *-lodbc* y con la opción *-std=gnu99*, puede que aparezca algún error Warning; y luego lo ejecutamos:

```
$ gcc conexionODBC.c -lodbc -std=gnu99
uno.c: In function 'main':
uno.c:113: warning: format '%u' expects type 'unsigned int', but argument 2 has type
'SQLINTEGER'
usuario@ubuntu2012:~$ ./a.out
Conectado
* Número de Columnas: 3
* Número de Filas: 4

* Columna 10* Columna CONTABILIDAD* Columna SEVILLA
* Columna 20* Columna INVESTIGACIÓN* Columna MADRID
* Columna 30* Columna VENTAS* Columna BARCELONA
* Columna 40* Columna PRODUCCIÓN* Columna BILBAO

$
```

2.6 ACCESO A DATOS MEDIANTE JDBC

JDBC proporciona una librería estándar para acceder a fuentes de datos principalmente orientados a bases de datos relacionales que usan SQL. No solo provee una interfaz 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. JDBC dispone de una interfaz distinta para cada base de datos (Figura 2.8), es lo que llamamos **driver** (controlador o conector). Esto permite que las llamadas a los métodos Java de las clases JDBC se correspondan con el API de la base de datos.

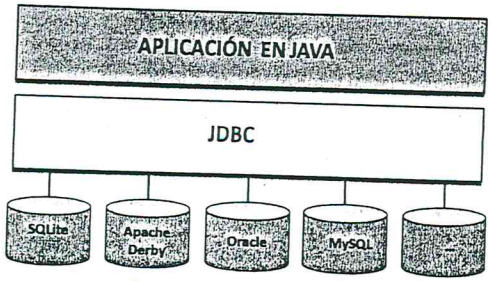


Figura 2.8. Acceso mediante JDBC.

JDBC consta de un conjunto de clases e interfaces que nos permite escribir aplicaciones Java para gestionar las siguientes tareas con una base de datos relacional:

- Conectarse a la base de datos.
- Enviar consultas e instrucciones de actualización a la base de datos.
- Recuperar y procesar los resultados recibidos de la base de datos en respuesta a las consultas.

2.6.1 ARQUITECTURAS JDBC

La API JDBC es compatible con los modelos tanto de dos como de tres capas para el acceso a la base de datos. En el **modelo de dos capas**, un applet o una aplicación Java "hablan" directamente con la base de datos, esto requiere un driver JDBC residiendo en el mismo lugar que la aplicación (Figura 2.9). Desde el programa Java se envían sentencias SQL al sistema gestor de base de datos para que las procese y los resultados se envian de vuelta al programa. La base de datos puede encontrarse en otra máquina diferente a la de la aplicación y las solicitudes se hacen a través de la red (arquitectura cliente-servidor). El driver será el encargado de manejar la comunicación a través de la red de forma transparente al programa.

En el **modelo de tres capas,** los comandos se envían a una capa intermedia que se encargará de enviar los comandos SQL a la base de datos y de recoger los resultados de la ejecución de las sentencias. Es decir, tenemos una aplicación o applet corriendo en una máquina y accediendo a un driver de base de datos situado en otra máquina, Figura 2.10. En este caso los drivers no tienen que residir en la máquina cliente.

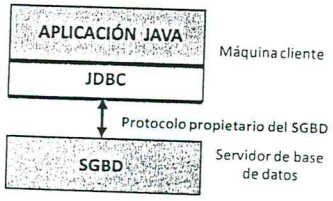


Figura 2.9. Arquitectura JDBC de dos capas.

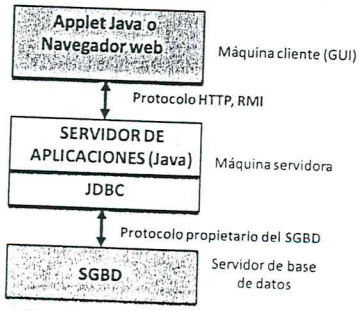


Figura 2.10. Arquitectura JDBC de tres capas.

2.6.2 TIPOS DE DRIVERS

Existen 4 tipos de conectores (drivers o controladores) JDBC:

- **Tipo 1. JDBC-ODBC Bridge** (*JDBC-ODBC bridge plus ODBC driver*): permite el acceso a bases de datos JDBC mediante un driver ODBC. Exige la instalación y configuración de ODBC en la máquina cliente.
- **Tipo 2. Native** (*Native-API partly-Java driver*): controlador escrito parcialmente en Java y en código nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del motor de base de datos. Exige instalar en la máquina cliente código binario propio del cliente de base de datos y del sistema operativo.
- **Tipo 3. Network** (*JDBC-Net pure Java driver*): controlador de Java puro que utiliza un protocolo de red (por ejemplo HTTP) para comunicarse con un servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red. No exige instalación en cliente.
- **Tipo 4. Thin** (*Native-protocol pure Java driver*): controlador de Java puro con protocolo nativo. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red usado por el motor de base de datos. No exige instalación en cliente.

Los tipos 3 y 4 son la mejor forma para acceder a bases de datos JDBC. Los tipos 1 y 2 se usan normalmente cuando no queda otro remedio, porque el único sistema de acceso final al gestor de bases de datos es ODBC (es decir, no existen drivers disponibles para el SGBD); pero exigen instalación de software en el puesto cliente. En la mayoría de los casos la opción más adecuada será el tipo 4.

2.6.3 CÓMO FUNCIONA JDBC

JDBC define varias interfaces que permite realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes. Estas están definidas en el paquete **java.sql**. La siguiente tabla muestra las clases e interfaces más importantes:

CLASE E INTERFACE	DESCRIPCIÓN
`Driver`	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
`DriverManager`	Permite gestionar todos los drivers instalados en el sistema.
`DriverPropertyInfo`	Proporciona diversa información acerca de un driver.
`Connection`	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión.
`DatabaseMetadata`	Proporciona información acerca de una Base de Datos, como las tablas que contiene, etc.
`Statement`	Permite ejecutar sentencias SQL sin parámetros.
`PreparedStatement`	Permite ejecutar sentencias SQL con parámetros de entrada.
`CallableStatement`	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
`ResultSet`	Contiene las filas resultantes de ejecutar una orden SELECT.
`ResultSetMetadata`	Permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.
http://docs.oracle.com/javase/6/docs/api/java/sql/package-summary.html	

La Figura 2.11 muestra las 4 clases principales que usa cualquier programa Java con JDBC. El trabajo con JDBC comienza con la clase *DriverManager* que es la encargada de establecer las conexiones con los orígenes de datos a través de los drivers JDBC. El funcionamiento de un programa con JDBC requiere los siguientes pasos:

1. Importar las clases necesarias.
2. Cargar el driver JDBC.
3. Identificar el origen de datos.
4. Crear un objeto *Connection*.
5. Crear un objeto *Statement*.
6. Ejecutar una consulta con el objeto *Statement*.
7. Recuperar los datos del objeto *ResultSet*.
8. Liberar el objeto *ResultSet*.
9. Liberar el objeto *Statement*.
10. Liberar el objeto *Connection*.

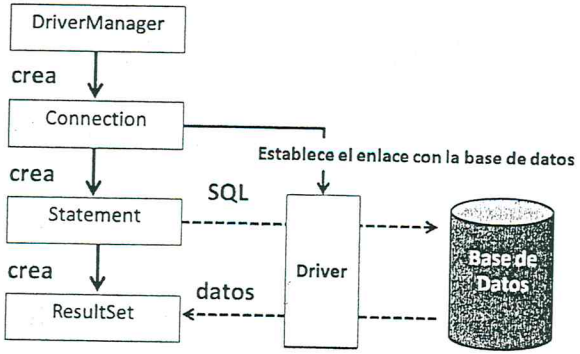


Figura 2.11. Funcionamiento de JDBC.

Para el siguiente ejemplo JAVA creamos desde MySQL una base de datos y un usuario con nombre **EJEMPLO,** la clave del usuario es la misma. Este usuario tendrá todos los privilegios sobre esta base de datos. A continuación creamos las siguientes tablas e insertamos datos en ellas, las relaciones se muestran en la Figura 2.12 (si ya tenemos creada la BD y las tablas no es necesario realizar estos pasos):

```
CREATE TABLE departamentos (
 dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
 dnombre  VARCHAR(15),
 loc      VARCHAR(15)
);

CREATE TABLE empleados (
 emp_no    SMALLINT(4) UNSIGNED NOT NULL PRIMARY KEY,
 apellido  VARCHAR(10),
```



```
 oficio    VARCHAR(10),
 dir       SMALLINT,
 fecha_alt DATE,
 salario   FLOAT(6,2),
 comision  FLOAT(6,2),
 dept_no   TINYINT(2) NOT NULL REFERENCES departamentos(dept_no)
);
```

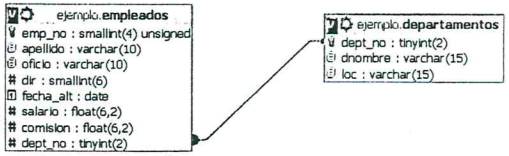


Figura 2.12- Base de datos EJEMPLO.

El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos EJEMPLO:

```
import java.sql.*;
public class Main {
     public static void main(String[] args) {
     try
     {
     // Cargar el driver
     Class.forName("com.mysql.jdbc.Driver");
     // Establecemos la conexion con la BD
     Connection conexion = DriverManager.getConnection
     ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");

     // Preparamos la consulta
     Statement sentencia = conexion.createStatement();
     ResultSet resul = sentencia.executeQuery ("SELECT * FROM departamentos");

     // Recorremos el resultado para visualizar cada fila
     // Se hace un bucle mientras haya registros, se van visualizando
     while (resul.next())
       {
        System.out.println (resul.getInt(1) + " " + resul.getString(2)+ " " +
        resul.getString(3));
       }

     resul.close();// Cerrar ResultSet
     sentencia.close();// Cerrar Statement
     conexion.close();//Cerrar conexion

       }
     catch (ClassNotFoundException cn) {cn.printStackTrace();}
     catch (SQLException e) {e.printStackTrace();}

}//fin de main
}//fin de la clase
```

Visualiza el contenido de la tabla DEPARTAMENTOS de la base de datos EJEMPLO de MySQL:

```
10 CONTABILIDAD SEVILLA
20 INVESTIGACIÓN MADRID
30 VENTAS BARCELONA
40 PRODUCCIÓN BILBAO
```

Para poder probar el programa hemos de obtener el JAR que contiene el driver MySQL (en el ejemplo se ha utilizado *mysql-connector-java-5.1.18-bin.jar*) e incluirlo en el CLASSPATH o añadirlo a nuestro IDE, por ejemplo en Eclipse pulsamos en el proyecto con el botón derecho del ratón y seleccionamos **Build Paths-> Add External Archives...** para localizar el fichero JAR. Desde la URL http://www.mysql.com/products/connector/ se puede descargar el conector. Si ejecutamos el programa desde la línea de comandos hemos de asegurarnos que el lugar donde se encuentra el fichero JAR se encuentre definido en la variable CLASSPATH.

Se puede observar que en nuestro programa Java, todos los *import* que necesitamos para manejar la base de datos están en **java.sql.***. También se ha incluido todo el programa en un **try-catch** ya que casi todos los métodos relativos a base de datos pueden lanzar la excepción **SQLException**. La llamada al método *forName()* para cargar el driver puede lanzar la excepción **ClassNotFoundException** si este no se encuentra.

Cargar el driver:

En primer lugar se carga el driver, con el método *forName* de la clase *Class*, se le pasa un objeto String con el nombre de la clase del driver como argumento. En el ejemplo como se accede a una base de datos MySQL necesitamos cargar el driver *com.mysql.jdbc.Driver*:

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

Establecer la conexión:

A continuación se establece la conexión con la base de datos, el servidor MySQL debe estar arrancado, usamos la clase *DriverManager* con el método *getConnection()*:

```
Connection conexion = DriverManager.getConnection
        ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
```

El primer parámetro del método *getConnection()* representa la URL de conexión a la base de datos:

- *jdbc:mysql* indica que estamos utilizando un driver JDBC para MySQL.
- *localhost* indica que el servidor de base de datos está en la mima máquina en la que se ejecuta el programa Java. Aquí puede ponerse una IP o un nombre de máquina que esté en la red.
- *ejemplo* es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL.

El segundo parámetro es el nombre de usuario que accede a la base de datos, en este caso se llama *ejemplo*.

El tercer parámetro es la clave del usuario, que en este caso también es *ejemplo*.

Ejecutar sentencias SQL:

A continuación se realiza la consulta, para ello recurrimos a la interfaz *Statement* para crear una sentencia. Para obtener un objeto *Statement* se llama al método *createStatement()* de un objeto *Connection* válido. La sentencia obtenida (o el objeto obtenido) tiene el método *executeQuery()* que sirve para realizar una consulta a la base de datos, se le pasa un String en el que está la consulta SQL, en el ejemplo "*SELECT * FROM departamentos*":

```
Statement sentencia = conexion.createStatement();
ResultSet resul = sentencia.executeQuery ("SELECT * FROM departamentos");
```

El resultado nos lo devuelve como un *ResultSet*, que es un objeto similar a una lista en la que está el resultado de la consulta. Cada elemento de la lista es uno de los registros de tabla DEPARTAMENTOS. *ResultSet* no contiene todos los datos, sino que los va consiguiendo de la base de datos según se van pidiendo. Por ello, el método *executeQuery()* puede tardar poco, pero recorrer todos los elementos del *ResultSet* puede no ser tan rápido.

Resultset tiene internamente un puntero que apunta al primer registro de la lista. Mediante el método *next()* el puntero avanza al siguiente registro. Para recorrer la lista de registros usaremos dicho método dentro de un bucle while que se ejecutará mientras *next()* devuelva true (es decir, mientras haya registros):

```
while (resul.next())
        {
         System.out.println (resul.getInt(1) + " " + resul.getString(2)+ " " +
           resul.getString(3));
        }
```

Los métodos *getInt()* y *getString()* nos van devolviendo los valores de los campos de dicho registro. Entre paréntesis se pone la posición de la columna en la tabla, es decir, la columna que deseamos. También se puede poner una cadena que indica el nombre de la columna (se hará referencia a estos métodos más adelante):

```
System.out.println (resul.getInt("dept_no") + " " + resul.getString("dnombre")+
                    " " + resul.getString("loc"));
```

Liberar recursos:

Por último se liberan todos los recursos y se cierra la conexión:

```
resul.close();      // Cerrar ResultSet
sentencia.close(); // Cerrar Statement
conexion.close();  //Cerrar conexión
```

Actividad 6: Siguiendo el ejemplo anterior obtén el APELLIDO, OFICIO y SALARIO de los empleados del departamento 10. Realiza otro programa Java que visualice el Apellido del empleado con máximo salario, visualiza también su SALARIO.

2.6.4 ACCESO A DATOS MEDIANTE EL PUENTE JDBC-ODBC

Hay productos (aunque muy pocos) para los que no hay controlador (o driver) JDBC pero sí hay un controlador ODBC. En estos casos se utiliza un puente denominado normalmente *JDBC-ODBC Bridge*. El puente JDBC-ODBC es un controlador JDBC que implementa operaciones JDBC traduciéndolas en operaciones ODBC, para ODBC aparece como una aplicación normal. El puente está implementado en Java y usa métodos nativos de Java para llamar a ODBC, se instala automáticamente con el JDK como el paquete *sun.jdbc.odbc* por lo que no es necesario añadir ningún JAR a nuestros proyectos para trabajar con él.

Por ejemplo para acceder a una base de datos MySQL usando el puente JDBC-ODBC necesitaremos crear un origen de datos o DSN (*Data Source Name*). En Windows nos vamos al *Panel de Control-> Herramientas administrativas-> Orígenes de datos (ODBC)*. Pulsamos en el botón *Agregar*, a continuación seleccionamos el driver ODBC para MySQL y pulsamos el botón *Finalizar*, véase Figura 2.13.

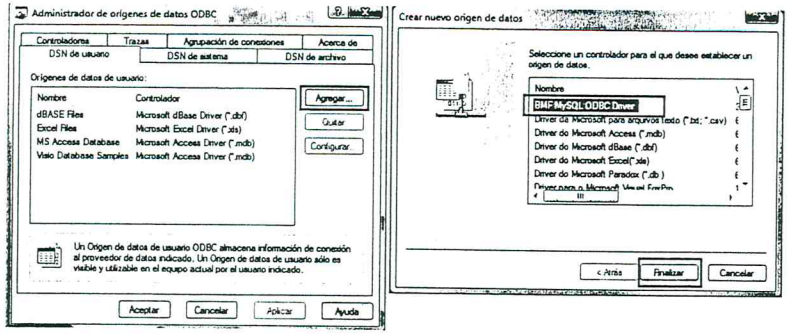


Figura 2.13. Crear un origen de datos ODBC.

En la siguiente pantalla hemos de dar un nombre al origen de datos en el campo *Data Source Name*, por ejemplo escribimos el nombre: *Mysql-odbc*. El siguiente campo es opcional. En el campo *Server* escribimos el nombre de máquina (o la dirección IP) donde reside la base de datos, si reside en la misma máquina desde la que creamos el origen de datos escribimos *localhost*. Como nombre de usuario escribimos *ejemplo* y a continuación su password (se escribe un usuario que exista en la base de datos). De la lista *Database* elegimos un esquema de base de datos, en este caso EJEMPLO. Pulsamos el botón *OK*, véase Figura 2.14.

Para cada esquema de base de datos es necesario crear un origen de datos. Desde la pestaña *Connect Options* podemos escribir el número de puerto por el que escucha MySQL; por defecto asume

3306. A continuación se visualiza la pantalla inicial del *Administrador de Orígenes de datos ODBC*, con el nuevo origen creado, clic en el botón *Aceptar* para finalizar.

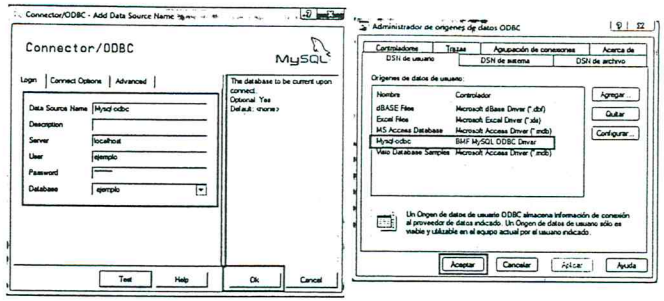


Figura 2.14. Origen de datos ODBC para acceder a MySQL desde Windows.

Puede ocurrir que no esté instalado el driver ODBC para conectar con bases de datos MySQL. En ese caso debemos descargarnos el driver e instalarlo. Accedemos a la web http://www.mysql.com/products/connector/ y descargamos el driver: *ODBC Driver for MySQL (Connector/ODBC)*. Puede ser un fichero con el nombre *mysql-connector-odbc-5.1.9-win32.msi*, lo ejecutamos y seguimos los pasos.

Ahora en nuestro programa Java anterior cambiamos 2 líneas, la carga del driver (*sun.jdbc.odbc.JdbcOdbcDriver*) y el establecimiento de la conexión en el que hay que escribir el nombre del origen de datos creado (*jdbc:odbc:Mysql-odbc*):

```
//Cargar el driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

// Establecemos la conexión con la BD
Connection conexion = DriverManager.getConnection ("jdbc:odbc:Mysql-odbc");
```

Para ejecutar este programa desde Linux (Ubuntu) hemos de instalar los paquetes *unixodbc*, *unixodbc-dev* y *libmyodbc* y crear el origen de datos ***Mysql-odbc*** como se hizo en el epígrafe de acceso a datos mediante ODBC. El código del programa Java sería el mismo.

Actividad 7: Realiza la Actividad 6 utilizando el puente JDBC-ODBC.

2.7 ESTABLECIMIENTO DE CONEXIONES

Hemos visto en ejemplos anteriores cómo se realiza la conexión con una base de datos MySQL utilizando JDBC y el puente JDBC-ODBC, a continuación vamos a ver como conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de tener creada la base de datos EJEMPLO con las tablas EMPLEADOS y DEPARTAMENTOS y vamos a utilizar el mismo programa Java, solo cambiaremos la carga del driver y la conexión a la base de datos.

En Windows supongamos que la base de datos EJEMPLO la tenemos en las carpetas D:\db\SQLite, D:\db\Hsqldb\ejemplo, D:\db\H2 y D:\db\Derby. En Linux en las carpetas /home/usuario/db/SQLite/, /home/usuario/db/Hsqldb/, /home/usuario/db/H2 y /home/usuario/db/Derby/. También para realizar las pruebas, necesitaremos tener en las carpetas el conector (fichero JAR) correspondiente para cada una de las bases de datos y el programa Java.

Conexión a SQLite

Para conectarnos a SQLite necesitamos la librería *sqlite-jdbc-3.7.2.jar* que se puede descargar desde la URL http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC. En la misma carpeta donde está la base de datos creamos nuestro programa Java (su nombre es Main.java) e incluimos el fichero JAR. El programa Java es el mismo que se utilizó anteriormente para acceder a la base de de datos MySQL de nombre *ejemplo* y que obtenía el listado de la tabla DEPARTAMENTOS. Cambiamos la carga del driver, en este caso se llama *org.sqlite.JDBC* y la conexión a la base de datos:

```
Class.forName("org.sqlite.JDBC");
Connection conexion = DriverManager.getConnection
          ("jdbc:sqlite:D:/db/SQLite/ejemplo.db");
```

Para compilar y ejecutar nuestro programa desde Windows ejecutamos desde la línea de comandos:

```
D:\db\SQLite>javac Main.java
D:\db\SQLite>Java  -classpath ".;sqlite-jdbc-3.7.2.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
                        ("jdbc:sqlite:/home/usuario/db/SQLite/ejemplo.db");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/SQLite/* definimos la variable CLASSPATH con la carpeta donde estará la clase (Main.class) y el fichero JAR:

```
$ export CLASSPATH=/home/usuario/db/SQLite/:/home/usuario/db/SQLite/sqlite-jdbc-
3.7.2.jar
$ javac Main.java
$ java Main
```

Conexión a Apache Derby

Para conectarnos a Apache Derby necesitamos la librería *derby.jar* que ya vimos como obtenerla al instalar la base de datos. En la misma carpeta donde está la base de datos copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *org.apache.derby.jdbc.EmbeddedDriver* y la conexión a la base de datos:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection conexion = DriverManager.getConnection
("jdbc:derby:D:/db/Derby/ejemplo");
```


Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Derby>javac Main.java
D:\db\Derby>java -classpath ".;derby.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
                        ("jdbc:derby:/home/usuario/db/Derby/ejemplo");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/Derby/* y definimos la variable CLASSPATH como antes:

```
$ export CLASSPATH=/home/usuario/db/Derby/:/home/usuario/db/Derby/derby.jar
$ javac Main.java
$ java Main
```

Conexión a HSQLDB

Para conectarnos a HSQLDB necesitamos la librería *hsqldb.jar* que se puede obtener de la carpeta *lib* que se encuentra al descomprimir el fichero *hsqldb-2.2.6.zip*. En la carpeta *ejemplo* (donde está la base de datos) copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *org.hsqldb.jdbcDriver* y la conexión a la base de datos:

```
Class.forName("org.hsqldb.jdbcDriver" );
Connection conexion = DriverManager.getConnection
                    ("jdbc:hsqldb:file:D:/db/Hsqldb/ejemplo/ejemplo");
```

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Hsqldb\ejemplo>javac Main.java
D:\db\Hsqldb\ejemplo>java -classpath ".;hsqldb.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
     ("jdbc:hsqldb:file:/home/usuario/db/Hsqldb/ejemplo/ejemplo");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/Hsqldb/ejemplo* y definimos la variable CLASSPATH:

```
$export CLASSPATH=/home/usuario/db/Hsqldb/ejemplo/:/home/usuario/db/Hsqldb/ejemplo/
hsqldb.jar
$ javac Main.java
$ java Main
```

Conexión a H2

Para conectarnos a H2 necesitamos la librería *h2-1.3.162.jar* que se puede obtener de la carpeta *bin* en la que se encuentra al descomprimir el fichero *h2-2011-11-26.zip*. En la carpeta donde está la base de datos (*D:\db\H2\ejemplo*) copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *org.h2.Driver* y la conexión a la base de datos:

```
Class.forName("org.h2.Driver" );
Connection conexion = DriverManager.getConnection
                    ("jdbc:h2:D:/db/H2/ejemplo/ejemplo","sa","");
```

En este caso se ha incluido el nombre de usuario *"sa"* ya que al crear la base de datos se dejó este nombre de usuario y la clave en blanco. Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\H2\ejemplo>javac Main.java
D:\db\H2\ejemplo>java -classpath ".;h2-1.3.162.jar" Main
```

En Linux cambiamos la conexión a la base de datos:

```
Connection conexion = DriverManager.getConnection
                    ("jdbc:h2:/home/usuario/db/H2/ejemplo/ejemplo","sa","");
```

Para ejecutarlo nos vamos a la carpeta */home/usuario/db/H2/ejemplo* y definimos la variable CLASSPATH:

```
$ export CLASSPATH=/home/usuario/db/H2/ejemplo/:/home/usuario/db/H2/ejemplo/
h2-1.3.162.jar
$ javac Main.java
$ java Main
```

Conexión a MySQL

Para conectarnos a MySQL necesitamos la librería *mysql-connector-java-5.1.18-bin.jar*. Nos creamos la carpeta MySQL dentro de D:\db, copiamos el programa Java, el fichero JAR y cambiamos la carga del driver, en este caso se llama *com.mysql.jdbc.Driver* y la conexión a la base de datos:

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection
       ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
```

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\MySQL>javac Main.java
D:\db\MySQL>java -classpath ".;mysql-connector-java-5.1.18-bin.jar" Main
```

Desde Linux nos vamos a la carpeta */home/usuario/db/MySQL* copiamos el programa Java y el JAR y definimos la variable CLASSPATH:


```
$export CLASSPATH=/home/usuario/db/MySQL/:/home/usuario/db/MySQL/mysql-connector-
$java-5.1.18-bin.jar
$javac Main.java
$java Main
```

Conexión a ORACLE

Para conectarnos a Oracle utilizando el puente ODBC-JDBC hemos de crear un origen de datos. Asumiento que tenemos instalado en nuestra máquina Windows *Oracle Database 10g Express Edition* creamos un usuario de nombre y clave *EJEMPLO* y creamos las tablas EMPLEADOS y DEPARTAMENTOS. Al instalarse la base de datos se crea el servicio XE para acceder a ella. Para crear el origen de datos seleccionamos *Oracle in XE*, pulsamos el botón *Finalizar* (también se podía haber elegido el driver *Microsoft ODBC for Oracle*) y a continuación rellenamos el resto de campos para la configuración del driver, véase Figura 2.15. Damos el nombre de *ORACLE-XE* al origen de datos.

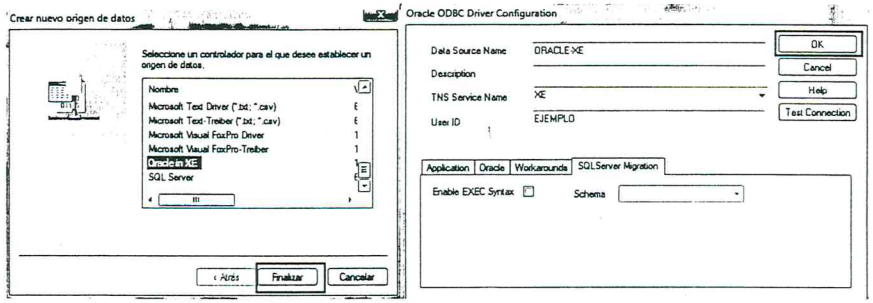


Figura 2.15. Origen de datos ODBC para acceder a Oracle-XE desde Windows.

Nos creamos la carpeta *oracle* dentro de D:\db, copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *sun.jdbc.odbc.JdbcOdbcDriver* y la conexión a la base de datos:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conexion = DriverManager.getConnection
                    ("jdbc:odbc:ORACLE-XE","ejemplo", "ejemplo");
```

En este caso para compilar y ejecutar el programa desde Windows escribimos: *javac Main.java* y *java Main*.

Para conectarnos mediante JDBC usamos el driver *JDBC Thin*. Se puede descargar desde la página web de Oracle (es necesario comprobar antes la versión de la base de datos instalada), desde la dirección http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html. Para el ejemplo se ha descargado el driver *classes12.jar* para la versión *Oracle Database 10g Release 2 (10.2.0.1.0)*.

Copiamos el JAR en la carpeta *D:\db\oracle* y cambiamos la carga del driver, en este caso se llama *oracle.jdbc.driver.OracleDriver*, y la conexión a la base de datos:

```
Class.forName ("oracle.jdbc.driver.OracleDriver");
Connection conexion = DriverManager.getConnection
                ("jdbc:oracle:thin:@localhost:1521:XE", "ejemplo", "ejemplo");
```

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\oracle>javac Main.java
D:\db\oracle>java -classpath ".;classes12.jar" Main
```

Desde Linux nos vamos a la carpeta */home/usuario/db/oracle/* copiamos el programa Java y el JAR y definimos la variable CLASSPATH:

```
$ export CLASSPATH=/home/usuario/db/oracle/:/home/usuario/db/oracle/classes12.jar
$ javac Main.java
$ java Main
```

Conexión a ACCESS

Para conectarnos a Microsoft Access necesitamos crear un origen de datos, en este caso elegimos el origen de datos *Microsoft Access Driver (*.mdb, *.accdb)* o *Microsoft Access Driver (*.mdb)*, dependerá de la versión de Access que tengamos instalada. Una vez elegido, le damos un nombre y pulsamos el botón *Seleccionar* para localizar el fichero .mdb o .accdb. Tras pulsar en el botón *Aceptar* tenemos creado el origen de datos, Figura 2.16. Para el ejemplo se ha creado una base de datos en Access de nombre *ejemplo.accdb*, que contiene una tabla de nombre DEPARTAMENTOS.

Figura 2.16. Origen de datos ODBC para acceder a una base de datos Access

Nos creamos la carpeta *Access* dentro de *D:\db*, copiamos el programa Java y cambiamos la carga del driver, en este caso se llama *sun.jdbc.odbc.JdbcOdbcDriver* y la conexión a la base de datos:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conexion = DriverManager.getConnection("jdbc:odbc:EJEMPLO-ACCESS");
```

Para compilar y ejecutar nuestro programa desde Windows:

```
D:\db\Access>javac Main.java
D:\db\Access>java Main
```


2.8 EJECUCIÓN DE SENTENCIAS DE DESCRIPCIÓN DE DATOS

Normalmente, cuando desarrollamos una aplicación JDBC conocemos la estructura de las tablas y datos que estamos manejando, es decir, conocemos, las columnas que tienen y cómo están relacionadas entre sí. Es posible que no conozcamos la estructura de las tablas de una base de datos, en este caso la información de la base de datos se puede obtener a través de los *metaobjetos*, que no son más que objetos que proporcionan información sobre la base de datos.

El objeto **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información. El siguiente ejemplo conecta con la base de datos MySQL de nombre *ejemplo* y muestra información sobre el producto de base de datos, el driver, la URL para acceder a la base de datos, el nombre de usuario y las tablas y vistas del esquema actual (o de todos los esquemas dependiendo del sistema gestor de base de datos), un esquema se corresponde generalmente con un usuario de la base de datos; el método **getMetaData()** de la clase *Connection* devuelve un objeto **DataBaseMetaData** con el que se obtendrá la información sobre la base de datos:

```
import java.sql.*;
public class Main {
public static void main(String[] args) {
      try
      {
        Class.forName("com.mysql.jdbc.Driver"); //Cargar el driver
        //Establecemos la conexion con la BD
        Connection conexion = DriverManager.getConnection
           ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");

        DatabaseMetaData dbmd = conexion.getMetaData();//Creamos objeto
        ResultSet resul = null;

        String nombre  = dbmd.getDatabaseProductName();
        String driver  = dbmd.getDriverName();
        String url     = dbmd.getURL();
        String usuario = dbmd.getUserName() ;

        System.out.println("INFORMACIÓN SOBRE LA BASE DE DATOS:");
        System.out.println("==================================");
        System.out.println("Nombre : " + nombre );
        System.out.println("Driver : " + driver );
        System.out.println("URL    : " + url );
        System.out.println("Usuario: " + usuario );

        //Obtener información de las tablas y vistas que hay
        resul = dbmd.getTables(null, "ejemplo", null, null);

        while (resul.next()) {
        String catalogo = resul.getString(1); //columna 1 que devuelve ResulSet
        String esquema = resul.getString(2);  //columna 2
        String tabla = resul.getString(3);    //columna 3
        String tipo = resul.getString(4);     //columna 4
        System.out.println(tipo + " - Catalogo: " + catalogo +
```

```
        ", Esquema : " + esquema  + ", Nombre : " + tabla);
    }
    conexion.close(); //Cerrar conexion
    }
    catch (ClassNotFoundException cn) {cn.printStackTrace();}
    catch (SQLException e) {e.printStackTrace();}
  }//fin de main
}//fin de la clase
```

Visualiza la siguiente información:

```
INFORMACIÓN SOBRE LA BASE DE DATOS:
===================================
Nombre : MySQL
Driver : MySQL-AB JDBC Driver
URL    : jdbc:mysql://localhost/ejemplo
Usuario: ejemplo@localhost
TABLE - Catalogo: ejemplo, Esquema : null, Nombre : departamentos
TABLE - Catalogo: ejemplo, Esquema : null, Nombre : empleados
VIEW - Catalogo: ejemplo, Esquema : null, Nombre : vista
```

El método **getTables()** devuelve un objeto *ResultSet* que proporciona información sobre las tablas y vistas de la base de datos. Necesita 4 parámetros que en el ejemplo anterior tenían el valor null:

- Primer parámetro: catálogo de la base de datos. El método obtiene las tablas del catálogo indicado, al poner *null*, indicamos todos los catálogos.
- Segundo parámetro: esquema de la base de datos. Obtiene las tablas del esquema indicado, el valor *null* indica el esquema actual (o todos los esquemas, dependiendo del SGBD, como en Oracle).
- Tercer parámetro: es un patrón en el que se indica el nombre de las tablas que queremos que obtenga el método. Se puede utilizar el carácter guión bajo o porcentaje, por ejemplo *"de%"* obtendría todas las tablas cuyo nombre empiece por *"de"*.
- El cuarto parámetro es un array de *String*, en el que indicamos qué tipos de tablas queremos: TABLE (para tablas), VIEW (para vistas); al poner *null*, nos devolverá todos los tipos ya sean tablas o vistas. El siguiente ejemplo nos devolvería solo las tablas:

```
String[] tipos = {"TABLE"};
resul = dbmd.getTables(null, null, null, tipos);
```

Cada fila de *ResultSet* que devuelve **getTables()** tiene información sobre una tabla. Las columnas de *ResultSet* que devuelve el método son: TABLE_CAT (el nombre del catálogo al que pertenece la tabla), TABLE_SCHEM (el nombre del esquema al que pertenece la tabla), TABLE_NAME (el nombre de la tabla o vista), TABLE_TYPE (el tipo TABLE o VIEW), REMARKS (comentarios). Para obtener estos resultados también podríamos haber puesto en el código anterior:

```
String catalogo  = resul.getString("TABLE_CAT");
String esquema   = resul.getString("TABLE_SCHEM");
String tabla     = resul.getString("TABLE_NAME");
String tipo      = resul.getString("TABLE_TYPE");
```


Actividad 8: Prueba el programa anterior para visualizar información de las bases de datos Oracle y SQLite con las que estás trabajando en este tema.

Otros métodos importantes del objeto **DatabaseMetaData** son:

- *getColumns(catálogo, esquema, nombre_de_tabla, nombre_de_columna)* devuelve información sobre las columnas de una tabla o tablas. Para el nombre de la tabla y de la columna se puede utilizar el carácter guión bajo o porcentaje. Por ejemplo *getColumns(null, "ejemplo", "departamentos", "d%")*, obtiene todos los nombres de columna que empiezan por la letra d en la tabla *departamentos* y en el esquema de nombre *ejemplo*. El valor null en los 4 parámetros indica que obtiene información de todas las columnas y tablas del esquema actual. El siguiente ejemplo muestra información sobre todas las columnas de la tabla *departamentos*:

```
System.out.println("COLUMNAS TABLA DEPARTAMENTOS:");
System.out.println("===================================");
ResultSet columnas=null;
columnas = dbmd.getColumns(null, "ejemplo", "departamentos", null);
while (columnas.next()) {
        String nombreCol = columnas.getString("COLUMN_NAME");//getString(4)
        String tipoCol = columnas.getString("TYPE_NAME");  //getString(6)
        String  tamCol  = columnas.getString("COLUMN_SIZE");//getString(7)
        String  nula  = columnas.getString("IS_NULLABLE"); //getString(18)

        System.out.println("Columna: " + nombreCol + ", Tipo: " + tipoCol  +
                  ", Tamaño: " + tamCol +", ¿Puede ser Nula:? " + nula);
}
```

Visualiza:

```
COLUMNAS TABLA DEPARTAMENTOS:
===================================
Columna: dept_no, Tipo: TINYINT, Tamaño: 3, ¿Puede ser Nula:? NO
Columna: dnombre, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula:? YES
Columna: loc, Tipo: VARCHAR, Tamaño: 15, ¿Puede ser Nula:? YES
```

- *getPrimaryKeys(catálogo, esquema, tabla)* devuelve la lista de columnas que forman la clave primaria. El siguiente ejemplo muestra la clave primaria de la tabla *departamentos* (ejemplo en MySQL):

```
ResultSet pk = dbmd.getPrimaryKeys(null, "ejemplo", "departamentos");
String pkDep="", separador="";
while (pk.next()) {
     pkDep = pkDep + separador + pk.getString("COLUMN_NAME");//getString(4)
     separador="+";
}
System.out.println("Clave Primaria: " + pkDep);
```

- *getExportedKeys(catalogo,esquema,tabla)* devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla. El siguiente ejemplo muestra las tablas y sus claves ajenas que

referencian a la tabla DEPARTAMENTOS, en este caso solo la tabla EMPLEADOS (ejemplo en ORACLE):

```
ResultSet fk = dbmd.getExportedKeys(null, "EJEMPLO", "DEPARTAMENTOS");
while (fk.next()) {
  String fk_name =fk.getString("FKCOLUMN_NAME");
  String pk_name =fk.getString("PKCOLUMN_NAME");
  String pk_tablename =fk.getString("PKTABLE_NAME");
  String fk_tablename =fk.getString("FKTABLE_NAME");
  System.out.println("Tabla PK: "+ pk_tablename+ ", Clave Primaria: "+ pk_name);
  System.out.println("Tabla FK: "+ fk_tablename+ ", Clave Ajena: " + fk_name);
}
```

Visualiza (en Oracle):

```
Tabla PK: DEPARTAMENTOS, Clave Primaria: DEPT_NO
Tabla FK: EMPLEADOS, Clave Ajena: DEPT_NO
```

El método no devuelve nada si escribimos *dbmd.getExportedKeys(null, "EJEMPLO", "EMPLEADOS")* ya que la tabla EMPLEADOS no es referenciada por ninguna clave ajena.

- *getImportedKeys(catálogo, esquema, tabla)* devuelve la lista de claves ajenas existentes en la tabla. Se utiliza igual que el método anterior, en este caso *dbmd.getImportedKeys(null, "EJEMPLO", "EMPLEADOS")* devuelve la salida anterior, en cambio *dbmd.getImportedKeys(null, "EJEMPLO", "DEPARTAMENTOS")* no devuelve nada ya que no tiene claves ajenas.
- *getProcedures(catálogo, esquema, procedure)* devuelve la lista de procedimientos almacenados. El siguiente ejemplo muestra los procedimientos y funciones que tiene el esquema de nombre EJEMPLO (ejemplo en ORACLE):

```
ResultSet proc = dbmd.getProcedures(null,"EJEMPLO", null);
while (proc.next()) {
   String proc_name =proc.getString("PROCEDURE_NAME");
   String proc_type =proc.getString("PROCEDURE_TYPE");
   System.out.println("Nombre Procedimiento: "+ proc_name +
                      " - Tipo: "+ proc_type);
}
```

Métodos de **DatabaseMetaData:**
http://docs.oracle.com/javase/6/docs/api/java/sql/DatabaseMetaData.html

2.8.1 RESULTSETMETADATA

Se pueden obtener metadatos (datos sobre los datos) a partir de un *ResultSet* mediante la interfaz **ResultSetMetaData**; es decir podemos obtener más información sobre los tipos y propiedades de las columnas como por ejemplo, el número de columnas devueltas. El siguiente trozo de código muestra el uso de la interfaz para conocer más información acerca de las columnas devueltas por *ResultSet*, en este caso desconocemos el nombre de las columnas devueltas por la consulta *SELECT * FROM departamentos*; el método **getMetaData()** del objeto *ResultSet* devuelve una referencia a un objeto **ResultSetMetaData** con el que se obtendrá la información acerca de las columnas devueltas:


```
Statement sentencia = conexion.createStatement();
ResultSet rs = sentencia.executeQuery("SELECT * FROM departamentos");
ResultSetMetaData rsmd = rs.getMetaData();
int nColumnas =rsmd.getColumnCount();
String nula;
System.out.println("Número de columnas recuperadas: "+ nColumnas);
for (int i = 1; i <= nColumnas; i++) {
  System.out.println("Columna " +i+":");
  System.out.println("  Nombre : " +rsmd.getColumnName(i));
  System.out.println("  Tipo : " +rsmd.getColumnTypeName(i));
  if (rsmd.isNullable(i)==0) nula="NO"; else nula ="SI";
  System.out.println("  Puede ser nula? : " +nula);
  System.out.println("  Máximo ancho de la columna: "+rsmd.getColumnDisplaySize(i));
}
```

Obtiene la siguiente información:

```
Número de columnas recuperadas: 3
Columna 1:
  Nombre : dept_no
  Tipo : TINYINT
  Puede ser nula? : NO
  Máximo ancho de la columna: 2
Columna 2:
  Nombre : dnombre
  Tipo : VARCHAR
  Puede ser nula? : SI
  Máximo ancho de la columna: 15
Columna 3:
  Nombre : loc
  Tipo : VARCHAR
  Puede ser nula? : SI
  Máximo ancho de la columna: 15
```

Los métodos utilizados son:

- *getColumnCount()*: devuelve el número de columnas devueltas por la consulta (*SELECT * FROM departamentos*).
- *getColumnName(índice de la columna)*: devuelve el nombre de la columna.
- *getColumnTypeName(índice)*: devuelve el nombre del tipo de dato que contiene la columna específico del sistema de bases de datos.
- *isNullable(índice)* : devuelve 0 si la columna puede contener valores nulos.
- *getColumnDisplaySize(índice)* : devuelve el máximo ancho en caracteres de la columna.

Actividad 9: Visualiza información sobre las columnas de la tabla EMPLEADOS en SQLite.

2.9 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

En ejemplos anteriores vimos cómo se podían ejecutar sentencias SQL mediante la interfaz **Statement** (sentencia), esta proporciona métodos para ejecutar sentencias SQL y obtener los resultados. Como **Statement** es una interfaz no se pueden crear objetos directamente, en su lugar los objetos se obtienen con una llamada al método *createStatement()* de un objeto *Connection* válido:

```
Statement sentencia = conexion.createStatement();
```

Al crearse un objeto **Statement** se crea un espacio de trabajo para crear consultas SQL, ejecutarlas y para recibir los resultados de las consultas. Una vez creado el objeto se pueden usar los siguientes métodos:

- *executeQuery(String):* se utiliza para sentencias SQL que recuperan datos de un único objeto *ResultSet*, se utiliza para las sentencias SELECT.
- *executeUpdate(String):* se utiliza para sentencias que no devuelven un *ResultSet* como son las sentencias de manipulación de datos (DML): INSERT, UPDATE y DELETE; y las sentencias de definición de datos(DDL): CREATE, DROP y ALTER. El método devuelve un entero indicando el número de filas que se vieron afectadas y en el caso de las sentencias DDL devuelve el valor 0.
- *execute(String):* se utiliza para sentencias que devuelven más de un *ResultSet*, se suele utilizar para ejecutar procedimientos almacenados.

A través de un objeto *ResultSet* se puede acceder al valor de cualquier columna de la fila actual por nombre o por posición, también se puede obtener información sobre las columnas como el número de columnas o su tipo; en ejemplos anteriores vimos como se podía averiguar el número de columnas devueltas por una orden SELECT usando el método **getMetaData()** de un objeto *ResultSet*. Algunos de los métodos de la interfaz *ResultSet* son:

MÉTODO	DEVUELVEN OBJETOS DE TIPO:
`getString(columna)`	String
`getBoolean(columna)`	boolean
`getByte(columna)`	byte
`getShort(columna)`	short
`getInt(columna)`	int
`getLong(columna)`	long
`getFloat(columna)`	float
`getDouble(columna)`	double
`getDate(columna)`	Date
`getTime(columna)`	Time

El siguiente ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se introducen al ejecutar el programa desde la línea de comandos, el primer parámetro es el departamento, el siguiente el nombre y el tercero la localidad. Antes de ejecutar la orden INSERT construimos la sentencia en un String, las cadenas de caracteres deben ir encerradas entre comillas simples:


```
import java.sql.*;
public class InsertarDep {
    public static void main(String[] args) {
      try
      {
        Class.forName("com.mysql.jdbc.Driver");//Cargar el driver
        //Establecemos la conexion con la BD
        Connection conexion = DriverManager.getConnection
        ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
        //recuperar argumentos de main
        String dep=args[0];          // num. departamento
        String dnombre=args[1];      // nombre
        String loc=args[2];          // localidad

        //construir orden INSERT
        tring sql= "INSERT INTO departamentos VALUES (" + dep + ",'"
        + dnombre + "','"+loc+ "')";
        System.out.println(sql);
        Statement sentencia = conexion.createStatement();
        int filas = sentencia.executeUpdate(sql);
        System.out.println("Filas afectadas: "+filas);

        sentencia.close();           // Cerrar Statement
        conexion.close();            //Cerrar conexión

      }
      catch (ClassNotFoundException cn) {cn.printStackTrace();}
      catch (SQLException e) {e.printStackTrace();}

    }//fin de main
}//fin de la clase
```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se inserta el departamento 15 de nombre INFORMÁTICA y localidad MADRID:

```
java InsertarDep 15 INFORMÁTICA MADRID
INSERT INTO departamentos VALUES (15,'INFORMÁTICA','MADRID')
Filas afectadas: 1
```

El siguiente código sube el salario a los empleados de un departamento (supongamos que el programa se llama *ModificarSalario.java*). El número de departamento y la subida se reciben desde la línea de argumentos:

```
//recuperar parametros de main
String dep=args[0];     // num. departamento
String subida=args[1]; // subida

//   construir orden UPDATE
String sql= "UPDATE empleados SET salario=salario + " + subida
            + " WHERE dept_no = " +dep;
System.out.println(sql);
Statement sentencia = conexion.createStatement();
```

```
int filas = sentencia.executeUpdate(sql);
System.out.println("Filas modificadas: "+filas);
```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se sube 100 euros a los empleados del departamento 10:

```
java ModificarSalario 10 100
UPDATE empleados SET salario=salario + 100 WHERE dept_no = 10
Filas modificadas: 3
```

El siguiente ejemplo crea una vista (de nombre *totales*) que contiene por cada departamento el número, el nombre, el número de empleados que tiene y la suma de salarios:

```
//construir orden CREATE
String sql= "CREATE OR REPLACE VIEW totales (dep, dnombre, nemp, media) AS " +
            "SELECT d.dept_no,dnombre, COUNT(emp_no), AVG(salario) " +
            "FROM   departamentos d LEFT JOIN  empleados e " +
            "ON e.dept_no = d.dept_no " +
            "GROUP BY d.dept_no, dnombre " ;
System.out.println(sql);
Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
System.out.println("Resultado ejecución: "+filas);
```

La ejecución muestra la siguiente información:

```
java  CrearVista
CREATE OR REPLACE VIEW totales (dep, dnombre, nemp, media) AS SELECT d.dept_no,
dnombre, COUNT(emp_no), AVG(salario) FROM   departamentos d LEFT JOIN  empleados
 e ON e.dept_no = d.dept_no GROUP BY d.dept_no, dnombre
Resultado ejecucion: 0
```

Actividad 10: Crea un programa Java que inserte un empleado en la tabla EMPLEADOS, el programa recibe desde la línea de argumentos los valores a insertar. Los argumentos que recibe son los siguientes: *EMP_NO, APELLIDO, OFICIO, DIR, SALARIO, COMISIÓN, DEPT_NO.* Antes de insertar se deben realizar las siguientes comprobaciones:

- que el departamento exista en la tabla DEPARTAMENTOS, si no existe no se inserta.
- que el número del empleado no exista, si existe no se inserta.
- que el salario sea > que 0, si es <=0 no se inserta.
- que el director (DIR, es el número de empleado de su director) exista en la tabla EMPLEADOS, si no existe no se inserta.

La fecha de alta del empleado es la fecha actual.

Cuando se inserte la fila visualizar el mensaje y si no se inserta visualizar el motivo (departamento inexistente, número de empleado duplicado, director inexistente, etc.).

2.9.1 SENTENCIAS PREPARADAS

En los ejemplos anteriores hemos creado sentencias SQL a partir de cadenas de caracteres en las que íbamos concatenando los datos necesarios para construir la sentencia completa. La interfaz **PreparedStatement** nos va a permitir construir una cadena de caracteres SQL con *placeholder* (marcadores de posición) que representarán los datos que serán asignados más tarde, el *placeholder* se representa mediante el símbolo interrogación (?). Por ejemplo la orden INSERT del ejemplo anterior se representaría así:

```
String sql= "INSERT INTO departamentos VALUES (?, ?, ?)";
                                                // 1  2  3 valor del índice
```

Cada *placeholder* tiene un índice, el 1 correspondería al primero que se encuentre en la cadena, el 2 al segundo y así sucesivamente. Solo se pueden utilizar para ocupar el sitio de los datos en la cadena SQL, no se pueden usar para representar una columna o un nombre de una tabla, por ejemplo *FROM ?* sería incorrecto. Antes de ejecutar un **PreparedStatement** es necesario asignar los datos para que cuando se ejecute la base de datos asigne variables de unión con estos datos y ejecute la orden SQL. Los objetos **PreparedStatement** se pueden preparar o precompilar una sola vez y ejecutar las veces que queramos asignando diferentes valores a los marcadores de posición, en cambio en los objetos **Statement**, la sentencia SQL se suministra en el momento de ejecutar la sentencia.

Los métodos de **PreparedStatement** tienen los mismos nombres (*executeQuery(), executeUpdate()* y *execute()*) que en **Statement** pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método *prepareStatement(String)*:

```
PreparedStatement sentencia = conexion.prepareStatement(sql);
```

El ejemplo anterior en el que se inserta una fila en la tabla DEPARTAMENTOS quedaría así:

```
//construir orden INSERT
String sql= "INSERT INTO departamentos VALUES ( ?, ?, ?)";
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(1,Integer.parseInt(dep)); // num departamento
sentencia.setString(2,dnombre);  // nombre
sentencia.setString(3,loc);      // localidad

int filas = sentencia.executeUpdate();  // filas afectadas
```

El ejemplo en el que se modifica el salario de los empleados quedaría así:

```
//construir orden UPDATE
String sql= "UPDATE empleados SET salario= salario + ? WHERE dept_no = ?";
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(2,Integer.parseInt(dep));       // num departamento
sentencia.setFloat(1,Float.parseFloat(subida)); // subida

int filas = sentencia.executeUpdate ();  // filas afectadas
```

Se utilizan los métodos *setInt(índice, entero)*, *setString(índice, cadena)* y *setFloat(índice, float)* para asignar los valores a cada uno de los marcadores de posición. Con *Integer.parseInt(dep)* convertimos la cadena *dep* a entero y con *Float.parseFloat(subida)* convertimos la cadena *subida* a float.

También se puede utilizar esta interfaz con la orden SELECT. El siguiente ejemplo muestra el apellido y salario de los empleados de un departamento y un oficio concreto, el departamento y oficio se introducen desde la línea de comandos al ejecutar el programa:

```
//recuperar parametros de main
String dep=args[0];          //departamento
String oficio=args[1]; //oficio
//construir orden SELECT
String sql= "SELECT apellido, salario FROM empleados "
          + " WHERE dept_no = ? AND oficio = ? ORDER BY 1";
// Preparamos la sentencia
PreparedStatement sentencia = conexion.prepareStatement(sql);

sentencia.setInt(1,Integer.parseInt(dep));
sentencia.setString(2,oficio);

ResultSet rs = sentencia.executeQuery();
while (rs.next())
      System.out.println (rs.getString("apellido")+ " => " +
                          rs.getFloat("salario"));

rs.close();// liberar recursos
sentencia.close();
conexion.close();
```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información en la que se muestran los vendedores del departamento 30:

```
java  VerEmpleado 30 VENDEDOR
ARROYO => 1500.0
MARTÍN => 1600.0
SALA => 1625.0
TOVAR => 1350.0
```

Algunos de los métodos de la interfaz **PreparedStatement** para dar valor a los marcadores de posición son:

MÉTODO
void setString(indice, String)
Void setBoolean(indice, boolean)
void setByte(indice, byte)
void setShort(indice, short)
void setInt(indice, int)
void setLong(indice, long)
void setFloat(indice, float)

void setDouble(indice, double)
void setDate(indice, Date)
void setTime(indice, Time)
void setNull(indice, int tipoSQL)

Los valores NULL se insertan en la base de datos usando *setNull()*.

Actividad 11: Utiliza la interfaz **PreparedStatement** para visualizar el APELLIDO, SALARIO y OFICIO de los empleados de un departamento cuyo valor se recibe desde la línea de argumentos. Visualiza también el nombre de departamento. Visualiza al final el salario medio y el número de empleados del departamento. Si el departamento no existe en la tabla DEPARTAMENTOS visualiza un mensaje indicándolo.

2.10 EJECUCIÓN DE PROCEDIMIENTOS

Los procedimientos almacenados en la base de datos consisten en un conjunto de sentencias SQL y del lenguaje procedural utilizado por el sistema gestor de base de datos que se pueden llamar por su nombre para llevar a cabo alguna tarea en la base de datos. Pueden definirse con parámetros de entrada (IN), de salida (OUT), de entrada/salida (INOUT) o sin ningún parámetro. También pueden devolver un valor, en este caso se trataría de una función. Las técnicas para desarrollar procedimientos y funciones almacenadas dependen del sistema gestor de base de datos, en MySQL por ejemplo las funciones no admiten parámetros OUT e INOUT, solo admiten parámetros IN. A continuación se exponen unos ejemplos sencillos para Oracle y MySQL.

El siguiente ejemplo muestra un procedimiento de nombre *subida_sal* que sube el sueldo a los empleados de un departamento, el procedimiento recibe dos parámetros de entrada que son el número de departamento (*d*) y la subida (*subida*):

Procedimiento en ORACLE:

```
CREATE OR REPLACE PROCEDURE subida_sal (d NUMBER, subida NUMBER) AS
BEGIN
    UPDATE empleados SET salario= salario + subida
    WHERE dept_no=d;
   COMMIT;
END;
/
```

Procedimiento en MySQL:

```
delimiter//
CREATE PROCEDURE subida_sal (d INT, subida INT)
BEGIN
   UPDATE empleados SET salario = salario + subida WHERE dept_no=d;
```

```
    COMMIT;
END;
//
```

El siguiente ejemplo crea una función (en ORACLE) de nombre *nombre_dep* con dos parámetros, el primero es de entrada y recibe un número de departamento, el segundo es de salida, se utilizará para guardar la localidad del departamento; la función devuelve el nombre; si el departamento no existe devuelve "INEXISTENTE",:

Función en ORACLE:

```
CREATE OR REPLACE FUNCTION nombre_dep (d NUMBER, locali OUT VARCHAR2)
                                        RETURN VARCHAR2 AS
     nom VARCHAR2(15);
 BEGIN
     SELECT dnombre, loc INTO nom, locali FROM departamentos WHERE dept_no=d;
     RETURN nom;
 EXCEPTION
     WHEN NO_DATA_FOUND THEN
        nom:= 'INEXISTENTE';
        RETURN nom;
 END;
/
```

La interfaz **CallableStatement** permite que se pueda llamar desde Java a los procedimientos almacenados, para crear un objeto se llama al método *prepareCall(String)* del objeto *Connection*, el siguiente ejemplo declara la llamada al procedimiento *subida_sal* que tiene dos parámetros y para darles valor se utilizan los marcadores de posición (?):

```
String sql= "{ call subida_sal (?, ?) } ";
CallableStatement llamada = conexion.prepareCall(sql);
```

Hay cuatro formas de declarar las llamadas a los procedimientos y funciones que dependen del uso u omisión de parámetros, y de la devolución de valores. Son las siguientes:

- *{ call procedimiento}*: para un procedimiento almacenado sin parámetros.
- *{ ? = call función }*: para una función almacenada que devuelve un valor y no recibe parámetros, el valor se recibe a la izquierda del igual y es el primer parámetro.
- *{ call procedimiento(?, ?, ...) }*: para un procedimiento almacenado que recibe parámetros.
- *{ ? = call función(?, ?, ...) }*: para una función almacenada que devuelve un valor (primer parámetro) y recibe varios parámetros.

En el siguiente ejemplo se realiza una llamada al procedimiento *subida_sal* (de MySQL); los valores de los parámetros se introducen al ejecutar el programa desde la línea de comandos:

```
import java.sql.*;
public class ProcSubida {
    public static void main(String[] args) {
      try
```



```
{
      Class.forName("com.mysql.jdbc.Driver");    //Cargar el driver
      Connection conexion = DriverManager.getConnection
            ("jdbc:mysql://localhost/ejemplo?noAccessToProcedureBodies=true",
              "ejemplo", "ejemplo");

      //recuperar parametros de main
      String dep=args[0]; //departamento
      String subida=args[1]; //subida
      //construir orden DE LLAMADA
      String sql= "{ call subida_sal (?, ?) } ";

      // Preparamos la llamada
      CallableStatement llamada = conexion.prepareCall(sql);
      // Damos valor a los argumentos
      llamada.setInt(1,Integer.parseInt(dep));       // primer argumento-dep
      llamada.setFloat(2,Float.parseFloat(subida)); // segundo argumento-subida

      llamada.executeUpdate();  //ejecutar el procedimiento
      System.out.println ("Subida realizada....");
      llamada.close();
      conexion.close();
      }
      catch (ClassNotFoundException cn) {cn.printStackTrace();}
      catch (SQLException e) {e.printStackTrace();}

    }//fin de main
}//fin de la clase
```

La ejecución desde la línea de comandos y suponiendo que el conector MySQL está en el CLASSPATH visualiza la siguiente información:

```
java ProcSubida 30 200
Subida realizada....
```

En MySQL puede ocurrir que el usuario (en este caso se llama *ejemplo*) no tenga permisos para ejecutar procedimientos, en este caso debemos darle el privilegio SELECT sobre la tabla de sistema *mysql.proc*, que contiene la información sobre todos los procedimientos almacenados en la base de datos; se ejecutaría la siguiente orden desde la línea de comandos de MySQL o desde el entorno gráfico que usemos:

```
GRANT SELECT ON mysql.proc TO 'ejemplo'@'localhost';
```

También es necesario incluir en la conexión el parámetro *noAccessToProcedureBodies* con el valor *true*: *"jdbc:mysql://localhost/ejemplo?noAccessToProcedureBodies=true"*.

Cuando un procedimiento o función tiene parámetros de salida (OUT) deben ser registrados antes de que la llamada tenga lugar, si no se registra se producirá un error. El método que se utilizará es: *registerOutParameter(índice, tipoJDBC)*, el primer parámetro es la posición y el siguiente es una constante definida en la clase *java.sql.Types*. La clase *Types* define una constante para cada tipo genérico SQL, algunas son: TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC,

DECIMAL, CHAR, VARCHAR, DATE, TIME, TIMESTAMP, JAVA_OBJECT, ARRAY, BOOLEAN, ROWID etc. Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR en la base de datos se pondría:

```
llamada.registerOutParameter(2, java.sql.Types.VARCHAR);
```

Una vez ejecutada la llamada al procedimiento, los valores de los parámetros OUT e INOUT se obtienen con los métodos *getXXX(índice)* similares a los utilizados para obtener los valores de las columnas en un *Resultset*. El siguiente ejemplo ejecuta el procedimiento *nombre_dep* (de Oracle); desde la línea de comandos se recibe el número de departamento cuyos datos se visualizarán:

```
import java.sql.*;
public class FuncNombre {
    public static void main(String[] args) {
      try
      {
         Class.forName ("oracle.jdbc.driver.OracleDriver");
         Connection conexion = DriverManager.getConnection
                  ("jdbc:oracle:thin:@localhost:1521:XE", "ejemplo", "ejemplo");

         //recuperar parametro de main
         String dep=args[0];      //departamento

         //construir orden DE LLAMADA
         String sql= "{ ? = call nombre_dep (?, ?) } ";

         // Preparamos la llamada
         CallableStatement llamada = conexion.prepareCall(sql);

         llamada.registerOutParameter(1,Types.VARCHAR); //valor devuelto

         llamada.setInt(2,Integer.parseInt(dep));        //param de entrada
         llamada.registerOutParameter(3,Types.VARCHAR); //parametro OUT


         llamada.executeUpdate();  //ejecutar el procedimiento
         System.out.println ("Nombre Dep: "+llamada.getString(1) +
                  "          Localidad: "+llamada.getString(3));

         llamada.close();
         conexion.close();
       }
      catch (ClassNotFoundException cn) {cn.printStackTrace();}
      catch (SQLException e) {e.printStackTrace();}
      }//fin de main
}//fin de la clase
```

La ejecución desde la línea de comandos y suponiendo que el conector ORACLE está en el CLASSPATH visualiza la siguiente información:

```
java  FuncNombre 10
Nombre Dep: CONTABILIDAD Localidad: SEVILLA
```



```
java  FuncNombre 120
Nombre Dep: INEXISTENTE Localidad: null
```

Actividad 12: Crea una función en Oracle que reciba un número de departamento y devuelva su nombre si existe y si no existe devuelva INEXISTENTE. Una vez realizada la función modifica la Actividad 11 para que el nombre del departamento se obtenga llamando a dicha función.

2.11 GESTIÓN DE ERRORES

Hasta ahora en todos los ejemplos cuando se producía un error se visualizaba con *printStackTrace()* la secuencia de llamadas al método que ha producido la excepción y la línea de código donde se produce el error. Por ejemplo, se muestra el siguiente error cuando se intenta insertar una fila en una tabla inexistente en la base de datos:

```
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'ejemplo.depart
amento' doesn't exist
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)

    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)

    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Sou
rce)
     at java.lang.reflect.Constructor.newInstance(Unknown Source)
     at com.mysql.jdbc.Util.handleNewInstance(Util.java:411)
     at com.mysql.jdbc.Util.getInstance(Util.java:386)
     at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1052)
     at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3609)
     at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:3541)
     at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:2002)
     at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:2163)
     at com.mysql.jdbc.ConnectionImpl.execSQL(ConnectionImpl.java:2618)
     at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1749)
     at com.mysql.jdbc.StatementImpl.executeUpdate(StatementImpl.java:1666)
     at InsertarDep.main(InsertarDep.java:20)
```

Cuando se produce un error con **SQLException** podemos acceder a cierta información usando los siguientes métodos:

- *getMessage():* devuelve una cadena que describe el error.
- *getSQLState():* es una cadena que contiene un estado definido por el estándar X/OPEN SQL.
- *getErrorCode():* es un entero que proporciona el código de error del fabricante. Normalmente, este será el código de error real devuelto por la base de datos.

A continuación se utilizan esos métodos para visualizar los mensajes de error:

```
try
    {
        //Código
    }
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e)
{
        System.out.println("HA OCURRIDO UNA EXCEPCIÓN:");
        System.out.println("Mensaje:    "+ e.getMessage());
        System.out.println("SQL estado: "+ e.getSQLState());
        System.out.println("Cód error:  "+ e.getErrorCode());    }
```

El siguiente ejemplo muestra la salida que se produce cuando se intenta hacer SELECT de una tabla que no existe (en MySQL):

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje:    Table 'ejemplo.departamentoss' doesn't exist
SQL estado: 42S02
Cód error:  1146
```

En Oracle se visualizaría información diferente:

```
HA OCURRIDO UNA EXCEPCIÓN:
Mensaje:    ORA-00942: la tabla o vista no existe
SQL estado: 42000
Cód error:  942
```

2.12 PATRÓN MODELO-VISTA-CONTROLADOR. ACCESO A DATOS

El patrón MVC (*Model-View-Controller*) es un patrón de diseño que se utiliza como guía para el diseño de arquitecturas software que ofrecen una fuerte interactividad con el usuario y donde se requiere una separación de conceptos para que el desarrollo se realice más eficazmente facilitando la programación en diferentes capas de manera paralela e independiente. Este patrón organiza la aplicación en 3 bloques cada cual especializado en una tarea:

- **El Modelo**: representa los datos de la aplicación y sus reglas de negocio.
- **La Vista:** es la representación del modelo de forma gráfica para interactuar con el usuario, un ejemplo son los formularios de entrada y salida de información o páginas HTML con contenido dinámico.
- **El Controlador:** interpreta los datos que recibe del usuario analizando la petición, coordinando la vista y el modelo para que la aplicación produzca los resultados esperados.

Las ventajas de hacer uso de este patrón son:

- Separación de los datos de la representación visual de los mismos.
- Diseño de aplicaciones modulares.

- Reutilización de código.
- Facilidad para probar las unidades por separado.
- Facilita el mantenimiento y la detección de errores.

Entre las desventajas cabe destacar la complejidad que se agrega al sistema al separar los conceptos en capas o la cantidad de ficheros a desarrollar que se incrementa considerablemente.

La Figura 2.17 describe el flujo general de la solicitud de un usuario construida en una arquitectura MVC, como se puede observar, el controlador es el que dirige la aplicación. Los pasos son los siguientes:

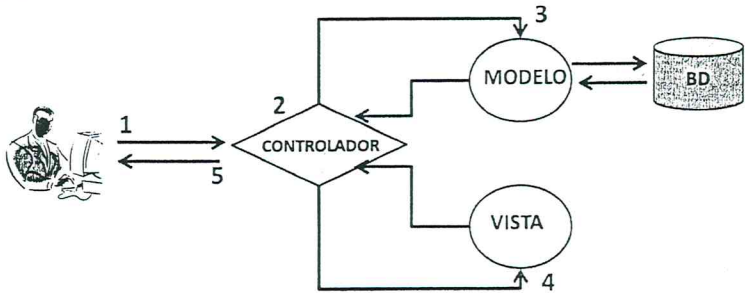


Figura 2.17. Flujo de solicitud en el MVC

1. Un usuario realiza una solicitud a través de la aplicación (pulsa un botón, un enlace,). La solicitud es dirigida al controlador.
2. El controlador examina la solicitud y decide qué regla de negocio aplicar, es decir, determinará el componente de negocio a aplicar para procesar la solicitud, este componente de negocio es el modelo.
3. El modelo contiene las reglas de negocio que procesan la solicitud y que dan lugar al acceso a los datos que necesita el usuario. Estos datos se devuelven al controlador.
4. El controlador toma los datos que devuelve el modelo y selecciona la vista en la que se van a presentar esos datos al usuario.
5. El controlador devuelve los resultados al usuario tras procesar la solicitud.

El patrón MVC es utilizado en múltiples frameworks: Java Enterprise Edition (J2EE), Apache Struts (para aplicaciones web J2EE), Ruby on Rails (para aplicaciones web con Ruby), Google Web Toolkit (GWT, para crear aplicaciones Ajax con Java), ASP.NET MVC Framework (Microsoft), etc. La mayoría de los frameworks para web implementan este patrón. Una aplicación de este patrón en aplicaciones web J2EE es lo que se conoce con el nombre de **Modelo 2**. Esta arquitectura se basa en los siguientes elementos:

- La utilización de Servlets para capturar las peticiones realizadas por el cliente web y redirigirlas a las páginas JSP adecuadas. Estos actúan como controlador.

- Páginas JSP utilizadas para mostrar la interfaz de usuario, implementan la vista. Estas páginas usan los JavaBeans como componente modelo.
- JavaBeans o POJOs (*Plain Old Java Object*) que implementan el modelo.

A continuación se muestra un ejemplo de una aplicación web MVC Java. La arquitectura **Modelo 2** de nuestra aplicación se muestra en la Figura 2.18. Para poder trabajar con ella necesitaremos instalar un contenedor web con soporte para Servlets y JSPs, en este ejemplo se realizará la aplicación sobre Tomcat.

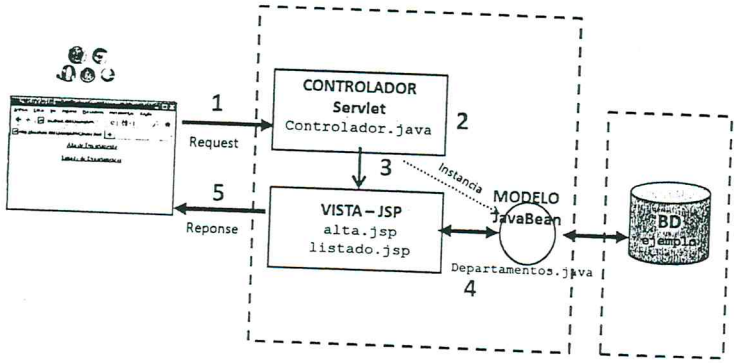


Figura 2.18. Arquitectura Modelo 2.

Se puede descargar la última versión desde la web http://tomcat.apache.org/, para el ejemplo se ha descargado la versión *apache-tomcat-7.0.23.zip* (*apache-tomcat-7.0.23.tar.gz* en Linux). La instalación es sencilla, solo hay que descomprimir el fichero que contiene una única carpeta de nombre *apache-tomcat-7.0.23*. Para iniciar el servidor se ejecuta el fichero *startup.bat* en sistemas Windows o *startup.sh* en Linux que se encuentran en la carpeta *\apache-tomcat-7.0.23\bin*.

Antes de empezar con la aplicación introducimos una serie de conceptos sobre los Servlets y las páginas JSP.Los Servlets son clases Java que no tienen el método *main()*, en su lugar se invocan otros métodos cuando se reciben las peticiones. Los métodos son: *init()*, *service()* y *destroy()*. El ciclo de vida de un Servlet se divide en varios pasos:

- El cliente solicita una petición a un servidor a través de una URL. El servidor recibe la petición.
- Si es la primera vez, el servidor carga el Servlet y se llama al método *init()* para iniciarlo.
- Se llama al método *service()* para procesar las peticiones de los clientes web.
- Se llama al método *destroy()* para eliminar al Servlet y liberar los recursos. El servidor los destruye porque cesan las llamadas desde el cliente, un temporizador del servidor así lo indica o el propio administrador lo decide.

Las páginas JSP nos permiten generar contenido dinámico en la web, son páginas web con etiquetas especiales y código Java incrustado. La diferencia con el Servlet es que en este el código es Java puro que recibe peticiones y genera una página web a partir de ellas. Una página JSP consta de 2 partes:

- HTML o XML para el contenido estático.
- Etiquetas JSP y scriplets escritos en lenguaje de programación Java para encapsular la lógica que genera el contenido dinámico.

El código de las partes dinámicas se encierra entre unas etiquetas especiales, la mayoría de las cuales empiezan con "<%" y terminan con "%>". Algunos elementos JSP son:

- Declaraciones JSP: dentro de las etiquetas *<%! código Java %>*
- Expresiones: dentro de las etiquetas *<%= código Java %>*
- Scriptlets: dentro de las etiquetas *<% código Java %>*
- Comentarios: se tienen los siguientes tipos:
  - De HTML: *<!-- comentario -->*
  - De JSP: *<%-- comentario --%>*
  - Del lenguaje de script Java: *<%// comentario línea %>* y *<%/* comentario varias líneas */%>*
- Directivas: dentro de las etiquetas *<%@ … %>* , por ejemplo: *<%@ page import = "ejemplo.*, java.util.*" %>*.
- Acciones: que permiten trabajar con componentes complementarios a la página JSP como applets, otras páginas JSP, javabeans, etc. Son las siguientes:
  - No asociadas a los javabeans:
    <jsp:include> … </jsp:include> o *<jsp:include … />* si solo tiene atributos
    <jsp:plugin> … </jsp:plugin>
    <jsp:forward> … </jsp:forward> o *<jsp:forward … />* si solo tiene atributos
  - Asociadas a los javabeans:
    <jsp:useBean … /> si solo tiene atributos o *<jsp:useBean …> … </jsp:useBean>*
    <jsp:setProperty … />
    <jsp:getProperty … />

Nuestra aplicación web estará formada por una serie de ficheros JSP, HTML, Java, XML y librerías JAR, la estructura de directorios y ficheros es la siguiente:

```
🗁 ejemploMVC
   alta.jsp
   index.html
   listado.jsp
   🗁 WEB-INF
      web.xml
      🗁 classes
         Controlador.class
         Controlador.java
         OperacionesBD.class
         OperacionesBD.java
         🗁 ejemplo
            Departamentos.class
            Departamentos.java
      🗁 lib
         mysql-connector-java-5.1.18-bin.jar
```

Esta estructura estará dentro de la carpeta *webapps* (*D:\apache-tomcat-7.0.23\webapps\ejemploMVC*). Hay algunos ficheros y carpetas con un significado especial:

- Carpeta **WEB-INF**: es imprescindible y su nombre debe aparecer siempre en mayúsculas. Contiene las siguientes carpetas y ficheros:
  - Carpeta **classes**: es necesaria si usamos ficheros *.class*. Contiene los paquetes de nuestras clases Java, reproduciendo la estructura de paquetes y subpaquetes. Es dentro de este directorio donde generalmente residen los Servlets (en el ejemplo por simplificar solo hay un paquete de nombre *ejemplo* donde está el JavaBean y 2 clases).
  - Carpeta **lib**: contiene todos los JAR que necesita la aplicación, en el ejemplo se necesita el conector Java para acceder a la base de datos MySQL.
  - Fichero **web.xml**: es el descriptor de despliegue de la aplicación e indica la ubicación de los servlets (mapeo) contenidos en la aplicación (en el ejemplo solo hay un Servlet que es *Controlador.java*). Puede contener otros parámetros para componentes adicionales y manejo de errores.

La aplicación visualizará una página web inicial (**index.html**) con dos enlaces, uno para realizar el alta de un departamento y el otro para realizar el listado de todos. El contenido del fichero es el siguiente:

```
<html>
<body>
  <p align='center'>
  <a href="/ejemploMVC/controlador?accion=alta">Alta de Departamento</a></p>
  <p align='center'>
  <a href="/ejemploMVC/controlador?accion=listado">Listado de Departamentos</a>
</p>
</body>
</html>
```

Desde los enlaces *href* se llama al controlador con la acción a realizar alta o listado. Para poder llevar a cabo esta acción los contenedores web tienen un fichero llamado **web.xml** que se encarga de mapear la URL. El contenedor web (Tomcat) lee el documento **web.xml** y realiza el mapeo entre el alias encontrado en el path de la URL y el Servlet en cuestión. El contenido del fichero es:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0"
  metadata-complete="true">
<servlet>
   <servlet-name>Controlador</servlet-name>
   <servlet-class>Controlador</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>Controlador</servlet-name>
   <url-pattern>/controlador</url-pattern>
</servlet-mapping>
```



```
<session-config>
   <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

Nos fijamos en los elementos *<servlet>* y *<servlet-mapping>*:

El elemento *<servlet>* define las características de un Servlet. Está compuesto por los elementos *<servlet-name>* donde se indica el nombre del Servlet (en el ejemplo *Controlador*) y *<servlet-class>* que indica el nombre de la clase Java que contiene el Servlet (*nombredirectorio.ficheroclase*, en el ejemplo el fichero *Controlador.class* no está dentro de ningún directorio (o paquete), por ello se escribe en este elemento solo *Controlador*).

El elemento *<servlet-mapping>* define la ubicación en términos de directorios de un sitio (URL). El Servlet de nombre *Controlador* (*<servlet-name>Controlador</servlet-name>*) será accedido cada vez que se acceda a la URL */controlador* (*<url-pattern>/controlador</url-pattern>*). *<url-pattern>* indica la forma en que se debe invocar al Servlet, en el documento HTML poníamos lo siguiente para invocar al Servlet: *<a href="/ejemploMVC/**controlador?accion=alta**">*, además se le envía el parámetro *accion*.

Controlador.java: es el Servlet controlador que maneja todas las solicitudes entrantes. Recibe un parámetro de nombre *accion*. Dependiendo del valor de este parámetro podrá realizar varias acciones: redirigir la petición a *alta.jsp*, redirigir la respuesta a *listado.jsp* enviando los departamentos a listar e invocar al método para la inserción de un departamento (cuyos datos se reciben de *alta.jsp*) en la base de datos y después redirigir la respuesta a *index.html*. Antes de realizar el listado carga los datos de la tabla DEPARTAMENTOS en un array, para ello utilizan el método *listarDep()* de la clase OperacionesBD. Los datos para insertar un departamento los recibe de la página *alta.jsp* mediante el atributo *depart*, la inserción la realiza invocando al método *insertaDepartamento()* de la clase OperacionesBD:

```
import ejemplo.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Controlador extends HttpServlet {
public void service(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException {

   //parámetro acción, se obtiene de la URL de index.html, puede ser 'alta' o
'listado'
   String op=request.getParameter("accion");

   //si se ha pulsado en alta de departamento se visualiza la pantalla de alta
   if(op.equals("alta")) response.sendRedirect("alta.jsp");

   //si se ha pulsado en listado, primero se cargan los datos de departamentos en
```

```
    //una lista y luego se envian a listado.jsp
    if(op.equals("listado")){
       OperacionesBD operBD=new OperacionesBD();
       ArrayList lista=operBD.listarDep(); //se cargan los datos de los dep
       request.setAttribute("departamentos",lista); //se preparan para enviar al jsp
       RequestDispatcher rd=request.getRequestDispatcher("/listado.jsp");
       rd.forward(request,response);
     }

    //se inserta departamento en la tabla y luego se visualiza index.html
    if(op.equals("insertar")){
       ejemplo.Departamentos dep=
              (ejemplo.Departamentos)request.getAttribute("depart");//obtener deps
       OperacionesBD operBD=new OperacionesBD();
       operBD.insertaDepartamento(dep); //se insertan en tabla  departamentos
       response.sendRedirect("index.html"); //se muestra la página inicial
     }
  }
}//fin de la clase Controlador
```

En el ejemplo solo se ha utilizado el método **service()** del Servlet. Este acepta 2 argumentos:

- *HttpServletRequest request*: en este argumento se recibe la petición del cliente.
- *HttpServletResponse response*: es la respuesta que da el servidor al cliente.

Algunos métodos usados en la petición (objeto *HttpServletRequest*) son:

- *request.getParameter("nombre_de_parametro")*: lee el parámetro enviado por el cliente, devuelve el valor del parámetro especificado o null si no existe.
- *request.setAttribute("nombre", objeto)*: se almacena un objeto en la sesión con el nombre indicado.
- *request.getAttribute("nombre")*: devuelve el atributo asociado al nombre especificado en esta sesión o null si no hay ningún objeto asociado bajo el nombre.

Algunos métodos usados en la respuesta al cliente (objeto *HttpServletResponse*) son:

- *response.sendRedirect("url")*: la respuesta al cliente es la url indicada, puede ser un fichero JSP, un Servlet o una página HTML.

La interfaz *RequestDispatcher* encapsula una referencia a otro recurso web. El método *getRequestDispatcher("url")* acepta una ruta de URL que haga referencia al recurso objetivo, la ruta debe ser absoluta, es decir, el nombre debe empezar por /. El método *forward(request,response)* permite reenviar la solicitud a otro servlet, página JSP o fichero HTML.

alta.jsp: Visualiza un formulario para realizar la entrada de datos de un departamento. Los datos del formulario se enviarán a través del JavaBean (el modelo) *Departamentos.java* al controlador; el nombre del parámetro que contendrá los datos del departamento a insertar es *depart* (*id="depart"*) y se usará en el controlador con el método *getAttribute()* (*request.getAttribute("depart")*) :


```
<html>
<head>
<title>ALTA DE DEPARTAMENTOS</title>
</head>
<!--Form de entrada de datos e Inserción en el JavaBean-clase Departamentos-->
<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />
<jsp:setProperty name="depart" property="*"/>
<%
 if(request.getParameter("deptno")!= null) {%>
<jsp:forward page="/controlador?accion=insertar"/>
<%}%>

<body>
  <center><h2>ENTRADA DE DATOS DE DEPARTAMENTOS</h2>
  </br>
  <form method="post">
          <p>Número de departamento: <input name="deptno" type="text" size="5"></p>
          <p>Nombre: <input name="dnombre" type="text" size="15"> </p>
          <p>Localidad: <input name="loc" type="text" size="15"> </p>
          <input type="submit" name="insertar" value="Insertar departamento.">
          <input type="reset" name="cancelar" value="Cancelar entrada.">
  </form>
  </center>
</body>
</html>
```

Un JavaBean es una clase Java que nos permite ocultar su implementación mostrando al exterior solo los métodos y propiedades públicos. Para acceder al Bean desde una página JSP se utilizan una serie de etiquetas, algunas son: *<jsp:useBean>*, *<jsp:setProperty>* ,*<jsp:getProperty>*, *<jsp:forward>*. En el ejemplo se han usado:

- *<jsp:useBean id="depart" scope="request" class="ejemplo.Departamentos" />*: *id* indica el nombre del Bean, *scope* es el alcance del Bean, un alcance "request" implica que el bean es accesible hasta otra JSP que haya sido invocada por medio de *jsp:forward* o *jsp:include*. El atributo *class* es el nombre de la clase.
- *<jsp:setProperty name="depart" property="*"/>*: el atributo *name* debe ser igual al especificado en *id*, con *property="*"* indicamos que se van a extraer todos los valores de la solicitud (*deptno, dnombre* y *loc*).
- *<jsp:forward page="/controlador?accion=insertar"/>*: permite que la solicitud sea enviada a otra página JSP, a un Servlet o a un recurso estático. En este caso se envia al controlador con el parámetro *accion* con valor *insertar* para que inserte los datos del Bean en la tabla.

Departamentos.java: es el JavaBean que facilita el intercambio de datos entre el controlador y el modelo y posteriormente entre el controlador y la vista:

```
package ejemplo;
    public class Departamentos {
          private byte deptno;
          private String dnombre;
          private String loc;
```

```
        public Departamentos() {}
        public Departamentos(byte deptno, final String dnombre, final String loc) {
                this.deptno = deptno;
                this.dnombre = dnombre;
                this.loc = loc;
                }
        public byte getDeptno() {return this.deptno;}
        public void setDeptno(byte deptno) {this.deptno = deptno;}
        public String getDnombre() {return this.dnombre;}
        public void setDnombre(String dnombre) {this.dnombre = dnombre;}
        public String getLoc() {return this.loc;}
        public void setLoc(String loc) {this.loc = loc;    }
        }//fin clase Departamentos
```

listado.jsp: Obtiene el listado de los departamentos, es llamado por el controlador y recibe el atributo *departamentos* conteniendo un array con los departamentos a listar:

```
<%@ page import="ejemplo.*,java.util.*"%>
<html><head><title>LISTADO DE DEPARTAMENTOS</title></head>
<body>
<center>
<h2>LISTADO DE DEPARTAMENTOS</h2>
<table border='1'>
<tr><th>Departamento</th><th>Nombre</th><th>Localidad</th></tr>
<%
ArrayList listadep=(ArrayList)request.getAttribute("departamentos");
if(listadep!=null)
  for(int i=0;i<listadep.size();i++){
   Departamentos d=(Departamentos)listadep.get(i);%>
   <tr><td><%=d.getDeptno()%></td>
   <td><%=d.getDnombre()%></td>
   <td><%=d.getLoc()%></td>
   </tr>
<%}%>
</table><br/><br/>
<a href="index.html">Inicio</a>
</center>
</body>
</html>
```

OperacionesBD.java: es la clase que realiza las operaciones contra la base de datos, la utiliza el controlador:

```
import ejemplo.*;
import java.sql.*;
import java.util.*;

public class OperacionesBD {
    //OBTENER LA CONEXIÓN
    public Connection getConnection(){
       Connection conexion=null;
       try{
           Class.forName("com.mysql.jdbc.Driver");//Cargar el driver
```



```
           conexion = DriverManager.getConnection
              ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
       }
       catch(Exception e){e.printStackTrace();}
       return conexion;
    }
    //LISTAR - Devuelve un array con la lista de departamentos
    public ArrayList listarDep(){
           ArrayList departamentos = new ArrayList ();
           try{
                Connection conexion =getConnection();
                Statement sentencia=conexion.createStatement();
                String sql= "SELECT * FROM departamentos";
                ResultSet resul=sentencia.executeQuery(sql);
                while(resul.next()){ //se crea un array con los datos de los
                                     departamentos
                ejemplo.Departamentos d=new ejemplo.Departamentos
                  resul.getByte("dept_no"),
                  resul.getString("dnombre"),resul.getString("loc"));

                  departamentos.add(d);//añadir dep al array
                }
           conexion.close();
           }
           catch(Exception e){e.printStackTrace();}
           return(departamentos);
    }//fin listarDep

    //INSERTAR - Recibe los datos del departamento a insertar en la tabla
    public void insertaDepartamento(ejemplo.Departamentos d){
           try{
                Connection conexion=getConnection();
                Statement sentencia=conexion.createStatement();
                String sql = "INSERT INTO departamentos VALUES('"
                       +d.getDeptno()+"','"+d.getDnombre()+"','"+d.getLoc()+"')";

                if (d.getDeptno () != 0) sentencia.execute(sql);
                System.out.println("SQL: "+sql);
                conexion.close();
           }
           catch(Exception e){e.printStackTrace();}
    }//fin insertaDepartamento
}//fin clase OperacionesBD
```

Para compilar las clases seguimos estos pasos:

1.Nos dirigimos a la carpeta donde están las clases:

```
CD D:\apache-tomcat-7.0.23\webapps\ejemploMVC\WEB-INF\classes
```

2.Definimos la variable CLASSPATH:

```
SET CLASSPATH=D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/classes/
```

```
ejemplo/;D:/apache-tomcat-7.0.23/webapps/ejemploMVC/WEB-INF/lib/mysql-connector-java-
5.1.18-bin.jar;D:/apache-tomcat-7.0.23/lib/servlet-api.jar;D:/apache-tomcat-7.0.23/
webapps/ejemploMVC/WEB-INF/classes/;
```

3.Compilamos la clase Departamentos que está en la carpeta de nombre *ejemplo*:

```
javac ejemplo/Departamentos.java
```

4.Compilamos las otras dos clases:

```
javac -Xlint OperacionesBD.java

javac Controlador.java
```

Una vez compiladas será necesario recargar la aplicación desde Tomcat. Para ello abrimos el navegador y escribimos la URL: http://localhost:8080/. Puede ocurrir que el servidor no se inicie (al ejecutar el fichero *startup.bat* o *startup.sh*) porque el puerto 8080 esté ocupado, en este caso cambiamos el puerto 8080 por otro que no se esté utilizando, por ejemplo 8081. Este cambio se hace en el fichero *D:\apache-tomcat-7.0.23\conf\server.xml* (línea 70). También será necesario crear un usuario para manejar el entorno web de Tomcat. Añadimos al fichero *D:\apache-tomcat-7.0.23\conf\tomcat-users.xml* las siguientes líneas entre las etiquetas *<tomcat-users>* *</tomcat-users>*:

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
```

Donde indicamos que el usuario *admin* con clave *admin* tendrá acceso a la gestión de las aplicaciones web de Tomcat. Hecho esto, iniciamos de nuevo el servidor y escribimos la URL http://localhost:8081/, a continuación pulsamos el botón *Manager App*; nos pedirá un nombre de usuario y su clave, escribimos *admin*. Desde la pantalla de gestión de aplicaciones de Tomcat recargamos nuestra aplicación */ejemploMVC*, véase Figura 2.19.

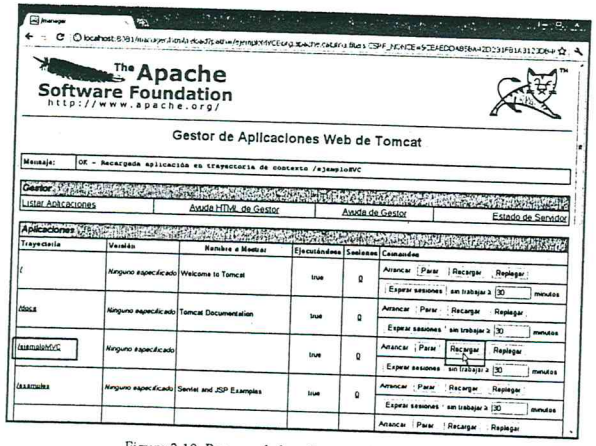


Figura 2.19. Recarga de la aplicación desde Tomcat.

Después escribimos la URL: http://localhost:8081/ejemploMVC/index.html para comenzar con la aplicación. En la pantalla inicial se visualizan 2 enlaces, el primero nos lleva a la pantalla de alta de departamentos el segundo nos lleva a la pantalla de listado, véase Figura 2.20.

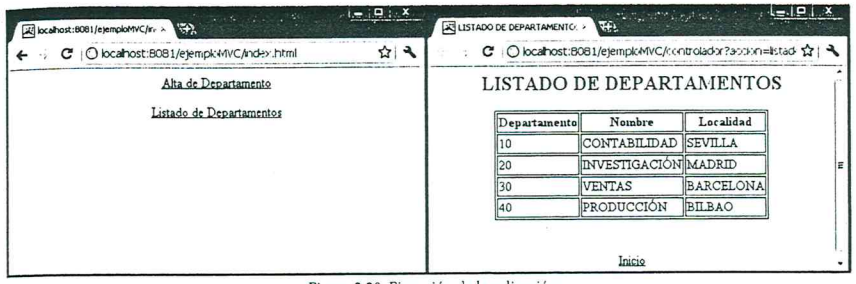


Figura 2.20. Ejecución de la aplicación.

2.13 EJERCICIOS

❶ Rellena la siguiente tabla en donde a la izquierda aparece el nombre de la base de datos y a la derecha debe aparecer la librería necesaria para la conexión mediante un programa Java:

Base de datos	Librería Java necesaria para la conexión
SQLite	sqlite-jdbc-3.7.2.jar
Apache Derby	
HSQLDB	
H2	
MySQL	
ORACLE	
Db4o	
Berkeley	

❷ Rellena la siguiente tabla resumen para que aparezca por cada base de datos estudiada el driver y la URL necesaria para el establecimiento de la conexión:

Base de datos	Driver	URL
SQLite	org.sqlite.JDBC	jdbc:sqlite:D:/db/SQLite/ejemplo.db
Apache Derby		
HSQLDB		
H2		
MySQL		
ORACLE		

❸ Cuál de las siguientes afirmaciones sobre JDBC NO es correcta:

a) JDBC define una API que pueden usar los programas Java para conectarse a bases de datos relacionales y orientadas a objetos.

b) JDBC no solo provee una interfaz 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.

c) JDBC dispone de la misma interfaz para todas las bases de datos.

d) Los tipos de conectores 3 y 4 se usan normalmente cuando el único sistema de acceso final al gestor de bases de datos es ODBC (es decir, no existen drivers disponibles para el SGBD).

e) Los tipos de conectores 1 y 2 exigen instalación de software en el puesto cliente. El tipo 4 no exige instalación en el cliente.

❹ ¿El siguiente código Java es correcto? Razona la respuesta:

```
import java.sql.*;
public class Main {
 public static void main(String[] args) {
    Class.forName("com.mysql.jdbc.Driver");
    Connection conexion = DriverManager.getConnection
       ("jdbc:mysql://localhost/ejemplo","ejemplo", "ejemplo");
    Statement sentencia = conexion.createStatement();
    ResultSet resul = sentencia.executeQuery ("SELECT * FROM empleados");
    while (resul.next())
    {
      System.out.println (resul.getInt("EMP_NO") + " * " +
      resul.getString("APELLIDO"));
    }
    resul.close();
    sentencia.close();
    conexion.close();
    }
}
```

❺ Crea las tablas PRODUCTOS, CLIENTES y VENTAS en las bases de datos MySQL, Oracle y SQLite. En MySQL crea una base de datos de nombre UNIDAD2, y usuario y clave con el mismo nombre. En Oracle crea un usuario de base de datos de nombre y clave UNIDAD2 y en SQLite guarda las tablas en un fichero que se llame UNIDAD2.DB. Las tablas son las siguientes:

PRODUCTOS:	CLIENTES:
ID numérico, clave primaria.	ID numérico, clave primaria.
DESCRIPCION varchar(50), no nulo.	NOMBRE varchar(50), no nulo.
STOCKACTUAL numérico.	DIRECCION varchar(50).
STOCKMINIMO numérico.	POBLACION varchar(50).
PVP numérico.	TELEF varchar(20).
	NIF varchar(10).

VENTAS:

IDVENTA numérico, clave primaria.
FECHAVENTA no nulo.
IDCLIENTE numérico, clave ajena a CLIENTES.
IDPRODUCTO numérico, clave ajena a PRODUCTOS.
CANTIDAD numérico.
Un cliente puede tener muchas ventas.

Una vez creadas, haz un programa Java que llene las tablas PRODUCTOS y CLIENTES (los datos a insertar de definen en el propio programa). El programa Java recibe un argumento al ejecutarlo desde la línea de comandos cuyo valor válido es 1, 2 o 3. Si el valor es 1 debes llenar las tablas de la base de datos de MySQL, si es 2 debes llenarlas de la base de datos ORACLE y si es 3 debes llenarlas en SQLite.

Una vez rellenas, visualiza los datos insertados y el número de filas que se han insertado en cada tabla.

Puedes crear las clases y métodos que creas convenientes.

❻ Partimos de las tablas anteriores, realiza un programa Java para insertar ventas en la tabla VENTAS. El programa recibe varios parámetros desde la línea de comandos:

- El primero indica la base de datos donde se insertará la venta (1,2, o 3, su significado es como en el ejemplo anterior).
- El segundo parámetro indica el identificador de venta.
- El tercer parámetro indica el identificador del cliente.
- El cuarto parámetro indica el identificador del producto.
- Y el quinto parámetro indica la cantidad.

Realiza las siguientes comprobaciones antes de insertar la venta en la tabla:

- El identificador de venta no debe existir en la tabla VENTAS.
- El identificador de cliente debe existir en la tabla CLIENTES.
- El identificador de producto debe existir en la tabla PRODUCTOS.
- La cantidad debe ser > que 0.

La fecha de venta es la fecha actual.

Una vez insertada la fila en la tabla visualizar un mensaje indicándolo. Si no se ha podido realizar la inserción visualizar el motivo (no existe el cliente, no existe el producto, cantidad menor o igual a 0 ...)

Ejecuta el programa e inserta varias ventas en las distintas bases de datos.

❼ Se pretende realizar un listado de las ventas de un cliente. El programa recibe dos parámetros, el primero indica la base de datos de la que se consultarán las ventas y el segundo el identificador del cliente cuyas ventas se van a consultar. El programa debe visualizar:

Ventas del cliente: *Nombre de cliente*
Venta: *idventa*
    Producto: *descripción* -- PVP: *pvp*
    Cantidad: *cantidad*
    Importe: *cantidad * pvp*
Venta: *idventa*
    Producto: *descripción* -- PVP: *pvp*
    Cantidad: *cantidad*
    Importe: *cantidad * pvp*
.....

Número total de ventas:___
Importe Total: ___

❽ Partiendo del ejemplo presentado en el epígrafe 2.12 (Patrón Modelo-Vista-Controlador) módifícalo y añade todo lo necesario para que nuestra aplicación web incluya dos nuevas opciones: Alta de empleados y Listado de empleados.

UNIDAD 3

HERRAMIENTAS DE MAPEO OBJETO RELACIONAL (ORM)

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Instalar y configurar una herramienta ORM.
- Definir los ficheros de mapeo.
- Aplicar mecanismos de persistencia.
- Desarrollar aplicaciones para insertar, modificar y recuperar objetos persitentes.
- Realizar consultas con el lenguaje de la herramienta ORM.
- Gestionar transacciones.

CONTENIDOS

3.1 INTRODUCCIÓN
3.2 CONCEPTO DE MAPEO OBJETO-RELACIONAL
3.3 HERRAMIENTAS ORM. CARACTERÍSTICAS
3.4 ARQUITECTURA HIBERNATE
3.5 INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE
3.5.1 Instalación del plugin
3.5.2 Configuración del driver MySQL
3.5.3 Configuración de Hibernate
3.5.4 Generar las clases de la base de datos
3.5.5 Primera consulta en HQL
3.5.6 Empezando a programar con Hibernate en Eclipse
3.6 ESTRUCTURA DE LOS FICHEROS DE MAPEO
3.7 CLASES PERSISTENTES
3.8 SESIONES Y OBJETOS HIBERNATE
3.8.1 Transacciones
3.8.2 Estados de un Objeto Hibernate
3.8.3 Carga de Objetos
3.8.4 Almacenamiento, modificación y borrado de objetos
3.9 CONSULTAS
3.9.1 Parámetros en las consultas
3.9.2 Consultas sobre clases no asociada
3.9.3 Funciones de grupo en las consultas
3.9.4 Objetos devueltos por las consultas
3.10 INSERT, UPDATE y DELETE
3.11 RESUMEN DEL LENGUAJE HQL
3.11.1 Asociaciones y uniones (joins)
3.12 EJEMPLOS RESUELTOS
3.13 EJERCICIOS

3.1 INTRODUCCIÓN

En este tema aprenderemos a acceder a una base de datos relacional utilizando el lenguaje orientado a objetos Java. Para acceder de forma efectiva a la base de datos desde un contexto orientado a objetos, es necesaria una interfaz que traduzca la lógica de los objetos a la lógica relacional. Esta interfaz se llama ORM *(Object Relational Mapping)* y es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales a representaciones de objetos; es decir, las tablas de nuestra base de datos pasan a ser clases y las filas (o registros) objetos que podemos manejar con facilidad.

3.2 CONCEPTO DE MAPEO OBJETO-RELACIONAL

Según la Wikipedia el **mapeo objeto-relacional** (más conocido por su nombre en inglés, *Object-Relational mapping,* o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia, véase Figura 3.1. En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos, aunque algunos programadores prefieren crear sus propias herramientas ORM.

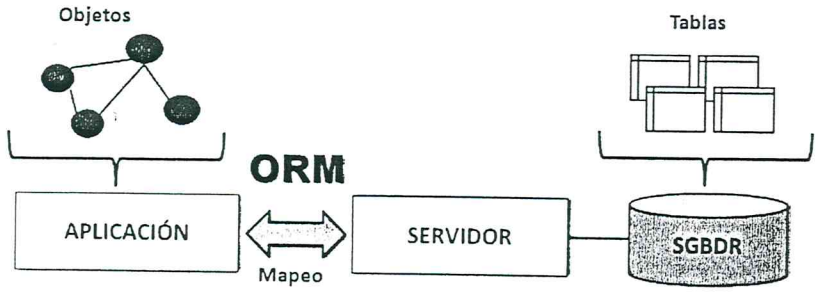


Figura 3.1. Mapeo objeto-relacional.

3.3 HERRAMIENTAS ORM. CARACTERÍSTICAS

Las herramientas ORM nos permiten crear una capa de acceso a datos; una forma sencilla y válida de hacerlo es crear una clase por cada tabla de la base de datos y mapearlas una a una. Estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, solo será necesario cambiar alguna línea en el fichero de configuración.

Algunas de las ventajas que aportan estas herramientas son:

- Ayudan a reducir el tiempo de desarrollo de software.
- Abstracción de la base de datos.
- Reutilización.
- Permiten la producción de mejor código.
- Son independientes de la Base de Datos.
- Lenguaje propio para realizar las consultas.
- Incentivan la portabilidad y escalabilidad de los programas de software.

Uno de los inconvenientes es que las aplicaciones son algo más lentas debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

Existen muchas herramientas ORM, algunas son: *Doctrine, Propel* o *ADOdb Active Record* para incluir en proyectos PHP, *LINQ* desarrollado por Microsoft para el mapeo objeto-relacional para los lenguajes Visual Basic .Net y C#, *Hibernate* desarrollado para la tecnología Java y disponible también para la tecnología .NET con el nombre de *Nhibernate*, es software libre bajo la licencia GNU LGPL. Además de estos que hemos nombrado hay otros muchos como pueden ser *QuickDB, iPersist, Java Data Objects, Oracle Toplink,* etc. En este tema estudiaremos Hibernate que es uno de los ORM más populares.

Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java (y disponible también para .Net) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML) que permiten establecer estas relaciones, Figura 3.2.

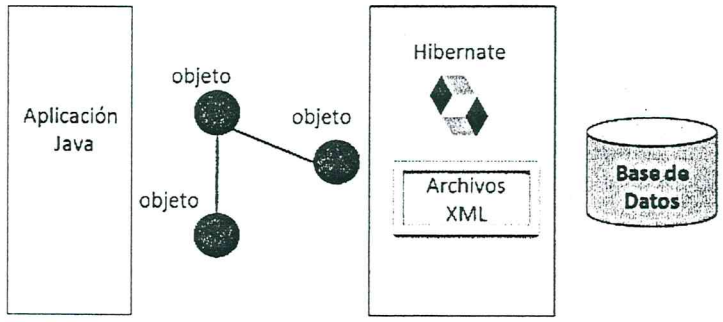


Figura 3.2. Hibernate, mapeo objeto-relacional.

Se está convirtiendo en el estándar de facto para almacenamiento persistente cuando queremos independizar la capa de negocio del almacenamiento de la información. Esta capa de persistencia permite abstraer al programador Java de las particularidades de una determinada base de datos proporcionando clases que envolverán los datos recuperados de las filas de las tablas. Hibernate busca

solucionar la diferencia entre los dos modelos de datos usados para organizar y manipular datos: el modelo de objetos proporcionado por el lenguaje de programación y el modelo relacional usado en las bases de datos.

Con Hibernate no emplearemos habitualmente SQL para acceder a datos sino que el propio motor de Hibernate, mediante el uso de factorías (patrón de diseño *Factory*) y otros elementos de programación construirá esas consultas para nosotros. Hibernate pone a disposición del diseñador un lenguaje llamado **HQL** *(Hibernate Query Language)* que permite acceder a datos mediante POO.

3.4 ARQUITECTURA HIBERNATE

Hibernate parte de una filosofía de mapear objetos Java normales o más conocidos en la comunidad como "POJOs" *(Plain Old Java Objects)*. Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe mantener una conversación con el motor de Hibernate mediante un objeto especial que es la **sesión** (clase **Session**) (equiparable al concepto de conexión de JDBC). Igual que con las conexiones JDBC hemos de crear y cerrar sesiones. La arquitectura Hibernate se muestra en la Figura 3.3.

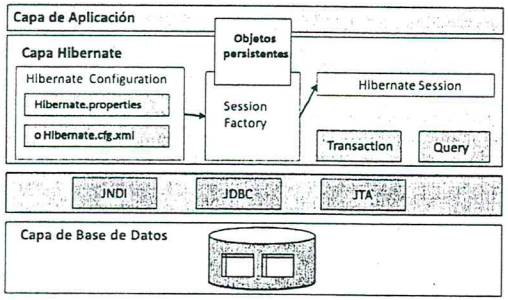


Figura 3.3. Arquitectura Hibernate.

La clase **Session** *(org.hibernate.Session)* ofrece métodos como *save(Object objeto), createQuery(String consulta), beginTransaction(), close()*, etc. para interactuar con la BD tal como se hace con una conexión JDBC, con la diferencia de que resulta más simple; por ejemplo, guardar un objeto consiste en algo así como *session.save(miObjecto)*, sin necesidad de especificar una sentencia SQL. Una instancia de **Session** no consume mucha memoria y su creación y destrucción es muy barata. Esto es importante, ya que nuestra aplicación necesitará crear y destruir sesiones todo el tiempo, quizá en cada petición. Puede ser útil pensar en una sesión como en una caché o colección de objetos cargados (a o desde una base de datos) relacionados con una única unidad de trabajo. Hibernate puede detectar cambios en los objetos pertenecientes a una unidad de trabajo.

Las interfaces de Hibernate son las siguientes:

- La interfaz **SessionFactory** *(org.hibernate.SessionFactory)* permite obtener instancias **Session.** Esta interfaz debe compartirse entre muchos hilos de ejecución. Normalmente hay una única **SessionFactory** para toda la aplicación, creada durante la inicialización de la misma, y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Si la aplicación accede a varias bases de datos se necesitará una **SessionFactory** por cada base de datos.
- La interfaz **Configuration** *(org.hibernate.cfg.Configuration)* se utiliza para configurar Hibernate. La aplicación utiliza una instancia de **Configuration** para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate, y a continuación crea la **SessionFactory.**
- La interfaz **Query** *(org.hibernate.Query)* permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escriben en HQL o en el dialecto SQL nativo de la base de datos que estemos utilizando. Una instancia **Query** se utiliza para enlazar los parámetros de la consulta, limitar el número de resultados devueltos y para ejecutar dicha consulta.
- La interfaz **Transaction** *(org.hibernate.Transaction)* nos permite asegurar que cualquier error que ocurra entre el inicio y final de la transacción produzca el fallo de la misma

Hibernate hace uso de APIs de Java, tales como JDBC, JTA *(Java Transaction Api)* y JNDI *(Java Naming Directory Interface)*.

La Figura 3.4 muestra cómo sería una aplicación con Hibernate.

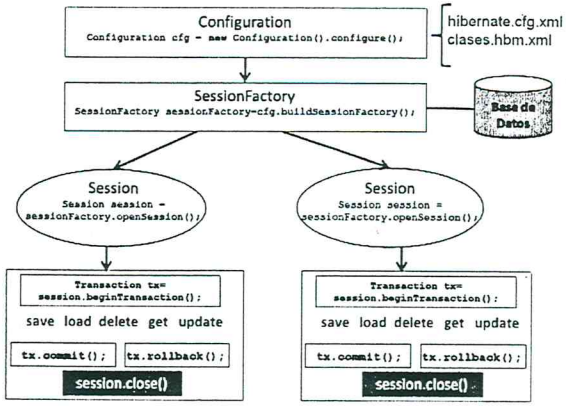


Figura 3.4. Aplicación con Hibernate.

3.5 INSTALACIÓN Y CONFIGURACIÓN DE HIBERNATE

En esta apartado vamos a instalar y configurar Hibernate en el entorno Eclipse, se ha utilizado el IDE Eclipse Helios y la versión *Eclipse IDE for Java EE Developers.* El paquete Eclipse Helios se puede descargar desde la URL: http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/heliossr2. Necesitamos tener **JDK** *(Java Development Kit/ Kit de Desarrollo para Java)* instalado, preferiblemente **JDK 6.** Se puede descargar desde la URL http://www.oracle.com/technetwork/java/javase/downloads/jdk6-downloads-1637591.html.

Para realizar las pruebas, en Windows se ha instalado el paquete *eclipse-jee-helios-SR2-win32.zip* y se ha descomprimido en la carpeta *D:/eclipse*. En Linux se ha instalado *eclipse-jee-helios-SR2-linux-gtk.tar.gz* y se ha descomprimido en la carpeta */opt/eclipse*. A lo largo del tema todos los paquetes que instalemos colgarán de la carpeta *eclipse*. El proceso de instalación se va a realizar en el sistema operativo Windows, en Linux los pasos son similares.

Para los ejemplos vamos a utilizar la base de datos MySQL de nombre **EJEMPLO,** creada en la unidad anterior, recordemos que su propietario era el usuario *ejemplo* y la clave la misma que el nombre del usuario. Se creaban las tablas EMPLEADOS y DEPARTAMENTOS, las relaciones se muestran en la Figura 3.5. La creación de las tablas es la siguiente:

```
CREATE TABLE departamentos (
  dept_no   TINYINT(2) NOT NULL
                       PRIMARY KEY,
  dnombre   VARCHAR(15),
  loc       VARCHAR(15)
) ENGINE=InnoDB;
```

```
CREATE TABLE empleados (
  emp_no     SMALLINT(4)  NOT NULL
                          PRIMARY KEY,
  apellido   VARCHAR(10),
  oficio     VARCHAR(10),
  dir        SMALLINT,
  fecha_alt  DATE       ,
  salario    FLOAT(6,2),
  comision   FLOAT(6,2),
  dept_no    TINYINT(2) NOT NULL
  REFERENCES departamentos(dept_no)
) ENGINE=InnoDB;
```

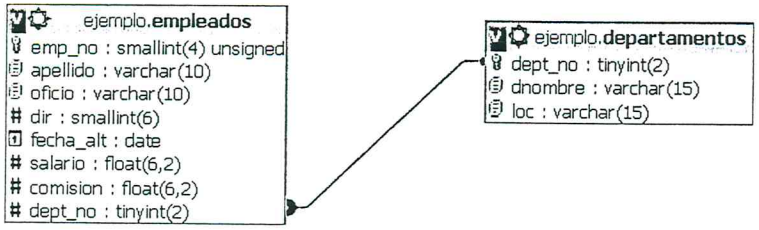


Figura 3.5. Base de datos EJEMPLO.

También a lo largo de los ejemplos, trabajaremos con las tablas Oracle EMPLEADOS y DEPARTAMENTOS propiedad del usuario *ejemplo* y clave con el mismo nombre de usuario. La creación de las tablas en Oracle es la siguiente:

```
CREATE TABLE departamentos (
 dept_no   NUMBER(2) NOT NULL
                     PRIMARY KEY,
 dnombre   VARCHAR2(15),
 loc       VARCHAR2(15)
);
```

```
CREATE TABLE empleados (
 emp_no     NUMBER(4)  NOT NULL
                       PRIMARY KEY,
 apellido   VARCHAR2(10),
 oficio     VARCHAR2(10),
 dir        NUMBER(4),
 fecha_alt  DATE       ,
 salario    NUMBER(6,2),
 comision   NUMBER(6,2),
 dept_no    NUMBER(2) NOT NULL
 REFERENCES departamentos(dept_no)
);
```


3.5.1 INSTALACIÓN DEL PLUGIN

Para instalar el plugin de Hibernate para Eclipse se necesita tener conexión a Internet. Primero iniciamos Eclipse. A continuación pulsamos en la opción del menú horizontal **Help-> Install New Software,** rellenamos el campo **Work With** con la siguiente URL: http://download.jboss.org/jbosstools/updates/stable/ y pulsamos el botón *Add,* nos pide un nombre, escribimos por ejemplo *Hibernate* y pulsamos el botón OK, véase Figura 3.6.

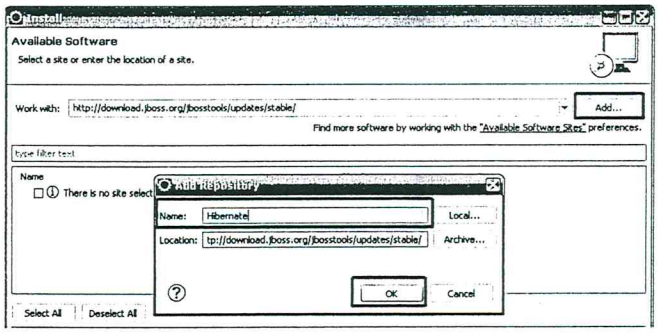


Figura 3.6. Instalar plugin de Hibernate.

Al rato aparece la lista de plugins. Pulsamos en la flechita que aparece a la izquierda de **Data Services** y seleccionamos solo **Hibernate Tools,** el resto de opciones las desmarcamos. A continuación pulsamos al botón *Next* (Figura 3.7), comienza el proceso de descarga.

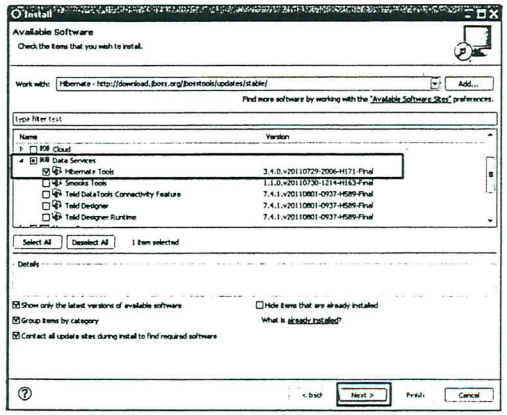


Figura 3.7. Seleccionar plugin Hibernate Tools.

Una vez descargado se visualiza una ventana con los detalles del elemento a instalar, pulsamos de nuevo *Next*. A continuación aceptamos la licencia y pulsamos el botón *Finish*, Figura 3.8. El proceso de instalación comienza, puede tardar un rato.

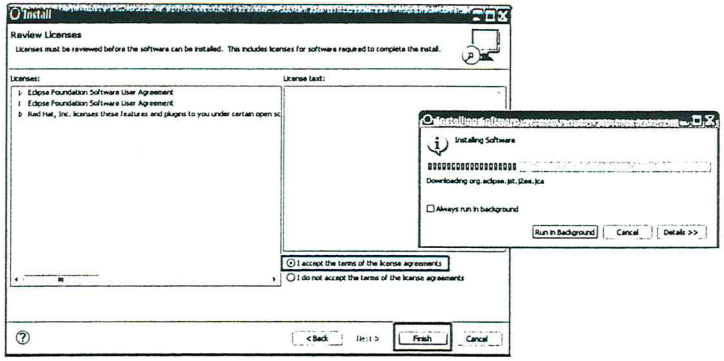


Figura 3.8. Aceptar licencia del software a instalar.

Una vez instalado nos pide reiniciar Eclipse. Para comprobar que se ha instalado correctamente podemos pulsar en la opción de menú **Windows -> Show View -> Other**, deben aparecer las opciones de Hibernate. También se puede comprobar desde el menú **Window-> Open Perspective-> Other-> Hibernate,** Figura 3.9.

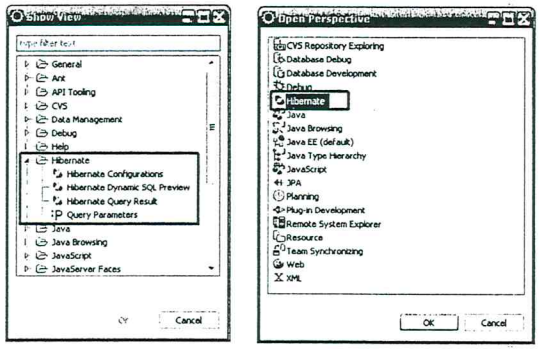


Figura 3.9. Comprobar la instalación de Hibernate.

3.5.2 CONFIGURACIÓN DEL DRIVER MYSQL

Una vez instalado Hibernate, el siguiente paso es configurarlo para que se comunique con MySQL. Utilizaremos la base de datos EJEMPLO. En primer lugar hemos de descargarnos el driver MySQL desde la URL http://dev.mysql.com/downloads/connector/j/, en este caso se ha descargado el archivo *mysql-connector-java-5.1.18.zip* (para Linux se descarga el fichero *mysql-connector-*

java-5.1.18.tar.gz). Se ha descomprimido y se ha instalado en la carpeta donde está instalado Eclipse, en este caso en *D:\eclipse* (en Linux */opt/eclipse)*.

A continuación desde el menú: **Window -> Preferences -> Data Management -> Connectivity -> Driver Definitions** se pulsa el botón *Add:*

- Desde la pestaña *Name/Type* se selecciona *MySQL JDBC Driver* en la versión 5.1.
- Desde la pestaña *JAR List* pulsamos el botón *Add Jar/Zip* para localizar el conector anterior, se debe seleccionar el fichero *mysql-connector-java-5.1.18-bin.jar* que está en la carpeta que se descomprimió del driver *(D:\eclipse\mysql-connector-java-5.1.18)*. Si se visualiza el driver *mysql-connector-java-5.1.0-bin.jar* lo eliminaremos pulsando el botón *Remove Jar/Zip,* Figura 3.10. Se pulsa el botón *OK*. Y de nuevo *OK* para salir de esta pantalla.

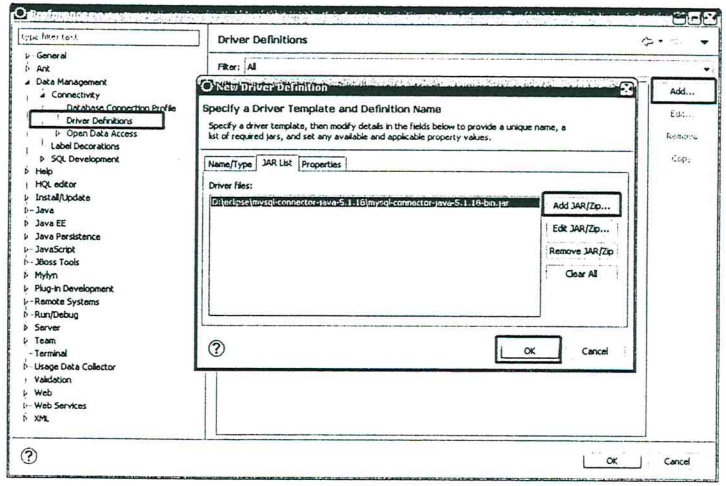


Figura 3.10. Instalación del driver MySQL.

A continuación vamos a crear un proyecto (le damos el nombre de *Proyecto1*) y configuraremos Hibernate para que se comunique con MySQL y nos cree las clases correspondientes de cada tabla de la base de datos EJEMPLO. Pulsamos en el menú **File-> New -> Project->Java Project** y pulsamos el botón *Next,* nos pide el nombre del proyecto, por ejemplo escribimos *Proyecto1* y pulsamos el botón *Finish*. Pregunta si queremos abrir la perspectiva asociada al proyecto, pulsamos el botón *Yes*.

Ahora hemos de agregar el driver MySQL, para ello seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Paths-> Add Libreries.** Se visualiza una ventana desde la que hemos de elegir la opción **Connectivity Driver Definition** y pulsar el botón *Next*. A continuación se visualiza una nueva ventana donde aparecen las opciones disponibles para conectarnos a una fuente de datos, elegimos *MySQL JDBC Driver* y pulsamos el botón *Finish,* véase Figura 3.11.

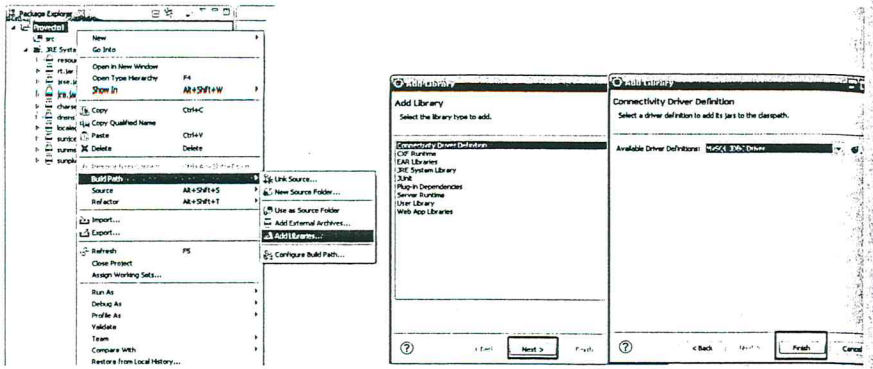


Figura 3.11. Agregar el driver MySQL al proyecto.

El proyecto debe mostrar un aspecto similar al de la Figura 3.12.

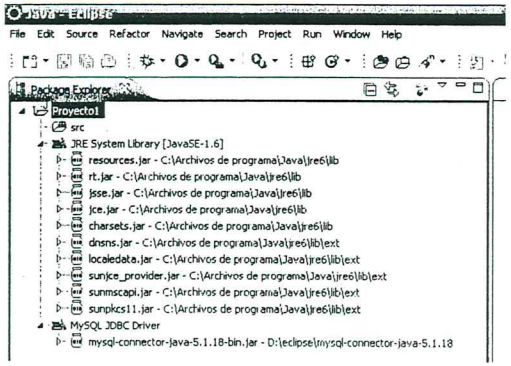


Figura 3.12. Proyecto con el driver MySQL.

3.5.3 CONFIGURACIÓN DE HIBERNATE

Una vez que tenemos el driver MySQL en nuestro proyecto hemos de crear el fichero de configuración de Hibernate *hibernate.cfg.xml*. Seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y pulsamos sobre: **New-> Other->Hibernate->Hibernate Configuration File(cfg.xml)**, Figura 3.13. Este fichero es un XML que contiene todo lo necesario para realizar la conexión a la base de datos.

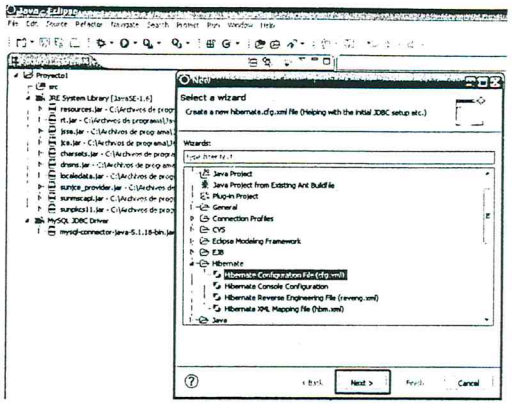


Figura 3.13. Crear fichero hibernate.cfg.xml.

Se pulsa el botón *Next*. A continuación nos pide dónde crear el fichero, en este ejemplo lo crearemos en la carpeta por defecto llamada **src**, pulsamos de nuevo *Next*, véase Figura 3.14. Seguidamente hemos de escribir los datos para poder realizar la conexión a la base de datos, véase Figura 3.15.

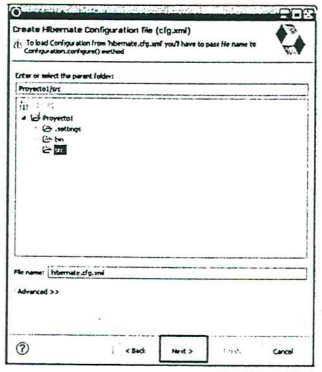


Figura 3.14. Carpeta para *hibernate.cfg.xml*.

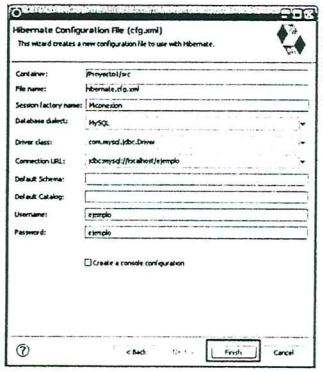


Figura 3.15. Datos para la configuración del fichero *cfg.xml*.

Los campos a rellenar son los siguientes:

- *Session factory name:* aquí se escribe el nombre de nuestra conexión a MySQL, en este caso se le ha dado el nombre *Miconexion*.
- *Database Dialect:* desde esta lista se elige como se comunica JDBC con la base de datos, se elige MySQL.
- *Driver Class:* se selecciona la clase de JDBC que se va a usar para la conexión, se elige *com.mysql.jdbc.Driver*.

- *Conection URL:* se elige la ruta de conexión a nuestra base de datos, se elige de la lista *jdbc:mysq l://<hostname>/<database>* y se cambia por *jdbc:mysql://localhost/ejemplo.*
- *Username:* usuario que se conectará a la base de datos EJEMPLO, en este caso es *ejemplo.*
- *Password:* clave del usuario que realiza la conexión con MySQL, en este caso es *ejemplo.*

Para terminar se pulsa el botón *Finish*. Se visualiza el editor de configuración de Hibernate, véase Figura 3.16.

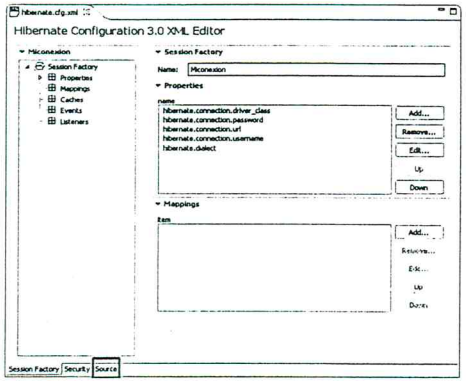


Figura 3.16. Editor de configuración de Hibernate.

Desde la pestaña *Source* se puede editar el fichero XML **cfg.xml** generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="Miconexion">
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.password">ejemplo</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost/ejemplo
        </property>
        <property name="hibernate.connection.username">ejemplo</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect
        </property>
    </session-factory>
</hibernate-configuration>
```

Una vez creado el fichero *hibernate.cfg.xml* hemos de crear el fichero *XML Hibernate Console Configuration.* Seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos: **New-> Other->Hibernate->Hibernate Console Configuration,** se pulsa el botón *Next,* Figura 3.17.

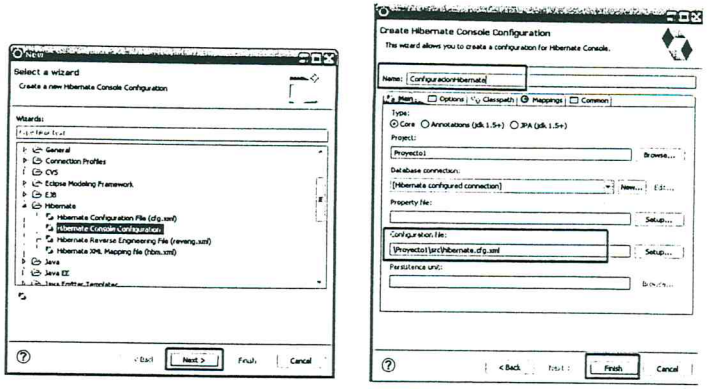


Figura 3.17. Crear fichero *Hibernate Console Configuration.*

Se abre una nueva ventana, en el campo *Name* escribimos un nombre para nuestra configuración, por ejemplo *ConfiguracionHibernate,* nos aseguramos de que en el campo *Configuration file* debe aparecer el fichero de configuración creado anteriormente *(hibernate.cfg.xml).* Pulsamos *Finish* para terminar el proceso de creación.

Por último, hemos de crear el fichero XML *Hibernate Reverse Engineering (reveng.xml)* que es el encargado de crear las clases de nuestras tablas MySQL. Pulsamos el botón derecho del ratón y seleccionamos: **New-> Other->Hibernate-> Hibernate Reverse Engineering File(reveng.xml).** Pulsamos el botón *Next* y nos pide que indiquemos dónde se va a guardar el fichero, se debe guardar en la misma carpeta que el fichero *hibernate.cfg.xml,* en este caso en la carpeta **src,** Figura 3.18.

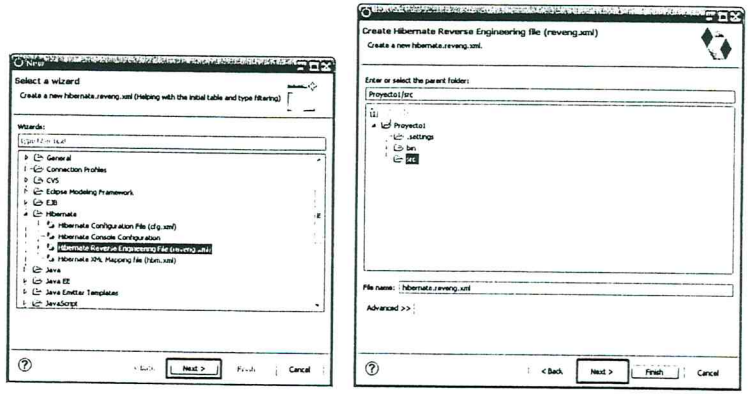


Figura 3.18. Crear fichero *reveng.xml.*

Pulsamos *Next*, se visualiza una nueva ventana desde donde indicaremos las tablas que queremos mapear. Desde la lista *Console configuration* seleccionamos el nombre que dimos al fichero *Hibernate Console Configuration*, en este caso *ConfiguracionHibernate*, y pulsamos el botón *Refresh* para que muestre la base de datos EJEMPLO y sus tablas, véase Figura 3.19. Seleccionamos una a una o todas las tablas y pulsamos el botón *Include*. Para finalizar pulsamos el botón *Finish*.

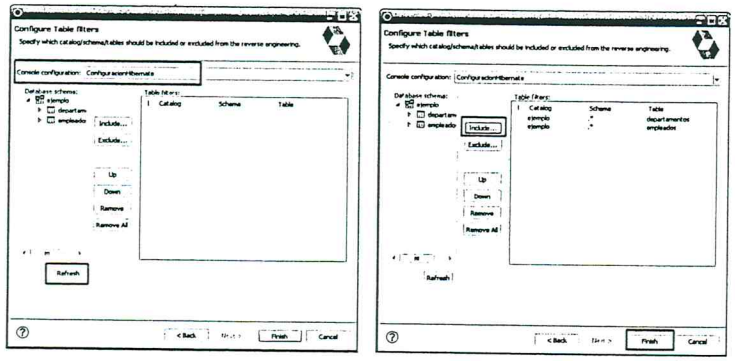


Figura 3.19. Tablas a mapear con Hibernate.

Se visualiza el editor de *Hibernate Reverse Engineering*, véase Figura 3.20.

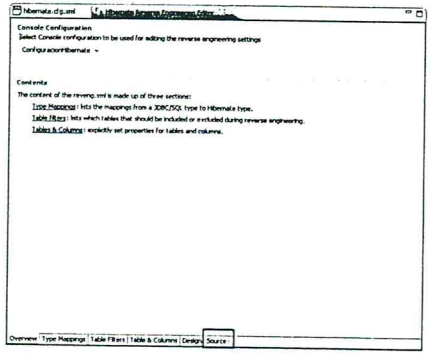


Figura 3.20. Editor de Hibernate Reverse Engineering.

Desde la pestaña *Source* se puede editar el fichero XML **reveng.xml** generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering PUBLIC "-//Hibernate/Hibernate Reverse
Engineering DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-reverse-
engineering-3.0.dtd" >
```



```
<hibernate-reverse-engineering>
  <table-filter match-catalog="ejemplo" match-name="departamentos"/>
  <table-filter match-catalog="ejemplo" match-name="empleados"/>
</hibernate-reverse-engineering>
```

Actividad 1: Crea un nuevo proyecto Java para acceder a Oracle usando Hibernate, le damos el nombre *Proyecto1OracleHibernate*. Accederemos a las tablas EMPLEADOS Y DEPARTAMENTOS del usuario EJEMPLO creado en la Unidad 2.Seguimos el Epígrafe 3.5.2 para configurar el driver Oracle:

- Desde la pestaña *Name/Type* se selecciona *Oracle Thin Driver* en la versión 10.
- Desde la pestaña *JAR List* pulsamos el botón *Add Jar/Zip* para localizar el conector *classes12.jar*. Si se visualiza el driver *ojdbc14.jar* lo eliminaremos pulsando el botón *Remove Jar/Zip*.

Seguimos el Epígrafe 3.5.3 para crear el fichero de configuración de Hibernate, escribimos los siguientes valores:

- *Session factory name:* MiConexionOracle
- *Database Dialect:* Oracle 10g
- *Driver Class:* oracle.jdbc.driver.OracleDriver
- *Conection URL:* jdbc:oracle:thin:@localhost:1521:XE
- *Default Schema:* EJEMPLO
- *Username:* EJEMPLO
- *Password:* EJEMPLO

3.5.4 GENERAR LAS CLASES DE LA BASE DE DATOS.

El siguiente paso es generar las clases de nuestra base de datos EJEMPLO. Para ello pulsamos en la flechita situada a la derecha del botón *Run As* y seleccionamos **Hibernate Code Generation Configurations,** Figura 3.21.

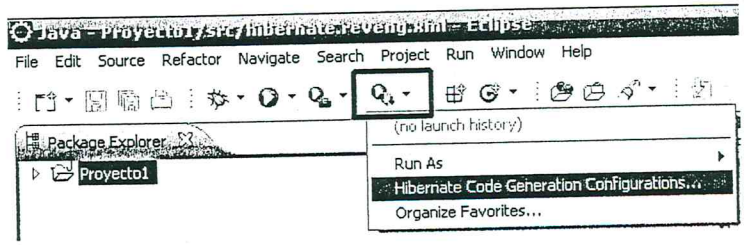


Figura 3.21. Botón Run As...

Desde la ventana que aparece hacemos doble clic en el nombre: Hibernate Code Generation Configurations... que se visualiza en el marco de la izquierda. Al hacer doble clic se visualizan varias pestañas. Desde la pestaña *Main* configuramos los siguientes campos, véase Figura 3.22:

- *Console Configuration:* seleccionamos *ConfiguracionHibernate.*
- *Output directory:* debe ser la carpeta src, *\Proyecto1\src.*
- *Package:* escribimos *primero.*
- *reveng.xml:* localizamos el fichero *reveng.xml* creado anteriormente. Pulsando en el botón *Setup* se puede localizar.

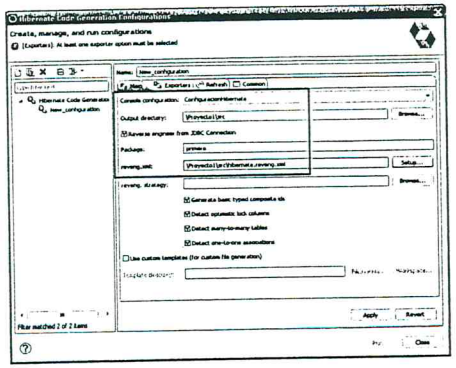


Figura 3.22. Pestaña *Main.*

Desde la pestaña *Exporters* se indica los ficheros que queremos generar, se marcan las casillas: *Domain code, Hibernate XML Mappings* e *Hibernate XML Configuration.* Una vez seleccionadas se pulsa el botón *Apply* y posteriormente se pulsa *Run,* Figura 3.23.

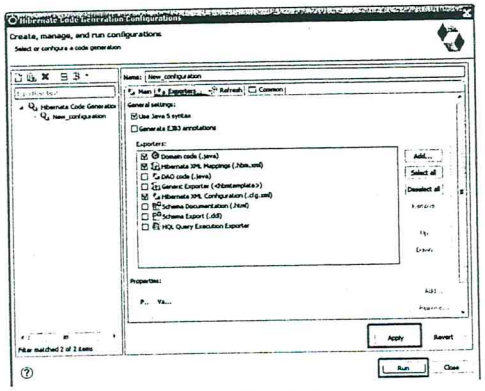


Figura 3.23. Pestaña *Exporters.*

AL ejecutarse nos genera un paquete llamado **primero** con las clases Java de las tablas EMPLEADOS *(Empleados.java)* y DEPARTAMENTOS *(Departamentos.java)* que contienen los *getters* y *setters* de cada campo de la tabla; y los ficheros XML, *Departamentos.hbm.xml* y *Empleados.hbm.xml* que contienen la información del mapeo de su respectiva tabla, Figura 3.24.

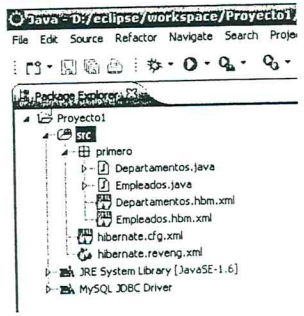


Figura 3.24. *Proyecto1* con el paquete creado.

Pueden aparecer algunos errores *warning* en las clases Departamentos y Empleados. En el ejemplo aparece el error *"The serializable class Departamentos does not declare a static final serialVersionUID field of type long"*, esto es porque las clases implementan la interfaz **Serializable.** Para solucionarlo se pulsa con el ratón sobre el símbolo de *warning* para que nos dé las posibles soluciones. Por ejemplo, se añade esta declaración de variable:

```
private static final long serialVersionUID = 1L;
```

O bien, se añade esta otra declaración:

```
@SuppressWarnings("serial")
```

Actividad 2: Sigue los pasos anteriores en el proyecto *Proyecto1OracleHibernate* para generar las clases de las tablas Oracle de EMPLEADOS y DEPARTAMENTOS.

Visualiza después el código de las clases *Empleados.java* y *Departamentos.java,* fíjate en los atributos generados y compáralos con los generados en el ejemplo de MySQL:

Empleados.java:

ATRIBUTOS GENERADOS PARA ORACLE	ATRIBUTOS GENERADOS PARA MYSQL

`private short empNo;`
`private Departamentos departamentos;`
`private String apellido;`
`private String oficio;`
`private Short dir;`
`private Date fechaAlt;`
`private BigDecimal salario;`
`private BigDecimal comision;`
	`private short empNo;`
	`private String apellido;`
	`private String oficio;`
	`private Short dir;`
	`private Date fechaAlt;`
	`private Float salario;`
	`private Float comision;`
	`private byte deptNo;`

Departamentos.java:

ATRIBUTOS GENERADOS PARA ORACLE	ATRIBUTOS GENERADOS PARA MYSQL

`private byte deptNo;`
`private String dnombre;`
`private String loc;`
`private Set<Empleados> empleadoses = new HashSet<Empleados>(0);`
	`private byte deptNo;`
	`private String dnombre;`
	`private String loc;`

Se observa variación en los tipos Java generados según el sistema de base de datos y para el caso de Oracle en la clase *Empleados.java* el atributo de número de departamento *(deptNo)* no aparece, en su lugar aparece un atributo de nombre *departamentos* que es un objeto Departamentos y que hace referencia al departamento del empleado.
En la clase *Departamentos.java* aparece un nuevo atributo *(empleadoses)* para almacenar los empleados del departamento. Mediante estos atributos se establecen las relaciones entre estos objetos. También se generan los métodos *get* y *set* para estos atributos:
Departamentos.java:

```
public Set<Empleados> getEmpleadoses() {
        return this.empleadoses;
}

public void setEmpleadoses(Set<Empleados> empleadoses) {
        this.empleadoses = empleadoses;
}
```

Empleados.java:

```
public Departamentos getDepartamentos() {
        return this.departamentos;
}

public void setDepartamentos(Departamentos departamentos) {
        this.departamentos = departamentos;
}
```

3.5.5 PRIMERA CONSULTA EN HQL

A continuación vamos a realizar consultas en HQL para comprobar si la conexión a la base de datos funciona correctamente. Abrimos la perspectiva de Hibernate desde la opción de menú **Window-> Open Perspective-> Other-> Hibernate.** Desde la pestaña *Hibernate* pulsamos en nuestra configuración *ConfiguracionHibernate* y pulsamos con el botón derecho del ratón en **Database-> HQL Editor,** véase Figura 3.25.

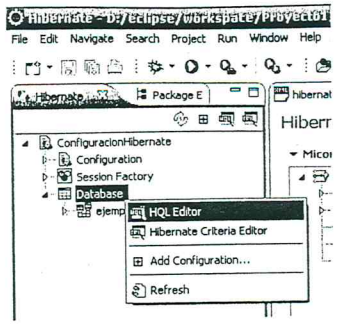


Figura 3.25. Pestaña *Hibernate.*

Nos pregunta si queremos abrir la sesión, pulsamos en el botón *Yes.* A continuación escribimos el siguiente código HQL desde la pestaña *ConfiguracionHibernate:* `from Empleados` y pulsamos el botón para ejecutar la consulta , véase Figura 3.26. Desde la pestaña *Hibernate Query Result* se puede ver el resultado de la consulta. Si seleccionamos una fila, en el panel **Property** veremos el contenido de la misma.

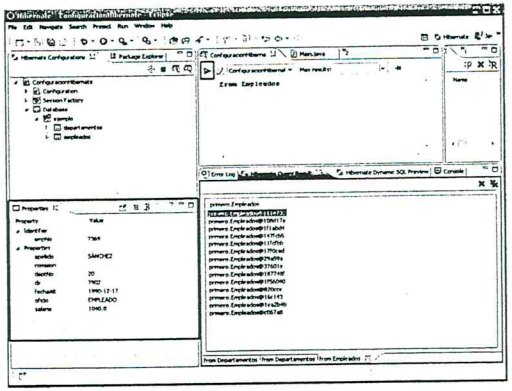


Figura 3.26. Ejecución de una consulta en HQL.

Desde este entorno también se pueden realizar consultas al estilo SQL, por ejemplo:

```
select  dnombre, loc, deptNo from Departamentos
select empNo, apellido, salario from Empleados where dept_no =10
```

Ten en cuenta que, el * no se puede utilizar a la derecha de SELECT, este ejemplo: **`select * from Empleados`** produce un error. También hemos de tener cuidado con las mayúsculas y minúsculas, hemos de respetar los nombres de las clases y las propiedades Java. Por ejemplo, las siguientes SELECT son erróneas:

```
from departamentos
select  Dnombre from Departamentos
from Departamentos as dep where dep.dept_no =10
select  emp_no, apellido, salario from Empleados where emp_no=7839
```

Al final del tema encontrarás un resumen de sentencias en HQL.

Desde la perspectiva Hibernate y pulsando con el botón derecho del ratón en **Configuration** seleccionamos **Mapping Diagram** para visualizar el diagrama de mapeo entre las clases Java y las tablas de la base de datos, Figura 3.27. Se muestra el diagrama similar al mostrado en la Figura 3.28.

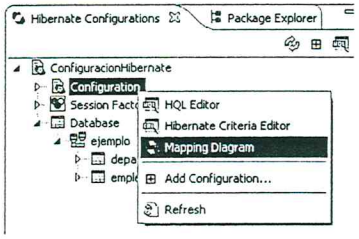


Figura 3.27. Visualizar diagrama de mapeo.

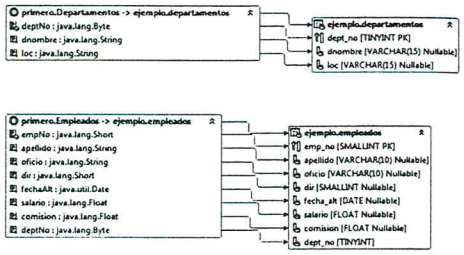


Figura 3.28. Clases generadas y las tablas de la base de datos MySQL.

Actividad 3: Realiza consultas HQL con las tablas Oracle de EMPLEADOS y DEPARTAMENTOS. Visualiza el diagrama de mapeo y fíjate en las diferencias con respecto al diagrama anterior, sobre todo en las columnas que hacen referencia al departamento y que ya se comentaron en la actividad anterior.

Prueba estas consultas:

```
from Empleados
from Empleados as e where e.departamentos.deptNo = 10
select apellido, salario from Empleados as e where e.departamentos.deptNo =10
from Departamentos as d join d.empleadoses
from Departamentos as d left outer join d.empleadoses
```

3.5.6 EMPEZANDO A PROGRAMAR CON HIBERNATE EN ECLIPSE

Con la configuración realizada hasta ahora no se puede programar en Java, es necesario realizar los siguientes pasos:

1. Debemos bajar la última distribución de Hibernate. En este caso se ha bajado la versión: *hibernate-distribution-3.6.8.Final-dist.zip* (en Linux descargamos *hibernate-distribution-3.6.8.Final-dist.tar.gz*) desde la URL: http://sourceforge.net/projects/hibernate/files/hibernate3/.

2. A continuación hemos de preparar la distribución para agregarla a Eclipse. Creamos una carpeta dentro de eclipse con nombre Hibernate y aquí es donde copiaremos las librerías que vamos a necesitar, nos debe quedar: *D:\eclipse\Hibernate*. Descomprimimos el ZIP en esta carpeta (en Linux lo descargamos en *opt/eclipse/Hibernate*).
3. Buscamos el fichero *hibernate3.jar* en la carpeta que se ha generado al descomprimir el fichero y lo copiamos en *D:\eclipse\Hibernate*.
4. Desde la carpeta *D:\eclipse\Hibernate\hibernate-distribution-3.6.8.Final\lib\required* seleccionamos todos los ficheros y los copiamos a nuestra carpeta *D:\eclipse\Hibernate*.
5. Desde la carpeta *D:\eclipse\Hibernate\hibernate-distribution-3.6.8.Final\lib\jpa* seleccionamos todos los ficheros y los copiamos a nuestra carpeta *D:\eclipse\Hibernate (/opt/eclipse/Hibernate* en Linux).
6. Descargamos la última versión de la librería *slf4j* desde la URL http://www.slf4j.org/download.html, se ha descargado el fichero *slf4j-1.6.4.zip* (en Linux *slf4j-1.6.4.tar.gz*), lo descomprimimos y localizamos los ficheros *slf4j-simple-1.6.4.jar* y *slf4j-api-1.6.4.jar* para copiarlos en la carpeta *D:\eclipse\Hibernate*.

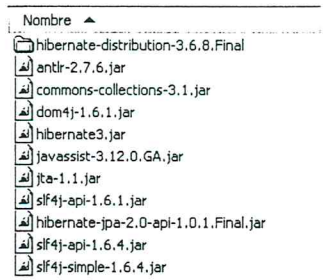


Figura 3.29. Contenido de la carpeta *D:\eclipse\Hibernate*

7. El contenido final de la carpeta se muestra en la Figura 3.29.
8. Desde Eclipse hacemos clic con el botón derecho del ratón en nuestro proyecto y pulsamos en **Build Path -> Add Libreries,** véase Figura 3.30. Se visualiza una nueva ventana desde la que elegimos **User Library** y pulsamos el botón *Next* (Figura 3.31).

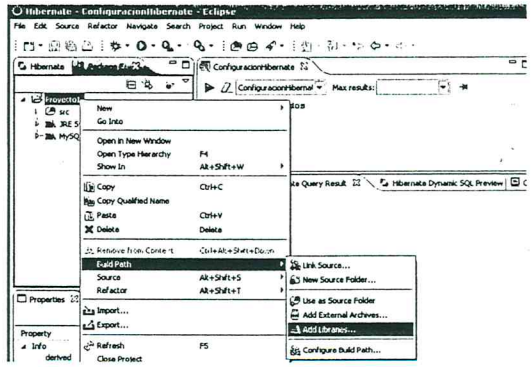


Figura 3.30. Añadir librería al proyecto.

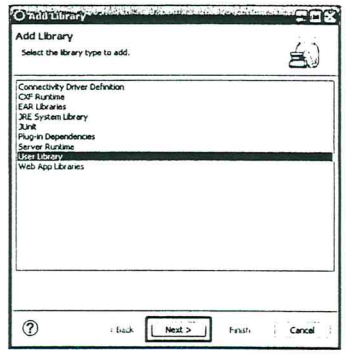


Figura 3.31. Añadir librería al proyecto, selección de *User Library*.

9. Desde la siguiente ventana pulsamos el botón *User Libraries*, a continuación pulsamos el botón *New*, nos pedirá el nombre de la librería que queremos agregar, escribimos por ejemplo: *HibernateLib* y pulsamos el botón *OK*, véase Figura 3.32.

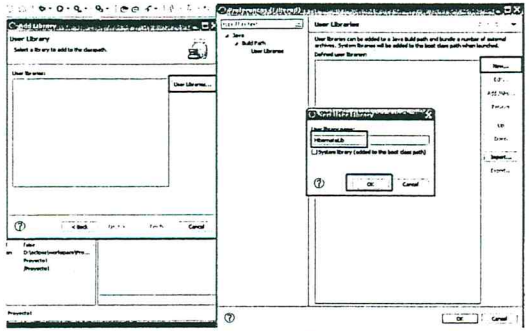
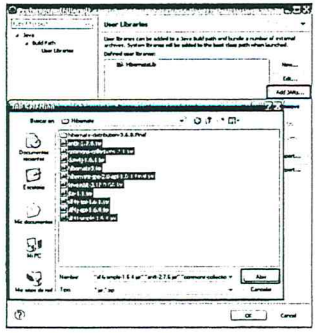
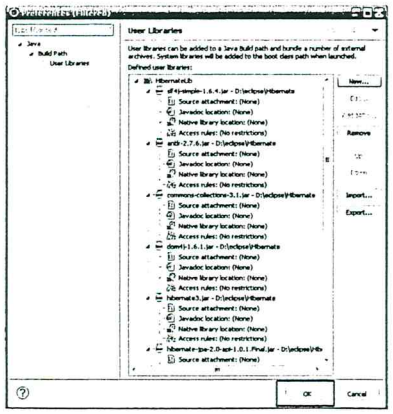


Figura 3.32. Nombrar nuestra librería Hibernate.



10. Seguidamente pulsamos el botón **Add JARs** y seleccionamos todos los JAR de nuestra carpeta *Hibernate* que contiene todas las librerías que vamos a incluir, véase Figura 3.33. Se pulsa el botón *Abrir*. Nos debe quedar una pantalla similar a la mostrada en la Figura 3.34. Pulsamos el botón *OK* y a continuación el botón *Finish* para finalizar. En nuestro proyecto aparecerá la nueva librería

Figura 3.33. Añadir los JAR de la carpeta *Hibernate*.



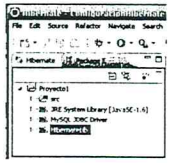


Figura 3.34. Librería *HibernateLib* en nuestro proyecto.

Con esto ya podemos crear el primer programa Java en nuestro proyecto que nos va a permitir comunicarnos con nuestra base de datos.

En primer lugar, crearemos una instancia a la base de datos para poder trabajar con ella y que se utilizará a lo largo de toda la aplicación. Necesitaremos crear un *singleton*.

SABIAS QUE:

El ***singleton*** es un patrón de diseño diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón ***singleton*** se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

El *singleton* es una clase de ayuda que accede a **SessionFactory** para obtener un objeto sesión. La clase llama a **configure()** y carga el fichero de configuración (**hibernate.cfg.xml**) y entonces construye la **SessionFactory** para obtener el objeto de sesión. El nombre de la clase es **SessionFactoryUtil.java** y se incluirá en el paquete **primero** de nuestro proyecto, el código es el siguiente:

```
package primero;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class SessionFactoryUtil {
  private static final SessionFactory sessionFactory;
     static {
          try {
            sessionFactory= new Configuration().configure().buildSessionFactory();
          } catch (Throwable ex) {
               // Log the exception.
               System.err.println("Initial SessionFactory creation failed." + ex);
               throw new ExceptionInInitializerError(ex);
          }
     }
     public static SessionFactory getSessionFactory() {
        return sessionFactory;
     }
}
```

Con esta clase podemos obtener la sesión actual desde cualquier parte de nuestra aplicación. Ahora pulsando con el botón derecho del ratón en nuestro proyecto creamos una clase de nombre **Main.java**. Esta se creará en el *default package* del proyecto (véase Figura 3.35). Desde esta clase ejecutaremos nuestro programa. Lo que hace esta clase es insertar una fila en la tabla DEPARTAMENTOS. El código es el siguiente:

```
import primero.SessionFactoryUtil;
import primero.*;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class Main {
        public static void main(String[] args) {

        SessionFactory sesion = SessionFactoryUtil.getSessionFactory();
        Session session = sesion.openSession();
        Transaction tx =session.beginTransaction();

        System.out.println("Inserto una fila en la tabla DEPARTAMENTOS." );

        Departamentos dep = new Departamentos();
        dep.setDeptNo((byte) 60);
        dep.setDnombre("MARKETING");
        dep.setLoc("GUADALAJARA");

        session.save(dep);
        tx.commit();

        session.close();
    }
}
```

Con la siguiente línea *SessionFactory sesion = SessionFactoryUtil.getSessionFactory();* obtenemos la sesión creada por el singleton. Esta instrucción se usará a lo largo de todas las clases en las que deseemos realizar operaciones con nuestra base de datos. Antes de ejecutar la aplicación debemos editar el fichero **hibernate.cfg.xml** y cambiar la línea:

```
<session-factory="Miconexion">
Por
<session-factory>
```

Cuando se cambia algún fichero de configuración hay que pulsar con el botón derecho en el nombre del proyecto y luego en la opción **Refresh.**

Para ejecutar la aplicación, pulso con el botón derecho del ratón en la clase que tiene el *main()* **(Main.java),** selecciono **Run as-> Java Application,** también puedo ejecutar pulsando el botón *Run* . En la pestaña *Console* se visualiza el resultado, véase Figura 3.35.

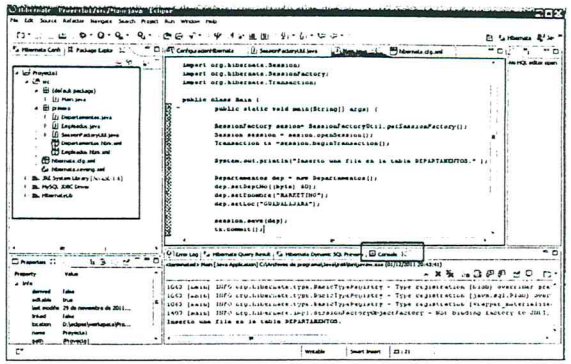


Figura 3.35. Estructura del proyecto y ejecución de Main.java.

Actividad 4: Sigue los pasos anteriores en el proyecto Proyecto1OracleHibernate para insertar filas en la tabla DEPARTAMENTOS.

En el proyecto con las tablas Oracle al visualizar el diagrama de mapeo se visualizan las relaciones entre las tablas EMPLEADOS y DEPARTAMENTOS, Figura 3.36. Cada empleado tiene un objeto departamentos entre sus atributos (atributo *departamentos)*, y cada objeto departamentos tiene entre sus atributos una colección de objetos empleados (atributo *empleadoses*).

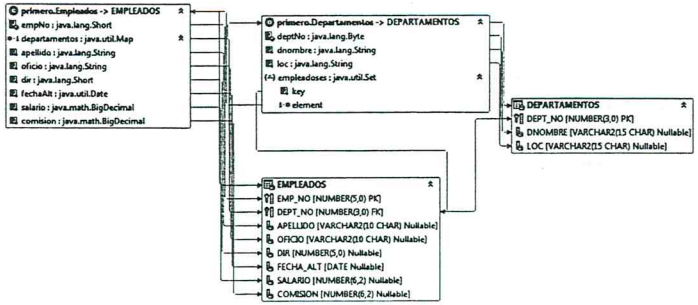


Figura 3.36. Diagrama de mapeo de las tabla Oracle.

Veamos como se inserta un empleado en la tabla EMPLEADOS en el departamento 10:

```
import java.math.BigDecimal;
import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
```

```
public class MainOracle {
  public static void main(String[] args) {
    SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
    Session session = sesion.openSession();
    Transaction tx =session.beginTransaction();

    System.out.println("Inserto un EMPLEADO EN EL DEPARTAMENTO 10." );

    BigDecimal salario= new BigDecimal(1500);//inicializo el salario
    BigDecimal comision=new BigDecimal(10); //inicializo la comisión

    Empleados em = new Empleados(); //creo un objeto empleados
    em.setEmpNo((short) 4455);  //el numero de empleado es 4455
    em.setApellido("PEPE");
    em.setDir((short) 7499);    //el director es el numero de empleado 7499
    em.setOficio("VENDEDOR");
    em.setSalario(salario);
    em.setComision(comision);

    Departamentos d= new Departamentos(); //creo un objeto Departamentos
    d.setDeptNo((byte) 10);  //el número de dep es 10
    em.setDepartamentos(d);

    //fecha de alta
    java.util.Date hoy = new java.util.Date();
    java.sql.Date fecha = new java.sql.Date(hoy.getTime());
    em.setFechaAlt(fecha);

    session.save(em);
    tx.commit();
    session.close();
  }
}
```

Si el departamento no existe o el empleado ya existe se visualizará un error.

> RECUERDA:
>
> Cuando creemos un nuevo proyecto Java **(File-> New-> Java Project)** hemos de seguir los pasos realizados:
> - Añadir la librería *HibernateLib* y el driver MySQL **(Build Paths-> Add Libreries)**.
> - Generar los ficheros de configuración de Hibernate.
> - Generar las clases de la base de datos.
> - Crear el singleton.
> - Crear la clase main.
> - Crear el resto de clases.
> - Cambiar la línea <session-factory> del fichero **hibernate.cfg.xml .**

3.6 ESTRUCTURA DE LOS FICHEROS DE MAPEO

Hibernate utiliza unos ficheros de mapeo para relacionar las tablas de la base de datos con los objetos Java. Estos ficheros están en formato XML y tienen la extensión **.hbm.xml.** En el proyecto anterior se han creado los ficheros: **Empleados.hbm.xml** y **Departamentos.hbm.xml,** el primero asociado a la tabla de EMPLEADOS y el segundo a la tabla DEPARTAMENTOS. Estos ficheros se guardan en el mismo directorio que las clases Java **Empleados.java** y **Departamentos.java.** Estas clases representan un objeto EMPLEADOS y un objeto DEPARTAMENTOS respectivamente y todos los ficheros forman parte del paquete **primero.** La Figura 3.37 muestra la correspondencia entre los ficheros de mapeo y las clases generadas.

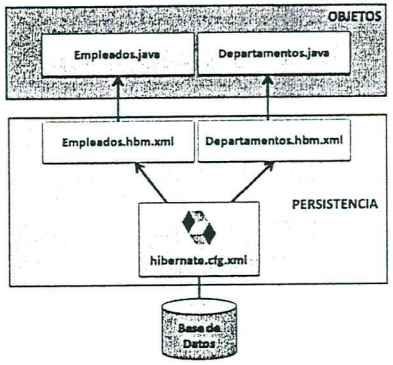


Figura 3.37. Ficheros de mapeo y clases Java.

La estructura del fichero **Departamentos.hbm.xml** es la siguiente:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 03-dic-2011 15:29:42 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
    <class name="primero.Departamentos" table="departamentos" catalog="ejemplo">
        <id name="deptNo" type="byte">
            <column name="dept_no" />
            <generator class="assigned" />
        </id>
        <property name="dnombre" type="string">
            <column name="dnombre" length="15" />
        </property>
        <property name="loc" type="string">
            <column name="loc" length="15" />
        </property>
    </class>
</hibernate-mapping>
```

Antes de nada, recordemos la descripción de la tabla DEPARTAMENTOS:

```
TABLE departamentos (
 dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
 dnombre  VARCHAR(15),
 loc      VARCHAR(15)
);
```

Veamos el significado del contenido del fichero XML:

- **hibernate-mapping:** Todos los ficheros de mapeo comienzan y acaban con esta etiqueta.
- **class:** Esta etiqueta engloba a la clase con sus atributos, indicando siempre el mapeo a la tabla de la base de datos. En *name* se indica el nombre de la clase y en *table* el nombre de la tabla a la que representa este objeto, en *catalog* se indica el nombre de la base de datos:

```
<class name="primero.Departamentos" table="departamentos" catalog="ejemplo">
```

  - Dentro de *class* distinguimos la etiqueta **id** en la cual se indica en *name* el campo que representa al atributo clave en la clase y en *column* su nombre sobre la tabla, en *type* el tipo de datos. En *id* además tenemos la propiedad *generator* que indica la naturaleza del campo clave. En este caso es *assigned* porque es el usuario el que se encarga de asignar la clave. Si fuese *increment* indicaría que es un identificador autogenerado por la base de datos. Este atributo se correspondería con la columna DEPT_NO *(dept_no TINYINT(2) NOT NULL PRIMARY KEY)* de la tabla DEPARTAMENTOS:

```
<id name="deptNo" type="byte">
            <column name="dept_no" />
            <generator class="assigned" />
</id>
```

  - El resto de atributos se indican en las etiquetas **property** asociando el nombre del campo de la clase con el nombre de la columna de la tabla y el tipo de datos. La columna DNOMBRE de la tabla DEPARTAMENTOS *(dnombre VARCHAR(15))* se definiría así:

```
<property name="dnombre" type="string">
      <column name="dnombre" length="15" />
</property>
```

La columna LOC de la tabla se declara de forma similar:

```
<property name="loc" type="string">
       <column name="loc" length="15" />
</property>
```

Los tipos que declaramos y utilizamos en los ficheros de mapeo no son tipos de datos Java. Tampoco son tipos de base de datos SQL. Estos tipos se llaman **tipos de mapeo Hibernate,** convertidores que pueden traducir de tipos de datos Java a SQL y viceversa. De nuevo, Hibernate tratará de determinar el tipo correcto de conversión y de mapeo por sí mismo.

La estructura del fichero **Empleados.hbm.xml** es la siguiente:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 03-dic-2011 15:29:42 by Hibernate Tools 3.4.0.CR1 -->
<hibernate-mapping>
    <class name="primero.Empleados" table="empleados" catalog="ejemplo">
        <id name="empNo" type="short">
            <column name="emp_no" />
            <generator class="assigned" />
        </id>
        <property name="apellido" type="string">
            <column name="apellido" length="10" />
        </property>
        <property name="oficio" type="string">
            <column name="oficio" length="10" />
        </property>
        <property name="dir" type="java.lang.Short">
            <column name="dir" />
        </property>
        <property name="fechaAlt" type="date">
            <column name="fecha_alt" length="10" />
        </property>
        <property name="salario" type="java.lang.Float">
            <column name="salario" precision="6" />
        </property>
        <property name="comision" type="java.lang.Float">
            <column name="comision" precision="6" />
        </property>
        <property name="deptNo" type="byte">
            <column name="dept_no" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Nos fijamos ahora en los ficheros de mapeo generados para las tablas Oracle. Hay una serie de líneas que no aparecen en el mapeo en MySQL, estas líneas mapean las relaciones entre las tablas EMPLEADOS y DEPARTAMENTOS. En **Empleados.hbm.xml** nos encontramos con la relación **many-to-one,** es una asociación unidireccional *muchos-a-uno* (una clave ajena en una tabla referencia la columna (o columnas) de la clave primaria de la tabla destino), es decir muchos empleados pertenecen a un departamento). El mapeo indica que la clase *Empleados.java* tiene un atributo de nombre **departamentos** que es una instancia de la clase **primero.Departamentos**:

```
<many-to-one name="departamentos" class="primero.Departamentos" fetch="select">
        <column name="DEPT_NO" precision="2" scale="0" not-null="true" />
</many-to-one>
```

El atributo **fetch** (por defecto es *select*) escoge entre la recuperación de unión exterior (*outer-join*) o la recuperación por selección secuencial.

En **Departamentos.hbm.xml** en el mapeo se define una lista de empleados para un departamento. La relación es **one-to-many,** es una asociación *uno-a-muchos,* es decir, un departamento puede tener muchos empleados. El mapeo indica que la clase *Departamentos.java* tiene un atributo de nombre **empleadoses** que es una lista de instancias de la clase **primero.Empleados**:

```
<set name="empleadoses" table="EMPLEADOS" inverse="true"
                                        lazy="true" fetch="select">
      <key>
            <column name="DEPT_NO" precision="2" scale="0" not-null="true" />
      </key>
      <one-to-many class="primero.Empleados" />
</set>
```

3.7 CLASES PERSISTENTES

Hemos visto que entre las etiquetas *<hibernate-mapping> </hibernate-mapping>* de los ficheros XML se incluye un elemento *class* que hace referencia a una clase:

```
<class name="primero.Departamentos" table="departamentos" catalog="ejemplo">

<class name="primero.Empleados" table="empleados" catalog="ejemplo">
```

En nuestro proyecto se han generado las clases **Empleados.java** y **Departamentos.java.** A estas clases se les llama **clases persistentes,** equivalen a una tabla de la base de datos, y un registro o fila es un objeto persistente de esa clase. Estas clases representan un objeto empleados y un objeto departamentos, por lo tanto podemos crear objetos empleados y departamentos a partir de ellas. Tienen unos atributos y unos métodos *get* y *set* para acceder a los mismos.

Utilizan convenciones de nombrado estándares de JavaBean para los métodos de propiedades *getter* y *setter* así como también visibilidad privada para los campos. Al ser los atributos de los objetos privados se crean métodos públicos para retornar un valor de un atributo, **método getter,** o para cargar un valor a un atributo, **método setter,** por ejemplo el método *getDnombre()* devuelve el nombre de un departamento (atributo *dnombre*) y el método *setDnombre()* carga un valor en el atributo dnombre. A estas reglas también se las llama modelo de programación **POJO** – *Plain Old Java Object.*

Para completar el nombre de un método *getter* o *setter,* solo hay que poner la primera letra que los une en mayúsculas. Si nos fijamos en el fichero **Departamentos.hbm.xml,** el elemento **id** es la declaración de la propiedad identificadora, el atributo de mapeo **name="deptNo"** declara el nombre de la propiedad JavaBean y le dice a Hibernate que utilice los métodos **getDeptNo()** y **setDeptNo()** para acceder a la propiedad:

```
<id name="deptNo" type="byte">
```

Al igual que con el elemento **id,** el atributo **name** del elemento **property** le dice a Hibernate que métodos *getter* y *setter* utilizar. Así que en este caso, Hibernate buscará los métodos *getDnombre(), setDnombre(), getLoc()* y *setLoc().*


```
<property name="dnombre" type="string">
<property name="loc" type="string">
```

La clase **Departamentos.java** es la siguiente:

```
/**
 * Departamentos generated by hbm2java
 */
public class Departamentos implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    private byte deptNo;
    private String dnombre;
    private String loc;

    public Departamentos() {
    }

    public Departamentos(byte deptNo) {
            this.deptNo = deptNo;
    }

    public Departamentos(byte deptNo, String dnombre, String loc) {
            this.deptNo = deptNo;
            this.dnombre = dnombre;
            this.loc = loc;
    }

    public byte getDeptNo() {
            return this.deptNo;
    }

    public void setDeptNo(byte deptNo) {
            this.deptNo = deptNo;
    }

    public String getDnombre() {
            return this.dnombre;
    }

    public void setDnombre(String dnombre) {
            this.dnombre = dnombre;
    }

    public String getLoc() {
            return this.loc;
    }

    public void setLoc(String loc) {
            this.loc = loc;
    }
}
```

Convenciones de nombrado estándares de JavaBean para los métodos de propiedades *getter* y *setter*:

- Si la propiedad no es de tipo booleana, el nombre del método que lee el valor debe estar precedido por **get.** Por ejemplo: *getDnombre()* es un nombre válido.
- Si la propiedad es un booleano, el prefijo es **get** o **is.** Por ejemplo, *getCasado()* o *isCasado()* son nombres válidos para una propiedad booleana.
- El método para almacenar el valor debe tener el prefijo **set.** Por ejemplo, *setDnombre()* es un nombre válido para la propiedad *dnombre.*
- Para completar el nombre de un método *getter* o *setter*, solo hay que poner **la primera letra que los une en mayúsculas.**
- Los métodos marcados como *setter* deben ser declarados como públicos, devolver un tipo *void* y recibir un argumento del tipo de propiedad al que van a dar valor.
- Los métodos de tipo *getter* deben ser declarados como públicos, no aceptan argumentos y devuelven un valor del mismo tipo que el que recibe el método *setter.*

3.8 SESIONES Y OBJETOS HIBERNATE

Pada poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto **Session** utilizando la clase **SessionFactory** de Hibernate. El siguiente fragmento de código ilustra este proceso:

```
// Inicializa el entorno Hibernate
Configuration cfg = new Configuration().configure();

// Crea el ejemplar de sesion factory
SessionFactory sessionFactory = cfg.buildSessionFactory();

// Obtiene un objeto session
Session session = sessionFactory.openSession();
```

La llamada a *Configuration().configure()* carga el fichero de configuración **hibernate.cfg.xml** e inicializa el entorno de Hibernate. Una vez inicializada la configuración se crea el ejemplar de **SessionFactory,** que normalmente solo se crea una vez y se utiliza para crear todas las sesiones relacionadas con un contexto dado. Es lo que se hizo en el *Proyecto1* al crear la clase **SessionFactoryUtil.java,** para crear una vez el ejemplar de **SessionFactory**:

```
sessionFactory= new Configuration().configure().buildSessionFactory();
```


3.8.1 TRANSACCIONES

Un objeto **Session** de Hibernate representa una única unidad de trabajo para un almacén de datos dado y lo abre un ejemplar de **SessionFactory.** Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción. El siguiente código ilustra una sesión de persistencia de Hibernate:

```
Session session = sessionFactory.openSession();//comienza la sesión
Transaction tx = session.beginTransaction();   //comienza la transacción
//Código de persistencia
tx.commit();     //valida la transacción
session.close();//finaliza la sesión
```

El método *beginTransaction()* marca el comienzo de una transacción. El método *commit()* valida una transacción, y *rollback()* deshace la transacción.

3.8.2 ESTADOS DE UN OBJETO HIBERNATE

Hibernate define y soporta los siguientes estados de objeto:

- ***Transitorio*** - un objeto es transitorio, si ha sido recién instanciado utilizando el operador *new* y no está asociado a una **Session** de Hibernate. No tiene una representación persistente en la base de datos y no se le ha asignado un valor identificador. Las instancias transitorias serán destruidas por el recolector de basura si la aplicación no mantiene más una referencia. Utiliza la **Session** de Hibernate para hacer un objeto persistente (y deja que Hibernate se ocupe de las declaraciones SQL que necesitan ejecutarse para esta transición). Las instancias recién instanciadas de una clase persistente, Hibernate las considera como *transitorias.* Podemos hacer una instancia transitoria *persistente* asociándola con una sesión:

```
//Inserto el departamento 60 en la tabla DEPARTAMENTOS
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 60);
dep.setDnombre("MARKETING");
dep.setLoc("GUADALAJARA");
session.save(dep); //save() hace que la instancia sea persistente
```

- ***Persistente*** - una instancia persistente tiene una representación en la base de datos y un valor identificador. Puede haber sido guardado o cargado, sin embargo, por definición, se encuentra en el ámbito de una **Session.** Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo. Los desarrolladores no ejecutan declaraciones UPDATE manuales, o declaraciones DELETE cuando un objeto se deba poner como transitorio.

Resumiendo, los objetos transitorios solo existen en memoria y no en un almacén de datos, han sido instanciados por el desarrollador sin haberlos almacenado mediante una sesión. Los persis-

tentes se caracterizan por haber sido ya creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.

- *Separado* - una instancia separada es un objeto que se ha hecho persistente, pero su **Session** ha sido cerrada. La referencia al objeto todavía es válida, por supuesto, y la instancia separada podría incluso ser modificada en este estado. Una instancia separada puede ser reunida a una nueva **Session** más tarde, haciéndola persistente de nuevo (con todas las modificaciones). Este aspecto habilita un modelo de programación para unidades de trabajo de ejecución larga que requieren tiempo-para-pensar por parte del usuario. Las llamamos *transaccciones de aplicación*, por ejemplo, una unidad de trabajo desde el punto de vista del usuario.

Fuente: http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/objectstate.html

3.8.3 CARGA DE OBJETOS

Los métodos **load()** de **Session** proporcionan una forma de recuperar **una instancia persistente** conociendo el identificador del registro. **load()** toma un objeto clase y carga el estado dentro de una instancia recién instanciada de esa clase, en un estado persistente. El siguiente ejemplo visualiza los datos del departamento 20 (hay que señalar que el tipo para el atributo identificador de la clase *(deptNo)* es *byte*):

```
//Visualizo los datos del departamento 20
Departamentos dep = new Departamentos();
dep = (Departamentos) session.load(Departamentos.class, (byte)20);
System.out.println("Nombre Dep:"+dep.getDnombre());
System.out.println("Localidad:"+dep.getLoc());
```

El método **load()** lanza una excepción irrecuperable si la fila no existe. Si no tenemos la certeza de que la fila exista debemos utilizar el método **get()**, que llama a la base de datos inmediatamente y devuelve nulo si no existe una fila correspondiente, el siguiente ejemplo comprueba si el departamento 11 existe, si existe visualiza sus datos, si no existe visualiza un mensaje:

```
dep = (Departamentos) session.get(Departamentos.class, (byte)11);
if (dep==null) {
        System.out.println("El departamento no existe");
}
else
{
        System.out.println("Nombre Dep:"+dep.getDnombre());
        System.out.println("Localidad:"+dep.getLoc());
}
```

Actividad 5: Visualiza los datos del empleado con número 7369.

El siguiente ejemplo obtiene los datos del departamento 10 y el APELLIDO y SALARIO de sus empleados (en Oracle), para obtener los empleados usamos el método *getEmpleadoses()* de la clase

Departamentos:

```
import java.util.Iterator;
import java.util.Set;
import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

 public class ListadoDep {
    public static void main(String[] args) {
    SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
    Session session = sesion.openSession();

    System.out.println("============================" );
    System.out.println("DATOS DEL DEPARTAMENTO 10." );

    Departamentos dep = new Departamentos();
    dep = (Departamentos) session.load(Departamentos.class, (byte)10);
    System.out.println("Nombre Dep:"+dep.getDnombre());
    System.out.println("Localidad:"+dep.getLoc());

    System.out.println("============================" );
    System.out.println("EMPLEADOS DEL DEPARTAMENTO 10." );

    Set<Empleados> listaemple = dep.getEmpleadoses();//obtenemos empleados
    Iterator <Empleados> it = listaemple.iterator();
    System.out.println("Número de empleados: "+listaemple.size() );
    while(it.hasNext()) {
            Empleados emple = new Empleados();
            emple=it.next();
            System.out.println(emple.getApellido() + " * " +emple.getSalario());
     }
    System.out.println("============================" );
    session.close();
   }
 }
```

La ejecución muestra la siguiente salida:

```
============================
DATOS DEL DEPARTAMENTO 10.
Nombre Dep:CONTABILIDAD
Localidad:SEVILLA
============================
EMPLEADOS DEL DEPARTAMENTO 10.
Número de empleados: 4
REY * 4100
PEPE * 1500
MUÑOZ * 1690
CEREZO * 2885
============================
```

Consulta la API de Hibernate: http://docs.jboss.org/hibernate/core/3.5/javadoc/

3.8.4 ALMACENAMIENTO, MODIFICACIÓN Y BORRADO DE OBJETOS

Para almacenar un objeto en la base de datos utilizamos el método **save()**. El siguiente ejemplo crea un nuevo objeto Departamentos y se lo entrega a Hibernate, en ese momento, Hibernate se encarga de SQL y ejecuta un INSERT en la base de datos:

```
Departamentos dep = new Departamentos();
dep.setDeptNo((byte) 70);
dep.setDnombre("INFORMÁTICA");
dep.setLoc("TOLEDO");
session.save(dep); //almacena el objeto
```

Para realizar el borrado de un objeto, primero debe ser cargado con el método **load()**, a continuación podemos borrarlo con el método **delete()**, hemos de asegurarnos de que no existan referencias de otros objetos. El siguiente ejemplo borra el empleado cuyo número de empleado *(empNo)* es 7369:

```
Empleados em = new Empleados();
em = (Empleados) session.load(Empleados.class, (short)7369);
session.delete(em); //elimina el objeto
```

Para modificar un objeto, igual que para borrarlo, primero hemos de cargarlo, a continuación realizamos las modificaciones con los métodos *setter* y por último utilizamos el método **update()**. El siguiente ejemplo modifica el SALARIO y la COMISION del empleado con número 7499:

```
Empleados em = new Empleados();
em = (Empleados) session.load(Empleados.class, (short)7499);
System.out.println("Salario antiguo: "+em.getSalario());
System.out.println("Comisión antigua: "+em.getComision());
em.setSalario((float) 3000);
em.setComision((float) 100);
session.update(em); //modifica el objeto
System.out.println("Salario nuevo: "+em.getSalario());
System.out.println("Comisión nueva: "+em.getComision());
```

Actividad 6: Inserta y modifica objetos empleado en la base de datos MySQL. Después desde HQL comprueba las modificaciones.

3.9 CONSULTAS

Hibernate soporta un lenguaje de consulta orientado a objetos denominado **HQL** *(Hibernate Query Language)* fácil de usar pero potente a la vez. Este lenguaje es una extensión orientada a objetos de SQL. Las consultas HQL y SQL nativas son representadas con una instancia de *org.hibernate.Query*. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto resultado y para la ejecución de la consulta real. Siempre obtiene una **Query** utilizando la **Session** actual. Para realizar una consulta usaremos la función **createQuery("consulta")** de la siguiente manera:


```
Query q = session.createQuery("from Departamentos");
```

Para recuperar los datos de la consulta usaremos el método *list()*:

```
List <Departamentos> lista = q.list();
```

o el método *iterate()*:

```
Iterator iter = q.iterate();
```

El método *list()* devuelve en una colección todos los resultados de la consulta, en la colección se encuentran instanciadas todas las entidades que corresponden al resultado de la ejecución de la consulta. Este método realiza una única comunicación con la base de datos en donde se traen todos los resultados y requiere que haya memoria suficiente para almacenar todos los objetos resultantes de la consulta. Si la cantidad de resultados es extensa, el retraso del acceso a la base de datos será notorio.

El método *iterate()* devuelve un iterador Java para iterar los resultados de la consulta. En este caso Hibernate ejecuta la consulta obteniendo solo los ids de las entidades y en cada llamada el método *Iterator.next()* ejecuta la consulta propia para obtener la entidad completa. Esto implica una mayor cantidad de accesos a la base de datos y por tanto, mayor tiempo de procesamiento total. La ventaja de este método es que no requiere que todas las entidades estén cargadas en memoria simultáneamente. Se puede utilizar el método *setFetchSize()* para fijar la cantidad de resultados a recuperar en cada acceso a la base de datos, de esta manera no se hará un acceso a la base de datos en cada llamada al método *Iterator.next()*:

```
q.setFetchSize(10);
```

El siguiente ejemplo realiza una consulta de todas las filas de la tabla DEPARTAMENTOS:

```
Departamentos depar = new Departamentos();
Query q = session.createQuery("from Departamentos");
List <Departamentos> lista = q.list();
// Obtenemos un Iterador y recorremos la lista.
Iterator <Departamentos> iter = lista.iterator();
while (iter.hasNext())
{
        //extraer el objeto
        depar = (Departamentos) iter.next();
        System.out.println(depar.getDeptNo() +"*"+depar.getDnombre());
}
```

Visualiza la siguiente información:

```
10*CONTABILIDAD
20*INVESTIGACIÓN
30*VENTAS
40*PRODUCCIÓN
60*MARKETING
```

El ejemplo con el método *iterate()* quedaría así:

```
Query q = session.createQuery("from Departamentos");
q.setFetchSize(10);
Iterator iter = q.iterate();
Departamentos depar = new Departamentos();
while (iter.hasNext() ) {
        //extraer el objeto
        depar = (Departamentos) iter.next();
        System.out.println( depar.getDeptNo() +"*"+depar.getDnombre());
}
```

El método *uniqueResult()* ofrece un atajo si sabemos que la consulta devolverá un objeto. Los siguientes ejemplos obtienen los datos de un único departamento, el primero visualiza los datos del departamento 10 y el segundo los datos del departamento con nombre CONTABILIDAD:

```
//Visualiza los datos del departamento 10

Departamentos depart = (Departamentos) session.createQuery(
                "from Departamentos as dep " +
                "where dep.deptNo = ?").setInteger(0, 10).uniqueResult();
System.out.println(depart.getLoc() +"*"+depart.getDnombre());

//Visualiza los datos del departamento con nombre CONTABILIDAD

depart = (Departamentos) session.createQuery(
            "from Departamentos as dep " +
            "where dep.dnombre = ?").setString(0, "CONTABILIDAD").uniqueResult();
System.out.println(depart.getLoc() +"*"+depart.getDeptNo());
```

El siguiente ejemplo en Oracle realiza una consulta para visualizar el apellido y salario de los empleados del departamento 20, la consulta es la siguiente: *from Empleados as e where e.departamentos.deptNo =20:*

```
Query q = session.createQuery("from Empleados as e where
                                e.departamentos.deptNo = 20");
List <Empleados> lista = q.list();
// Obtenemos un Iterador y recorremos la lista
Iterator <Empleados> iter = lista.iterator();
Empleados emp2 = new Empleados();
while (iter.hasNext())
{
        emp = (Empleados) iter.next();//extraer el objeto
        System.out.println(emp.getApellido() +"*"+emp.getSalario());
}
```

Actividad 7: Realiza una consulta con *createQuery()* en Oracle para obtener los datos del departamento 20 y visualiza también el apellido de sus empleados.

3.9.1 PARÁMETROS EN LAS CONSULTAS

Con **createQuery()** se pueden enlazar valores a los parámetros con nombre, que son identificadores de la forma *:nombre* en la cadena de consulta, o a los parámetros ? de estilo JDBC. Hibernate numera los parámetros desde cero, el primero que aparece estará en la posición 0, el siguiente en la posición 1, y así sucesivamente. Las ventajas de los parámetros con nombre son las siguientes:

- son insensibles al orden en que aparecen en la cadena de consulta,
- pueden aparecer múltiples veces en la misma petición,
- son auto-documentados.

Algunos métodos que se pueden usar para asignar valores a los parámetros son:

MÉTODO	FUNCIÓN
setCharacter(int posicion, char valor) setCharacter(String nombre, char valor)	Asigna valor a un parámetro de tipo CHAR.
setDate(int posicion, Date fecha) setDate(String nombre, Date fecha)	Asigna valor a un parámetro de tipo DATE.
setDouble(int posicion, double valor) setDouble(String nombre, double valor)	Asigna valor a un parámetro de tipo decimal (en MySQL tipo FLOAT).
setInteger(int posicion, int valor) setInteger(String nombre, int valor)	Asigna valor a un parámetro de tipo entero.
setString(int posicion, String valor) setString(String nombre, String valor)	Asigna valor a un parámetro de tipo VARCHAR.
setParameterList(String nombre, Collection valores)	Asigna una lista de valores a un parámetro.

posicion => indica la posición del parámetro dentro de la consulta, empieza en 0.
nombre => indica el nombre (:nombre) del parámetro dentro de la consulta.
Consultar la API de Hibernate http://docs.jboss.org/hibernate/core/3.5/javadoc/

Veamos algunos ejemplos:

```
//Utilizando parámetros nombrados
//Empleados cuyo número de departamento sea 10 y el oficio DIRECTOR

Query q = session.createQuery("from Empleados emp where emp.deptNo = :ndep
            and emp.oficio = :ofi");
q.setInteger("ndep",10);
q.setString("ofi", "DIRECTOR");

//Utilizando parámetros por posición
//Empleados cuyo número de departamento sea 10 y el oficio DIRECTOR

Query q = session.createQuery("from Empleados emp where emp.deptNo = ?
            and emp.oficio = ?");
q.setInteger(0,10);
q.setString(1, "DIRECTOR");

//Lista de parámetros nombrados
//Empleados cuyo número de departamento sea 10 o 20
```

```
List  <Byte> numeros = new ArrayList <Byte> ();
numeros.add((byte)10);
numeros.add((byte)20);
Query q = session.createQuery("from Empleados emp where
                                emp.deptNo in :listadep)");
q.setParameterList("listadep", numeros);

//Parámetro de tipo Date
//Empleados cuya fecha de alta es 1991-12-03

SimpleDateFormat formatoDelTexto = new SimpleDateFormat("yyyy-MM-dd");
String strFecha = "1991-12-03";
Date fecha = null;
try {fecha = formatoDelTexto.parse(strFecha);}
catch (ParseException ex) {ex.printStackTrace();}
Query q = session.createQuery(" from Empleados e where e.fechaAlt = ?");
q.setDate(0, fecha);
```

3.9.2 CONSULTAS SOBRE CLASES NO ASOCIADAS

Si queremos recuperar los datos de una consulta en la que intervienen varias tablas y no tenemos asociada a ninguna clase los atributos que devuelve esa consulta, podemos utilizar la clase **Object.** Los resultados se reciben en un array de objetos, donde el primer elemento del array se corresponde con la primera clase que ponemos a la derecha de FROM, el siguiente elemento con la siguiente clase y así sucesivamente. El siguiente ejemplo realiza una consulta para obtener los datos de los empleados y de sus departamentos. El resultado de la consulta se recibe en un array de objetos, donde el primer elemento del array pertenece a la clase Empleados y el segundo a la clase Departamentos:

```
Query cons =  session.createQuery("from Empleados e, Departamentos d where
      e.deptNo=d.deptNo order by e.apellido");
Iterator q = cons.iterate();
while (q.hasNext()) {
  Object[] par =(Object[]) q.next();
  Empleados em = (Empleados) par[0];
  Departamentos de = (Departamentos) par[1];
  System.out.println( em.getApellido() +"*"+em.getSalario() +
            "*"+ de.getDnombre()+"*"+de.getLoc());
}
```

Para estos casos se puede crear una vista a partir de las tablas y realizar el mapeo de la vista con Hibernate.

3.9.3 FUNCIONES DE GRUPO EN LAS CONSULTAS

Los resultados devueltos por una consulta HQL o SQL en la que se ha utilizado una función de grupo como por ejemplo *avg(), sum(), count,* etc. se pueden recoger como un único valor utilizando el método *uniqueResult().* El siguiente ejemplo muestra el salario medio de los empleados:


```
//salario medio de los empleados
cons =  session.createQuery("select avg(em.salario) from Empleados as em");
Double  suma = (Double) cons.uniqueResult();
System.out.println( "Salario medio: "+suma);
```

Si en la consulta intervienen varias funciones de grupo y además devuelve varias filas podemos utilizar objetos devueltos por las consultas.

3.9.4 OBJETOS DEVUELTOS POR LAS CONSULTAS

Anteriormente, vimos cómo se pueden tratar los resultados obtenidos por una SELECT que no está asociada a ninguna entidad. Supongamos que a partir de las tablas EMPLEADOS y DEPARTAMENTOS queremos obtener una consulta en la que aparezcan el nombre del departamento, su número, el número de empleados y el salario medio. Como los datos de esta consulta no están asociados a ninguna clase, puedo crear una y utilizarla sin necesidad de mapearla. Cada fila devolverá un objeto de esa clase. Por ejemplo, creo la clase **Totales** en el paquete **primero** con 4 atributos: *numero, cuenta, media* y *nombre* (para guardar los datos de el número de departamento, número de empleados, la media de salario y el nombre del departamento) y los *getter* y *setter* asociados. La clase **Totales.java** es la siguiente:

```
package primero;
public class Totales {
private Object cuenta;
private Object numero;
private Double media;
private String nombre;

public Totales(final Object numero, final Object cuenta,
               final Double media, final String nombre ) {
    this.cuenta = cuenta;
    this.media = media;
    this.nombre = nombre;
    this.numero = numero;
}

public Totales(){}

public Object getCuenta() {return this.cuenta;}
public void setCuenta(final Object cuenta) {this.cuenta = cuenta;}

public Object getNumero() {return this.numero;}
public void setNumero(final Object numero) {this.cuenta = numero;}

public Double getMedia() {return this.media;}
public void setMedia(final Double media) {this.media = media;}

public String getNombre() {return this.nombre;}
public void setNombre(final String nombre) {this.nombre = nombre;}
}
```

Para hacer uso de la clase anterior procedemos de la siguiente manera:

```
Query cons = session.createQuery("select new primero.Totales(" +
            "de.deptNo, count(em.empNo), " +
            "coalesce(avg(em.salario),0), de.dnombre" +
            ")" +
            " from Departamentos as de, Empleados as em" +
            " where de.deptNo=em.deptNo" +
            " group by de.deptNo,de.dnombre" );
Iterator q = cons.iterate();
while (q.hasNext()) {
   Totales tot =(Totales) q.next();
   System.out.println( "Numero Dep:"+tot.getNumero()+" Nombre:"+tot.getNombre()
      +" Salario medio: "+tot.getMedia() + " N° emple:" +tot.getCuenta());
}
```

En el ejemplo de Oracle la consulta tendría el siguiente aspecto:

```
Query cons = session.createQuery("select new primero.Totales(" +
            "em.departamentos.deptNo, count(em.empNo)," +
            "coalesce(avg(em.salario),0), em.departamentos.dnombre" +
            ")" +
            " from Empleados as em " +
            " group by em.departamentos.deptNo,em.departamentos.dnombre" );
```

También podemos recuperar los valores de una consulta que no está asociada a ninguna clase mediante un array de objetos, clase **Object** (un ejemplo similar se vio anteriormente). Los resultados se reciben en un array de objetos, donde el primer elemento del array se corresponde con la primera fila, el siguiente con la siguiente fila. Dentro de cada fila será necesario acceder a los atributos o columnas mediante otro array de objetos:

```
cons =  session.createQuery("select de.deptNo as dep,count(em.empNo) as cuenta,"+
                  "coalesce(avg(em.salario),0) as media, de.dnombre as nombre " +
                  " from Departamentos as de,Empleados as em" +
                  " where de.deptNo=em.deptNo" +
                  " group by de.deptNo,de.dnombre" );

List<Object[]> filas = cons.list(); //Todas las filas
for (int i = 0; i < filas.size(); i++)
{
       Object[] filaActual = filas.get(i); //Acceso a una fila
       System.out.println("Dep:"+filaActual[0] + "* Nombre:" + filaActual[3] + "*"+
                          "Media:"+filaActual[2] + "*Empleados:" + filaActual[1]);
}
```

Otra forma de recuperar valores de una consulta que no está asociada a ninguna clase es recuperando una lista de objetos del tipo **"java.util.Map"**, en donde las claves son las columnas que queremos consultar y los valores son los valores recuperados de la base de datos. Por ejemplo, para recuperar los valores de la consulta anterior haríamos lo siguiente:

```
cons =  session.createQuery("select new map de.deptNo as dep, count(em.empNo) as
     cuenta, coalesce(avg(em.salario),0) as media, de.dnombre as nombre) " +
```



```
            " from Departamentos as de,Empleados as em" +
                  " where de.deptNo=em.deptNo" +
                  " group by de.deptNo,de.dnombre" );

List <Map>  lista = cons.list();//Todas las filas
for (int i = 0; i < lista.size(); i++)
{
      Map mapa = lista.get(i);
      System.out.println("Datos de la fila:" + i); //Datos de la fila
      Set columnasTodas = mapa.keySet();//devuelve las claves del conjunto
      for(Iterator<String> it = columnasTodas.iterator(); it.hasNext();)
       {
         String columna = it.next();
         System.out.println("\tColumna: " + columna + ", valor: " +
                            mapa.get(columna));
       }
}
```

3.10 INSERT, UPDATE Y DELETE

Con el lenguaje HQL también podremos realizar operaciones INSERT, UPDATE y DELETE. La sintaxis para las operaciones UPDATE y DELETE es la siguiente:

```
(UPDATE | DELETE) [FROM] NombreEntidad [WHERE condición]
```

Donde:

- La palabra clave FROM es opcional.
- La cláusula WHERE también es opcional.
- Solo puede haber una entidad mencionada en la cláusula FROM y puede tener un alias, en ese caso cualquier referencia a la propiedad tiene que ser calificada utilizando ese alias. Si el nombre de la entidad no tiene un alias entonces es ilegal calificar cualquier referencia de la propiedad.
- No se puede especificar ninguna asociación en una consulta masiva de HQL. Se pueden utilizar subconsultas en la cláusula WHERE (las subconsultas pueden contener asociaciones).

Para ejecutar un UPDATE o DELETE en HQL, utilizaremos el método **Query.executeUpdate()**. El valor *int* retornado por este método indica el número de entidades afectadas por la operación; no hemos de olvidar realizar el commit para validar la transacción. Veamos algunos ejemplos:

```
// Modificamos el salario de GIL

String hqlModif = "update Empleados e set e.salario = :nuevoSal
                  where e.apellido = :ape";
int  filasModif=session.createQuery( hqlModif )
                  .setDouble( "nuevoSal", 2500.34)
                  .setString( "ape", "GIL" )
                  .executeUpdate();

System.out.println("FILAS MODIFICADAS: "+filasModif);   //N° entidades afectadas
```

```
// Eliminamos los empleados del departamento 20

String hqlDel = "delete Empleados e where e.deptNo = ?";
int  filasDel=session.createQuery( hqlDel )
                    .setInteger( 0, 20 )
                    .executeUpdate();

System.out.println("FILAS ELIMINADAS: "+filasDel); //N° entidades afectadas

tx.rollback(); //Deshace la transacción
tx.commit();   //valida la transacción
```

La sintaxis para la operación INSERT es la siguiente:

```
INSERT INTO NombreEntidad (lista_propiedades) sentencia_select
```

Donde:

- Solo se soporta la forma INSERT INTO ... SELECT ..., no la forma INSERT INTO ... VALUES ... Es decir, solo podemos insertar datos procedentes de otra tabla.
- La lista de propiedades es análoga a la lista de columnas en la declaración INSERT de SQL.
- La *sentencia_select* puede ser cualquier consulta SELECT de HQL válida, hay que tener en cuenta que los tipos devueltos por la consulta coincidan con los esperados por el INSERT.
- Para el caso de la propiedad **id** hay dos opciones: se puede especificar en la lista de propiedades (en tal caso su valor se toma de la expresión de selección correspondiente) o se puede omitir de la lista de propiedades (en este caso se utiliza un valor generado). Esta última opción solamente está disponible cuando se utilizan generadores de id que operan en la base de datos (por ejemplo, cuando se usa AUTO_INCREMENT PRIMARY KEY en MySQL, la clave primaria se crea de forma automática sin necesidad de dar valor).

```
// Insertamos los departamentos de la tabla NUEVOS, la tabla tiene
// que estar mapeada en nuestro proyecto

String hqlInsert = "insert into Departamentos (deptNo, dnombre, loc)
                   select n.deptNo, n.dnombre, n.loc from Nuevos n";
int createdEntities = session.createQuery( hqlInsert )
                        .executeUpdate();

System.out.println("FILAS INSERTADAS: "+createdEntities); //N° entidades afectadas
```

Actividad 8: Crea desde MySQL una tabla para insertar nuevos departamentos en la tabla DEPARTAMENTOS, llamamos a la tabla NUEVOSDEP. Inserta varios departamentos en dicha tabla, mapéala en tu proyecto y después inserta los nuevos departamentos a partir de esta tabla.

Para mapear una nueva tabla podemos editar el fichero **hibernate.reveng.xml** y añadir una nueva línea con la tabla, por ejemplo: *<table-filter match-name="nuevosdep" match-catalog="ejemplo"></table-filter>*

En Oracle pondríamos : *<table-filter match-name="NUEVOSDEP"></table-filter>*

O también podemos acceder al editor de *Hibernate Reverse Engineering* (Figura 3.20) haciendo doble clic en el fichero *reveng.xml* y añadir la tabla a través de la pestaña *Table Filters*. Guardamos los cambios. A continuación generamos la nueva clase desde **Hibernate Code Generation Configurations** (Epígrafe 3.5.4). Se tiene que generar la clase y el fichero **hbm.xml.**

3.11 RESUMEN DEL LENGUAJE HQL

Las consultas en HQL no son sensibles a mayúsculas, a excepción de los nombres de las clases y propiedades Java. Podemos escribir FROM, from, SELECT, sElect, etc.

La cláusula más simple que existe en Hibernate es **from,** que obtiene todas las instancias de una clase, por ejemplo **from Empleados** obtiene todas las instancias de la clase Empleados (en SQL, obtiene todas las filas de la tabla EMPLEADOS). La cláusula **order by** ordena los resultados de la consulta.

La cláusula **where** permite refinar la lista de instancias retornadas y **order by** ordena la lista. Ejemplos:

```
from Empleados where deptNo=10 order by apellido
from Empleados as em where deptNo=10 order by 1 desc
from Empleados as em where em.deptNo=10
```

Podemos asignar alias a las clases usando la cláusula AS: *from Empleados as em,* o sin usar dicha cláusula: *from Empleados em.*

Pueden aparecer múltiples clases a la derecha de FROM, lo que causa un producto cartesiano o una unión "cruzada" (cross join): *from Empleados as em, Departamentos as dep.*

Para obtener determinadas propiedades (columnas) en una consulta utilizamos la cláusula SELECT: *select apellido, salario from Empleados,* obtiene los atributos apellido y salario de la clase Empleados.

Las consultas pueden retornar múltiples objetos y/o propiedades como un array de tipo **Object[],** una lista, un map o una clase. En apartados anteriores vimos algunos ejemplos.

Las funciones de grupo soportadas son las siguientes, la semántica es similar a SQL:

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Se puede utilizar alias para nombrar los atributos y expresiones. Se pueden utilizar operadores aritméticos, de concatenación y funciones SQL reconocidas en la cláusula SELECT. Veamos algunos ejemplos:

```
select avg(salario) as med, count(empNo) as c from Empleados
select avg(salario) , count(empNo)  from Empleados
select avg(salario) + sum(salario), count(empNo) from Empleados
select apellido ||'*'||oficio as campo from Empleados
select count(distinct deptNo) from Empleados
```

Las expresiones utilizadas en la cláusula WHERE incluyen lo siguiente:

- operadores matemáticos: `+, -, *, /`
- operadores de comparación binarios: `=, >=, <=, <>, !=, like`
- operadores lógicos `and, or, not`
- Paréntesis `( )` que indican agrupación
- `in, not in, between, is null, is not null, is empty, is not empty, member of` y `not member of`
- Caso "`simple`", `case ... when ... then ... else ... end`, y caso "buscado", `case when ... then ... else ... end`
- concatenación de cadenas `...||...` o `concat(...,...)`
- `current_date(), current_time()` y `current_timestamp()`
- `second(...), minute(...), hour(...), day(...), month(...)`, and `year(...)`
- Cualquier función u operador definido por EJB-QL 3.0: `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()`
- `coalesce()` y `nullif()`
- `str()` para convertir valores numéricos o temporales a una cadena legible.
- `cast(... as ...)`, donde el segundo argumento es el nombre de un tipo de Hibernate y `extract(... from ...)` si `cast()` y `extract()` es soportado por la base de datos subyacente.
- la función `index()` de HQL, que se aplica a alias de una colección indexada unida.
- Las funciones de HQL que tomen expresiones de ruta valuadas en colecciones: `size(), minelement(), maxelement(), minindex(), maxindex(),` junto con las funciones especiales `elements()` e `indices`, las cuales se pueden cuantificar utilizando `some, all, exists, any, in.`
- Cualquier función escalar SQL soportada por la base de datos como `sign(), trunc(), rtrim()` y `sin()`
- parámetros posicionales JDBC ?
- parámetros con nombre : `name, :start_date` y `:x1`
- literales SQL `'foo', 69, 6.66E+2, '1970-01-01 10:00:01.0'`
- constantes Java.

(Fuente: http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html#queryhql-expressions*)*

Ejemplos:

```
from Empleados where deptNo in (10,20)
from Empleados where deptNo not in (10,20)
from Empleados where salario between 2000 and 3000
from Empleados where salario not between 2000 and 3000
from Empleados where comision is null
from Empleados where comision is not null
```



```
select lower(apellido),coalesce(comision,0) from Empleados
select apellido from Empleados where apellido like 'A%'
```

Se pueden agrupar consultas usando **group by** y **having.** Las funciones SQL y las funciones de agregación SQL están permitidas en las cláusulas *having* y *order by*, si están soportadas por la base de datos subyacente. Ni la cláusula *group by* ni la cláusula *order by* pueden contener expresiones aritméticas. Ejemplos:

```
select de.dnombre, avg(em.salario) from Empleados em, Departamentos de
where em.deptNo = de.deptNo
group by de.dnombre
having avg(em.salario) > 2000
```

Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis. Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior). Ejemplos:

```
from Empleados as em where em.salario >
   (select avg(em2.salario) from Empleados em2 where em2.deptNo=em.deptNo)

from Empleados as em where em.salario >
   (select avg(salario) from Empleados)
```

3.11.1 ASOCIACIONES Y UNIONES (JOINS)

En el ejemplo de mapeo de las tablas Oracle, vimos cómo se generaban de forma automática las asociaciones entre las tablas. A continuación vamos a ver cómo se realizan asociaciones de forma manual sobre las tablas MySQL.

Para poder realizar en HQL asociaciones con **join** hemos de realizar unos cambios en nuestros ficheros XML. En nuestra base de datos tenemos las tablas EMPLEADOS y DEPARTAMENTOS, en cada una tenemos un campo clave primaria y otro clave ajena:

Tabla DEPARTAMENTOS: Clave primaria columna *dept_no.*
Tabla EMPLEADOS: Clave primaria columna *emp_no,* clave ajena columna *dept_no* que referencia a la tabla DEPARTAMENTOS.

Podemos decir que tenemos una relación de uno a muchos *(one-to-many)* entre las clases persistentes Departamentos y Empleados, significa que en la clase Departamentos tendremos una colección de objetos de la clase Empleados, es decir por cada departamento podemos tener muchos empleados.

Para mapear esta relación añadimos las siguientes líneas al *fichero* XML **Departamentos.hbm.xml** antes de la finalización de la definición de la clase (*</class>*):

```
. . . . . .
        <set name="enlace" table="empleados" >
            <key column="dept_no" />
            <one-to-many class="primero.Empleados" />
```

```
                </set>
            </class>
</hibernate-mapping>
```

En donde la colección se indica mediante la etiqueta **set,** que se le da un nombre *(enlace),* se indica el nombre de la tabla de donde se tomará esa colección de objetos *(empleados),* la columna de la tabla por la que se relacionan *(dept_no),* el tipo de relación *(one-to-many)* y la clase con la que se establece la relación *(primero.Empleados).* También es necesario añadir a la clase **Departamentos.java** la colección (enlace) con los *setter* y los *getter:*

```
private Set  enlace =  new HashSet();

public Set getEnlace() {
   return enlace;
}
public void setEnlace(Set enlace) {
   this.enlace = enlace;
}
```

Una vez realizados los cambios se pueden realizar consultas con joins. Los tipos de joins soportados son *inner join, left outer join (left join), right outer join (right join), full join* (no es útil usualmente). Aunque la forma de utilizarlos es diferente a la usada en MySQL. El siguiente ejemplo:

```
from Departamentos as dep left outer join dep.enlace
```

Devuelve tantas instancias de dos objetos como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo aquellos departamentos que no tengan empleados. Se obtiene por cada fila un objeto Departamento y un objeto Empleado. En estos joins no es necesario especificar en la cláusula FROM las instancias (tablas) que se combinan, solo hay que hacer el join con el atributo *enlace* que es donde se define la asociación.

A la derecha de *join* se puede poner la cláusula **fetch,** entonces el resultado es diferente. La siguiente orden:

```
from Departamentos as dep left outer join fetch dep.enlace
```

Obtiene tantas instancias como resulte de combinar las tablas EMPLEADOS y DEPARTAMENTOS, incluyendo aquellos departamentos que no tengan empleados, pero solo del objeto Departamentos con los datos de sus empleados cargados en la colección (atributo *enlace).* La salida de un join se puede ordenar:

```
from Departamentos as dep right outer join fetch dep.enlace order by dep.deptNo
```

El siguiente ejemplo recupera los datos de la consulta mediante dos objetos, el primero con los datos del departamento y el siguiente con los del empleado:

```
cons =  session.createQuery("from Departamentos as dep left join  dep.enlace
            order by dep.deptNo");
```



```
Iterator q = cons.iterate();
while (q.hasNext()) {
  Object[] par =(Object[]) q.next();          //obtener los dos objetos
  Departamentos de = (Departamentos) par[0]; //primer objeto Departamento
  Empleados em = (Empleados) par[1];          //siguiente objeto Empleado
  if(em !=null && de !=null)
     System.out.println(em.getApellido() +"*"+em.getSalario() +
              "*" + de.getDnombre()+"*"+de.getLoc());
     if(em ==null && de !=null) //departamento sin empleados
          System.out.println("No hay empleados..." +
                          de.getDnombre()+"*"+de.getLoc());
}
```

El siguiente ejemplo recupera los datos de la consulta mediante **FETCH,** lo que indica que por cada departamento se carga una colección con los datos de los empleados de ese departamento:

```
cons =  session.createQuery("from Departamentos as dep left join fetch dep.enlace
               order by dep.deptNo");
Departamentos depar = new Departamentos();
Empleados emple = new Empleados();
List <Departamentos> lis = cons.list();
Iterator <Departamentos> ite = lis.iterator();
depar = (Departamentos) ite.next();
System.out.println( "========================================");
System.out.println( depar.getDeptNo() +"*"+depar.getDnombre()+"*"+depar.getLoc());
while (ite.hasNext())
{
   Set set= depar.getEnlace();//obtengo todos los empleados del departamento
   System.out.println("Numero empleados: " + set.size());
   System.out.println( "----------------------------------------");
   Iterator it = set.iterator();
   while( it.hasNext()) {            //recorro la colección de empleados
     emple = (Empleados) it.next(); //obtener un empleado
     System.out.println(emple.getApellido()+"***"+emple.getOficio());
     depar = (Departamentos) ite.next();//avanza siguiente fila de departamentos
   }
   System.out.println( "========================================");
   System.out.println(depar.getDeptNo()+"*"+depar.getDnombre()+
            "*"+depar.getLoc());
}
```

En Hibernate (y en general en las bases de datos) existen 4 tipos de relaciones: uno-a-uno, uno-a-muchos, muchos-a- uno y muchos-a-muchos; en los ejemplos solo se ha usado *one-to-many* que en el modelo relacional es una asociación uno a muchos. A la hora de definir la asociación se puede especificar más atributos, en este ejemplo solo se han indicado los necesarios.

El lenguaje HQL es mucho más extenso, aquí se han expuestos las nociones mínimas para empezar a trabajar.

Más información: http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html

3.12 EJEMPLOS RESUELTOS

EJEMPLO 1:

En el siguiente ejemplo vamos a utilizar las librerías gráficas de **Java AWT** y **Swing.** Creamos un nuevo proyecto y mapeamos las tablas de EMPLEADOS y DEPARTAMENTOS de MySQL. Al ejecutar el programa nos va pedir introducir un número de departamento, véase Figura 3.38. Al pulsar en el botón *Consultar* se debe visualizar el nombre del departamento y a continuación los datos de los empleados de dicho departamento, Figura 3.39. Donde debe aparecer el número de empleados que tiene. Si el departamento no existe debe visualizar un mensaje indicándolo. Se ha utilizado una estructura **JTextarea** para visualizar los datos de empleados. Las pantallas son las siguientes:

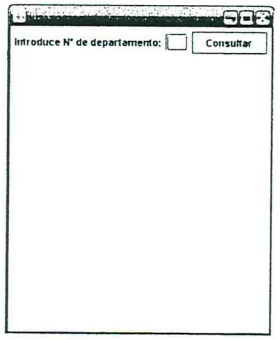


Figura 3.38. Petición del número de Departamento.

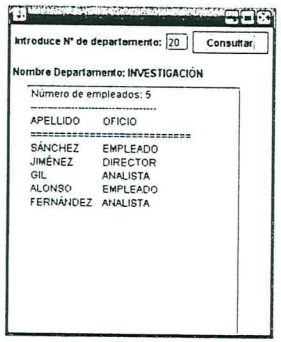


Figura 3.39. Datos del departamento introducido.

Se ha añadido la clase **SessionFactoryUtil.java** que ya vimos en ejemplos anteriores como se utiliza. Y se han desarrollado dos clases Java: **Main.Java** y **Pantalla.Java.** Desde la clase main se crea una instancia de la clase Pantalla y se llama al método *iniciar()* que pinta la pantalla y espera que introduzcamos un valor y pulsemos el botón para consultar. La clase **Main.Java** es la siguiente:

```
public class Main {
    public static void main(String[] args) {

      final Pantalla ventana = new Pantalla();
        ventana.iniciar();
    }
}//fin de la clase Main
```

En la clase Pantalla se definen varios métodos, *iniciar(), actionPerformed(ActionEvent e)* desde donde se controla la acción a realizar cuando pulsemos el botón para consultar; desde aquí se llama al método *VisualizarDep (int dep),* que recibe un número de departamento y visualiza sus datos. Desde este método se llama al método *VisualizarEmp (int dep)* que recibe el número de departamento y visualiza los datos de los empleados de ese departamento. El código es el siguiente:


```
import java.awt.Color;
import java.awt.FlowLayout;
import  java.awt.GridLayout;
import  java.awt.Rectangle;
import  java.awt.event.ActionEvent;
import  java.awt.event.ActionListener;

import  java.util.Iterator;
import  java.util.List;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import primero.*;

public class Pantalla extends JFrame implements ActionListener
{
    private static final long serialVersionUID = 1L;
    JLabel etiqueta=new JLabel("");
    JLabel label = new JLabel("Introduce N° de departamento:");
    JTextField depar = new JTextField(2);
    JButton boton = new JButton("Consultar");

    // contenedor para botones
    public void iniciar() {
      getContentPane().setLayout(new GridLayout(10, 1));
      // contenedor para botones
      JPanel panelBotones = new JPanel(new FlowLayout());
      panelBotones.add(label);
      panelBotones.add(depar);
      panelBotones.add(boton);

      getContentPane().add(panelBotones);
      getContentPane().add(etiqueta);
      setVisible(true);
      pack();
      boton.addActionListener(this);  //pulsamos el botón
    }

    //acción cuando pulsamos botones
    public void actionPerformed(ActionEvent e)    {
      if (e.getSource() == boton) { //SE PULSA EL BOTON
      etiqueta.setText(" Departamento a consultar: "+depar.getText());
      int dep;
      try {
          dep=Integer.parseInt(depar.getText());
```

```
        VisualizarDep(dep);//visualiza datos del departamento
    } catch(java.lang.NumberFormatException ex){
          etiqueta.setText(" DEPARTAMENTO ERRÓNEO");
 }
   }
 }

 //LOCALIZAR DATOS DEL DEPARTAMENTO
 void VisualizarDep (int dep) { //pinto datos del dep
        SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
        Session session = sesion.openSession();

        Departamentos depart = (Departamentos) session.createQuery(
                        "from Departamentos as de " +
                        "where de.deptNo = ?").setInteger(0, dep).uniqueResult();
        if (depart != null){
          etiqueta.setText(" Nombre Departamento: "+depart.getDnombre());
          VisualizarEmp(dep); //visualizo los empleados
        }
        else
               etiqueta.setText(" NO EXISTE EL DEPARTAMENTO.....");
        session.close();

  }//fin VisualizarDep

 //LOCALIZAR DATOS DE LOS EMPLEADOS
 void VisualizarEmp (int dep)  //pinto datos de los empleados del depart
 {
   JTextArea  area = new JTextArea();
   JScrollPane scroll= new  JScrollPane(area);
   //preparo el área para escribir los empleados
   area.setBounds(new Rectangle(20,70,250,400));
   scroll.setBounds(new Rectangle(20,70,250,400));
   area.setEditable(false);
   getContentPane().add(scroll,null);
   getContentPane().setLayout(null);
   area.setForeground(Color.blue);
   //
   SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
   Session session = sesion.openSession();

   Query q = session.createQuery(" from Empleados e where e.deptNo = :ndep");
   q.setInteger("ndep",dep);
   Empleados emple = new Empleados();
   List <Empleados> lista = q.list();

   area.append(" Número de empleados: " + lista.size()+ "\n");
   area.append(" ------------------------------------");
   area.append("\n APELLIDO\tOFICIO");
   area.append("\n ==========================");
   Iterator <Empleados> iter = lista.iterator();
   while (iter.hasNext())
         {
          emple = (Empleados) iter.next();
          String cad ="\n "+emple.getApellido() + "\t"+emple.getOficio();
```



```
          area.append(cad);
         }
    session.close();

      }//fin VisualizarEmp
}//fin de la clase Pantalla
```

Para ejecutar nuestra aplicación en otra máquina creamos el ejecutable de la siguiente manera. Pulsamos en nuestro proyecto con el botón derecho del ratón y seleccionamos la opción *Export* . Se abre una pantalla, seleccionamos *Runnable JAR File* y pulsamos el botón *Next*, Figura 3.40.

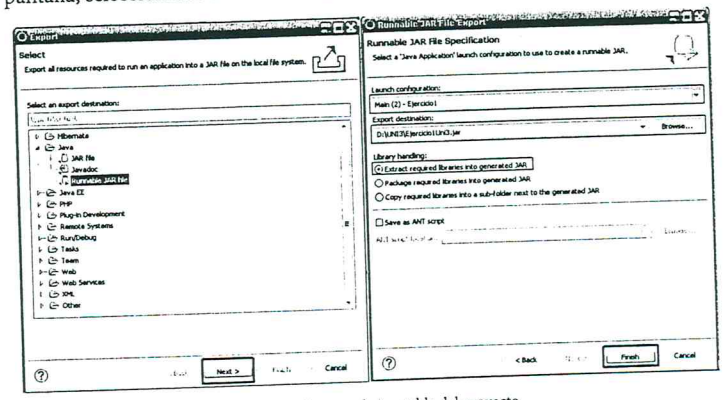


Figura 3.40. Generar el ejecutable del proyecto.

En la siguiente pantalla seleccionamos nuestro proyecto Java de la lista y buscamos el destino donde alojar el ejecutable, en el ejemplo el JAR que se genere se guardará en la carpeta *D:\UNI3* y el nombre que le daremos es *Ejercicio1Uni3.jar*, seleccionamos *Extract required libraries into generated JAR*. Pulsamos *Finish* para finalizar, Figura 3.40. Aparece una nueva ventanita, pulsamos *OK* y para finalizar, pulsamos otra vez *OK* (puede ocurrir que al realizar el empaquetamiento aparezcan algunos errores warnings).

Para ejecutar nuestra aplicación Java nos dirigimos a la carpeta donde está desde la línea de comandos y la ejecutamos de la siguiente manera:

```
D:\UNI3>java -jar Ejercicio1Uni3.jar
```

EJEMPLO 2

El siguiente ejemplo realiza las altas bajas y modificaciones sobre la tabla DEPARTAMENTOS. Al ejecutar el programa se visualiza la pantalla mostrada en la Figura 3.41 (a la izquierda), donde se visualizan 3 campos de entrada para introducir los datos del departamento y 4 botones que nos permitirán dar de alta un departamento, eliminarlo, modificarlo y limpiar la pantalla. Después de dar de alta un departamento o modificarlo o eliminarlo se debe visualizar un mensaje indicando la acción realizada, véase Figura 3.41.

Se visualizarán mensajes de error cuando se intente dar de alta un departamento que exista, eliminar o modificar un departamento que NO exista y eliminar un departamento que tiene empleados, en este caso no se podrá eliminar.

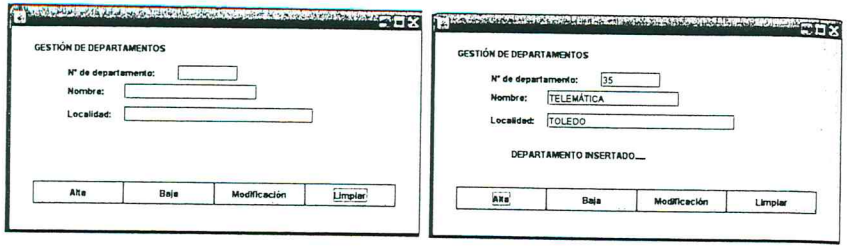


Figura 3.41. Ejercicio resuelto 2.

Igual que en el ejemplo anterior se ha añadido la clase **SessionFactoryUtil.java** y se han desarrollado dos clases Java: **Main.java** y **Pantalla.java.** La clase **Main** es igual que la anterior. En la clase Pantalla además de los métodos *iniciar()* y *actionPerformed(ActionEvent e)* se han incluido los métodos *InsertarDep (dep,nombre.toString(),loc.toString())* que inserta los datos de un departamento, el método recibe todos los datos que se insertarán; *BajaDep(dep)* para dar de baja el departamento, recibe el departamento a eliminar; *ModificaDep(dep, nombre.toString(),loc.toString())* para modificar el departamento, recibe los datos del departamento a modificar; y *LimpiarCampos()* que limpia los campos de entrada. El código es el siguiente:

```
import java.awt.Rectangle;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import primero.*;

public class Pantalla extends JFrame implements ActionListener
{
    private static final long serialVersionUID = 1L;
    JTextField depar = new JTextField(2);
    JTextField nombre = new JTextField();
    JTextField loc = new JTextField();

    JLabel etiqueta= new JLabel("");
    JLabel Lnumero = new JLabel("N° de departamento:");
```



```
    JLabel Lnombre = new JLabel("Nombre:");
    JLabel Lloc = new JLabel("Localidad:");

    JButton btnAlta= new JButton("Alta");
    JButton btnBaja=new JButton("Baja");
    JButton btnModif=new JButton("Modificación");
    JButton btnLimpiar=new JButton("Limpiar");

    //Inicia la pantalla
    public void iniciar() {
      JPanel panel = new JPanel(false);
      JLabel cab = new JLabel("GESTIÓN DE DEPARTAMENTOS");
      panel.setLayout(null);
      cab.setBounds(new Rectangle(30,15,200,25));
      panel.add(cab);

      Lnumero.setBounds(new Rectangle(75,50,140,20));
      depar.setBounds(new Rectangle(220,50,80,20));

      Lnombre.setBounds(new Rectangle(75,75,120,20));
      nombre.setBounds(new Rectangle(150,75,175,20));

      Lloc.setBounds(new Rectangle(75,105,120,20));
      loc.setBounds(new Rectangle(150,105,250,20));

      etiqueta.setBounds(new Rectangle(100,150,400,20));
      panel.add(etiqueta);

      panel.add(Lnumero);panel.add(depar,null);
      panel.add(Lnombre);panel.add(nombre);
      panel.add(Lloc);panel.add(loc);

      btnAlta.setBounds(new Rectangle(30,205,120,30));
      btnBaja.setBounds(new Rectangle(150,205,120,30));
      btnModif.setBounds(new Rectangle(270,205,120,30));
      btnLimpiar.setBounds(new Rectangle(390,205,120,30));

      panel.add(btnAlta);
      panel.add(btnBaja);
      panel.add(btnModif);
      panel.add(btnLimpiar);

      getContentPane().add(panel);
      pack();
      setSize(550,300);
      setVisible(true);

      btnAlta.addActionListener(this);  //pulsamos el botón
      btnBaja.addActionListener(this);  //pulsamos el botón
      btnModif.addActionListener(this);  //pulsamos el botón
      btnLimpiar.addActionListener(this);  //pulsamos el botón
    }//fin iniciar pantalla
    //
    public void actionPerformed(ActionEvent e) //acción cuando pulsamos botones
    {
```

```
    int dep;
    try {
          dep=Integer.parseInt(depar.getText());
      if (e.getSource() == btnAlta) { //SE PULSA EL BOTON ALTA
            InsertarDep(dep,nombre.getText(),loc.getText());
       } else
         if (e.getSource() == btnBaja) { //SE PULSA EL BOTON BAJA
             BajaDep(dep);
         } else
           if (e.getSource() == btnModif) { //SE PULSA EL BOTON MOFIFICACION
           ModificaDep(dep,nombre.getText(),loc.getText());
         }
           else { //SE PULSA LIMPIAR
             LimpiarCampos();
           }
    } catch(java.lang.NumberFormatException ex){
          etiqueta.setText(" DEPARTAMENTO ERRÓNEO");
         return;
  }
  }//fin acción de botones

  //
 void InsertarDep (int num, String nom, String loc){//INSERTO UN DEPART.
    SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
    Session session = sesion.openSession();
    Transaction tx =session.beginTransaction();

    Departamentos depart = (Departamentos) session.createQuery(
                        "from Departamentos as de " +
                        "where de.deptNo = ?").setInteger(0, num).uniqueResult();
      if (depart != null){
          etiqueta.setText(" DEPARTAMENTO EXISTENTE - NO SE PUEDE DAR DE ALTA");
          tx.rollback();
      }
      else {
          Departamentos d = new Departamentos();
          if (nom.length()>15) nom=nom.substring(0, 15);
          d.setDnombre(nom);
          if (loc.length()>15) loc=loc.substring(0, 15);
          d.setLoc(loc)       ;
          d.setDeptNo((byte) num);
          session.save(d);
          tx.commit();
          etiqueta.setText(" DEPARTAMENTO INSERTADO.....");
      }
        session.close();
      }//fin InsertarDep

  //
  void BajaDep (int num) { //ELIMINO UN DEPARTAMENTO
     SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
   Session session = sesion.openSession();
   Transaction tx =session.beginTransaction();

   Departamentos de = new Departamentos();
```



```
   de = (Departamentos) session.load(Departamentos.class, (byte)num);
   try {
       //COMPROBAR SI TIENE EMPLEADOS
       Query cons =  session.createQuery("select count(em.empNo) from
        Empleados as em where em.deptNo = ?").setInteger(0, num);
       Long numemp = (Long) cons.uniqueResult();
       if(numemp==0) { //NO TIENE EMPLEADOS
             session.delete(de); //elimina el objeto
             tx.commit();
            etiqueta.setText(" ELIMINADO ....................");
       }else { //TIENE EMPLEADOS
            etiqueta.setText(" NO SE PUEDE ELIMAR, TIENE "+numemp +
                          " EMPLEADOS............");
          tx.rollback();
       }
    }
     catch( org.hibernate.ObjectNotFoundException t) {
        etiqueta.setText("DEPARTAMENTO no EXISTENTE - NO SE PUEDE ELIMINAR");
        tx.rollback();
     }
    session.close();
  }//fin EliminarDep

  //
  void ModificaDep (int num, String nom, String loc) { //MODIFICO UN DEPART.
    SessionFactory sesion= SessionFactoryUtil.getSessionFactory();
    Session session = sesion.openSession();
    Transaction tx =session.beginTransaction();

    Departamentos de = new Departamentos();
    de = (Departamentos) session.load(Departamentos.class, (byte)num);
    try {
          if (nom.length()>15) nom=nom.substring(0, 15);
          de.setDnombre(nom);
          if (loc.length()>15) loc=loc.substring(0, 15);
          de.setLoc(loc);
          session.update(de); //modifica el objeto
          tx.commit();
         etiqueta.setText(" DEPARTAMENTO MODIFICADO.........");
     }
     catch( org.hibernate.ObjectNotFoundException t) {
        etiqueta.setText(" DEPARTAMENTO no EXISTENTE - NO SE PUEDE MODIFICAR");
        tx.rollback();
     }
     session.close();
   }//fin ModificaDep

   //
   void LimpiarCampos(){
   depar.setText("");
   nombre.setText("");
   loc.setText("");
   etiqueta.setText("");
}

}//fin de la clase Pantalla
```

3.13 EJERCICIOS

❶ ¿Qué ventajas aportan las herramientas de mapeo objeto-relacional?

❷ Cita alguno de los inconvenientes del uso de herramientas de mapeo objeto-relacional.

❸ Busca en Internet más herramientas ORM, haz una lista e indica algunas características de ellas, plataformas sobre las que se utilizan, lenguajes, si son de software libre o propietario, etc.

❹ Añade al ejemplo 2 resuelto un botón para consultar el departamento, de tal forma que al pulsarlo se visualicen los datos del departamento en caso de que exista y si no existe que se visualice un mensaje. La pantalla se muestra en la Figura 3.42.

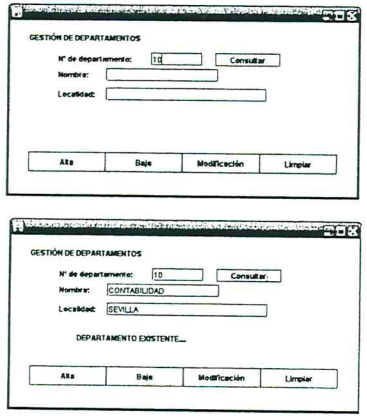


Figura 3.42. Ejercicio propuesto 4.

❺ Realiza los ejemplos resueltos 1 y 2 para las tablas EMPLEADOS y DEPARTAMENTOS de la base de datos Oracle.

❻ Crea un nuevo proyecto Java para visualizar los datos de los empleados (puedes crearlo con Oracle o con MySQL). La pantalla debe presentar 5 botones que nos permitirán iniciar la visualización y movernos entre los empleados, los botones son los siguientes:

- Botón **Ejecutar consulta:** al pulsarlo se debe visualizar el primer empleado (el que tiene menor número de empleado). Si no hay empleados visualizar mensaje indicándolo.
- Botón **Primer registro:** al pulsarlo se debe visualizar el primer empleado.
- Botón **Último registro:** al pulsarlo se debe visualizar el último empleado (el que tiene el mayor número de empleado).
- Botón **Siguiente:** al pulsarlo se debe visualizar el siguiente empleado al que se está visualizando en ese momento. Si se pulsa y estamos ante el último empleado se debe visualizar un mensaje indicándolo (por ejemplo: "No hay más empleados").
- Botón **Anterior:** al pulsarlo se debe visualizar el anterior empleado al que se está visualizando. Si se pulsa y estamos ante el primer empleado se debe visualizar un mensaje indicándolo (por ejemplo: "primer empleado").

UNIDAD 4

BASES DE DATOS OBJETO RELACIONALES Y ORIENTADAS A OBJETOS

CONTENIDOS

4.1 INTRODUCCIÓN
4.2 BASES DE DATOS OBJETO-RELACIONALES
4.2.1 Características
4.2.2 Tipos de objetos
4.2.3 Tablas de objetos
4.2.4 Tipos colección
4.2.5 Referencias
4.2.6 Herencia de tipos
4.2.7 Ejemplo de modelo relacional y objeto-relacional
4. 3 BASES DE DATOS ORIENTADAS A OBJETOS
4.3.1 Características de las bases de datos OO
4.3.2 El estándar ODMG
4.3.3 El lenguaje de consultas OQL
4. 4 EJEMPLO DE BDOO
4.4.1 Consultas sencillas
4.4.2 Consultas más complejas
4.4.3 Modo cliente/servidor de la BD
4.5 EJERCICIOS

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Identificar las ventajas e inconvenientes de las bases de datos que almacenan objetos.
- Identificar y gestionar tipos de objetos.
- Gestionar la persistencia de objetos simples y estructurados.
- Modificar los objetos almacenados.
- Identificar las características del estándar ODMG.
- Utilizar el lenguaje OQL.
- Gestionar una BDOO.

4.1 INTRODUCCIÓN

Las bases de datos constituyen una de las piezas fundamentales de muchos sistemas de información, muchas de ellas, sobre todo las más tradicionales, son difíciles de utilizar cuando las aplicaciones que acceden a los datos utilizan lenguajes de programación orientado a objetos como C++ o Java. Este fue uno de los motivos de la creación de las bases de datos orientadas a objetos, además de dar solución al surgimiento de aplicaciones más sofisticadas que necesitan tipos de datos y operaciones más complejas.

Los fabricantes de SGBD relacionales han ido incorporando en las nuevas versiones muchas de las propuestas para las bases de datos orientadas a objetos, un ejemplo son Informix, Oracle o PostgreSQL. Esto ha dado lugar al modelo relacional extendido y a los sistemas que lo implementan que son los llamados sistemas objeto-relacionales.

4.2 BASES DE DATOS OBJETO-RELACIONALES

Las Bases de Datos Objeto-Relacionales (BDOR) son una extensión de las bases de datos relacionales tradicionales a las que se les ha añadido conceptos del modelo orientado a objetos, por tanto un Sistema de Gestión de Base de Datos Objeto-Relacional (SGBDOR) contiene características del modelo relacional y del orientado a objetos; es decir, es un sistema relacional que permite almacenar objetos en las tablas.

4.2.1 CARACTERÍSTICAS

Los modelos de datos relacionales orientados a objetos extienden el modelo de datos relacional proporcionando un sistema de tipos más rico e incluyendo tipos de datos complejos y la programación orientada a objetos. Los lenguajes de consulta relacionales como SQL también necesitan ser extendidos para trabajar con estos nuevos tipos de datos. Las extensiones orientadas a objetos que comúnmente se encuentran en las bases de datos relacionales orientadas a objetos son: objetos de datos de gran tamaño, tipos de datos estructurados/abstractos, tipos de datos definidos por el usuario, tablas en tablas, secuencias, conjuntos y arrays, procedimientos almacenados, etc. Las características orientadas a objetos se definieron en el estándar SQL:1999.

En definitiva, las características más importantes de los SGBDOR son las siguientes:

- Soporte de tipos de datos básicos y complejos. El usuario puede crear sus propios tipos de datos.
- Soporte para crear métodos para esos tipos de datos. Se pueden crear funciones miembro usando tipos de datos definidos por el usuario.
- Gestión de tipos de datos complejos con un esfuerzo mínimo.
- Herencia.
- Se pueden almacenar múltiples valores en una columna de una misma fila.
- Relaciones (tablas) anidadas.
- Compatibilidad con las bases de datos relacionales tradicionales. Es decir, se pueden pasar las aplicaciones sobre bases de datos relacionales al nuevo modelo sin tener que rescribirlas.

- El inconveniente de las BDOR es que aumenta la complejidad del sistema, esto ocasiona un aumento del coste asociado.

En este apartado estudiaremos la orientación a objetos que proporciona Oracle.

4.2.2 TIPOS DE OBJETOS

Para crear tipos de objetos utilizamos la orden **CREATE TYPE**. El siguiente ejemplo crea dos objetos, un objeto que representa una dirección formada por tres atributos: calle, ciudad y código postal, cada uno de los cuales con su tipo de dato, y el siguiente representa una persona con los atributos código, nombre, dirección y fecha de nacimiento:

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
CALLE VARCHAR2(25),
CIUDAD VARCHAR2(20),
CODIGO_POST NUMBER(5)
);
/

CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
  CODIGO NUMBER,
  NOMBRE VARCHAR2(35),
  DIREC  DIRECCION,
  FECHA_NAC DATE
);
/
```

Oracle responderá con el mensaje: *Tipo creado* para cada tipo creado. Una vez creados podemos usarlos para declarar e inicializar objetos como si se tratase de cualquier otro tipo predefinido, hemos de tener en cuenta que al declarar el objeto dentro de un bloque PL/SQL hemos de inicializarlo. El siguiente ejemplo muestra la declaración y uso de los tipos creados anteriormente:

```
DECLARE
  DIR DIRECCION := DIRECCION(NULL,NULL,NULL);
  P PERSONA := PERSONA(NULL,NULL,NULL,NULL);
BEGIN
  DIR.CALLE  := 'La Mina, 3';
  DIR.CIUDAD := 'Guadalajara';
  DIR.CODIGO_POST := 19001;
  P.CODIGO := 1;
  P.NOMBRE := 'JUAN';
  P.DIREC := DIR;
  P.FECHA_NAC := '10/11/1988';
END;
/
```

Para borrar un tipo usamos la orden **DROP TYPE** indicando a la derecha el nombre de tipo a borrar: *DROP TYPE nombre_tipo;*

Actividad 1: Crea un tipo con nombre T_ALUMNO, con 4 atributos, uno de tipo PERSONA y tres que indican las notas de la primera, segunda y tercera evaluación. Después crea un bloque PL/SQL e inicializa un objeto de ese tipo.

4.2.2.1 MÉTODOS

Normalmente cuando creamos un objeto también creamos los métodos que definen el comportamiento del mismo y que permiten actuar sobre él. Los métodos son procedimientos y funciones que se especifican después de los atributos del objeto. Pueden ser de varios tipos:

- **MEMBER:** son los métodos que sirven para actuar con los objetos. Pueden ser procedimientos y funciones.
- **STATIC:** son métodos estáticos independientes de las instancias del objeto. Pueden ser procedimientos y funciones.
- **CONSTRUCTOR:** sirve para inicializar el objeto. Se trata de una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

Por cada objeto existe un constructor predefinido por Oracle, no obstante podemos sobreescribirlo y/o crear otros constructores adicionales; además creando nuestros propios constructores podemos incluir valores por defecto, restricciones, etc. Los constructores llevarán en la cláusula RETURN la expresión *RETURN SELF AS RESULT*.

El siguiente ejemplo muestra el tipo DIRECCION con la declaración de un procedimiento que asigna valor al atributo CALLE y una función que devuelve el valor del atributo CALLE (antes de ejecutar el siguiente código hemos de borrar los tipos creados anteriormente con la orden *DROP TYPE nombretipo*):

```
CREATE OR REPLACE TYPE DIRECCION AS OBJECT
(
  CALLE        VARCHAR2(25),
  CIUDAD       VARCHAR2(20),
  CODIGO_POST NUMBER(5),
  MEMBER PROCEDURE SET_CALLE(C VARCHAR2),
  MEMBER FUNCTION GET_CALLE RETURN VARCHAR2
);
/
```

El siguiente ejemplo define un tipo rectángulo con 3 atributos y un constructor que recibe 2 parámetros:

```
CREATE OR REPLACE TYPE RECTANGULO AS OBJECT
(
  BASE   NUMBER,
  ALTURA NUMBER,
  AREA   NUMBER,
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
    RETURN SELF AS RESULT
);
/
```


Una vez creado el tipo con la especificación de los métodos crearemos el cuerpo del nuevo tipo OBJECT mediante la instrucción **CREATE OR REPLACE TYPE BODY:**

```
CREATE OR REPLACE TYPE BODY nombre_del_tipo AS
  <implementación de los métodos>
END;
```

Donde *<implementación de los métodos>* tiene el siguiente;

```
[STATIC | MEMBER] PROCEDURE nombreProc [(parametro1, parámetro2, ...)]
IS
    Declaraciones;
BEGIN
    Instrucciones;
END;
```

```
[STATIC | MEMBER | CONSTRUCTOR] FUNCTION nombreFunc
              [(parametro1, parámetro2, ... )] RETURN tipo_valor_retorno
IS
    Declaraciones;
BEGIN
    Instrucciones;
END;
```

La implementación de los métodos del objeto DIRECCION es la siguiente:

```
CREATE OR REPLACE TYPE BODY DIRECCION AS
  --
  MEMBER PROCEDURE SET_CALLE(C VARCHAR2) IS
  BEGIN
    CALLE := C;
  END;
  --
  MEMBER FUNCTION GET_CALLE RETURN VARCHAR2 IS
  BEGIN
    RETURN CALLE;
  END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto DIRECCION, visualizará el nombre de la calle, al no definir constructor es necesario inicializar el objeto:

```
DECLARE
  DIR DIRECCION := DIRECCION(NULL,NULL,NULL);
BEGIN
  DIR.SET_CALLE('La Mina, 3');
  DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
END;
/
```

La implementación del método constructor del objeto RECTANGULO es la siguiente:

```
CREATE OR REPLACE TYPE BODY RECTANGULO AS
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
    RETURN SELF AS RESULT
  AS
  BEGIN
    SELF.BASE := BASE;
    SELF.ALTURA := ALTURA;
    SELF.AREA := BASE * ALTURA;
    RETURN;
  END;
END;
/
```

El siguiente bloque PL/SQL muestra el uso del objeto RECTANGULO, se puede llamar al constructor usando los 3 atributos; pero es más robusto llamarlo usando 2 atributos de esta manera nos aseguramos que el atributo AREA tiene el valor inicial correcto. En este caso no es necesario inicializar los objetos R1 y R2 ya que se inicializan al llamar al constructor con NEW:

```
DECLARE
  R1 RECTANGULO;
  R2 RECTANGULO;
  R3 RECTANGULO := RECTANGULO(NULL,NULL,NULL);
BEGIN
  R1 := NEW RECTANGULO(10,20,200);
  DBMS_OUTPUT.PUT_LINE('AREA R1:'||R1.AREA);
  R2 := NEW RECTANGULO(10,20);
  DBMS_OUTPUT.PUT_LINE('AREA R2:'||R2.AREA);
  R3.BASE := 5;
  R3.ALTURA := 15;
  R3.AREA := R3.BASE * R3.ALTURA;
END;
/
```

Para borrar el cuerpo de un tipo usamos la orden **DROP TYPE BODY** indicando a la derecha el nombre del tipo cuyo cuerpo deseamos borrar: *DROP TYPE BODY nombre_tipo.*

Actividad 2: Crea un método y el cuerpo del mismo en el tipo T_ALUMNO que devuelva la nota media del alumno.

En muchas ocasiones necesitamos comparar e incluso ordenar datos de tipos definidos como OBJECT. Para ello es necesario crear un método **MAP** u **ORDER**, debiéndose definir al menos uno de ellos por cada objeto que se quiere comparar:

- Los métodos **MAP** consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE, ...) que será el que se utilice en las comparaciones y ordenaciones aplicando los criterios establecidos para este tipo de datos.

- Un método **ORDER** utiliza los atributos del objeto sobre el que se ejecuta para realizar un cálculo y compararlo con otro objeto del mismo tipo que toma como argumento de entrada. Este método devuelve un valor negativo si el parámetro de entrada es mayor que el atributo, un valor positivo si ocurre lo contrario y cero si ambos son iguales. Suelen ser menos funcionales y eficientes, se utilizan cuando el criterio de comparación es muy complejo como para implementarlo con un método **MAP.** No lo trataremos en este tema.

Por ejemplo, la siguiente declaración indica que los objetos de tipo PERSONA se van a comparar por su atributo CODIGO:

```
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
  CODIGO NUMBER,
  NOMBRE VARCHAR2(35),
  DIREC DIRECCION,
  FECHA_NAC DATE,
  MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER
);
/
CREATE OR REPLACE TYPE BODY PERSONA AS
  MAP MEMBER FUNCTION POR_CODIGO RETURN NUMBER IS
  BEGIN
    RETURN CODIGO;
  END;
END;
/
```

El siguiente código PL/SQL compara dos objetos de tipo PERSONA, y visualiza '*OBJETOS IGUALES*' ya que el atributo CODIGO tiene el mismo valor para los dos objetos:

```
DECLARE
  P1 PERSONA := PERSONA(NULL,NULL,NULL,NULL);
  P2 PERSONA:= PERSONA(NULL,NULL,NULL,NULL);
BEGIN
  P1.CODIGO := 1;
  P1.NOMBRE := 'JUAN';
  P2.CODIGO :=1;
  P2.NOMBRE :='MANUEL';
  IF P1= P2 THEN
    DBMS_OUTPUT.PUT_LINE('OBJETOS IGUALES');
  ELSE
    DBMS_OUTPUT.PUT_LINE('OBJETOS DISTINTOS');
  END IF;
END;
/
```

Es necesario un método **MAP** u **ORDER** para comparar objetos en PL/SQL. Un tipo de objeto solo puede tener un método **MAP** o uno **ORDER.**

4.2.3. TABLAS DE OBJETOS

Una vez definidos los objetos podemos utilizarlos para definir nuevos tipos, para definir columnas de tablas de ese tipo o para definir tablas que almacenan objetos. Una tabla de objetos es una tabla que almacena un objeto en cada fila, se accede a los atributos de esos objetos como si se tratasen de columnas de la tabla. El siguiente ejemplo crea la tabla ALUMNOS de tipo PERSONA con la columna CODIGO como clave primaria y muestra su descripción:

```
CREATE TABLE ALUMNOS OF PERSONA (
  CODIGO PRIMARY KEY
);
DESC ALUMNOS;
 Nombre                                    Nulo     Tipo
 ----------------------------------------- -------- ------------------
 CODIGO                                    NOT NULL NUMBER
 NOMBRE                                             VARCHAR2(35)
 DIREC                                              DIRECCION
 FECHA_NAC                                          DATE
```

A continuación se insertan filas en la tabla ALUMNOS. Hemos de poner delante el tipo (DIRECCION) a la hora de dar valores a los atributos que forman la columna de dirección:

```
INSERT INTO ALUMNOS VALUES(
  1, 'Juan Pérez',
  DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19005),
  '18/12/1991'
);

INSERT INTO ALUMNOS (CODIGO, NOMBRE, DIREC, FECHA_NAC) VALUES (
  2, 'Julia Breña',
  DIRECCION ('C/Los espartales 25', 'GUADALAJARA', 19004),
  '18/12/1987'
);
```

Veamos algunos ejemplos de consultas sobre la tabla:

- Seleccionar aquellas filas cuya CIUDAD = 'GUADALAJARA':

```
SELECT * FROM ALUMNOS A WHERE A.DIREC.CIUDAD = 'GUADALAJARA';
```

- Para seleccionar columnas individuales, si la columna es un tipo OBJECT se necesita definir un **alias** para la tabla; en una base de datos con tipos y objetos se recomienda usar **alias** para el nombre de las tablas. A continuación seleccionamos el codigo y la dirección de los alumnos:

```
SELECT CODIGO, A.DIREC FROM ALUMNOS A;
```

- Para llamar a los métodos hay que utilizar su nombre y paréntesis que encierren los argumentos de entrada (aunque no tenga argumentos los paréntesis deben aparecer). En el siguiente ejemplo obtenemos el nombre y la calle de los alumnos, usamos el método GET_CALLE del tipo DIRECCION:


```
SELECT NOMBRE, A.DIREC.GET_CALLE() FROM ALUMNOS A;
```

- Modificamos aquellas filas cuya ciudad es GUADALAJARA, convertimos la ciudad a minúscula:

```
UPDATE ALUMNOS A
    SET A.DIREC.CIUDAD=LOWER(A.DIREC.CIUDAD)
WHERE A.DIREC.CIUDAD ='GUADALAJARA';
```

- Eliminamos aquellas filas cuya ciudad sea 'guadalajara':

```
DELETE ALUMNOS A WHERE A.DIREC.CIUDAD='guadalajara';
```

- El siguiente bloque PL/SQL muestra el nombre y la calle de los alumnos:

```
DECLARE
  CURSOR C1 IS SELECT * FROM ALUMNOS;
BEGIN
  FOR I IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.NOMBRE || ' - Calle: '|| I.DIREC.CALLE);
  END LOOP;
END;
/
```

Actividad 3: Crea la tabla ALUMNOS2 del tipo T_ALUMNO e inserta objetos en ella. Realiza luego una consulta que visualice el nombre del alumno y la nota media.

4.2.4 TIPOS COLECCIÓN

Las bases de datos relacionales orientadas a objetos pueden permitir el almacenamiento de colecciones de elementos en una única columna. Tal es el caso de los VARRAYS en Oracle, que son similares a los arrays de C, que permiten almacenar un conjunto de elementos, todos del mismo tipo, y cada elemento tiene un índice asociado; y de las tablas anidadas que permiten almacenar en una columna de una tabla otra tabla.

4.2.4.1 VARRAYS

Para crear una colección de elementos varrays se usa la orden **CREATE TYPE**. El siguiente ejemplo crea un tipo **VARRAY** de nombre TELEFONO de tres elementos donde cada elemento es del tipo VARCHAR2:

```
CREATE TYPE TELEFONO AS VARRAY(3) OF VARCHAR2(9);
```

Cuando se declara un tipo **VARRAY** no se produce ninguna reserva de espacio. Para obtener información de un **VARRAY** usamos la orden **DESC** (*DESC TELEFONO*). La vista **USER_VARRAYS** obtiene información de las tablas que tienen columnas varrays.

Veamos algunos ejemplos del uso de varrays:

Creamos una tabla donde una columna es de tipo **VARRAY**:

```
CREATE TABLE AGENDA
(
  NOMBRE VARCHAR2(15),
  TELEF TELEFONO
);
```

Insertamos varias filas:

```
INSERT INTO AGENDA VALUES
('MANUEL', TELEFONO ('656008876', '927986655', '639883300'));

INSERT INTO AGENDA (NOMBRE, TELEF) VALUES
('MARTA', TELEFONO ('649500800'));
```

En las consultas es imposible poner condiciones sobre los elementos almacenados dentro del **VARRAY**, además los valores del **VARRAY** solo pueden ser accedidos y recuperados como bloque, no se puede acceder individualmente a los elementos (desde un programa PL/SQL sí se puede). Seleccionamos determinadas columnas:

```
SELECT TELEF FROM AGENDA;
```

Podemos usar alias para seleccionar las columnas:

```
SELECT A.TELEF FROM AGENDA A;
```

Modificamos los teléfonos de MARTA:

```
UPDATE AGENDA SET TELEF=TELEFONO('649500800', '659222222') WHERE NOMBRE = 'MARTA';
```

Desde un programa PL/SQL se puede hacer un bucle para recorrer los elementos del **VARRAY**. El siguiente bloque visualiza los nombres y los teléfonos de la tabla AGENDA, *I.TELEF.COUNT* devuelve el número de elementos del **VARRAY**:

```
DECLARE
  CURSOR C1 IS SELECT * FROM AGENDA;
  CAD VARCHAR2(50);
BEGIN
  FOR I IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.NOMBRE ||', Número de Telefonos: '||I.TELEF.COUNT);
    CAD:='*';
    FOR J IN 1 .. I.TELEF.COUNT LOOP
      CAD:=CAD ||I.TELEF(J)||'*';
```



```
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(CAD);
  END LOOP;
END;
/
```

Muestra la siguiente salida:

```
MANUEL, Número de Telefonos: 3
*656008876*927986655*639883300*
MARTA, Número de Telefonos:2
*649500800*659222222*
```

El siguiente ejemplo crea un procedimiento almacenado para insertar datos en la tabla AGENDA, a continuación se muestra la llamada al procedimiento:

```
CREATE OR REPLACE PROCEDURE INSERTAR_AGENDA (N VARCHAR2, T TELEFONO) AS
BEGIN
  INSERT INTO AGENDA VALUES (N,T);
END;
/
BEGIN
  INSERTAR_AGENDA('LUIS', TELEFONO('949009977'));
  INSERTAR_AGENDA('MIGUEL', TELEFONO('949004020', '678905400'));
  COMMIT;
END;
/
```

Actividad 4: Crea una función almacenada que reciba un nombre de la agenda y devuelva el primer teléfono que tenga. Realiza un bloque PL/SQL que haga uso de la función.

4.2.4.2 TABLAS ANIDADAS

Una tabla anidada está formada por un conjunto de elementos, todos del mismo tipo. La tabla anidada está contenida en una columna y el tipo de esta columna debe ser un tipo de objeto existente en la base de datos. Para crear una tabla anidada usamos la orden **CREATE TYPE**. El siguiente ejemplo crea un tipo tabla anidada que almacenará objetos del tipo DIRECCION (creado al principio de la unidad):

```
CREATE TYPE TABLA_ANIDADA AS TABLE OF DIRECCION;
```

No es necesario especificar el tamaño máximo de una tabla anidada. Veamos cómo se define una columna de una tabla con el tipo tabla anidada creada anteriormente:

```
CREATE TABLE EJEMPLO_TABLA_ANIDADA
(
 ID NUMBER(2),
 APELLIDOS VARCHAR2(35),
```

```
 DIREC  TABLA_ANIDADA
)
NESTED TABLE DIREC STORE AS DIREC_ANIDADA;
```

La cláusula **NESTED TABLE** identifica el nombre de la columna que contendrá la tabla anidada. La cláusula **STORE AS** especifica el nombre de la tabla (DIREC_ANIDADA) en la que se van a almacenar las direcciones que se representan en el atributo DIREC de cualquier objeto de la tabla EJEMPLO_TABLA_ANIDADA. La descripción del tipo TABLA_ANIDADA y de la tabla EJEMPLO_TABLA_ANIDADA es la siguiente:

```
SQL> DESC TABLA_ANIDADA;
 TABLA_ANIDADA TABLE OF DIRECCION
 Nombre                                    Nulo   Tipo
 ----------------------------------------- ------ -------------------------
 CALLE                                            VARCHAR2(25)
 CIUDAD                                           VARCHAR2(20)
 CODIGO_POST                                      NUMBER(5)

METHOD
------
 MEMBER PROCEDURE SET_CALLE
 Nombre de Argumento            Tipo                    E/S    Por Defecto
 ------------------------------ ----------------------- ------ --------
 C                              VARCHAR2                IN

METHOD
------
 MEMBER FUNCTION GET_CALLE RETURNS VARCHAR2

SQL> DESC EJEMPLO_TABLA_ANIDADA ;
 Nombre                                    Nulo   Tipo
 ----------------------------------------- ------ ----------------------
 ID                                               NUMBER(2)
 APELLIDOS                                        VARCHAR2(35)
 DIREC                                            TABLA_ANIDADA
```

Veamos algunos ejemplos con la tabla. Insertamos varias filas con varias direcciones en la tabla EJEMPLO_TABLA_ANIDADA:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (1, 'RAMOS',
  TABLA_ANIDADA (
    DIRECCION ('C/Los manantiales 5', 'GUADALAJARA', 19004),
    DIRECCION ('C/Los manantiales 10', 'GUADALAJARA', 19004),
    DIRECCION ('C/Av de Paris 25', 'CÁCERES', 10005),
    DIRECCION ('C/Segovia 23-3A', 'TOLEDO', 45005)
  )
);

INSERT INTO EJEMPLO_TABLA_ANIDADA VALUES (2, 'MARTÍN',
  TABLA_ANIDADA (
    DIRECCION ('C/Huesca 5', 'ALCALÁ DE H', 28804),
    DIRECCION ('C/Madrid 20', 'ALCORCÓN', 28921)
  )
);
```


Seleccionamos todas las filas: `SELECT * FROM EJEMPLO_TABLA_ANIDADA;`

Obtenemos el identificador, los apellidos y las calles de cada fila de la tabla, se obtienen tantas filas como filas hay en la tabla:

```
SELECT ID, APELLIDOS, CURSOR(SELECT TT.CALLE FROM TABLE(T.DIREC) TT)
FROM EJEMPLO_TABLA_ANIDADA T;
```

Se pueden usar cursores dentro de una SELECT para acceder o poner condiciones a las filas de una tabla anidada.

Obtenemos las calles de la fila con ID=1 cuya ciudad sea GUADALAJARA, se obtienen tantas filas como calles hay en la ciudad de GUADALAJARA:

```
SELECT CALLE FROM THE
    (SELECT T.DIREC FROM EJEMPLO_TABLA_ANIDADA T WHERE ID=1)
WHERE CIUDAD='GUADALAJARA';
```

La cláusula **THE** también sirve para seleccionar filas en una tabla anidada, la sintaxis es: *SELECT ... FROM THE (subconsulta) WHERE ...*

Insertamos una dirección al final de la tabla anidada para el identificador 1 (ahora el identificador 1 tendrá cinco direcciones):

```
INSERT INTO TABLE (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)
VALUES (DIRECCION ('C/Los manantiales 15', 'GUADALAJARA', 19005));
```

El operador **TABLE** se utiliza para acceder a la fila que nos interesa, en este caso la que tiene ID=1.

Modificamos la primera dirección del identificador 1:

```
UPDATE TABLE (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET VALUE(PRIMERA) = DIRECCION ('C/Pilon 11', 'TOLEDO', 45589)
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 5', 'GUADALAJARA', 19005);
```

El alias PRIMERA recoge los datos devueltos por la SELECT, que debe devolver una fila. Para obtener el objeto almacenado en una fila (y no solo el valor de los campos de dicho objeto) se necesita la función **VALUE**.

Eliminamos la segunda dirección del identificador 1:

```
DELETE FROM TABLE (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 10', 'GUADALAJARA', 19005);
```

El siguiente bloque PL/SQL crea un procedimiento que recibe un identificador y visualiza las calles que tiene, debajo se muestra el bloque PL/SQL que prueba el procedimiento:

```
CREATE OR REPLACE PROCEDURE VER_DIREC(IDENT NUMBER) AS
 CURSOR C1 IS
    SELECT CALLE FROM THE
    (SELECT T.DIREC FROM EJEMPLO_TABLA_ANIDADA T WHERE ID = IDENT);
BEGIN
  FOR I IN C1 LOOP
    DBMS_OUTPUT.PUT_LINE(I.CALLE);
  END LOOP;
END VER_DIREC;
/
BEGIN
  VER_DIREC(1);
END;
/
```

La vista **USER_NESTED_TABLES** obtiene información de las tablas anidadas.

4.2.5 REFERENCIAS

Mediante el operador **REF** asociado a un atributo se pueden definir referencias a otros objetos. Un atributo de este tipo almacena una referencia al objeto del tipo definido e implementa una relación de asociación entre los dos tipos de objetos. Una columna de tipo **REF** guarda un puntero a una fila de la otra tabla, contiene el OID (identificador del objeto fila) de dicha fila.

El siguiente ejemplo crea un tipo EMPLEADO_T donde uno de los atributos es una referencia a un objeto EMPLEADO_T, después se crea una tabla de objetos EMPLEADO_T:

```
CREATE TYPE EMPLEADO_T AS OBJECT (
  NOMBRE     VARCHAR2(30),
  JEFE       REF EMPLEADO_T
);
/
CREATE TABLE EMPLEADO OF EMPLEADO_T;
```

Insertamos filas en la tabla, el segundo INSERT asigna al atributo JEFE la referencia al objeto con apellido GIL:

```
INSERT INTO EMPLEADO VALUES (EMPLEADO_T ('GIL', NULL));

INSERT INTO EMPLEADO
  SELECT EMPLEADO_T ('ARROYO', REF(E))
    FROM EMPLEADO E WHERE E.NOMBRE = 'GIL';
```

Para acceder al objeto referido por un **REF** se utiliza el operador **DEREF**, en el ejemplo se visualiza el nombre del empleado y los datos del jefe de cada empleado:

```
 SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;
```

La siguiente consulta obtiene el identificador del objeto cuyo nombre es GIL:

```
SELECT REF(P) FROM EMPLEADO P WHERE NOMBRE='GIL';
```


4.2.6 HERENCIA DE TIPOS

La herencia facilita la creación de objetos a partir de otros ya existentes e implica que un subtipo obtenga todo el comportamiento (métodos) y eventualmente los atributos de su supertipo. Los subtipos definen sus propios atributos y métodos y puede redefinir los métodos que heredan, esto se conoce como polimorfismo. El siguiente ejemplo define un tipo persona y a continuación el subtipo tipo alumno:

```
-- Se define el tipo persona
--
CREATE OR REPLACE TYPE TIPO_PERSONA AS OBJECT(
 DNI VARCHAR2(10),
 NOMBRE VARCHAR2(25),
 FEC_NAC DATE,
 MEMBER FUNCTION EDAD RETURN NUMBER,
 FINAL MEMBER FUNCTION GET_DNI RETURN VARCHAR2,
 MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2,
 MEMBER PROCEDURE VER_DATOS
) NOT FINAL;
/
-- Cuerpo del tipo persona
--
CREATE OR REPLACE TYPE BODY TIPO_PERSONA AS
 MEMBER FUNCTION EDAD RETURN NUMBER IS
  ED NUMBER;
 BEGIN
    ED := TO_CHAR(SYSDATE, 'YYYY') - TO_CHAR(FEC_NAC, 'YYYY');
    RETURN ED;
 END;
 --
 FINAL MEMBER FUNCTION GET_DNI RETURN VARCHAR2 IS
 BEGIN
   RETURN DNI;
 END;
 --
 MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2 IS
 BEGIN
   RETURN NOMBRE;
 END;
 --
 MEMBER PROCEDURE VER_DATOS IS
 BEGIN
   DBMS_OUTPUT.PUT_LINE(DNI|| '*'||NOMBRE||'*'||EDAD());
 END;
END;
/
-- Se define el tipo alumno
--
CREATE OR REPLACE TYPE TIPO_ALUMNO UNDER TIPO_PERSONA(
 CURSO VARCHAR2(10),
 NOTA_FINAL NUMBER,
 MEMBER FUNCTION NOTA RETURN NUMBER,
 OVERRIDING MEMBER PROCEDURE VER_DATOS
```

```
);
/
-- Cuerpo del tipo alumno
--
CREATE OR REPLACE TYPE BODY  TIPO_ALUMNO AS
 MEMBER FUNCTION NOTA RETURN NUMBER IS
 BEGIN
   RETURN NOTA_FINAL;
 END;
 --
 OVERRIDING MEMBER PROCEDURE VER_DATOS IS
 BEGIN
    DBMS_OUTPUT.PUT_LINE(CURSO|| '*'||NOTA_FINAL);
 END;
END;
/
```

Mediante la cláusula **NOT FINAL** (incluida al final de la definición del tipo) se indica que se pueden derivar subtipos, si no se incluye esta cláusula se considera que es **FINAL** (no puede tener subtipos). Igualmente si un método es **FINAL** los subtipos no pueden redefinirlo. La cláusula **OVERRIDING** se utiliza para redefinir el método. El siguiente bloque PL/SQL muestra un ejemplo de uso de los tipos definidos, al definir el objeto se inicializan todos los atributos ya que no se ha definido constructor para inicializar el objeto:

```
DECLARE
 A1 TIPO_ALUMNO := TIPO_ALUMNO(NULL,NULL,NULL,NULL,NULL);
 A2 TIPO_ALUMNO := TIPO_ALUMNO('871234533A', 'PEDRO', '12/12/1996','SEGUNDO',7);
 NOM A1.NOMBRE%TYPE;
 DNI A1.DNI%TYPE;
 NOTAF A1.NOTA_FINAL%TYPE;
BEGIN
 A1.NOTA_FINAL:=8;
 A1.CURSO:='PRIMERO';
 A1.NOMBRE:='JUAN';
 A1.FEC_NAC:='20/10/1997';
 A1.VER_DATOS;
 NOM := A2.GET_NOMBRE();
 DNI := A2.GET_DNI();
 NOTAF := A2.NOTA();
 A2.VER_DATOS;
 DBMS_OUTPUT.PUT_LINE(A1.EDAD());
 DBMS_OUTPUT.PUT_LINE(A2.EDAD());
END;
/
```

A continuación se crea una tabla de TIPO_ALUMNO con el DNI como clave primaria, se insertan filas y se realiza alguna consulta:

```
CREATE TABLE TALUMNOS OF TIPO_ALUMNO (DNI PRIMARY KEY);

INSERT INTO TALUMNOS VALUES ('871234533A', 'PEDRO', '12/12/1996','SEGUNDO',7);
INSERT INTO TALUMNOS VALUES ('809004534B', 'MANUEL', '12/12/1997','TERCERO',8);
```



```
SELECT * FROM TALUMNOS;
SELECT DNI, NOMBRE, CURSO, NOTA_FINAL FROM TALUMNOS;
SELECT P.GET_DNI(), P.GET_NOMBRE(), P.EDAD(), P.NOTA() FROM TALUMNOS P;
```

4.2.7 EJEMPLO DE MODELO RELACIONAL Y OBJETO-RELACIONAL

A continuación vamos a ver una solución con el modelo relacional para gestión de ventas y otra usando el enfoque objeto-relacional. En la Figura 4.1 se muestra el modelo de datos para las tablas CLIENTES, PRODUCTOS, VENTAS y LINEASVENTAS. Las órdenes de creación de las tablas son:

```
CREATE TABLE CLIENTES
IDCLIENTE NUMBER PRIMARY KEY,
  NOMBRE      VARCHAR2(50),
  DIRECCION   VARCHAR2(50),
  POBLACION   VARCHAR2(50),
  CODPOSTAL   NUMBER(5),
  PROVINCIA   VARCHAR2(40),
  NIF         VARCHAR2(9) UNIQUE,
  TELEFONO1   VARCHAR2(15),
  TELEFONO2   VARCHAR2(15),
  TELEFONO3   VARCHAR2(15)
);
CREATE TABLE PRODUCTOS (
  IDPRODUCTO  NUMBER PRIMARY KEY,
  DESCRIPCION varchar2(80),
  PVP   NUMBER,
  STOCKACTUAL NUMBER
  );
```

```
CREATE TABLE VENTAS(
  IDVENTA     NUMBER PRIMARY KEY,
  IDCLIENTE   NUMBER NOT NULL
              REFERENCES CLIENTES,
  FECHAVENTA  DATE
);
CREATE TABLE LINEASVENTAS (
  IDVENTA       NUMBER,
  NUMEROLINEA   NUMBER,
  IDPRODUCTO    NUMBER,
  CANTIDAD      NUMBER,
  FOREIGN KEY (IDVENTA)
          REFERENCES VENTAS (IDVENTA),
  FOREIGN KEY (IDPRODUCTO)
       REFERENCES PRODUCTOS(IDPRODUCTO),
  PRIMARY KEY (IDVENTA,NUMEROLINEA)
);
```

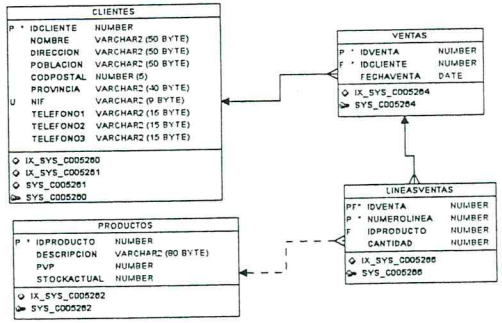


Figura 4.1. Modelo de datos.

Definimos los siguientes tipos:

Definimos un tipo VARRAY de 3 elementos para contener los teléfonos:

```
CREATE TYPE TIP_TELEFONOS AS VARRAY(3) OF VARCHAR2(15);
/
```

A continuación se crean los tipos dirección, cliente, producto y línea de venta:

```
CREATE TYPE TIP_DIRECCION AS OBJECT(
 CALLE       VARCHAR2(50),
  POBLACION  VARCHAR2(50),
  CODPOSTAL  NUMBER(5),
  PROVINCIA  VARCHAR2(40)
);
/
CREATE TYPE TIP_CLIENTE AS OBJECT(
  IDCLIENTE  NUMBER,
  NOMBRE     VARCHAR2(50),
  DIREC      TIP_DIRECCION,
  NIF        VARCHAR2(9),
  TELEF      TIP_TELEFONOS
);
/
```

```
CREATE TYPE TIP_PRODUCTO AS OBJECT (
  IDPRODUCTO    NUMBER,
  DESCRIPCION   VARCHAR2(80),
  PVP           NUMBER,
  STOCKACTUAL   NUMBER
);
/

CREATE TYPE  TIP_LINEAVENTA AS OBJECT (
  NUMEROLINEA   NUMBER,
  IDPRODUCTO    REF TIP_PRODUCTO,
  CANTIDAD      NUMBER
);
/
```

Creamos un tipo tabla anidada para contener las líneas de una venta:

```
CREATE TYPE  TIP_LINEAS_VENTA AS TABLE OF TIP_LINEAVENTA;
/
```

Creamos un tipo venta para los datos de las ventas, cada venta tendrá un atributo LINEAS del tipo tabla anidada definida anteriormente:

```
CREATE TYPE  TIP_VENTA AS OBJECT (
  IDVENTA      NUMBER,
  IDCLIENTE    REF TIP_CLIENTE,
  FECHAVENTA   DATE,
  LINEAS TIP_LINEAS_VENTA,
  MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER
);
/
```

En el tipo TIP_VENTA se ha definido la función miembro TOTAL_VENTA que calcula el total de la venta de las líneas de venta que forman parte de una venta. COUNT cuenta el número de elementos de una tabla o de un array, LINEAS.COUNT devuelve el número de líneas que tiene la venta.

```
CREATE OR REPLACE TYPE BODY TIP_VENTA AS
  MEMBER FUNCTION TOTAL_VENTA RETURN NUMBER IS
   TOTAL NUMBER:=0;
   LINEA   TIP_LINEAVENTA;
   PRODUCT TIP_PRODUCTO;
  BEGIN
    FOR I IN 1..LINEAS.COUNT LOOP
      LINEA:=LINEAS(I);
      SELECT DEREF(LINEA.IDPRODUCTO) INTO PRODUCT FROM DUAL;
      TOTAL:=TOTAL + LINEA.CANTIDAD * PRODUCT.PVP;
    END LOOP;
    RETURN TOTAL;
  END;
END;
/
```


Creamos las tablas donde almacenar los objetos de la aplicación, la tabla para los clientes, los productos y las ventas, también se definen las claves primarias de dichas tablas:

```
CREATE TABLE TABLA_CLIENTES OF TIP_CLIENTE (
  IDCLIENTE PRIMARY KEY,
  NIF UNIQUE
);
/
CREATE TABLE TABLA_PRODUCTOS OF TIP_PRODUCTO (
  IDPRODUCTO PRIMARY KEY
);
/
CREATE TABLE TABLA_VENTAS OF TIP_VENTA (
  IDVENTA PRIMARY KEY
)NESTED TABLE LINEAS STORE AS TABLA_LINEAS;
/
```

En la tabla TABLA_VENTAS se define una tabla anidadada para el atributo LINEAS del tipo TIP_VENTA, contendrá las líneas de venta.

Insertamos 2 clientes y 5 productos:

```
INSERT INTO TABLA_CLIENTES VALUES
(1, 'Luis Gracia',
 TIP_DIRECCION ('C/Las Flores 23', 'Guadalajara', '19003', 'Guadalajara'),
 '34343434L', TIP_TELEFONOS( '949876655', '949876655')
);

INSERT INTO TABLA_CLIENTES VALUES
(2, 'Ana Serrano',
 TIP_DIRECCION ('C/Galiana 6', 'Guadalajara', '19004', 'Guadalajara'),
  '76767667F', TIP_TELEFONOS('94980009')
);

INSERT INTO TABLA_PRODUCTOS VALUES (1, 'CAJA DE CRISTAL DE MURANO', 100, 5);
INSERT INTO TABLA_PRODUCTOS VALUES (2, 'BICICLETA CITY', 120, 15);
INSERT INTO TABLA_PRODUCTOS VALUES (3, '100 LÁPICES DE COLORES', 20, 5);
INSERT INTO TABLA_PRODUCTOS VALUES (4, 'OPERACIONES CON BD', 25, 5);
INSERT INTO TABLA_PRODUCTOS VALUES (5, 'APLICACIONES WEB', 25.50, 10);
```

Inserto en TABLA_VENTAS la venta con IDVENTA 1 para el IDCLIENTE 1:

```
INSERT INTO TABLA_VENTAS
SELECT 1, REF(C), SYSDATE, TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE=1;
```

Inserto en TABLA_VENTAS dos líneas de venta para el IDVENTA 1 para los productos 1 (la CANTIDAD es 1) y 2 (la CANTIDAD es 2):

```
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=1)
(SELECT 1, REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=1);
```

```
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=1)
(SELECT 2, REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=2);
```

Inserto en TABLA_VENTAS la venta con IDVENTA 2 para el IDCLIENTE 1:

```
INSERT INTO TABLA_VENTAS
SELECT 2, REF(C), SYSDATE, TIP_LINEAS_VENTA()
FROM TABLA_CLIENTES C WHERE C.IDCLIENTE=1;
```

Inserto en TABLA_VENTAS tres líneas de venta para el IDVENTA 2 para los productos 1 (la CANTIDAD es 2), 4 (la CANTIDAD es 1) y 5 (la CANTIDAD es 4):

```
INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 1, REF(P), 2 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=1);

INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 2, REF(P), 1 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=4);

INSERT INTO TABLE (SELECT V.LINEAS FROM TABLA_VENTAS V WHERE V.IDVENTA=2)
(SELECT 3, REF(P), 4 FROM TABLA_PRODUCTOS P WHERE P.IDPRODUCTO=5);
```

El siguiente procedimiento almacenado visualiza los datos de la venta cuyo identificador recibe:

```
CREATE OR REPLACE PROCEDURE VER_VENTA (ID NUMBER) AS
  LIN NUMBER;
  CANT NUMBER;
  IMPORTE NUMBER;
  TOTAL_V NUMBER;
  PRODUC TIP_PRODUCTO:=TIP_PRODUCTO(NULL,NULL,NULL,NULL);
  CLI TIP_CLIENTE:=TIP_CLIENTE(NULL,NULL,NULL,NULL, NULL);
  DIR TIP_DIRECCION:=TIP_DIRECCION(NULL,NULL,NULL,NULL);
  FEC DATE;
  CURSOR C1 IS
    SELECT NUMEROLINEA, DEREF(IDPRODUCTO), CANTIDAD FROM THE
     (SELECT T.LINEAS FROM TABLA_VENTAS T WHERE IDVENTA=ID);
BEGIN
  SELECT DEREF(IDCLIENTE), FECHAVENTA, V.TOTAL_VENTA()
  INTO CLI, FEC, TOTAL_V
  FROM TABLA_VENTAS V WHERE IDVENTA=ID;
  DIR :=CLI.DIREC;
  DBMS_OUTPUT.PUT_LINE('NUMERO DE VENTA: '||ID|| ' * Fecha de venta: '|| FEC);
  DBMS_OUTPUT.PUT_LINE('CLIENTE: '||CLI.NOMBRE);
  DBMS_OUTPUT.PUT_LINE('DIRECCION: '||DIR.CALLE);
  DBMS_OUTPUT.PUT_LINE('==========================================');

  OPEN C1;
  FETCH C1 INTO LIN, PRODUC, CANT;
  WHILE C1%FOUND LOOP
    IMPORTE:= CANT * PRODUC.PVP;
    DBMS_OUTPUT.PUT_LINE(LIN||'*'||PRODUC.DESCRIPCION ||'*'||PRODUC.PVP||'*'||
                         CANT||'*'||IMPORTE);
    FETCH C1 INTO LIN, PRODUC, CANT;
  END LOOP;
```



```
  CLOSE C1;
  DBMS_OUTPUT.PUT_LINE('Total Venta: '||TOTAL_V);
END VER_VENTA;
/
```

Ejecutamos el procedimiento para visualizar los datos de la venta 2:

```
BEGIN
  VER_VENTA(2);
END;
/

NUMERO DE VENTA: 2 * Fecha de venta: 19/03/12
CLIENTE: Luis Gracia
DIRECCION: C/Las Flores 23
==========================================
1*CAJA DE CRISTAL DE MURANO*100*2*200
2*OPERACIONES CON BD*25*1*25
3*APLICACIONES WEB*25,5*4*102
Total Venta: 327
```

Actividad 5: Crea una función almacenada que reciba un identificador de venta y retorne el total de la venta. Realiza un bloque PL/SQL anónimo que haga uso de la función.

4.3 BASES DE DATOS ORIENTADAS A OBJETOS

No

Las Bases de Datos Orientadas a Objetos (BDOO) son aquellas cuyo modelo de datos está orientado a objetos, soportan el paradigma orientado a objetos almacenando métodos y datos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (POO) almacenando directamente los objetos en la BD y empleando las mismas estructuras y relaciones que los lenguajes de POO.

Podemos decir que un Sistema Gestor de Base de Datos Orientada a Objetos (SGBDOO) es un sistema gestor de base de datos (SGBD) que almacena objetos.

4.3.1 CARACTERÍSTICAS DE LAS BASES DE DATOS OO

Las características asociadas a las BDOO son las siguientes:

- Los datos se almacenan como **objetos.**
- Cada objeto se identifica mediante un identificador único u **OID** *(Object Identifier)*, este identificador no es modificable por el usuario.
- Cada objeto define sus métodos y atributos y la interfaz mediante la cual se puede acceder a él, el usuario puede especificar qué atributos y métodos pueden ser usuados desde fuera.

- En definitiva, un SGBDOO debe cumplir las características de un SGBD: **persistencia, concurrencia, recuperación ante fallos, gestión del almacenamiento secundario y facilidad de consultas;** y las características de un sistema orientado a objetos (OO): **encapsulación, identidad, herencia y polimorfismo.**

En 1989 se hizo el manifiesto *Malcolm Atkinson* que propone 13 características obligatorias para los SGBDOO basado en dos criterios: debe ser un sistema orientado a objetos y debe ser un SGBD. Las características son:

1. Debe soportar objetos complejos.
2. Identidad del objeto: todos los objetos deben tener un identificador que sea independiente de los valores de sus atributos.
3. Encapsulamiento: los programadores solo tendrán acceso a la interfaz de los métodos, de modo que sus datos e implementación estén ocultos.
4. Soporte para tipos o clases.
5. Herencia: un subtipo o una subclase heredará los atributos y métodos de su supertipo o superclase, respectivamente.
6. Debe soportar sobrecarga: los métodos deben poder aplicarse a diferentes tipos.
7. El DML debe ser completo.
8. El conjunto de tipos de datos debe ser extensible.
9. Debe soportar persistencia de datos: los datos deben mantenerse después de que la aplicación que los creó haya finalizado.
10. Debe ser capaz de manejar grandes BD: debe proporcionar mecanismos que aseguren independencia entre los niveles lógico y físico del sistema.
11. Debe soportar concurrencia.
12. Debe ser capaz de recuperarse ante fallos hardwarte y software.
13. Debe proporcionar un método de consulta sencillo.

The Object-Oriented Database System Manifesto:
http://www.cs.cmu.edu/afs/cs.cmu.edu/user/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

Las ventajas que aporta un SGBDOO son las siguientes:

- Mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan guardar.
- Extensibilidad. Se pueden construir nuevos tipos de datos a partir de tipos existentes.
- Existe una única interfaz entre el LMD (lenguaje de manipulación de datos) y el lenguaje de programación. Esto elimina el tener que incrustar un lenguaje declarativo como SQL en un lenguaje imperativo como Java o C.
- Lenguaje de consultas más expresivo. El lenguaje de consultas es navegacional de un objeto al siguiente, en contraste con el lenguaje declarativo SQL.
- Soporte a transacciones largas, necesario para muchas aplicaciones de bases de datos avanzadas.
- Adecuación a aplicaciones avanzadas de bases de datos (CASE, CAD, sistemas multimedia, etc.).

Entre los inconvenientes hay que destacar:

- Falta de un modelo de datos universal, la mayoría de los modelos carecen de una base teórica.
- Falta de experiencia, el uso de los SGBDOO es todavía relativamente limitado.
- Falta de estándares, no existe un lenguaje de consultas estándar como SQL, aunque está el lenguaje OQL (*Object Query Language*) de ODMG que se está convirtiendo en un estándar *de facto.*
- Competencia con los SGBDR y los SGBDOR, que tienen gran experiencia de uso.
- La optimización de consultas compromete la encapsulación: optimizar consultas requiere conocer la implementación para acceder a la BD de una manera eficiente.
- Complejidad: el incremento de funcionalidad provisto por un SGBDOO lo hace más complejo que un SGBDR. La complejidad conlleva productos más caros y difíciles de usar.
- Falta de soporte a las vistas: la mayoría de SGBDOO no proveen mecanismos de vistas.
- Falta de soporte a la seguridad.

4.3.2 EL ESTÁNDAR ODMG

ODMG (*Object Database Management Group*) es un grupo formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. Uno de sus estándares, el cual lleva el mismo nombre del grupo (ODMG) especifica los elementos que se definirán, y en qué manera se hará, para la consecución de persistencia en las BDOO que soporten el estándar.

La última versión del estándar, ODMG 3.0 propone los siguientes componentes:

- Modelo de objetos.
- Lenguaje de definición de objetos (ODL, *Object Definition Language*).
- Lenguaje de consulta de objetos (OQL, *Object Query Language*).
- Conexión con los lenguajes C++, Smalltalk y Java.

El modelo de objetos ODMG especifica las características de los objetos, cómo se relacionan, cómo se identifican, construcciones soportadas, etc. Las primitivas de modelado básicas son: los objetos, caracterizados por un identificador único (**OID**-*Object IDentifier*) y los literales que son objetos que no tienen identificador, no pueden aparecer como objetos, están embebidos en ellos.

Los tipos de objetos son:

- Atómicos: boolean, short, long, unsigned long, unsigned short, float, double, char, string, enum, octect.
- Tipos estructurados: date, time, timestamp, interval.
- Colecciones <*interfaceCollection*>:
  - set<tipo>: grupo desordenado de objetos del mismo tipo que no admite duplicados.
  - bag<tipo>: grupo desordenado de objetos del mismo tipo que permite duplicados.
  - list<tipo>: grupo ordenado de objetos del mismo tipo que permite duplicados.
  - array<tipo>: grupo ordenado de objetos del mismo tipo a los que se puede acceder por su posición. El tamaño es dinámico.
  - dictionary<clave,valor>: grupo de objetos del mismo tipo, cada valor está asociado a su clave.

Los **literales** pueden ser atómicos (long, short, boolean, unsigned long, etc.), colecciones (set, bag, list, array, dictionary), estructuras (date, interval, time, timestamp) y NULL.

Mediante las **Clases** especificamos el estado y el comportamiento de un tipo de objeto, pueden incluir métodos. Son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales. Son equivalentes a una clase concreta en los lenguajes de programación. Una clase es un tipo de objetos asociado a un "*extent*".

El lenguaje **ODL** es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. Define los atributos, las relaciones entre los tipos y especifica la signatura de las operaciones. La sintaxis de ODL extiende el lenguaje de definición de interfaces de CORBA (*Common Object Request Broker Architecture*). Algunas de las palabras reservadas para definir los objetos son:

- *class:* declaración del objeto, define el comportamiento y el estado de un tipo de objeto.
- *extent:* define la extensión, nombre para el actual conjunto de objetos de la clase. En las consultas se hace referencia al nombre definido en esta cláusula, no se hace referencia al nombre definido a la derecha de class.
- *key[s]:* declara la lista de claves para identificar las instancias.
- *attribute:* declara un atributo.
- *set | list | bag | array:* declara un tipo de colección.
- *struct:* declara un tipo estructurado.
- *enum:* declara un tipo enumerado.
- *relationship:* declara una relación.
- *inverse:* declara una relación inversa.
- *extends:* define la herencia simple.

Veamos como se puede definir un objeto Cliente, similar al tipo cliente visto anteriormente, la clave es el NIF. Se definen los atributos y un método; uno de los atributos es un tipo estructurado (*struct*), otro es enumerado (*enum*) y también tenemos un tipo colección (*set*):

```
class Cliente (extent Clientes key NIF)
{
 /*Definición de atributos*/
    attribute struct Nombre_Persona {
                     string apellidos,
                     string nombrepern} nombre;
    attribute string NIF;
    attribute date fecha_nac;
    attribute enum Genero{H,M} sexo;
    attribute struct Direccion{
                     string calle,
                     string poblac} direc;
    attribute set<string> telefonos;
 /*Definición de operaciones*/
    short edad();
}
```


Definimos el objeto Producto:

```
class Producto (extent Productos key IDPRODUCTO)
{
 /*Definición de atributos*/
    attribute short IDPRODUCTO;
    attribute string descripcion;
    attribute float pvp;
    attribute short stockactual;
}
```

Definimos el objeto línea de venta con datos de la línea y la operación para calcular el importe de la línea:

```
class LineaVenta (extent Lineaventas)
{
 /*Definición de atributos*/
    attribute short numerolinea;
    attribute Producto product;
    attribute short cantidad;
 /*Definición de operaciones*/
    float importe();
}
```

A continuación definimos el objeto Venta y sus relaciones: una venta pertence a un cliente (*pertenece_a_cliente*) y la inversa, el cliente tiene venta (*tiene_venta*), también se define un atributo colección (set) para las líneas de venta:

```
class Venta (extent Ventas key IDVENTA)
{
 /*Definición de atributos*/
    attribute short IDVENTA;
    attribute date fecha_venta;
    attribute set <LineaVenta> lineas;
 /*Definición de relaciones*/
    relationship Cliente pertenece_a_cliente inverse Cliente::tiene_venta;
 /*Definición de operaciones*/
    float total_venta();
}
```

4.3.3 EL LENGUAJE DE CONSULTAS OQL

OQL (*Object Query Language*) es el lenguaje estándar de consultas de BDOO. Las características son las siguientes:

- Es orientado a objetos y está basado en el modelo de objetos de la ODMG.
- Es un lenguaje declarativo del tipo de SQL. Su sintaxis básica es similar a SQL.
- Acceso declarativo a los objetos de la base de datos (propiedades y métodos).
- Semántica formal bien definida.

- No incluye operaciones de actualización (solo de consulta). Las modificaciones se realizan mediante los métodos que los objetos poseen.
- Dispone de operadores sobre colecciones (*max, min, count, etc.*) y cuantificadores (*for all, exists*).

La sintaxis básica de OQL es una estructura SELECT como en SQL:

```
SELECT <lista de valores>
FROM <lista de colecciones y miembros típicos>
[WHERE <condición>]
```

Donde las colecciones en FROM pueden ser extensiones (los nombres que aparecen a la derecha de *extent*) o expresiones que evalúan una colección. Se suele utilizar una variable iterador que vaya tomando valores de los objetos de la colección. Las variables iterador se pueden especificar de varias formas utilizando las cláusulas IN o AS:

```
FROM Clientes x
FROM x IN Clientes
FROM Clientes AS x
```

Para acceder a los atributos y objetos relacionados se utilizan expresiones de camino. Una expresión de camino empieza normalmente con un nombre de objeto o una variable iterador seguida de atributos conectados mediante un punto o nombres de relaciones. Por ejemplo, para obtener el nombre de los clientes que son mujeres, podemos escribir:

```
SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"
SELECT x.nombre.nombreper FROM Clientes x WHERE x.sexo="M"
SELECT x.nombre.nombreper FROM Clientes AS x WHERE x.sexo="M"
```

En general, supongamos que v es una variable cuyo tipo es Venta:

- *v.IDVENTA* es el identificador de venta del objeto v.
- *v.fecha_venta* es la fecha de venta del objeto v.
- *v.total_venta()* obtiene el total venta del objeto v.
- *v.pertenece_a_cliente* es un puntero al cliente mencionado en v.
- *v.pertenece_a_cliente.direc* es la dirección del cliente mencionado en v.
- *v.lineas* es una colección de objetos del tipo LineaVenta.
- El uso de *v.lineas.numerolinea* NO es correcto porque *v.lineas* es una colección de objetos y no un objeto simple.
- Cuando tenemos una colección como en *v.lineas*, para poder acceder a los atributos de la colección podemos usar la orden FROM.

Ejemplos:

Obtener los datos del cliente cuyo IDVENTA = 1:

```
SELECT v.pertenece_a_cliente.nombre, v.pertenece_a_cliente.direc,
       v.fecha_venta, v.total_venta()
FROM Ventas v
WHERE v.IDVENTA=1;
```


Obtenemos las líneas de venta del IDVENTA = 1, en este ejemplo el objeto *v* es usado para definir la segunda colección *v.lineas*:

```
SELECT lin.numerolinea, lin.product.descripcion, lin.cantidad, lin.importe()
FROM Ventas v, v.lineas lin
WHERE v.IDVENTA=1;
```

El resultado de una consulta OQL puede ser de cualquier tipo soportado por el modelo. Por ejemplo, la consulta anterior devuelve un conjunto de estructuras del tipo short, string, short y float; el resultado es del tipo colección: *bag (struct(numerolinea:short, descripcion:string, cantidad: short, importe: float))*.

En cambio la consulta: *SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"*, devuelve un conjunto de nombres; el tipo devuelto es: *bag(string)*.

Recordemos la diferencia entre las colecciones *set* y *bag*, *set* es un grupo desordenado de objetos del mismo tipo que no permite duplicados y *bag* permite duplicados.

Se pueden usar alias en las consultas, por ejemplo la SELECT anterior:

```
SELECT lin.numerolinea, lin.product.descripcion, lin.cantidad, lin.importe()
```

Se puede expresar usando alias de la siguiente manera:

```
SELECT nlin:lin.numerolinea,
       dpro:lin.product.descripcion,
       can:lin.cantidad,
       imp:lin.importe()
```

Y el tipo devuelto en este caso es: *bag (struct(nlin:short, dpro:string, can: short, imp: float))*.

Para obtener un set de estructuras (colección que no admite duplicados) podemos usar DISTINCT a la derecha de SELECT:

```
SELECT DISTINCT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"
```

En este caso el tipo devuelto es: *set(string)*.

Para obtener una lista (un tipo list) de estructuras usamos la cláusula ORDER BY:

```
SELECT nlin:lin.numerolinea,
       dpro:lin.product.descripcion,
       can:lin.cantidad,
       imp:lin.importe()
FROM Ventas v, v.lineas lin
WHERE v.IDVENTA=1
ORDER BY nlin ASC;
```

Y el tipo devuelto en este caso es: *list(struct(nlin:short, dpro:string, can: short, imp: float))*.

4.3.3.1 OPERADORES DE COMPARACIÓN

Lo valores pueden ser comparados usando los siguientes operadores:

> Mayor que
>=Mayor o igual que
< Menor que
<=Menor o igual que
!= Distinto de

Para comparar cadenas de caracteres podemos usar los operadores IN y LIKE:

- IN: comprueba si existe un carácter en una cadena de caracteres: *carácter IN cadena.*
- LIKE: comprueba si dos cadenas son iguales: *cadena1 LIKE cadena2. cadena2* puede contener caracteres especiales:
  _ o ?: indicador de posición, representa cualquier carácter.
  * o %: representa una cadena de caracteres.

Ejemplo: Obtener los datos de las ventas de los clientes de la población de TOLEDO y cuyos apellidos empiecen por la letra A:

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta()
FROM Ventas v
WHERE v.pertenece_a_cliente.direc.pobla="TOLEDO"
AND   v.pertenece_a_cliente.nombre.apellidos LIKE "A%";
```

Obtener para el IDVENTA 1 aquellas líneas de venta cuya descripción del producto contenga el carácter P en su descripción:

```
SELECT lin.numerolinea, lin.product.descripcion, lin.cantidad , lin.importe()
FROM Ventas v, v.lineas lin
WHERE v.IDVENTA=1 AND 'P' IN lin.product.descripcion;
```

4.3.3.2 CUANTIFICADORES Y OPERADORES SOBRE COLECCIONES

Mediante el uso de cuantificadores podemos comprobar si todos los miembros, al menos un miembro, o algunos miembros, etc., satisfacen alguna condición:

Todos los miembros:	*FOR ALL x IN colección : condición*
Al menos uno:	*EXISTS x IN colección : condición*
	EXISTS x
Solo uno:	*UNIQUE x*
Algunos / cualquier:	*colección comparación SOME/ANY condición*
	Donde *comparación* puede ser : <, >, <=, >=, o =

Ejemplos:

Obtener todas las ventas que tengan líneas de venta cuya descripción del producto sea "PNY Pendrive 16 GB":

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta()
FROM Ventas v
WHERE EXISTS x IN v.lineas : x.product.descripcion="PNY Pendrive 16 GB";
```

Obtener las ventas que solo tienen líneas de venta cuya descripción de producto es "PNY Pendrive 16 GB":

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta()
FROM Ventas v
WHERE FOR ALL x IN v.lineas : x.product.descripcion="PNY Pendrive 16 GB";
```

Los operadores AVG, SUM, MIN, MAX y COUNT, se pueden aplicar a cualquier colección, siempre y cuando los tengan sentido para el tipo de elemento. Por ejemplo, para calcular la media del total venta de todas las ventas necesitaríamos asignar el valor devuelto a una variable:

```
Media = AVG(SELECT v.total_venta() FROM Ventas v)
```

El tipo devuelto es una colección de un elemento: *bag(struct(total_venta: float)).*

Como hemos visto OQL es bastante complejo y actualmente ningún creador de software lo ha implementado completamente. OQL ha influenciado el diseño de algunos lenguajes de consulta nuevos como JDOQL (*Java Data Objects Query Language*) EJBQL (*Enterprise Java Bean Query Language*), pero estos no pueden ser considerados como versiones de OQL.

4.4 EJEMPLO DE BDOO

En el capítulo 3 se presentó la base de datos orientada a objetos *DB4o*, a continuación se presenta una base de datos sencilla de utilizar que aporta una API simple que no requiere el aprendizaje de técnicas de mapeo, se trata de *NeoDatis Object Database. NeoDatis ODB* es una base de datos orientada a objetos de código abierto que funciona con Java, .Net, Groovy y Android. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos a tablas.

Desde la web http://neodatis.wikidot.com/downloads podemos descargar la última versión. Para los ejemplos se ha descargado el fichero *neodatis-odb-1.9.30.689.zip*. Lo descomprimimos y copiamos el fichero *neodatis-odb-1.9.30.689.jar* en la carpeta adecuada para después definirlo en el CLASSPATH (por ejemplo *SET CLASSPATH=.;D:\uni4\neodatis-odb-1.9.30.689.jar*) o incluirlo en nuestro proyecto Eclipse. Desde la carpeta */doc/javadoc* podemos acceder a la documentación de la API de *Neodatis ODB.*

El ejemplo que se presenta a continuación almacena objetos *Jugadores* en la base de datos de nombre *neodatis.test*. Para abrir la base de datos se usa la clase **ODBFactory** con el método *open()* que devuelve un ODB (es la interfaz principal, lo que el usuario ve):

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
```

Para almacenar los objetos se usa el método *store()*:

```
Jugadores j4 = new Jugadores("Alicia","tenis","Madrid",14);
odb.store(j4);
```

Una vez almacenados los objetos, para recuperarlos usamos el método *getObjects()*, que recibe la clase cuyos objetos se van a recuperar y devuelve una colección de objetos de esa clase:

```
Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
```

Para validar los cambios en la base de datos se usa el método *close()*. El código completo del ejemplo (*EjemploNeodatis.java*) donde se ha incluido la clase *Jugadores* es el siguiente:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

//Clase Jugadores
class Jugadores {
    private String nombre;
    private String deporte;
    private String ciudad;
    private int edad;

    public Jugadores() {}

    public Jugadores(String nombre, String deporte, String ciudad, int edad) {
            this.nombre = nombre;
            this.deporte = deporte;
            this.ciudad = ciudad;
            this.edad = edad;
    }
    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getNombre() {return nombre;}
    public void setDeporte(String deporte) {this.deporte = deporte;}
    public String getDeporte() {return deporte;}
    public void setCiudad(String ciudad) {this.ciudad = ciudad;}
    public String getCiudad () {return ciudad;}
    public void setEdad(int edad) {this.edad = edad;}
    public int getEdad() {return edad;}
}
//
public class EjemploNeodatis {
    public static void main(String[] args) {
      // Crear instancias para almacenar en BD
      Jugadores j1 = new Jugadores("Maria","voleibol", "Madrid",14);
      Jugadores j2 = new Jugadores("Miguel","tenis", "Madrid",15);
      Jugadores j3 = new Jugadores("Mario", "baloncesto","Guadalajara",15);
      Jugadores j4 = new Jugadores("Alicia","tenis","Madrid",14);

      ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
```



```
      // Almacenamos objetos
      odb.store(j1);
      odb.store(j2);
      odb.store(j3);
      odb.store(j4);

      //recuperamos todos los objetos
      Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
      System.out.println(objects.size() + " Jugadores:");

      int i = 1;
      // visualizar los objetos
      while(objects.hasNext()){
         Jugadores jug = objects.next();
         System.out.println((i++) + "\t: " + jug.getNombre() + "*" +
              jug.getDeporte()+ "*" + jug.getCiudad()+ "*" + jug.getEdad());
       }
      odb.close(); // Cerrar BD
    }
}
```

Actividad 6: Crea un nuevo proyecto en Eclipse y añade el JAR para trabajar con Neodatis. Dentro del proyecto crea un paquete de nombre *clases*. Crea la clase Paises con dos atributos y sus *getter* y *setter*. Los atributos son:

```
private int id;
private String nombrepais;
```

Añade también el método *toString()* para que devuelva el nombre del país:

```
public String toString() {return nombrepais;}
```

Crea la clase Jugadores (en el paquete *clases*, como en el ejemplo anterior) y añade el siguiente atributo con sus *getter* y *setter*:

```
private Paises pais;
```

Crea una clase Java (con el método main) que cree una base de datos de nombre EQUIPOS. DB e inserta países y los jugadores de esos países.

Añade otra clase Java para visualizar los países y los jugadores que hay en la BD.

NeoDatis dispone de un explorador que nos permite navegar por los objetos. Para ejecutarlo hacemos doble clic en el fichero *odb-explorer.bat* (sistemas Windows) u *odb-explorer.sh* (sistemas Linux). Es necesario abrir la base de datos por la que vamos a navegar, pulsamos en el menú *NeoDatis ODB->Open Database* (Figura 4.2).

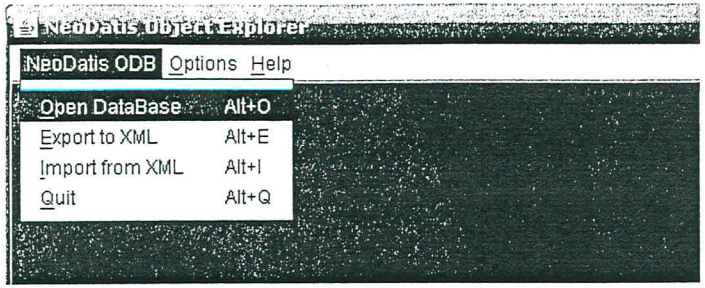


Figura 4.2. NeoDatis Open DataBase.

Se abre una nueva ventana con 2 pestañas, nos quedamos en la primera (*Local connections*), localizamos en nuestro disco el fichero *neodatis.test* y pulsamos el botón *Connect*. Se abre el explorador, al pulsar con el botón derecho del ratón sobre la clase *Jugadores* podemos acceder a la vista de los objetos, vista en formato de tabla, realizar consultas, crear un nuevo objeto, etc.; véase Figura 4.3. Desde el explorador se pueden modificar los objetos, eliminarlos, etc. después de realizar cada operación hemos de pulsar el botón *Commit* para validar los cambios. Para finalizar con la base de datos pulsamos el botón *Close Database*.

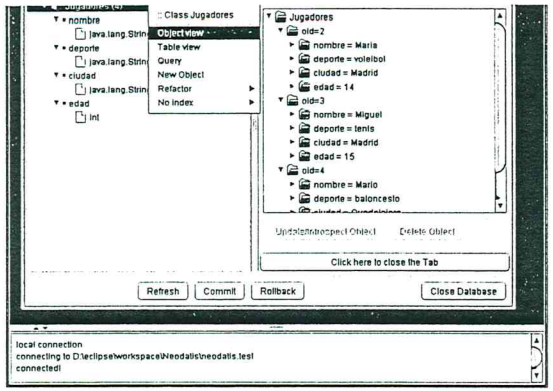


Figura 4.3. Explorador de Objetos de NeoDatis.

Podemos acceder a los objetos conociendo su **OID** (identificador de objeto). El siguiente ejemplo muestra los datos del objeto cuyo **OID** es el 3:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.OID;
import org.neodatis.odb.core.oid.OIDFactory;
```



```
public class ejemploOid {
  public static void main(String[] args) {
      ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
      OID oid = OIDFactory.buildObjectOID(3); //obtener objeto con OID 3

      Jugadores jug  = (Jugadores) odb.getObjectFromId(oid);
      System.out.println( jug.getNombre() + "*" +  jug.getDeporte()+ "*" +
              jug.getCiudad()+ "*" + jug.getEdad());
      odb.close(); // Cerrar BD
  }
}
```

El **OID** de un objeto es devuelto por los métodos store(Objeto) y getObjectId(Objeto), se necesita importar el paquete *import org.neodatis.odb.OID*. Ejemplo, para obtener el **OID** del objeto j1 podemos hacerlo de las siguientes maneras:

```
OID oid1=odb.store(j1);
OID oid1=odb.getObjectId(j1) ;
```

4.4.1 CONSULTAS SENCILLAS

Para realizar consultas usamos la clase **CriteriaQuery** en donde especificaremos la clase y el criterio para realizar la consulta. Necesitaremos importar los siguientes paquetes:

```
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
```

El siguiente ejemplo obtiene todos los jugadores que practican el deporte tenis:

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("deporte", "tenis"));
query.orderByAsc("nombre,edad"); //ordena ascendentemente por nombre y edad

Objects<Jugadores> objects = odb.getObjects(query); //obtiene todos los jugadores
```

Para obtener el primer objeto usamos el método *getFirst()*:

```
Jugadores j = (Jugadores) odb.getObjects(query).getFirst();//obtiene solo el 1°
```

Como se puede ver el método *CriteriaQuery()* utiliza *Where.equal* para seleccionar los objetos que cumplan la condición especificada. Para ordenar la salida ascendentemente se usa el método *orderByAsc()*, entre paréntesis se ponen los atributos por los que se realiza la ordenación; para ordenar descendentemente se usa *orderByDesc()*.

Actividad 7: Añade al proyecto Eclipse creado anteriormente otra clase Java para realizar la consulta anterior, pero por cada jugador visualiza también su país.

Para modificar un objeto, primero es necesario cargarlo, después lo modificamos usando los métodos set del objeto y a continuación lo actualizamos con el método *store()*. No olvidemos que para validar los cambios es necesario cerrar la BD. El siguiente ejemplo cambia el deporte de Maria a vóley-playa:

```
query = new CriteriaQuery(Jugadores.class,Where.equal("nombre", "Maria"));
Objects<Jugadores> objetos = odb.getObjects(query);
// Obtiene solo el primer objeto encontrado
Jugadores jug = (Jugadores) objetos.getFirst();
jug.setDeporte("vóley-playa"); // Cambia el deporte
odb.store(jug); // actualiza el objeto
odb.close();    // Valida los cambios
```

Para eliminar un objeto, primero lo localizamos como antes y luego usamos el método *delete()*:

```
odb.delete(jug); // elimina el objeto
```

Con **CriteriaQuery** se puede usar la interfaz **ICriterion** para construir el criterio de la consulta, para usarlo será necesario importar otros paquetes:

```
import org.neodatis.odb.core.query.criteria.ICriterion;
```

Por ejemplo, para obtener los jugadores cuya edad es 14 utilizamos el criterio *Where.equal()*:

```
ICriterion criterio = Where.equal("edad", 14);
CriteriaQuery query = new CriteriaQuery(Jugadores.class, criterio);
Objects<Jugadores> objects = odb.getObjects(query);
```

Para obtener los jugadores cuyo nombre empieza por la letra M usamos *Where.like()*:

```
ICriterion criterio = Where.like("nombre","M%");
```

Para obtener los jugadores cuya edad es > que 14 usamos *Where.gt()*:

```
ICriterion criterio = Where.gt("edad",14);
```

Para mayor o igual que 14 usamos: `Where.ge("edad",14)`.
Para menor que 14 usamos: `Where.lt("edad",14)`.
Para menor o igual que 14 usamos: `Where.le("edad",14)`.

Para comprobar si un array o una colección contiene un valor determinado usamos *Where.contain()*:

```
ICriterion criterio = Where.contain("nombrearray",valor);
```

Para comprobar si un atributo es nulo usamos *Where.isNull()*:

```
ICriterion criterio = Where.isNull("atributo");
```


Para comprobar si no es nulo usamos *Where.isNotNull()*:

```
ICriterion criterio = Where.isNotNull("atributo");
```

La construcción de expresiones lógicas añade complejidad al criterio de consulta. Es necesario importar los paquetes:

```
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Not;
```

Y añadir mediante el método *add()* a los criterios de búsqueda. Por ejemplo, para obtener los jugadores de Madrid y edad 15 escribimos el siguiente criterio AND:

```
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid"))
                               .add(Where.equal("edad", 15));
```

Para obtener los jugadores cuya ciudad sea Madrid o la edad sea >= que 15 construimos el criterio OR:

```
ICriterion criterio = new Or().add(Where.equal("ciudad", "Madrid"))
                              .add(Where.ge("edad", 15));
```

Para obtener los jugadores cuyo nombre no empiece por la letra M usamos *Where.not()*:

```
ICriterion criterio = Where.not(Where.like("nombre","M%"));
```

O también se puede poner:

```
ICriterion criterio1 = Where.like("nombre","M%");
ICriterion criterio = Where.not(criterio1);
```

4.4.2 CONSULTAS MÁS COMPLEJAS

Para utilizar agrupamientos GROUP BY y las funciones de grupo SUM, MAX, MIN, AVG y COUNT usamos la API *Object Values* de Neodatis que provee acceso a los atributos de los objetos. También se puede usar para recuperar objetos, por ejemplo, para obtener el nombre y la ciudad de todos los jugadores usando esta API escribimos el siguiente código:

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD
Values valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
                    .field("nombre").field("ciudad"));
while(valores.hasNext()){
          ObjectValues objectValues= (ObjectValues) valores.next();
          System.out.println(
                 objectValues.getByAlias("nombre")
                 + " ciudad " + objectValues.getByIndex(1));
}
```

Mediante los métodos *getByAlias("alias")* o *getByIndex(índice)* podemos obtener los valores de los atributos de un objeto. Se puede definir un alias a cada atributo de la clase. En el ejemplo anterior en el método *field()* no se especificó alias, entonces para recuperar el valor del atributo se indica su nombre en el método *getByAlias()*. Si definimos el atributo *nombre* con alias: *field("nombre", "n")*; entonces a la hora de recuperar el valor del atributo usamos *getByAlias("n")*.

Para trabajar con la API *Object Values* será necesario importar algunos paquetes:

```
import java.math.BigDecimal;
import java.math.BigInteger;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Values;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;
```

Veamos algunos ejemplos de uso de las funciones de grupo, se pone comentada la sentencia SQL a la que equivaldrían las líneas de código expuestas:

```
ODB odb = ODBFactory.open("neodatis.test");// Abrir BD

//SUMA - Obtiene la suma de las edades
//SELECT SUM(edad) FROM Jugadores
Values  val = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).sum("edad"));
ObjectValues ov = val.nextValues();
BigDecimal value = (BigDecimal) ov.getByAlias("edad");
System.out.println("Suma de edad : "+ value.longValue());

//CUENTA - Obtiene el número de jugadores
//SELECT COUNT(nombre) FROM Jugadores
Values  val2 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
                        .count("nombre"));
ObjectValues ov2 = val2.nextValues();
BigInteger value2 = (BigInteger) ov2.getByAlias("nombre");
System.out.println("Numero de jugadores : "+ value2.intValue());

//MEDIA - Obtiene la edad media de los jugadores
//SELECT AVG(edad) FROM Jugadores
Values  val3 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class).avg("edad"));
ObjectValues ov3 = val3.nextValues();
BigDecimal value3 = (BigDecimal) ov3.getByAlias("edad");
System.out.println("Edad media : "+ value3.floatValue());

//MAXIMO Y MINIMO - Obtiene la edad máxima y la edad mínima
//SELECT MAX(edad) edad_max , MIN(edad) edad_min FROM Jugadores
Values  val4 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
                        .max("edad","edad_max")
                        .min("edad","edad_min"));
ObjectValues ov4 = val4.nextValues();
BigDecimal maxima = (BigDecimal) ov4.getByAlias("edad_max");
BigDecimal minima = (BigDecimal) ov4.getByAlias("edad_min");

System.out.println("Edad maxima: "+ maxima.intValue()+
                   "  Edad minima: "+ minima.intValue());
```


En este ejemplo usamos GROUP BY, obtenemos por cada ciudad el número de jugadores:

```
Values  groupby = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
        .field("ciudad").count("nombre").groupBy("ciudad"));

while(groupby.hasNext()){
      ObjectValues objetos= (ObjectValues) groupby.next();
      System.out.println(objetos.getByAlias("ciudad") +
                         " * "+objetos.getByIndex(1));
}
```

En SQL la consulta sería: *SELECT ciudad, count(nombre) FROM Jugadores GROUP BY ciudad.*

Cuando un objeto está relacionado con otro, por ejemplo, un jugador está relacionado con su país, podremos acceder a la información del objeto relacionado para obtener la información necesaria gracias a la API *Object Values*. A esta característica se le da el nombre de **vistas dinámicas**, y son como los joins en SQL. Por ejemplo, partimos de las clases Paises y Jugadores (creadas en la Actividad 6), vamos a obtener el nombre, edad y país del jugador; en SQL sería como realizar la siguiente consulta: *SELECT nombre, edad, nombrepais FROM Jugadores j, Paises p WHERE j.id = p.id;*

Veamos como sería la consulta:

```
Values  valores = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
          .field("nombre")
          .field("edad")
          .field("pais.nombrepais"));

while (valores.hasNext()) {
      ObjectValues objectValues = (ObjectValues) valores.next();
      System.out.println("Nombre: "+ objectValues.getByAlias("nombre") +
                  " Edad: " + objectValues.getByIndex(1) +
                  " Pais: " + objectValues.getByIndex(2));
 }
```

Para obtener el nombre del país en el método *field()* escribimos la relación completa, el atributo de la clase Jugadores, un punto y a continuación el atributo de la clase Paises: *field("pais.nombrepais")*.

Para obtener el nombre y la edad de los jugadores cuyo nombre de pais es ESPAÑA escribimos:

```
Values valores = odb.getValues(new ValuesCriteriaQuery(
            Jugadores.class, Where.equal("pais.nombrepais","ESPAÑA")
          )
          .field("nombre")
          .field("ciudad"));
```

Para obtener el nombre y la ciudad de los jugadores cuyo pais es ESPAÑA escribimos:

```
Values valores = odb.getValues(new ValuesCriteriaQuery(
                  Jugadores.class,
                  new And().add(Where.equal("pais.nombrepais","ESPAÑA"))
```

```
            .add(Where.equal("edad",15))
          )
      .field("nombre")
      .field("ciudad"));
```

Actividad 8: Realiza una consulta para obtener el número de jugadores de España y edad = 15.

4.4.3 MODO CLIENTE/SERVIDOR DE LA BD

NeoDatis ODB también puede ser utilizada como una base de datos cliente/servidor donde el cliente y el servidor se pueden ejecutar en la misma máquina virtual o en una máquina virtual diferente. Lo primero que hemos de hacer es iniciar el servidor y configurar los siguientes parámetros:

- El puerto donde se va a ejecutar y recibir las conexiones de clientes, hemos de asegurarnos que esté libre. Por ejemplo, para crear el servidor en el puerto 8000 escribimos:

```
ODBServer server = ODBFactory.openServer(8000);
```

- La base de datos que debe ser manejada por el servidor. Esto se hace mediante el método *addBase()* en el que se especifica el nombre y el fichero de base de datos; este nombre será utilizado por los clientes. Un servidor puede "servir" más de una base de datos. Para indicar que trabajaremos con el fichero de base de datos *neodatist.test* (supongamos que está en la carpeta *D:/uni4/server)* y que los clientes usarán el nombre *base* para dirigirse a ella escribimos:

```
server.addBase("base", "D:/uni4/server/neodatis.test");
```

- El servidor se inicia como un subproceso en segundo plano usando el método *StartServer (true)*. Para iniciar el servidor escribimos:

```
server.startServer(true);
```

A continuación se debe crear el cliente. Hay dos formas; si el cliente se ejecuta en la misma máquina virtual que el servidor se puede crear una instancia del servidor de esta manera:

```
ODB odb = server.openClient("base");
```

Si el cliente se ejecuta en otra máquina virtual, se debe utilizar *ODBFactory:*

```
ODB odb = ODBFactory.openClient("localhost",8000,"base");
```

Para usar el modo cliente/servidor es necesario añadir el paquete *org.neodatis.odb.ODBServer.* Hay que tener en cuenta que las clases deben implementar *Serializable.* El siguiente ejemplo muestra el uso del modo cliente/servidor en la misma máquina virtual. La clase *Jugadores* que en ejemplos anteriores

estaba dentro del fichero JAVA que contiene la función main se ha incluido en un fichero aparte (debe implementar *Serializable*). Cuando se trabaja con *NeoDatis ODB,* es importante llamar al método *close()* para confirmar los cambios. Para estar seguros de hacer esto, es una buena práctica utilizar un bloque **try - finally:**

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.ODBServer;

public class EjemploNeodatisCliente {
  public static void main(String[] args) {
     ODB odb = null;
     ODBServer server = null;
     try {
      // Crea el servidor en el Puerto 80
      server = ODBFactory.openServer(8000);
      // BD a usar y el nombre que se usará para refererirse a ella
      server.addBase("base", "D:/uni4/server/neodatis.test");
      // Se ejecuta el servidor
      server.startServer(true);

      // Se abre la BD
      odb = server.openClient("base");

      // Se llama al método que visualize los datos
      VisualizaDatos(odb);

     } finally {
        if (odb != null) {
          // Primero se cierra el cliente
          odb.close();
        }
        if (server != null) {
          // Y luego el servidor
          server.close();
        }
     }
  }// fin main
//
  static void VisualizaDatos(ODB odb){
      //recuperamos todos los objetos
      Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
      System.out.println(objects.size() + " Jugadores:");

      int i = 1;
      // visualizar los objetos
      while(objects.hasNext()){
        Jugadores jug = objects.next();
          System.out.println((i++) + "\t: " + jug.getNombre() + "*" +
               jug.getDeporte()+ "*" + jug.getCiudad()+ "*" + jug.getEdad());
      }
  }
}// fin clase
```

A continuación se muestra el resultado de la ejecución, antes hemos de añadir el fichero JAR (*neodatis-odb-1.9.30.689.jar*) al CLASSPATH y compilar las clases **Jugadores.java** y **EjemploNeodatisCliente.java**:

```
D:\uni4\server>set CLASSPATH=.;D:\uni4\server\neodatis-odb-1.9.30.689.jar

D:\uni4\server>javac Jugadores.java

D:\uni4\server>javac EjemploNeodatisCliente.java

D:\uni4\server>java EjemploNeodatisCliente
NeoDatis ODB Server [version=1.9.30 - build=689-10-11-2010-08-21-21] running on
port 8000
Managed bases: [base]
4 Jugadores:
1       : Maria*voleibol*Madrid*14
2       : Miguel*tenis*Madrid*15
3       : Mario*baloncesto*Guadalajara*15
4       : Alicia*tenis*Madrid*14
NeoDatis ODB Server (port 8000) shutdown [uptime=0Hours]

D:\uni4\server>
```

El siguiente ejemplo muestra el modo cliente/servidor ejecutándose el servidor y el cliente en distintas máquinas virtuales. Por un lado tenemos el proceso servidor que se ejecutará en segundo plano:

```
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.ODBServer;

public class EjemploServer{
  public static void main(String[] args) {
    ODBServer server = null;
      server = ODBFactory.openServer(8000); //Crea el servidor en el puerto 8000
      server.addBase("base1", "D:/uni4/server/neodatis.test");// Abrir BD
      // Se inicia el servidor ejecutándose en segundo plano
      server.startServer(true);
   }
}
```

Lo compilamos y ejecutamos desde la línea de comandos del DOS, permanece ejecutándose hasta que cerremos la ventana del DOS o paremos el proceso (pulsando las teclas [Ctrl+C]):

```
D:\uni4\server>javac EjemploServer.java

D:\uni4\server>java EjemploServer
NeoDatis ODB Server [version=1.9.30 - build=689-10-11-2010-08-21-21] running on
port 8000
Managed bases: [base1]
```

El siguiente código Java muestra el cliente que se ejecutará en otra máquina virtual, se insertará un nuevo jugador en la base de datos. Se debe abrir una nueva consola DOS para ejecutarlo (no olvide añadir el JAR al CLASSPATH):


```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

public class EjemploCliente {
  public static void main(String[] args) {
     ODB odb = null;
     try {
       odb = ODBFactory.openClient("localhost",8000,"base1");
     Jugadores j4 = new Jugadores("Andrea","padel","Guadalajara",14);
     odb.store(j4);
    } finally {
       if (odb != null) {odb.close();}
    }
  }//--main
}
```

Lo compilamos y ejecutamos desde la línea de comandos del DOS:

```
d:\uni4\server>javac EjemploCliente.java

d:\uni4\server>java EjemploCliente

d:\uni4\server>
```

4.5 EJERCICIOS

❶ Responde a las siguientes cuestiones sobre BD objeto-relacional:

A) La orden CREATE TYPE:

a) Permite definir distintos tipos de tablas.

b) Permite definir tipos de objetos.

c) Permite definir restricciones a los tipos de datos.

d) Ninguna es correcta.

B) Los modelos de datos relacionales orientados a objetos:

a) Extienden el modelo de datos relacional proporcionando tipos de datos complejos y la programación orientada a objetos.

b) No permiten el uso de SQL como lenguaje de consulta ya que hay tipos complejos de datos.

c) Respuestas a) y b) correctas.

d) Ninguna de las anteriores.

C) ¿Cuál de estas afirmaciones es correcta?

a) Los lenguajes de consulta relacionales como SQL necesitan ser extendidos para trabajar con tipos de datos orientados a objetos.

b) En las BBDD relacionales orientadas a objetos se encuentran extensiones orientadas a objetos, como los tipos de datos definidos por el usuario.

c) En las BBDD relacionales orientadas a objetos no se encuentran extensiones sobre las tablas anidadas.

d) Respuestas a) y b) correctas.

❷) Realiza las siguientes operaciones en Oracle:

- Crea un tipo de objeto denominado SITUACION. Este tipo contiene las columnas NUMEROEDIFICIO de tipo NUMBER(4) y CIUDAD de tipo VARCHAR2(30).
- Crea un VARRAY de nombre EMAIL para cinco elementos.
- Crea un tipo de nombre DATOSEMPLE con las siguientes columnas: IDENTIFICADOR de tipo NUMBER(4), NOMBRE de tipo VARCHAR2(30), DIRECCION de tipo VARCHAR2(40), OFICINA de tipo SITUACION, CORREOS de tipo EMAIL. Crea una tabla de nombre UNIDAD4 con dos columnas: CLAVE, de tipo NUMBER(3) clave primaria y DATOS del tipo DATOSEMPLE definido anteriormente.
- ¿Cuál de las siguientes sentencias para insertar una fila es correcta?:

```
a)  INSERT INTO UNIDAD4 VALUES(22,DATOSEMPLE (1,'MANUEL', 'PILON 10',
        SITUACION(1,'MADRID'), EMAIL('a@uno.es','b@uno.es' )));
b)  INSERT INTO UNIDAD4 VALUES(22,DATOS( 1,'MANUEL', 'PILON
    10',SITUACION(1,'MADRID'), CORREOS('a@uno.es','b@uno.es' )));
c)  INSERT INTO UNIDAD4 (CLAVE,DATOS) VALUES(22,DATOSEMPLE (1,'MANUEL',
       'PILON 10',SITUACION(1,'MADRID'),EMAIL('a@uno.es','b@uno.es' )));
```

❸ ¿Cuál de las siguientes afirmaciones sobre ODMG es correcta?

a) ODMG es un grupo formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. La última versión del estándar del mismo nombre propone 3 componentes: el modelo de objetos, el lenguaje ODL y el lenguaje OQL.

b) El lenguaje ODL es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. Define los atributos, las relaciones entre los tipos y especifica la signatura de las operaciones.

c) El lenguaje OQL no incluye operaciones de actualización, solo son de consulta. Las actualizaciones se realizan mediante los métodos que los objetos poseen.

d) OQL no dispone de los operadores sobre colecciones para obtener el valor máximo, el mínimo, la cuenta... en su lugar se utilizan los métodos definidos en el objeto.

❹ Construye a partir del modelo relacional de la Figura 4.4 el modelo de objetos. Se dispone de 4 tablas: TALUMNOS con información de alumnos, TCURSOS con información de cursos, TASIGNATURAS con información de asignaturas y TNOTAS con información de notas en cada asignatura.

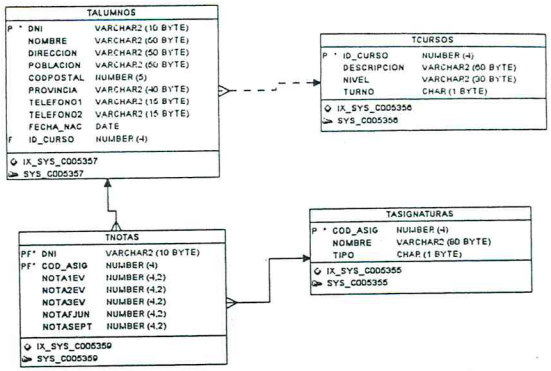


Figura 4.4. Modelo de datos ejercicio 4.

En la tabla TNOTAS tenemos las notas que tiene el alumno en cada asignatura. Existen tantas filas como asignaturas tenga el alumno. Las columnas NOTA1EV, NOTA2EV, ... corresponden a la nota de la 1ª, 2ª y 3ª evaluación, la final y la nota de septiembre (puede no tener nota en septiembre si lo aprueba todo).

Crea en Oracle los tipos necesarios. Por ejemplo: un tipo array para guardar las 5 notas, otro tipo array para los teléfonos, un tipo para almacenar la dirección, un tipo tabla anidada que contenga la asignatura y la nota del alumno. Define también un tipo alumno, este deberá tener la tabla anidada con las notas, un tipo para cursos y un tipo asignaturas.

En el nuevo modelo de objetos se deberán crear sólo 3 tablas, una para alumnos, otra para asignaturas y otra para cursos.

Realiza después un procedimiento almacenado que reciba un DNI de alumno y visualice el nombre, la dirección y las notas obtenidas en cada asignatura y en cada evaluación. Prueba el procedimiento.

❺ Partimos de las clases Departamentos y Empleados, similares a las definidas en unidades anteriores. Los atributos son los siguientes:

Departamentos:

```
private int deptNo;
private String dnombre;
private String loc;
```

Empleados:

```
private int empNo;
private String apellido;
private String oficio;
private Empleados dir;
private java.sql.Date fechaAlt;
private float salario;
private float comision;
private Departamentos dept;
```

Define los métodos *getter* y *setter* para cada atributo y el método *toString()* para que devuelva el nombre del departamento en la clase Departamentos y el nombre del empleado en la clase Empleados. Definir también los constructores necesarios. Todos los empleados deben tener un director. El empleado de mayor rango (el presidente) tiene como director a él mismo.

Crea una base de datos en Neodatis para guardar datos de empleados y departamentos. Hazlo como un proyecto Eclipse, dentro del proyecto crea un paquete para las clases anteriores. Realiza las siguientes clases Java dentro del proyecto:

1. Una clase que inserte objetos en la base de datos.
2. Clase para visualizar todos los departamentos y empleados de la base de datos. Por cada empleado visualizar el nombre de su director y el de su departamento.
3. Clase donde se realizan diferentes consultas:
   a) Apellidos de los empleados del departamento 10.
   b) Número de empleados del departamento de VENTAS .
   c) Apellido de los empleados cuyo director es FERNÁNDEZ.
   d) Por cada departamento el número de empleados.

UNIDAD 5

BASES DE DATOS XML

CONTENIDOS

5.1 BASES DE DATOS NATIVAS XML
5.2 BASE DE DATOS EXIST
5.2.1 Instalación de eXist
5.2.2 Primeros pasos con eXist
5.2.3 El cliente de administración de eXist
5.3 LENGUAJES DE CONSULTAS XPATH Y XQUERY
5.3.1 Expresiones XPath
5.3.2 Nodos atributos XPath
5.3.3 Axis XPath
5.3.4 Consultas XQuery
5.3.5 Operadores y funciones más comunes en XQuery
5.3.6 Consultas complejas con XQuery
5. 4 ACCESO A EXIST DESDE JAVA
5.4.1 La API XML:DB para bases de datos XML
5.4.2 La API XQJ (XQUERY)
5.4.3 Tratamiento de excepciones
5.5 EJERCICIOS

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Instalar y utilizar bases de datos XML nativas.
- Realizar consultas en documentos y colecciones XML, utilizando lenguajes XPath y XQuery.
- Desarrollar aplicaciones para acceder a los datos de la base de datos XML nativa.
- Desarrollar aplicaciones para añadir, modificar y eliminar documentos XML de la base de datos.

5.1 BASES DE DATOS NATIVAS XML

A diferencia de las bases de datos relacionales (centradas en los datos), las bases de datos nativas XML, no poseen campos, ni almacenan datos, lo que almacenan son documentos XML, son bases de datos centradas en documentos y la unidad mínima de almacenamiento es el documento XML.

Podemos definir una base de datos nativa XML (o XML nativa), como un sistema de gestión de información que cumple con lo siguiente:

- Define un modelo lógico para un documento XML (y no para los datos que contiene el documento), y almacena y recupera los documentos según este modelo.
- Tiene una relación transparente con el mecanismo de almacenamiento, que debe incorporar las características *ACID* de cualquier SGBD *(Atomicity, Consistency, Isolation and Durability: Atomicidad, Consistencia, Aislamiento y Durabilidad)*.
- Incluye un número arbitrario de niveles de datos y complejidad.
- Permite las tecnologías de consulta y transformación propias de XML, XQuery, XPath, XSLT, etc., como vehículo principal de acceso y tratamiento.

Ventajas de las bases de datos XML:

- Ofrecen un acceso y almacenamiento de información ya en formato XML, sin necesidad de incorporar código adicional.
- La mayoría incorpora un motor de búsqueda de alto rendimiento.
- Es muy sencillo añadir nuevos documentos XML al repositorio.
- Se pueden almacenar datos heterogéneos.

Por el contrario, se pueden considerar como desventajas de las bases de datos XML:

- Puede resultar difícil indexar documentos para realizar búsquedas.
- No suelen ofrecer funciones para la agregación (crucial para el procesamiento de transacciones en línea OLTP -*Online Transaction Processing*) en muchos casos hay que reintroducir todo el documento para modificar una sola línea.
- Se suele almacenar la información en XML como un documento o como un conjunto de nodos, por lo que su síntesis para formar nuevas estructuras sobre la marcha puede resultar complicada y lenta.

En la actualidad la mayoría de los SGBD incorporan mecanismos para extraer y almacenar datos en formato XML. A continuación se muestra un ejemplo de soporte de datos XML en Oracle y en MySQL:

– La siguiente select de ORACLE devuelve las filas de la tabla EMPLE en formato XML:

```
SELECT XMLELEMENT("EMP_ROW",
          XMLFOREST(EMP_NO  ,APELLIDO,OFICIO, DIR, FECHA_ALT ,
          SALARIO , COMISION ,DEPT_NO )) FILA_EN_XML
     FROM EMPLE;
```


– Creación de una tabla que almacena datos del tipo XML (el tipo de dato XMLTYPE permite almacenar y consultar datos XML):

```
CREATE TABLE TABLA_XML_PRUEBA (COD NUMBER, DATOS XMLTYPE);
```

– Insertar filas en formato XML:

```
INSERT INTO TABLA_XML_PRUEBA VALUES (1,
XMLTYPE('<FILA_EMP><EMP_NO>123</EMP_NO><APELLIDO>RAMOS MARTÍN</APELLIDO>
<OFICIO>PROFESORA</OFICIO><SALARIO>1500</SALARIO></FILA_EMP>'));

INSERT INTO TABLA_XML_PRUEBA VALUES (1, XMLTYPE('<FILA_EMP><EMP_NO>124</EMP_NO>
<APELLIDO>GARCÍA SALADO</APELLIDO><OFICIO>FONTANERO</OFICIO>
<SALARIO>1700</SALARIO></FILA_EMP>'));
```

– Extracción de datos de la tabla, se utilizan expresiones XPath:

Visualiza los apellidos de los empleados:

```
SELECT EXTRACTVALUE(DATOS,'/FILA_EMP/APELLIDO') FROM TABLA_XML_PRUEBA;
```

Visualiza el nombre del empleado con número de empleado = 123

```
SELECT EXTRACTVALUE(DATOS,'/FILA_EMP/APELLIDO') FROM TABLA_XML_PRUEBA
WHERE EXISTSNODE(DATOS, '/FILA_EMP[EMP_NO=123]') = 1;
```

– En el siguiente ejemplo se muestra como MySQL devuelve los datos de una tabla en formato XML. Por ejemplo, disponemos de la BD en MySQL *ejemplo* y se desea obtener los datos de la tabla *departamentos* en formato XML. El usuario de la BD y su clave se llama también *ejemplo*.

Desde la línea de comando y en la carpeta donde se encuentra instalado MySQL (por ejemplo, desde la carpeta D:\xampp\mysql\bin>) escribiremos la siguiente orden para obtener los datos de la tabla *departamentos*:

```
mysql --xml -u ejemplo -p -e "select * from departamentos;" ejemplo
```

Donde --xml, se utiliza para producir una salida en XML.
-u, a continuación se pone el nombre de usuario de la BD *ejemplo*.
-p, al pulsar en la tecla [Intro] para ejecutar la orden nos pide la clave del usuario (*ejemplo*)
-e, ejecuta el comando y sale de MySQL.

El resultado es:

```
<?xml version="1.0"?>
<resultset statement="select * from departamentos"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="dept_no">10</field>
        <field name="dnombre">CONTABILIDAD</field>
```

```
        <field name="loc">SEVILLA</field>
    </row>
    <row>
        <field name="dept_no">20</field>
        <field name="dnombre">INVESTIGACIÓN</field>
        <field name="loc">MADRID</field>
    </row>
    <row>
        <field name="dept_no">30</field>
        <field name="dnombre">VENTAS</field>
        <field name="loc">BARCELONA</field>
    </row>
    <row>
        <field name="dept_no">40</field>
        <field name="dnombre">PRODUCCIÓN</field>
        <field name="loc">BILBAO</field>
  </row>
</resultset>
```

Si deseamos redireccionar la salida al fichero *departamentos.xml* escribiremos en la misma línea:

```
mysql --xml -u ejemplo -p -e "select * from departamentos;" ejemplo
>departamentos.xml
```

5.2 BASE DE DATOS EXIST

eXist es un SGBD libre de código abierto que almacena datos XML de acuerdo a un modelo de datos XML. El motor de base de datos está completamente escrito en Java, soporta los estándares de consulta XPath, XQuery y XSLT, además de indexación de documentos y soporte para la actualización de los datos y para multitud de protocolos como SOAP, XML-RPC, WebDav y REST. Con el SGBD se dan aplicaciones que permiten ejecutar consultas directamente sobre la BD.

Los documentos XML se almacenan en colecciones, las cuales pueden estar anidadas; desde un punto de vista práctico el almacén de datos funciona como un sistema de ficheros. Cada documento está en una colección. No es necesario que los documentos tengan una *DTD* o un *XML Schema* asociado, y dentro de una colección pueden almacenarse documentos de cualquier tipo.

En la carpeta eXist\webapp\WEB-INF\data es donde se guardan los ficheros más importantes de la BD:

- ***dom.dbx,*** el almacén central nativo de datos; es un fichero paginado donde se almacenan todos los nodos del documento de acuerdo al modelo DOM del W3C.
- ***collections.dbx,*** se almacena la jerarquía de colecciones y relaciona esta con los documentos que contiene; se asigna un identificador único a cada documento de la colección que es almacenado también junto al índice.
- ***elements.dbx***, es donde se guarda el índice de elementos y atributos. El gestor automáticamente indexa todos los documentos utilizando índices numéricos para identificar los nodos del mismo (elementos, atributos, texto y comentarios). Durante la indexación se asigna un identificador único a cada documento de la colección, que es almacenado también junto al índice.

- ***words.dbx,*** por defecto eXist indexa todos los nodos de texto y valores de atributos dividiendo el texto en palabras. En este fichero se almacena esta información. Cada entrada del índice está formada por un par <id de colección, palabra> y después una lista al nodo que contiene dicha palabra.

5.2.1 INSTALACIÓN DE EXIST

eXist puede funcionar de distintos modos:

- Funcionando como servidor autónomo (ofreciendo servicios de llamada remota a procedimientos que funciona sobre Internet como XML-RPC, WebDAV y REST).
- Insertado dentro de una aplicación Java.
- En un servidor J2EE, ofreciendo servicios XML-RPC, SOAP y WebDAV.

En el sitio http://exist-db.org/exist/download.xml podremos descargar la última versión de la BD. En este caso se instalará la versión estable ***eXist-setup-1.4.2-rev16251.jar***. Se instala utilizando el fichero JAR proporcionado en la web que ha de invocarse desde la línea de comandos (método recomendado: *java -jar eXist-setup-1.4.2-rev16251.jar*) o bien haciendo doble clic sobre el icono correspondiente una vez descargado.

En el proceso de instalación, se nos pide introducir la ruta donde está instalado Java JDK (mínimo versión 5, C:\Program Files\Java\jdk1.6.0_29). También indicaremos donde se desea que se instale la BD, nos pedirá contraseña para el usuario administrador de la BD *admin* y marcaremos que instale todas las aplicaciones, véase la Figura 5.1.

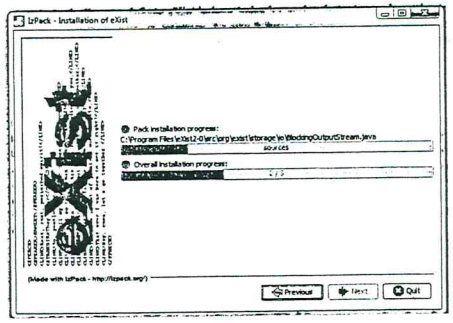


Figura 5.1. Instalación de eXist.

Una vez instalado para poder trabajar con la BD primero hay que iniciar el servidor y luego escribimos en el navegador: http://localhost:8080/exist/ . También se puede instalar el servidor como un servicio (*Install eXist as Service*), en este caso se iniciaría la BD de forma automática. En la Figura 5.2 se muestra el menú de eXist.

Figura 5.2. Menú de eXist.

Al arrancar eXist se ejecuta ***startup.bat*** (Windows) o ***startup.sh*** en (Linux), que se encuentra en la carpeta *bin* de eXist. Una vez arrancado eXist no se puede cerrar el programa mientras se esté trabajando con dicho SGBD. Si se hace, automáticamente el sistema deja de funcionar. En la Figura 5.3 se muestra la pantalla inicial de la BD eXist.

También es importante tener en cuenta que es posible que el puerto de conexión de la base de datos (el 8080) se encuentre bloqueado por alguna aplicación (por ejemplo, por el Firewall de Windows). Hay que asegurar que el puerto está abierto o habilitado.

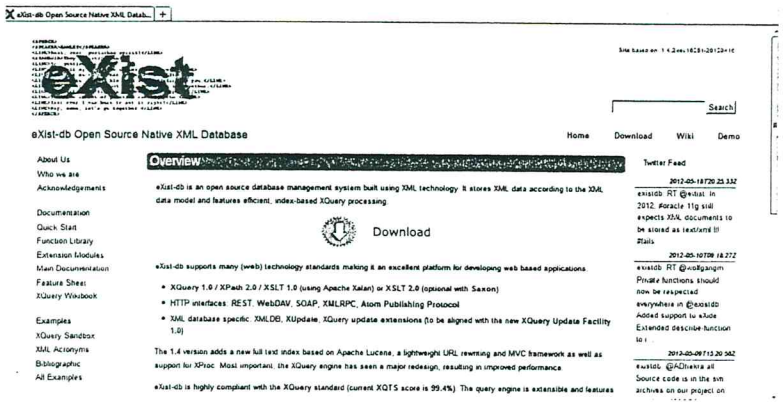


Figura 5.3. Vista inicial de eXist.

5.2.2 PRIMEROS PASOS CON EXIST

Lo primero que vamos a hacer es cargar un fichero XML que nos va a servir como base de datos para realizar consultas utilizando XQuery y XPath, así pues será necesario crear una colección y luego almacenar el documento XML en la colección.

En el menú de la izquierda bajando la pantalla, aparece el menú de *Administración*, seleccionamos el elemento ***Admin*** y accedemos a la administración de la BD eXist, nos conectamos con el usuario ***admin***, y escribimos la contraseña que hayamos elegido en la instalación. Véase la Figura 5.4.

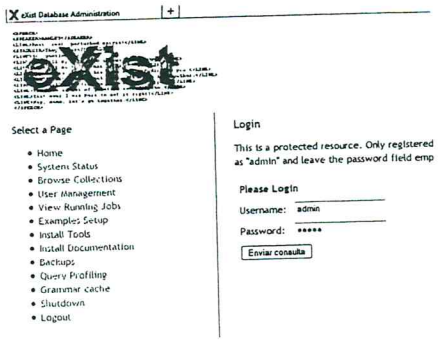


Figura 5.4. Panel de administración de eXist.

Desde la ventana de administración podremos crear usuarios si seleccionamos ***User Management***, se pueden instalar los ejemplos que vienen con la BD para hacer pruebas de consultas desde ***Examples Setup***, o se pueden realizar copias de seguridad desde la opción ***Backups***.

En nuestro caso deseamos crear una colección de documentos XML y subir los documentos *empleados.xml* y *departamentos.xml* para luego realizar consultas. Hacemos clic en ***Browse Collections***, creamos la colección ***Pruebas***, haciendo clic en ***Create Collection.*** Entramos en la colección creada y subimos los documentos que se encuentran en la carpeta de la unidad *empleados.xml* y *departamentos.xml*. En la Figura 5.5 vemos como queda la colección creada con los dos documentos XML subidos. Recuerda que los documentos XML deben estar bien formados para no tener problemas en la subida de los mismos a la base de datos.

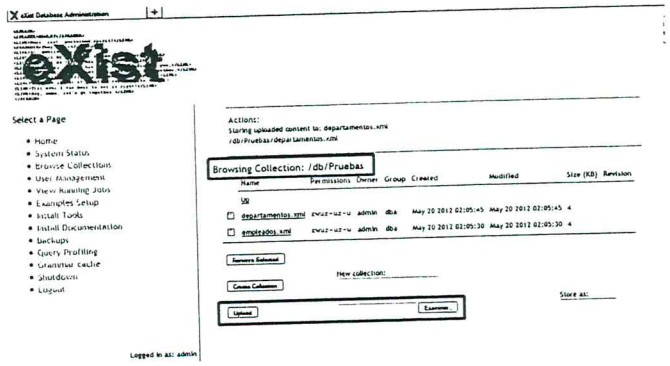


Figura 5.5. Colección *Pruebas*.

Una vez creada la colección hay que abrir el gestor de consultas de eXist, para ello regresamos a la ventana inicial pulsando el enlace *Home*, y desde ahí se selecciona ***XQuery Sandbox*** que aparece en el menú de la izquierda dentro de *Examples*. Desde el gestor de consultas se podrán realizar consultas a los documentos XML de nuestra colección, en la Figura 5.6 se muestra la ventana para la realización de consultas, en la parte superior se escribe la consulta y en la inferior se muestra el resultado. El resultado de las consultas son etiquetas XML.

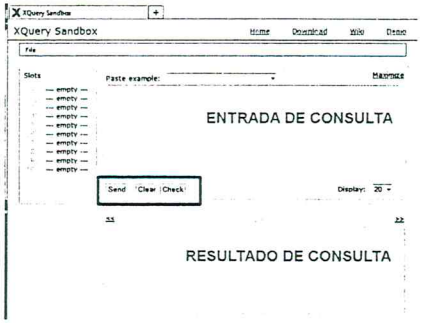


Figura 5.6. ***XQuery Sandbox***, gestor de consultas de eXist.

5.2.3 EL CLIENTE DE ADMINISTRACIÓN DE EXIST

Para ejecutar el cliente abrimos ***eXist Client Shell***, desde el menú de programas de eXist (Inicio -> Programas -> eXist XML Database), o bien ejecutando *eXist\bin\client.bat*. También se descarga el cliente desde la URL *http://localhost:8080/exist/webstart/**exist.jnlp*** y luego se ejecuta. A continuación se muestra la ventana donde pide que se configure la conexión o se elija una ya creada. Tecleraremos el nombre de usuario, su contraseña, la URL de la BD y un título para la conexión, estos datos se quedarán guardados para posteriores conexiones, véase la Figura 5.7.

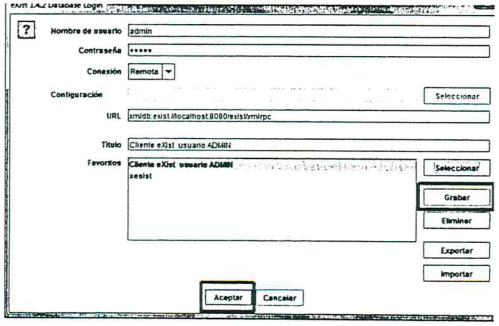


Figura 5.7. Ejecución de *eXist Client Shell* y configuración de la conexión.

A continuación se muestra la ventana del *Cliente de Administración eXist* (Figura 5.8), desde aquí se podrán realizar todo tipo de operaciones sobre la BD, crear y borrar colecciones, añadir y eliminar documentos a las colecciones, modificar los documentos, crear copias de seguridad y restaurarlas, administrar usuarios y realizar consultas XPath, entre otras operaciones.

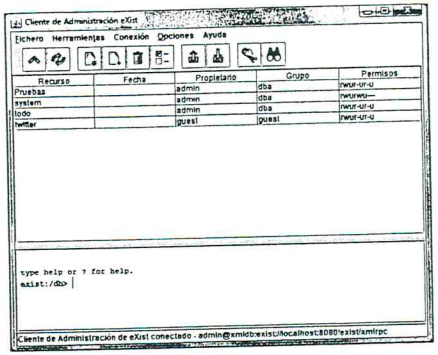


Figura 5.8. Cliente de administración de eXist.

Si pulsamos al botón *Consultar la BD usando XPath*, aparece la ventana de consultas ***Query Dialog***, desde aquí podremos elegir el ***contexto*** sobre el que ejecutaremos las consultas, se pueden también guardar las consultas y los resultados de las consultas en ficheros externos. En la parte superior escribimos la consulta, pulsamos el botón *Submit* y en la parte inferior se muestra el resultado, también se puede ver la traza de ejecución seguida en la ejecución de la consulta. En la Figura 5.9 se muestra la ventana de *Query Dialog*.

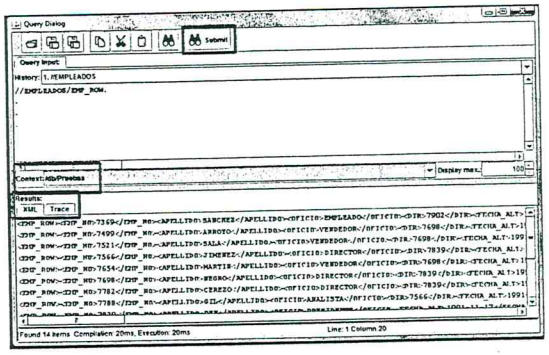


Figura 5.9. Ventana de consultas ***Query Dialog*** de eXist.

5.3 LENGUAJES DE CONSULTAS XPATH Y XQUERY

Ambos son estándares para acceder y obtener datos desde documentos XML, estos lenguajes tienen en cuenta que la información en los documentos está semiestructurada o jerarquizada como árbol.

XPath, es el lenguaje de rutas de XML, se utiliza para navegar dentro de la estructura jerárquica de un XML.

XQuery es a XML lo mismo que SQL es a las bases de datos relacionales, es decir, es un lenguaje de consulta diseñado para consultar documentos XML. Abarca desde ficheros XML hasta bases de datos relacionales con funciones de conversión de registros a XML. XQuery contiene a XPath, toda expresión de consulta en XPath es válida en XQuery, pero XQuery permite mucho más.

5.3.1 EXPRESIONES XPATH

XPath es un lenguaje que permite seleccionar nodos de un documento XML y calcular valores a partir de su contenido. Existen dos versiones de XPath aprobadas por el W3C, aunque la versión más utilizada sigue siendo la versión 1.

La forma en que XPath selecciona partes del documento XML se basa en la representación arbórea que se genera del documento. A la hora de recorrer un árbol XML, podemos encontrarnos con los siguientes tipos de nodos:

- ***nodo raíz,*** es la raíz del árbol, se representa por /.
- ***nodos elemento,*** cualquier elemento de un documento XML, son las etiquetas del árbol.
- ***nodos texto,*** los caracteres que están entre las etiquetas.
- ***nodos atributo,*** son como propiedades añadidas a los nodos elemento, se representan con @.
- ***nodos comentario,*** las etiquetas de comentario.
- ***nodos espacio de nombres,*** contienen espacios de nombres.
- ***nodos instrucción de proceso***, contienen instrucciones de proceso, van entre las etiquetas <? ..... ?>.

Por ejemplo. Dado este documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<universidad>
<espacio xmlns="http://www.misitio.com"
            xmlns:prueba="http://www.misitio.com/pruebas"  />
  <!-- DEPARTAMENTO 1 -->
  <departamento telefono="112233" tipo="A">
   <codigo>IFC1</codigo>
   <nombre>Informática</nombre>
  </departamento>
  <!-- DEPARTAMENTO 2 -->
  <departamento telefono="990033" tipo="A">
    <codigo>MAT1</codigo>
    <nombre>Matemáticas</nombre>
```



```
  </departamento>
  <!-- DEPARTAMENTO 3 -->
  <departamento telefono="880833" tipo="B">
   <codigo>MAT2</codigo>
   <nombre>Análisis</nombre>
  </departamento>
</universidad>
```

Nos encontramos los siguientes tipos de nodos:

TIPO NODO	
Elemento	`<universidad>, <departamento>, <codigo>, <nombre>`
Texto	`IFC1, Informática, MAT1, Matemáticas, MAT2, Análisis`
Atributo	`telefono="112233" tipo="A" , telefono="990033" tipo="A", telefono="880833" tipo="B"`
Comentario	`<!-- DEPARTAMENTO 1 -->, <!-- DEPARTAMENTO 2 --> <!-- DEPARTAMENTO 3 -->`
Espacio de nombres	`<espacio xmlns="http://www.misitio.com" xmlns:prueba=http://www.misitio.com/pruebas />`
Instrucción de proceso	`<?xml version="1.0" encoding="ISO-8859-1"?>`

Los test sobre los tipos de nodos pueden ser:

- Nombre del nodo, para seleccionar un nodo concreto, ej: /universidad
- ***prefix:****, para seleccionar nodos con un espacio de nombres determinado.
- ***text()***, selecciona el contenido del elemento, es decir el texto, ej: //nombre/text().
- ***node()***, selecciona todos los nodos , los elementos y el texto, ej: /universidad/node().
- ***processing-instruction()***, selecciona nodos que son instrucciones de proceso.
- ***comment()***, selecciona los nodos de tipo comentario, /universidad/comment().

La sintaxis básica de XPath es similar a la del direccionamiento de ficheros. Utiliza ***descriptores de ruta o de camino*** que sirven para seleccionar los nodos o elementos que se encuentran en cierta ruta en el documento. Cada descriptor de ruta o paso de búsqueda puede a su vez dividirse en tres partes:

- ***eje:*** indica el nodo o los nodos en los que se realiza la búsqueda.
- ***nodo de comprobación:*** especifica el nodo o los nodos seleccionados dentro del eje.
- ***predicado:*** permite restringir los nodos de comprobación. Los predicados se escriben entre corchetes.

Las expresiones XPath se pueden escribir utilizando una sintaxis abreviada, fácil de leer, o una sintaxis más completa en la que aparecen los nombres de los ejes (AXIS), más compleja. Por ejemplo, estas dos expresiones devuelven los departamentos con más de 3 empleados, la primera es la forma abreviada y la segunda es la completa:

```
/universidad/departamento[count(empleado)>3]
/child::universidad/child::departamento[count(child::empleado)>3]
```

En este tema empezaremos con la sintaxis abreviada, así pues los descriptores se forman simplemente nombrando la etiqueta separada por / (hay que poner el nombre de la etiqueta tal cual está en el documento XML, recuerda que hace distinción entre mayúsculas y minúsculas).

Si el descriptor comienza con / se supone que es una **ruta desde la raíz**. Para seguir una ruta indicaremos los distintos nodos de paso: /paso1/paso2/paso3... Si las rutas comienzan con / son **rutas absolutas**, en caso contrario serán relativas.

Si el descriptor comienza con // se supone que la ruta descrita puede comenzar en cualquier parte de la colección.

– Ejemplos XPath utilizando una sintaxis abreviada:

A partir de la colección *Pruebas*, que contiene los documentos *departamentos.xml* y *empleados.xml*, cuyas estructuras se muestras en la Figura 5.10. Realizar desde ***Query Dialog*** o ***XQuery Sandbox*** las siguientes consultas:

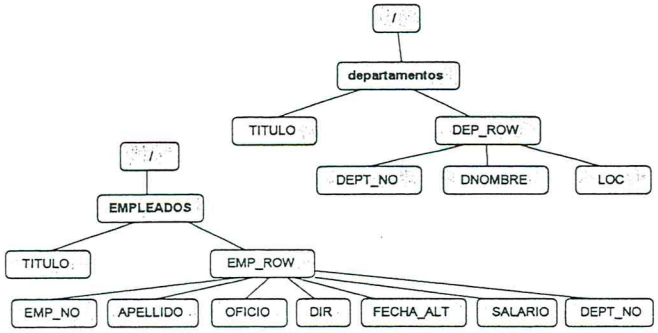


Figura 5.10. Estructuras de *departamentos.xml* y *empleados.xml*.

- `/`, si ejecutamos esta orden en el ***Query Dialog*** y se está dentro del contexto /db/Pruebas devuelve todos los datos de los departamentos y los empleados, es decir incluye todas las etiquetas que cuelgan del nodo *departamentos* y del nodo *EMPLEADOS*, que están dentro de la colección *Pruebas*. Si se ejecuta desde XQuery Sandbox, se obtiene otro resultado, pues el contexto no es el mismo.
- `/departamentos`, devuelve todos los datos de los departamentos, es decir incluye todas las etiquetas que cuelgan del nodo raíz *departamentos*.
- `/departamentos/DEP_ROW`, devuelve todas las etiquetas dentro de cada DEP_ROW.
- `/departamentos/DEP_ROW/node()`, devuelve todas las etiquetas dentro de cada DEP_ROW, no incluye DEP_ROW.
- `/departamentos/DEP_ROW/DNOMBRE`, devuelve los nombres de departamentos de cada DEP_ROW, entre etiquetas.
- `/departamentos/DEP_ROW/DNOMBRE/text()`, devuelve los nombres de departamentos de cada DEP_ROW, ya sin las etiquetas.
- `//LOC/text()`, devuelve todas las localidades, de toda la colección (//), solo hay 4.

- `//DEPT_NO`, devuelve todos los números de departamentos, entre etiquetas. Observa que en este caso devuelve 18 filas en lugar de 4, es porque recoge todos los elementos DEPT_NO de la colección y, recuerda que en la colección también hemos incluido el documento *empleados.xml*, que tiene 14 empleados con su DEPT_NO, véase la Figura 5.11.

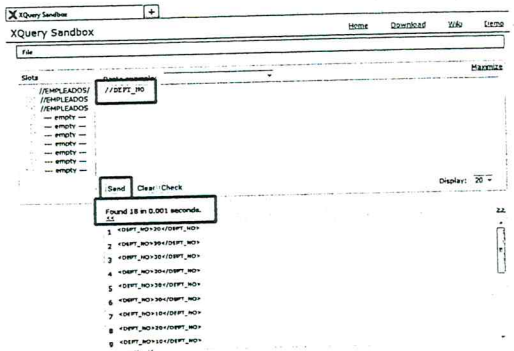


Figura 5.11. ***XQuery Sanbox***, ejecución de la consulta `//DEPT_NO`.

- ***El operador*** * se usa para nombrar a cualquier etiqueta, se usa como comodín. Por ejemplo:
  - El descriptor `/*/DEPT_NO` selecciona las etiquetas DEPT_NO que se encuentran a 1 nivel de profundidad desde la raíz, en este caso ninguna.
  - `/*/*/DEPT_NO` selecciona las etiquetas DEPT_NO que se encuentran a dos niveles de profundidad desde la raíz, en este caso 18.
  - `/departamentos/*` selecciona las etiquetas que van dentro de la etiqueta departamentos y sus subetiquetas.
  - /* ¿Qué salida produce este descriptor?, depende del contexto en el que se ejecute.

- ***Condiciones de selección.*** Se utilizarán los corchetes para seleccionar elementos concretos, en las condiciones se pueden usar los comparadores: <, >, <=, >=, =, *!=* , ***or, and y not*** (or, and y not deben escribirse en minúscula). Se utilizará el separador | para unir varias rutas. Ejemplos:
  - `/EMPLEADOS/EMP_ROW[DEPT_NO=10]`, selecciona todos los elementos o nodos (etiquetas) dentro de EMP_ROW de los empleados del DEPT_NO= 10.
  - `/EMPLEADOS/EMP_ROW/APELLIDO|/EMPLEADOS/EMP_ROW/DEPT_NO` selecciona los nodos APELLIDO y DEPT_NO de los empleados.
  - `/EMPLEADOS/EMP_ROW[DEPT_NO=10]/APELLIDO/text()`, selecciona los apellidos de los empleados del DEPT_NO=10.
  - `/EMPLEADOS/EMP_ROW[not(DEPT_NO = 10)]`, selecciona todos los empleados (etiquetas) que NO son del DEPT_NO igual a 10.
  - `/EMPLEADOS/EMP_ROW[not(OFICIO = 'ANALISTA')]/APELLIDO/text()`, selecciona los APELLIDOS de los empleados que NO son analistas.
  - `/EMPLEADOS/EMP_ROW[DEPT_NO=10]/APELLIDO | /EMPLEADOS/EMP_ROW[DEPT_`

NO=10]/OFICIO, selecciona el APELLIDO y el OFICIO de los empleados del DEPT_NO=10.

- //*[DEPT_NO=10]/DNOMBRE/text(), /departamentos/DEP_ROW[DEPT_NO=10]/DNOMBRE/text(), estas dos consultas devuelven el nombre del departamento 10.
- //*[OFICIO="EMPLEADO"]//EMP_ROW, devuelve los empleados con OFICIO "EMPLEADO", por cada empleado devuelve todos sus elementos. Busca en cualquier parte de la colección //. Esto devuelve lo mismo: /EMPLEADOS/EMP_ROW[OFICIO="EMPLEADO"]//EMP_ROW.
- /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=10], devuelve los datos de los empleados con SALARIO mayor de 1300 y del departamento 10.
- /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=20]/APELLIDO | /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=20]/OFICIO, devuelve el APELLIDO y el OFICIO de los empleados con SALARIO mayor de 1300 y del departamento 20. Se utiliza el separador | para unir las dos rutas.

- **Utilización de funciones y expresiones matemáticas.**

- Un número dentro de los corchetes representa la posición del elemento en el conjunto seleccionado. Ejemplos:
  /EMPLEADOS/EMP_ROW[1], devuelve todos los datos del primer empleado.
  /EMPLEADOS/EMP_ROW[5]/APELLIDO/text(), devuelve el APELLIDO del quinto empleado.
- La función **last()** selecciona el último elemento del conjunto seleccionado. Ejemplos:
  /EMPLEADOS/EMP_ROW[last()], selecciona todos los datos del último empleado.
  /EMPLEADOS/EMP_ROW[last()-1]/APELLIDO/text(), devuelve el APELLIDO del penúltimo empleado.
- Lo función **position()** devuelve un número igual a la posición del elemento actual.
  /EMPLEADOS/EMP_ROW[position()=3], obtiene los elementos del empleado que ocupa la posición 3.
  /EMPLEADOS/EMP_ROW[position()<3]/APELLIDO, selecciona el apellido del los elementos cuya posición es menor de 3, es decir devuelve los apellidos del primer y segundo empleado.
- La función **count()** cuenta el número de elementos seleccionados. Ejemplos:
  /EMPLEADOS/count(EMP_ROW), devuelve el número de empleados.
  /EMPLEADOS/count(EMP_ROW[DEPT_NO=10]), cuenta el número de empleados del departamento 10.
  /EMPLEADOS/count(EMP_ROW[OFICIO="EMPLEADO" and SALARIO>1300]), cuenta el nº de empleados con oficio EMPLEADO y SALARIO mayor de 1300.
  //*[count(*)=3], devuelve elementos que tienen 3 hijos.
  //*[count(DEP_ROW)=4], devuelve los elementos que contienen 4 hijos DEP_ROW, devolverá la etiqueta departamentos y todas las subetiquetas.
- La función **sum()** devuelve la suma del elemento seleccionado. Ejemplos:
  sum(/EMPLEADOS/EMP_ROW/SALARIO), devuelve la suma del SALARIO. Si la etiqueta a sumar la considera *string* hay que convertirla a número utilizando la función ***number***.

sum(/EMPLEADOS/EMP_ROW[DEPT_NO=20]/SALARIO), devuelve la suma de SALARIO de los empleados del DEPT_NO = 20.

- Función **max()** devuelve el máximo, **min()** devuelve el mínimo y **avg()** devuelve la media del elemento seleccionado. Ejemplos:
  max(/EMPLEADOS/EMP_ROW/SALARIO), devuelve el salario máximo.
  min(/EMPLEADOS/EMP_ROW/SALARIO), devuelve el salario mínimo.
  min(/EMPLEADOS/EMP_ROW[OFICIO="ANALISTA"]/SALARIO), devuelve el salario mínimo de los empleados con OFICIO ANALISTA.
  avg(/EMPLEADOS/EMP_ROW/SALARIO), devuelve la media del salario.
  avg(/EMPLEADOS/EMP_ROW[DEPT_NO=20]/SALARIO), devuelve la media del salario de los empleados del departamento 20.
- La función **name()** devuelve el nombre del elemento seleccionado. Ejemplos:
  //*[name()='APELLIDO'], devuelve todos los apellidos, entre sus etiquetas.
  count(//*[name()='APELLIDO']), cuenta las etiquetas con nombre APELLIDO.
- La función **concat(cad1, cad2, ...)** concatena las cadenas. Ejemplos:
  /EMPLEADOS/EMP_ROW[DEPT_NO=10]/concat(APELLIDO, " - ",OFICIO)
  Devuelve el apellido y el oficio concatenados de los empleados del departamento 10.
  /EMPLEADOS/EMP_ROW/concat(APELLIDO, " - ", OFICIO, " - ", SALARIO)
  Devuelve la concatenación de apellido, oficio y salario de los empleados. Véase la Figura 5.12.

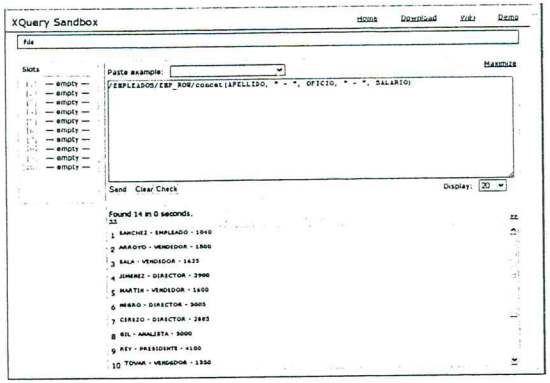


Figura 5.12. Ejecución de una consulta de concatenación.

- La función **starts-with(cad1, cad2)** es verdadera cuando la cadena cad1 tiene como prefijo a la cadena cad2. Ejemplos:
  /EMPLEADOS/EMP_ROW[starts-with(APELLIDO,'A')], obtiene los elementos de los empleados cuyo APELLIDO empieza por 'A'.
  /EMPLEADOS/EMP_ROW[starts-with(OFICIO,'A')]/concat(APELLIDO," - ",OFICIO)
  obtiene APELLIDO y NOMBRE concatenados de los empleados cuyo OFICIO empieza por 'A'.
- La función **contains(cad1, cad2)** es verdadera cuando la cadena cad1 contiene a la cadena cad2.

`/EMPLEADOS/EMP_ROW[contains(OFICIO,'OR')]/OFICIO`, devuelve los oficios que contienen la la sílaba 'OR'.

`/EMPLEADOS/EMP_ROW [contains(APELLIDO,'A')]/APELLIDO`, devuelve los apellidos que contienen una 'A'.

- La función **string-length(argumento)** devuelve el número de caracteres de su argumento.

  `/EMPLEADOS/EMP_ROW/concat(APELLIDO,' = ', string-length(APELLIDO))`, devuelve concatenados el apellido con su número de caracteres.

  `/EMPLEADOS/EMP_ROW[string-length(APELLIDO)<4]`, devuelve los datos de los empleados cuyo APELLIDO tiene menos de 4 caracteres.
- Operador matemático **div( )** realiza divisiones en punto flotante.

  `/EMPLEADOS/EMP_ROW/concat(APELLIDO,' , ',SALARIO,' - ',SALARIO div 12)`, devuelve los datos concatenados de APELLIDO, SALARIO y el salario dividido por 12.

  `sum(/EMPLEADOS/EMP_ROW/SALARIO) div count(/EMPLEADOS/EMP_ROW)`, devuelve la suma de salarios dividido por el contador de empleados.
- Operador matemático **mod( )** calcula el resto de la división

  `/EMPLEADOS/EMP_ROW/concat(APELLIDO,' , ',SALARIO,' - ',SALARIO mod 12)`, devuelve los datos concatenados de APELLIDO, SALARIO y el resto de dividir el SALARIO por 12.

  `/EMPLEADOS/EMP_ROW[(SALARIO mod 12)=4]`, devuelve los datos de los empleados cuyo resto de dividir el SALARIO entre 12 sea igual a 4.

- **Otras funciones.**

  - ***data(expresión XPath)***, devuelve el texto de los nodos de la expresión sin las etiquetas.
  - ***number(argumento)***, para convertir a número el argumento, que puede ser cadena, booleano o un nodo.
  - ***abs(num)***, devuelve el valor absoluto del número.
  - ***ceiling(num)***, devuelve el entero más pequeño mayor o igual que la expresión numérica especificada.
  - ***floor(num)***, devuelve el entero más grande que sea menor o igual que la expresión numérica especificada.
  - ***round(num)***, redondea el valor de la expresión numérica.
  - ***string(argumento)***, convierte el argumento en cadena.
  - ***compare(exp1,exp2)***, compara las dos expresiones, devuelve 0 si son iguales, 1 si exp1>exp2, y -1 si exp1<exp2.
  - ***substring(cadena,comienzo,num)***, extrae de la *cadena*, desde la posición indicada en *comienzo* el número de caracteres indicado en *num*.
  - ***substring(cadena,comienzo)***, extrae de la *cadena*, los caracteres desde la posición indicada por *comienzo*, hasta el final.
  - ***lower-case(cadena)***, convierte a minúscula la *cadena*.
  - ***upper-case(cadena)***, convierte a mayúscula la *cadena*.
  - ***translate(cadena1,caract1,caract2)***, reemplaza dentro de *cadena1*, los caracteres que se expresan en *caract1*, por los correspondientes que aparecen en *caract2*, uno por uno.
  - ***ends-with(cadena1,cadena2)***, devuelve true si la *cadena1* termina en *cadena2*.

  - ***year-from-date(fecha)***, devuelve el año de la fecha, el formato de fecha es AÑO-MES-DIA.
  - ***month-from-date(fecha)***, devuelve el mes de la fecha.
  - ***day-from-date(fecha)***, devuelve el día de la fecha.

Puedes encontrar más funciones en la URL: http://www.w3schools.com/xpath/xpath_functions.asp

---

Actividad 1: Sube el documento *productos.xml* dentro de la colección *Pruebas*. Este documento contiene los datos de los productos de una distribuidora de componentes informáticos. Por cada producto tenemos el código, la denominación, el precio, el stock actual, el stock mínimo y el código de zona. Estos datos son:

```
<produc>
  <cod_prod>xxxxxx</cod_prod>
  <denominacion>xxxxxxxxxxxx</denominacion>
  <precio>xxxx</precio>
  <stock_actual>xxx</stock_actual>
  <stock_minimo>xxxx</stock_minimo>
  <cod_zona>xxxx</cod_zona>
</produc>
```

Realiza las siguientes consultas XPath:

- Obtén los nodos denominación y precio de todos los productos.
- Obtén los nodos de los productos que sean placas base.
- Obtén los nodos de los productos con precio mayor de 60 € y de la zona 20.
- Obtén el número de productos que sean memorias y de la zona 10.
- Obtén la media de precio de los micros.
- Obtén los datos de los productos cuyo stock mínimo sea mayor que su stock actual.
- Obtén el nombre de producto y el precio de aquellos cuyo stock mínimo sea mayor que su stock actual y sean de la zona 40
- Obtén el producto más caro.
- Obtén el producto más barato de la zona 20.
- Obtén el producto más caro de la zona 10.

5.3.2 NODOS ATRIBUTOS XPATH

Un nodo puede tener tantos atributos como se desee y para cada uno se le creará un ***nodo atributo***. Los *nodos atributo* NO se consideran como hijos, sino más bien como etiquetas añadidas al nodo elemento. Cada *nodo atributo* consta de un nombre, un valor (que es siempre una cadena) y un posible "espacio de nombres".

Partimos del documento *universidad.xml*, que se encuentra en la carpeta de la unidad. Sube este documento dentro de la colección *Pruebas* de la BD.

```
<universidad>
  <departamento telefono="112233"
tipo="A">
   <codigo>IFC1</codigo>
   <nombre>Informática</nombre>
   <empleado salario="2000">
      <puesto>Asociado</puesto>
      <nombre>Juan Parra</nombre>
   </empleado>
   <empleado salario="2300">
      <puesto>Profesor</puesto>
      <nombre>Alicia Martín</nombre>
   </empleado>
  </departamento>

  <departamento telefono="990033"
tipo="A">
   <codigo>MAT1</codigo>
   <nombre>Matemáticas</nombre>
   <empleado salario="1900">
      <puesto>Técnico</puesto>
      <nombre>Ana García</nombre>
   </empleado>
   <empleado salario="2100">
      <puesto>Profesor</puesto>
      <nombre>Mª Jesús Ramos</nombre>
   </empleado>
<empleado salario="2300">
      <puesto>Profesor</puesto>
      <nombre>Pedro Paniagua</nombre>
   </empleado>
   <empleado salario="2500">
      <puesto>Tutor</puesto>
      <nombre>Antonia González</nombre>
   </empleado>
  </departamento>

  <departamento telefono="880833"
tipo="B">
   <codigo>MAT2</codigo>
   <nombre>Análisis</nombre>
   <empleado salario="1900">
      <puesto>Asociado</puesto>
      <nombre>Laura Ruiz</nombre>
   </empleado>
   <empleado salario="2200">
      <puesto>Asociado</puesto>
      <nombre>Mario Garcia</nombre>
   </empleado>
  </departamento>

</universidad >
```

En este documento los elementos o nodos que llevan atributos son los siguientes: los elementos departamento llevan los atributos teléfono y tipo, y los elementos empleado llevan el atributo salario. El árbol del documento se muestra en la Figura 5.13, los atributos se representan con elipses.

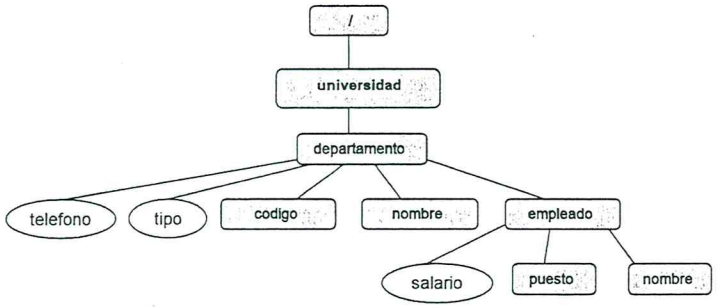


Figura 5.13. Árbol del documento *universidad.xml*.

Para referirnos a los atributos de los elementos se usa @ antes del nombre, por ejemplo @telefono, @tipo, @salario. En un descriptor de ruta los atributos se nombran como si fueran etiquetas hijo pero anteponiendo @.

Ejemplos:

- `/universidad/departamento[@tipo]`, se obtienen los datos de los departamentos que tengan el atributo tipo. Si ponemos `data(/universidad/departamento[@tipo])`, nos devuelve los datos sin las etiquetas.
- `/universidad/departamento/empleado[@salario]`, se obtienen los datos de los empleados que tengan el atributo salario.
- `/universidad/departamento[@telefono="990033"]`, se obtienen los datos del departamento cuyo teléfono es 990033. Si ponemos `data(/universidad/departamento[@telefono="990033"])`, devuelve lo mismo pero sin las etiquetas de los elementos.
- `/universidad/departamento[@telefono="990033"]/nombre/text()`, se obtienen el nombre de departamento cuyo teléfono es 990033.
- `//departamento[@tipo='B']`, se obtienen los datos de los departamentos cuyo tipo es B.
- `/universidad/departamento[@tipo="A"]/empleado`, se obtienen los datos de los empleados de los departamentos del tipo A.
- `/universidad/departamento/empleado[@salario>"2100"]`, se obtienen los datos de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento/empleado[@salario>"2100"]/nombre/text()`, se obtienen los nombres de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento/empleado[@salario>"2100"] /concat(nombre,' ', @salario)`, se obtienen los datos concatenados del nombre de empleado y su salario, de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento[@tipo="A"]/count(empleado)`, devuelve el número de empleados que hay en los departamentos de tipo=A.
- `/universidad/departamento[@tipo="A"]/concat(nombre,' ',count(empleado))`, devuelve por cada departamento del tipo=A, la concatenación de su nombre y el número de empleados.
- `/universidad/departamento/concat(nombre,' ',count(empleado))`, devuelve el número de empleados por cada departamento
- `sum(//empleado/@salario)`, devuelve la suma total del salario de todos los empleados, hace lo mismo que esto: `sum(/universidad/departamento/empleado/@salario)`
- `/universidad/departamento/concat(nombre,' Total=', sum(empleado/@salario))`, obtiene por cada departamento la concatenación de su nombre y el total salario.
- `min(//empleado/@salario)`, devuelve el salario mínimo de todos los empleados.
- `/universidad/departamento/concat(nombre,' Minimo=', min(empleado/@salario))`
  `/universidad/departamento/concat(nombre,' Máximo=', max(empleado/@salario))`
  La primera obtiene por cada departamento la concatenación de su nombre y el mínimo salario, y la segunda el máximo salario.

- `/universidad/departamento/concat(nombre,' Media=', avg(empleado/@salario))`, obtiene por cada departamento la concatenación de su nombre y la media de salario.
- `/universidad/departamento[count(empleado)>3]`, obtiene los datos de departamentos con más de 3 empleados. `/universidad/departamento[count(empleado)>3]/nombre/text()`, en este caso devuelve el nombre de los departamentos con más de 3 empleados.
- `/universidad/departamento[@tipo="A" and count(empleado)>2]/nombre/text()`, devuelve el nombre de los departamentos de tipo A y con más de 2 empleados.

Actividad 2: Sube el documento sucursales.xml dentro de la colección *Pruebas*. Este documento contiene los datos de las sucursales de un banco. Por cada sucursal tenemos el teléfono, el código, el director de la sucursal, la población y las cuentas de la sucursal. Y por cada cuenta tenemos el tipo de cuenta AHORRO o PENSIONES, el nombre de la cuenta, el numero, el saldohaber y el saldodebe. Estos datos son:

```
<sucursales>
  <sucursal telefono="xxxxxxxx" codigo="xxxx">
      <director>xxxxxxxxxxxxxxxx</director>
          <poblacion>xxxxxxxxxxx</poblacion>
          <cuenta tipo="xxxxxxxxx">
                          <nombre>xxxx</nombre>
                                <numero>xxxx</numero>
                                <saldohaber>xxxxxx</saldohaber>
                                <saldodebe>xxxxx</saldodebe>
                      </cuenta>
                  . . . . . . . . . . .
  </sucursal>
  . . . . . . . . . . . . . . .
</sucursales>
```

Realiza las siguientes consultas XPath:

- Obtén los datos de las cuentas bancarias cuyo stipo sea AHORRO.
- Obtén por cada sucursal la concatenación de su código y el número de cuentas del tipo AHORRO que tiene.
- Obtén las cuentas de tipo PENSIONES de la sucursal con código SUC3.
- Obtén por cada sucursal la concatenación de los datos, código sucursal, director y total saldo haber.
- Obtén todos los elementos de las sucursales con más de 3 cuentas.
- Obtén todos los elementos de las sucursales con más de 3 cuentas del tipo AHORRO.
- Obtén los nodos del director y la población de las sucursales con más de 3 cuentas.
- Obtén el número de sucursales cuya población sea Madrid.
- Obtén por cada sucursal, su código y la suma de las aportaciones de las cuentas del tipo PENSIONES.

- Obtén los nodos número de cuenta, nombre de cuenta y el saldo haber de las cuentas con saldo haber mayor de 10000.
- Obtén por cada sucursal con más de 3 cuentas del tipo AHORRO, su código y la suma del saldo debe de esas cuentas.

5.3.3 AXIS XPATH

Un AXIS o eje, especifica la dirección que se va a evaluar, es decir, si nos vamos a mover hacia arriba en la jerarquía o hacia abajo, si va a incluir el nodo actual o no, es decir, define un conjunto de nodos relativo al nodo actual. Los nombres de los ejes son los siguientes:

NOMBRE DE AXIS	RESULTADO
ancestor	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual
ancestor-or-self	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual y el nodo actual en sí
attribute	Selecciona los atributos del nodo actual
child	Selecciona los hijos del nodo actual
descendant	Selecciona los descendientes (hijos, nietos, etc) del nodo actual
descendant-or-self	Selecciona los descendientes (hijos, nietos, etc) del nodo actual y el nodo actual en sí
following	Selecciona todo el documento después de la etiqueta de cierre del nodo actual
following-sibling	Selecciona todos los hermanos que siguen al nodo actual
parent	Selecciona el padre del nodo actual
self	Selecciona el nodo actual

La sintaxis para utilizar ejes es la siguiente: `Nombre_de_eje::nombre_nodo[expresión]`

En la ventana del ***Query Dialog*** del *Cliente de Administración de eXist*, se puede ver la traza de las consultas y en la traza podemos observar los ejes utilizados. Véase la Figura 5.14.

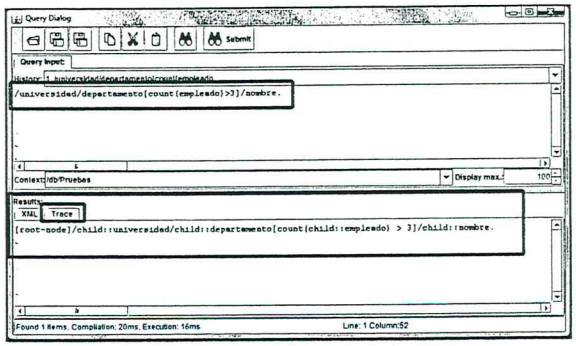


Figura 5.14. Ejes en la traza de ejecución de las consultas XPath.

Ejemplos:

- `/universidad/child::*`, es lo mismo que `/child::universidad/child::element()`. Devuelve todos los hijos de universidad, es decir los nodos de los departamentos.
- `/universidad/departamento/descendant::*`, devuelve los descendientes del nodo departamento, esto hace lo mismo: `/child::universidad/child::departamento/descendant::element()`
- `/universidad/departamento/descendant::empleado`, devuelve los nodos empleado descendientes de los nodos departamento.
- `/universidad/descendant::nombre`, devuelve todos los elementos nombre descendientes de universidad, tanto nombres de departamentos como de empleados. Si ponemos esto nos devuelve el texto del nombre: `data(/universidad/descendant::nombre)`.
- `/universidad/departamento/following-sibling::*`, selecciona todos los hermanos de departamento a partir del primero, siguiendo el orden en el documento.
  Si ponemos `/universidad/departamento[2]/following-sibling::*`, selecciona todos los hermanos de departamento a partir del segundo.
- `//empleado/following-sibling::node()`, selecciona todos los hermanos de los elementos empleado que encuentre en el contexto.
  En este caso `//empleado/following-sibling::empleado[@salario>2100]`, selecciona todos los hermanos de los elementos empleado que tienen el salario >2100.
- `//empleado[nombre="Ana García"]/following-sibling::*`, selecciona los nodos hermanos de Ana García.
- `//empleado[nombre="Ana García"]/following-sibling::empleado/nombre/text()`, selecciona los nombres de los empleados hermanos de Ana García.
- `//empleado[nombre="Ana García"]/following-sibling::empleado[puesto="Profesor"]/nombre/text()`, selecciona los nombres de los empleados hermanos de Ana García que son profesores.
- `//empleado/parent::departamento/nombre`, selecciona el nombre de los padres de los elementos empleado.
- `//empleado[nombre="Ana García"]/parent::departamento/nombre`, selecciona el nombre del padre de la empleada Ana García.
- `/descendant::departamento[1]`, selecciona los descendientes del departamento que ocupa la posición 3 en el documento.
- `/child::universidad/child::departamento[count(child::empleado)>3]`, es lo mismo que `/universidad/departamento[count(empleado)>3]`, obtiene los departamentos con más de 3 empleados.
- `/child::universidad/child::departamento/child::nombre`, obtiene las etiquetas con los nombres de los departamentos. Es lo mismo que `/universidad/departamento/nombre`, y que `data(/universidad/departamento/nombre)`.
- `/child::universidad/child::departamento/child::nombre/child::text()`, obtiene los nombres de los departamentos.
- `/child::universidad/child::departamento[attribute::tipo="B"][count(child::empleado) >= 2]/child::nombre/child::text()`, devuelve el nombre de los departamentos de tipo B y con 2 o más empleados. Es lo mismo que poner `/universidad/departamento[@tipo="B" and count(empleado)>=2]/nombre/text()`.

5.3.4 CONSULTAS XQUERY

Una consulta en XQuery es una expresión que lee datos de uno o más documentos en XML y devuelve como resultado otra secuencia de datos en XML, en la Figura 5.15 se ve el procesamiento básico de una consulta XQUERY. XQuery contiene a XPath, toda expresión de consulta en XPath es válida y devuelve el mismo resultado en XQuery. Xquery nos va a permitir:

- Seleccionar información basada en un criterio específico.
- Buscar información en un documento o conjunto de documentos.
- Unir datos desde múltiples documentos o colección de documentos.
- Organizar, agrupar y resumir datos.
- Transformar y reestructurar datos XML en otro vocabulario o estructura.
- Desempeñar cálculos aritméticos sobre números y fechas.
- Manipular cadenas de caracteres a formato de texto.

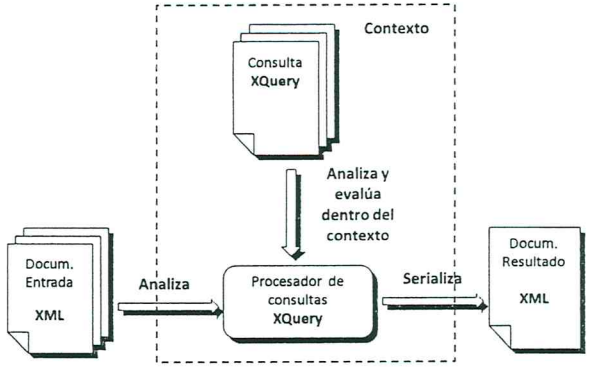


Figura 5.15. Procesamiento de una consulta XQuery.

En XQuery las consultas siguen la norma **FLWOR** (leído como flower), corresponde a las siglas de ***For, Let, Where, Order*** y ***Return***. Permite a diferencia de XPat manipular, transformar y organizar los resultados de las consultas. La sintaxis general de una estructura FLWOR es esta:

```
for <variable> in <expresión XPath>
let <variables vinculadas>
where <condición XPath>
order by <expresión>
return <expresión de salida>
```

- **For:** se usa para seleccionar nodos y almacenarlos en una variable, similar a la cláusula from de SQL. Dentro del for escribimos una expresión XPath que seleccionará los nodos. Si se especifica más de una variable en el for se actúa como producto cartesiano. Las variables comienzan con **$**.
  Las consultas XQuery deben llevar obligatoriamente una orden **Return,** donde indicaremos lo que queremos que nos devuelva la consulta. Por ejemplo estas consultas devuelve la primera los elementos EMP_ROW y la segunda los apellidos de los empleados. Unas escritas en XQuery y las otras en XPath:

XQUERY	XPATH
`for $emp in /EMPLEADOS/EMP_ROW` `return $emp`	`/EMPLEADOS/EMP_ROW`
`for $emp in /EMPLEADOS/EMP_ROW` `return $emp/APELLIDO`	`/EMPLEADOS/EMP_ROW/APELLIDO`

- **Let:** permite que se asignen valores resultantes de expresiones XPath a variables para simplificar la representación. Se pueden poner varias líneas let una por cada variable o separar las variables por comas.

  En el siguiente ejemplo se crean 2 variables, el APELLIDO del empleado se guarda en ***$nom***, y el OFICIO en ***$ofi***. La salida sale ordenada por OFICIO y se crea una etiqueta <APE_OFI> </APE_OFI> que incluye el nombre y el oficio concatenado. Se utilizarán las llaves en el return {} para añadir el contenido de las variables. Véase el ejemplo:

```
for $emp in /EMPLEADOS/EMP_ROW
let $nom:=$emp/APELLIDO, $ofi :=$emp/OFICIO
order by $emp/OFICIO
return <APE_OFI>{concat($nom,' ',$ofi)}</APE_OFI>
```

La cláusula let se puede utilizar sin for, prueba el siguiente caso y observa la diferencia:

SIN FOR	CON FOR
`let $ofi := /EMPLEADOS/EMP_ROW/OFICIO` `return <OFICIOS>{$ofi}</OFICIOS>`	`for $ofi in /EMPLEADOS/EMP_ROW/OFICIO` `return <OFICIOS>{$ofi}</OFICIOS>`
La cláusula let vincula la variable $ofi con todo el resultado de la expresión. En este caso vincula todos los oficios creando un elemento <OFICIOS> con todos ellos.	La cláusula for vincula la variable $ofi con cada nodo oficio que encuentre en la colección de datos, creando una elemento por cada oficio. Por eso aparece la etiqueta <OFICIOS> para cada oficio.

- **Where:** filtra los elementos, eliminando todos los valores que no cumplan las condiciones dadas.
- **Order by:** ordena los datos según el criterio dado.
- **Return:** construye el resultado de la consulta en XML, se pueden añadir etiquetas XML a la salida, si añadimos etiquetas los datos a visualizar los encerramos entre llaves {}. Además en el return se pueden añadir ***condicionales usando if-then-else*** y así tener más versatilidad en la salida.
  Hay que tener en cuenta que la cláusula else es obligatoria y debe aparecer siempre en la expresión condicional, se debe a que toda expresión XQuery debe devolver un valor. Si no existe ningún valor a devolver al no cumplirse la cláusula if, devolvemos una secuencia vacía con else ().

El siguiente ejemplo devuelve los departamentos de tipo A encerrados en una etiqueta:

```
for $dep in /universidad/departamento
return if ($dep/@tipo='A')
  then <tipoA>{data($dep/nombre)}</tipoA>
  else ()
```

Utilizaremos la función **data()** para extraer el contenido en texto de los elementos. También se utiliza data() para extraer el contenido de los atributos p.ej. esta consulta XPath **`//empleado/@salario`** es errónea, pues salario no es un nodo, pero esta otra consulta **`data(//empleado/@salario)`** devuelve los salarios.

Dentro de las asignaciones **`let`** en las consultas XQuery podremos utilizar expresiones del tipo **`let $var:=//empleado/@salario`** esto no da error, pero si queremos extraer los datos pondremos **`let $var:=data(//empleado/@salario)`** o **`return data($var)`**

Ejemplos:

`for $emp in /EMPLEADOS/EMP_ROW`
`order by $emp/APELLIDO`
`return if ($emp/OFICIO='DIRECTOR')`
`then <DIRECTOR>{$emp/APELLIDO/text()}</DIRECTOR>`
`else <EMPLE>{data($emp/APELLIDO)}</EMPLE>`
	Devuelve los nombres de los empleados, los que son directores entre las etiquetas <DIRECTOR> </DIRECTOR> y los que no lo son entre las etiquetas <EMPLE></EMPLE>.
	
	`<EMPLE>ALONSO</EMPLE>`
	`<EMPLE>ARROYO</EMPLE>`
	`<DIRECTOR>CEREZO</DIRECTOR>`
	`<EMPLE>FERNANDEZ</EMPLE>`
	`<EMPLE>GIL</EMPLE>`
	`<DIRECTOR>JIMENEZ</DIRECTOR>`
	. . . . .

`for $de in doc('file:///D:/XML /prueba-xquery/NUEVOS_DEP.xml')/NUEVOS_DEP/DEP_ROW return $de`	Devuelve los nodos DEP_ROW de un documento ubicado en una carpeta del disco duro.

`for $prof in /universidad/departamento[@tipo='A']/empleado`
`let $profe:= $prof/nombre,`
`$puesto:= $prof/puesto`
`where $puesto='Profesor'`
`return $profe`
	Obtiene los nombres de empleados de los departamentos de tipo A, cuyo puesto es Profesor. Esto hace lo mismo:
	
	`for $prof in /universidad/departamento[@tipo='A']/empleado`
	`where $prof/puesto='Profesor'`
	`return $prof/nombre`
	
	El resultado es:
	
	`<nombre>Alicia Martín</nombre>`
	`<nombre>Mª Jesús Ramos</nombre>`
	`<nombre>Pedro Paniagua</nombre>`

`for $dep in /universidad/departamento`
`return if ($dep/@tipo='A')`
`then <tipoA>{data($dep/nombre)}</tipoA>`
`else <tipoB>{data($dep/nombre)}</tipoB>`
	Devuelve el nombre de departamento encerrado entre las etiquetas <tipoA></tipoA>, si es del tipo = A, y <tipoB></tipoB>, si no lo es.
	
	`<tipoA>Informática</tipoA>`
	`<tipoA>Matemáticas</tipoA>`
	`<tipoB>Análisis</tipoB>`

`for $dep in /universidad/departamento`
`let $nom:= $dep/empleado`
`return <depart>{data($dep/nombre)}`
`<emple>{count($nom)}</emple></depart>`
	Obtiene los nombres de departamento y los empleados que tiene entre etiquetas:
	
	`<depart>Informática<emple>2</emple></depart>`
	`<depart>Matemáticas<emple>4</emple></depart>`
	`<depart>Análisis<emple>2</emple></depart>`

`for $dep in /universidad/departamento`
`let $emp:= $dep/empleado`
`let $sal:= $dep/empleado/@salario`
`return <depart>{data($dep/nombre)}`
`<emple>{count($emp)}</emple><medsal>`
`{avg($sal)}</medsal></depart>`
	Obtiene los nombres de departamento, los empleados que tiene y la media del salario entre etiquetas:
	
	`<depart>Informática<emple>2</emple>`
	`<medsal>2150</medsal></depart>`
	`<depart>Matemáticas<emple>4</emple>`
	`<medsal>2200</medsal></depart>`
	`<depart>Análisis<emple>2</emple>`
	`<medsal>2050</medsal></depart>`

5.3.5 OPERADORES Y FUNCIONES MÁS COMUNES EN XQUERY

Las funciones y operadores soportados por XQuery prácticamente son los mismos que los soportados por XPath. Soporta operadores y funciones matemáticas, de cadenas, para el tratamiento de expresiones regulares, comparaciones de fechas y horas, manipulación de nodos XML, manipulación de secuencias, comprobación y conversión de tipos y lógica booleana. Los operadores y funciones más comunes se muestran en la siguiente tabla.

- Matemáticos: +, -, *, div (se utiliza div en lugar de la /), idiv(es la división entera), mod.
- Comparación: =, !=, <, >, <=, >=, not().
- Secuencia: union (|), intersect, except.
- Redondeo: floor(), ceiling(), round().
- Funciones de agrupación: count(), min(), max(), avg(), sum().
- Funciones de cadena: concat(), string-length(),starts-with(), ends-with(), substring(), upper-case(), lower-case(), string().
- Uso general: ***distinct-values***() extrae los valores de una secuencia de nodos y crea una nueva secuencia con valores únicos, eliminando los nodos duplicados. ***empty***() devuelve cierto cuando la expresión entre paréntesis está vacía. Y ***exists***() devuelve cierto cuando una secuencia contiene, al menos, un elemento.
- Los comentarios en XQuery van encerrados entre caras sonrientes: (: Esto es un comentario :)

Ejemplos utilizando el documento EMPLEADOS.xml:

– Los nombres de oficio que empiezan por P.

```
for $ofi in /EMPLEADOS/EMP_ROW/OFICIO
where starts-with(data($ofi),'P')
return $ofi
```

SALIDA:

```
<OFICIO>PRESIDENTE</OFICIO>
```

– Obtiene los nombres de oficio y los empleados de cada oficio. Utiliza la función distinct-values para devolver los distintos oficios.

```
for $ofi in distinct-values(/EMPLEADOS/EMP_ROW/OFICIO)
let $cu:=count(/EMPLEADOS/EMP_ROW[OFICIO =$ofi])
return concat($ofi,' = ',$cu)
```

SALIDA:

```
EMPLEADO = 4
VENDEDOR = 4
DIRECTOR = 3
ANALISTA = 2
PRESIDENTE = 1
```

– Obtiene el número de empleados que tiene cada departamento y la media de salario redondeada:

```
for $dep in distinct-values(/EMPLEADOS/EMP_ROW/DEPT_NO)
let $cu:=count(/EMPLEADOS/EMP_ROW[DEPT_NO =$dep])
let $sala:= round(avg(/EMPLEADOS/EMP_ROW[DEPT_NO =$dep]/SALARIO))
return concat('Departamento: ',$dep,'. Num emples = ',$cu, '. Media salario =
',$sala)
```

SALIDA:

```
Departamento: 20. Num emples = 5. Media salario = 2274
Departamento: 30. Num emples = 6. Media salario = 1736
Departamento: 10. Num emples = 3. Media salario = 2892
```

Si se desea devolver el resultado entre etiquetas pondremos en el return (p.ej):

```
return <depart><cod>{$dep}</cod><emples>{$cu}</emples><medsal>{$sala}</medsal></
depart>
```

```
<depart><cod>20</cod><emples>5</emples><medsal>2274</medsal></depart>
<depart><cod>30</cod><emples>6</emples><medsal>1736</medsal></depart>
<depart><cod>10</cod><emples>3</emples><medsal>2892</medsal></depart>
```

Actividad 3: Utilizando el documento ***productos.xml.*** Realiza las siguientes consultas XQuery:

- Obtén por cada zona el número de productos que tiene.
- Obtén la denominación de los productos entres las etiquetas <zona10></zona10> si son del código de zona 10, <zona20></zona20> si son de la zona 20, <zona30></zona30> si son de la 30 y <zona40></zona40> si son de la 40.
- Obtén por cada zona la denominación del o de los productos más caros.
- Obtén la denominación de los productos contenida entre las etiquetas <placa></placa> para los productos en cuya denominación aparece la palabra *Placa Base*, <memoria></memoria>, para los que contienen a la palabra *Memoria* <micro></micro>, para los que contienen la palabra *Micro* y <otros></otros> para el resto de productos.

Utilizando el documento ***sucursales.xml.*** Realiza las siguientes consultas XQuery:

- Devuelve el código de sucursal y el número de cuentas que tiene de tipo AHORRO y de tipo PENSIONES
- Devuelve por cada sucursal el código de sucursal, el director, la población, la suma del total debe y la suma del total haber de sus cuentas.
- Devuelve el nombre de los directores, el código de sucursal y la población de las sucursales con más de 3 cuentas.
- Devuelve por cada sucursal , el código de sucursal y los datos de las cuentas con más saldo debe.
- Devuelve la cuenta del tipo PENSIONES que ha hecho más aportación.

5.3.6 CONSULTAS COMPLEJAS CON XQUERY

Dentro de las consultas XQuery podremos trabajar con varios documentos xml para extraer su información, podremos incluir tantas sentencias for como se deseen, incluso dentro del return. Además podremos añadir, borrar e incluso modificar elementos, bien generando un documento xml nuevo o utilizando las sentencias de actualización de eXist. A continuación se muestran ejemplos de diversa complejidad:

- **Joins de documentos.**

  - Visualizar por cada empleado del documento *empleados.xml*, su apellido, su número de departamento y el nombre del departamento que se encuentra en el documento *departamentos.xml*

```
for $emp in (/EMPLEADOS/EMP_ROW)
let $emple:= $emp/APELLIDO
let $dep:= $emp/DEPT_NO
let $dnom:= (/departamentos/DEP_ROW[DEPT_NO =$dep]/DNOMBRE)
return <res>{$emple, $dep, $dnom} </res>
```



```
SALIDA:
<res>
    <APELLIDO>SANCHEZ</APELLIDO>
    <DEPT_NO>20</DEPT_NO>
    <DNOMBRE>INVESTIGACION</DNOMBRE>
</res>
<res>
    <APELLIDO>ARROYO</APELLIDO>
    <DEPT_NO>30</DEPT_NO>
    <DNOMBRE>VENTAS</DNOMBRE>
</res>
<res>
    <APELLIDO>SALA</APELLIDO>
    <DEPT_NO>30</DEPT_NO>
    <DNOMBRE>VENTAS</DNOMBRE>
</res>
. . . . . .
```

  - Utilizando los documentos *departamentos.xml* y *empleados.xml*, obtener por cada departamento, el nombre de departamento, el número de empleados, y la media de salario.

```
for $dep in /departamentos/DEP_ROW
let $d:=$dep/DEPT_NO
let $tot:=sum(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/SALARIO)
let $cu:=count(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/EMP_NO)
return <resul>{$dep/DNOMBRE}<sumsalario>{$tot}</sumsalario><totemple>{$cu}</
totemple></resul>
```

```
SALIDA:
<resul>
    <DNOMBRE>CONTABILIDAD</DNOMBRE>
    <sumasalario>8675</sumasalario>
    <numemple>3</numemple>
</resul>
<resul>
    <DNOMBRE>INVESTIGACION</DNOMBRE>
    <sumasalario>11370</sumasalario>
    <numemple>5</numemple>
</resul>
<resul>
    <DNOMBRE>VENTAS</DNOMBRE>
    <sumasalario>10415</sumasalario>
    <numemple>6</numemple>
</resul>
<resul>
    <DNOMBRE>PRODUCCION</DNOMBRE>
    <sumasalario>0</sumasalario>
    <numemple>0</numemple>
</resul>
```

  - Convertir la salida de la consulta anterior, de manera que el total salario, y el total empleados, sean atributos de cada departamento. Hacemos que las salida que se cree sea una concatenación de los datos a obtener:

```
for $dep in /departamentos/DEP_ROW
let $d:=$dep/DEPT_NO
let $tot:=sum(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/SALARIO)
let $cu:=count(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/EMP_NO)
return concat('<departamento sumasalario="',$tot,'" totemple="',$cu,'">',
   data($dep/DNOMBRE),'</departamento>')
```

SALIDA:

```
<departamento sumasalario="8675" totemple="3">CONTABILIDAD</departamento>
<departamento sumasalario="11370" totemple="5">INVESTIGACION</departamento>
<departamento sumasalario="10415" totemple="6">VENTAS</departamento>
<departamento sumasalario="0" totemple="0">PRODUCCION</departamento>
```

- Utilizando los documentos *departamentos.xml* y *empleados.xml*, obtener por cada departamento, el nombre de empleado que más gana.

```
for $emp in /EMPLEADOS/EMP_ROW
let $d:=$emp/DEPT_NO, $nom:=$emp/APELLIDO, $sal:=$emp/SALARIO
let $ndep:=(/departamentos/DEP_ROW[DEPT_NO=$d]/DNOMBRE)
let $salmax:= max(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/SALARIO)
return
if ($sal=$salmax)
then
<depart>{data($ndep)}<salmax>{data($sal)}</salmax><emple>{data($nom)}</emple></
depart>
else ()
```

SALIDA:

```
<depart>VENTAS <salmax>3005</salmax><emple>NEGRO</emple></depart>
<depart>INVESTIGACION<salmax>3000</salmax><emple>GIL</emple></depart>
<depart>CONTABILIDAD<salmax>4100</salmax><emple>REY</emple></depart>
<depart>INVESTIGACION<salmax>3000</salmax><emple>FERNANDEZ</emple></depart>
```

Actividad 4: Sube a la colección *Pruebas* el documento *zonas.xml*, contiene información de las zonas donde se venden los productos del documento *productos.xml*. Utilizando estos dos documentos realiza las siguientes consultas XQuery:

- Obtén los datos denominación, precio y nombre de zona de cada producto, ordenado por nombre de zona.
- Obtén por cada zona, el nombre de zona y el número de productos que tiene.
- Obtén por cada zona, el nombre de la zona, su código y el nombre del producto con menos stock actual.

- **Utilización de varios for.** La utilización de varios for es muy útil para consultas en documentos XML anidados y también cuando utilizamos varios documentos unidos por una cláusula *where* como una combinación de tablas en SQL. Ejemplos:

- Esta consulta visualiza por cada departamento del documento *universidad.xml*, el número de empleados que hay en cada puesto de trabajo. Utilizamos un for para obtener los nodos departamento y el segundo for para obtener los distintos puestos de cada departamento.

```
for $dep in /universidad/departamento
for $pue in distinct-values($dep/empleado/puesto)
let $cu:=count($dep/empleado[puesto=$pue])
return <depart>{data($dep/nombre)}<puesto>{data($pue)}</puesto><profes>{$cu}</
profes></depart>
```

SALIDA:

```
<depart>Informática<puesto>Asociado</puesto><profes>1</profes></depart>
<depart>Informática<puesto>Profesor</puesto><profes>1</profes></depart>
<depart>Matemáticas<puesto>Técnico</puesto><profes>1</profes></depart>
<depart>Matemáticas<puesto>Profesor</puesto><profes>2</profes></depart>
<depart>Matemáticas<puesto>Tutor</puesto><profes>1</profes></depart>
<depart>Análisis<puesto>Asociado</puesto><profes>2</profes></depart>
```

- Esta consulta visualiza por cada departamento del documento *universidad.xml*, el salario máximo y el empleado que tiene ese salario. El primer for obtiene los nodos departamento y el segundo for los empleados de cada departamento. Para sacar el máximo en la salida preguntamos si el salario es el máximo.

```
for $dep in /universidad/departamento
for $emp in $dep/empleado
let $emple:= $emp/nombre
let $sal:= $emp/@salario
return if ($sal = $dep/max(empleado/@salario))
       then
            <depart>{data($dep/nombre)} <salamax>{data($sal)}</salamax>
            <empleado>{data($emple)}</empleado></depart>
       else ()
```

También se pueden poner los dos for de la siguiente manera:

```
for $dep in /universidad/departamento, $emp in $dep/empleado
```

SALIDA:

```
<depart>Informática
  <salamax>2300</salamax>
  <empleado>Alicia Martín</empleado>
</depart>
<depart>Matemáticas
  <salamax>2500</salamax>
  <empleado>Antonia González</empleado>
</depart>
<depart>Análisis
  <salamax>2200</salamax>
  <empleado>Mario García</empleado>
</depart>
```

- Esta consulta visualiza por cada puesto del documento *universidad.xml*, el empleado con salario máximo y ese salario. El primer for obtiene los distintos puestos de trabajo y el

segundo for obtiene los empleados que tienen ese puesto de trabajo. En el if se pregunta si el salario del empleado es el salario máximo de los empleados del oficio del primer for.

```
for $pue in distinct-values (/universidad/departamento/empleado/puesto)
for $emp in /universidad/departamento/empleado[puesto=$pue]
let $sal:= $emp/@salario
let $nom:= $emp/nombre
return
if ($sal = max(/universidad/departamento/empleado[puesto=$pue]/@salario))
  then
    <puesto>{data($pue)}<maxsalario>{data($sal)}</maxsalario><emple>{data($nom)}</
emple>
    </puesto>
  else ()

SALIDA:
<puesto>Asociado<maxsalario>2200</maxsalario><emple>Mario García</emple></puesto>
<puesto>Profesor<maxsalario>2300</maxsalario><emple>Alicia Martín</emple></puesto>
<puesto>Profesor<maxsalario>2300</maxsalario><emple>Pedro Paniagua</emple></puesto>
<puesto>Técnico<maxsalario>1900</maxsalario><emple>Ana García</emple></puesto>
<puesto>Tutor<maxsalario>2500</maxsalario><emple>Antonia González</emple></puesto>
```

También podemos resolver la consulta utilizando un solo for, de la siguiente manera:

```
for $emp in  (/universidad/departamento/empleado)
let $pue:=$emp/puesto
let $sal:= $emp/@salario
let $nom:= $emp/nombre
order by $pue
return
if ($sal = max(/universidad/departamento/empleado[puesto=$pue]/@salario))
  then
<puesto>{data($pue)}<maxsalario>{data($sal)}</maxsalario><emple>{data($nom)}</emple>
    </puesto>
  else ()
```

La primera consulta del apartado anterior la podemos escribir con dos for y el where:

```
for $emp in (/EMPLEADOS/EMP_ROW), $dep in /departamentos/DEP_ROW
let $emple:= $emp/APELLIDO
let $d:= $emp/DEPT_NO
where data($d) = data($dep/DEPT_NO)
return <res>{$emple, $d} {$dep/DNOMBRE} </res>
```

Actividad 5: Utiliza el documento ***sucursales.xml*** para realizar las siguientes consultas XQuery:

- Obtén por cada sucursal el mayor saldo haber y el nombre de la cuenta que tiene ese saldo.
- Obtén por cada sucursal el nombre de la cuenta del tipo AHORRO cuyo saldo debe sea el máximo. Saca también el máximo.

Utiliza los documentos ***productos.xml y zonas.xml***

- Visualiza los nombres de productos con su nombre de zona. Utiliza dos for en la consulta.
- Visualiza los nombres de productos con stock_minimo > 5. su código de zona, su nombre y el director de esa zona. Utiliza dos for en la consulta.

- **Altas, bajas y borrado de nodos en documentos XML.**
  - Esta consulta va a generar una salida en la que se van a incluir todos los nodos del documento *departamentos.xml*, más un nuevo nodo a insertar. En la variable *$nuevo* añadimos el nodo a insertar, en *$titulo* guardamos el nodo TITULO del documento *departamentos.xml*. En el return creamos una etiqueta llamada <departamentos> e incluimos el título, todos los nodos *DEP_ROW* de *departamentos.xml* (estos se obtienen con el for) y el nodo nuevo. Si se desea guardar la salida generada se pulsa el botón ***Write result to file*** del ***Query Dialog*** de eXist, véase la Figura 5.16.

Figura 5.16. Query Dialog, guardar el resultado de una consulta.

  - En la siguiente consulta se elimina el nodo cuyo número de departamento es el 10. Utilizamos el mismo documento. Se trata de generar una consulta que obtenga todos los nodos del documento menos el nodo con ese número de departamento. Utilizamos where para no seleccionar el departamento 10.

```
let $titulo:=/departamentos/TITULO
return <departamentos>{$titulo}
                      {for $dep in  (/departamentos/DEP_ROW)
                         where data(($dep/DEPT_NO))!=10
                       return  $dep
                      }
     </departamentos>
```

  - En la siguiente consulta se va a actualizar el departamento cuyo número de departamento es el 20. Vamos a cambiar la localidad (LOC) por GRANADA. Lo que hacemos es sacar los

datos del nodo a modificar, para luego insertar ese nodo modificado, en nuestro caso sacamos el nombre del departamento (el número y la localidad no se sacan, pues no se necesitan). Y luego se crea el nodo modificado en *$modif* con todas las etiquetas; de manera que cuando se van generando las etiquetas con los departamentos al llegar al departamento a modificar (20) se devuelve el nodo *$modif*, y si no es el 20, se visualiza el nodo *$dep*.

```
let $titulo:=/departamentos/TITULO
let $dnom:=/departamentos/DEP_ROW[DEPT_NO=20]/DNOMBRE/text()
let $modif:=<DEP_ROW><DEPT_NO>20</DEPT_NO><DNOMBRE>{$dnom}</DNOMBRE><LOC>GRANADA</
            LOC> </DEP_ROW>
return <departamentos>{$titulo}
                      {for $dep in  (/departamentos/DEP_ROW)
                        return
                           if ($dep/DEPT_NO/text()=20) then  $modif
                           else  $dep
                    }
      </departamentos>
```

- En la siguiente consulta se van a actualizar los salarios de los empleados del departamento 10, se les sube a todos la cantidad de 200. Lo que se hace para todos los empleados es sacar los nodos a variables, de manera que si el empleado es del departamento 10, se creará un nodo nuevo <EMP_ROW> con los datos de sus nodos y el salario actualizado. Si el empleado no es del departamento 10 se devuelve el nodo EMP_ROW leído.

```
for $em in /EMPLEADOS/EMP_ROW
let $no:=$em/EMP_NO, $ape:=$em/APELLIDO, $ofi:=$em/OFICIO, $dir:=$em/DIR
let $fec:=$em/FECHA_ALT, $de:=$em/DEPT_NO, $sal:=$em/SALARIO
let $salnue:=200+ number($em/SALARIO/text())
return
     if ($em/DEPT_NO/text()=10) then
      <EMP_ROW>{$no, $ape,$ofi,$dir,$fec}<SALARIO>{$salnue}</SALARIO>{$de}</EMP_ROW>
    else  $em
```

- En la siguiente consulta se han realizado los cambios necesarios para que la salida sea el documento *empleados.xml* pero con el salario actualizado.

```
let $titulo:=/EMPLEADOS/TITULO
return <EMPLEADOS>
         {$titulo}
         {
            for $em in /EMPLEADOS/EMP_ROW
             let $no:=$em/EMP_NO, $ape:=$em/APELLIDO, $ofi:=$em/OFICIO, $dir:=$em/DIR
             let $fec:=$em/FECHA_ALT, $de:=$em/DEPT_NO, $sal:=$em/SALARIO
             let $salnue:=200+ number($em/SALARIO/text())
            return
                if ($em/DEPT_NO/text()=10) then
                  <EMP_ROW>{$no,$ape,$ofi,$dir,$fec}<SALARIO>{$salnue}</SALARIO>{$de}
                  </EMP_ROW>
                else  $em
         }
       </EMPLEADOS>
```


Actividad 6: Crea un nuevo documento XML llamado *productos_nuevo.xml*, que incorpore el siguiente producto: cod_prod: 1023, denominación: HD externo Seagate expansión 250GB, precio: 70, stock_actual: 100, stock_minimo: 20, y el código de zona tiene que ser el código de zona correspondiente a Andalucía.

Crea un nuevo documento XML llamado *productos_30.xml*, con el <TITULO>DATOS DE LOS PRODUCTOS DE LA ZONA 30</TITULO>, este documento solo debe contener los productos cuyo código de zona sea 30.

Crea una consulta que visualice todo el documento de *productos.xml* pero con los siguientes cambios:

- Sube el precio de cada producto a un 3% más caro.
- Añade 10 unidades a los stock_actual de todos los productos.

- **Sentencias de actualización de eXist.**
  Estas sentencias permiten hacer altas, bajas y modificaciones de nodos y elementos en documentos xml. Se pueden usar las sentencias de actualización en cualquier punto pero si se utiliza en la cláusula RETURN de una sentencia FLWOR, el efecto de la actualización es inmediato. Todas las sentencias de actualización comienzan con la palabra UPDATE y a continuación la instrucción. Son las siguientes:

  - **Insert,** se utiliza para insertar nodos. El lugar de inserción se especifica con: **into** (el contenido se añade como último hijo de los nodos especificados); **following** (el contenido se añade inmediatamente después de los nodos especificados), o **preceding** (el contenido se añade antes de los nodos especificados). El formato es:

```
update insert ELEMENTO into EXPRESION
update insert ELEMENTO following EXPRESION
update insert ELEMENTO preceding EXPRESION
```

update insert <zona><cod_zona>50</cod_zona> <nombre>Madrid-OESTE </nombre> <director>Alicia Ramos Martín</director> </zona> into /zonas	Inserta una zona en *zonas.xml*, en la última posición.
update insert <cuenta tipo="PENSIONES"><nombre>Alberto Morales </nombre><numero>30302900</numero> <aportacion>5000</aportacion></cuenta> into /sucursales/sucursal[@codigo="SUC1"]	Inserta una cuenta en el documento *sucursales.xml* del tipo PENSIONES a la sucursal SUC1.
for $de in doc('file:///D:/XML/pruebaxquery/NUEVOS_DEP.xml') /NUEVOS_DEP/DEP_ROW return update insert $de into /departamentos	Inserta en el documento departamentos de la BD los nodos DEP_ROW del documento externo NUEVOS_DEP.xml ubicado en la carpeta D:/XML/pruebaxquery/

- **Replace**, sustituye el nodo especificado en NODO con VALOR NUEVO (ver formato). NODO debe devolver un único ítem: si es un elemento, VALOR_NUEVO debe ser también un elemento. Si es un nodo de texto o atributo su valor será actualizado con la concatenación de todos los valores de VALOR_NUEVO.

```
update replace NODO with VALOR_NUEVO
```

`update replace /zonas/zona[cod_zona=50]/director with <directora>Pilar Martín Ramos</directora>`	Cambia la etiqueta director de la zona 50 y su contenido, en el documento *zonas.xml*
`update replace /departamentos/DEP_ROW[DEPT_NO=10] with <DEPT_ROW><DEPT_NO>10</DEPT_NO><DNOMBRE>NUEVO10</DNOMBRE> <LOC>TALAVERA</LOC></DEPT_ROW>`	Cambia el nodo completo DEP_ROW del departamento 10, por los nuevos datos y las etiquetas que escribamos.

- **Value**, actualiza el valor del nodo especificado en NODO con VALOR_NUEVO. Si NODO es un nodo de texto ó atributo su valor será actualizado con la concatenación de todos los valores de VALOR_NUEVO.

```
update value NODO with 'VALOR NUEVO'
```

`update value /EMPLEADOS/EMP_ROW[EMP_NO= 7369]/APELLIDO with 'Alberto Montes Ramos'`	Cambia el apellido del empleado 7369, del documento *empleados.xml*
`update value /sucursales/sucursal[@codigo='SUC3']/cuenta[1]/@tipo with 'NUEVOTIPO'`	Cambia el atributo tipo de la primera cuenta de la sucursal SUC3, del documento *sucursales.xml.*
`for $em in /EMPLEADOS/EMP_ROW[DEPT_NO=10] let $sal := $em/SALARIO return update value $em/SALARIO with data($sal)+200`	Cambia el salario de los empleados del departamento10, del documento empleados.xml, subirles 200.

- **Delete**, elimina los nodos indicados en la expresión: **`update delete expr`**

`update delete /zonas/zona[cod_zona=50]`	Elimina la zona con código 50, en el documento *zonas.xml*

- **Rename.** Renombra los nodos devueltos en NODO (debe devolver una relación de nodos o atributos) por el NUEVO_NOMBRE.

```
update rename NODO as NUEVO_NOMBRE
```

`update rename /EMPLEADOS/EMP_ROW as 'fila_emple'`	Cambia de nombre el nodo EMP_ROW del documento *empleados.xml*

Actividad 7: A partir del documento *universidad.xml.*

Añade un empleado al departamento que ocupa la posición 2. Los datos son el salario 2340, el puesto Técnico, y nombre Pedro Fraile.

Actualiza el salario de los empleados del departamento con código MAT1. Suma al salario 100.

Renombra el nodo DEP_ROW del documento *departamentos.xml por filadepar.*

5.4 ACCESO A EXIST DESDE JAVA

Ya se ha estudiado en la UNIDAD 1 como leer documentos XML, accediendo a su estructura y contenido utilizando los parser o analizadores **DOM** (*Modelo de Objetos de Documento*) y **SAX** (*API Simple para XML*). En este apartado vamos a ver distintas APIs que acceden a la BD eXist para procesar documentos XML.

5.4.1 LA API XML:DB PARA BASES DE DATOS XML

Desde el 2001 el grupo **XML:DB initiative** comenzó el desarrollo de una interfaz de aplicación (API), independiente del proveedor para bases de datos XML. Al igual que con otras aplicaciones la API simplemente define un conjunto de interfaces que cada proveedor implementa. Actualmente, la API está siendo utilizada por la mayoría de bases de datos. La API propuesta se basa en tres paquetes: **org.xmldb.api, org.xmldb.api.base** y **org.xmldb.api.modules.** Para saber más sobre estos paquetes y sus clases consulta el sitio web: http://xmldb-org.sourceforge.net/xapi/api/index.html.

Los componentes básicos empleados por el código XML: DB API son los drivers, las colecciones, los recursos y los servicios. Veamos cada uno de ellos:

- ***Los drivers*** son implementaciones de la interfaz de base de datos que encapsula la lógica de acceso a la base de datos XML. Los proporciona el proveedor del producto y debe ser registrado con el gestor de base de datos. Ejemplo:

```
String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
Class cl = Class.forName(driver); //Cargar del driver
Database database = (Database) cl.newInstance(); //Instancia de la BD
DatabaseManager.registerDatabase(database); //Registro del driver
```

- ***Una colección*** es un contenedor de recursos y otras sub-colecciones. La API define dos recursos diferentes: XMLResource y BinaryResource. Un XMLResource representa un documento XML o un fragmento del documento, seleccionados por la ejecución de una consulta XPath. Una vez utilizado el recurso se debe de cerrar. Ejemplo:

```
String URI="xmldb:exist://localhost:8080/exist/xmlrpc/db/Pruebas"; //Colección
String usu="admin"; //Usuario
```

```
String usuPwd="admin"; //Clave
Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
```

- ***Los servicios*** se solicitan para tareas como consultar una colección con XPath o la gestión de una colección. Ejemplo:

```
XPathQueryService servicio =
            (XPathQueryService) col.getService("XPathQueryService", "1.0");
ResourceSet result = servicio.query("for $em in /EMPLEADOS/EMP_ROW return $em");
```

Internamente, eXist no distingue entre las expresiones XPath y XQuery. *XPathQueryService* y *XQueryService* son lo mismo, aunque la segunda proporciona algunos métodos adicionales.

Para ejecutar la consulta llamaremos al método ***nombre_servicio.query(xpath),*** este método devuelve un ***ResourceSet***, que contiene el recurso, en el ejemplo anterior el resultado de la consulta es devuelto en *result*. El siguiente código lo escribiremos para recorrer el recurso *result* devuelto por la consulta anterior:

```
ResourceIterator i; //se utiliza para recorrer un set de recursos
i = result.getIterator();
if (!i.hasMoreResources()){
    System.out.println(" LA CONSULTA NO DEVUELVE NADA.");
  }
while (i.hasMoreResources()) {
    Resource r = i.nextResource();
    System.out.println((String) r.getContent());
}
```

Donde ***result.getIterator()*** nos da un iterador sobre el recurso, cada recurso contiene el valor seleccionado por la expresión XPath. El método ***getContent(),*** devuelve el contenido del recurso, en nuestro ejemplo devuelve la consulta y el tipo es String.

Para localizar si existe un documento en una colección utilizaremos este código:

```
Collection col= DatabaseManager.getCollection(URI, usu, usuPwd);
XMLResource res = null;
res = (XMLResource)col.getResource("empleados.xml");
if(res == null)
    System.out.println("NO EXISTE EL DOCUMENTO");
```

Para realizar un programa que consulte la BD eXist debemos incluir las siguientes librerías: *exist.jar, exist-optional.jar, xmldb.jar, xml-apis-1.3.04.jar, xmlrpc-client-3.1.1.jar, xmlrpc-common-3.1.1.jar* situadas en la instalación de eXist y *log4j-1.2.15.jar*.

En el siguiente ejemplo vemos cómo utilizar las clases vistas anteriormente, este ejercicio realiza una conexión con la BD para acceder al documento *empleados.xml* y obtener en XML los empleados del departamento 10.


```
import org.xmldb.api.*;
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;

public static void verempleados10(String[] args) throws XMLDBException  {
String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
Collection col = null; // Colección
String URI="xmldb:exist://localhost:8080/exist/xmlrpc/db/Pruebas"; //URI colección
String usu="admin"; //Usuario
String usuPwd="admin"; //Clave
try {
    Class cl = Class.forName(driver); //Cargar del driver
    Database database = (Database) cl.newInstance(); //Instancia de la BD
    DatabaseManager.registerDatabase(database); //Registro del driver
       } catch (Exception e) {
           System.out.println("Error al inicializar la BD eXist");
           e.printStackTrace();    }

col = DatabaseManager.getCollection(URI, usu, usuPwd);
if(col == null)
    System.out.println(" *** LA COLECCION NO EXISTE. ***");

XPathQueryService servicio = (XPathQueryService) col.getService("XPathQueryService",
"1.0");
ResourceSet result = servicio.query ("for $em in /EMPLEADOS/EMP_ROW[DEPT_NO=10] return
$em");
// recorrer los datos del recurso.
ResourceIterator i;
i = result.getIterator();
if (!i.hasMoreResources())
    System.out.println(" LA CONSULTA NO DEVUELVE NADA.");
while (i.hasMoreResources()) {
    Resource r = i.nextResource();
    System.out.println((String) r.getContent());
}
col.close(); //borramos
}// FIN verempleados10
```

Actividad 8: Realiza los cambios necesarios al ejercicio anterior para leer de teclado un departamento y visualizar sus empleados. Utiliza la entrada estándar. El código para la entrada estándar es el siguiente:

```
// import para la entrada estandard
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

. . . . . .
System.out.println("Teclea departamento:"); // se lee un dato de tipo cadena
String s = null;
try{
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    s = in.readLine();
```

```
    }catch(IOException e){
     System.out.println("Error al leer");
     e.printStackTrace();
    }
int dep=Integer.parseInt(s); // convertimos a numérico
```

Realiza un programa Java que inserte, elimine y modifique departamentos del documento *departamentos.xml*. Utiliza las *Sentencias de actualización de eXist*. Los datos se leerán de la entrada estándar de teclado. Haz que la función *main()* llame y ejecute los siguientes métodos (no devuelven nada):

- ***Insertadep()***, este método leerá de teclado un departamento, su nombre y su localidad, y deberá añadirlo al documento. Si el código de departamento existe visualiza que no se puede insertar porque ya existe.
- ***Borradep()***, este método leerá de teclado un departamento y deberá borrarlo si existe, si no existe visualiza que no se puede borrar porque ya existe.
- ***Modificadep()***, este método leerá de teclado un departamento, su nombre nuevo y la localidad nueva y deberá actualizar todos los datos si existe, si no existe visualiza que no se puede modificar porque ya existe.

5.4.1.1. OPERACIONES SOBRE COLECCIONES Y DOCUMENTOS

La API *XML:DB* nos va a permitir además de consultar documentos en la BD, crear y eliminar colecciones, y crear y eliminar documentos.

- **Crear una colección:** para crear una nueva colección, se llama al método ***createCollection*** del servicio ***CollectionManagementService***. El siguiente ejemplo crea la colección NUEVA_COLECCION dentro de la colección ***col:***

```
CollectionManagementService mgtService =
  (CollectionManagementService)col.getService("CollectionManagementService","1.0");
mgtService.createCollection("NUEVA_COLECCION");
```

- **Borrar una colección:** para borrar utilizamos el método `removeCollection`:

```
mgtService =
  (CollectionManagementService)col.getService("CollectionManagementService","1.0");
mgtService.removeCollection("NUEVA_COLECCION");
```

- **Crear un nuevo documento:** este ejemplo añade un documento nuevo a la colección ***col***, el documento se llama *NUEVOS_DEP.xml*, y se encuentra en nuestro disco, en la carpeta donde está el programa. Utilizamos el paquete *java.io.File*, para declarar el fichero a subir a la BD, y el método ***createResource*** para crear el recurso.

```
import java.io.File;
. . . . . .
```



```
File archivo= new File("NUEVOS_DEP.xml");
if(!archivo.canRead()) System.out.println("ERROR AL LEER EL FICHERO");
else
 { Resource nuevoRecurso = col.createResource(archivo.getName(), "XMLResource");
   nuevoRecurso.setContent(archivo); //comprueba si es un archivo
   col.storeResource(nuevoRecurso); }
```

- **Borrar un documento de la colección:** este ejemplo borra el documento creado anteriormente, comprueba si existe. Se utiliza el método ***removeResource***:

```
try
    {   Resource recursoParaBorrar = col.getResource("NUEVOS_DEP.xml");
        col.removeResource(recursoParaBorrar);
}catch(NullPointerException e)
  {         System.out.println("No se puede borrar. No se encuentra."); }
```

Actividad 9:

- Haz un programa Java que cree la colección GIMNASIO y suba los documentos que se encuentran en la carpeta ***ColeccionGimnasio***. Los documentos son los siguientes:
  - *socios_gim.xml* contiene información de los socios que asisten a hacer deporte en un Gimnasio.
  - *actividades_gim.xml* contiene información de las actividades que se pueden realizar en el Gimnasio. Hay 3 tipos de actividades:
    1- son actividades de libre horario, el socio no paga cuota adicional por ellas, por ejemplo: aparatos o piscina.
    2- representan actividades que se realizan en grupo, como por ejemplo: aerobic o pilates. El socio paga una cuota adicional de 2€ por cada hora que dedique a la actividad.
    3- representan actividades en las que se alquila un espacio, por ejemplo padel o tenis. El socio paga una cuota adicional de 4€ por cada hora que dedique a la actividad.
  - *actividades_gim.xml* contiene información de las actividades que se pueden realizar en el Gimnasio. Hay 3 tipos de actividades:
  - *Uso_gimnasio.xml*, contiene las actividades que realizan los socios en el Gimnasio durante el año, cada fila representa una actividad realizada por el socio con la fecha (dd/mm/yy), la hora de inicio (por ejemplo 17) y la hora de finalización (por ejemplo 18):
- A partir de esos documentos haz un método java para obtener por cada socio la cuota que tiene que pagar. Obtener el CODSOCIO y la CUOTA_FINAL.

  Esta CUOTA_FINAL será igual a la suma de la CUOTA_FIJA y las CUOTAS ADICIONALES que dependerán de las actividades realizadas por el socio.

El programa java debe crear un documento XML intermedio que obtenga la cuota adicional a obtener por cada actividad realizada por cada usuario, el documento debe contener estas etiquetas:

```
<datos><COD>xxxxx</COD><NOMBRESOCIO>xxxxxxxx</NOMBRESOCIO>
  <CODACTIV>xxxxxxx</CODACTIV><NOMBREACTIVIDAD>xxxxxxxxx</NOMBREACTIVIDAD>
  <horas>xxxx</horas><tipoact>xxxxx</tipoact><cuota_adicional>xxxxx</cuota_adicional>
</datos>
```

Añade el documento a la colección GIMNASIO. Una vez creado y añadido el documento, el programa java debe obtener la cuota final total, que será la suma de las cuotas adicionales de las actividades mas la cuota fija. Obtén las siguientes etiquetas:

```
<datos>
  <COD>xxxxxx</COD>
  <NOMBRESOCIO>xxxxxxxxxxx</NOMBRESOCIO><CUOTA_FIJA>xxxxxx</CUOTA_FIJA>
  <suma_cuota_adic>xxxxx</suma_cuota_adic><cuota_total>xxxxxxx</cuota_total>
</datos>
```

5.4.2 LA API XQJ (XQUERY)

La API XQJ es una propuesta de estandarización de interfaz Java para el acceso a bases de datos XML nativas basado en el modelo de datos XQuery. El objetivo es conseguir un método sencillo y estable de acceso a bases de datos XML nativos, este estándar cumple los siguientes objetivos:

- Estándar independiente del fabricante.
- Soporte al estándar XQuery 1.0
- Facilidad de uso.
- Potenciar el uso de esta tecnología en la comunidad Java.

Al igual que en JDBC la filosofía gira en torno al origen de datos y la conexión a este, y partiendo de la conexión poder lanzar peticiones al sistema.

Para descargar la API accedemos a la URL: http://xqj.net/exist/. Utilizaremos las siguientes interfaces para conectarnos y obtener los datos de la base de datos:

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQPreparedExpression;
import javax.xml.xquery.XQResultItem;
import javax.xml.xquery.XQResultSequence;
import net.xqj.exist.ExistXQDataSource;
```


Configurar una conexión.

- ***XQDataSource.*** Identifica el origen de datos a partir del cual se va a crear la conexión. Cada implementación definirá las propiedades necesarias para efectuar la conexión. Por ejemplo, este código realiza la conexión con eXist, hay que poner el nombre del servidor (Localhost), el puerto de la BD (8080), el usuario (admin) y su password (admin). Aunque obligatoria es solo la propiedad *serverName* si estamos en local.

```
XQDataSource server = new ExistXQDataSource();
server.setProperty("serverName", "localhost");
server.setProperty("port", "8080");
server.setProperty("user", "admin");
server.setProperty("password", "admin");
```

- ***XQConnection.*** Representa una sesión con la base de datos, manteniendo información de estado, transacciones, expresiones ejecutadas y resultados. Por ejemplo:

```
XQConnection conn = server.getConnection();
```

También se puede indicar el usuario y paswoord que abre la sesión:

```
XQConnection conn = server.getConnection("admin","admin");
```

La conexión la cerramos escribiendo `conn.close();`

Procesar los resultados de una consulta:

- ***XQPreparedExpression***: Objeto creado a partir de una conexión para la ejecución de una expresión múltiples veces. Devuelve un ***XQResultsetSequence*** con los datos obtenidos. Igual que en *XQExpression*, la ejecución se produce llamando al método ***executeQuery*** y se evalúa teniendo en cuenta el contexto estático en vigor.
- ***XQResultSecuence***: Resultado de la ejecución de una sentencia; contiene, un conjunto de 0 o más ***XQResultItem***. Es un objeto recorrible. Este ejemplo obtiene los empleados del departamento 10, utilizamos ***getItemAsString*** para que devuelva los elementos como cadenas:

```
XQPreparedExpression consulta;
XQResultSequence resultado;
consulta =conn.prepareExpression("/EMPLEADOS/EMP_ROW[DEPT_NO=10]");
resultado = consulta.executeQuery();
while(resultado.next())
        System.out.println("Elemento: " +resultado.getItemAsString(null));
```

- ***XQResultItem***: Objeto que representa un elemento de un resultado, inmutable, válido hasta que se llama al método close suyo o de la *XQResultSequence* a la que pertenece. El ejemplo anterior lo podemos poner así:

```
consulta =conn.prepareExpression("/EMPLEADOS/EMP_ROW[DEPT_NO=10]");
resultado = consulta.executeQuery();
```

```
XQResultItem r_item;
while(resultado.next()){
    r_item = (XQResultItem) resultado.getItem();
    System.out.println("Elemento: " + r_item.getItemAsString(null));    }
```

- *XQItem*: Representación de un elemento en XQuery. Es inmutable y una vez creado su estado interno no cambia.
- *XQSecuence*: Representación de una secuencia del modelo de datos XQuery, contiene un conjunto de 0 o más XQItem. Es un objeto recorrible.

Con XQJ no necesitamos seleccionar la colección de los documentos XML, la búsqueda la realiza en todas las colecciones. Tampoco admite el uso de *Sentencias de actualización de eXist*, con lo cual las inserciones, borrados y actualizaciones resultan bastante engorrosas. El siguiente ejemplo mostrará los datos del documento *productos.xml*, si tenemos productos en distintos documentos se mostrarán las etiquetas de todos los documentos:

```
public static void productos (String[] args) throws XQException
      {
        XQDataSource server = new ExistXQDataSource();
        server.setProperty("serverName", "localhost");
        XQPreparedExpression consulta;
        XQResultSequence resultado;
        XQConnection conn = server.getConnection();

        System.out.println("------- Consulta documentos productos.xml ------");
        consulta =conn.prepareExpression("for $de in /productos return $de");
        resultado = consulta.executeQuery();

        while(resultado.next())
         { System.out.println("Elemento "+ resultado.getItemAsString(null)); }
        conn.close();
      }
```

Actividad 10: Realiza un programa Java que contenga una función *main()* que lea de la entrada estándar el tipo de departamento cuyos empleados se quieren visualizar. Debe invocar al método *Visualizar(tipo)*, que recibe el tipo y visualiza los empleados de los departamentos del tipo indicado. Si no hay empleados o si el tipo no existe se debe visualizar un mensaje indicándolo. Utiliza el documento *universidad.xml*.

El siguiente ejemplo crea un fichero XML en el disco con nombre *NUEVO_EMPLE10.xml*, a partir de los datos extraídos de una consulta. Para crear el fichero utilizaremos la **clase File** del del paquete **java.io.** La consulta devuelve los empleados del departamento 10, más el título del documento *empleados.xml*, de la base de datos, la consulta es la siguiente:

```
let $titulo:= /EMPLEADOS/TITULO
return  <EMPLEADOS>{$titulo}
```



```
        {for $em in /EMPLEADOS/EMP_ROW[DEPT_NO=10]
        return $em}</EMPLEADOS>
```

El ejercicio nos queda así:

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import javax.xml.xquery.*;
import net.xqj.exist.ExistXQDataSource;

public class creaficheroexterno {
  public static void main(String[] args) throws XQException {
      String nom_archivo = "NUEVO_EMPLE10.xml";
      File fichero = new File(nom_archivo);

      XQDataSource server = new ExistXQDataSource();
      server.setProperty("serverName", "localhost");
      server.setProperty("port", "8080");
      XQConnection conn = server.getConnection();
      XQPreparedExpression consulta = conn.prepareExpression
        "let $titulo:= /EMPLEADOS/TITULO return  <EMPLEADOS>{$titulo}" +
        " {for $em in /EMPLEADOS/EMP_ROW[DEPT_NO=10]" +
        " return $em}</EMPLEADOS>");
       XQResultSequence result = consulta.executeQuery();
       if (fichero.exists())
        { //borramos el archivo si existe y se crea de nuevo
           if(fichero.delete()) System.out.println("Archivo borrado. Creo de nuevo.");
           else System.out.println("Error al borrar el archivo");
        }
       try{
          BufferedWriter bw = new BufferedWriter(new FileWriter(nom_archivo));
          bw.write("<?xml version='1.0' encoding='ISO-8859-1'?>" + "\n");
          while (result.next())   {
             String cad = result.getItemAsString(null);
            System.out.println(" output "+ cad); //visualizamos
            bw.write(cad+ "\n"); // grabamos en el fichero
            }
            bw.close(); // Cerramos el fichero el fichero
          } catch (IOException ioe){ioe.printStackTrace();}
    conn.close();
  }
}
```

Actividad 11: A partir del los documentos *productos.xml* y *zonas.xml*

Realiza un programa Java que cree un documento externo con nombre *zonas20.xml*. Debe contener los productos de la zona 20 y las siguientes etiquetas para cada producto: *<cod_prod>*, *<denominacion>*, *<precio>*, *<nombre_zona>*, *<director>* y *<stock>*, este stock debe ser el cálculo del stock_actual - stock_minimo.

5.4.3 TRATAMIENTO DE EXCEPCIONES

Hasta ahora en los ejemplos hemos utilizado la palabra clave **throws** en la función *main()* para indicar que podía lanzar la excepción **XMLDBException** o **XQException**. En los siguientes ejemplos se tratan estas excepciones dentro de un bloque **try-catch**.

XMLDBEXCEPTION:

Esta excepción se lanza cuando se produce un error en la API XML:DB. Contiene dos códigos de error, uno es el código de error XML de la API, y el otro es definido por el proveedor específico. El error del proveedor vendrá definido por *ErrorCodes.VENDO _ERROR*. **XMLDBException** extiende **java.lang.Exception.**

Cuando se produce un error con **XMLDBException** podemos acceder a cierta información usando el método *getMessage()* que devuelve una cadena que describe el error.

En el siguiente ejemplo podemos ver como capturar los posibles errores que nos pueden ocurrir:

```
import org.xmldb.api.*;
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;

public class pruebaexcepciones {
protected static String driver = "org.exist.xmldb.DatabaseImpl";
public static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
private static Database database;
private static String usu="admin";
private static String pwd="admin";
private static Class cl=null;
private static XPathQueryService service;
private static ResourceSet result=null;
private static Collection col = null; // Colección

public static void main(String[] args) {
try {
      cl = Class.forName(driver);
      } catch (ClassNotFoundException e) {
        System.out.println("No se encuentra la clase del driver: " + e.getMessage());
      }

try {
        database = (Database) cl.newInstance();
        DatabaseManager.registerDatabase(database);
      } catch (InstantiationException e) {
        System.out.println("Error instanciando el driver. ");
      } catch(NullPointerException e) {
        System.out.println("Error al instanciar la clase del driver:"+
                                e.getMessage());
      } catch (IllegalAccessException e) {
        System.out.println("Se ha producido una IllegalAccessException");
      } catch (XMLDBException e) {
```



```
        System.out.println("Error XMLDB :" + e.getMessage());
    }

try {
        col = DatabaseManager.getCollection(URI+"/db/Pruebas", usu, pwd);
      } catch (XMLDBException e) {
        System.out.println("ERROR XMLDBException en getCollection."+
                                e.getMessage());
      }

try {
        service =(XPathQueryService) col.getService("XPathQueryService", "1.0");
      } catch (NullPointerException n){
        System.out.println("Error en getService, no se puede crear el servicio.")
      } catch (XMLDBException e) {
        System.out.println("ERROR XMLDBException, en get service."+ e.getMessage());
      }
//Consulta a la BD
try {
     result = service.query("for $b in /EMPLEADOS/EMP_ROW[APELLIDO='TOVAR']
                                                                return $b");
   }catch (NullPointerException n){
          System.out.println("Error en query, no se ha inicializado BD o el
                                servicio");
        }catch (XMLDBException e) {
          System.out.println("Error XMLDBException en la query: "+ e.getMessage());
        }

try {
      ResourceIterator i;
      i = result.getIterator();
      if (!i.hasMoreResources())System.out.println("LA CONSULTA NO DEVUELVE NADA");
      while (i.hasMoreResources()) { //Procesamos el resultado
        Resource r = i.nextResource();
        System.out.println((String) r.getContent());
      }
      }catch ( NullPointerException n){
        System.out.println("Error getIterator. Problemas con el servicio.");
      }catch (XMLDBException  e) {
        System.out.println("Error XMLDBException, getIterator. : "+ e.getMessage());
      }

try {
        col.close();
      }catch ( NullPointerException n){
        System.out.println("Error en el cierre de la colección.");
      } catch (XMLDBException  e) {
        System.out.println("Error XMLDBException, col.close. : "+ e.getMessage());
      }
}//fin main
}// fin de la clase pruebaexcepciones
```

- El error ***NullPointerException***, es el error que se dispara en todos los casos y siempre que no se haya inicializado la BD, es el caso de escribir mal el driver.

- Si escribimos mal la URI, se disparará ***XMLDBException*** a la hora de asignar la colección, en *getCollection*, y a partir de ahí se disparan todos los *NullPointerException*, pues el servicio para esa colección no se ha podido crear.
- Si escribimos mal la carpeta donde se encuentra la colección, se disparan los *NullPointerException* siguientes, ya que el servicio no se ha podido crear.
- Si escribimos mal la query se dispara el error ***XMLDBException,*** a la hora de crear el service.query, en el mensaje visualiza en qué línea de la consulta está el error. Por ejemplo si escribo esta consulta:

```
res = service.query("forr $b in /EMPLEADOS/EMP_ROW[APELLIDO='TOVAR'] return $b");
```

Se visualiza este mensaje:

```
Error XMLDBException en la query: Failed to invoke method queryP in class org.
exist.xmlrpc.RpcConnection: org.exist.xquery.StaticXQueryException: org.exist.xquery.
XPathException: err:XPST0003 in line 1, column 6: unexpected token: $
```

Actividad 12: Escribe y prueba esta clase en tu ordenador y prueba las distintas situaciones de error.

Para saber más sobre esta excepción acceder al sitio:
http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/base/XMLDBException.html

XQEXCEPTION:

Cuando se produce un error con **XQException** disponemos de dos métodos que nos van a permitir saber la causa del error y poderlo arreglar. Estos métodos son *getMessage()* que devuelve una cadena que describe el error y *getCause()* que nos indica la causa del error para proceder a su solución.

En el siguiente ejemplo vemos como capturar los posibles errores a la hora sobre todo de realizar consultas:

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQPreparedExpression;
import javax.xml.xquery.XQResultSequence;
import net.xqj.exist.ExistXQDataSource;

public class PruebaXQJExcepciones {
  public static void main(String[] args)
   {
    XQConnection conn = null;
    XQDataSource server = new ExistXQDataSource();
    XQPreparedExpression consulta=null;
    XQResultSequence rs=null,resultado=null;
```



```
    try {
        server.setProperty("serverName", "localhost");
    }catch (XQException  e) {
            System.out.println("Error en el servidor. Mensaje : "+ e.getMessage());
            System.out.println("Error en el servidor. Causa :  "+ e.getCause());
    }

    try {
       conn = server.getConnection();
    }catch (XQException  e) {
            System.out.println("Error en la conexión : "+ e.getMessage());
     }

    System.out.println("--------Ejemplo Consulta Productos -----------------");
    try {
       consulta =conn.prepareExpression("for $de in /productos/product return $de");
    }catch (XQException  e) {
            System.out.println("Error en la expresión. Mensaje : "+ e.getMessage());
            System.out.println("Error en la expresión. Causa : "+ e.getCause());
    }

     try {
       resultado = consulta.executeQuery();
       while(resultado.next()){
            System.out.println("Elemento "+ resultado.getItemAsString(null));}
     }catch (XQException  e) {
            System.out.println("Error en la ejecución. Mensaje: "+ e.getMessage());
            System.out.println("Error en la ejecución. Causa: "+ e.getCause());
      }

     try {
       conn.close();
     } catch (XQException e) {
            System.out.println("Error al cerrar la conexión.");
     }
    } //   FIn maim
}// FIN CLASS PruebaXQJExcepciones
```

Actividad 13: Escribe y prueba esta clase en tu ordenador y provoca situaciones de error, escribiendo mal la consulta, o el nombre del servidor o cerrando la BD.

Para saber más sobre esta excepción acceder al sitio:
http://www.xquery.com/docs/ddxq3.0/javadoc/com/ddtek/xquery3/XQException.html

5.5 EJERCICIOS

❶ A partir del documento *departamentos.xml* de la colección *Pruebas* realiza un programa Java para gestionar dicho documento. Utiliza las clases **Main.java** y **Pantalla.java** del Ejemplo 2 de la Unidad 3. El programa debe mostrar la pantalla inicial (Figura 5.17) donde podremos consultar, insertar, borrar y eliminar nodos <DEPT_ROW> del documento *departamentos.xml*.

Comprueba que al insertar un departamento no exista ya en el documento, igualmente a la hora de borrar o de modificar hay que comprobar que el departamento exista. Visualiza los mensajes informativos en cada caso.

Crea una clase para gestionar las operaciones sobre el documento, esta clase debe contener los métodos para conectarnos a eXist, insertar, borrar, modificar o consultar en el documento. Utiliza las *Sentencias de actualización de eXist*.

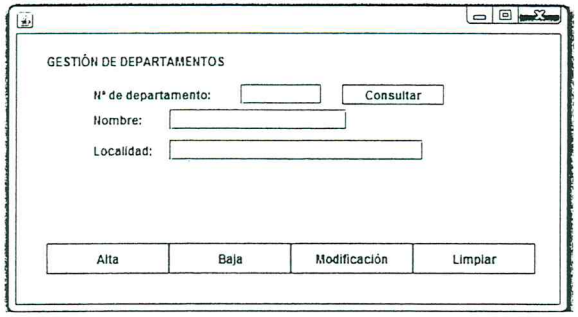


Figura 5.17. Pantalla inicial ejercicio Gestión de Departamentos.

A PARTIR DE LA COLECCIÓN VENTAS:

❷ Haz un programa Java que cree la colección VENTAS y suba los documentos que se encuentran en la carpeta ***ColeccionVentas***. Esta colección contiene los siguientes documentos XML:

clientes.xml: contiene datos de clientes que compran los productos. Por cada cliente tenemos

```
<clien numero="nn"> <nombre>xxxxxx</nombre>
                    <poblacion>xxxxxxx</poblacion>
                    <tlf>xxxxxx</tlf>
                    <direccion>xxxxxxxx</direccion>
</clien>
```

productos.xml: contiene los datos de los productos que son comprados por los clientes. Por cada producto tenemos su categoría y precio, que van como atributos, el código de producto, el nombre y el stock del producto:

```
<product  categoria="xxx" pvp="xxxx">
          <codigo>xxxxxx</codigo>
          <nombre>xxxxxx</nombre>
```



```
          <stock>xxxxxxx</stock>
</product>
```

facturas.xml: contiene los datos de las facturas de los clientes. Por cada factura tenemos el número de factura como atributo, la fecha de factura, el importe y el número de cliente, los nodos son:

```
<factura numero="xxxx">
      <fecha>xxxxxx</fecha>
      <importe>xxxxxx</importe>
      <numcliente>xxxxxxx</numcliente>
</factura>
```

detallefacturas.xml: contiene el detalle de cada factura, es decir los datos de los productos y las unidades compradas por los clientes. También cada producto de la factura lleva asociado un porcentaje de descuento. Por cada factura se dispone de la siguiente información

```
<factura numero="xxxx">
      <codigo>xxxxxxx</codigo>
      <producto descuento="xxx">
         <codigo>xxx</codigo>
         <unidades>xxxxx</unidades>
      </producto>
      <producto descuento="xxx">
         <codigo>xxxxx</codigo>
         <unidades>xxxx</unidades>
      </producto>
      . . . . . . . . . . . . . .
</factura>
```

REALIZAR LOS SIGUIENTES EJERCICIOS:

❸ Realiza una consulta XQuery para obtener las facturas que tiene cada cliente, que aparezca solo el nombre del cliente y el número de factura entre las etiquetas <facturaclientes></facturaclientes>. Debe obtener algo parecido a esto:

```
<facturasclientes> <nombre>Pilar Martín</nombre> <nufac>10</nufac></facturasclientes>
<facturasclientes> <nombre>Pilar Martín</nombre> <nufac>11</nufac></facturasclientes>
<facturasclientes> <nombre>Antonio Reus</nombre> <nufac>12</nufac></facturasclientes>
. . . . . .
```

❹ Realiza una consulta XQuery para obtener el detalle de las ventas de los productos de la factura número 10. Obtén por cada producto de esa factura, el código, nombre, la cantidad vendida, el precio, y el importe que será la suma de las unidades por el precio del producto. Utiliza los documentos *productos.xml* y *detallefacturas.xml*. La salida debe ser similar a lo que se muestra:

```
<detalle><codigo>1</codigo><nombre>Silla Plegable</nombre>
    <cant>10</cant><pvp>100</pvp><importe>1000</importe>
</detalle>
<detalle><codigo>2</codigo><nombre>BH Prisma</nombre>
    <cant>3</cant><pvp>900</pvp><importe>2700</importe>
</detalle>
. . . . . .
```

❺ Realiza una consulta XQuery para obtener el detalle de las ventas de los productos de cada una de las facturas del documento *detallefacturas.xml*. Crear una salida que muestre para cada factura, el número de factura y el código como atributos dentro de la etiqueta <factura>. Y a continuación los artículos, el código de artículo será un atributo de la etiqueta <articulo>. Utiliza los documentos *productos.xml* y *detallefacturas.xml*. La salida debe ser similar a lo que se muestra:

```
<factura numero="10" codigo="FACT10">
    <articulo codigo="1"><nombre>Silla Plegable</nombre>
        <cant>10</cant><pvp>100</pvp><importe>1000</importe>
    </articulo>
    <articulo codigo="2"><nombre>BH Prisma</nombre>
        <cant>3</cant><pvp>900</pvp><importe>2700</importe>
    </articulo>
. . . . . .
</factura>
<factura numero="11" codigo="FACT11">
. . . . . .
```

❻ Haz un programa Java que genere un fichero XML a partir de la consulta creada anteriormente, llama al nuevo fichero *facturastotal.xml*. Una vez creado sube el fichero a la colección *Ventas* de la base de datos. Recuerda que el documento XML debe estar bien formado porque si no, no se podrá cargar en la base de datos.

❼ Utilizando el documento creado anteriormente, realiza una consulta de actualización para actualizar la etiqueta <importe> del documento *facturas.xml*. La actualización consiste en actualizar el importe de cada factura con la suma de los importes de los artículos que componen la factura, obtén la suma de importes por cada factura a partir del documento creado anteriormente (*facturastotal.xml*).

❽ Utilizando los documentos *facturas.xml* y *clientes.xml*, realiza una consulta XQuery para obtener todo lo que tiene que pagar cada cliente. La salida debe ser similar a:

```
<nombre>Alicia Díaz<totalapagar>0</totalapagar></nombre>
<nombre>Pilar Martín<totalapagar>8180</totalapagar></nombre>
. . . . . .
```

❾ Utilizando los datos devueltos por la consulta anterior, haz un programa Java para obtener el nombre del cliente que más importe tiene que pagar.

UNIDAD 6

PROGRAMACIÓN DE COMPONENTES DE ACCESO A DATOS

CONTENIDOS

6.1 INTRODUCCIÓN
6.2 CONCEPTO DE COMPONENTE
6.2.1 Características
6.2.2 Ventajas e inconvenientes
6.3 JAVABEANS
6.3.1 Propiedades y atributos
6.3.2 Eventos
6.3.3 Persistencia del componente
6.4 HERRAMIENTAS PARA EL DESARROLLO DEL COMPONENTE
6.4.1 Crear JavaBeans con NetBeans
6.5 EMPAQUETADO DE COMPONENTES
6.6 USANDO JAVABEANS PARA ACCEDER A BASES DE DATOS
6.6.1 Usando un componente para acceder a una BDOO
6.6.2 Uso de un componente para acceder a bases de datos SQL
6.7 EJERCICIOS

OBJETIVOS

El alumno al término de esta unidad debe ser capaz de:

- Utilizar herramientas para desarrollar componentes de acceso a datos.
- Programar componentes para acceder a bases de datos.
- Utilizar un mismo componente en distintas bases de datos.

6.1 INTRODUCCIÓN

Los continuos avances en la Informática y las Telecomunicaciones están haciendo cambiar la forma en la que se desarrollan actualmente las aplicaciones software. Los modelos de programación existentes se ven incapaces de manejar la complejidad de los requisitos de los sistemas abiertos y distribuidos. Surgen nuevos paradigmas de programación como la programación orientada a componentes que supone una "extensión" de la programación orientada a objetos y está basada en la noción de COMPONENTE. El Desarrollo de *Software Basado en Componentes* (DSBC), trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes software reutilizables.

6.2 CONCEPTO DE COMPONENTE

Existen muchas definiciones de componentes software, pero una de las más difundidas es la de *Szyperski*, 1998:

"*Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio*". El objetivo del desarrollo basado en componentes es construir aplicaciones mediante ensamblado de módulos software reutilizables, que han sido diseñados previamente con independencia de las aplicaciones en las que van a ser utilizados.

6.2.1 CARACTERÍSTICAS

Existen algunas características claves para que un elemento pueda ser catalogado como componente:

- **Independiente de la plataforma:** Hardware, Software, Sistema Operativo.
- **Identificable**: Debe tener una identificación que permita acceder fácilmente a sus servicios y que permita su clasificación.
- **Autocontenido**: Un componente no debe requerir de la utilización de otros para llevar a cabo la función para la cual fue diseñado.
- **Puede ser remplazado por otro componente**: Se puede remplazar por nuevas versiones u otro componente que lo mejore.
- **Con acceso solamente a través de su interfaz:** Una interfaz define el conjunto de operaciones que un componente puede realizar; estas operaciones se llaman también servicios o responsabilidades. Las interfaces proveen un mecanismo para interconectar componentes y controlar las dependencias entre ellos.
- **Sus servicios no varían**: Las funcionalidades ofrecidas en su interfaz no deben variar, pero su implementación sí.
- **Bien documentado**: Un componente debe estar correctamente documentado para *facilitar su búsqueda* si se quiere actualizar, integrar con otros, adaptarlo, etc.
- **Es genérico:** Sus servicios deben servir para varias aplicaciones.
- **Reutilizado dinámicamente:** Puede ser cargado en tiempo de ejecución en una aplicación.

- **Se distribuye como un paquete:** En este paquete se almacenan todos los elementos que lo constituyen.

La forma concreta de especificar, implementar o empaquetar un componente depende de la tecnología utilizada. Las tecnologías basadas en componentes incluyen dos elementos:

- **Modelo de componentes**. Especifica las reglas de diseño que deben obedecer los componentes, sus interfaces y la interacción entre componentes.
- **Plataforma de componentes**. Es la infraestructura de software requerida para la ejecución de aplicaciones basadas en componentes. Se basan en un determinado modelo de componentes.

Ejemplo de tecnologías de componentes son las siguientes:

- La plataforma .NET de Microsoft para sistemas Windows. Representa una nueva etapa en la evolución de COM (*Component Object Model*), la plataforma de componentes de Microsoft. Con ella, Microsoft impulsa la idea de 'industrializar' el software utilizando tecnologías de componentes.
- JavaBeans y EJB (*Enterprise JavaBeans*) de *Oracle Corporation* (inicialmente desarrollado por *Sun Microsystems*). Es la tecnología de componentes basada en Java que funciona en cualquier plataforma.

A lo largo de este tema trabajaremos con los JavaBeans, la tecnología de componentes basada en Java.

6.2.2 VENTAJAS E INCONVENIENTES

Un componente es una pieza de software que describe o ejecuta funciones específicas a través de una interfaz bien definida. Se pueden juntar o combinar varios componentes para llevar a cabo una tarea. El uso de componentes aporta una serie de ventajas:

- Reutilización de software.
- Disminuye la complejidad del software. Se pueden ir probando trozos de componentes antes de probar el conjunto de todos los componentes ensamblados.
- Mejora el mantenimiento del sistema, los errores son más fáciles de detectar.
- Incrementa la calidad del software ya que un componente puede ser construido y luego mejorado.

El actual diseño basado en componentes también tiene limitaciones ya que solo existen en algunos campos como las GUIs y no siempre se pueden encontrar los componentes adecuados para cada proyecto. A esto hay que añadir la falta de estándares y la falta de procesos de certificación que garanticen la calidad de los componentes.

6.3 JAVABEANS

Un JavaBean es un componente de software reutilizable que está escrito en lenguaje Java. Puede ser manipulado visualmente mediante herramientas de desarrollo Java. Ejemplos de JavaBeans son las librerías gráficas AWT (*Abstract Window Toolkit*), API de Java que permite hacer aplicaciones con componentes GUI (como ventanas, botones, barras, campos de texto, etc.); y SWT (*Standard Widget Toolkit*) de Eclipse.

A menudo nos referimos a un JavaBean como un Bean. Han de cumplir las siguientes características:

- **Introspección.** Mecanismo mediante el cual los propios JavaBeans proporcionan información sobre sus características (propiedades, métodos y eventos). Los Beans soportan la introspección de dos formas: utilizando patrones de nombrado (que son como reglas para nombrar las características del Bean) y proporcionando las características mediante una clase *Bean Information* relacionada.
- **Manejo de eventos.** Los Beans se comunican con otros Bean utilizando los eventos. Un Bean puede tener interés en recibir y responder a eventos enviados por otro Bean.
- **Propiedades.** Las propiedades definen las características de apariencia y comportamiento de un Bean que pueden ser modificadas durante el diseño de los componentes.
- **Persistencia.** Permite a los Beans almacenar su estado y restaurarlo posteriormente. Se basa en la serialización.
- **Personalización.** Los programadores pueden alterar la apariencia y conducta del Bean durante el diseño. La personalización se soporta de dos formas: utilizando editores de propiedades o utilizando personalizadores de Beans más sofisticados.

Un JavaBean es una clase Java que se define a través de las propiedades que expone, los métodos que ofrece y los eventos que atiende o genera. Su definición requiere ciertas reglas:

- Debe tener un constructor sin argumentos, aunque puede tener más de uno.
- Debe implementar la interfaz **Serializable** (para poder implementar **persistencia**).
- Sus propiedades deben ser accesibles mediante métodos **get** y **set**.
- Los nombres de los métodos deben obedecer a ciertas convenciones de nombrado.

Aunque los JavaBean se diseñaron para ser utilizados y manipulados por herramientas visuales, en este tema los trataremos como componentes de acceso a base de datos; y veremos cómo el mismo componente lo podemos usar para acceder a diferentes bases de datos. El siguiente ejemplo muestra un JavaBean (no visual) con una propiedad (de nombre *propiedad*), el constructor sin parámetros y los métodos *get*, para obtener el valor de dicha propiedad (*getPropiedad()*) y *set* para dar valor a la propiedad (*setPropiedad()*):

```
public class MiPrimerBean implements Serializable {
    private String propiedad;

    public MiPrimerBean() {propiedad = "";}
    public String getPropiedad() {return propiedad;}
    public void setPropiedad(String propiedad) {this.propiedad = propiedad;}
}
```


No debemos confundir los JavaBeans, diseñados para crear componentes reusables que suelen usarse en herramientas de desarrollo IDE y en componentes visuales (aunque no exclusivamente), pensados para ser procesos locales; con la especificación *Enterprise Java Beans* (EJB) que describe un modelo de componentes del lado servidor, los componentes se almacenan en un servidor para ser invocados por los clientes.

6.3.1 PROPIEDADES Y ATRIBUTOS

Las propiedades de un Bean son los atributos que determinan su apariencia y comportamiento. Por ejemplo, un *Producto* puede tener las siguientes propiedades: *descripción, pvp, stockactual, idproducto*, etc. Para acceder a las mismas se utilizan los métodos *getter* (para leer el valor de la propiedad) y *setter* (para cambiar el valor de la propiedad) de la siguiente forma:

```
public TipoPropiedad  getNombrePropiedad() { ... }
public void setNombrePropiedad (TipoPropiedad valor) { ... }
```

Las propiedades pueden ser simples, indexadas, ligadas o restringidas.

PROPIEDADES SIMPLES

Las propiedades simples representan un único valor; por ejemplo, si un atributo se llama *pvp* y es de tipo *float* los métodos *getter* y *setter* son los siguientes:

```
private float pvp;

public float getPvp() { return pvp; }
public void setPvp(float pvp) { this.pvp = pvp; }
```

Si la propiedad es booleana se escribe **isNombrePropiedad()** para obtener su valor:

```
private boolean conectado = false;

public boolean isConectado() { return conectado; }
public void setConectado(boolean conectado) { this.conectado = conectado; }
```

PROPIEDADES INDEXADAS

Las propiedades indexadas representan un array de valores a los que se accede mediante un índice. Se deben definir métodos *getter* y *setter* para acceder al array completo y a valores individuales. Ejemplo:

```
private int[] categorias = {1,2,3};

//métodos get y set para acceder al array completo
public void setCategorias(int[] valor){ categorias = valor; }
public int[] getCategorias(){ return categorias; }

//métodos get y set para acceder a valores individuales
```

```
public void setCategorias(int indice, int valor){ categorias[indice]=valor; }
public int getCategorias(int indice){ return categorias[indice]; }
```

PROPIEDADES LIGADAS:

Son las propiedades asociadas a eventos. Cuando la propiedad cambia se notifica a todos los objetos interesados la naturaleza del cambio, permiténdoles realizar alguna acción. Para que el Bean soporte propiedades ligadas (también llamadas compartidas), debe mantener una lista de los receptores de la propiedad y alertar a dichos receptores cuando cambie la propiedad; para ello proporciona una serie de métodos de la clase **PropertyChangeSupport**. La lista de receptores se mantiene gracias a los métodos ***addPropertyChangeListener()*** y ***removePropertyChangeListener()***. Un Bean que produce eventos (llamado Bean fuente) tiene la forma:

```
public class BeanFuente implements Serializable {

    private PropertyChangeSupport propertySupport;
    ......
    //en el constructor se indica el objeto al que se le quiere dar soporte (this)
    public BeanFuente () {
        propertySupport = new PropertyChangeSupport(this);
    }
    ......

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

Para que nuestro Bean receptor pueda oir los eventos de cambio de propiedad, debe implementar el método ***propertyChange()*** de la interfaz **PropertyChangeListener**. El Bean fuente llama a este método de notificación de todos los receptores de su lista de receptores. La notificación no es mas que un objeto **PropertyChangeEvent** que encapsula la propiedad que ha cambiado (su antiguo y nuevo valor). El receptor cada vez que reciba este objeto lo examinará y podrá descubrir el cambio. Un Bean receptor (oyente) de un cambio de propiedad tiene la forma:

```
public class BeanReceptor implements PropertyChangeListener {
   .....
   public void propertyChange(PropertyChangeEvent evt) {
      System.out.println("Valor anterior: "+ evt.getOldValue());
      System.out.println("Valor actual: "+ evt.getNewValue());
   }
   .....
}
```

Al Bean que tiene la propiedad ligada se le conoce como fuente y al que recibe la notificación de los cambios como receptor. La Figura 6.1 muestra los métodos que deben implementar los Bean

fuente y receptor de eventos cuando utilizan propiedades ligadas, el lanzamiento del evento se realiza mediante el método ***firePropertyChange()***.

Figura 6.1. Beans fuente y receptor de eventos, propiedades ligadas.

PROPIEDADES RESTRINGIDAS

Son similares a las propiedades ligadas, pero en este caso los objetos a los que se les notifica el cambio del valor de la propiedad pueden vetarlo si no se ajusta a unas determinadas características; la propiedad que puede dar lugar a un veto se le llama restringida. Se deben proporcionar dos métodos de registro para los receptores:

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

El método ***addVetoableChangeListener()*** hace que el objeto oyente (listener) sea capaz de capturar eventos de tipo **PropertyVetoEvent** y el método ***removeVetoableChangeListener()*** hace que el objeto oyente deje de escuchar los eventos de veto. Para definir estos métodos se puede crear un objeto **VetoableChangeSupport** para conseguir un soporte sencillo a fin de definir los métodos anteriores:

```
public class BeanFuente implements Serializable {

    private VetoableChangeSupport soporteVeto;
    ......
    //en el constructor se indica el objeto al que se le quiere dar soporte (this)
    public BeanFuente () {
        soporteVeto = new VetoableChangeSupport(this);
    }
    ......

    public void addVetoableChangeListener(VetoableChangeListener listener) {
        soporteVeto.addVetoableChangeListener(listener);
    }

    public void removeVetoableChangeListener(VetoableChangeListener listener){
        soporteVeto.removeVetoableChangeListener(listener);
    }
}
```

Para que el Bean receptor pueda oir los eventos debe implementar el método ***vetoableChange()*** de la interfaz **VetoableChangeListener.** El método recibe una copia del objeto **PropertyChangeEvent** y lanza excepciones del tipo *PropertyVetoException* (mediante la instrucción **throw**) cuando el cambio de valor en la propiedad no se pueda realizar debido a que no cumple una condición. El lanzamiento de estas excepciones tiene esta sintaxis: *throw new PropertyVetoException(String mensaje, PropertyChangeEvent evento)*:

```
public class BeanReceptor implements VetoableChangeListener {
   .....
   public void vetoableChange(PropertyChangeEvent evt) trows PropertyVetoException{
    //comprobación de las condiciones
    .....
    //se lanza excepción si no se aprueba el cambio
    throw new PropertyVetoException ("mensaje",evt);

   }
   .....
}
```

El objeto de excepción lanzado permite obtener el mensaje (método *getMessage()*) y el evento que desencadenó la excepción (*getPropertyChangeEvent()*). El Bean que tiene la propiedad restringida captura la excepción y asigna a la propiedad el valor anterior, sin tener que notificar a todas las partes interesadas el nuevo cambio de valor.

La Figura 6.2 muestra los métodos que deben implementar los Bean fuente y receptor de eventos cuando utilizan propiedades restringidas, el lanzamiento del evento se realiza mediante el método ***fireVetoableChange()***. Las propiedades restringidas son más complejas de implementar que las ligadas.

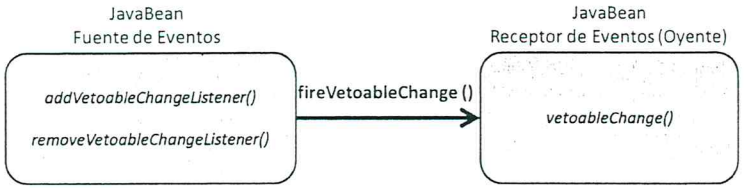


Figura 6.2. Beans fuente y receptor de eventos, propiedades restringidas.

6.3.2 EVENTOS

Los Beans utilizan los eventos para comunicarse con otros Beans. Para lanzar eventos **PropertyChangeEvent** a los oyentes, se utiliza el método ***firePropertyChange()*** cada vez que se cambia el valor de la propiedad. A este método se le pasan tres parámetros: el nombre de la propiedad, el valor antiguo y el valor nuevo. Los valores antiguos y nuevos deben de ser objetos (*Object*), lo que obliga a utilizar las clases envoltorio Integer, Double, Boolean, Char, etc. para poder pasar valores de tipos básicos.

El siguiente ejemplo muestra el método *set* para la propiedad *stockactual* en el Bean fuente. Si el stock actual es menor que el stock mínimo (es decir hay un cambio en la propiedad que nos interesa controlar) se lanza un evento mediante el método ***firePropertyChange()***, al que se le envían los 3 parámetros ya indicados anteriormente:

```
public void setStockactual(int valorNuevo) {
    int valorAnterior = stockactual;
    stockactual = valorNuevo;
    if (stockactual < getStockminimo()) //hay que realizar pedido
        propertySupport.firePropertyChange("stockactual", valorAnterior,
                                                          stockactual);
}
```

El Bean receptor de eventos de tipo **PropertyChangeEvent** debe implementar el método ***propertyChange()*** de la interfaz **PropertyChangeListener.** Este método es llamado cuando se modifica el valor de la propiedad. El evento **PropertyChangeEvent** que recibe este método proporciona varios métodos:

- *Object getOldValue()*: Obtiene el antiguo valor de la propiedad.
- *Object getNewValue()*: Obtiene el nuevo valor de la propiedad.
- *String getPropertyName()*: Obtiene el nombre de la propiedad que cambió (puede ser null si cambiaron varias propiedades a la vez).

En el caso de las propiedades restringidas, para lanzar eventos **PropertyVetoEvent** a los oyentes, se utiliza el método ***fireVetoableChange()*** que también requiere tres parámetros: el nombre de la propiedad que cambia, el valor antiguo y el valor nuevo de la propiedad; este método debe capturar excepciones *PropertyVetoException*:

```
public void setStockactual(int valorNuevo) {
    try{
      int valorAnterior = stockactual;
      stockactual = valorNuevo;
      if (stockactual < getStockminimo()) //hay que realizar pedido
         soporteVeto.fireVetoableChange("stockactual", valorAnterior,
                                                       stockactual);
    } catch(PropertyVetoException pve ){
      //Código que se ejecuta en caso de excepción
    }
}
```

El Bean receptor de eventos de tipo **PropertyVetoEvent** debe implementar el método ***vetoableChange()*** de la interfaz **VetoableChangeListener.**

6.3.3 PERSISTENCIA DEL COMPONENTE

Mediante el mecanismo de persistencia un Bean es capaz de almacenar su estado en un momento determinado y recuperarlo posteriormente. Para que un Bean sea persistente debe implementar la interfaz **Serializable** usando las librerías APIs de serialización de Java; entonces, todos los valores del Bean serán trasladados a una cadena de bytes que se almacenarán en un fichero. Esta cadena contendrá la información suficiente para reconstruir el Bean almacenado en su último estado. Cuando se carga el componente en el programa se hace a partir del fichero serializado.

A la hora de implementar la interfaz **Serializable** hay que tener en cuenta que:

1. Las clases que implementan **Serializable** deben tener un constructor sin argumentos.
2. Todos los campos excepto **static** y **transient** son serializados. Para especificar los campos que no queremos serializar utilizaremos el modificador **transient:**

```
transient int campo;
```

Hemos de tener en cuenta que se deben guardar aquellas características del Bean que le permitan reincorporarse al estado en que se encontraba.

6.4 HERRAMIENTAS PARA EL DESARROLLO DEL COMPONENTE

En este apartado vamos a crear dos Beans. El primer Bean (fuente) de nombre *Producto* tiene una propiedad ligada denominada *stockactual* de tipo *int*. El segundo Bean (receptor) de nombre *Pedido* está interesado en los cambios de dicha propiedad cuando sea inferior al stock mínimo. Supongamos que el problema que se trata de resolver es que cuando el stock actual de un producto sea inferior al stock minimo se debe generar un pedido. Crearemos dos clases con los siguientes atributos y que implementarán las siguientes interfaces:

CLASE PRODUCTO, IMPLEMENTS SERIALIZABLE. FUENTE DE EVENTOS	CLASE PEDIDO, IMPLEMENTS SERIALIZABLE, PROPERTYCHANGELISTENER. RECEPTOR DE EVENTOS

`private String descripcion;`
`private int idproducto;`
`private int stockactual;`
`private int stockminimo;`
`private float pvp;`
	`private int numeropedido;`
	`private int idproducto;`
	`private Date fecha;`
	`private int cantidad;`
	`private boolean pedir;`

Utilizaremos una herramienta para crear los JavaBeas que después usaremos en varios programas Java.

6.4.1 CREAR JAVABEANS CON NETBEANS

Abrimos NetBeans, se selecciona *Archivo-> Proyecto Nuevo-> Java-> Java ClassLibrary* y pulsamos el botón *Siguiente*; véase Figura 6.3. Se escribe el nombre de la librería Java, por ejemplo *LibreriaJava1*, la ubicación y la carpeta del proyecto y se hace clic en el botón *Terminar*, Figura 6.4.

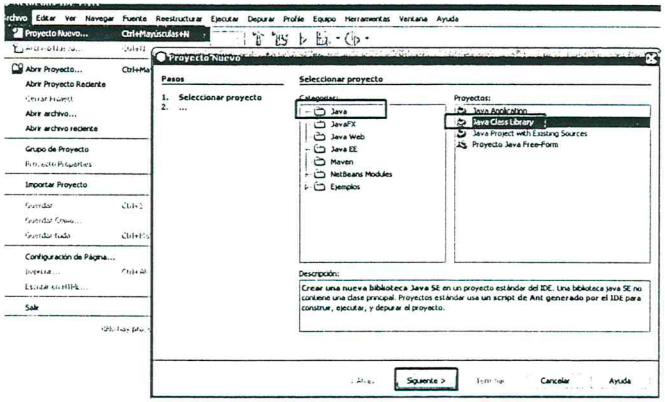


Figura 6.3. Nueva librería Java

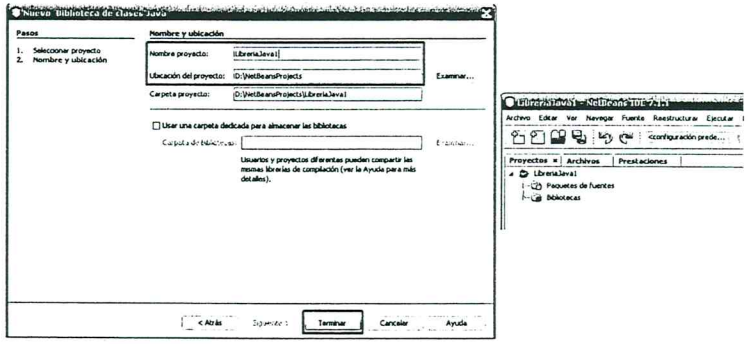


Figura 6.4. Nombre y ubicación de la librería.

Una vez creado el proyecto creamos los JavaBeans. Con el botón derecho del ratón pulsamos en *Nuevo->Componente JavaBeans* (Figura 6.5); a continuación escribimos el nombre del Bean (*Producto*) y el nombre del paquete del que formará parte, en el ejemplo *MisBeans*. Pulsamos el botón *Terminar*.

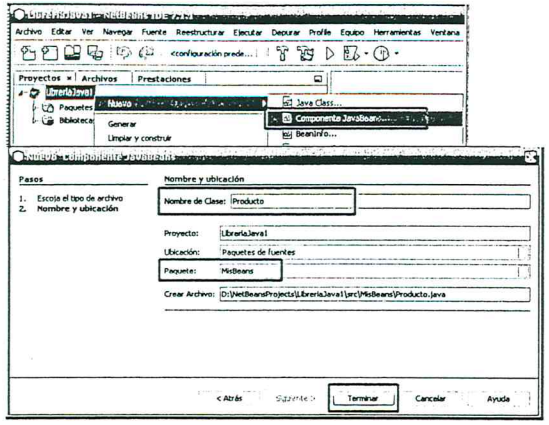


Figura 6.5. Creamos el Bean Producto.

Se crea el Bean *Producto* con un código por defecto, Figura 6.6.

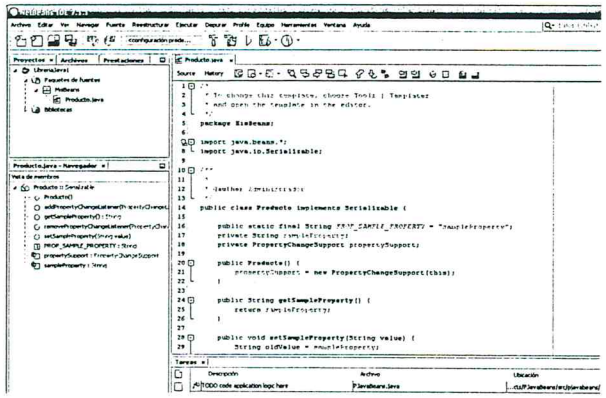


Figura 6.6. Código generado al crear el JavaBean Producto.

Dentro del código aparece una propiedad ejemplo de nombre *sampleProperty* con sus métodos *get* y *set* correspondientes, se crea un objeto **PropertyChangeSupport** de nombre *propertySupport* para conseguir un soporte sencillo a fin de definir los métodos anteriores *addPropertyChangeListener()* y *removePropertyChangeListener()*. El código es el siguiente:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package MisBeans;
```



```
import java.beans.*;
import java.io.Serializable;

/**
 *
 * @author Administrador
 */
public class Producto implements Serializable {

    public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";
    private String sampleProperty;
    private PropertyChangeSupport propertySupport;

    public Producto() {
        propertySupport = new PropertyChangeSupport(this);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY, oldValue,
                                           sampleProperty);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

Modificamos el código del JavaBean. Eliminamos el atributo *sampleProperty* y sus métodos *get* y *set* (*getSampleProperty()* y *setSampleProperty()*) y añadimos los siguientes atributos:

```
    private String descripcion;
    private int idproducto;
    private int stockactual;
    private int stockminimo;
    private float pvp;
```

Añadimos el siguiente constructor:

```
public Producto(int idproducto,String descripcion,
                int stockactual, int stockminimo, float pvp){
        propertySupport = new PropertyChangeSupport(this);
        this.idproducto =idproducto;
        this.descripcion =descripcion;
```

```
        this.stockactual=stockactual;
        this.stockminimo=stockminimo;
        this.pvp=pvp;
    }
```

Y generamos los *getter* y *setter* para los atributos haciendo clic con el botón derecho del ratón debajo de los constructores; seleccionamos *Insertar código->Getter y Setter*, a continuación marcamos los atributos para los que insertaremos dichos métodos y pulsamos el botón *Generar*, véase Figura 6.7.

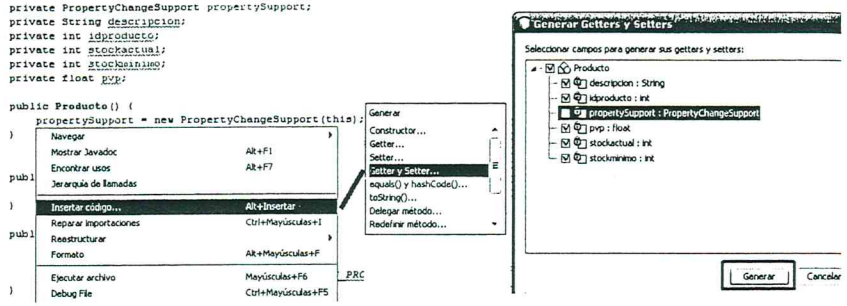


Figura 6.7. Generar los Getter y Setter.

Se cambia el método *setStockactual()* por el siguiente para generar un evento si el stock actual es menor que el mínimo (se usa el método *firePropertyChange()*):

```
public void setStockactual(int valorNuevo) {
  int valorAnterior = stockactual;
  stockactual = valorNuevo;
   if (stockactual < getStockminimo()) //hay que realizar pedido
     propertySupport.firePropertyChange("stockactual",
                                        valorAnterior, stockactual);
}
```

Creamos (como antes) el JavaBean *Pedido* en el paquete *MisBeans*. Eliminamos todo el código de la clase y añadimos **PropertyChangeListener** a la cláusula *implements*; será necesario sobreescribir el método *propertyChange()*. En este método es donde indicamos las acciones a realizar cuando el stock actual es menor que el stock minimo, debe quedar algo parecido a esto:

```
package MisBeans;

import java.beans.*;
import java.io.Serializable;

public class Pedido implements Serializable , PropertyChangeListener {

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```


Añadimos los siguientes atributos para esta clase (será necesario añadir *import java.util.Date*):

```
private int numeropedido;
private int idproducto;
private Date fecha;
private int cantidad;
private boolean pedir; //se usa para controlar si hay que hacer pedido
```

Añadimos los *getter* y los *setter* para estos atributos y cambiamos el método *propertyChange()* con las siguientes líneas:

```
public void propertyChange(PropertyChangeEvent evt) {
        System.out.println("Stock anterior: "+ evt.getOldValue());
        System.out.println("Stock actual: "+ evt.getNewValue());
        setPedir(true);     //hay que hacer pedido
}
```

Añadimos también los siguientes constructores (los usaremos después en los ejemplos que hagamos):

```
public Pedido() { }

public Pedido (int numeropedido, int idproducto, java.sql.Date fecha, int cantidad)
{
            this.numeropedido =numeropedido;
            this.idproducto=idproducto;
            this.fecha=fecha;
            this.cantidad=cantidad;
}
```

Una vez que tenemos los JavaBeans, generamos el JAR. Para ello, pulsamos con el botón derecho del ratón en el proyecto y a continuación en la opción de menú *Generar*, véase Figura 6.8. NetBeans nos mostrará los mensajes de la generación y si se ha realizado correctamente.

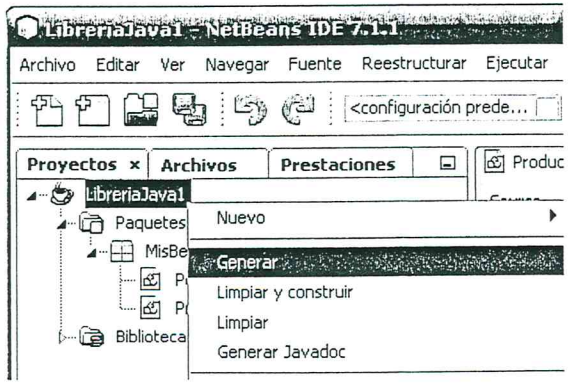


Figura 6.8. Generar archivo JAR.

Si todo va bien, se generará el archivo *LibreriaJava1.jar* en la carpeta *dist* del proyecto, en el ejemplo en *D:\NetBeansProjects\LibreriaJava1\dist*. Si posteriormente modificamos los JavaBean pulsamos en la opción *Limpiar y construir*.

Para probar los componentes creamos un nuevo proyecto Java, por ejemplo, *PruebaLibreriaJava1* con una clase con el mismo nombre e incluimos *LibreriaJava1.jar*. Para ello pulsamos con el botón derecho del ratón en *Bibliotecas-> Agregar archivo JAR/Carpeta*, véase Figura 6.9, hemos de localizar el archivo JAR y pulsar el botón *Abrir*. Una vez realizado, al abrir el nodo *Bibliotecas* podemos ver la librería incluída con el paquete *MisBeans* creado anteriormente y la carpeta *META-INF* con el archivo *MANIFEST.MF* que se genera al crear el JAR.

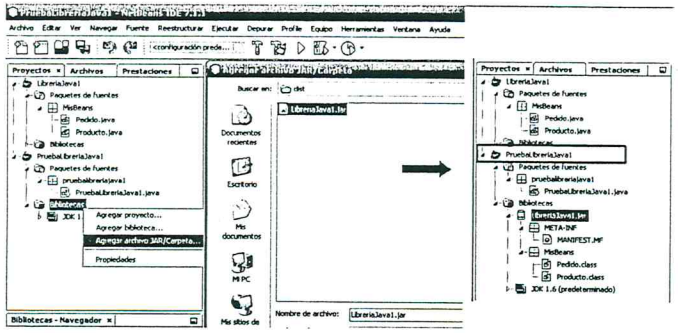


Figura 6.9. Incluir el archivo JAR en nuestro proyecto Java.

Para detectar si hay que realizar un pedido en el producto, utilizamos el método *addPropertyChangeListener()* del objeto Producto para agregar a la lista de oyentes el objeto Pedido, que es el que está interesado en el cambio del stock: *producto.addPropertyChangeListener(pedido)*. Escribimos el siguiente código en el fichero Java:

```
package pruebalibreriajava1;

import MisBeans.Pedido;
import MisBeans.Producto;

public class PruebaLibreriaJava1 {
   public static void main(String[] args) {
     Producto producto = new Producto(1,"Dabber Sur Femme 2011", 10, 3,16);
     Pedido pedido= new Pedido();

     producto.addPropertyChangeListener(pedido);
     producto.setStockactual(2);//cambiamos el stock actual, le damos el valor 2
     if(pedido.isPedir()) {
    System.out.println("REALIZAR PEDIDO EN PRODUCTO: "+producto.getDescripcion());
     }
     else
      System.out.println("NO ES NECESARIO REALIZAR PEDIDO EN PRODUCTO: " +
                              producto.getDescripcion());
   }
}
```


Y lo ejecutamos. Al ejecutarlo se visualiza:

```
Stock anterior: 10
Stock actual: 2
REALIZAR PEDIDO EN PRODUCTO: Dabber Sur Femme 2011
```

Actividad 1: Crea un proyecto en Eclipse para probar la librería (*LibreriaJava1.jar*). Define varios productos y cambia el stock actual en alguno de ellos. Visualiza por cada producto su stock mínimo.

6.5 EMPAQUETADO DE COMPONENTES

Una vez desarrollado el JavaBean se debe empaquetar para su distribución y utilización por las aplicaciones. La forma habitual de distribución es mediante un archivo JAR (vimos anteriormente como se creaba el JAR desde NetBeans), este debe contener además un fichero de manifiesto (*MANIFEST.MF*) que describa su contenido. En el JAR correspondiente a un Bean se deben incluir las clases y recursos que lo forman. El siguiente ejemplo muestra un fichero *MANIFEST.MF* donde se define un Bean de nombre *miBean*, una clase auxiliar para el Bean y una imagen:

```
Manifest-Version: 1.0
Name: ejemploBeans/miBean.class
Java-Bean: True

Name: ejemploBeans/auxiliar.class
Java-Bean: False

Name: ejemploBeans/imagen.png
Java-Bean: False
```

Hemos de asegurarnos de que exista una línea vacía entre las entradas y que la última línea termine en un carácter de salto de línea. Además no debe haber espacios en blanco al final de las líneas. Este fichero suele encontrarse en una carpeta de nombre *META-INF*. Una vez creado el fichero para empaquetar el Bean generamos el fichero JAR usando el comando **jar** de Java:

```
jar cfm archivogenerado.jar META-INF/MANIFEST.MF carpetadeclases/*.class
```

Donde:

- **c**: Indica que vamos a crear un fichero.
- **f**: Indica que la salida va a un fichero y no a la salida estándar.
- **m**: Indica que se deben añadir las líneas del fichero manifiesto que especifique el usuario.
- ***archivogenerado.jar***: Es el nombre que damos al fichero JAR de salida que contendrá todas las clases de la *carpetadeclases*.
- ***META-INF/MANIFEST.MF***: Indica el lugar donde se encuentran las líneas que se quieren añadir al fichero de manifiesto que se genera por defecto con la orden **jar**.
- ***carpetadeclases/*.class***: Indica dónde están los ficheros de entrada, en este caso son los que tienen la extensión *class* de la *carpetadeclases*.

Por ejemplo, partimos de los Beans creados anteriormente. Supongamos que tengo una carpeta en la unidad D:\ de nombre *UNI6* y dentro tengo la carpeta *MisBeans* con los Beans *Producto.java* y *Pedido.java* y una carpeta de nombre META-INF con el fichero *MANIFEST.MF*, la estructura inicial de la carpeta es la siguiente:

```
D:\UNI6
├──META-INF
│    MANIFEST.MF
│
└──MisBeans
     Pedido.java
     Producto.java
```

El contenido de nuestro fichero *MANIFEST.MF* es el siguiente:

```
Manifest-Version: 1.0
Name: MisBeans/Producto.class
Java-Bean: True

Name: MisBeans/Pedido.class
Java-Bean: True
```

Para crear el componente compilamos los JavaBeans desde la línea de comandos:

```
D:\UNI6>javac MisBeans/Producto.java
D:\UNI6>javac MisBeans/Pedido.java
```

Y generamos el paquete JAR con la siguiente orden:

```
D:\UNI6>jar cfm MisBeans.jar META-INF/MANIFEST.MF MisBeans/*.class
```

Con esto, ya tenemos nuestros JavaBeans en el paquete *MisBeans.jar* que podremos usar en cualquier programa Java.

6.6 USANDO JAVABEANS PARA ACCEDER A BASES DE DATOS

En el siguiente ejemplo vamos a añadir una clase auxiliar a nuestros Beans, la clase se llama *Venta* y se definen atributos que describen los datos de una venta de un producto, también se definen los métodos *getter* y *setter*. La clase es la siguiente:

```
package MisBeans;

import java.io.Serializable;

public class Venta implements Serializable {
    private static final long serialVersionUID = 1L;
    private int numeroventa;
```



```
    private int idproducto;
    private java.sql.Date fechaventa;
    private int cantidad;

    public Venta (){}

    public Venta(int numeroventa, int idproducto, Java.sql.Date fechaventa,
                 int cantidad) {
        this.numeroventa =numeroventa;
        this.idproducto=idproducto;
        this.fechaventa=fechaventa;
        this.cantidad=cantidad;
    }
    public int getNumeroventa() { return numeroventa; }
    public void setNumeroventa(int numeroventa) {this.numeroventa = numeroventa;}
    public int getIdproducto() { return idproducto; }
    public void setIdproducto(int idproducto) {this.idproducto = idproducto;}
    public java.sql.Date getFechaventa() {return fechaventa;}
    public void setFechaventa(java.sql.Date fechaventa)
                 {this.fechaventa = fechaventa;}
    public int getCantidad() {return cantidad;}
    public void setCantidad(int cantidad) {this.cantidad = cantidad;}
}
```

Variamos el contenido del archivo *MANIFEST.MF*, incluyendo esta nueva clase:

```
Manifest-Version: 1.0
Name: MisBeans/Producto.class
Java-Bean: True

Name: MisBeans/Pedido.class
Java-Bean: True

Name: MisBeans/Venta.class
Java-Bean: False
```

Compilamos la nueva clase y generamos el JAR, con esto tenemos el componente *MisBeans.jar* con las 3 clases:

```
D:\UNI6>javac MisBeans/Venta.java

D:\UNI6>jar cfm MisBeans.jar META-INF/MANIFEST.MF MisBeans/*.class
```

6.6.1 USANDO UN COMPONENTE PARA ACCEDER A UNA BDOO

Vamos a utilizar la BDOO **Neodatis** para gestionar las ventas y pedidos de una serie de productos. Creo una carpeta donde almacenaré mis programas Java, la carpeta se llama *PRO-PED-VEN* dentro de UNI6, dentro de esta carpeta hay una carpeta *lib* donde debo almacenar los JAR que voy a necesitar: *MisBeans.jar* y *neodatis-odb-1.9.30.689.jar*, la estructura de directorios es la siguiente:

```
D:\UNI6\PRO-PED-VEN
|   LLenarProductos.java
|   llenarVentas.java
|   VerPedidos.java
|   VerProductos.java
|   VerVentas.java
|
└──lib
       MisBeans.jar
       neodatis-odb-1.9.30.689.jar
```

Configuro la variable CLASSPATH de la siguiente manera:

```
SET CLASSPATH=.;D:\UNI6\PRO-PED-VEN\lib\MisBeans.jar;
                D:\UNI6\PRO-PED-VEN\lib\neodatis-odb-1.9.30.689.jar
```

También podemos utilizar los entornos NetBeans o Eclipse para realizar el ejemplo, sin olvidarnos de incluir los JAR en el proyecto (en este caso no hace falta configurar la variable CLASSPATH).

En primer lugar realizo un programa Java para dar de alta los productos, *LLenarProductos.java*, se almacenarán en la base de datos *Producto_Ped.BD*:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

import MisBeans.Producto;

public class LLenarProductos {
    public static void main(String[] args) {

     ODB odb = ODBFactory.open("Producto_Ped.BD");// Abrir BD
     Producto p1 =   new Producto(1,"Dabber Sur Femme 2011",  10, 3,160);
     Producto p2 =   new Producto(2,"Varlion Avant Carbon Oversize", 5, 2,176);
     Producto p3 =   new Producto(3,"Star Vie Pyramid R50", 20, 5, 193);
     Producto p4 =   new Producto(4,"Dunlop Titan Star amarilla", 8, 3,85);
     Producto p5 =   new Producto(5,"Vision King jm", 7, 1,159);
     Producto p6 =   new Producto(6,"Slazenger Reflex Pro", 5, 2,80);

     //Almacenamos productos
     odb.store(p1);
     odb.store(p2);
     odb.store(p3);
     odb.store(p4);
     odb.store(p5);
     odb.store(p6);

     odb.close(); // Cerrar BD
   }
}
```

Para visualizar los productos creo el programa *VerProductos.java*:


```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

import MisBeans.Producto;

public class VerProductos {
    public static void main(String[] args) {

     ODB odb = ODBFactory.open("Producto_Ped.BD");// Abrir BD
     //recuperamos todos los objetos
     Objects<Producto> objects = odb.getObjects(Producto.class);
     System.out.println(objects.size() + " Productos:");

     int i = 1;
     // visualizar los productos
     while(objects.hasNext()){
       Producto pro = objects.next();
        System.out.println((i++) + "\t: " + pro.getDescripcion()
              + " *STOCK ACTUAL:" + pro.getStockactual()+ " * PVP:" +
              pro.getPvp()+ " * MINIMO: " +  pro.getStockminimo());
     }
     odb.close(); // Cerrar BD
   }
}
```

Los programas *VerPedidos.java* y *VerVentas.java* son similares a *VerProductos.java*. En este caso los objetos a visualizar son pedidos y ventas. Para crear las ventas creo el programa *llenarVentas.java*. Los datos para este programa son el *idproducto* del que se va a realizar la venta y la *cantidad* que se vende. Se realizará una venta siempre y cuando haya stock y el stock actual que quede tras la venta no sea inferior al stock mínimo; si el stock actual es inferior al mínimo se generará un pedido y no se realizará la venta ni se actualizará el stock del producto. El código es el siguiente:

```
import java.math.BigDecimal;
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Objects;
import org.neodatis.odb.Values;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;

import MisBeans.Pedido;
import MisBeans.Producto;
import MisBeans.Venta;

public class llenarVentas {
   public static void main(String[] args) {

   // Datos de entrada para la venta
   int idproducto=2; // idproducto
```

```
int cantidad=4;   // cantidad

ODB odb = ODBFactory.open("Producto_Ped.BD"); // Abrir BD
//Comprobamos si existe el producto
IQuery query = new CriteriaQuery(Producto.class,
                    Where.equal("idproducto", idproducto));
Objects<Producto> objetos = odb.getObjects(query);

try{
   // Obtiene solo el primer objeto encontrado
   Producto pro = (Producto) objetos.getFirst();
   System.out.println("ID=>" +idproducto + ": "+ pro.getDescripcion()
            + "*STOCK-ACT:" + pro.getStockactual()+ "*PVP:" + pro.getPvp()+
              "*STOCK-MIN:" + pro.getStockminimo());

   if(cantidad >0){    //comprobar cantidad > que 0
     java.sql.Date fechaActual = getCurrentDate(); //Obtener fecha

     System.out.println("CANTIDAD A VENDER :"+ cantidad);

     if(actualizarStock(pro, odb, cantidad)) {
       int numeroventa=obtenerNumeroVenta(odb);
       Venta ven =new Venta(numeroventa,idproducto,fechaActual,cantidad);
       odb.store(ven); //almacenar venta
       System.out.println("VENTA : "+ numeroventa +" INSERTADA...");
       }
     else
       System.out.println("VENTA NO INSERTADA - NO HAY STOCK...");
   } else {
   System.out.println("LA CANTIDAD DEBE SER MAYOR QUE 0");
   }
}catch (IndexOutOfBoundsException e){
 System.out.println("NO EXISTE EL PRODUCTO");
}finally {
   odb.close(); // Cerrar BD
}
}//fin llenarVenta (main)
```

Lo primero que hacemos es una consulta a la BD (con *IQuery*) para obtener todos los objetos que cumplan la condición (en este caso será un producto), con esto se comprueba si el producto existe:

```
IQuery query = new CriteriaQuery(Producto.class,
                    Where.equal("idproducto", idproducto));
```

Si no existe se produce la excepción *IndexOutOfBoundsException* donde se visualiza un mensaje indicándolo. Si el producto existe se visualizan sus datos y se comprueba si la cantidad es > que 0, si no es mayor que 0 se visualiza un mensaje indicándolo.

Si la cantidad es > 0 se obtiene la fecha actual y se invoca a la función *actualizarStock()*, que devuelve *true* si la venta se puede llevar a cabo y *false* en caso contrario. Si la venta se puede llevar a cabo se genera el último número de venta llamando a la function *obtenerNumeroVenta()*. Una vez que tenemos el número de venta, la almacenamos en la BD.


```
//-----------------------------------Actualizar stock o Generar pedido---------
  private static boolean actualizarStock(Producto producto, ODB odb, int cantidad)
  {
      Pedido pedido= new Pedido();
      java.sql.Date fechaActual = getCurrentDate();
      producto.addPropertyChangeListener(pedido);//añadir oyente
      int nuevostock = producto.getStockactual()- cantidad;//cálculo de stock
      boolean actualizar=false;

      producto.setStockactual(nuevostock);//cambiamos el stock actual

      if(pedido.isPedir()) {
       System.out.println("REALIZAR PEDIDO EN PRODUCTO: " +
                          producto.getDescripcion() + " CANTIDAD:"+ cantidad);
       int numeropedido = obtenerNumeroPedido(odb);
       pedido.setCantidad(cantidad);
       pedido.setIdproducto(producto.getIdproducto());
       pedido.setNumeropedido(numeropedido);
       pedido.setFecha(fechaActual);
       odb.store(pedido); //Generaar pedido
       System.out.println("PEDIDO " +numeropedido +" GENERADO ...");
     }
      else {
         odb.store(producto); //actualizar stock en BD
         System.out.println("STOCK ACTUALIZADO ...");
         actualizar=true;
      }
      return actualizar;
  }//fin actualizarStock
```

La función *actualizarStock()* crea un objeto *Pedido* y lo añade a la lista de oyentes. Se modifica el stock actual y se realizará un pedido si es menor que el mínimo, en este caso no se realizará la actualización del stock del producto en la BD. La función devuelve *true* si el stock ha sido actualizado y *false* en caso contrario. La función *obtenerNumeroPedido()* obtiene el siguiente número de pedido a insertar en la base de datos.

```
//--------------------------------Obtener el n° de pedido a insertar en la BD--
private static int obtenerNumeroPedido(ODB odb) {
   Values  val4 = odb.getValues(new ValuesCriteriaQuery(Pedido.class)
                        .max("numeropedido","ped_max"));
   ObjectValues ov4 = val4.nextValues();
   BigDecimal maxima = (BigDecimal) ov4.getByAlias("ped_max");

   return maxima.intValue()+1;
}//fin obtenerNumeroPedido

//-----------------------------------Obtener el n° de venta insertar en la BD--
private static int obtenerNumeroVenta(ODB odb) {
   Values  val4 = odb.getValues(new ValuesCriteriaQuery(Venta.class)
                        .max("numeroventa","ven_max"));
   ObjectValues ov4 = val4.nextValues();
   BigDecimal maxima = (BigDecimal) ov4.getByAlias("ven_max");

   return maxima.intValue()+1;
```

```
  }//fin obtenerNumeroVenta

  //---------------------------------- Obtener la fecha actual --------------------
  private static java.sql.Date getCurrentDate() {
    java.util.Date hoy = new java.util.Date();
    return new java.sql.Date(hoy.getTime());
  }
}//fin de la clase
```

Un ejemplo de la ejecución para una venta que no se puede insertar en la base de datos porque deja un stock actual inferior al stock mínimo, en este caso se genera un pedido:

```
D:\uni6\PRO-PED-VEN>java llenarVentas
ID=>2: Varlion Avant Carbon Oversize*STOCK-ACT:5*PVP:176.0*STOCK-MIN:2
CANTIDAD A VENDER :4
Stock anterior: 5
Stock actual: 1
REALIZAR PEDIDO EN PRODUCTO: Varlion Avant Carbon Oversize CANTIDAD:4
PEDIDO 1 GENERADO ...
VENTA NO INSERTADA - NO HAY STOCK...
```

Actividad 2: Como ejercicio se propone que el IDPRODUCTO y la CANTIDAD se acepten desde la línea de comandos. Hazlo también para que se acepten desde la entrada estándar.

6.6.2 USO DE UN COMPONENTE PARA ACCEDER A BASES DE DATOS SQL

En el siguiente ejemplo vamos a desarrollar un componente que nos permitirá acceder a una base de datos basada en SQL. Las tablas que usaremos se llaman PRODUCTOS, VENTAS y PEDIDOS y las columnas de estas tablas son similares a los atributos de los Beans creados anteriormente. La estructura de tablas se muestra en la Figura 6.10.

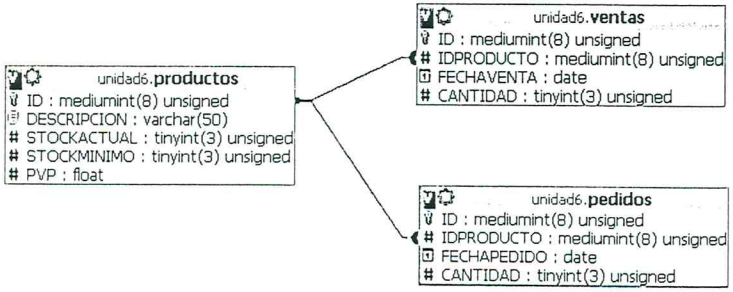


Figura 6.10. Base de datos Unidad 6.

Partimos del componente *MisBeans* desarrollado anteriormente, vamos a añadir una nueva clase donde incluiremos las operaciones a realizar con la base de datos como por ejemplo la conexión con la misma, la obtención de una lista de pedidos, productos o ventas, inserción de una venta, consulta de un producto, etc. La clase se llama *BaseDatos* (la creamos en la misma carpeta donde estaban los Beans) y se definen los atributos necesarios para realizar la conexión con la base de datos:

```
private static Connection conexion; //objeto Connection para conectarse a la BD
private String urldb;   //url para acceder a la BD
private String usuario; //nombre de usuario
private String clave;   //clave del usuario
private  String driver; //driver para la conexión a la BD
private boolean crearConexion; //atributo para saber si la conexión está creada
```

Y los siguientes métodos:

- Desde el constructor con parámetros se inician los atributos para la conexión a la BD.
- ***public void*** *setCrearConexion():* carga el driver y establece la conexión con la BD.
- ***public boolean*** *getCrearConexion():* devuelve el valor del atributo *crearConexion*.
- ***public Connection*** *getConexion():* devuelve el objeto *conexion*.
- ***public void*** *cerrarConexion():* cierra la conexión a la base de datos.
- ***public ArrayList<Producto>*** *consultaPro(String consulta):* devuelve una lista de productos según el parámetro *consulta* que recibe.
- ***public ArrayList<Pedido>*** *consultaPed(String consulta):* devuelve una lista de pedidos según el parámetro *consulta* que recibe.
- ***public ArrayList<Venta>*** *consultaVen(String consulta)*: devuelve una lista de ventas según el parámetro *consulta* que recibe.
- ***public int*** *obtenerUltimoID(String tabla)*: obtiene el último identificador de la tabla que recibe como parámetro.
- ***public int*** *insertarVenta(Venta ven)*: inserta una venta, recibe un objeto Venta.
- ***public int*** *actualizarStock(Producto producto, int cantidad, java.sql.Date fechaActual)*: realiza las operaciones para actualizar o no el stock del producto o realizar el pedido del mismo.
- ***public Producto*** *consultaUnProducto(int idproducto)*: obtiene un producto a partir de su ID.

El código de la clase *BaseDatos* (fichero *BaseDatos.java*) es el siguiente:

```
package MisBeans;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class BaseDatos {
  private static Connection conexion; //objeto Connection para conectarse a la BD
  private String urldb;   //url para acceder a la BD
  private String usuario; //nombre de usuario
```

```
    private String clave;    //clave del usuario
    private  String driver; //driver para la conexión a la BD
    private boolean crearConexion; //atributo para saber si la conexión está creada

    //Constructores
    public BaseDatos () { }// Fin del Constructor

    public  BaseDatos(String urldb, String usuario, String clave, String driver) {
      this.urldb=urldb;
      this.usuario=usuario;
      this.clave=clave;
      this.driver=driver;
    }
    //
    public void setCrearConexion()//carga el driver y establece la conexión con la BD
    {
      crearConexion =false;
      try {
           Class.forName(driver);
           conexion = DriverManager.getConnection(urldb,usuario,clave);
           crearConexion =true;
        } catch(Exception e) {
                System.out.println("PROBLEMAS EN LA CONEXION ....");
                e.printStackTrace();
        }
    }
    // devuelve el valor del atributo crearConexion
    public boolean getCrearConexion() { return crearConexion; }
    // devuelve el objeto conexion.
    public Connection getConexion() { return conexion; }
    //
    public void cerrarConexion(){// cierra la conexión a la BD
      try {
             conexion.close();
      } catch (SQLException e) {
             e.printStackTrace();
      }
   }
  //-----------------------------------------------------------------------------
  //Devuelve una lista de productos consultados
  public ArrayList<Producto> consultaPro(String consulta) {
      ArrayList <Producto> lista = new ArrayList<Producto>();
      try {
             Statement sentencia = getConexion().createStatement();
             ResultSet resul = sentencia.executeQuery(consulta);
             while (resul.next())
                {
                 Producto p= new Producto(resul.getInt(1),resul.getString(2),
                           resul.getInt(3), resul.getInt(4),resul.getFloat(5));
                 lista.add(p);
                }
             resul.close();
             sentencia.close();
      } catch (SQLException e) {
             System.out.println("PROBLEMAS AL CONSULTAR PRODUCTOS ....");
```



```
             e.printStackTrace();
      }
      return lista;
  }
  //-----------------------------------------------------------------------------
  //Obtiene la lista de pedidos
  public ArrayList<Pedido> consultaPed(String consulta) {
      ArrayList <Pedido> lista = new ArrayList<Pedido>();
      try {
             Statement sentencia = getConexion().createStatement();
             ResultSet resul = sentencia.executeQuery(consulta);
             while (resul.next())
                {
                     Pedido p= new Pedido(resul.getInt(1),resul.getInt(2),
                                  resul.getDate(3), resul.getInt(4));
                     lista.add(p);
                }
             resul.close();
             sentencia.close();
      } catch (SQLException e) {
             System.out.println("PROBLEMAS AL CONSULTAR PEDIDOS ....");
             e.printStackTrace();
      }
      return lista;
  }
  //-----------------------------------------------------------------------------
  //Obtiene la lista de ventas
  public ArrayList<Venta> consultaVen(String consulta) {
      ArrayList <Venta> lista = new ArrayList<Venta>();
      try {
             Statement sentencia = getConexion().createStatement();
             ResultSet resul = sentencia.executeQuery(consulta);
             while (resul.next())
                {
                     Venta p= new Venta(resul.getInt(1),resul.getInt(2),
                                  resul.getDate(3), resul.getInt(4));
                     lista.add(p);
                }
             resul.close();
             sentencia.close();
      } catch (SQLException e) {
             System.out.println("PROBLEMAS AL CONSULTAR VENTAS ....");
             e.printStackTrace();
      }
      return lista;
  }
  //-----------------------------------------------------------------------------
  //Obtener ultimo identificador de la tabla que recibe
  public int obtenerUltimoID(String tabla) {
       int id=0;
       String consulta= "SELECT MAX(ID) FROM "+tabla;
       try {
         Statement sentencia = getConexion().createStatement();
         ResultSet resul = sentencia.executeQuery(consulta);
         resul.next();
```

```
        id=resul.getInt(1)+1;
        resul.close();
        sentencia.close();
      } catch (SQLException e) {
                System.out.println("PROBLEMAS AL OBTENER MAXIMO ID EN "+tabla);
                e.printStackTrace();
            }
      return id;
  }//fin obtenerUltimoID
//-----------------------------------------------------------------------------
//Insertar Una Venta
public int insertarVenta(Venta ven) {
    int filas=0;
    String sql= "INSERT INTO VENTAS VALUES ( ?, ?, ? , ?)";
    try {
       PreparedStatement sentencia = getConexion().prepareStatement(sql);
       sentencia.setInt(1,ven.getNumeroventa()); // num venta
       sentencia.setInt(2,ven.getIdproducto());  // idproducto
       sentencia.setDate(3,(Date) ven.getFechaventa()); // fecha
       sentencia.setInt(4,ven.getCantidad());       // cantidad
       filas = sentencia.executeUpdate();  // filas afectadas
       sentencia.close();
    } catch (SQLException e) {
                System.out.println("ERROR AL INSERTAR VENTA");
                e.printStackTrace();
    }
    return filas;
}
//-----------------------------------------------------------------------------
//Actualizar stock y generar pedido
public int actualizarStock(Producto producto, int cantidad,
                                        java.sql.Date fechaActual) {
    Pedido pedido= new Pedido();
    producto.addPropertyChangeListener(pedido); //añadimos el oyente
    int nuevostock = producto.getStockactual()- cantidad;//calculo de stock

    producto.setStockactual(nuevostock);//cambiamos el stock actual

    int filas=0;
    String sql= "";
    PreparedStatement sentencia=null;
    try {
       if(pedido.isPedir()) {//NO se actualiza stock en BD
                System.out.println("REALIZAR PEDIDO EN PRODUCTO: " +
                                        producto.getDescripcion());

                String tabla="PEDIDOS";
                int numeropedido = obtenerUltimoID(tabla);
                pedido.setCantidad(cantidad);
                pedido.setIdproducto(producto.getIdproducto());
                pedido.setNumeropedido(numeropedido);
                pedido.setFecha(fechaActual);
```



```
                sql= "INSERT INTO PEDIDOS VALUES ( ?, ?, ?, ?)";
                sentencia = getConexion().prepareStatement(sql);
                sentencia.setInt(1,pedido.getNumeropedido()); // num pedido
                sentencia.setInt(2,pedido.getIdproducto());  // idproducto
                sentencia.setDate(3,(Date) pedido.getFecha()); // fecha
                sentencia.setInt(4,pedido.getCantidad());       // cantidad
                filas = sentencia.executeUpdate();  // filas afectadas
                System.out.println("PEDIDO " +numeropedido +" GENERADO ...");
                filas=-1;//NO SE PUEDE REALIZAR VENTA
                sentencia.close();
       }
       else //Se actualiza stock en BD
       {
              sql= "UPDATE PRODUCTOS SET STOCKACTUAL = ? WHERE  ID = ?";
              sentencia = getConexion().prepareStatement(sql);
              sentencia.setInt(1,producto.getStockactual()); // stock
              sentencia.setInt(2,producto.getIdproducto());  // idproducto
              filas = sentencia.executeUpdate();  // filas afectadas
              System.out.println("STOCK ACTUALIZADO ...");
              sentencia.close();
       }
    } catch (SQLException e) {
          System.out.println("PROBLEMAS AL ACTUALIZAR STOCK ....");
          filas=-1;//NO SE PUEDE REALIZAR VENTA
    }
   return filas;
}
//-----------------------------------------------------------------------------
// Obtiene un producto a partir de su ID
public Producto consultaUnProducto(int idproducto) {
    Producto p=null;
    String consulta= "SELECT * FROM PRODUCTOS WHERE ID = ? ";
    try {
       PreparedStatement sentencia = getConexion().prepareStatement(consulta);
       sentencia.setInt(1,idproducto);
       ResultSet resul = sentencia.executeQuery();
       resul.next();
       p= new Producto(resul.getInt(1),resul.getString(2),
                        resul.getInt(3), resul.getInt(4),resul.getFloat(5));
       resul.close();
       sentencia.close();
    } catch (SQLException e) {
          System.out.println("PROBLEMAS AL OBTENER EL PRODUCTO CON ID: "
                                                          +idproducto);
          e.printStackTrace();
    }
    return p; //devuelve un producto
}
//
} //Fin Clase BaseDatos
```

Añadimos al fichero *MANIFEST.MF* las siguientes líneas:

```
Name: MisBeans/BaseDatos.class
Java-Bean: False
```

Compilamos la nueva clase y generamos el JAR, de esta manera tendremos en el componente *MisBeans* todas las clases; lo hacemos desde la línea de comandos (también se puede hacer desde Eclipse o NetBeans):

```
D:\UNI6>javac MisBeans/BaseDatos.java
D:\UNI6>jar cfm MisBeans.jar META-INF/MANIFEST.MF MisBeans/*.class
```

El siguiente ejemplo muestra el uso del componente *MisBeans*. Se crea la clase *Ejemplo* en la carpeta D:\UNI6\BD-SQL para acceder a una base de datos MySQL de nombre y clave *unidad6* en la que tenemos las tablas PRODUCTOS, VENTAS y PEDIDOS. El código es el siguiente:

```
import java.util.ArrayList;

import MisBeans.Pedido;
import MisBeans.Producto;
import MisBeans.Venta;
import MisBeans.BaseDatos;

public class Ejemplo {
  public static void main(String[] args) {
   /*      MYSQL*/
   String urldb= "jdbc:mysql://localhost/unidad6";
   String usuario ="unidad6";
   String clave ="unidad6";
   String driver ="com.mysql.jdbc.Driver";

   //Se crea un objeto BaseDatos
   BaseDatos bd = new BaseDatos(urldb, usuario,clave, driver);
   bd.setCrearConexion();//se crea la conexión a la BD

   if (bd.getCrearConexion()){
    System.out.println("Conectado");

    System.out.println("==============================================");
    System.out.println("LISTA INICIAL DE PRODUCTOS");
    VerProductos(bd);

    //creo una venta
    System.out.println("==============================================");
    System.out.println("CREAMOS VENTA EN ID 3 CON CANTIDAD 2 ");
    CrearVenta(bd, 3, 2);//SI NO HAY STOCK NO SE CREA VENTA

    System.out.println("==============================================");
    System.out.println("LISTA DE PRODUCTOS DESPUÉS DE CREAR VENTA");
    VerProductos(bd);

    System.out.println("==============================================");
    System.out.println("LISTA DE VENTAS");
    VerVentas(bd);

    System.out.println("==============================================");
    System.out.println("LISTA DE PEDIDOS");
    VerPedidos(bd);
    }
```



```
    else  System.out.println("NO conectado");
   //CERRAR CONEXION
   bd.cerrarConexion();
  }

  //-----------------------------------------------------------------------------
  // Visualizar los productos
  private static void VerProductos(BaseDatos bd) {
     ArrayList <Producto> lista = new ArrayList<Producto>();
     lista = bd.consultaPro("SELECT * FROM PRODUCTOS");
     if(lista!=null)
       for(int i=0;i<lista.size();i++){
        Producto p=(Producto)lista.get(i);
        System.out.println("ID=>" +p.getIdproducto() + ": "+
             p.getDescripcion() + "* Stock: " + p.getStockactual()+ "* Pvp: " +
             p.getPvp()+ "* STK Minimo:" + p.getStockminimo());
               }
  }
  //-----------------------------------------------------------------------------
  // Vamos a insertar una venta
  private static void CrearVenta(BaseDatos bd,int idproducto, int cantidad) {

     Producto prod = bd.consultaUnProducto(idproducto);
     java.sql.Date fechaActual = getCurrentDate();    //fecha sql
     if(prod!=null){
        if(bd.actualizarStock(prod, cantidad, fechaActual)>0){//hay stock
          String tabla="VENTAS";
          int idventa=bd.obtenerUltimoID(tabla);
          Venta ven =new Venta(idventa,prod.getIdproducto(),
                                                fechaActual,cantidad);
          if(bd.insertarVenta(ven)>0)
               System.out.println("VENTA INSERTADA ....");
      }
      else
        System.out.println("NO SE PUEDE REALIZAR LA VENTA, NO HAY STOCK ....");
     }
     else
               System.out.println("NO EXISTE EL PRODUCTO");
  }
  //-----------------------------------------------------------------------------
  // Ver pedidos creados
  private static void VerPedidos(BaseDatos bd) {
     ArrayList <Pedido> lista = new ArrayList<Pedido>();
     lista = bd.consultaPed("SELECT * FROM PEDIDOS");
     if(lista!=null)
          for(int i=0;i<lista.size();i++){
               Pedido p=(Pedido)lista.get(i);
               Producto prod = bd.consultaUnProducto(p.getIdproducto());
               System.out.println("ID Pedido=>" +p.getNumeropedido() +
                  "* Producto: "+ prod.getDescripcion() + "* Cantidad: " +
                   p.getCantidad()+ "* Fecha: " + p.getFecha());
          }
  }
  //-----------------------------------------------------------------------------
  // Ver ventas creadas
```

```
 private static void VerVentas(BaseDatos bd) {
      ArrayList <Venta> lista = new ArrayList<Venta>();
     lista = bd.consultaVen("SELECT * FROM VENTAS");
     if(lista!=null)
        for(int i=0;i<lista.size();i++){
          Venta p=(Venta)lista.get(i);
          Producto prod = bd.consultaUnProducto(p.getIdproducto());
          System.out.println("ID Venta =>" +p.getNumeroventa() +
           "* Producto: "+ prod.getDescripcion()
           + "* Cantidad: " + p.getCantidad()+ "* Fecha: " + p.getFechaventa());
        }
 }
 //-----------------------------------------------------------------------------
 // Obtiene la fecha actual
 private static java.sql.Date getCurrentDate() {
        java.util.Date hoy = new java.util.Date();
        return new java.sql.Date(hoy.getTime());
 }
}//Fin de Ejemplo
```

Dentro de esta carpeta creamos una carpeta de nombre *lib* y almacenamos los JAR que vamos a necesitar: el componente *MisBeans* y los conectores para acceder a las bases de datos, en este caso se han incluido el conector para acceder a MySQL y a Oracle:

```
D:\UNI6\BD-SQL>
|
|   Ejemplo.java
└──lib
        classes12.jar
        MisBeans.jar
        mysql-connector-java-5.1.18-bin.jar
```

Definimos la variable CLASSPATH:

```
SET CLASSPATH=.;D:\UNI6\BD-SQL\lib\MisBeans.jar;D:\UNI6\BD-SQL\lib\classes12.jar;D:\
UNI6\BD-SQL\lib\mysql-connector-java-5.1.18-bin.jar
```

Compilamos el ejemplo y lo ejecutamos:

```
D:\UNI6\BD-SQL>javac Ejemplo.java
D:\UNI6\BD-SQL>java Ejemplo

Conectado
=============================================
LISTA INICIAL DE PRODUCTOS
ID=>1: Dabber Sur Femme 2011* Stock: 10* Pvp: 160.0* STK Minimo:3
ID=>2: Varlion Avant Carbon Oversize* Stock: 2* Pvp: 176.0* STK Minimo:2
ID=>3: Star Vie Pyramid R50* Stock: 16* Pvp: 193.0* STK Minimo:5
ID=>4: Dunlop Titan Star amarilla* Stock: 8* Pvp: 85.0* STK Minimo:3
ID=>5: Vision King jm* Stock: 7* Pvp: 159.0* STK Minimo:1
ID=>6: Slazenger Reflex Pro* Stock: 5* Pvp: 80.0* STK Minimo:2
=============================================
CREAMOS VENTA EN ID 3 CON CANTIDAD 2
```



```
STOCK ACTUALIZADO ...
VENTA INSERTADA ....
=============================================
LISTA DE PRODUCTOS DESPUÉS DE CREAR VENTA
ID=>1: Dabber Sur Femme 2011* Stock: 10* Pvp: 160.0* STK Minimo:3
ID=>2: Varlion Avant Carbon Oversize* Stock: 2* Pvp: 176.0* STK Minimo:2
ID=>3: Star Vie Pyramid R50* Stock: 14* Pvp: 193.0* STK Minimo:5
ID=>4: Dunlop Titan Star amarilla* Stock: 8* Pvp: 85.0* STK Minimo:3
ID=>5: Vision King jm* Stock: 7* Pvp: 159.0* STK Minimo:1
ID=>6: Slazenger Reflex Pro* Stock: 5* Pvp: 80.0* STK Minimo:2
=============================================
LISTA DE VENTAS
ID Venta =>1* Producto: Star Vie Pyramid R50* Cantidad: 2* Fecha: 2012-05-15
=============================================
LISTA DE PEDIDOS
```

Para acceder con el mismo componente a Oracle solo cambiaríamos las siguientes líneas:

```
String urldb= "jdbc:oracle:thin:@localhost:1521:XE";
String usuario ="unidad6";
String clave ="unidad6";
String driver ="oracle.jdbc.driver.OracleDriver";
```

Necesitaríamos crear un usuario de nombre y clave *unidad6* y dentro del usuario tienen que existir las tablas PRODUCTOS, PEDIDOS y VENTAS con una estructura similar a la expuesta anteriormente.

Actividad 3: Utiliza el componente anterior para acceder a una BD Oracle y SQLite. Modifica la clase Ejemplo para que los datos de la venta se introduzcan desde la línea de comandos o los pida desde la entrada estándar. Añade a la clase *BaseDatos* un nuevo método que nos permita insertar un producto en la base de datos y genera de nuevo el JAR. Después realiza un programa que utilice el método para insertar un producto.

6.7 EJERCICIOS

❶ ¿Que es un componente? Cita algunas ventajas e inconvenientes del uso de componentes.

❷ ¿Qué características debe tener un componente?

❸ ¿Cuál es la diferencia entre el modelo de componentes y la plataforma de componentes?

❹ ¿Qué características debe cumplir un JavaBean?

❺ Una librería quiere realizar la gestión de las ventas de libros que realiza a los clientes. Nos pide diseñar una aplicación para gestionar los libros que hay en la misma (altas, bajas y modificaciones), los clientes (que son los que se llevan los libros, también se gestionarán altas, bajas y modificaciones) y las ventas.

Cuando un cliente viene a la librería a comprar libros (por ejemplo, supongamos que se lleva 3 libros diferentes y una unidad de cada libro) los registramos en una venta, cada libro que se lleve será una línea de venta de esa venta; por tanto una venta puede estar formada por varias líneas de venta o por una única línea si solo se lleva un libro. La información necesaria para las líneas de venta es la siguiente: número de venta, número de línea en esa venta, ISBN del libro, la cantidad que se lleva y el pvp.

Cada vez que se realice una venta será necesario actualizar el stock actual de los libros que se lleve. No se podrá vender un libro si no hay stock actual suficiente, es decir, antes de añadir una línea de venta a una venta será necesario comprobar si hay stock.

A un cliente se le pueden realizar las ventas que quiera, y en una venta no puede haber varias líneas de venta con el mismo libro (mismo ISBN). No se permite dar de baja un cliente si tiene ventas. Tampoco se puede dar de baja un libro si aparece en alguna línea de venta.

La aplicación debe permitir consultar productos, clientes y ventas. Las consultas nos han de permitir movernos de un registro a otro, al anterior, al último o al primero. La consulta de ventas podrá realizarse por cliente y por número o identificador de venta.

Datos a tener en cuenta en los libros: ISBN, título, autor, stock actual, PVP, año de publicación, editorial, temática, ubicación y fecha de alta. Tanto para obtener la editorial como la temática de los libros puedes usar tablas y gestionarlas (es decir realizar altas bajas, modificaciones y consultas) o usar listas de elementos.

En los clientes necesitamos su DNI, nombre y apellidos y la dirección. Para las ventas necesitaremos el identificador de la venta, la fecha y los datos de las líneas de venta: ISBN, cantidad, pvp, etc.

Diseña las tablas o el modelo de objetos que consideres necesario y desarrolla la aplicación en lenguaje Java definiendo componentes, de tal forma que se puedan usar en bases de datos diferentes, por ejemplo en bases de datos SQL como Oracle y MySQL o en bases de datos OO como Neodatis y Db4o.