Alicia Ramos Martín
Jesús Ramos Martín

Acceso a Datos

2ª Edición

Técnico Superior en **Desarrollo de Aplicaciones Multiplataforma**

Garceta

Acceso a Datos. 2ª Edición

Alicia Ramos Martín
Mª Jesús Ramos Martín

ISBN: 978-84-1622-860-7
IBERGARCETA PUBLICACIONES, S.L., Madrid 2016

Edición: 2.ª
Impresión: 1.ª
N.º de páginas: 434
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 © 2016 IBERGARCETA PUBLICACIONES, S.L.
info@garceta.es

Acceso a Datos. 2ª Edición

© Alicia Ramos Martín, Mª Jesús Ramos Martín

2.ª edición, 1.ª impresión
OI: 430/2016
ISBN: 978-84-1622-860-7
Deposito Legal: M-27537-2016
Imagen de cubierta: © Andrés Sanz Mulas

Impresión: Print House, marca registrada de Coplar, S.A.

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.

PRÓLOGO

Este libro sigue fielmente los contenidos expuestos en el módulo de **Acceso a datos** perteneciente a la titulación de *Técnico Superior en Desarrollo de Aplicaciones Multiplataforma* (Real Decreto 450/2010, de 16 de abril, publicado en el 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, y además el uso de ***JAXB***, tecnología Java que permite mapear clases *Java* a representaciones XML, y viceversa.

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, Access...) y orientados a objetos (Db4o). Además. se añade cómo crear informes utilizando plantillas ***jrxml*** de ***JasperReport***.

En el tercer capítulo se utilizará Hibernate como herramienta de mapeo Objeto-Relacional (ORM) para acceder a Oracle y MySQL.

En el cuarto capítulo se estudian las bases de datos objeto-relacionales. Se estudiará Oracle con los elementos que ofrece para convertir un modelo relacional en modelo a objetos, para ello utilizaremos SQL y PL/SQL. Además, se estudia la base de datos orientada a objetos ***Neodatis***, se harán programas Java para hacer altas, bajas, modificaciones y consultas en este tipo de base de datos.

En el capítulo quinto se estudian las bases de datos ***NoSQL***, y nos centraremos en dos bases de datos orientadas a documentos: la BD ***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. Y la base de datos ***MongoDB***, como base de datos de documentos ***JSON***, igualmente se crearán colecciones *JSON* y se realizarán altas, bajas y modificaciones de documentos *JSON*.

En el capítulo sexto se estudiarán los ***JavaBeans*** y los patrones de diseño ***DAO*** y ***FACTORY*** para construir componentes de acceso a datos y usarlos en diferentes aplicaciones. Se desarrollará una aplicación web según el patrón Modelo-Vista-Controlador (***MVC***), en esta aplicación el acceso a los datos se realizará usando los componentes desarrollados en el capítulo.

Los recursos de cada capítulo se pueden descargar desde esta URL:
https://sites.google.com/site/libroaccesoadatos/

ÍNDICE

CAPÍTULO 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 6
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 9
1.5.1. Operaciones sobre ficheros secuenciales 10
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. Objetos en ficheros binarios 18
1.6.4. Ficheros de acceso aleatorio 22
1.7. Trabajo con ficheros XML 26
1.7.1. Acceso a ficheros XML con DOM 27
1.7.2. Acceso a ficheros XML con SAX 33
1.7.3. Serialización de objetos a XML 36
1.7.4. Conversión de ficheros XML a otro formato 40
1.8. Excepciones: detección y tratamiento 42
1.8.1. Capturar excepciones 42
1.8.2. Especificar excepciones 45
1.9. Introducción a JAXB 46
1.9.1. Mapear clases Java a representaciones XML 47
1.9.2. Paso de esquemas XML (.xsd) a clases Java 52
1.9.3. Creación de una aplicación JAXB con Eclipse 56
COMPRUEBA TU APRENDIZAJE 63

CAPÍTULO 2. MANEJO DE CONECTORES **65**

2.1. Introducción 66
2.2. El desfase objeto-relacional 66
2.3. Bases de datos embebidas 66
2.3.1. SQLite 67
2.3.2. Apache Derby 68
2.3.3. HSQLDB 71
2.3.4. H2 73
2.3.5. Db4o 75
2.3.6. Otras 80
2.4. Protocolos de acceso a bases de datos 81
2.5. Acceso a datos mediante ODBC 82
2.6. Acceso a datos mediante JDBC 85
2.6.1. Dos modelos de acceso a bases de datos 86
2.6.2. Tipos de drivers 86
2.6.3. Cómo funciona JDBC 87
2.6.4. Acceso a datos mediante el puente JDBC-ODBC 93
2.7. Establecimiento de conexiones 95
2.8. Ejecución de sentencias de descripción de datos 97
2.8.1. ResultSetMetaData 103
2.9. Ejecución de sentencias de manipulación de datos 105
2.9.1. Ejecución de Scripts 109
2.9.2. Sentencias preparadas 112
2.10. Ejecución de procedimientos 114
2.11. Informes con JasperReports 119
2.11.1. El fichero .JRXML, la plantilla 122
2.12. Gestión de errores 131
COMPRUEBA TU APRENDIZAJE 133
ACTIVIDADES DE AMPLIACIÓN 141

CAPÍTULO 3. HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL (ORM) **143**

3.1. Introducción 144
3.2. Concepto de mapeo objeto-relacional 144
3.3. Herramientas ORM. Características 144
3.4. Arquitectura Hibernate 146
3.5. Instalación y configuración de Hibernate 147
3.5.1. Instalación del plugin 148
3.5.2. Configuración del driver MySQL 150
3.5.3. Configuración de Hibernate 151
3.5.4. Generar las clases de la base de datos 157
3.5.5. Primera consulta en HQL 159
3.5.6. Empezando a programar con Hibernate en Eclipse 161
3.6. Estructura de los ficheros de mapeo 168
3.7. Clases persistentes 171
3.8. Sesiones y objetos Hibernate 173

3.8.1. Transacciones 174
3.8.2. Estados de un objeto Hibernate 174
3.8.3. Carga de objetos 175
3.8.4. Almacenamiento, modificación y borrado de objetos 177
3.9. Consultas 180
3.9.1. Parámetros en las consultas 183
3.9.2. Consultas sobre clases no asociadas 185
3.9.3. Funciones de grupo en las consultas 185
3.9.4. Objetos devueltos por las consultas 186
3.10. Insert, update y delete 187
3.11. Resumen del lenguaje HQL 189
3.11.1. Asociaciones y uniones (joins) 192
COMPRUEBA TU APRENDIZAJE 194
ACTIVIDADES DE AMPLIACIÓN 198

CAPÍTULO 4. BASES DE DATOS OBJETO-RELACIONALES Y ORIENTADAS A OBJETOS 201

4.1. Introducción 202
4.2. Bases de datos objeto-relacionales 202
4.2.1. Características 202
4.2.2. Tipos de objetos 203
4.2.2.1. Métodos 204
4.2.3. Tablas de objetos 209
4.2.4. Tipos colección 211
4.2.4.1. VARRAYS 211
4.2.4.2. Tablas anidadas 214
4.2.5. Referencias 221
4.2.6. Herencia de tipos 222
4.2.7. Ejemplo de modelo relacional y objeto relacional 224
4.3. Bases de datos orientada a objetos 230
4.3.1. Características de las bases de datos OO 230
4.3.2. El estándar ODMG 232
4.3.3. El lenguaje de consultas OQL 234
4.3.3.1. Operadores de comparación 236
4.3.3.2. Cuantificadores y operadores sobre colecciones 237
4.4. Ejemplo de BDOO 238
4.4.1. Consultas sencillas 241
4.4.2. Consultas más complejas 245
4.4.2.1 Crear una BD Neodatis a partir de un modelo relacional 249
4.4.3. Modelo cliente/servidor de la base de datos 255
COMPRUEBA TU APRENDIZAJE 259
ACTIVIDADES DE AMPLIACIÓN 264

CAPÍTULO 5. BASES DE DATOS NoSQL **267**

5.1. Introducción 268
5.2. Ventajas de los sistemas NoSQL 269
5.3. Diferencias con las bases de datos SQL 269
5.4. Tipos de bases de datos NoSQL 270
5.5. Bases de datos nativas XML 271
5.5.1. Base de datos eXist 273
5.5.1.1. Instalación de eXist 273
5.5.1.2. Primeros pasos con eXist 275
5.5.1.3. El cliente de administración de eXist 277
5.5.2. Lenguajes de consultas XPath y XQuery 279
5.5.2.1. Expresiones XPath 279
5.5.2.2. Nodos atributos XPath 286
5.5.2.3. Axis XPath 289
5.5.2.4. Consultas XQuery 291
5.5.2.5. Operadores y funciones más comunes en XQuery 296
5.5.2.6. Consultas complejas con XQuery 298
5.5.2.7. Sentencias de actualización de eXist 302
5.5.3. Acceso a eXist desde Java 304
5.5.3.1. La API XML:DB para bases de datos XML 304
5.5.3.2. La API XQJ (XQuery) 311
5.5.4. Tratamiento de excepciones 316
5.6. Base de datos MongoDB 320
5.6.1. Estructuras JSON 321
5.6.2. Instalación MongoDB 324
5.6.3. Operaciones básicas en MongoDB 326
5.6.4. Consultar registros 327
5.6.5. Actualizar registros en MongoDB 329
5.6.6. Borrar registros 332
5.6.7. Funciones de agregado 333
5.6.8. La agregación pipeline 334
5.6.9. Relaciones entre documentos en MongoDB 341
5.6.10. Herramienta Robomongo 342
5.6.11. Mongodb en Java 344
COMPRUEBA TU APRENDIZAJE 352
ACTIVIDADES DE AMPLIACIÓN 355

CAPÍTULO 6. PROGRAMACIÓN DE COMPONENTES DE ACCESO A DATOS **357**

6.1. Introducción 358
6.2. Concepto de componente 358
6.2.1. Características 358
6.2.2. Ventajas e inconvenientes 359
6.3. JavaBeans 359
6.3.1. Propiedades y atributos 361
6.3.2. Eventos 365

6.3.3. Persistencia del componente 366
6.4. Herramientas para el desarrollo del componente 366
6.4.1. Crear JavaBeans con NetBeans 367
6.5. Empaquetado de componentes 374
6.6. Usando JavaBeans para acceder a bases de datos 375
6.7. Patrón Data Access Object (DAO) 382
6.7.1. Ejemplo de aplicación 383
6.8. Patrón Factory 387
6.8.1. Ejemplo de aplicación 388
6.9. Patrón modelo vista controlador (MVC) 400
6.9.1. Servlets 402
6.9.2. Páginas JSP 402
6.9.3. Ejemplo de aplicación 404
COMPRUEBA TU APRENDIZAJE 419
ACTIVIDADES DE AMPLIACIÓN 422

CAPÍTULO 1

MANEJO DE FICHEROS

OBJETIVOS

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.

Serializar objetos Java a representaciones XML.

CONTENIDOS

Clases asociadas a las operaciones de gestión de ficheros. Flujos o stream. Tipos.

Formas de acceso a un fichero.

Clases para gestión de flujos de datos desde/hacia ficheros.

Operaciones básicas sobre ficheros de acceso secuencial.

Operaciones básicas sobre ficheros de acceso aleatorio.

Ficheros XML. Librerías para conversión de documentos XML a otros formatos.

Excepciones: detección y tratamiento.

Introducción a JAXB.

RESUMEN

En este capítulo aprenderemos a a leer y escribir datos en ficheros secuenciales y directos en Java. Utilizaremos diferentes clases Java para el acceso a ficheros. Utilizaremos distintas librerías para conversión de ficheros XML a otros formatos. Aprenderemos a utilizar y gestionar excepciones. Utilizaremos la tecnología JAXB para serializar objetos Java a representaciones XML.

1.1. INTRODUCCIÓN

Un **fichero** o **archivo** es un conjunto de bits almacenados 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 fichero.

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

El paquete **java.io** contiene las clases para manejar la entrada/salida en Java, por tanto, necesitaremos importar dicho paquete cuando trabajemos con 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(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").*

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:

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

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

File direc = new File(directorio);
```

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

Algunos de los métodos más importantes de la clase **File** son los siguientes:

Método	Función
`String[] list()`	Devuelve un array de String con los nombres de ficheros y directorios asociados al objeto **File**
`File[] listFiles()`	Devuelve un array de objetos **File** conteniendo los ficheros que estén dentro del directorio representado por el objeto **File**
`String getName()`	Devuelve el nombre del fichero o directorio
`String getPath()`	Devuelve el camino relativo
`String getAbsolutePath()`	Devuelve el camino absoluto del fichero/directorio
`boolean exists()`	Devuelve *true* si el fichero/directorio existe
`boolean canWrite()`	Devuelve *true* si el fichero se puede escribir
`boolean canRead()`	Devuelve *true* si el fichero se puede leer
`boolean isFile()`	Devuelve *true* si el objeto **File** corresponde a un fichero normal
`boolean isDirectory()`	Devuelve *true* si el objeto **File** corresponde a un directorio
`long length()`	Devuelve el tamaño del fichero en bytes
`boolean mkdir()`	Crea un directorio con el nombre indicado en la creación del objeto **File.** Solo se creará si no existe
`boolean renameTo(File nuevonombre);`	Renombra el fichero representado por el objeto **File** asignándole *nuevonombre*
`boolean delete()`	Borra el fichero o directorio asociado al objeto **File**
`boolean createNewFile()`	Crea un nuevo fichero, vacío, asociado a **File** si y solo si no existe un fichero con dicho nombre
`String getParent()`	Devuelve el nombre del directorio padre, o *null* si no existe

El siguiente ejemplo muestra la lista de ficheros en el directorio actual. Se utiliza el método *list()* que devuelve un array de String 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 la variable *dir* con el valor ".". Se define un segundo objeto **File** utilizando el tercer constructor para saber si el fichero obtenido es un fichero o un directorio:

```
import java.io.*;
public class VerDir {
  public static void main(String[] args) {
    String dir = "."; //directorio actual
    File f = new File(dir);
    String[] archivos = f.list();
    System.out.printf("Ficheros en el directorio actual: %d %n",
                      archivos.length);
    for (int i = 0; i < archivos.length; i++) {
      File f2 = new File(f, archivos[i]);
      System.out.printf("Nombre: %s, es fichero?: %b, es directorio?:
                %b %n", archivos[i], f2.isFile(),f2.isDirectory());
    }
  }
}
```

Un ejemplo de ejecución de este programa mostraría la siguiente salida:

```
Ficheros en el directorio actual: 3
Nombre: VerDir.class, es fichero?: true, es directorio?: false
Nombre: VerDir.java, es fichero?: true, es directorio?: false
Nombre: VerInf.java, es fichero?: true, es directorio?: false
```

La siguiente declaración aplicada al ejemplo anterior 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 1.1

Realiza un programa Java que utilice el método **listFiles()** para mostrar la lista de ficheros en un directorio cualquiera, o en el directorio actual.

Realiza un programa Java que muestre los ficheros de un directorio. El nombre del directorio se pasará al programa desde los argumentos de *main()*. Si el directorio no existe se debe mostrar un mensaje indicándolo.

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("D:\\ADAT\\UNI1\\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 leer       : "+f.canRead());
    System.out.println("Se puede escribir   : "+f.canWrite());
    System.out.println("Tamaño              : "+f.length());
    System.out.println("Es un directorio    : "+f.isDirectory());
    System.out.println("Es un fichero       : "+f.isFile());
    System.out.println("Nombre del directorio padre: "+f.getParent());
  }
 }
}
```

Visualiza la siguiente información del fichero:

```
INFORMACIÓN SOBRE EL FICHERO:
Nombre del fichero  : VerInf.java
Ruta                : D:\ADAT\UNI1\VerInf.java
Ruta absoluta       : D:\ADAT\UNI1\VerInf.java
Se puede leer       : true
Se puede escribir   : true
```

```
Tamaño                  : 824
Es un directorio        : false
Es un fichero           : true
Nombre del directorio padre: D:\ADAT\UNI1
```

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
   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 eliminar estos ficheros. Para borrar el objeto *f2* 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`	Flujo de entrada hacia fichero, lo usaremos 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 dónde 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`	Flujo de salida hacia fichero, lo usaremos 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. Jerarquía 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** y **OutputStreamWriter** que convierte un **OutputStream** en un **Writer** (convierte streams de caracteres a streams de bytes).

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. Jerarquía de clases para 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 programa.
- **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 extraídos 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 renombrarlo 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 cómo 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 (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*, etc.)

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:

Método	Función
`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");
    //crear el flujo de entrada hacia el fichero
    FileReader fic = new FileReader(fichero);
    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. También se puede declarar el fichero de la siguiente manera:

```
FileReader fic =
          new FileReader("C:\\EJERCICIOS\\UNI1\\LeerFichTexto.java");
```

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 1.2

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 el programa desde la línea de comandos.

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

Método	Función
`void write(int c)`	Escribe un carácter.
`void write(char[] buf)`	Escribe un array de caracteres.
`void write(char[] buf, int desplazamiento , int n)`	Escribe n caracteres de datos en la matriz *buf* comenzando por *buf[desplazamiento]*.
`void write(String str)`	Escribe una cadena de caracteres.
`void append(char c)`	Añade un carácter a un fichero.

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* que se convierte en array de caracteres:

```
import java.io.*;
public class EscribirFichTexto {
  public static void main(String[] args) throws IOException {
   File fichero = new
      File("C:\\EJERCICIOS\\UNI1\\FichTexto.txt");//declarar fichero
   //crear flujo de salida
   FileWriter fic = new FileWriter(fichero);

   String cadena ="Esto es una prueba con FileWriter";
   //convierte la cadena en array de caracteres para extraerlos 1 a 1
   char[] cad = cadena.toCharArray();
   for(int i=0; i<cad.length; i++)
         fic.write(cad[i]);  //se va escribiendo un carácter

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

En vez de escribir los caracteres uno a uno, también podemos escribir todo el array usando ***fic.write(cad)***. El siguiente ejemplo escribe cadenas de caracteres que se obtienen de un array de String, las cadenas se irán insertando en el fichero 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)
        System.out.println(linea);

      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 **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 escriben en un fichero, el segundo método, además, produce un salto 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 así:

```
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:

Método	Función
`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:

Método	Función
`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 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");//declara fichero
   //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 definen diversos métodos *readXXX* y *writeXXX* que son variaciones de los métodos ***read()*** y ***write()*** de la clase base para leer y escribir datos de tipo primitivo. 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);`

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

```
File fichero = new File("FichData.dat");
FileInputStream filein = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(filein);
```

O bien

```
File fichero = new File("FichData.dat");
DataInputStream dataIS = new
                DataInputStream(new FileInputStream(fichero));
```

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

```
File fichero = new File("FichData.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileout);
```

O bien

```
File fichero = new File("FichData.dat");
DataOutputStream dataOS = new
                 DataOutputStream(new FileOutputStream(fichero));
```

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 (mediante el método ***writeUTF(String)***) y la edad (mediante el método ***writeInt(int)***):

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

   File fichero = new File("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]); //escribe nombre
      dataOS.writeInt(edades[i]);  //escribe 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 escribieron, 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("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 al ejecutar el programa:

```
Nombre: Ana, edad: 14
Nombre: Luis Miguel, edad: 15
Nombre: Alicia, edad: 13
Nombre: Pedro, edad: 15
Nombre: Manuel, edad: 16
Nombre: Andrés, edad: 12
Nombre: Julio, edad: 16
Nombre: Antonio, edad: 14
Nombre: María Jesús, edad: 13
```

1.6.3. Objetos en ficheros binarios

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:

- **Object readObject():** se utiliza para leer un objeto del **ObjectInputStream.** Puede lanzar las excepciones ***IOException*** y ***ClassNotFoundException***.
- **void writeObject(Object obj):** se utiliza para escribir el objeto especificado en el **ObjectOutputStream.** Puede lanzar la excepción ***IOException***.

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 nombre){this.nombre = nombre;}
      public void setEdad(int edad){this.edad = edad;}

      public String getNombre(){return this.nombre;}//devuelve nombre
      public int getEdad(){return this.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("FichPersona.dat");
FileOutputStream fileout = new FileOutputStream(fichero);
ObjectOutputStream dataOS = new ObjectOutputStream(fileout);
```

O bien:

```
File fichero = new File("FichPersona.dat");
ObjectOutputStream dataOS = new
                    ObjectOutputStream(new FileOutputStream(fichero));
```

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("FichPersona.dat");
```

```
    //crea el flujo de salida
    FileOutputStream fileout = new FileOutputStream(fichero);
    //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]);
        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("FichPersona.dat");
FileInputStream filein = new FileInputStream(fichero);
ObjectInputStream dataIS = new ObjectInputStream(filein);
```

O bien:

```
File fichero = new File("FichPersona.dat");
ObjectInputStream dataIS = new
                  ObjectInputStream(new FileInputStream(fichero));
```

El método ***readObject()*** lee los objetos del flujo de entrada, puede lanzar la excepción ***ClassNotFoundException*** e ***IOException***, por lo que será necesario controlarlas. El proceso de lectura se hace en un bucle *while(true)*, este se encierra en un bloque **try-catch** ya que la lectura finalizará cuando se llegue al final de fichero, entonces, se lanzará la excepción ***EOFException.*** 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("FichPersona.dat");
   //crea el flujo de entrada
   FileInputStream filein = new FileInputStream(fichero);
   //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.printf("Nombre: %s, edad: %d %n",
```

```
                    persona.getNombre(),persona.getEdad());
      }
    }catch (EOFException eo) {
       System.out.println("FIN DE LECTURA.");
    }

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

Problema con los ficheros de objetos:

Existe un problema con los ficheros de objetos. Al crear un fichero de objetos se crea una cabecera inicial con información, y a continuación se añaden los objetos. Si el fichero se utiliza de nuevo para añadir más registros, se crea una nueva cabecera y se añaden los objetos a partir de esa cabecera. El problema surge al leer el fichero cuando en la lectura se encuentra con la segunda cabecera, y aparece la excepción **StreamCorruptedException** y no podremos leer más objetos.

La cabecera se crea cada vez que se pone ***new ObjectOutputStream(fichero).*** Para que no se añadan estas cabeceras lo que se hace es ***reedefinir la clase ObjectOutputStream creando una nueva clase que la herede (extends).*** Y dentro de esa clase se redefine el método ***writeStreamHeader()*** que es el que escribe las cabeceras, y hacemos que ese método no haga nada. De manera que si el fichero ya se ha creado se llamará a ese método de la clase redefinida.

La clase redefinida quedará así:

```
public class MiObjectOutputStream extends ObjectOutputStream
{
    public MiObjectOutputStream(OutputStream out) throws IOException
    {    super(out);     }
    protected MiObjectOutputStream()
            throws IOException, SecurityException
    {   super();    }
    // Redefinición del método de escribir la cabecera
    // para que no haga nada.
    protected void writeStreamHeader() throws IOException
    {  }
}
```

Y dentro de nuestro programa a la hora de abrir el fichero para añadir nuevos objetos se pregunta si ya existe, si existe, se crea el objeto con la clase redefinida, y si no existe, el fichero se crea con la clase ***ObjectOutputStream***:

```
File fichero = new File(nombrefichero);
ObjectOutputStream dataOS;
if (!fichero.exists())
{  //Si el fichero no existe crea un ObjectOutputStream, la primera vez
    FileOutputStream fileout;
    fileout = new FileOutputStream(fichero);
    dataOS = new ObjectOutputStream(fileout);
}
else
{   // Si ya existe el fichero creará un ObjectOutputStream
```

```
    // con el método writeStreamHeader redefinido (sin hacer nada)
    dataOS = new MiObjectOutputStream
                (new FileOutputStream(fichero,true));
} //fin if
```

1.6.4. 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 jerarquía **InputStream/OutputStream**, ya que su comportamiento es totalmente distinto puesto que se puede avanzar y retroceder dentro de un fichero.

Disponemos de dos constructores para crear el fichero de acceso aleatorio, estos pueden lanzar la excepción ***FileNotFoundException***:

- ***RandomAccessFile(String nombrefichero, String modoAcceso):*** escribiendo el nombre del fichero incluido el path.
- ***RandomAccessFile(File objetoFile, String modoAcceso):*** con un objeto **File** asociado a un fichero.

El argumento *modoAcceso* puede tener dos valores:

Modo de acceso	Significado
`r`	Abre el fichero en modo de solo lectura. El fichero debe existir. Una operación de escritura en este fichero lanzará la excepción ***IOException***
`rw`	Abre el fichero en modo lectura y escritura. Si el fichero no existe se crea

Una vez abierto el fichero pueden usarse los métodos readXXX y writeXXX de las clases **DataInputStream** y **DataOutputStream** (vistos anteriormente). La clase **RandomAccessFile** maneja un puntero que indica la posición actual en el fichero. Cuando el fichero se crea el puntero al fichero 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:

Método	Función
`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: apellido, departamento y salario, se obtienen de varios arrays que se llenan en el programa, los datos se van introduciendo de forma secuencial por lo que no va a ser necesario usar el método ***seek()***. Por cada empleado también se insertará un identificador (mayor que 0) 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. Como Java utiliza caracteres UNICODE, cada carácter de una cadena de caracteres ocupa 16 bits (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("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;//nÚmero 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("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

      //recorro uno a uno los caracteres del apellido
      for (int i = 0; i < apellido.length; i++) {
         aux = file.readChar();
         apellido[i] = aux;    //los voy guardando en el array
      }

      //convierto a String el array
      String apellidos = new String(apellido);
      dep = file.readInt();          //obtengo dep
      salario = file.readDouble(); //obtengo salario

      if(id > 0)
        System.out.printf("ID: %s, Apellido: %s, Departamento: %d,
                           Salario: %.2f %n",
                           id,   apellidos.trim(), dep, salario);

      //me posiciono para el sig empleado, cada empleado ocupa 36 bytes
      posicion= posicion + 36;

      //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 (en este ejemplo son 36 bytes):

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

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;    //buffer 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
```

ACTIVIDAD 1.3

Consulta. Crea un programa Java que consulte los datos de un empleado del fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir un identificador de empleado. Si el empleado existe se visualizarán sus datos, si no existe se visualizará un mensaje indicándolo.

Inserción. Crea un programa Java que inserte datos en el fichero aleatorio. El programa se ejecutará desde la línea de comandos y debe recibir 4 parámetros: identificador de empleado, apellido, departamento y salario. Antes de insertar se comprobará si el identificador existe, en ese caso se debe visualizar un mensaje indicándolo; si no existe se deberá insertar.

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 lo siguiente:

```
int registro = 4;                    //id a modificar
long posicion = (registro -1 ) * 36; //calculo la posición
posición = posición + 4 + 20;        //sumo el tamaño de ID + apellido
file.seek(posicion);                 //nos posicionamos
file.writeInt(40);                   //modifico departamento
file.writeDouble(4000.87);           //modifico salario
```

ACTIVIDAD 1.4

Modificación. Crea un programa Java que reciba desde la línea de comandos un identificador de empleado y un importe. Se debe realizar la modificación del salario. La modificación consistirá en sumar al salario del empleado el importe introducido. El programa debe visualizar el apellido, el salario antiguo y el nuevo. Si el identificador no existe se visualizará mensaje indicándolo.

Borrado. Crea un programa Java que al ejecutarlo desde la línea de comandos reciba un identificador de empleado y lo borre. Se hará un borrado lógico marcando el registro con la siguiente información: el identificador será igual a -1, el apellido será igual al identificador que se borra, y el departamento y salario serán 0.

A continuación haz otro programa Java (o crea un método dentro del anterior programa) que muestre los identificadores de los empleados borrados.

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 y 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, etc.) 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, la interfaz **DOMImplementation** permite crear objetos **Document** con nodo raíz:

```
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:

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

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

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

Como se puede ver el método 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 elemento
```

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>
```

El método es el 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")); //fichero 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 args[]) throws IOException{
   File fichero = new File("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");

     for(;;) {
       file.seek(posicion); //nos posicionamos
       id=file.readInt();   // obtengo id de empleado
```

```
      for (int i = 0; i < apellido.length; i++) {
        aux = file.readChar();
        apellido[i] = aux;
      }
      String apellidos = new String(apellido);
      dep = file.readInt();
      salario = file.readDouble();

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

        //añadir ID
        CrearElemento("id",Integer.toString(id), raiz, document);
        //Apellido
        CrearElemento("apellido",apellidos.trim(), raiz, document);
        //añadir DEP
        CrearElemento("dep",Integer.toString(dep), raiz, document);
        //añadir salario
        CrearElemento("salario",Double.toString(salario), raiz,
                                                        document);
      }
      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);
   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[] args) {

  DocumentBuilderFactory factory =
                        DocumentBuilderFactory.newInstance();

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

    System.out.printf("Elemento raiz: %s %n",
                      document.getDocumentElement().getNodeName());
    //crea una lista con todos los nodos empleado
    NodeList empleados = document.getElementsByTagName("empleado");
    System.out.printf("Nodos empleado a recorrer: %d %n",
                      empleados.getLength());

    //recorrer la lista
    for (int i = 0; i < empleados.getLength(); i ++) {
      Node emple = empleados.item(i); //obtener un nodo empleado
      if (emple.getNodeType() == Node.ELEMENT_NODE) {//tipo de nodo
        //obtener los elementos del nodo
        Element elemento = (Element) emple;
        System.out.printf("ID = %s %n",
           elemento.getElementsByTagName("id").
                                    item(0).getTextContent());
        System.out.printf(" * Apellido = %s %n",
           elemento.getElementsByTagName("apellido").
                                    item(0).getTextContent());
        System.out.printf(" * Departamento = %s %n",
           elemento.getElementsByTagName("dep").
                                    item(0).getTextContent());
        System.out.printf(" * Salario = %s %n",
           elemento.getElementsByTagName("salario").
                                    item(0).getTextContent());
      }
    }
  } catch (Exception e)
    {e.printStackTrace();}

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

¡ ¡ INTERESANTE ! !

API DOM: *http://docs.oracle.com/javase/8/docs/api/org/w3c/dom/package-tree.html.*

ACTIVIDAD 1.5

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"?>`
`<listadealumnos>`
`  <alumno>`
`  <nombre>`
`  Juan`
`  </nombre>`
`  <edad>`
`  19`
`  </edad>`
`  </alumno>`
`  <alumno>`
`  <nombre>`
`  Maria`
`  </nombre>`
`  <edad>`
`  20`
`  </edad>`
`  </alumno>`
`</listadealumnos>`
	`startDocument()`
	`startElement()`
	`  startElement()`
	`  startElement()`
	`  characters()`
	`  endElement()`
	`  startElement()`
	`  characters()`
	`  endElement()`
	`  endElement()`
	`  startElement()`
	`  startElement()`
	`  characters()`
	`  endElement()`
	`  startElement()`
	`  characters()`
	`  endElement()`
	`  endElement()`
	`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.printf("\tPrincipio Elemento: %s %n",nombre);
          }
          public void endElement(String uri, String nombre,
                         String nombreC) {
              System.out.printf("\tFin Elemento: %s %n", nombre);
          }
          public void characters(char[] ch, int inicio, int longitud)
                                          throws SAXException {
               String car = new String(ch, inicio, longitud);
               //quitar saltos de línea
               car = car.replaceAll("[\t\n]","");
               System.out.printf ("\tCaracteres: %s %n", 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
```

¡ ¡ INTERESANTE ! !

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

ACTIVIDAD 1.6

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://x-stream.github.io/download.html*. Para el ejemplo se ha descargado el fichero **xstream-distribution-1.4.8-bin.zip** que hemos de descomprimir y buscar el JAR **xstream-1.4.8.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 localiza en la carpeta *lib\xstream*. Una vez que tenemos los dos ficheros los añadimos a nuestro proyecto Eclipse o NetBeans o los definimos en el CLASSPATH, por ejemplo, supongamos que tenemos los JAR en la carpeta *D:\uni1\xstream*, el CLASSPATH nos quedaría:

```
SET CLASSPATH =
   .;D:\uni1\xstream\kxml2-2.3.0.jar;D:\uni1\xstream\xstream-1.4.8.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);
    ObjectInputStream dataIS = new ObjectInputStream(filein);

    System.out.println("Comienza el proceso...");

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

    try {
        while (true) { //lectura del fichero
           Persona persona= (Persona) dataIS.readObject();
           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(String alias, Class clase)***. 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);
```

El método ***aliasField(String alias, Class clase, String nombrecampo)***, permite crear un alias para un nombre de campo. Por ejemplo, si queremos cambiar el nombre de los campos *nombre* y *edad* (de la clase *Persona*) crearíamos los siguientes alias:

```
xstream.aliasField("Nombre alumno", Persona.class, "nombre");
xstream.aliasField("Edad alumno", Persona.class, "edad");
```

Entonces en el fichero XML se crearían con las etiquetas *<Nombre alumno>* en lugar de *<nombre>* y *<Edad alumno>* en lugar de *<edad>*.

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

```
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(Object objeto, OutputStream out)***:

```
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();
    while( iterador.hasNext() ) {
       Persona p = (Persona) iterador.next();
       System.out.printf("Nombre: %s, edad: %d %n",
                         p.getNombre(), 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 a partir del fichero, utilizamos el método ***fromXML (InputStream input)*** que devuelve un tipo **Object:**

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

¡¡ INTERESANTE ! !
API XStream: *http://x-stream.github.io/javadoc/index.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:

FICHERO alumnos.xml:

```
<?xml version="1.0"?>
<listadealumnos>
   <alumno>
        <nombre>Juan</nombre>
        <edad>19</edad>
   </alumno>
   <alumno>
        <nombre>Maria</nombre>
        <edad>20</edad>
   </alumno>
</listadealumnos>
```

FICHERO alumnosPlantilla.xsl:

```
<?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(Source source)*** 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(Source fuenteXml, Result resultado)***, 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[] args) throws IOException{
  String hojaEstilo = "alumnosPlantilla.xsl";
  String datosAlumnos = "alumnos.xml";
  File pagHTML = new File("mipagina.html");

  //crear fichero HTML
  FileOutputStream os = new FileOutputStream(pagHTML);
  Source estilos = new StreamSource(hojaEstilo); //fuente XSL
  Source datos = new StreamSource(datosAlumnos); //fuente XML

  //resultado de la transformación
  Result result = new StreamResult(os);

  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.

¡ ¡ INTERESANTE ! !
API de Java: *http://docs.oracle.com/javase/8/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, es capturada 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.printf("Resultado: %d", cociente);
   }
}//

D:\uni1>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); //sí 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:

Método	Función
`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)`	Visualiza 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 (marcada en negrita) 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 (marcadas en negrita).

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

```
public class EscribirPersonas2 {
  public static void main(String[] args)  {
      File fichero = new File("FichPersona.dat");
      FileInputStream filein;
      try {
```

```
        filein = new FileInputStream(fichero);
        ObjectInputStream dataIS = new ObjectInputStream(filein);
        System.out.println("Comienza el proceso ...");

        ListaPersonas listaper = new ListaPersonas();

        try {
           while (true) { //lectura del fichero
               Persona persona= (Persona) dataIS.readObject();
               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/8/docs/api/* para ver las excepciones que se pueden producir en los diferentes paquetes.

1.9. INTRODUCCIÓN A JAXB

JAXB es una tecnología Java que permite mapear clases Java a representaciones XML, y viceversa, es decir, serializar objetos Java a representaciones XML. JAXB provee dos funciones fundamentales:

- La capacidad de presentar un objeto Java en XML (serializar), al proceso lo llamaremos *marshall* o *marshalling*. Java Object a XML.
- Lo contrario, es decir ,presentar un XML en un objeto Java (deserializar), al proceso lo llamaremos *unmarshall* o *unmarshalling*. XML a Java Object

También el compilador que proporciona JAXB nos va a permitir generar clases Java a partir de esquemas XML, que podrán ser llamadas desde las aplicaciones a través de métodos *sets* y *gets* para obtener o establecer los datos de un documento XML.

1.9.1. Mapear clases Java a representaciones XML

Para crear objetos Java en XML, vamos a utilizar ***JavaBeans***, que serán las clases que se van a mapear. Son clases primitivas Java (**POJOs** - *Plain Old Java Objects*) con las propiedades, *getter* y *setter*, el constructor sin parámetros y el constructor con las propiedades. En estas clases que se van a mapear se añadirán las **Anotaciones**, que son las indicaciones que ayudan a convertir el JavaBean en XML.

Las principales **anotaciones** son:

- @**XmlRootElement**(namespace = "namespace"): Define la raíz del XML. Si una clase va a ser la raíz del documento se añadirá esta anotación, el *namespace* es opcional.

```
@XmlRootElement
   public class ClaseRaiz {
...
}
```

- @**XmlType**(propOrder = { "field2", "field1",.. }): Permite definir en qué orden se van a escribir los elementos (o las etiquetas) dentro del XML. Si es una clase **que no va a ser raíz** añadiremos **@XmlType.**
- @**XmlElement**(name = "nombre"): Define el elemento de XML que se va usar.

A cualquiera de ellos podemos ponerle entre paréntesis el nombre de etiqueta que queramos que salga en el documento XML para la clase, añadiendo el atributo ***name***. Sería algo como esto

```
@XmlRootElement(name = "Un_Nombre_para_la_raiz")
@XmlType(name = "Otro_Nombre")
```

Para cada atributo de la clase que queramos que salga en el XML, el método get correspondiente a ese atributo debe llevar una anotación ***@XmlElement***, a la que a su vez podemos ponerle un nombre (estas anotaciones no son obligatorias, solo si se desean nombres diferentes del atributo):

```
@XmlRootElement(name = "La_ClaseRaiz")
public class UnaClase {
   private String unAtributo;
   @XmlElement(name = "El_Atributo")
   String getUnAtributo() {
      return this.unAtributo;
   }
}
```

Si el atributo es una colección (array, list, etc...) debe llevar dos anotaciones, ***@XmlElementWrapper*** y ***@XmlElement***, esta última, con un nombre si se desea. Por ejemplo:

```
@XmlRootElement(name = "La_ClaseRaiz")
public class UnaClase {
   private String [] unArray;

   @XmlElementWrapper
   @XmlElement(name = "Elemento_Array")
   String [] getUnArray() {
      return this.unArray;
   }
}
```

Si el atributo es otra clase (otro JavaBean), le ponemos igualmente ***@XmlElement*** al método *get*, pero la clase que hace de atributo debería llevar a la vez sus anotaciones correspondientes.

Ejemplo1: se desea generar el siguiente documento XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<libreria>
    <ListaLibro>
        <Libro>
            <autor>Alicia Ramos</autor>
            <nombre>Entornos de Desarrollo</nombre>
            <editorial>Garceta</editorial>
            <isbn>978-84-1545-297-3</isbn>
        </Libro>
        <Libro>
            <autor>María Jesús Ramos</autor>
            <nombre>Acceso a Datos</nombre>
            <editorial>Garceta</editorial>
            <isbn>978-84-1545-228-7</isbn>
        </Libro>
    </ListaLibro>
    <lugar>Talavera, como no</lugar>
    <nombre>Prueba de libreria JAXB</nombre>
</libreria>
```

Se trata de representar los libros de una librería. Crearemos las siguientes clases:

- La clase *Libreria*, con la lista de libros, el lugar y el nombre de la librería.
- La clase *Libro*, con los datos del autor, el nombre, la editorial y el ISBN.

En la clase *Libro*, vamos a indicar la anotación ***@XmlType*** pues es una clase que no es raíz, y además indicamos el orden de las etiquetas con ***propOrder***, es decir, cómo se desea que salgan en el documento XML. La clase tendrá la siguiente descripción:

```
package clasesjaxb;

import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = {"autor", "nombre", "editorial", "isbn"})
public class Libro {
    private String nombre;
    private String autor;
    private String editorial;
    private String isbn;
    public Libro(String nombre, String autor, String editorial,
             String isbn) {
            super();
            this.nombre = nombre;
            this.autor = autor;
            this.editorial = editorial;
            this.isbn = isbn;
      }
    public Libro() {}
    public String getNombre() { return nombre; }
    public String getAutor() { return autor;  }
    public String getEditorial() {return editorial; }
```

```
    public String getIsbn() { return isbn;}
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setAutor(String autor) { this.autor = autor; }
    public void setEditorial(String editorial)
        { this.editorial = editorial;  }
    public void setIsbn(String isbn) {  this.isbn = isbn;  }
}
```

En la clase *Libreria*, vamos a indicar la anotación ***@XmlRootElement*** pues es una clase raíz. También tenemos que indicar que hay un atributo, qué es una colección, con lo que hay que añadir con las anotaciones ***@XmlElementWrapper*** y ***@XmlElement***, en el método *get*. En estas anotaciones indicamos como se van a llamar las etiquetas dentro del documento generado. La clase tendrá la siguiente descripción:

```
package clasesjaxb;
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement()
public class Libreria {
    private ArrayList<Libro> listaLibro;
    private String nombre;
    private String lugar;

    public Libreria(ArrayList<Libro> listaLibro, String nombre,
        String lugar) {
            super();
            this.listaLibro = listaLibro;
            this.nombre = nombre;
            this.lugar = lugar; }
    public Libreria(){}
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setLugar(String lugar) { this.lugar = lugar;    }
    public String getNombre() {return nombre;    }
    public String getLugar() { return lugar;    }

    //Wrapper, envoltura alrededor la representación XML
    @XmlElementWrapper(name = "ListaLibro") //
    @XmlElement(name = "Libro")
    public ArrayList<Libro> getListaLibro() {
        return listaLibro;    }

    public void setListaLibro(ArrayList<Libro> listaLibro) {
        this.listaLibro = listaLibro;    }
}
```

Una vez que tenemos las clases ya definidas, lo siguiente es ver el **código Java para mapear los objetos** que definamos de esas clases.

Utilizando la anotación ***@XmlRootElement***. El código Java para conseguir el fichero XML es el siguiente:

- Instanciamos el contexto, indicando la clase que será el ***RootElement***, en nuestro ejemplo es la clase *Libreria*:

```
JAXBContext jaxbContext = JAXBContext.newInstance(Libreria.class);
```

- Creamos un ***Marshaller***, que es la clase capaz de convertir nuestro JavaBean, en una cadena XML:

```
Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
```

- Indicamos que vamos a querer el XML con un formato amigable (saltos de línea, sangrado, etc)

```
jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

- Hacemos la conversión llamando al método ***marshal***, pasando una instancia del JavaBean que queramos convertir a XML y un **OutpuStream** donde queramos que salga el XML, por ejemplo, la salida estándar, o también podría ser un fichero o cualquier otro stream:

```
jaxbMarshaller.marshal(unaInstanciaDeUnaClase, System.out);
//Si ponemos un fichero
jaxbMarshaller.marshal(unaInstanciaDeUnaClase,
                          new File("./mifichero.xml"));
```

Ahora vamos a crear objetos de las clases y vamos a ver cómo generar el XML:

```
package clasesjaxb;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

public class Ejemplo1_JAXB {
    private static final String MIARCHIVO_XML = "./libreria.xml";
    public static void main(String[] args)
       throws JAXBException, IOException {
        //Se crea la lista de libros
        ArrayList<Libro> libroLista = new ArrayList<Libro>();
        // Creamos dos libros y los añadimos
        Libro libro1 = new Libro("Entornos de Desarrollo",
                 "Alicia Ramos","Garceta","978-84-1545-297-3" );
        libroLista.add(libro1);
        Libro libro2 = new Libro("Acceso a Datos","Maria Jesús Ramos",
                 "Garceta","978-84-1545-228-7" );
        libroLista.add(libro2);

        // Se crea La libreria y se le asigna la lista de libros
        Libreria milibreria = new Libreria();
        milibreria.setNombre("Prueba de libreria JAXB");
        milibreria.setLugar("Talavera, como no");
```

```
        milibreria.setListaLibro(libroLista);

        // Creamos el contexto indicando la clase raíz
        JAXBContext context = JAXBContext.newInstance(Libreria.class);
        //Creamos el Marshaller, convierte el java bean en una cadena XML
        Marshaller m = context.createMarshaller();
        //Formateamos el xml para que quede bien
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
        // Lo visualizamos con system out
        m.marshal(milibreria, System.out);
        // Escribimos en el archivo
        m.marshal(milibreria, new File(MIARCHIVO_XML));
    }
}
```

Si ahora deseamos hacer lo contrario, es decir, **leer los datos del documento XML** y convertirlos a objetos Java, utilizaremos las siguientes órdenes:

- Instanciamos el contexto, indicando la clase que será el **RootElement**, en nuestro ejemplo *Libreria*:

```
JAXBContext context = JAXBContext.newInstance(Libreria.class);
```

- Se crea ***Unmarshaller*** en el cotexto de la clase Libreria:

```
Unmarshaller unmars = context.createUnmarshaller();
```

- Utilizamos el método ***unmarshal***, para obtener datos de un ***Reader*** (un file):

```
UnaClase objeto = (UnaClase) unmars.unmarshal(new
FileReader("mifichero.xml"));
```

- Recuperamos un atributo del objeto:

```
System.out.println(objeto.getUnAtributo());
```

- Recuperamos el Arraylist, si lo tiene y visualizamos:

```
ArrayList<ClaseDelArray> lista = objeto.getListadeobjetos();
for (ClaseDelArray obarray : lista) {
    System.out.println("Atributo array: " + obarray.getAtributo());
```

En nuestro ejercicio el código para visualizar el contenido del fichero XML es el siguiente:

```
// Visualizamos ahora los datos del documento XML creado
System.out.println("------------ Leo el XML ---------");
//Se crea Unmarshaller en el contexto de la clase Libreria
Unmarshaller unmars = context.createUnmarshaller();

//Utilizamos el método unmarshal, para obtener datos de un Reader
Libreria libreria2 =(Libreria)
     unmars.unmarshal(new FileReader(MIARCHIVO_XML));

//Recuperamos los datos y visualizamos
System.out.println("Nombre de libreria: "+ libreria2.getNombre());
System.out.println("Lugar de la libreria: " +
                   libreria2.getLugar());
System.out.println("Libros de la libreria: ");
```

```
ArrayList<Libro> lista = libreria2.getListaLibro();
for (Libro libro : lista) {
        System.out.println("\tTítulo del libro: "
        + libro.getNombre()
        + ", autora: " + libro.getAutor());
}
```

ACTIVIDAD 1.7.

Realiza cambios en las clases anteriores y añade las clases que se necesitan, para generar un documento XML que agrupe a varias librerías con varios libros. Haz el programa Java que utilice esas clases, cree dos objetos librerías, una con dos libros, y otra con tres libros y genere un documento con nombre *Librerias.xml* con esta estructura:

```
<MISLIBRERIAS>
 <Libreria>
  <nombre>xxxxx</nombre>
  <lugar>xxxxx</lugar>
  <MiListaLibros>
    <Libro>
      <nombre>xxxxx</nombre>
      <autor>xxxxx</autor>
      <editorial>xxx</editorial>
      <isbn>xxxx</isbn>
    </Libro>
    <Libro>
            .........
            .........
    </Libro>
  </MiListaLibros>
 </Libreria>
```

```
<Libreria>
  <nombre>xxxxx</nombre>
  <lugar>xxxxx</lugar>
  <MiListaLibros>
    <Libro>
      <nombre>xxxxx</nombre>
      <autor>xxxxx</autor>
      <editorial>xxxx</editorial>
      <isbn>xxxx</isbn>
    </Libro>
    <Libro>
        .........
        .........
    </Libro>
  </MiListaLibros>
 </Libreria>
</MISLIBRERIAS>
```

1.9.2. Paso de esquemas XML (.xsd) a clases Java

El compilador de JAXB nos va a permitir generar una serie de clases Java a partir de un esquema XML. Un esquema XML describe la estructura de un documento XML. A los esquemas XML se le llama XSD (*XML Schema Definition*).

JAXB, compila el fichero XSD, creando una serie de clases para cada uno de los tipos que se haya especificado en el XSD. Esas clases serán clases POJO, y las podremos utilizar para crear, y modificar documentos XML utilizando Java.

Ejemplo 2:

Partimos de un XSD, en el que vamos a representar a un artículo y sus ventas. El artículo puede tener varias ventas, mínimo 1 venta. Datos del artículo *(DatosArtic)* son: *codigo*, *denominacion*, *stock* y *precio*. Datos de cada venta *(ventas)* son: *numventa*, *unidades*, *nombrecliente* y *fecha*.

El fichero XSD se llama *ventas_articulo.xsd*, y el contenido es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
```

```
<xsd:element name="ventasarticulos" type="VentasType"/>

<xsd:complexType name="VentasType">
 <xsd:sequence>
  <xsd:element name="articulo" type="DatosArtic"/>
  <xsd:element name="ventas" type="ventas"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="DatosArtic">
 <xsd:sequence>
  <xsd:element name="codigo" type="xsd:string"/>
  <xsd:element name="denominacion" type="xsd:string"/>
  <xsd:element name="stock" type="xsd:integer"/>
  <xsd:element name="precio" type="xsd:decimal"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ventas">
 <xsd:sequence>
  <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
   <xsd:complexType>
    <xsd:sequence>
       <xsd:element name="numventa" type="xsd:integer"/>
       <xsd:element name="unidades">
       <xsd:simpleType>
         <xsd:restriction base="xsd:positiveInteger">
           <xsd:maxExclusive value="100"/>
         </xsd:restriction>
       </xsd:simpleType>
       </xsd:element>
       <xsd:element name="nombrecliente" type="xsd:string"/>
       <xsd:element name="fecha" type="xsd:string"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

¡¡ INTERESANTE ! !

Para saber más sobre XSD puedes consultar estas URLs:

http://www.w3schools.com/xml/schema_howto.asp

http://www.w3schools.com/xml/schema_schema.asp

Este XSD está formado por los siguientes elementos:

- El **elemento principal** (raíz del documento) se va a llamar *ventasarticulos* y su tipo de dato lo vamos a llamar *VentasType*:

```
<xsd:element name = "ventasarticulos" type = "VentasType"/>
```

- El **tipo VentasType**, *type = "VentasType"* formado por las ventas de un artículo, se llama *ventasarticulos*. Es el tipo del elemento principal *<xsd:element name = "ventasarticulos" type = "VentasType"/>*.

Es un tipo complejo, formado por dos elementos, estos van a ser etiquetas en el documento XML, cada uno tiene su tipo, un tipo es el *artículo*, y el otro tipo son las *ventas* del artículo. Estos a su vez estarán compuestos por otras etiquetas *(<xsd:element..)*. En ***name*** se indica el nombre del elemento, y este nombre es el de las etiquetas que luego aparecen en el XML. En el ejercicio lo declaramos así:

```
<xsd:complexType name="VentasType">
  <xsd:sequence>
   <xsd:element name="articulo" type="DatosArtic"/>
   <xsd:element name="ventas" type="ventas"/>
  </xsd:sequence>
 </xsd:complexType>
```

- El **element name artículo** lo utilizamos para representar los datos del artículo su tipo es *type = "DatosArtic"*. Este tipo está formado por las etiquetas del artículo. Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="DatosArtic">
  <xsd:sequence>
   <xsd:element name="codigo" type="xsd:string"/>
   <xsd:element name="denominacion" type="xsd:string"/>
   <xsd:element name="stock" type="xsd:integer"/>
   <xsd:element name="precio" type="xsd:decimal"/>
  </xsd:sequence>
 </xsd:complexType>
```

Para indicar los tipos de datos que van a contener las etiquetas utilizamos:
type = "xsd:string", para representar cadenas,
type = "xsd:integer", para los *Integer*,
type = "xsd:decimal", para los *Float*.

¡ ¡ INTERESANTE ! !

Para saber más sobre tipos XSD podemos consultar las siguientes URLs:

http://www.w3schools.com/xml/schema_dtypes_string.asp
http://www.w3schools.com/xml/schema_dtypes_date.asp
http://www.w3schools.com/xml/schema_dtypes_numeric.asp
http://www.w3schools.com/xml/schema_dtypes_misc.asp
http://www.w3schools.com/xml/schema_example.asp

- El **element name ventas** lo declaramos para representar los datos de las ventas del artículo, su tipo es *type = "ventas"* y se llama *ventas*. Cada artículo podrá tener varias ventas (cada detalle de venta se llamará *venta*). Es un tipo complejo y lo representamos así:

```
<xsd:complexType name="ventas">
  <xsd:sequence>
   <xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
```

```
        <xsd:element name="numventa" type="xsd:integer"/>
        <xsd:element name="unidades">
           <xsd:simpleType>
             <xsd:restriction base="xsd:positiveInteger">
               <xsd:maxExclusive value="100"/>
             </xsd:restriction>
           </xsd:simpleType>
        </xsd:element>
        <xsd:element name="nombrecliente" type="xsd:string"/>
        <xsd:element name="fecha" type="xsd:string"/>
       </xsd:sequence>
      </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
   </xsd:complexType>
```

- Para indicar que del tipo ventas puede haber varias ocurrencias (un artículo puede tener varias ventas) utilizamos los atributos **minOccurs** y **maxOccurs,** mínimo número de filas y sin límite en el máximo. Cada detalle de venta se va a llamar *venta.* Lo escribimos así:

```
<xsd:element name="venta" minOccurs="1" maxOccurs="unbounded">
```

- También para cada elemento *unidades* se ha añadido una restricción, será un número positivo y el valor máximo va a ser 100. Lo escribimos así:

```
<xsd:element name="unidades">
        <xsd:simpleType>
           <xsd:restriction base="xsd:positiveInteger">
             <xsd:maxExclusive value="100"/>
           </xsd:restriction>
        </xsd:simpleType>
</xsd:element>
```

¡¡INTERESANTE!!

Para saber más sobre restricciones en esquemas XSD podemos consultar las siguientes URLs:

http://www.w3schools.com/xml/el_restriction.asp
http://www.w3schools.com/xml/schema_facets.asp

Una vez que tenemos el esquema XSD, un ejemplo de un documento XML, con este esquema podría ser el siguiente: observa la raíz del documento *<ventasarticulos>*, los datos del artículo *<articulo>*, las ventas del artículo *<ventas>* y el detalle de cada venta *<venta>*:

```
<?xml version="1.0" encoding="UTF-8"?>
<ventasarticulos xmlns:xsi='http://www.w3.org/2001/XMLSchema-
instance' xsi:noNamespaceSchemaLocation='ventas_articulo.xsd'>

 <articulo>
      <codigo>ART-112</codigo>
      <denominacion>Pala Padel NOX</denominacion>
      <stock>20</stock>
      <precio>70</precio>
```

```
</articulo>
<ventas>
     <venta>
          <numventa>10</numventa>
          <unidades>2</unidades>
          <nombrecliente>Alicia Ramos</nombrecliente>
          <fecha>10-10-2015</fecha>
     </venta>
     <venta>
          <numventa>11</numventa>
          <unidades>2</unidades>
          <nombrecliente>Pedro García</nombrecliente>
           <fecha>15-10-2015</fecha>
     </venta>
     <venta>
          <numventa>12</numventa>
          <unidades>6</unidades>
          <nombrecliente>Alberto Gil</nombrecliente>
          <fecha>20-10-2015</fecha>
     </venta>
</ventas>
</ventasarticulos>
```

1.9.3. Creación de una aplicación JAXB con Eclipse

Lo siguiente que vamos a hacer es crear las clases a partir del XSD (*ventas_articulo.xsd)*. Y luego trabajaremos con este documento XML para hacer operaciones, como visualizar los datos del XML, añadir, modificar o borrar ventas. El fichero XML con datos de un artículo y sus ventas se llamará *ventasarticulos.xml.* Pasos:

- Creamos un proyecto en Eclipse. Y para esta prueba nos aseguramos de guardar el XSD dentro de la carpeta *src* del proyecto. Y el XML dentro de la carpeta raiz del proyecto.
- Añadimos los siguientes JAR: **jaxb-api.jar, jaxb-core.jar, jaxb-impl.jar, jaxb-jxc.jar y jaxb-xjc.jar.** Se pueden descargar de *https://jaxb.java.net/*, el fichero se llama **jaxb-ri-2.2.11.zip**, los JAR se encuentran en la carpeta *lib* de este fichero.
- Generamos los tipos, es decir, las clases, a partir del fichero XSD, *ventas_articulo.xsd*: botón derecho sobre el fichero XSD, seleccionamos **Generate ->JAXB Classes.** En la siguiente ventana se selecciona el proyecto, y se añade el nombre del paquete donde queremos que se guarden los tipos generados, y dejamos el resto de opciones. Véase Figura 1.4.
- Una vez generados los tipos observa que las clases que se han creado se corresponden con los *types* del fichero *ventas_articulo.xsd*. Véase Figura 1.5

Las clases creadas son *VentasType*, *DatosArtic*, y *Ventas* (el nombre de clase va en mayúscula).

```
<xsd:element name="ventasarticulos" type="VentasType"/>
<xsd:complexType name="VentasType">
  <xsd:sequence>
   <xsd:element name="articulo" type="DatosArtic"/>
```

```
  <xsd:element name="ventas" type="ventas"/>
 </xsd:sequence>
</xsd:complexType>
```

Figura 1.4. Creación de las clases JAXB.

Figura 1.5. Creación de las clases JAXB.

- Se crea además la clase ***ObjectFactory***, que nos va a permitir crear un **ObjectFactory** que se va a utilizar para crear nuevas instancias de las clases Java del esquema XSD. En nuestro ejercicio creará instancias de las clases del paquete *clasesdatos*.

  Esta clase crea un objeto **QName**. **QName** representa un nombre completo tal como se define en las especificaciones XML, está formado por el nombre del namespace (espacio de nombres URI), y el nombre que hemos puesto al elemento principal en el XSD, se le llama prefijo local, en el ejercicio es *ventasarticulos*. **QName** es inmutable, una vez creado no puede ser cambiado. La clase creada es la siguiente:

```
package clasesdatos;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;
```

```
@XmlRegistry
public class ObjectFactory {

 private final static QName _Ventasarticulos_QNAME
                    = new QName("", "ventasarticulos");

 public ObjectFactory() {   }

 public Ventas createVentas() {
        return new Ventas();   }

 public VentasType createVentasType() {
        return new VentasType();  }

 public DatosArtic createDatosArtic() {
        return new DatosArtic();   }

 public Ventas.Venta createVentasVenta() {
        return new Ventas.Venta();   }

 @XmlElementDecl(namespace = "", name = "ventasarticulos")
 public JAXBElement<VentasType>
    createVentasarticulos(VentasType value) {
       return new JAXBElement<VentasType>
         (_Ventasarticulos_QNAME, VentasType.class, null, value);
    }
}
```

- **@XmlRegistry**, se utiliza para marcar una clase que tiene anotaciones ***@XmlElementDecl***. Para implementar JAXB, hay que asegurar que se incluye en la lista de clases y se utilizan para arrancar el **JAXBContext** .
- **@XmlElementDecl.** Si el valor del campo / propiedad va a ser un **JAXBElement** entonces se necesita añadir esta anotación. Un **JAXBElement** obtiene la siguiente información :
  - Nombre del elemento, esto es necesario si se ha mapeado una estructura donde hay varios elementos del mismo tipo. En el ejercicio es *VentasType*.
  - **JAXBElement** puede ser usado para representar un elemento con *xsi:nil = "true"*.

- En nuestro programa Java para crear el contexto JAXB, podemos utilizar el nombre del paquete que contiene a todas las clases, o el nombre de clase **ObjectFactory**:

```
JAXBContext contexto = JAXBContext.newInstance("clasesdatos");
```

O también:

```
JAXBContext contexto =
    JAXBContext.newInstance("clasesdatos.ObjectFactory.class");
```

Vamos ahora a crear una clase principal y con un método para visualizar el contenido del fichero XML *ventasarticulos.xml* utilizando este contexto, el método es el siguiente:

```
public static void visualizarxml() {
  System.out.println("------------------------------ ");
  System.out.println("-------VISUALIZAR XML--------- ");
  System.out.println("------------------------------ ");
  try {
      //Creamos el contexto
      JAXBContext jaxbContext =
          JAXBContext.newInstance(ObjectFactory.class);
      Unmarshaller u = jaxbContext.createUnmarshaller();
      JAXBElement jaxbElement = (JAXBElement) u.unmarshal
             (new FileInputStream("./ventasarticulos.xml"));
      Marshaller m = jaxbContext.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                    Boolean.TRUE);
      // Visualiza por consola
      m.marshal(jaxbElement, System.out);

      //Cargamos ahora el documento en los tipos
      VentasType miventa = (VentasType) jaxbElement.getValue();

      //Obtenemos una instancia para obtener todas las ventas
      Ventas vent = miventa.getVentas();

      // Guardamos las ventas en la lista
      List listaVentas = new ArrayList();
      listaVentas = vent.getVenta();

      System.out.println("----------------------------- ");
      System.out.println("---VISUALIZAR LOS OBJETOS--- ");
      System.out.println("----------------------------- ");
      // Cargamos los datos del artículo
      DatosArtic miartic = (DatosArtic) miventa.getArticulo();
      System.out.println("Nombre art: " +
                         miartic.getDenominacion());
      System.out.println("Coodigo art: " + miartic.getCodigo());
      System.out.println("Ventas  del artículo , hay: " +
                         listaVentas.size());
      //Visualizamos las ventas del artículo
      for (int i = 0; i < listaVentas.size(); i++) {
            Ventas.Venta ve = (Venta) listaVentas.get(i);
            System.out.println("Número de venta: " +
               ve.getNumventa() + ". Nombre cliente: " +
               ve.getNombrecliente() + ", unidades: " +
               ve.getUnidades() + ", fecha: " + ve.getFecha());
      }
  } catch (JAXBException je) {
                System.out.println(je.getCause());
  } catch (IOException ioe) {
                System.out.println(ioe.getMessage());}
}
```

El siguiente método recibe datos de una venta y lo añade al documento XML. Se comprobará antes de insertar que el número de venta no exista:

```
private static void insertarventa
   (int numeventa, String nomcli, int uni, String fecha) {
  System.out.println("---------------------------- ");
  System.out.println("-------AÑADIR VENTA--------- ");
  System.out.println("---------------------------- ");
  try {
     JAXBContext jaxbContext =
             JAXBContext.newInstance(ObjectFactory.class);
     Unmarshaller u = jaxbContext.createUnmarshaller();
     JAXBElement jaxbElement = (JAXBElement)
     u.unmarshal(new FileInputStream("./ventasarticulos.xml"));

     VentasType miventa = (VentasType) jaxbElement.getValue();

     Ventas vent = miventa.getVentas();

     List listaVentas = new ArrayList();
     listaVentas = vent.getVenta();

     // comprobar si existe el número de venta,
     // reccorriendo el arraylist
     int existe = 0;
     for (int i = 0; i < listaVentas.size(); i++) {
          Ventas.Venta ve = (Venta) listaVentas.get(i);
          if (ve.getNumventa().intValue() == numeventa) {
              existe = 1; break;}
     }
     if (existe == 0) {
         // Crear el objeto Ventas.Venta
         Ventas.Venta ve = new Ventas.Venta();
         ve.setNombrecliente(nomcli);
         ve.setFecha(fecha); ve.setUnidades(uni);
         BigInteger nume = BigInteger.valueOf(numeventa);
         ve.setNumventa(nume);
         // Se añade la venta a la lista
         listaVentas.add(ve);
         //Se crea el Marshaller, volcar la lista al fichero XML
         Marshaller m = jaxbContext.createMarshaller();
         m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                       Boolean.TRUE);
         m.marshal(jaxbElement, new
              FileOutputStream("./ventasarticulos.xml"));
         System.out.println("Venta añadida: " + numeventa);
      } else
        System.out.println("En número de venta ya existe: " +
                           numeventa);
  } catch (JAXBException je) {
           System.out.println(je.getCause());
  } catch (IOException ioe) {
           System.out.println(ioe.getMessage());}

}
```

ACTIVIDAD 1.8

Añade al proyecto anterior 3 métodos:

Método que reciba un número de venta y la borre, si existe, del documento XML. El método devolverá *true* si se ha borrado la venta y *false* si no se ha borrado o ha ocurrido algún error.

Método para modificar el stock del artículo. El método recibe una cantidad numérica y se debe sumar al stock del artículo. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

Método para cambiar los datos de una venta, este método recibe el número de venta a modificar, las unidades y la fecha. Se desean modificar esos datos del número de venta. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

Método para cambiar los datos de una venta, este método recibe el número de venta a modificar, las unidades y la fecha. Se desean modificar esos datos del número de venta. El método devuelve *true* si la operación se realiza correctamente y *false* si ocurre algún error.

Ejemplo 3:

En este ejemplo partimos del mismo XSD *ventas_articulo.xsd*, y lo que vamos a hacer es crear un documento nuevo llamado *nuevo_ventasarticulo.xml*, con datos de un artículo y dos ventas, que asignaremos manualmente. Los datos serán los siguientes:

Datos para el artículo:

Codigo	Denominacion	Stock	Precio
Arti 1	Bicicleta Plegable	10	200

Datos para las ventas:

Num venta	Unidades	Nombre de cliente	Fecha
1	2	Alicia Ramos	10-02-2016
2	1	Dori Martín	21-02-2016

El código Java es el siguiente:

```
public static void crearnuevoventasxml() {

    /// Creo el objeto DatosArtic y asigno valores
    DatosArtic articulo = new DatosArtic();
    articulo.setCodigo("Arti 1");
    articulo.setDenominacion("Bicicleta Plegable");
    BigInteger stv = BigInteger.valueOf(10);
    BigDecimal pvv = BigDecimal.valueOf(200);
    articulo.setPrecio(pvv);
    articulo.setStock(stv);
```

```
// Creamos el objeto Ventas
Ventas ventas = new Ventas();

// Creo la primera venta y la añado a ventas
Ventas.Venta ven = new Ventas.Venta();
ven.setNombrecliente("Alicia Ramos");
ven.setNumventa(BigInteger.valueOf(1));
ven.setUnidades(2);
ven.setFecha("10-02-2016");
ventas.getVenta().add(ven);

// Creo la segunda venta y la añado a ventas
ven = new Ventas.Venta();
ven.setNombrecliente("Dori Martín");
ven.setNumventa(BigInteger.valueOf(2));
ven.setUnidades(1);
ven.setFecha("21-02-2016");
ventas.getVenta().add(ven);

// Creo un VentasType y asigno los datos del artículo y sus ventas
VentasType miventaarticulo = new VentasType();
miventaarticulo.setArticulo(articulo);
miventaarticulo.setVentas(ventas);

// Creo el ObjectFactory
ObjectFactory miarticulo = new ObjectFactory();

// Creo El JAXBElement lo obtenemos del ObjectFactory y del VentasType
JAXBElement<VentasType> element =
      miarticulo.createVentasarticulos(miventaarticulo);
// Creo el contexto y el Marshaller
JAXBContext jaxbContext;

try {
    jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
    Marshaller m = jaxbContext.createMarshaller();
    m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
          Boolean.TRUE);
          String fiche = "./nuevo_ventasarticulos.xml";

    m.marshal(element, new FileOutputStream(fiche));
    System.out.println("Venta creada. ");
    // Visualizamos por por consola
    m.marshal(element, System.out);

} catch (JAXBException e) {e.printStackTrace();

} catch (FileNotFoundException e){      e.printStackTrace();}
}
```

COMPRUEBA TU APRENDIZAJE

1. Realiza un programa Java que al ejecutarlo desde la línea de comandos reciba un nombre de directorio. El programa deberá eliminar el directorio y los ficheros contenidos en él.

2. 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 y Localidad: String*.

3. 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.

4. 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.

5. 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.

6. Cuál de las siguientes afirmaciones sobre **SAX** y **DOM** es correcta:

   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.

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

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

9. 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.

10. 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**.

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

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

11. Crea un esquema XSD, con nombre *centrosprofes.xsd* para representar los datos de un centro educativo y sus profesores. Y luego crea una aplicación JAXB para crear un fichero XML llamado *micentro.xml*, con los datos de un centro con tres profesores, y uno de ellos debe ser el director del centro.

La estructura del XML debe quedar de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<datoscentro xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xsi:noNamespaceSchemaLocation = 'centrosprofes.xsd'>
  <centro>
      <codigocentro></codigo>
      <nombrecentro></nombre>
      <direccion></direccion>
      <director>
           <codigoprofesor></codigoprofesor>
           <nombreprofe></nombreprofe >
      </director>
  </centro>
  <profesores>
      <profe>
            <codigoprofesor></codigoprofesor>
            <nombreprofe></nombreprofe >
      </profe>
      ...................
  </profesores>
</datoscentro>
```

CAPÍTULO 2

MANEJO DE CONECTORES

OBJETIVOS

- 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.
- Crear informes con datos almacenados en bases de datos.

CONTENIDOS

Desfase objeto-relacional.

Protocolos de acceso a bases de datos. Conectores.

Bases de datos embebidas.

Establecimiento de conexiones.

Ejecución de sentencias de descripción de datos.

Ejecución de sentencias de modificación de datos.

Ejecución de consultas.

Ejecución de procedimientos.

Generación de informes.

RESUMEN

En este capítulo aprenderemos a acceder a los datos almacenados en distintas bases de datos relacionales utilizando el lenguaje de programación Java. Realizaremos programas para acceder a las distintas bases de datos, para ello, usaremos diferentes conectores, cada base de datos necesitará su conector. Aprenderemos a generar informes con JassperReports.

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 Capítulo 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 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-tools-win32-x86-3100100.zip**, al descomprimirlo obtenemos varios ficheros ejecutables, entre ellos **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 la sesión se escribe el comando **.quit**, el comando **.tables** muestra las tablas creadas:

```
D:\>sqlite3 D:\DB\SQLITE\ejemplo.db
SQLite version 3.11.1 2016-03-03 16:17:53
Enter ".help" for usage hints.
sqlite> CREATE TABLE departamentos (
   ...>  dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
   ...>  dnombre  VARCHAR(15),
   ...>  loc      VARCHAR(15)
   ...> );
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> .tables
departamentos
sqlite> SELECT * FROM departamentos;
10|CONTABILIDAD|SEVILLA
20|INVESTIGACIÓN|MADRID
30|VENTAS|BARCELONA
40|PRODUCCIÓN|BILBAO
sqlite> .quit

D:\>sqlite3 D:\DB\SQLITE\ejemplo.db
```

Para instalar SQLite en Linux escribimos desde la línea de comandos:

```
$sudo apt-get install sqlite3
```

Y para ejecutar SQLite escribimos lo siguiente para crear la base de datos en la carpeta */home/usuario/DB/SQLITE* (que tiene que existir):

```
$ sqlite3 /home/usuario/DB/SQLITE/ejemplo.db
```

ACTIVIDAD 2.1

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

DEPARTAMENTOS:

DEPT_NO numérico clave primaria, DNOMBRE VARCHAR(15), LOC VARCHAR(15).

EMPLEADOS:

EMP_NO numérico clave primaria, APELLIDO VARCHAR(10), OFICIO VARCHAR(10), DIR numérico, FECHA_ALT DATE, SALARIO numérico, COMISION numérico, DEPT_NO numérico, es clave ajena y referencia a la tabla DEPARTAMENTOS.

2.3.2. Apache Derby

Apache Derby, es una base de datos relacional de código abierto, implementada en su totalidad en Java que forma parte del **Apache DB Project** 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 servidor de red Derby. 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-10.12.1.1-bin.zip**, y lo descomprimimos por ejemplo en *D:\db-derby-10.12.1.1-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 o en nuestro proyecto Eclipse o Netbeans.

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-10.12.1.1-bin\bin*). Desde la línea de comandos del DOS nos dirigimos a dicha carpeta y ejecutamos el fichero **ij.bat**:

```
D:\db-derby-10.12.1.1-bin\bin>ij
Versi¾n de ij 10.12
ij>
```

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

```
ij> 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 se encuentra en la carpeta *bin*) que crea la tabla *EMPLEADOS* e inserta filas en ella, al final se ejecuta el comando **exit;** para salir:

```
D:\db-derby-10.12.1.1-bin\bin>ij
Versión de ij 10.12
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> 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-
12-20', 1500,390,30);
1 fila insertada/actualizada/suprimida
ij> INSERT INTO empleados VALUES (7521,'SALA','VENDEDOR',7698,'1991-02-
22', 1625,650,30);
1 fila insertada/actualizada/suprimida
ij> exit;
```

El comando **show tables;** muestra las tablas existentes en la base de datos. Para obtener ayuda podemos escribir el comando **help;**

Para volver a usar la base de datos escribimos la siguiente orden desde la línea de comandos de ij: *connect 'jdbc:derby:D:\DB\DERBY\ejemplo';*.

Para utilizar la base de datos en Linux (Ubuntu) tenemos que tener la máquina virtual de Java instalada. Seguimos los siguientes pasos para instalar **Oracle Java 8** que incluye los paquetes JDK8 y JRE8:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

A continuación se ejecuta la siguiente orden que instala el paquete para establecer las variables de entorno con la instalación de Java (antes ejecutamos *sudo apt-get update*):

```
sudo apt-get install oracle-java8-set-default
```

A continuación descargamos la versión para Linux y la descomprimimos en la carpeta */opt*, en este caso se ha descargado la versión: **db-derby-10.12.1.1-bin.tar.gz**. Se creará la carpeta */opt/db-derby-10.12.1.1-bin*. Configuramos la variable **DERBY_HOME** con el nombre de carpeta donde se ha descargado, ejecutando desde la línea de comandos:

```
$ export DERBY_HOME=/opt/db-derby-10.12.1.1-bin
```

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

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

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

```
$ cd $DERBY_HOME/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/*, a continuación se crea una tabla y después se finaliza la conexión ejecutando la orden **exit**:

```
$ java org.apache.derby.tools.ij
ij version 10.12
ij> connect 'jdbc:derby:/home/usuario/DB/DERBY/ejemplo;create=true';
Wed Mar 23 17:31:12 PDT 2016 Thread[main,5,main]
java.io.FileNotFoundException: derby.log (Permission denied)
----------------------------------------------------------------
Wed Mar 23 17:31:14 PDT 2016:
Booting Derby version The Apache Software Foundation - Apache Derby -
10.12.1.1 - (1704137): instance a816c00e-0153-a608-42cc-000002f09800
on database directory /home/usuario/DB/DERBY/ejemplo with class loader
sun.misc.Launcher$AppClassLoader@610455d6
Loaded from file:/opt/db-derby-10.12.1.1-bin/lib/derby.jar
java.vendor=Oracle Corporation
java.runtime.version=1.8.0_74-b02
user.dir=/opt/db-derby-10.12.1.1-bin/bin
os.name=Linux
```

```
os.arch=amd64
os.version=4.2.0-16-generic
derby.system.home=null
Database Class Loader started - derby.database.classpath=''
ij> CREATE TABLE EJEMPLO1 (
> NOMBRE VARCHAR(15)
> );
0 rows inserted/updated/deleted
ij> exit;
------------------------------------------------------------
Wed Mar 23 17:33:04 PDT 2016: Shutting down Derby engine
------------------------------------------------------------
Wed Mar 23 17:33:05 PDT 2016:
Shutting down instance a816c00e-0153-a608-42cc-000002f09800 on database
directory /home/usuario/DB/DERBY/ejemplo with class loader
sun.misc.Launcher$AppClassLoader@610455d6
------------------------------------------------------------
```

Para volver a usar la base de datos escribimos 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 2.2

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. Soporta la mayor parte de las características y funciones incluidas en el estándar SQL:2011. Puede mantener la base de datos en memoria o en ficheros en disco. La última versión 2.3.4 mejora en el acceso y la gestión de grandes conjuntos de datos. Es compatible con hasta 270 mil millones de filas de datos en una sola base de datos y con capacidad de copia de seguridad en caliente.

Desde la URL *https://sourceforge.net/projects/hsqldb/files/* podemos descargarnos la última versión estable. En este caso, descargamos el fichero **hsqldb-2.3.3.zip**. Lo descomprimimos, por ejemplo, en la unidad D se creará una carpeta con el nombre del fichero ZIP (*D:\hsqldb-2.3.3\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* (si no se crea la carpeta para la base de datos se mezclarán en *data* todas las bases de datos).

Abrimos la línea de comandos del DOS (como usuario administrador) 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 instalar la base de datos en Ubuntu primero guardamos el fichero **hsqldb-2.3.3.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-2.3.3.zip
```

Se creará en */opt* una carpeta con nombre *hsqldb-2.3.3*, 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 de nombre *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 de la siguiente manera: *jdbc:hsqldb:file:/home/usuario/DB/HSQLDB/ejemplo*. El resto de operaciones son similares.

ACTIVIDAD 2.3

Crea las tablas EMPLEADOS y DEPARTAMENTOS en HSQLDB inserta filas en ellas.

2.3.4. H2

H2 es un sistema gestor de base de datos relacional programado íntegramente en Java. Está disponible como software de código libre bajo la Licencia Pública de Mozilla o la Eclipse Public License. Desde la web *http://www.h2database.com/html/main.html* podemos descargarnos la última versión. Nos descargamos la versión ZIP para todas las plataformas **h2-2016-01-21.zip**. Lo descomprimimos por ejemplo en la unidad D, se creará una carpeta con el nombre *D:/h2*. Desde la línea de comandos del DOS nos dirigimos a la carpeta *D:\h2\bin* y ejecutamos el fichero **h2.bat** para arrancar la consola (también podemos hacer doble clic en el fichero para ejecutarlo):

```
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 se crean automáticamente), pulsamos el botón *Guardar* para que guarde la configuración y cada vez que queramos conectarnos a nuestra base de datos usemos ese nombre y pulsamos el botón *Conectar*. Podemos pulsar el botón *Probar 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, véase 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.

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

Desde la zona de instrucciones SQL podremos escribir las sentencias para crear tablas, insertar filas, etc., véase Figura 2.5. Los botones Ejecutar 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.

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.

Figura 2.5. Ejecución de sentencias en H2.

En Ubuntu extraemos el fichero **h2-2016-01-21.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 haciendo doble clic en el fichero), desde la línea de comandos escribimos:

```
/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 ***JDBC URL*** escribimos 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*. Si las carpetas no existen se crean automáticamente, y dentro los ficheros para la base de datos de nombre *ejemplo*.

ACTIVIDAD 2.4

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 (tamaño de 2GB a 264GB).

Desde la URL *http://supportservices.actian.com/versant/default.html* podemos descargarnos la última versión. Para el ejemplo se ha descargado la versión **db4o-8.0.276.16149-java.zip**. Al descomprimirla hay una carpeta de nombre *lib* donde se encuentra el fichero JAR **db4o-8.0.276.16149-all-java5.jar** que necesitamos para utilizar el motor de la base de datos. Desde la URL *https://sourceforge.net/projects/db4o/files/db4o/* podemos descargar versiones anteriores de Db4o.

En Eclipse para usar el JAR seleccionamos nuestro proyecto, pulsamos el botón derecho del ratón y seleccionamos **Build Paths-> Add External Archives**, véase Figura 2.6. Se visualiza una ventana desde la que localizaremos el fichero JAR a incluir en el proyecto (conviene crear una carpeta e ir incluyendo los JAR que después usaremos en los ejercicios). Se mostrará en nuestro proyecto un elemento más, ***Referenced Libraries***, con el fichero JAR añadido.

Figura 2.6. Añadir el JAR al proyecto Eclipse.

También podemos añadir los JAR desde la opción **Build Path –> Configure Build Path**. 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.

A continuación se muestra la clase *Main.java* que crea una base de datos (si no existe) de nombre *DBPersonas.yap* y almacena objetos *Persona* en ella:

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

public class Main {
   static String BDPer = "DBPersonas.yap";

   public static void main(String[] args) {
      ObjectContainer db =
```

```
    Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

    // Se crean objetos Persona
    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 metodo main
}// fin de la clase Main
```

Se ha definido la clase *Persona* formada por los atributos *nombre* y *ciudad* y los métodos *get* y *set* para obtener y almacenar los valores de un objeto *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 nombre) {
        this.nombre = nombre;
     }
     public String getCiudad() {
        return ciudad;
     }
     public void setCiudad(String ciudad) {
       this.ciudad = ciudad;
    }

}//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 campos y pulsar el botón *OK*. Para generar los constructores pulsamos sobre la

opción ***Generate Constructor using Fields*** para generar el constructor con parámetros, o ***Generate Constructors from Superclass*** para generar el constructor sin parámetros.

Figura 2.7. Generar *getters*, *setters* y constructores.

El paquete Java **com.db4o** contiene casi toda la funcionalidad de la base de datos. Para este ejemplo necesitamos importar la clase **com.db4o.Db4oEmbedded**, y la interfaz **com.db4o.ObjectContainer**.

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 el nombre 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 al método ***close()***:

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

- Para almacenar un objeto utilizamos el método ***store()***:

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

- Para recuperar objetos podemos utilizar el sistema de consultas QBE (*Query-By-Example*) mediante el método ***queryByExample()***. El siguiente ejemplo muestra todos los objetos *Persona* existentes en la base de datos, los resultados se proporcionan en forma de **ObjectSet**. Si no existe ningún objeto el método ***size()*** sobre el objeto **ObjectSet** devolverá 0:

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

```
public class Consulta1 {
   static String BDPer = "DBPersonas.yap";
   public static void main(String[] args) {
     ObjectContainer db =
         Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), BDPer);

     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.printf("Número de registros: %d %n",
                                result.size());
          while (result.hasNext()) {
            Persona p = result.next();
            System.out.printf("Nombre: %s, Ciudad: %s %n",
                                p.getNombre(), p.getCiudad());
          }
     }
     db.close(); // cerrar base de datos

   }//main
}//fin Consulta1
```

La siguiente consulta obtiene los objetos *Persona* cuyo nombre es Juan:

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

La siguiente consulta obtiene 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 el método ***store()***. El siguiente ejemplo modifica la ciudad de Juan a Toledo y luego visualiza sus datos (si hay varios objetos *Persona* con nombre Juan solo se modifica el primero que encuentre, para modificarlos todos habría que hacer un bucle):

```
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.printf("Nombre:%s, Nueva Ciudad: %s %n",
```

```
                                existe.getNombre(), existe.getCiudad());
}
```

- Para eliminar objetos utilizamos el método *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 {
     System.out.printf("Registros a borrar: %d %n", result.size());
     while (result.hasNext()) {
       Persona p = result.next();
       db.delete(p);
       System.out.println("Borrado....");
     }
}
```

En el Capítulo 4 se profundizará más acerca de las bases de datos orientadas a objetos.

ACTIVIDAD 2.5

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 del 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.

- **Microsoft SQL Server Compact** (*SQL Server CE*): Es una base de datos compacta y con una gran variedad de funciones diseñada para entornos móviles. 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:
  - Posee un motor de base de datos así como un procesador y un optimizador de consultas especialmente diseñado para entornos móviles.
  - Soporta un subconjunto de tipos de datos y de sentencias *Transact-SQL* de *SQL Server*.
  - En cuanto a los datos de tipo texto, únicamente soporta tipos de datos de cadena compatibles con Unicode (nchar, nvarchar, ntext).
  - Se integra desde la versión 3.0, con *Microsoft Visual Studio* (incluyendo la edición *Express* desde la versión 3.5) y *SQL Server Management Studio*.
  - A nivel de seguridad ofrece la posibilidad de cifrado del fichero de base de datos con una contraseña de acceso restringida.
  - *SQL Server Compact 4.0* se ha optimizado y ajustado para usarse con aplicaciones Web *ASP.NET*.

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. Está escrito en C.
- **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 qué 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 interfaz 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 tienen el mismo nombre que la base de datos (en el ejemplo el servidor de base de datos MySQL está instalado mediante el paquete *XAMPP para Linux 1.8.2-2*) . Para poder acceder necesitamos crear un origen de datos ODBC. Desde Linux (Ubuntu 12) 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 la carpeta */usr/lib/i386-linux-gnu/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/i386-linux-gnu/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 (de nombre *conexión.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, &nFilas );
       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 conexion.c -lodbc -std=gnu99
conexion.c: En la función 'main':
conexion.c:33:8: aviso: formato '%u' espera un argumento de tipo
'unsigned int', pero el argumento 2 es de tipo 'SQLINTEGER' [-Wformat]
# ./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 (véase 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. Dos modelos de acceso a bases de datos

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 envían 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, véase Figura 2.10. En este caso los drivers no tienen que residir en la máquina cliente.

Figura 2.9. Modelo de dos capas.

Figura 2.10. Modelo de tres capas.

Un **servidor de aplicaciones** es una implementación de la especificación J2EE (*Java 2 Platform Enterprise Edition*). J2EE es un entorno centrado en Java para desarrollar, construir y desplegar aplicaciones empresariales multicapa basadas en la Web. Existen diversas implementaciones, cada una con sus propias características. Algunas de ellas son las siguientes: *BEA WebLogic, IBM WebSphere, Oracle IAS, Borland AppServer*, etc.

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. Convierte las llamadas al API de JDBC en llamadas 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 de la base de datos. 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 el servidor de base de datos. Traduce las llamadas al API de JDBC Java en llamadas propias del protocolo de red independiente de la base de datos y a continuación son traducidas por un software intermedio (*Middleware*) al protocolo usado por el motor de base de datos. El driver JDBC no comunica directamente con la base de datos, comunica con el software intermedio, que a su vez comunica con la base de datos. Son útiles para aplicaciones que necesitan interactuar con diferentes formatos de bases de datos, ya que usan el mismo driver JDBC sin importar la base de datos específica. 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 INTERFAZ	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/8/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:

Figura 2.12. Base de datos *ejemplo*.

```
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,
 CONSTRAINT FK_DEP FOREIGN KEY (dept_no) REFERENCES
                                         departamentos(dept_no)
) ENGINE=InnoDB;
```

El siguiente programa ilustra los pasos de funcionamiento de JDBC accediendo a la base de datos anterior y mostrando el contenido de la tabla *departamentos*:

```
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();
        String sql = "SELECT * FROM departamentos";
        ResultSet resul = sentencia.executeQuery(sql);
```

```
        //Recorremos el resultado para visualizar cada fila
        //Se hace un bucle mientras haya registros y se van mostrando
        while (resul.next()) {
          System.out.printf("%d, %s, %s %n",
              resul.getInt(1),
              resul.getString(2),
              resul.getString(3));
        }

        resul.close();      // Cerrar ResultSet
        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 muestra la siguiente salida:

```
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 el paquete **java.sql.***. También se ha incluido todo el programa en un **try-catch** ya que casi todos los métodos relativos a la 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()*** de la siguiente manera:

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

La sintaxis del método ***getConnection()*** es la siguiente:

```
public static Connection getConnection
      (String url, String user, String password) throws SQLException
```

El primer parámetro del método ***getConnection()*** representa la URL de conexión a la base de datos. Tiene el siguiente formato para conectarse a MySQL:

```
jdbc:mysql://nombre_host:puerto/nombre_basedatos
```

Donde

- **jdbc:mysql** indica que estamos utilizando un driver JDBC para MySQL.
- **nombre_host** indica el nombre del servidor donde está la base de datos. Aquí puede ponerse una IP o un nombre de máquina que esté en la red. Si especificamos *localhost* como nombre de servidor, estamos indicando que el servidor de base de datos se encuentra en la misma máquina en la que se ejecuta el programa Java.
- **puerto** es el puerto predeterminado para las bases de datos MySQL, por defecto es **3306**. Si no se pone se asume este valor.
- **nombre_basedatos** es el nombre de la base de datos a la que nos vamos a conectar y que debe existir en MySQL. En este caso el nombre es *ejemplo*.

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();
String sql = "SELECT * FROM departamentos";
ResultSet resul = sentencia.executeQuery(sql);
```

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 la 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, aunque recorrer 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.printf("%d, %s, %s %n",
            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.printf("%d, %s, %s %n",
        resul.getInt("dept_no"),
        resul.getString("dnombre"),
        resul.getString("loc") );
```

ResultSet dispone de varios métodos para mover el puntero del objeto **ResultSet**:

Método	Función
`boolean next()`	Mueve el puntero del objeto **ResultSet** una fila hacia adelante a partir de la posición actual. Devuelve *true* si el puntero se posiciona correctamente y *false* si no hay registros en el **ResultSet**
`boolean first()`	Mueve el puntero del objeto **ResultSet** al primer registro de la lista. Devuelve *true* si el puntero se posiciona correctamente y *false* si no hay registros
`boolean last()`	Mueve el puntero del objeto **ResultSet** al último registro de la lista. Devuelve true si el puntero se posiciona correctamente y false si no hay registros
`boolean previous()`	Mueve el puntero del objeto **ResultSet** al registro anterior de la lista. Devuelve *true* si el puntero se posiciona correctamente y *false* si se coloca antes del primer registro
`void beforeFirst()`	Mueve el puntero del objeto **ResultSet** justo antes del primer registro
`int getRow()`	Devuelve el número de registro actual. Para el primer registro del objeto **ResultSet** devuelve 1, para el segundo 2 y así sucesivamente

El siguiente código muestra el número de filas recuperadas en la consulta y seguidamente muestra los datos de cada fila acompañada del número de fila:

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

resul.last(); //Nos situamos en el último registro
```

```
System.out.println ("NÚMERO DE FILAS: " + resul.getRow());

resul.beforeFirst(); //Nos situamos antes del primer registro

//Recorremos el resultado para visualizar cada fila
while (resul.next())
    System.out.printf("Fila %d: %d, %s, %s %n",
      resul.getRow(),
      resul.getInt(1),
      resul.getString(2),
      resul.getString(3) );
```

La salida muestra la siguiente información:

```
NÚMERO DE FILAS: 4
Fila 1: 10, CONTABILIDAD, SEVILLA
Fila 2: 20, INVESTIGACIÓN, MADRID
Fila 3: 30, VENTAS, BARCELONA
Fila 4: 40, PRODUCCIÓN, BILBAO
```

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 2.6

Tomando como base el programa que ilustra los pasos de funcionamiento de JDBC 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 y el nombre de su departamento.

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 *MySQL ODBC 5.3 ANSI Driver* 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 ***TCP/IP 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*. En ***Port*** se escribe el puerto por el que escucha el servidor MySQL, en este caso es *3307*. Como nombre de usuario escribimos *root* 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 se ha elegido *test*. El botón *Test* nos permite probar la conexión. Pulsamos el botón *OK*, véase Figura 2.14. 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. Para cada esquema de base de datos es necesario crear un origen de datos.

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 desde la sección *MySQL Connectors* (*Connector/ODBC 5.3.6*). Se elige la plataforma en la que se usará el driver, en este caso se ha descargado la versión para Windows de 32 bits. Puede ser un fichero con el nombre ***mysql-connector-odbc-5.3.6-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.

IMPORTANTE!!

Estos ejemplos funcionarán en las versiones de Java inferiores a la 8. A partir de la versión Java SE 8 no se incluye el puente JDBC-ODBC con el JDK. El puente JDBC-ODBC no es compatible con las versiones más recientes de la especificación JDBC.

ACTIVIDAD 2.7

Realiza la Actividad 2.6 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 cómo conectarnos a través de JDBC a las bases de datos embebidas estudiadas anteriormente. Hemos de crear 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/*. Para realizar las pruebas necesitaremos tener el conector Java (fichero JAR) correspondiente para cada una de las bases de datos.

Conexión a SQLite

Para conectarnos a SQLite necesitamos la librería **sqlite-jdbc-3.8.11.2.jar** que se puede descargar desde la URL *https://bitbucket.org/xerial/sqlite-jdbc/downloads*. Partimos del programa Java inicial que recorre la tabla *departamentos* (su nombre es *Main.java*) de la base de datos *ejemplo* de MySQL. En el entorno gráfico que usemos para ejecutar el programa incluimos el fichero JAR o lo incluimos en el CLASSPATH si lo ejecutamos desde la línea de comandos.

En el programa Java cambiamos dos cosas: 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");
```

El ejemplo en Linux es similar, solo habría que cambiar en la conexión la carpeta donde se encuentra la base de datos: *"jdbc:sqlite:/home/usuario/DB/SQLITE/ejemplo.db"*.

Conexión a Apache Derby

Para conectarnos a Apache Derby necesitamos la librería **derby.jar** (que se encuentra en la carpeta donde se instaló Derby: */db-derby-10.12.1.1-bin/lib*). El driver para la conexión a la base de datos se llama: **org.apache.derby.jdbc.EmbeddedDriver**:

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

Conexión a HSQLDB

Para conectarnos a HSQLDB necesitamos la librería **hsqldb.jar** que se puede obtener de la carpeta *lib* obtenida al descomprimir el fichero **hsqldb-2.3.3.zip**. En este caso el driver se llama **org.hsqldb.jdbcDriver** y la conexión a la base de datos es la siguiente:

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

Conexión a H2

Para conectarnos a H2 necesitamos la librería **h2-1.4.191.jar** que se puede obtener de la carpeta *bin* en la que se encuentra al descomprimir el fichero **h2-2016-01-21.zip**. El driver se llama **org.h2.Driver** y la conexión a la base de datos es la siguiente:

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

En este caso es necesario incluir el nombre del usuario y la clave en la conexión. El nombre es *"sa"* y la clave se dejó en blanco cuando se creó la base de datos.

Conexión a Access

Para conectarnos a una base de datos Access necesitamos las siguientes librerías: **commons-lang-2.6.jar, commons-logging-1.1.1.jar, hsqldb.jar, jackcess-2.1.2.jar, ucanaccess-3.0.2.jar,** y **ucanload.jar.** Se pueden descargar de la URL: *http://ucanaccess.sourceforge.net/site.html*. Al acceder al sitio podremos descargar un fichero similar a **UCanAccess-3.0.4-src.zip** con ejemplos, o el fichero **UCanAccess-3.0.4-bin.zip** que contiene los JAR.

Para conectarnos a una base de datos Access escribiremos:

```
Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
Connection conn = DriverManager.getConnection
                  ("jdbc:ucanaccess://mibasedatosaccess");
```

Por ejemplo, quiero conectarme a la base de datos ***ejemplo.accdb***, que la tengo guardada en la carpeta raíz del proyecto, en la conexión escribiré lo siguiente:

```
Connection conn = DriverManager.getConnection
                  ("jdbc:ucanaccess://./ejemplo.accdb");
```

Conexión a MySQL

Para conectarnos a MySQL necesitamos la librería **mysql-connector-java-5.1.38-bin.jar**, que podemos descargar desde la URL *http://dev.mysql.com/downloads/connector/j/*. Se descarga un fichero ZIP, y dentro de él se encuentra el JAR. El driver se llama **com.mysql.jdbc** y la conexión es la siguiente:

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

Conexión a Oracle

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/apps-tech/jdbc-112010-090769.html*. Para el ejemplo se ha descargado el driver **ojdbc6.jar**. Necesitamos saber el nombre de servicio que usa la base de datos para incluirlo en la URL de la conexión. Normalmente para la versión *Express Edition* el nombre es *XE*. El driver se llama **oracle.jdbc.driver.OracleDriver**, y la conexión a la base de datos es la siguiente:

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

2.3. 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.

La interfaz **DatabaseMetaData** proporciona información sobre la base de datos a través de múltiples métodos de los cuales es posible obtener gran cantidad de información. Muchos de estos métodos devuelven un **ResultSet**, algunos de los que veremos en los siguientes ejemplos son:

Método	Descripción
getTables()	Proporciona información sobre las tablas y vistas de la base de datos
getColumns()	Devuelve información sobre las columnas de una tabla
getPrimaryKeys()	Proporciona información sobre las columnas que forman la clave primaria de una tabla
getExportedKeys()	*Devuelve información sobre las claves ajenas que utilizan la* clave primaria de una tabla
getImportedKeys()	Devuelve información sobre las claves ajenas existentes en una tabla
getProcedures()	Devuelve información sobre los procedimientos almacenados
Más información sobre métodos de DatabaseMetaData: *https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html*	

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 interfaz **Connection** devuelve un objeto **DataBaseMetaData** que contiene información sobre la base de datos representada por el objeto **Connection**:

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

      DatabaseMetaData dbmd = conexion.getMetaData();
      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.printf("Nombre : %s %n", nombre );
      System.out.printf("Driver : %s %n", driver );
      System.out.printf("URL    : %s %n", url );
      System.out.printf("Usuario: %s %n", 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
           String esquema = resul.getString(2); //columna 2
```

```
            String tabla = resul.getString(3);    //columna 3
            String tipo = resul.getString(4);     //columna 4
            System.out.printf("%s - Catalogo: %s, Esquema: %s,
               Nombre: %s %n", tipo, catalogo, esquema, tabla);
        }
        conexion.close(); //Cerrar conexión
    }
    catch (ClassNotFoundException cn) {cn.printStackTrace();}
    catch (SQLException e) {e.printStackTrace();}
  }//fin de main
}//fin de la clase
```

La ejecución del programa 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. Su sintaxis es:

```
public abstract ResultSet getTables(
            String catalogo, String esquema,
            String patronDeTabla, String tipos[]) throws SQLException
```

- Primer parámetro: catálogo de la base de datos. El método obtiene las tablas del catálogo indicado, al poner *null*, indicamos el catálogo actual.
- Segundo parámetro: esquema de la base de datos (nombre de usuario). 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 guion bajo o porcentaje, por ejemplo, *"de%"* obtendría todas las tablas cuyo nombre empieza por *"de"*.
- El cuarto parámetro es un array de *String*, en el que indicamos qué tipos de objetos queremos obtener, por ejemplo: *TABLE* (para tablas), *VIEW* (para vistas); al poner *null*, nos devolverá todos los tipos de objetos ya sean tablas o vistas. Los tipos válidos son: *TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS* y *SYNONYM*. El siguiente ejemplo nos devolvería las tablas y los sinónimos:

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

Cada fila de **ResultSet** que devuelve ***getTables()*** tiene información sobre una tabla. La descripción de cada columna tiene las siguientes columnas: *TABLE_CAT* (columna 1, el nombre del catálogo al que pertenece la tabla), *TABLE_SCHEM*, (columna 2, el nombre del esquema al que pertenece la tabla), *TABLE_NAME* (columna 3, el nombre de la tabla o vista), *TABLE_TYPE* (columna 4, el tipo TABLE o VIEW), *REMARKS* (columna 5, comentarios), *TYPE_CAT*, *TYPE_SCHEM*, *TYPE_NAME*, *SELF_REFERENCING_COL_NAME*, y *REF_GENERATION*. Para obtener estos resultados también podríamos haber puesto en el código anterior el nombre de la columna en lugar del número:

```
String catalogo  = resul.getString("TABLE_CAT");    //columna 1
String esquema   = resul.getString("TABLE_SCHEM"); //columna 2
String tabla     = resul.getString("TABLE_NAME");   //columna 3
String tipo      = resul.getString("TABLE_TYPE");   //columna 4
```

ACTIVIDAD 2.8

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()***: Devuelve un objeto **ResultSet** con información sobre las columnas de una tabla o tablas. La descripción de cada columna tiene las siguientes columnas: *TABLE_CAT*, *TABLE_SCHEM*, *TABLE_NAME*, *COLUMN_NAME*, *DATA_TYPE*, *TYPE_NAME*, *COLUMN_SIZE*, *BUFFER_LENGTH*, *DECIMAL_DIGITS*, *NUM_PREC_RADIX*, *NULLABLE*, *REMARKS*, *COLUMN_DEF*, *SQL_DATA_TYPE*, *SQL_DATETIME_SUB*, *CHAR_OCTET_LENGTH*, *ORDINAL_POSITION*, *IS_NULLABLE*, *SCOPE_CATALOG*, *SCOPE_SCHEMA*, *SCOPE_TABLE*, *SOURCE_DATA_TYPE*, *IS_AUTOINCREMENT* e *IS_GENERATEDCOLUMN*. Su sintaxis es:

```
public abstract ResultSet getColumns(
       String catalogo, String Esquema,
       String patronNombreDeTabla, String patronNombreDeColumna)
throws SQLException
```

Para el patrón de nombre de la tabla y de la columna se puede utilizar el carácter guion 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 nombCol = 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.printf("Columna: %s, Tipo: %s, Tamaño: %s,
      ¿Puede ser Nula:? %s %n", nombCol, tipoCol, tamCol, nula);
}
```

Visualiza la siguiente información:

```
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()***: devuelve la lista de columnas que forman la clave primaria de la tabla especificada. La descripción de cada columna de la clave primaria tiene las siguientes columnas: *TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME* y *KEY_SEQ*. La sintaxis es la siguiente:

```
public abstract ResultSet getPrimaryKeys(
              String catalogo, String esquema, String tabla)
throws SQLException
```

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()***:devuelve la lista de todas las claves ajenas que utilizan la clave primaria de la tabla especificada. La descripción de cada columna de clave ajena tiene las siguientes columnas: *PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, PKCOLUMN_NAME, FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, FKCOLUMN_NAME, KEY_SEQ, UPDATE_RULE, DELETE_RULE, FK_NAME, PK_NAME y DEFERRABILITY*. La sintaxis es:

```
public abstract ResultSet getExportedKeys
  (String catalogo, String esquema, String tabla) throws SQLException
```

El siguiente ejemplo muestra las tablas y sus claves ajenas que referencian a la tabla *departamentos*, en este caso solo la tabla *empleados*:

```
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.printf("Tabla PK: %s, Clave Primaria: %s %n",
                    pk_tablename, pk_name);
  System.out.printf("Tabla FK: %s, Clave Ajena: %s %n",
                    fk_tablename, fk_name);
}
```

Visualiza la siguiente información:

```
Tabla PK: departamentos, Clave Primaria: dept_no
Tabla FK: empleados, Clave Ajena: dept_no
```

El método no devuelve nada si queremos ver las claves ajenas que referencian a la tabla *empleados*, ***dbmd.getExportedKeys(null, "ejemplo", "empleados")***, ya que la tabla *empleados* no es referenciada por ninguna clave ajena.

A la hora de crear una tabla es recomendable definir las restricciones de clave ajena asignándolas un nombre, usando la cláusula *CONSTRAINT nombre FOREIGN KEY (col1, col2,...) REFERENCES tabla(col1,col2,...)*. De esta manera el método ***getExportedKeys()*** nos devolverá la información deseada.

- ***getImportedKeys():*** devuelve la lista de claves ajenas existentes en la tabla indicada. 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. La sintaxis es:

```
public abstract ResultSet getImportedKeys
        (String catalogo, String esquema, String tabla)
throws SQLException
```

- ***getProcedures()***: devuelve la lista de procedimientos almacenados. Cada descripción de procedimiento tiene las siguientes columnas: *PROCEDURE_CAT* (columna 1), *PROCEDURE_SCHEM* (columna 2), *PROCEDURE_NAME* (columna 3), *REMARKS* (columna 7), *PROCEDURE_TYPE* (columna 8) y *SPECIFIC_NAME* (columna 9). La sintaxis es:

```
public abstract ResultSet getProcedures
   (String catalogo, esquema, String procedure) throws SQLException
```

Para probar el método ***getProcedures()*** creamos algunos procedimientos y funciones. Por ejemplo, creamos la función de nombre *SUMAR* que recibe dos números y devuelve la suma:

Ejemplo de función en Oracle:

```
CREATE OR REPLACE FUNCTION SUMAR (N1 NUMBER, N2 NUMBER)
RETURN NUMBER AS
BEGIN
      RETURN N1 + N2;
END SUMAR;
/
Para probarla  escribimos: SELECT SUMAR(2,22) FROM DUAL;
```

Ejemplo de función en MySQL:

```
DELIMITER //
CREATE FUNCTION SUMAR (N1 INT, N2 INT)  RETURNS INT
BEGIN
  RETURN N1 + N2;
END;
//
```

Para probarla escribimos: *SELECT SUMAR(2,22)*

Ejemplo de creación de un procedimiento de nombre *SUBIDA* que sube 100 euros el salario de los empleados del departamento 30:

Ejemplo en Oracle:

```
CREATE OR REPLACE PROCEDURE SUBIDA AS
BEGIN
   UPDATE EMPLEADOS SET SALARIO = SALARIO +100 WHERE DEPT_NO=30;
   COMMIT;
END SUBIDA;
```

Ejemplo en MySQL:

```
DELIMITER //
CREATE PROCEDURE SUBIDA()
BEGIN
   UPDATE EMPLEADOS SET SALARIO = SALARIO + 100 WHERE DEPT_NO=30;
   COMMIT;
END;
//
```

El siguiente ejemplo muestra los procedimientos y funciones que tiene el esquema de nombre *ejemplo*:

```
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.printf("Nombre Procedimiento: %s - Tipo: %s %n",
                     proc_name, proc_type);
}
```

2.8.1. ResultSetMetaData

Se pueden obtener metadatos (datos sobre los datos) a partir de un objeto **ResultSet** mediante la interfaz **ResultSetMetaData**; es decir podemos obtener más información sobre los tipos y propiedades de las columnas de los objetos **ResultSet**, como por ejemplo, el número de columnas devueltas, el tipo, el nombre, etc. El siguiente ejemplo muestra el uso de la interfaz para conocer más información acerca de las columnas devueltas por esta consulta *SELECT * FROM departamentos*; en este caso al usar * en la *SELECT* desconocemos el nombre de las columnas devueltas.

Usaremos el método ***getMetadata()*** del objeto **ResultSet** que devuelve una referencia a un objeto **ResultSetMetaData** con el que se obtendrá la información acerca de las columnas devueltas:

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

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

      ResultSetMetaData rsmd = rs.getMetaData();

      int nColumnas = rsmd.getColumnCount();
      String nula;
      System.out.printf("Número de columnas recuperadas: %d%n",
                        nColumnas);
      for (int i = 1; i <= nColumnas; i++) {
        System.out.printf("Columna %d: %n ", i);
        System.out.printf("  Nombre: %s %n   Tipo: %s %n ",
             rsmd.getColumnName(i),  rsmd.getColumnTypeName(i));

        if (rsmd.isNullable(i) == 0)
            nula = "NO";
        else
            nula = "SI";

        System.out.printf("  Puede ser nula?: %s %n ", nula);

        System.out.printf("  Máximo ancho de la columna: %d %n",
                          rsmd.getColumnDisplaySize(i));
      }// for
      sentencia.close();
      rs.close();
      conexion.close();

     } catch (ClassNotFoundException cn) {
      cn.printStackTrace();
     } catch (SQLException e) {
      e.printStackTrace();
            }
  }// fin de main
}
```

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 usados son los siguientes:

Método	Descripción
`int getColumnCount()`	Devuelve el número de columnas devueltas por la consulta
`String getColumnName(int indiceColumna)`	Devuelve el nombre de la columna cuya posición se indica en *índiceColumna*
`String getColumnTypeName(int indiceColumna)`	Devuelve el nombre del tipo de dato específico del sistema de bases de datos que contiene la columna indicada en *índiceColumna*
`int isNullable(int indiceColumna)`	Devuelve 0 si la columna no puede contener valores nulos
`int getColumnDisplaySize(int indiceColumna)`	Devuelve el máximo ancho en caracteres de la columna indicada en *índiceColumna*
Más información sobre métodos de ResultSetMetaData: *https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSetMetaData.html*	

ACTIVIDAD 2.9

Visualiza información sobre las columnas de la tabla *empleados*.

2.9. EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

En ejemplos anteriores vimos como 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:

- ***ResultSet executeQuery(String)***: se utiliza para sentencias SQL que recuperan datos de un único objeto **ResultSet**, se utiliza para las sentencias SELECT.
- ***int 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.

- ***boolean execute(String)***: se puede utilizar para ejecutar cualquier sentencia SQL. Tanto para las que devuelven un **ResultSet** (por ejemplo, SELECT), como para las que devuelven el número de filas afectadas (por ejemplo, INSERT, UPDATE, DELETE) y para las de definición de datos como por ejemplo, CREATE. El método devuelve *true* si devuelve un **ResultSet** (para recuperar las filas será necesario llamar al método ***getResultSet()***) y *false* si se trata de un recuento de actualizaciones o no hay resultados; en este caso se usará el método ***getUpdateCount()*** para recuperar el valor devuelto. En este ejemplo ***execute()*** ejecuta una sentencia SELECT, devuelve *true*; por tanto, es necesario recuperar las filas devueltas usando el método ***getResultSet():***

```
import java.sql.*;
public class EjemploExecute {
      public static void main(String[] args) throws
         ClassNotFoundException, SQLException {
         //CONEXION A MYSQL
         Class.forName("com.mysql.jdbc.Driver");
         Connection conexion = DriverManager.getConnection
           ("jdbc:mysql://localhost/ejemplo","ejemplo","ejemplo");

         String sql="SELECT * FROM departamentos";
         Statement sentencia = conexion.createStatement();
         boolean valor = sentencia.execute(sql);

         if(valor){
            ResultSet rs = sentencia.getResultSet();
            while (rs.next())
                System.out.printf("%d, %s, %s %n",
                    rs.getInt(1), rs.getString(2), rs.getString(3));
            rs.close();
         } else {
            int f = sentencia.getUpdateCount();
            System.out.printf("Filas afectadas:%d %n", f);
         }
         sentencia.close();
         conexion.close();
      }//main
}//
```

Si cambiamos la orden SQL por esta otra: *String sql= "UPDATE departamentos SET dnombre = LOWER(dnombre)";* entonces la variable *valor* será *false* y la salida del programa será diferente.

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 *getXXX()* para la obtención de valores son los siguientes:

Método	Tipo Java devuelto
getString(int númerodecolumna) getString(String nombredecolumna)	String
getBoolean(int númerodecolumna) getBoolean(String nombredecolumna)	boolean
getByte(int númerodecolumna) getByte(String nombredecolumna)	byte
getShort(int númerodecolumna) getShort(String columna)	short
getInt(int númerodecolumna) getInt(String nombredecolumna)	int
getLong(int númerodecolumna) getLong(String nombredecolumna)	long
getFloat(int númerodecolumna) getFloat(String nombredecolumna)	float
getDouble(int númerodecolumna) getDouble(String nombredecolumna)	double
getBytes(int númerodecolumna) getBytes(String nombredecolumna)	byte[]
getDate(int númerodecolumna) getDate(String nombredecolumna)	Date
getTime(int númerodecolumna) getTime(String nombredecolumna)	Time
getTimestamp(int númerodecolumna) getTimestamp(String nombredecolumna)	Timestamp
Más información sobre métodos de ResultSet: *https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html*	

El siguiente ejemplo inserta un departamento en la tabla *departamentos*, los datos del nuevo departamento se introducen a través de los argumentos de *main()*. El primer parámetro es el departamento, el siguiente el nombre y el tercero la localidad. Antes de ejecutar la orden INSERT construimos la sentencia SQL en un *String*, las cadenas de caracteres (en este caso el nombre del departamento y la localidad) 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
        String sql = String.format
            ("INSERT INTO departamentos VALUES (%s, '%s', '%s')",
               dep, dnombre, loc);
```

```
        System.out.println(sql);

        Statement sentencia = conexion.createStatement();
        int filas = sentencia.executeUpdate(sql);
        System.out.printf("Filas afectadas: %d %n", 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 a través de los argumentos de *main()*:

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

//construir orden UPDATE
String sql = String.format
  ("UPDATE empleados SET salario = salario + %s WHERE dept_no = %s",
       subida, dep);
System.out.println(sql);

Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql);
System.out.printf("Empleados modificados: %d %n", 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
Empleados modificados: 3
```

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

```
//construir orden CREATE VIEW
StringBuilder sql = new StringBuilder();
sql.append("CREATE OR REPLACE VIEW totales ");
sql.append("(dep, dnombre, nemp, media) AS ");
sql.append("SELECT d.dept_no, dnombre, COUNT(emp_no), AVG(salario) ");
sql.append("FROM departamentos d LEFT JOIN empleados e " );
sql.append("ON e.dept_no = d.dept_no ");
sql.append("GROUP BY d.dept_no, dnombre ");
System.out.println(sql);

Statement sentencia = conexion.createStatement();
int filas = sentencia.executeUpdate(sql.toString());
System.out.printf("Resultado  de la ejecución: %d %n", filas);
```

La ejecución muestra la siguiente información:

```
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  de la ejecución: 0
```

ACTIVIDAD 2.10

Crea un programa Java que inserte un empleado en la tabla *empleados*, el programa recibe desde la línea de argumentos de *main()* 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.
- El APELLIDO y el OFICIO no pueden ser nulos.
- La fecha de alta del empleado es la fecha actual.

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

2.9.1. Ejecución de Scripts

Algunas bases de datos admiten la ejecución de varias sentencias DDL y/o DML en una misma cadena. Por ejemplo, cargar un script con la creación de tablas y los INSERT de las tablas desde un fichero a un *String* y hacer el ***executeUpdate()*** de ese *String*. Para ello es necesario indicarlo en la conexión añadiendo la propiedad ***allowMultiQueries=true*** de la siguiente manera:

```
Connection connmysql = DriverManager.getConnection
```

```
("jdbc:mysql://localhost/ejemplo?allowMultiQueries=true",
 "ejemplo", "ejemplo");
```

No todas las bases de datos admiten la ejecución de múltiples sentencias SQL, entre las que las admiten está MySQL. En el siguiente ejemplo se muestra la ejecución del siguiente script almacenado en el fichero *./script/scriptmysql.sql*, dentro del proyecto:

```
SET FOREIGN_KEY_CHECKS = 0;
drop table if EXISTS notas;
drop table if EXISTS alumnos;
drop table if EXISTS asignaturas;
CREATE TABLE IF NOT EXISTS ALUMNOS
(  DNI VARCHAR(10) NOT NULL primary key,
  APENOM VARCHAR(30),
  DIREC VARCHAR(30),
  POBLA  VARCHAR(15),
  TELEF  VARCHAR(10)  ) ;

CREATE TABLE IF NOT EXISTS ASIGNATURAS
(  COD int NOT NULL primary key,
  NOMBRE VARCHAR(25)) ;

CREATE TABLE IF NOT EXISTS NOTAS
(  DNI VARCHAR(10) NOT NULL ,
  COD int NOT NULL ,
  NOTA int,
  primary key(DNI,COD));

/* Create Foreign Keys */
ALTER TABLE NOTAS
      ADD CONSTRAINT FKNOTASALUM FOREIGN KEY (DNI)
      REFERENCES ALUMNOS (DNI)       ON UPDATE CASCADE
      ON DELETE RESTRICT;

ALTER TABLE NOTAS
      ADD CONSTRAINT FKNOTASASIG FOREIGN KEY (COD)
      REFERENCES ASIGNATURAS (COD)
      ON UPDATE CASCADE       ON DELETE RESTRICT;

/* Rellenar Datos */
INSERT IGNORE INTO ASIGNATURAS VALUES (1,'Prog. Leng. Estr.');
INSERT IGNORE INTO ASIGNATURAS VALUES (2,'Sist. Informáticos');
INSERT IGNORE INTO ASIGNATURAS VALUES (3,'Análisis');
INSERT IGNORE INTO ASIGNATURAS VALUES (4,'FOL');
INSERT IGNORE INTO ASIGNATURAS VALUES (5,'RET');

INSERT IGNORE INTO ALUMNOS VALUES ('12344345','Alcalde García, Elena',
'C/Las Matas, 24','Madrid','917766545');

INSERT  IGNORE  INTO  ALUMNOS  VALUES('4448242','Cerrato  Vela,  Luis',
'C/Mina 28 - 3A', 'Madrid','916566545');

INSERT  IGNORE INTO ALUMNOS VALUES('56882942','Díaz Fernández, María',
'C/Luis Vives 25', 'Móstoles','915577545');
```

```
INSERT IGNORE INTO NOTAS VALUES('12344345', 1,6);
INSERT IGNORE INTO NOTAS VALUES('12344345', 2,5);

INSERT IGNORE INTO NOTAS VALUES('4448242', 4,6);
INSERT IGNORE INTO NOTAS VALUES('4448242', 5,8);

INSERT IGNORE INTO NOTAS VALUES('56882942', 1,8);
INSERT IGNORE INTO NOTAS VALUES('56882942', 3,7);

commit;
```

El método lee el fichero de texto que contiene el script línea a línea, y lo almacena en un *StringBuilder*. Que después lo convierte a *String* para ejecutarlo con ***executeUpdate()*** :

```
public static void ejecutarScriptMySQL() {
  File scriptFile = new File("./script/scriptmysql.sql");
  System.out.println("\n\nFichero de consulta : " +
                     scriptFile.getName());
  System.out.println("Convirtiendo el fichero a cadena...");
  BufferedReader entrada = null;
  try {
       entrada = new BufferedReader(new FileReader(scriptFile));
      } catch (FileNotFoundException e) {
        System.out.println("ERROR NO HAY FILE: " + e.getMessage());
      }
  String linea = null;
  StringBuilder stringBuilder = new StringBuilder();
  String salto = System.getProperty("line.separator");
  try {
        while ((linea = entrada.readLine()) != null) {
            stringBuilder.append(linea);
            stringBuilder.append(salto);
            }
      } catch (IOException e) {
            System.out.println("ERROR de E/S, al operar " +
                   e.getMessage());
      }
  String consulta = stringBuilder.toString();
  System.out.println(consulta);
  try {
        Class.forName("com.mysql.jdbc.Driver");
      } catch (ClassNotFoundException e) {
            System.out.println("ERROR Driver:" + e.getMessage());
      }
  try {
        Connection connmysql = DriverManager.getConnection
        ("jdbc:mysql://localhost/ejemplo?allowMultiQueries=true",
            "ejemplo", "ejemplo");
        Statement sents = connmysql.createStatement();
        int res = sents.executeUpdate(consulta);
        System.out.println("Script creado con éxito, res = " + res);
        connmysql.close();
        sents.close();
      } catch (SQLException e) {
            System.out.println("ERROR AL EJECUTAR EL SCRIPT: "
```

```
                    + e.getMessage());
        }
}
```

2.9.2. 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* o 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 para insertar un departamento se representa 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 que en **Statement**: ***executeQuery()***, ***executeUpdate()*** y ***execute()*** pero no se necesita enviar la cadena de caracteres con la orden SQL en la llamada ya que lo hace el método ***prepareStatement(String)***. 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
```

Para asignar valor a cada uno de los marcadores de posición se utilizan los métodos ***setXXX()***. La sintaxis es la siguiente:

```
public abstract void setXXX(int indicedelParametro, tipoJava valor)
       throws SQLException
```

Donde, se asigna el valor indicado en *tipoJava* al parámetro cuyo índice coincide con *indicedelParametro*, que es transformado por el controlador JDBC en un tipo SQL correspondiente para pasarlo a la base de datos. Los métodos *setXXX()* son los siguientes:

Método	Tipo SQL
void setString(int índice, String valor)	VARCHAR
void setBoolean(int índice, boolean valor)	BIT
void setByte(int índice,byte valor)	TINYINT
void setShort(int índice, short valor)	SMALLINT
void setInt(int índice, int valor)	INTEGER
void setLong(int índice, long valor)	BIGINT
void setFloat(int índice, float valor)	FLOAT
void setDouble(int índice, double valor)	DOUBLE
void setBytes(int índice, byte[] valor)	VARBINARY
void setDate(int índice, Date valor)	DATE
void setTime(int índice, Time valor)	TIME
Más información sobre métodos de PreparedStatement: *https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html*	

Para asignar valores NULL a un parámetro se usa el método ***setNull()***, el formato es:

```
void setNull(int índice, int tipoSQL)
```

Donde *tipoSQL* es una constante que se define en la libreria **java.sql.Types**. Son las siguientes: *ARRAY, BIGINT, BINARY, BIT, BLOB, BOOLEAN, CHAR, CLOB, DATALINK, DATE, DECIMAL, DISTINCT, DOUBLE, FLOAT, INTEGER, JAVA_OBJECT, LONGNVARCHAR, LONGVARBINARY, LONGVARCHAR, NCHAR, NCLOB, NULL, NUMERIC, NVARCHAR, OTHER, REAL, REF, REF_CURSOR, ROWID, SMALLINT, SQLXML, STRUCT, TIME, TIME_WITH_TIMEZONE, TIMESTAMP, TIMESTAMP_WITH_TIMEZONE, TINYINT, VARBINARY, VARCHAR.*

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
```

En los ejemplos anteriores se ha usado ***Integer.parseInt(dep)*** y ***Float.parseFloat(subida)*** para convertir la cadena *dep* a un tipo entero y la cadena *subida* a un tipo 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 los argumentos de *main()* al ejecutar el programa desde la línea de comandos:

```
//recuperar parámetros de main
String dep = args[0];     //departamento
String oficio = args[1]; //oficio

//construimos la 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.printf("%s => %.2f %n", 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 visualizan los vendedores del departamento 30:

```
java VerEmpleado 30 VENDEDOR
ARROYO => 1500,00
MARTÍN => 1600,00
SALA => 1625,00
TOVAR => 1350,00
```

ACTIVIDAD 2.11

Utiliza la interfaz **PreparedStatement** para visualizar el APELLIDO, SALARIO y OFICIO de los empleados de un departamento cuyo valor se recibe desde los argumentos de *main()*. Visualiza también el nombre del 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. Utiliza la clase **DecimalFormat** para dar formato al salario. Ejemplo:

```
DecimalFormat formato = new DecimalFormat("##,##0.00");
String valorFormateado = formato.format(resul.getFloat(1));
```

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 salario 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 del departamento; si el departamento no existe devuelve como nombre *"INEXISTENTE"*:

```
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;
/
```

El siguiente ejemplo crea una función (en MySQL) de nombre *nombre_dep*, recibe un número de departamento (parámetro de entrada) y devuelve el nombre si existe; si no existe devuelve como nombre *"INEXISTENTE"*:

```
DELIMITER //
CREATE FUNCTION nombre_dep(d int) RETURNS VARCHAR(15)
BEGIN
   DECLARE nom VARCHAR(15);
   SET nom = 'INEXISTENTE';
   SELECT dnombre INTO nom FROM departamentos
   WHERE dept_no=d;
```

```
   RETURN nom;
 END;
//
```

Para ejecutarlo desde MySQL escribimos: *SELECT nombre_dep(10);*

A continuación se muestra un procedimiento (en MySQL) que recibe un número de departamento y devuelve en forma de parámetros de salida el nombre y la localidad (las funciones no pueden usar parámetros OUT), se asigna un valor inicial a los parámetros de salida por si el departamento no existe:

```
DELIMITER //
 CREATE PROCEDURE datos_dep
           (d int, OUT nom VARCHAR(15), OUT locali VARCHAR(15))
 BEGIN
   SET locali = 'INEXISTENTE';
   SET nom = 'INEXISTENTE';
   SELECT dnombre, loc INTO nom, locali FROM departamentos
   WHERE dept_no=d;
 END;
//
```

Para ejecutarlo desde MySQL escribimos las siguientes sentencias:

```
CALL datos_dep(10, @nom, @locali);
SELECT @nom;
SELECT @locali;
```

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**. En el *String* se declara la llamada al procedimiento o función, tiene dos formatos, uno incluye el parámetro de resultado (usado para las funciones) y el otro no:

```
{? = call <nombre_procedure>[(<arg1>,<arg2>, ...)]}
{call <nombre_procedure>[(<arg1>,<arg2>, ...)]}
```

Si los procedimientos y funciones incluyen parámetros de entrada o de salida es necesario indicarlos en forma de marcadores de posición. La referencia a los parámetros es secuencial, por número, el primer parámetro es el 1, el siguiente el 2, etc. El parámetro de resultado y los parámetros de salida deben ser registrados antes de realizar la llamada. El siguiente ejemplo declara la llamada al procedimiento *subida_sal* que tiene dos parámetros de entrada, se usan los marcadores de posición (?) para indicarlo:

```
String sql= "{ call subida_sal (?, ?) } ";
CallableStatement llamada = conexion.prepareCall(sql);
```

Hay 4 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 nombre_procedimiento }**: para un procedimiento almacenado sin parámetros.

- { ? = **call nombre_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 llamado parámetro de resultado.
- { **call nombre_procedimiento(?, ?, ...)** }: para un procedimiento almacenado que recibe parámetros.
- { ? = **call nombre_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 asignan a partir de los argumentos de *main()*:

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

          //recuperar parámetros de main
          String dep = args[0];     //departamento
          String subida = args[1]; //subida

          //construir orden de llamada
          String sql= "{ call subida_sal (?, ?) } ";

          //Preparar la llamada
          CallableStatement llamada = conexion.prepareCall(sql);

          //Dar valor a los argumentos
          llamada.setInt(1,Integer.parseInt(dep));       //primero
          llamada.setFloat(2,Float.parseFloat(subida)); // segundo

          //Ejecutar el procedimiento
          llamada.executeUpdate();
          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 al ejecutarlo puede que se muestre el siguiente error: *java.sql.SQLException: User does not have access to metadata required to determine stored procedure parameter types* ... si el usuario no tiene 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';*

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(int índice, int tipoSQL)***, el primer parámetro es la posición y el siguiente es una constante definida en la clase **java.sql.Types**. Estas constantes se nombraron en el apartado anterior. Por ejemplo, si el segundo parámetro de un procedimiento es OUT y de tipo VARCHAR en la base de datos en la llamada al método escribimos lo siguiente:

```
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()*** 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 los argumentos de *main()* 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 (?, ?) } "; // ORACLE

       //Preparar la llamada
       CallableStatement llamada = conexion.prepareCall(sql);

       //registrar parámetro de resultado
       llamada.registerOutParameter(1, Types.VARCHAR);//valor devuelto

       llamada.setInt(2,Integer.parseInt(dep));       //param de entrada

       //Registrar parámetro de salida
       llamada.registerOutParameter(3, Types.VARCHAR);//parámetro OUT

       //Ejecutar el procedimiento
       llamada.executeUpdate();
       System.out.printf("Nombre Dep: %s, Localidad: %s %n",
                       llamada.getString(1), 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 2.12

Crea una función en Oracle, que reciba un número de departamento y devuelva el salario medio de los empleados de ese departamento y como parámetro de salida el número de empleados. Si el departamento no existe debe devolver como salario medio el valor -1 y el número de empleados será 0. Si existe y no tiene empleados debe devolver 0. Realiza después un programa Java que use dicha función. El programa recorrerá la tabla *departamentos* y mostrará los datos del departamento, incluyendo el número de empleados y el salario medio.

Realiza un procedimiento en MySQL que funcione de forma similar a la función en Oracle, es decir, debe recibir un número de departamento y como parámetros de salida debe devolver el número de empleados y el salario medio. Realiza después un programa Java para usar dicho procedimiento, igual que antes el programa recorrerá la tabla *departamentos* y mostrará los datos del departamento, incluyendo el número de empleados y el salario medio.

La función y el procedimiento se crearán desde un programa Java.

2.11. INFORMES CON JASPERREPORTS

JasperReports es una herramienta para generar informes. De código abierto y licencia GPL (*Licencia Pública General*). Genera informes en distintos formatos: PDF, HTML, XLS, RTF, ODT, CSV, TXT y XML. Está escrita en Java y su principal objetivo es ayudar a crear documentos preparados para la impresión de una forma simple y flexible.

La página para descargarse la herramienta es: *http://community.jaspersoft.com/download*. Hay distintas versiones *Server*, *Library* y *Studio*. Y para saber más de JasperReports podemos acceder a *http://community.jaspersoft.com/wiki/jasperreports-library-tutorial*.

JasperReports organiza los datos recuperados de una fuente de datos de acuerdo con un informe de trazado definido en un fichero **JRXML**. Con el fin de llenar el informe con los datos, este informe de diseño (el fichero JRXML) debe ser compilado previamente.

En este capítulo solo nos interesa **generar informes utilizando la plantilla definida** en el fichero *JRXML*. Para nuestros proyectos necesitaremos añadir los JAR de JasperReports y el JAR **tools.jar** del JDK. En estas pruebas se utiliza la versión 6.2.0 de JasperReports. Así pues, añadiremos a los proyectos los siguientes JAR: *commons-beanutils-1.9.0.jar*, *commons-codec-1.5.jar*, *commons-collections-3.2.1.jar*, *commons-digester-2.1.jar*, *commons-javaflow-*

20060411.jar, commons-logging-1.1.1.jar, itext-2.1.7.js4.jar, jackson-annotations-2.1.4.jar, jackson-core-2.1.4.jar, jackson-databind-2.1.4.jar, jasperreports-6.2.0.jar, jasperreports-fonts-6.2.0.jar, jasperreports-javaflow-6.2.0.jar y *tools.jar*.

Ejemplo1: creamos un proyecto para generar un informe de datos de departamentos, de la base de datos MySQL. Utilizaremos las siguientes clases:

- net.sf.jasperreports.engine.***JasperCompileManager***,
- net.sf.jasperreports.engine.***JasperFillManager***,
- net.sf.jasperreports.engine.***JasperPrintManager***
- net.sf.jasperreports.engine.***JasperExportManager***

Para crear un informe con JasperReports seguiremos los siguientes pasos:

1. Generar el fichero **.jrxml**, será la plantilla en la que configuraremos como se desea el informe. En este fichero indicaremos los parámetros del informe, la consulta (SELECT) que se va a realizar, los datos que se van a visualizar, y además, se describirán cómo van a ser las líneas de cabecera, de detalle y de pies.
2. Ya dentro del proyecto Java, se compilará la plantilla, y obtendremos un objeto ***JasperReport*** de la siguiente manera:

```
JasperReport NombreJasperReport =
        JasperCompileManager.compileReport(MIPLANTILLA.JRXML);
```

3. Para rellenar de datos el informe se utiliza el método ***fillReport()*** de la clase **JasperFillManager** (***JasperFillManager.fillReport()***). Esto generará un fichero *.jrprint*. También se necesita el nombre del objeto *JasperReport*, creado anteriormente, los parámetros del informe y la conexión a la BD.

```
JasperPrint MiInforme = JasperFillManager.fillReport
   (NombreJasperReport, ParámetrosDelInforme, conexiónalaBD);
```

Los parámetros tienen que crearse y almacenarse en un *HashMap* (de **java.util**), ese *Map* se utiliza para crear el *JasperPrint* añadiendo parámetros. Por ejemplo, declaro 3 parámetros *titulo*, *autor* y *fecha* y los guardo en *params*:

```
Map<String, Object> params = new HashMap<String, Object>();
params.put("titulo", "LISTADO DE DEPARTAMENTOS.");
params.put("autor", "ARM");
params.put("fecha", (new java.util.Date()).toString());
```

4. Y finalmente podremos exportar el fichero *JasperPrint*, en el ejemplo *MiInforme*, generado anteriormente al formato que se desee. Por ejemplo:

   - Para visualizar en un visor, por consola, el informe generado escribiremos:

```
JasperViewer.viewReport(MiInforme);
```

Si se desea cerrar el visor sin cerrar la aplicación (por ejemplo, en una aplicación con ventanas) añadiremos *false*, es decir:

```
JasperViewer.viewReport(MiInforme, false);
```

   - Para generar el informe en HTML:

```
JasperExportManager.exportReportToHtmlFile
      (MiInforme, nombreFicheroHTML);
```

- Para generar el informe en PDF:

```
JasperExportManager.exportReportToPdfFile
        (MiInforme, nombreFicheroPDF);
```

- Para generar la salida en un documento XML:

```
//Convertir a XML,
//False es para indicar que no hay imágenes
//(isEmbeddingImages)
JasperExportManager.exportReportToXmlFile
          (MiInforme, nombreFicheroXML, false);
```

El código Java será el siguiente, la plantilla *.jrxml* se guarda en la carpeta *plantillas*, y los informes de salida en la carpeta *informes*.

```
import java.util.Map;

import com.mysql.jdbc.exceptions.jdbc4.CommunicationsException;
import com.mysql.jdbc.Connection;

import net.sf.jasperreports.engine.JRException;
import net.sf.jasperreports.engine.JasperCompileManager;
import net.sf.jasperreports.engine.JasperExportManager;
import net.sf.jasperreports.engine.JasperFillManager;
import net.sf.jasperreports.engine.JasperPrint;
import net.sf.jasperreports.engine.JasperReport;
import net.sf.jasperreports.view.JasperViewer;

import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.HashMap;

public class Principal {

  public static void main(String[] args) {
      String reportSource = "./plantilla/plantilla.jrxml";
      String reportHTML = "./informes/Informe.html";
      String reportPDF = "./informes/Informe.pdf";
      String reportXML = "./informes/Informe.xml";

      Map<String, Object> params = new HashMap<String, Object>();
      params.put("titulo", "LISTADO DE DEPARTAMENTOS.");
      params.put("autor", "ARM");
      params.put("fecha", (new java.util.Date()).toString());
      try {
            JasperReport jasperReport =
              JasperCompileManager.compileReport(reportSource);
            Class.forName("com.mysql.jdbc.Driver");
            Connection conn =(Connection) DriverManager.getConnection
              ("jdbc:mysql://localhost/ejemplo", "root", "");
            JasperPrint MiInforme =
               JasperFillManager.fillReport(jasperReport,params,conn);
            // Visualizar en pantalla
            JasperViewer.viewReport(MiInforme);
```

```
            // Convertir a HTML
            JasperExportManager.exportReportToHtmlFile(MiInforme,
                  reportHTML);
            // Convertir a PDF
            JasperExportManager.exportReportToPdfFile(MiInforme,
                  reportPDF);
            // Convertir a XML.
            JasperExportManager.exportReportToXmlFile
                   (MiInforme, reportXML, false);
            System.out.println("ARCHIVOS CREADOS");
         } catch (CommunicationsException c) {
             System.out.println(" Error de comunicación con la BD."+
                                " No está arrancada.");
         } catch (ClassNotFoundException e) {
             System.out.println(" Error driver. ");
         } catch (SQLException e) {
             System.out.println(" Error al ejecutar sentencia SQL ");
         } catch (JRException ex) {
                  System.out.println(" Error Jasper.");
                  ex.printStackTrace();    }
   }
}
```

2.11.1. El fichero .JRXML, la plantilla

Un informe de salida se va a estructurar en las siguientes secciones y el siguiente orden, estas secciones serán representadas en la plantilla:

Sección	Descripción
1. title	Su contenido se imprime solo una vez al comienzo del informe y como su nombre indica es el título que el informe tendrá
2. pageHeader	Esta es la cabecera de cada página, se imprimirá en cada página
3. columHeader	En esta zona se escribe la cabecera que vamos a poner para el detalle. Es decir, los nombres de las columnas que se visualizarán en el detalle (*detail*)
4. detail	Esta sección es el cuerpo del documento, es decir, donde se colocan la información a desplegar de nuestro informe (en nuestros ejercicios son columnas que devuelve la SELECT). En formato tabular
5. columFooter	En esta sección podremos poner los totales acumulados, y otras informaciones para cada una de las columnas del detalle
6. pageFooter	Este es el pie de página y se imprime al final de cada página. Útil para poner el contador de páginas, o alguna otra información
7. summary	Esto se utiliza para concluir el documento y se imprime una sola vez al final del informe

Para explicar el contenido de un fichero ***jrxml***, lo vemos con el estudio de la plantilla del ejercicio. La siguiente plantilla crea un informe para visualizar los datos de la tabla *departamentos* de la base de datos *ejemplo* de MySQL. El informe a visualizar es el siguiente, véase la Figura 2.15:

LISTADO DE DEPARTAMENTOS.

Realizado por: ARM on Wed Apr 13 03:22:55 CEST 2016

Codigo depart	Nombre departamento	Localidad departamento
10	CONTABILIDAD	SEVILLA
20	INVESTIGACION	MADRID
30	VENTAS	BARCELONA
40	PRODUCCIÓN	BILBAO
61	MARKETING	GUADALAJARA

Total Registros: 5

Pagina 1 of 1

Figura 2.15. Informe del ejercicio

La plantilla debe empezar con la etiqueta ***<jasperReport>***, este es el elemento raíz, debe ir acompañado del *namespace* de *JasperReports* y del nombre del informe. También en esta etiqueta se especificarán las características del documento, por ejemplo, el ancho de la página (*pageWidth*), el alto (*pageHeight*), ancho de columna (*columnWidth*), los márgenes izquierdo, derecho, superior o inferior (*leftMargin*, *rightMargin*, *topMargin*, *bottomMargin*), o también la orientación, por ejemplo, para escribir en apaisado pondremos *orientation="Landscape"*.

Por ejemplo:

```
<jasperReport
  xmlns="http://jasperreports.sourceforge.net/jasperreports"
  name="Listadodepartamentos" pageWidth="595" pageHeight="600"
  columnWidth="555" leftMargin="20" rightMargin="20" topMargin="30"
  bottomMargin="30" >
```

Si se desea que el informe sea apaisado escribiremos en la etiqueta ***<jasperReport>***:

```
<jasperReport name="ejemplo" orientation="Landscape" pageWidth="842">
```

En el ejemplo el fichero se va a llamar *plantilla.jrxml,* las partes de esta plantilla son las siguientes:

- **Declaración del documento**, se indica el espacio de nombres que se van a utilizar y el nombre del documento. Es la raíz del documento (al crearlo en Eclipse se añadirán de forma automática varios *namespaces*). En *xmlns* se indica el namespace de JasperReports, para calificar las etiquetas y atributos que pertenecen a este lenguaje:

```
<jasperReport
  xmlns="http://jasperreports.sourceforge.net/jasperreports"
  name="Listadodepartamentos">
```

- **Declaración de los parámetros**, se indica el tipo de dato de los parámetros. Los nombres son los que se ponen en el programa Java, dentro del *HashMap*.

```
<parameter name="titulo" class="java.lang.String" />
<parameter name="autor" class="java.lang.String" />
<parameter name="fecha" class="java.lang.String" />
```

- **Definición de la consulta <queryString>**, y de las columnas de la consulta ***<field name ..../>***. Acompañando a la columna se indica el tipo de dato en *class*. Si la consulta tiene campos calculados, contadores, medias, importes, etc., es necesario poner un alias al cálculo para luego referenciar la columna en el ***field name***. En el ejercicio se visualizan el código de departamento, el nombre y la localidad:

```
<queryString>
      <![CDATA[SELECT * FROM departamentos]]>
</queryString>
<field name="dept_no" class="java.lang.Integer"/>
<field name="dnombre" class="java.lang.String"/>
<field name="loc" class="java.lang.String"/>
```

  Para las columnas tipo *Date* pondremos el class **java.util.Date** y para las tipo *float* **java.lang.Float**.

- **Líneas de título del informe**, etiqueta **<title>**. El título solo se visualiza al inicio del informe. En el ejercicio el título definido en esta sección se muestra en la Figura 2.16:

LISTADO DE DEPARTAMENTOS.

Realizado por: ARM on Wed Apr 13 01:54:02 CEST 2016

Figura 2.16. Líneas de título del informe del ejercicio.

  Dentro del título se añade una etiqueta **<band>** para indicar la altura del título. Y dentro de ella se añaden dos etiquetas ***textField***, una para visualizar el título y otra para los valores de los parámetros.

  Dentro de estas se indica la posición con ***reportElement***. Dentro de la etiqueta ***textElement*** podremos indicar la alineación o la fuente. Y el contenido a visualizar se escribe en la etiqueta ***textFieldExpression***, para ello se utiliza la etiqueta ***<![CDATA[información]]>***. Donde indicaremos lo que se desea visualizar. Para hacer referencia a los parámetros escribimos el prefijo ***$P*** seguido del nombre del parámetro entre llaves, por ejemplo: ***$P{autor}***. Las posiciones de los elementos, el ancho y el alto, se miden en pixel.

  El código para el título es el siguiente:

```
<title>
  <band height="60"> <!-- Se indica el alto del título -->
      <textField>
            <reportElement x="0" y="10" width="500" height="40" />
            <textElement textAlignment="Center"><font size="24"/>
            </textElement>
            <textFieldExpression><![CDATA[$P{titulo}]]>
              </textFieldExpression>
      </textField>
      <textField>
```

```
        <reportElement x="0" y="40" width="500" height="20" />
        <textElement textAlignment="Center"/>
        <textFieldExpression><![CDATA["Realizado por: " + $P{autor}
              + " on " + $P{fecha}]]></textFieldExpression>
    </textField>
  </band>
</title>
```

- **Cabecera del informe,** etiqueta **<columnHeader>**. Como en el caso anterior el contenido se encierra entre la etiqueta ***band***, donde indicamos la altura. Se utiliza la etiqueta ***<rectangle>*** para encerrar la cabecera en un rectángulo, dentro se indica la posición, el ancho y el alto del rectángulo. Para cada literal a visualizar se añade la etiqueta ***<staticText>***, y dentro de ella se indica la posición x e y, el ancho y el alto del texto a visualizar con ***<reportElement>***. Y el contenido del texto a visualizar se escribe dentro de la etiqueta ***<text>*** y utilizando la etiqueta ***<![CDATA[información]]>***. En la Figura 2.17 se muestra como queda la cabecera.

Código depart	Nombre departamento	Localidad departamento

Figura 2.17. Cabecera del informe del ejercicio.

El código para esta sección es el siguiente.

```
<columnHeader>
  <band height="30">
    <rectangle>
      <reportElement x="0" y="0" width="500" height="25" />
    </rectangle>
    <staticText>
      <reportElement x="5" y="5" width="100" height="15" />
      <text><![CDATA[Código depart]]></text>
    </staticText>
    <staticText>
      <reportElement x="105" y="5" width="150" height="15" />
      <text><![CDATA[Nombre departamento]]></text>
    </staticText>
    <staticText>
      <reportElement x="255" y="5" width="150" height="15" />
      <text><![CDATA[Localidad departamento]]></text>
    </staticText>
  </band>
</columnHeader>
```

- **Línea de detalle del informe,** etiqueta **<detail>**. Como en el caso anterior el contenido se encierra entre la etiqueta ***band,*** para indicar la altura de cada línea de detalle. Para cada columna a visualizar se añade una etiqueta ***<textField>***, y dentro de ella se indica la posición x e y, el ancho y el alto del texto a visualizar con ***<reportElement>***. Y el contenido del texto a visualizar se escribe dentro de la etiqueta ***<textFieldExpression>*** indicando el tipo de dato y utilizando la etiqueta ***<![CDATA[información]]>***. En la Figura 2.18 se muestra cómo queda el detalle,

10	CONTABILIDAD	SEVILLA
20	INVESTIGACIÓN	MADRID
30	VENTAS	BARCELONA
40	PRODUCCIÓN	BILBAO
61	MARKETING	GUADALAJARA

Figura 2.18. Detalle del informe del ejercicio.

Observa que las columnas de la SELECT llevan el prefijo ***$F{Nombre_columna}***. El código para esta sección es el siguiente.

```
<detail>
  <band height="30">
      <textField>
            <reportElement x="35" y="7" width="100" height="15" />
            <textFieldExpression><![CDATA[$F{dept_no}]]>
                  </textFieldExpression>
      </textField>
      <textField>
            <reportElement x="105" y="7" width="150" height="15" />
            <textFieldExpression><![CDATA[$F{dnombre}]]>
                  </textFieldExpression>
      </textField>
      <textField>
            <reportElement x="255" y="7" width="150" height="15" />
            <textFieldExpression><![CDATA[$F{loc}]]>
                  </textFieldExpression>
      </textField>
  </band>
</detail>
```

- **Línea de pie del informe,** etiqueta **<pageFooter>**. En esta sección se añade una línea con color utilizando la etiqueta **<line>.** Para añadir el número de página actual se utiliza una etiqueta ***<textField>,*** y dentro de ella en la etiqueta ***<textFieldExpression>*** se indica la variable ***$V{PAGE_NUMBER}*** para obtener el número de página actual. Para obtener el total de páginas se utiliza también la misma variable, pero en el ***textField***, se indica cuando se calcula el contador, a nivel de reporte se indica así: ***<textField evaluationTime="Report">.*** En la Figura 2.19 se muestra como queda el pie:

Figura 2.19. Línea de pie.

El código para esta sección es el siguiente.

```
<pageFooter>
  <band height="32">
    <line>
        <reportElement positionType="FixRelativeToBottom" x="0"
              y="3" width="500" height="1" />
        <graphicElement>
            <pen lineWidth="2.0" lineColor="#FF0000"/>
        </graphicElement>
    </line>
    <textField>
        <reportElement x="390" y="10" width="90" height="20" />
        <textElement textAlignment="Right">
                  <font isBold="true" isItalic="true"/>
        </textElement>
        <textFieldExpression>
            <![CDATA["Página "+ $V{PAGE_NUMBER} +" of"]]>
        </textFieldExpression>
    </textField>
    <textField evaluationTime="Report">
       <reportElement x="480" y="10" width="40" height="20" />
       <textElement><font isBold="true" isItalic="true"/>
       </textElement>
       <textFieldExpression>
              <![CDATA[" " + $V{PAGE_NUMBER}]]>
       </textFieldExpression>
    </textField>
  </band>
</pageFooter>
```

- **Línea de sumario**, etiqueta **<summary>**. Se visualiza una vez al final del informe. En el ejercicio se obtiene el número de registros visualizados. Para obtener el número de registros de utiliza la variable ***$V{REPORT_COUNT}***. Las variables llevan el prefijo *$V*. La salida se muestra en la Figura 2.20.

Total Registros: 5

Figura 2.20. Línea de sumario.

El código para esta sección es el siguiente.

```
<summary>
  <band height="60">
    <rectangle>
          <reportElement x="0" y="0" width="500" height="25" />
    </rectangle>
    <textField>
       <reportElement x="10" y="5" width="300" height="15" />
       <textElement textAlignment="Left"/>
       <textFieldExpression> <![CDATA["Total Registros: "
                     +String.valueOf($V{REPORT_COUNT})]]>
       </textFieldExpression>
    </textField>
  </band>
</summary>
```

El código completo para la plantilla es el siguiente:

```
<jasperReport
  xmlns="http://jasperreports.sourceforge.net/jasperreports"
  name="Listadodepartamentos" >
  <parameter name="titulo" class="java.lang.String"/>
  <parameter name="autor" class="java.lang.String"/>
  <parameter name="fecha" class="java.lang.String"/>
  <queryString>
          <![CDATA[SELECT * FROM departamentos]]>
  </queryString>
  <field name="dept_no" class="java.lang.Integer"/>
  <field name="dnombre" class="java.lang.String"/>
  <field name="loc" class="java.lang.String"/>

  <title>
     <band height="60">
       <textField>
            <reportElement x="0" y="10" width="500" height="40" />
            <textElement textAlignment="Center"><font size="24"/>
            </textElement>
            <textFieldExpression><![CDATA[$P{titulo}]]>
            </textFieldExpression>
       </textField>
       <textField>
            <reportElement x="0" y="40" width="500" height="20" />
            <textElement textAlignment="Center"/>
            <textFieldExpression><![CDATA["Realizado por: " +
              $P{autor} +" on "+$P{fecha}]]></textFieldExpression>
         </textField>
     </band>
  </title>

  <columnHeader>
     <band height="30">
            <rectangle>
                  <reportElement x="0" y="0" width="500" height="25" />
            </rectangle>
            <staticText>
                  <reportElement x="5" y="5" width="100" height="15" />
                        <text><![CDATA[Código depart]]></text>
            </staticText>
            <staticText>
               <reportElement x="105" y="5" width="150" height="15" />
               <text><![CDATA[Nombre departamento]]></text>
            </staticText>
            <staticText>
               <reportElement x="255" y="5" width="150" height="15" />
               <text><![CDATA[Localidad departamento]]></text>
            </staticText>
     </band>
  </columnHeader>

  <detail>
     <band height="30">
        <textField>
```

```
            <reportElement x="35" y="7" width="100" height="15" />
            <textFieldExpression>
                  <![CDATA[$F{dept_no}]]></textFieldExpression>
        </textField>
        <textField>
            <reportElement x="105" y="7" width="150" height="15" />
            <textFieldExpression><![CDATA[$F{dnombre}]]>
                  </textFieldExpression>
        </textField>
        <textField>
            <reportElement x="255" y="7" width="150" height="15" />
            <textFieldExpression><![CDATA[$F{loc}]]>
                  </textFieldExpression>
        </textField>
    </band>
</detail>

<pageFooter>
    <band height="32">
      <line>
            <reportElement positionType="FixRelativeToBottom" x="0"
                  y="3" width="500" height="1" />
            <graphicElement>
                  <pen lineWidth="2.0" lineColor="#FF0000"/>
            </graphicElement>
      </line>
      <textField>
            <reportElement x="390" y="10" width="90" height="20" />
            <textElement textAlignment="Right">
                        <font isBold="true" isItalic="true"/>
            </textElement>
            <textFieldExpression><![CDATA["Página "+$V{PAGE_NUMBER}+
                  " de"]]></textFieldExpression>
      </textField>
      <textField evaluationTime="Report">
            <reportElement x="480" y="10" width="40" height="20" />
            <textElement>
                  <font isBold="true" isItalic="true"/>
            </textElement>
            <textFieldExpression><![CDATA[" " + $V{PAGE_NUMBER}]]>
              </textFieldExpression>
    </textField>
  </band>
</pageFooter>

<summary>
    <band height="60">
      <rectangle>
            <reportElement x="0" y="0" width="500" height="25" />
      </rectangle>
      <textField>
            <reportElement x="10" y="5" width="300" height="15" />
            <textElement textAlignment="Left"/>
            <textFieldExpression><![CDATA["Total Registros: "
              +String.valueOf($V{REPORT_COUNT})]]>
```

```
            </textFieldExpression>
        </textField>
      </band>
  </summary>
</jasperReport>
```

ACTIVIDAD 2.13

Sobre el mismo proyecto, crea una nueva plantilla para obtener por cada departamento, además de sus datos, el número de empleados que hay, la media de salario y la suma de salario. Si el departamento no tiene empleados debe de salir 0 en la media, el contador y la suma. Cambia las propiedades para que el documento se visualice en apaisado.

Figura 2.21. Informe propuesto.

También se pueden **añadir variables acumuladas.** Para añadir totales acumulados es necesario crear las variables. Se crean debajo de los campos de la consulta (*field name*). Y luego esas variables se pueden añadir o en el sumario en las líneas de pie. Por ejemplo, aquí defino dos totales, uno para sumar la cantidad y otro para contar artículos (*cantidad*, e *idart* son *field name*). En ***calculation*** se escribe la función de grupo:

```
<variable name="sumacant" class="java.lang.Integer" calculation="Sum">
      <variableExpression><![CDATA[$F{cantidad}]]></variableExpression>
</variable>
<variable name="numart" class="java.lang.Integer" calculation="Count">
      <variableExpression><![CDATA[$F{idart}]]></variableExpression>
</variable>
```

Las funciones a añadir en ***calculation*** pueden ser: *sum*, *count*, *max*, *min*, *average*, *DistinctCount*, entre otras funciones. Dentro del informe estas variables se añadirán dentro de las secciones ***<summary>***, o en ***<columnFooter>***, las escribimos dentro de un ***textField***. Por ejemplo:

```
<textField>
  <reportElement x="620" y="0" width="50" height="20"/>
  <textFieldExpression><![CDATA[$V{sumacant}]]></textFieldExpression>
</textField>
<textField>
```

```
<reportElement x="670" y="0" width="50" height="20"/>
<textFieldExpression><![CDATA[$V{numart}]]></textFieldExpression>
</textField>
```

2.12. GESTIÓN DE ERRORES

Hasta ahora en todos los ejemplos cuando se producía un error se visualizaba con el método *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:

```
java.sql.SQLSyntaxErrorException: ORA-00942: la tabla o vista no existe

      at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:439)
      at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:395)
      at oracle.jdbc.driver.T4C8Oall.processError(T4C8Oall.java:802)
      at oracle.jdbc.driver.T4CTTIfun.receive(T4CTTIfun.java:436)
      at oracle.jdbc.driver.T4CTTIfun.doRPC(T4CTTIfun.java:186)
      at oracle.jdbc.driver.T4C8Oall.doOALL(T4C8Oall.java:521)
      at oracle.jdbc.driver.T4CStatement.doOall8(T4CStatement.java:194)
      at
oracle.jdbc.driver.T4CStatement.executeForRows(T4CStatement.java:1000)
      at
oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout(OracleStatement
.java:1307)
      at
oracle.jdbc.driver.OracleStatement.executeUpdateInternal(OracleStatemen
t.java:1814)
      at
oracle.jdbc.driver.OracleStatement.executeUpdate(OracleStatement.java:1
779)
      at
oracle.jdbc.driver.OracleStatementWrapper.executeUpdate(OracleStatement
Wrapper.java:277)
      at InsertarDep.main(InsertarDep.java:38)
```

Cuando se produce un error con **SQLException** podemos acceder a cierta información usando los siguientes métodos:

Método	Función
`int getErrorCode()`	Devuelve un entero que proporciona el código de error del fabricante. Normalmente, será el código de error real devuelto por la base de datos.
`String getSQLState()`	Devuelve una cadena que contiene un estado definido por el estándar X/OPEN SQL.
`String getMessage()`	Devuelve una cadena que describe el error. Es un método heredado de la clase **java.lang.Throwable**.

A continuación se utilizan esos métodos para visualizar los mensajes de error:

```
try
{
```

```
    //Código
}
catch (ClassNotFoundException cn) {cn.printStackTrace();}
catch (SQLException e)
{
   System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:%n");
   System.out.printf("Mensaje   : %s %n", e.getMessage());
   System.out.printf("SQL estado: %s %n", e.getSQLState());
   System.out.printf("Cód error : %s %n", 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.departamento' 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
```

Cuando se intenta insertar una fila en una tabla cuya clave primaria ya existe en MYSQL, se muestra la siguiente información:

```
Mensaje   : Duplicate entry '10' for key 'PRIMARY'
SQL estado: 23000
Cód error : 1062
```

Y en Oracle:

```
Mensaje   : ORA-00001: restricción única (EJEMPLO.PK_DEP) violada
SQL estado: 23000
Cód error : 1
```

Cuando se inserta una clave ajena en una tabla y no existe su correspondiente clave primaria en la otra tabla, en MYSQL se muestra la siguiente información:

```
Mensaje   : Cannot add or update a child row: a foreign key constraint
fails   (`ejemplo`.`empleados`,   CONSTRAINT   `FK_DEP`   FOREIGN   KEY
(`dept_no`) REFERENCES `departamentos` (`dept_no`))
SQL estado: 23000
Cód error : 1452
```

En ORACLE:

```
Mensaje     : ORA-02291: restricción de integridad (EJEMPLO.FK_EMP)
violada - clave principal no encontrada
SQL estado: 23000
Cód error : 2291
```

COMPRUEBA TU APRENDIZAJE

1. 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-3.8.11.2.jar
Apache Derby	
HSQLDB	
H2	
MySQL	
ORACLE	
Db4o	

2. 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	

3. 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.

4. ¿El siguiente código Java es correcto? Razona la respuesta:

```
import java.sql.*;
public class Ejercicio4 {
  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.printf("%d, %s %n",
            resul.getInt("EMP_NO"), resul.getString("APELLIDO"));
    }
    resul.close();
    sentencia.close();
    conexion.close();
  }
}
```

5. 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. DESCRIPCION varchar(50), no nulo. STOCKACTUAL numérico. STOCKMINIMO numérico. PVP numérico.	ID numérico, clave primaria. NOMBRE varchar(50), no nulo. DIRECCION varchar(50). POBLACION varchar(50). 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.

6. 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, etc.)

Ejecuta el programa e inserta varias ventas en las distintas bases de datos.

7. Se pretende realizar un listado de las ventas de un cliente. El programa recibe dos parámetros desde los argumentos de *main()*, 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 la siguiente información:

Ventas del cliente: *Nombre de cliente*
Venta: *idventa* , Fecha venta: *fecha*
Producto: *descripción del producto*
Cantidad: *cantidad* PVP: *pvp*
Importe: *cantidad * pvp*
Venta: *idventa*, Fecha venta: *fecha*
Producto: *descripción del producto*
Cantidad: *cantidad* PVP: *pvp*
Importe: *cantidad * pvp*
.....
Número total de ventas: ___
Importe Total: ___

8. Realiza un programa Java con pantalla gráfica que nos permita consultar los datos de la tabla de *departamentos* (en MySQL). La pantalla mostrará los campos de la tabla departamentos y 4 botones. El botón *Primero* muestra el primer departamento, el botón *Siguiente* muestra el siguiente departamento al mostrado actualmente. El botón *Anterior* muestra el departamento anterior al que se está mostrando. El botón *Último* muestra el último departamento de la tabla. El primer departamento es aquel cuyo número de

departamento es el menor. El último departamento es aquel con mayor número de departamento. Mostrar los posibles errores que puedan surgir, por ejemplo, cuando pulsamos el botón *Anterior* y estamos en el primer registro, o cuando pulsamos el botón Siguiente y estamos en el último registro. Inicialmente se debe mostrar el primer departamento. La pantalla es la siguiente:

Figura 2.22. Ejercicio 8.

9. Realiza un programa con pantalla gráfica para llevar a cabo la gestión de la tabla *empleados* (en MySQL). La aplicación nos debe permitir consultar, insertar, modificar y eliminar datos en la tabla. A la hora de insertar, el departamento y el director se eligen desde una lista. Al visualizar los datos de un empleado se debe mostrar en las listas su director y su departamento correspondiente. A la hora de insertar es obligatorio que todos los campos tengan valor, excepto la comisión que si no se da valor se le asigna valor 0. Por defecto, para la fecha de alta se asume la fecha del sistema. Se muestran varios botones:

Figura 2.23. Pantalla inicial del Ejercicio 9.

- El botón *CONSULTAR EMPLEADO* muestra los datos del empleado cuyo número se ha introducido. Si no se introduce ningún número no se mostrará nada. Si se introduce un número de empleado y no existe se debe mostrar mensaje indicándolo.
- El botón *INSERTAR* inserta los datos en la tabla siempre y cuando se hayan introducido todos los valores en los campos, se debe controlar que el número de empleado no exista, si existe se debe visualizar un mensaje indicándolo. Cuando

se inserta un empleado se debe añadir a la lista de directores. La Figura 2.24 muestra la inserción de un empleado.

- El botón *ELIMINAR* elimina el empleado que se muestra en pantalla. NO se podrá eliminar un empleado que es director de otros. Se debe mostrar mensaje indicándolo. Cuando se elimina un empleado se debe eliminar de la lista de directores.
- El botón *MODIFICAR* modifica los datos del empleado. Mostrar mensaje si la modificación se ha realizado correctamente. Véase Figura 2.25.
- El botón *LIMPIAR* limpia los campos de la pantalla, se deben mostrar como en la Figura 2.23. El botón *SALIR* finaliza la ejecución.
- Después de las operaciones de insertar, modificar o eliminar se debe mostrar mensaje de cómo se ha realizado la operación y se debe limpiar la pantalla. Se debe controlar el tamaño de los campos de entrada, número de empleado 4 dígitos, oficio y apellido tendrán el número de caracteres definidos en las columnas de la tabla, etc. Controlar todos los posibles errores que puedan surgir visualizando mensajes indicando lo que ocurre.

Figura 2.24. Ejercicio 9, inserción de un empleado.

Figura 2.25. Ejercicio 9, modificación de un empleado.

10. Realiza un programa con pantalla gráfica para crear las tablas PRODUCTOS, CLIENTES y VENTAS dentro de la base de datos ACCESS *ejemplo.accdb*. Las tablas tendrán el mismo formato que las del apartado 5. Para los CREATE TABLE podemos utilizar los siguientes tipos de datos: *Integer* para enteros, *Float* para decimales, *Datetime* para la fecha, *Text(tamaño)* para la cadenas, *Long* para enteros largos, entre otros.

    Para las claves primarias añadimos ***PRIMARY KEY*** a la columna que es la clave primaria, y para las claves ajenas añadimos:
    REFERENCES nombreTabla(nombreColumna) a la columna que es clave ajena.

    La pantalla debe ser similar a la que se muestra en la Figura 2.26.

Figura 2.26. Ejercicio 10. Ventana de la actividad.

- En el textArea visualizaremos los mensajes de lo que vaya ocurriendo. Si se han creados las tablas, si no se han podido crear. Los registros que se van añadiendo a la BD Access, si ya existen indicad que ya existen. Para las operaciones de crear las tablas en la BD Access capturaremos la excepción **UcanaccessSQLException**, y visualizaremos en el textArea el código de error y el mensaje de error de la excepción. Nos interesa detectar si la tabla ya se ha creado o no.
- Al pulsar el botón *Llenar Tablas con Datos MySQL*, hay que leer las tablas correspondientes en MySQL y guardar los registros en las tablas correspondientes de Access. Capturar la excepción **SQLException** para detectar si al insertar en las tablas Access el registro ya se ha insertado y salta el error de clave duplicada. Visualizar en el textArea el código de error y el mensaje de error de la excepción.
- Al pulsar el botón *Informe Ventas* se debe visualizar un informe de todas las ventas, los datos a visualizar son: *Id Venta, Fecha Venta, Id Cliente, Nombre Cliente, Población Cliente, Id de Producto, Descripción de Producto, Precio, Cantidad* e *Importe* (que será el PVP*CANTIDAD). Ajustar el informe para que se visualicen todos los campos. Véase la Figura 2.27.

RESUMEN DATOS DE VENTAS.

Realizado por ARM on Thu Apr 21 01:02:00 CEST 2016

Id Venta	Fecha	Id	Nombre Cliente	Poblacion Cliente	Id	Descripcion Producto	Precio	Cantidad	Importe
1	2012-07-16	1	MARIA SERRANO	Guadalajara	4	Diccionario Maria Moliner	43	3	129.0
2	2012-07-17	4	ALICIA PEREZ	Talavera	5	Impresora HP Deskjet	30	2	61.3
3	2012-07-19	2	PEDRO BRAVO	Guadalajara	5	Impresora HP Deskjet	30	1	30.65
4	2012-08-20	1	MARIA SERRANO	Guadalajara	6	Pen Drive 8 Gigas	7	5	35.0
5	2012-08-22	3	MANUEL SERRA	Guadalajara	4	Diccionario Maria Moliner	43	1	43.0
Total Registros: 5									

Página 1 de 1

Figura 2.27. Ejercicio 10. Informe de ventas.

- Al pulsar el botón *Informe De Clientes* se debe visualizar un informe de clientes con estos datos: Id Cliente, Nombre Cliente, Población Cliente, Número de Ventas (que será el contador de ventas del cliente), Total Importe (la suma de los importes, pvp*cantidad, de las ventas del cliente).

Podemos **crear la BD Access** (si no la tenemos creada) con el siguiente método, el método recibe el nombre de la base de datos, con la extensión. La versión asignada es la 2007. Se crea la BD añadiendo ***newDatabaseVersion=V2007***:

```
public static boolean crearbd(String nombre){
    String url = UcanaccessDriver.URL_PREFIX +
          nombre+";newDatabaseVersion=V2007";
    try {
          Class.forName("net.ucanaccess.jdbc.UcanaccessDriver");
    } catch (ClassNotFoundException e) {
          e.printStackTrace();
          return false;
    }
    try {
          Connection conn = DriverManager.getConnection(url);
          conn.close();
    } catch (SQLException e) {
          e.printStackTrace();
          return false;
    }
    System.out.println("Base de datos "+ nombre + " creada");
    return true;
```

11. Crea un proyecto nuevo y utilizando JAXB estudiado en el Capítulo 1, crea las clases necesarias para obtener un documento XML, con los datos de departamentos y empleados de la BD MySQL. El documento debe presentar los datos de los departamentos y los empleados de cada departamento. Haz un método también, para visualizar el contenido del fichero XML creado. El fichero XML debe tener la siguiente estructura:

```
<Datosempledepart>
    <Departamento>
        <depno> </depno>
        <dnombre></dnombre>
        <loc></loc>
        <Empleados>
              <Emple>
                 <empno> </empno>
                 <apellido> </apellido>
                 <salario> </salario>
                 <oficio> </oficio>
              </Emple>
        </Empleados>
    <Departamento>
. . . . . . . . . . .
<Datosempledepart>
```

12. Crea un proyecto nuevo y realiza un método que proporcione la siguiente información sobre la BD *ejercicio12.db*, creada con SQLITE, que se encuentra en la carpeta de recursos del capítulo. La información a mostrar es la siguiente:

```
INFORMACIÓN SOBRE LA BASE DE DATOS:
===================================
Nombre :
Driver :
URL    :

INFORMACIÓN SOBRE LAS TABLAS (Solo las tablas), Para cada tabla
visualizar:
=====================================================================
Nombre :
Clave(s) Primaria(s):
Clave(s) ajena(s)(importadas):
Clave(s) exportadas(s):
Columnas: nombre, tipo y si puede ser nula.
Número de registros que tiene:
```

Realiza cada apartado dentro de un bloque **try-catch**, capturando la excepción **SQLException**, y visualiza el mensaje de error que devuelve la excepción.

13. Ahora se desea crear un documento XML similar al creado en el Ejercicio 11. En este caso se pide crear el XSD que describa el documento XML. Una vez creado el XSD se crea una clase Java para generar el fichero *empleydepar.xml* y cargar los datos de empleados y departamentos de MySQL. El fichero XSD debe estar formado por tres tipos: ***DatosempledepartType***, que será el elemento raíz y este estará compuesto por el elemento ***Departamentos***. Este elemento recogerá los datos de todos los departamentos, y cada departamento estará compuesto por *depno*, *dnombre*, *loc*, y el tipo ***Empleados***. Datos del tipo *Empleados* serán *empno*, *apellido*, *oficio* y *salario*.

ACTIVIDADES DE AMPLIACIÓN

11. Realiza la modificación de los departamentos de la tabla de *departamentos* (en MySQL) a partir de los datos contenidos en un fichero de texto. El fichero de texto contiene una línea por cada departamento a modificar, los campos del departamento estarán separados por comas. El formato es el siguiente: *departamento a modificar, nuevo nombre de departamento, nueva localidad.* El primer campo tiene que ser numérico. Ejemplo de fichero que contiene los datos a modificar:

```
10,INFORMÁTICA,
20, NÓMINAS, SEVILLA
40,  , SANTANDER
40,,SANTANDER
10, INFORMÁTICA
30,,
40
50 ,
```

La primera línea indica que se quiere modificar el departamento 10, el nuevo nombre de departamento es INFORMÁTICA. No se modifica la localidad. La quinta línea indica lo mismo.

La segunda línea indica que se quiere modificar el departamento 20, el nuevo nombre de departamento es NÓMINAS y la localidad es SEVILLA.

La tercera y la cuarta línea indican que se quiere modificar el departamento 40, no se modifica el nombre del departamento, se modifica la localidad que es SANTANDER.

Las tres últimas líneas indican que no se modifica el departamento correspondiente. Y así sucesivamente.

Controlar los posibles errores: departamento incorrecto, no hay datos para modificar, el departamento a modificar no existe, etc.

CAPÍTULO 3

HERRAMIENTAS DE MAPEO OBJETO-RELACIONAL (ORM)

OBJETIVOS

Instalar y configurar una herramienta ORM.

Interpretar y 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

Concepto de mapeo objeto-relacional.

Herramientas objeto-relacional.

Ficheros de mapeo. Elementos, propiedades.

Clases persistentes.

Sesiones; estados de un objeto.

Carga, almacenamiento y modificación de objetos.

Consultas SQL.

Lenguajes propios de la herramienta ORM.

RESUMEN

En este capítulo aprenderemos a instalar una herramienta de mapeo que nos permitirá crear una capa de acceso a los datos de una base de datos relacional, de tal forma que las tablas se transformarán en clases y las filas de las tablas serán objetos. Realizaremos programas Java que accederán a la base de datos relacional usando orientación a objetos.

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 de las tablas (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, funcionan en cualquier BD.
- 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, donde se observan varias capas. Entre la capa de Hibernate y la de base de datos se muestran diferentes APIs Java que usan Hibernate para interactuar con la base de datos.

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 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 los 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 (en este caso se han hecho las pruebas en la versión Mars). Para los ejemplos vamos a utilizar la base de datos MySQL de nombre *ejemplo*, creada en la Capítulo 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, la clave ajena en la tabla *empleados* debe crearse dando nombre a la restricción:

```
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,
  CONSTRAINT fkdep FOREIGN KEY (dept_no)
                 REFERENCES departamentos (dept_no)
) ENGINE=InnoDB;
```

Figura 3.5. Base de datos *ejemplo*.

3.5.1. Instalación del plugin

Para instalar el plugin de Hibernate en 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 **JBoss Data Services Development** y seleccionamos **Hibernate Tools**. 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 en *Next*. A continuación aceptamos la licencia y pulsamos el botón *Finish*. El proceso de instalación comienza, puede tardar un rato. Durante la instalación nos pide confirmación para continuar, ya que el plugin contiene software sin firmar.

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, Figura 3.8.

Figura 3.8. Comprobar la instalación de Hibernate

También se puede comprobar desde el menú **Window-> Perspective -> Open Perspective -> Other-> Hibernate**, véase Figura 3.8

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 el capítulo anterior ya lo usamos, debemos tenerlo localizado en alguna carpeta.

A continuación desde el menú: **Window -> Preferences -> Data Management -> Connectivity-> Driver Definitions** se pulsa el botón *Add*:

- Desde la pestaña **Name/Type** se selecciona de la lista *Vendor Filter* la opción *MySQL*.Y a continuación en la lista de drivers seleccionamos *MySQL JDBC Driver* en la versión 5.1, véase Figura 3.9

Figura 3.9. Comprobar la instalación de Hibernate

- Desde la pestaña **JAR List** pulsamos el botón *Add JAR/Zip* para localizar el conector **mysql-connector-java-5.1.38-bin.jar**, se debe buscar la carpeta donde tengamos dicho conector. Si se visualiza el driver **mysql-connector-java-5.1.0-bin.jar** lo eliminamos 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 Eclipse (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 Path-> 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 de la lista la opción *MySQL JDBC Driver* y pulsamos el botón *Finish*, véase Figura 3.11.

Figura 3.11. Agregar la librería MySQL JDBC al proyecto

El proyecto debe mostrar un aspecto similar al de la Figura 3.12.

Figura 3.12. Proyecto con la librería MySQL.

3.5.3. Configuración de Hibernate

Una vez que tenemos la librería MySQL en nuestro proyecto hemos de crear el fichero de configuración de Hibernate llamado **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 donde 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 3.14.

Figura 3.14.Datos para la configuración del fichero *hibernate.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 la opción: *jdbc:mysql://<hostname>/<database>* y se cambia por *jdbc:mysql://localhost/ejemplo*.
- ***Username***: usuario que se conectará a la base de datos *ejemplo*, en este caso el nombre se usuario es *ejemplo*.
- ***Password***: clave del usuario que realiza la conexión con MySQL, en este caso es *ejemplo*.

La casilla ***Create a console configuration*** crea el fichero **XML Hibernate Console Configuration** con el mismo nombre que el proyecto Eclipse, podemos marcarla, pulsar el botón *Next* y finalizar o seguir el paso siguiente para crear el fichero.

Para terminar se pulsa el botón *Finish*. Se visualiza el editor de configuración de Hibernate, véase Figura 3.15.

Figura 3.15.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</propert
y>
        <property
name="hibernate.connection.password">ejemplo</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost/ejemplo</propert
y>
        <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 3.16.

Figura 3.16. Crear *Hibernate Console Configuration.*

Se abre una nueva ventana, en el campo ***Name*** escribimos un nombre para nuestra configuración, por ejemplo *Proyecto1ConsoleConfiguration*. Nos aseguramos que en el campo ***Project*** aparezca nuestro proyecto y en el campo ***Configuration file*** aparezca 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 sobre el proyecto y seleccionamos: **New-> Other->Hibernate-> Hibernate Reverse**

Engineering File(reveng.xml). Pulsamos el botón *Next* y nos pide que indiquemos donde 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 3.17.

Figura 3.17.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 *Proyecto1ConsoleConfiguration*, y pulsamos el botón *Refresh* para que muestre la base de datos *ejemplo* y sus tablas, véase Figura 3.18. Seleccionamos una a una o todas las tablas y pulsamos el botón *Include*. Para finalizar pulsamos el botón *Finish*.

Figura 3.18. Tablas a mapear con Hibernate.

Se visualiza el editor de *Hibernate Reverse Engineering*, véase Figura 3.19.

Figura 3.19. Editor *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 3.1

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 de nombre EJEMPLO creado en la Capítulo 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 11. (Si de la lista *Available Driver Definitions* no aparece driver para Oracle pulsamos en el botón *New Driver Definition* localizado al lado de la lista).

- Desde la pestaña *JAR List* pulsamos el botón *Add JAR/Zip* para localizar el conector **ojdbc6.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 (versión Oracle Express)
- *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 3.20.

Figura 3.20. Botón *Run As...*

Desde la ventana que aparece hacemos doble clic en la opción: Hibernate Code Generation 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 3.21 (el resto se dejan los valores por defecto):

- ***Name***: escribimos un nombre para esta configuración, por ejemplo, *ConfiguracionProyecto1*.
- ***Console configuration***: seleccionamos de la lista *Proyecto1ConsoleConfiguration*.
- ***Output directory***: debe ser la carpeta *src*, pulsando en el botón *Browse* localizamos la carpeta *\Proyecto1\src*.
- ***Package***: escribimos el nombre del paquete donde se crearán las clases, por ejemplo, *primero*.
- ***reveng.xml***: localizamos el fichero **reveng.xml** creado anteriormente. Pulsando en el botón *Setup* se puede localizar.

Figura 3.21.Pestaña *Main*.

Desde la pestaña **Exporters** se indica los ficheros que queremos generar, se marcan las casillas: *Use Java 5 syntax*, *Domain code*, *Hibernate XML Mappings* e *Hibernate XML Configuration*. Una vez seleccionadas se pulsa el botón *Apply* y posteriormente se pulsa *Run*, Figura 3.22.

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 métodos *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, véase Figura 3.23.

Figura 3.22. Pestaña *Exporters*.

Figura 3.23. Proyecto1 con las clases Java y ficheros *hbm.xml* generados.

Para cada clase se generan una serie de atributos que tienen que ver con las columnas de la tabla que mapean y las relaciones de claves ajenas, varios constructores; y los métodos *getters* y *setters*. En la clase *Departamentos* se observan los siguientes atributos:

```
private byte deptNo;
private String dnombre;
private String loc;
private Set<Empleados> empleadoses = new HashSet<Empleados>(0);
```

Los atributos *deptNo, dnombre* y *loc* se corresponden con las columnas de la tabla. El atributo *empleadoses* define la relación de clave ajena entre las tablas *empleados* y *departamentos*. Este atributo sirve para almacenar los empleados del departamento.

En la clase *Empleados* se observan los siguientes atributos:

```
private short empNo;
private Departamentos departamentos;
private String apellido;
private String oficio;
private Short dir;
private Date fechaAlt;
private Float salario;
private Float comision;
```

El atributo de número de departamento (columna DEPT_NO de la tabla) no aparece, en su lugar aparece un atributo de nombre *departamentos* que es un objeto de la clase *Departamentos* y que hace referencia al departamento del empleado. Recordemos que la columna DEPT_NO de la tabla *empleados* es la clave ajena que referencia a la tabla *departamentos*, al mapear dicha columna se genera un atributo del tipo *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-> Perspective-> Open Perspective-> Other-> Hibernate**. Desde la pestaña **Hibernate Configurations** pulsamos en la configuración de nombre *Proyecto1ConsoleConfiguration* con el botón derecho del ratón y seleccionamos ***HQL Editor***, véase Figura 3.24.

Figura 3.24. Pestaña *Hibernate Configurations*.

Se abre una nueva pestaña con el mismo nombre que la configuración de Hibernate. Desde aquí podemos escribir sentencias HQL para consultar la base de datos. A continuación escribimos el siguiente código HQL para consultar los empleados: *from Empleados*, y pulsamos el botón con la flechita verde ▷ para ejecutar la consulta, véase Figura 3.25. 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 3.25. Ejecución de una consulta en HQL.

Desde este entorno también se pueden realizar consultas al estilo SQL, respetando los nombres de las clases y de los atributos de las mismas, 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. *Las siguientes SELECT son también erróneas porque no* respetan los nombres de las clases y los atributos:

```
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
```

Para comentar líneas usamos doble guion --. Al final del capítulo encontrarás un resumen de sentencias en HQL.

ACTIVIDAD 3.2

Realiza consultas HQL con las tablas mapeadas. Prueba estas consultas:

from Empleados as e where e.departamentos.deptNo = 10

from Departamentos where deptNo = 10

from Departamentos as d join d.empleadoses

from Departamentos as d left outer join d.empleadoses

select avg(salario), count(empNo) from Empleados where departamentos.deptNo = 10

select apellido, salario from Empleados as e where e.departamentos.deptNo =10

select departamentos.dnombre, avg(salario), count(empNo) from Empleados group by departamentos.deptNo

Desde la perspectiva Hibernate, pulsamos con el botón derecho del ratón en ***Configuration*** y seleccionamos ***Mapping Diagram***, se muestra el diagrama de mapeo entre las clases Java y las tablas de la base de datos, véase Figura 3.26.

Figura 3.26. Diagrama de mapeo.

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. Desde la URL *http://hibernate.org/orm/downloads/* descargamos la última distribución estable de Hibernate. En este caso se ha bajado la versión: **hibernate-release-5.1.0.Final.zip**.
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. Desde la carpeta *D:\eclipse\Hibernate\hibernate-release-5.1.0.Final\lib\required* seleccionamos todos los ficheros y los copiamos a nuestra carpeta *D:\eclipse\Hibernate*.
4. Descargamos la última versión de la librería **slf4j** desde la URL *http://www.slf4j.org/download.html*, se ha descargado el fichero **slf4j-1.7.21.zip** (en Linux **slf4j-1.7.21.tar.gz**), lo descomprimimos y localizamos los ficheros **slf4j-api-1.7.21.jar** y **slf4j-simple-1.7.21.jar** para copiarlos en la carpeta *D:\eclipse\Hibernate*.
5. Los ficheros JAR que deben contener la carpeta son los que se muestran en la Figura 3.27.

6. Desde Eclipse hacemos clic con el botón derecho del ratón en nuestro proyecto y pulsamos en **Build Path –> Add Libreries**. Se visualiza una nueva ventana desde la que elegimos **User Library**. En este paso crearemos una librería con todos los JAR que necesita Hibernate, pulsamos el botón *Next* (Figura 3.28).

Figura 3.27.Ficheros JAR necesarios.

Figura 3.28.Añadir librería de usuario.

7. 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 3.29.

Figura 3.29.Nombrar la librería con los jar para Hibernate.

8. Seguidamente pulsamos el botón *Add External JARs* para localizar los ficheros JAR de nuestra carpeta *Hibernate*, los seleccionamos todos y pulsamos el botón *Abrir*. Se mostrará una pantalla similar a la mostrada en la Figura 3.30. Pulsamos el botón *OK* y a continuación el botón *Finish* para finalizar. En nuestro proyecto aparecerá la nueva librería

Con esto ya podemos crear el primer programa Java en nuestro proyecto que nos va a permitir comunicarnos con nuestra base de datos.

Figura 3.30.Libreria *HibernateLib* con los JAR y en el proyecto Eclipse.

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**.

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 (así tenemos un único objeto creado de una clase).

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 pueda ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

Nuestro **Singleton** será una clase de ayuda que accede a **SessionFactory** para obtener un objeto sesión, hay una única **SessionFactory** para toda la aplicación. En la clase se define una variable estática llamada *sessionFactory* que recoge el objeto **SessionFactory** devuelto por el método ***buildSessionFactory()***; este objeto se crea a partir del fichero de configuración (**hibernate.cfg.xml**). El método ***getSessionFactory()*** devuelve el valor de la variable estática definida, o lo que es lo mismo, devuelve el objeto **SessionFactory** creado. El nombre de la clase es *HibernateUtil.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.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
  private static final SessionFactory sessionFactory = buildSessionFactory();
```

```
  private static SessionFactory buildSessionFactory() {
     try {
        //Create the SessionFactory from hibernate.cfg.xml
        return new Configuration().configure().buildSessionFactory(
                new StandardServiceRegistryBuilder().configure().build() );
     }
     catch (Throwable ex) {
        // Make sure you log the exception, as it might be swallowed
        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, o en otro paquete que definamos. El método *main()* inserta una fila en la tabla *departamentos*. El código es el siguiente:

```
import primero.*;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

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

    //obtener la sesión actual
    SessionFactory sesion = HibernateUtil.getSessionFactory();
    //crear la sesión
    Session session = sesion.openSession();
    //crear una transacción de la sesión
    Transaction tx = session.beginTransaction();

    System.out.println("Inserto una fila en la tabla DEPARTAMENTOS.");

    Departamentos dep = new Departamentos();
    dep.setDeptNo((byte) 62);
    dep.setDnombre("MARKETING");
    dep.setLoc("GUADALAJARA");

    session.save(dep);
    tx.commit();
    session.close();

    System.exit(0);
  }
}
```

Con la siguiente línea *SessionFactory sesion = HibernateUtil.getSessionFactory();* se obtiene 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 name ="Miconexion">
```

Por la siguiente, donde se elimina el parámetro *name*:

```
<session-factory>
```

Ya que si no se mostrará el siguiente error en la ejecución:

```
WARN: HHH000277: Could not bind factory to JNDI
org.hibernate.engine.jndi.JndiException: Error parsing JNDI name
[Miconexion]
```

Durante la ejecución también aparecerán errores warnings del tipo: *WARN: HHH90000012: Recognized obsolete hibernate namespace http://hibernate.sourceforge.net/hibernate-configuration. Use namespace http://www.hibernate.org/dtd/hibernate-configuration instead. Support for obsolete DTD/XSD namespaces may be removed at any time.* Para eliminar dicho error hemos de cambiar las cabeceras DOCTYPE de los ficheros XML, se deben actualizar las URL de los namespaces. Cambiamos en el fichero **hibernate.cfg.xml** la línea:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

Por esta otra:

```
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

En los ficheros de mapeo **Empleados.hbm.xml** y **Departamentos.hbm.xml** cambiamos la URL por la siguiente: *http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd.* Y en el fichero **hibernate.reveng.xml** escribimos la siguiente: *http://www.hibernate.org/dtd/hibernate-reverse-engineering-3.0.dtd.*

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** (o pulsar F5). La ejecución del programa mostraría en consola la imagen de la Figura 3.31.

```
Console
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (21 de abr. de 2016 17:38:56)
abr 21, 2016 5:38:57 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.1.0.Final}
abr 21, 2016 5:38:57 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
abr 21, 2016 5:38:57 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
abr 21, 2016 5:38:58 PM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for production use!)
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost/ejemplo]
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=ejemplo, password=****}
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
abr 21, 2016 5:39:02 PM org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
abr 21, 2016 5:39:04 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
Inserto una fila en la tabla DEPARTAMENTOS.
```

Figura 3.31.Consola de la ejecución del programa.

El siguiente ejemplo inserta un empleado en la tabla *empleados* en el departamento 10, para el departamento será necesario crear un objeto de tipo *Departamentos* y asignar como número de departamento el valor 10, es lo que se hace en el método *setDeptNo()* de esta clase:

```
import primero.*;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class MainEmpleado {
  public static void main(String[] args) {
    //obtener la sesión actual
    SessionFactory sesion = HibernateUtil.getSessionFactory();
    //crear la sesión
    Session session = sesion.openSession();
    //crear una transacción de la sesión
    Transaction tx = session.beginTransaction();

    System.out.println("Inserto un EMPLEADO EN EL DEPARTAMENTO 10.");

    Float salario = new Float(1500);// inicializo el salario
    Float comision = new Float(10); // inicializo la comisión

    Empleados em = new Empleados(); // creo un objeto empleados

    em.setEmpNo((short) 4455);  // el número de empleado es 4455
    em.setApellido("PEPE");     // el nombre es PEPE
    em.setDir((short) 7499);    // el director es el empleado 7499
    em.setOficio("VENDEDOR");   // el oficio es VENDEDOR

    em.setSalario(salario);
    em.setComision(comision);

    //se crea un objeto Departamentos para asignárselo al empleado
    Departamentos d = new Departamentos();
    d.setDeptNo((byte) 10); //el número de dep es 10
    em.setDepartamentos(d);

    //fecha de alta, calculamos fecha actual
```

```
    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();
    System.exit(0);
  }
}
```

En los ejemplos anteriores hemos usado los siguientes métodos:

- ***save()***: Este método de la sesión (interface **Session**) lo usaremos para guardar el objeto, le pasamos como argumento el objeto a guardar: ***save(Object object)***.
- ***commit()***: Este método hace un commit de la transacción actual. La transacción se crea al método ***beginTransaction()*** de la sesión actual. Es necesario para que los datos se almacenen en la BD.
- ***close()***: Este método se utiliza para cerrar la sesión.

¡ ¡ INTERESANTE ! !

Documentación Java sobre Hibernate: *https://docs.jboss.org/hibernate/orm/current/javadocs/*

Referencia en español de Hibernate: *https://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/*

Documentación sobre las distintas versiones de Hibernate incluyendo la actual: *https://docs.jboss.org/hibernate/orm/*

Si los ejemplos anteriores los ejecutamos más de una vez se producirán excepciones de error, ya que el departamento a insertar o el empleado a insertar existe. La excepción es la siguiente **org.hibernate.exception.ConstraintViolationException**, y se produce al hacer el commit en el método ***tx.commit()***. También se producirá error si al insertar un empleado creamos un objeto departamento que no exista, entonces, se produce la siguiente excepción: **org.hibernate.TransientPropertyValueException** esta vez sobre el método ***session.save(em)***. El siguiente código controlaría las excepciones de la existencia de un empleado y de la no existencia del departamento en el programa de inserción de un empleado:

```
try {
      session.save(em);
      try {
            tx.commit();
      } catch (ConstraintViolationException e) {
            System.out.println("EMPLEADO DUPLICADO");
            System.out.printf("MENSAJE: %s%n", e.getMessage());
            System.out.printf("COD ERROR: %d%n", e.getErrorCode());
            System.out.printf("ERROR SQL: %s%n",
                              e.getSQLException().getMessage());
      }
} catch (TransientPropertyValueException e) {
      System.out.println("EL DEPARTAMENTO NO EXISTE");
      System.out.printf("MENSAJE: %s%n", e.getMessage());
      System.out.printf("Propiedad: %s%n", e.getPropertyName());
```

```
} catch (Exception e) {
      System.out.println("ERROR NO CONTROLADO....");
      e.printStackTrace();
}
```

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 3.32 muestra la correspondencia entre los ficheros de mapeo y las clases generadas.

Figura 3.32. 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://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 21-abr-2016 9:48:48 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>
        <set name="empleadoses" table="empleados" inverse="true"
                                lazy="true" fetch="select">
            <key>
                <column name="dept_no" not-null="true" />
            </key>
            <one-to-many class="primero.Empleados" />
        </set>
    </class>
</hibernate-mapping>
```

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 define 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>
```

La etiqueta **set** se utiliza para mapear colecciones. Dentro de **set** se definen varios atributos. En ***name*** se indica el nombre del atributo generado, el nombre de la tabla de donde se tomará la colección se declara con el atributo ***table***. En el elemento ***key*** se define el nombre de la columna identificadora en la asociación, en este caso la columna *dept_no* de la tabla *departamentos*. El elemento **one-to-many** define la relación, en este caso es una asociación ***uno-a-muchos***, es decir, un departamento puede tener muchos empleados. En ***class*** se indica de qué tipo son los elementos de la colección. Resumiendo, este 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" not-null="true" />
    </key>
    <one-to-many class="primero.Empleados" />
</set>
```

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 de 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://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<!-- Generated 21-abr-2016 9:48:48 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>
        <many-to-one name="departamentos" class="primero.Departamentos"
                     fetch="select">
            <column name="dept_no" not-null="true" />
        </many-to-one>
        <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>
    </class>
</hibernate-mapping>
```

En este fichero 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); se utiliza para definir una relación muchos a uno entre las dos clases Java (es decir, muchos empleados pertenecen a un departamento). En el atributo ***name*** se indica el nombre del atributo en la clase Java y en ***class*** se indica la clase a la que referencia. En el elemento ***column name*** se indica el nombre de la columna de la tabla *empleados* (en este caso es *dept_no*). 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" 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.

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**, son las clases que implementan las entidades del problema, deben implementar la interfaz **Serializable**. 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 qué 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:

```
package primero;
// Generated 21-abr-2016 9:48:48 by Hibernate Tools 3.4.0.CR1

import java.util.HashSet;
import java.util.Set;

/**
 * Departamentos generated by hbm2java
 */
public class Departamentos implements java.io.Serializable {

      private byte deptNo;
      private String dnombre;
      private String loc;
      private Set<Empleados> empleadoses = new HashSet<Empleados>(0);

      public Departamentos() {
      }

      public Departamentos(byte deptNo) {
            this.deptNo = deptNo;
      }

      public Departamentos(byte deptNo, String dnombre,
                           String loc, Set<Empleados> empleadoses) {
            this.deptNo = deptNo;
            this.dnombre = dnombre;
            this.loc = loc;
            this.empleadoses = empleadoses;
      }

      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;
        }

        public Set<Empleados> getEmpleadoses() {
                return this.empleadoses;
        }

        public void setEmpleadoses(Set<Empleados> empleadoses) {
                this.empleadoses = empleadoses; `
        }
}
```

ACTIVIDAD 3.3

Realiza el mapeo de las tablas de PRODUCTOS, CLIENTES y VENTAS del capítulo anterior y estudia los ficheros de mapeo y las clases generadas.

3.8. SESIONES Y OBJETOS HIBERNATE

Para 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 SessionFactory
SessionFactory sessionFactory = cfg.buildSessionFactory(
        new StandardServiceRegistryBuilder().configure().build() );

// 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. Se necesita crear un objeto del tipo **StandardServiceRegistry** que contiene la lista de servicios que utiliza Hibernate para crear el ejemplar de **SessionFactory**; este 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 *HibernateUtil.java*, para crear una vez el ejemplar de **SessionFactory**.

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**. Al crear la sesión se crea la transacción para dicha sesión. 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 = sesion.openSession();       //crea la sesión
Transaction tx = session.beginTransaction(); //crea 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 (Transient)**: 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 (Persistent)**. Un objeto estará en este estado cuando ya está almacenado en la base de datos. 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. En definitiva, 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 persistentes se caracterizan por haber sido ya creados y almacenados en una sesión o bien devueltos en una consulta realizada con la sesión.

- **Separado (Detached)**. Un objeto está en este estado cuando cerramos la sesión mediante el método ***close()*** de **Session**. Una instancia separada es un objeto que se ha hecho persistente, pero su sesión 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 asociada a una nueva **Session** más tarde, haciéndola persistente de nuevo (con todas las modificaciones).

Fuente: *http://docs.jboss.org/hibernate/core/3.5/reference/es-ES/html/objectstate.html*

3.8.3. Carga de objetos

Para la carga de objetos usaremos los siguientes métodos de **Session**:

MÉTODO	DESCRIPCIÓN
`<T> T load(Class<T> Clase, Serializable id)`	Devuelve la **instancia persistente** de la clase indicada con el identificador dado. La instancia tiene que existir, si no existe el método lanza una excepción
`Object load(String nombreClase, Serializable id)`	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase en formato de *String*
`<T> T get(Class<T> Clase, Serializable id)`	Devuelve la **instancia persistente** de la clase indicada con el identificador dado. Si la instancia no existe, devuelve *null*
`Object get(String nombreClase, Serializable id)`	Similar al método anterior, pero en este caso indicamos en el primer parámetro el nombre de la clase

El siguiente ejemplo utiliza el método ***load()*** para obtener los datos del departamento 20. En el primer parámetro se indica la clase *Departamentos* y en el segundo el número de departamento que se quiere recuperar, se hace un *cast* para convertirlo al tipo de dato definido en el atributo identificador de la clase (*deptNo*) que es *byte*. Si la fila no existe se lanza la excepción **ObjectNotFoundException**:

```
// Visualiza los datos del departamento 20
Departamentos dep = new Departamentos();
try {
   dep = (Departamentos) session.load(Departamentos.class, (byte) 20);
   System.out.printf("Nombre Dep: %s%n", dep.getDnombre());
   System.out.printf("Localidad: %s%n", dep.getLoc());
} catch (ObjectNotFoundException o) {
   System.out.println("NO EXISTE EL DEPARTAMENTO!!");
}
```

Usando el segundo formato del método ***load()*** la obtención del departamento quedaría así:

```
dep = (Departamentos) session.load("primero.Departamentos", (byte) 20);
```

El método ***load()*** lanza la excepción **ObjectNotFoundException** 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 *null* si no existe la fila correspondiente, el siguiente ejemplo comprueba si el departamento 11 existe. Si existe visualiza sus datos, y si no existe visualiza un mensaje indicándolo:

```
Departamentos dep = (Departamentos)
                    session.get(Departamentos.class, (byte) 11);
if (dep==null) {
    System.out.println("El departamento no existe");
}
else
{
    System.out.printf("Nombre Dep: %s%n",dep.getDnombre());
    System.out.printf("Localidad: %s%n",dep.getLoc());
}
```

ACTIVIDAD 3.4

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, para obtener los empleados usamos el método ***getEmpleadoses()*** de la clase *Departamentos* y usamos un **Iterator** para recorrer la lista de empleados.

```
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 = HibernateUtil.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.");

      // obtenemos empleados
      Set<Empleados> listaemple = dep.getEmpleadoses();
      Iterator<Empleados> it = listaemple.iterator();

      System.out.printf("Número de empleados: %d %n",
                        listaemple.size());
      while (it.hasNext()) {
            Empleados emple = it.next();
            System.out.printf("%s * %.2f %n",
                        emple.getApellido(), emple.getSalario());
      }
```

```
        System.out.println("==============================");
        session.close();
        System.exit(0);
    }
}
```

La ejecución muestra la siguiente salida:

```
==============================
DATOS DEL DEPARTAMENTO 10.
Nombre Dep:CONTABILIDAD
Localidad:SEVILLA
==============================
EMPLEADOS DEL DEPARTAMENTO 10.
Número de empleados: 4
REY * 4100,00
MUÑOZ * 1690,00
PEPE * 1500,00
CEREZO * 2885,00
==============================
```

¡ ¡ INTERESANTE ! !

Consulta la API de Hibernate: *https://docs.jboss.org/hibernate/orm/current/javadocs/*

ACTIVIDAD 3.5

A partir de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS obtén un listado de ventas de un cliente. El cliente se obtendrá como un argumento de *main()*. La información a visualizar es similar a la mostrada en el Ejercicio 7 del Capítulo anterior.

3.8.4. Almacenamiento, modificación y borrado de objetos

Para almacenamiento, modificación y borrado de objetos usamos los siguientes métodos **Session**:

MÉTODO	DESCRIPCIÓN
`Serializable save(Object obj)`	Guarda el objeto que se pasa como argumento en la base de datos. Hace que la instancia transitoria del objeto sea persistente.
`void update(Object objeto)`	Actualiza en la base de datos el objeto que se pasa como argumento. El objeto a modificar debe ser cargado con el método ***load()*** o ***get()***
`void delete(Object objeto)`	Elimina de la base de datos el objeto que se pasa como argumento. El objeto a eliminar debe ser cargado con el método ***load()*** o ***get()***

El siguiente ejemplo crea un nuevo objeto *Departamentos* y lo almacena en la base de datos usando el método ***save()***. 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()*** o el método ***get()*** y a continuación podemos borrarlo con el método ***delete()***, hemos de asegurarnos de que no existan referencias de otros objetos al que se va a borrar, de lo contrario se producirá una excepción. 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
```

El siguiente ejemplo muestra la eliminación de un departamento controlando distintas excepciones, que el departamento no exista y que tenga empleados. La ejecución mostrará el mensaje: *NO SE PUEDE ELIMINAR, TIENE EMPLEADOS*:

```
import org.hibernate.ObjectNotFoundException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.exception.ConstraintViolationException;

import primero.Departamentos;
import primero.HibernateUtil;

public class BorradoDep {
  public static void main(String[] args) {
     SessionFactory sesion = HibernateUtil.getSessionFactory();
     Session session = sesion.openSession();
     Transaction tx = session.beginTransaction();

     //DEPARTAMENTO A ELIMINAR
     Departamentos de = (Departamentos)
                        session.load(Departamentos.class, (byte) 10);
     try {
        session.delete(de); // elimina el objeto
        tx.commit();
        System.out.println("Departamento eliminado...");
     } catch (ObjectNotFoundException o) {
        System.out.println("NO EXISTE EL DEPARTAMENTO...");
     } catch (ConstraintViolationException c) {
        System.out.println("NO SE PUEDE ELIMINAR, TIENE EMPLEADOS");
     }catch (Exception e) {
        System.out.println("ERROR NO CONTROLADO....");
        e.printStackTrace();
     }
     session.close();
     System.exit(0);
  }
}
```

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()*** para modificarlo. El siguiente ejemplo modifica el salario y el departamento del empleado 7369, sumamos 1000 al salario y le asignamos el departamento 30:

```
Empleados em = new Empleados();
try {
      em = (Empleados) session.load(Empleados.class, (short) 7369);
      System.out.printf("Modificación empleado: %d%n", em.getEmpNo());
      System.out.printf("Salario antiguo: %.2f%n", em.getSalario());
      System.out.printf("Departamento antiguo: %s%n",
                        em.getDepartamentos().getDnombre());
      //nuevo salario
      float NuevoSalario = em.getSalario() + 1000;
      em.setSalario(NuevoSalario);

      //nuevo departamento
      Departamentos dep = (Departamentos)
                        session.get(Departamentos.class, (byte) 30);
      if (dep == null) {
            System.out.println("El departamento NO existe");
      } else {
            em.setDepartamentos(dep);
            session.update(em); // modifica el objeto
            tx.commit();
            System.out.printf("Salario nuevo: %.2f%n",em.getSalario());
            System.out.printf("Departamento nuevo: %d%n",
                              em.getDepartamentos().getDnombre());
      }

} catch (ObjectNotFoundException o) {
      System.out.println("NO EXISTE EL EMPLEADO...");
} catch (ConstraintViolationException c) {
      System.out.println("NO SE PUEDE ASIGNAR UN DEPARTAMENTO
                         QUE NO EXISTE.....");
} catch (Exception e) {
      System.out.println("ERROR NO CONTROLADO....");
      e.printStackTrace();
}
```

ACTIVIDAD 3.6

Sube el salario a todos los empleados del departamento 10. El salario se recibe como argumento de *main()*. Muestra el apellido y salario del empleado antes y después de actualizar.

Partimos de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS. Realiza un programa Java para insertar varios productos, otro para insertar varios clientes y un tercero para insertar varias ventas a un cliente controlando que el cliente exista, el producto exista y tenga stock (stockactual – cantidad >= stockminimo) y la venta no esté duplicada.

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 el objeto **Session** actual. Algunos métodos importantes de esta interfaz son los siguientes:

MÉTODO	DESCRIPCIÓN
`Iterator iterate()`	Devuelve en un objeto **Iterator** el resultado de la consulta
`List list()`	Devuelve el resultado de la consulta en un **List**
`Query setFetchSize (int size)`	Fija el número de resultados a recuperar en cada acceso a la base de datos al valor indicado en *size*
`int executeUpdate()`	Ejecuta la sentencia de modificación o borrado. Devuelve el número de entidades afectadas
`String getQueryString()`	Devuelve la consulta en un String
`Object uniqueResult()`	Devuelve un objeto (cuando sabemos que la consulta devuelve un objeto) o nulo si la consulta no devuelve resultados
`Query setCharacter(int posición, char valor)` `Query setCharacter(String nombre, char valor)`	Asigna el *valor* indicado en el método a un parámetro de tipo CHAR *posición*, indica la posición del parámetro dentro de la consulta, empieza en 0 *nombre* es el nombre (se indica como *:nombre*) del parámetro dentro de la consulta
`Query setDate(int posición, Date fecha)` `Query setDate(String nombre, Date fecha)`	Asigna la *fecha* a un parámetro de tipo DATE
`Query setDouble(int posición, double valor)` `Query setDouble(String nombre, double valor)`	Asigna *valor* a un parámetro de tipo decimal (en MySQL tipo FLOAT)
`Query setInteger(int posición, int valor)` `Query setInteger(String nombre, int valor)`	Asigna *valor* a un parámetro de tipo entero
`Query setString(int posición, String valor)` `Query setString(String nombre, String valor)`	Asigna *valor* a un parámetro de tipo VARCHAR
`Query setParameterList(String nombre, Collection valores)`	Asigna una colección de valores al parámetro cuyo nombre se indica en *nombre*
`Query setParameter(int posición, Object valor)`	Asigna el *valor* al parámetro indicado en *posición*
`Query setParameter(String nombre, Object valor)`	Asigna el *valor* al parámetro indicado en *nombre*
`int executeUpdate()`	Ejecuta una sentencia UPDATE o DELETE, devuelve el número de entidades afectadas por la operación
Consulta la API de Hibernate: *https://docs.jboss.org/hibernate/orm/current/javadocs/*	

Para realizar una consulta usaremos el método ***createQuery()*** de la interface **SharedSessionContract,** se le pasará en un *String* la consulta HQL:

```
Query createQuery(String queryString)
```

Por ejemplo, para hacer una consulta sobre la tabla *departamentos*, mapeada con la clase *Departamentos* se escribe lo siguiente:

```
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 recuperar los resultados de la consulta. En este caso Hibernate ejecuta la consulta obteniendo solo los ids de las entidades, y en cada llamada al 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 se 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()***. El siguiente ejemplo obtiene 10 filas en cada acceso a la base de datos:

```
q.setFetchSize(10);
```

El siguiente ejemplo realiza una consulta de todas las filas de la tabla *departamentos*:

```
import java.util.Iterator;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import primero.Departamentos;
import primero.HibernateUtil;

public class ListaDepartamentos {
  public static void main(String[] args) {
    SessionFactory sesion = HibernateUtil.getSessionFactory();
    Session session = sesion.openSession();
```

```
    Query q = session.createQuery("from Departamentos");
    List <Departamentos> lista = q.list();

    // Obtenemos un Iterador y recorremos la lista.
    Iterator <Departamentos> iter = lista.iterator();
    System.out.printf("Número de departamentos: %d%n",lista.size());

    while (iter.hasNext())
    {
      //extraer el objeto
      Departamentos depar = (Departamentos) iter.next();
      System.out.printf("%d, %s%n",
                        depar.getDeptNo(), depar.getDnombre());
    }
    session.close();
    System.exit(0);
  }
}
```

Visualiza la siguiente información:

```
Número de departamentos: 5
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();

while (iter.hasNext())
{
   Departamentos depar = (Departamentos) iter.next();
   System.out.printf("%d, %s%n",depar.getDeptNo(), depar.getDnombre());

}
```

El ejemplo siguiente visualiza el apellido y salario de los empleados del departamento 20:

```
Query q = session.createQuery
        ("from Empleados as e where e.departamentos.deptNo = 20");
List<Empleados> lista = q.list();

Iterator<Empleados> iter = lista.iterator();
while (iter.hasNext()) {
      Empleados emp = (Empleados) iter.next();// extraer el objeto
      System.out.printf("%s, %.2f %n",
                        emp.getApellido(), emp.getSalario());
}
```

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 VENTAS:

```
//Visualiza los datos del departamento 10
String hql ="from Departamentos as dep where dep.deptNo = 10";
Query q = session.createQuery(hql);

Departamentos dep = (Departamentos) q.uniqueResult();
System.out.printf("%d, %s, %s%n",dep.getDeptNo(),
                      dep.getLoc(), dep.getDnombre());

//Visualiza los datos del departamento de nombre VENTAS
hql = "from Departamentos as dep where dep.dnombre = 'VENTAS' ";
q = session.createQuery(hql);

dep =(Departamentos) q.uniqueResult();
System.out.printf("%d, %s, %s%n", dep.getDeptNo(),
                      dep.getLoc(), dep.getDnombre());
```

ACTIVIDAD 3.7

Realiza una consulta con ***createQuery()*** para obtener los datos del departamento 20 y visualiza también el apellido de sus empleados.

Partimos de las tablas mapeadas de PRODUCTOS, CLIENTES y VENTAS. Realiza un programa Java que muestre la misma información que el Ejercicio 7 del Capítulo 2 (usa una consulta HQL para obtener la información).

3.9.1. Parámetros en las consultas

Hibernate soporta parámetros con nombre y parámetros de estilo JDBC (?) en las consultas HQL. Los parámetros con nombre son identificadores de la forma ***:nombre*** en la cadena de consulta. 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 autodocumentados.

Para asignar valores a los parámetros se utilizan los métodos ***setXXX*** vistos en la tabla anterior. La sintaxis más simple de utilizar estos parámetros es usando el método ***setParameter()***, por ejemplo, la siguiente consulta utiliza el parámetro nombrado *:numemple* y muestra el apellido y oficio del empleado con número 7369:

```
String hql = "from Empleados where empNo = :numemple";
Query q= session.createQuery(hql);

q.setParameter("numemple", (short) 7369);
Empleados emple = (Empleados) q.uniqueResult();
System.out.printf("%s, %s %n", emple.getApellido(), emple.getOficio());
```

El siguiente ejemplo consulta los empleados cuyo número de departamento es 10 y el oficio DIRECTOR:

```
String hql2 = "from Empleados emp where
                emp.departamentos.deptNo = :ndep and emp.oficio = :ofi";
Query q = session.createQuery(hql2);

q.setParameter("ndep", (byte) 10);
q.setParameter("ofi", "DIRECTOR");
List <Empleados> lista = q.list();
```

El mismo ejemplo con parámetros posicionales de estilo JDBC (el uso de estos parámetros se considera obsoleto por lo que se recomienda usar los parámetros nombrados) quedaría así:

```
String hql = "from Empleados emp
              where emp.departamentos.deptNo = ? and emp.oficio = ?";
Query q = session.createQuery(hql);
q.setParameter(0, (byte) 10);
q.setParameter(1, "DIRECTOR");
```

Para asignar valor a los parámetros también podemos usar los demás métodos ***setXXX***, por ejemplo, para dar valor al número de departamento usamos el método ***setInteger()*** y para el oficio ***setString()***:

```
q.setInteger("ndep", (byte) 10);
q.setString("ofi", "DIRECTOR");
```

El siguiente ejemplo usa el método ***setDate()*** para asignar valor a un parámetro nombrado de tipo *Date*. Obtiene los 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();
}
String hql = "from Empleados where fechaAlt = :fechalta";

Query q = session.createQuery(hql);
q.setDate("fechalta", fecha);
```

El siguiente ejemplo asigna a un parámetro nombrado llamado *:listadep* una colección de valores llamada *numeros* con los valores 10 y 20 para obtener aquellos empleados cuyo número de departamento sea 10 o 20; se usa el método ***setParameerList()***:

```
List <Byte> numeros = new ArrayList <Byte> ();
numeros.add((byte)10);
numeros.add((byte)20);

String hql = "from Empleados emp
              where emp.departamentos.deptNo in (:listadep)
            order by emp.departamentos.deptNo ";
```

```
Query q = session.createQuery(hql);

q.setParameterList("listadep", numeros);
```

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*:

```
String hql="from Empleados e, Departamentos d
            where e.departamentos.deptNo = d.deptNo order by apellido";
Query cons =  session.createQuery(hql);
Iterator q = cons.iterate();
while (q.hasNext()) {
      Object[] par =(Object[]) q.next();
      Empleados em = (Empleados) par[0];
      Departamentos de = (Departamentos) par[1];
      System.out.printf( "%s, %.2f, %s, %s %n",
        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:

```
// MOSTRAR SALARIO MEDIO DE LOS EMPLEADOS
String hql = "select avg(em.salario) from Empleados as em";
Query cons = session.createQuery(hql);
Double suma = (Double) cons.uniqueResult();
System.out.printf("Salario medio: %.2f%n", suma);
```

Si en la consulta intervienen varias funciones de grupo y además devuelve varias filas, podemos utilizar objetos devueltos por las consultas. Por ejemplo, a continuación se muestra el salario medio y el número de empleados por cada departamento:

```
//SALARIO MEDIO Y EL NÚMERO DE EMPLEADOS POR DEPARTAMENTO
String hql = "select e.departamentos.deptNo, avg(salario), count(empNo)
              from Empleados e group by e.departamentos.deptNo ";
```

```
Query cons = session.createQuery(hql);
Iterator iter = cons.iterate();

while (iter.hasNext()) {
      Object[] par = (Object[]) iter.next();
      Byte depar = (Byte) par[0];
      Double media = (Double) par[1];
      Long cuenta = (Long) par[2];
      System.out.printf("Dep: %d, Media: %.2f, N° emp: %d %n",
                        depar, media, cuenta);
}
```

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* quiero 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 constructores, *getter* y *setter* asociados. La clase *Totales.java* es la siguiente:

```
package primero;
public class Totales {
  private Long cuenta;   //número empleados
  private Byte numero;   //número departamento
  private Double media; //media salario
  private String nombre;//nombre depatamento

  public Totales( Byte numero,  Long cuenta,
                Double media,  String nombre) {
      this.cuenta = cuenta;
      this.media = media;
      this.nombre = nombre;
      this.numero = numero;
  }

  public Totales(){}

  public Long getCuenta() {return this.cuenta;}
  public void setCuenta( Long cuenta) {this.cuenta = cuenta;}

  public Byte getNumero() {return this.numero;}
  public void setNumero( Byte numero) {this.numero = 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 construimos la consulta HQL de la siguiente manera:

```
String hql= "select new primero.Totales(" +
     " d.deptNo, count(e.empNo), coalesce(avg(e.salario),0) , "+
     " d.dnombre) "+
     " from Empleados as e right join  e.departamentos as d "+
     " group by  d.deptNo, d.dnombre ";

Query cons = session.createQuery(hql);
Iterator q = cons.iterate();
while (q.hasNext()) {
   Totales tot =(Totales) q.next();
   System.out.printf(
    "Numero Dep: %d, Nombre: %s, Salario medio: %.2f, N° emple: %d%n",
        tot.getNumero(), tot.getNombre(),
        tot.getMedia(), tot.getCuenta());
}
```

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:

```
String hql = "select d.deptNo, count(e.empNo), "
      + " coalesce(avg(e.salario),0), d.dnombre  "
      + " from Empleados as e right join  e.departamentos as d "
      + " group by  d.deptNo, d.dnombre ";

Query cons = session.createQuery(hql);

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.printf(
      "Numero Dep: %d, Nombre: %s,Salario medio: %.2f, N° emple: %d%n",
       filaActual[0],filaActual[3],filaActual[2], filaActual[1]);
}
```

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 ***executeUpdate()*** que devuelve el número de entidades afectadas por la operación; no hemos de olvidar realizar el commit para validar la transacción. Veamos algunos ejemplos, dentro de la misma transacción modificamos un empleado y eliminamos los empleados del departamento 20:

```
Transaction tx = session.beginTransaction();
//Modificamos el salario de GIL
String hqlModif = "update Empleados set salario = :nuevoSal
                   where apellido = :ape";
Query q1 = session.createQuery(hqlModif);

q1.setParameter("nuevoSal", (float) 2500.34);
q1.setString("ape", "GIL");

int filasModif = q1.executeUpdate();
System.out.printf("FILAS MODIFICADAS: %d%n", filasModif);

//Eliminamos los empleados del departamento 20
String hqlDel = "delete Empleados e
                where e.departamentos.deptNo = :dep";
Query q = session.createQuery(hqlDel);

q.setInteger("dep", 20);

int filasDel = q.executeUpdate();
System.out.printf("FILAS ELIMINADAS: %d%n", filasDel);

tx.commit();// valida la transacción
```

Con ***tx.rollback()*** se deshace la transacción.

La sintaxis para la operación INSERT es la siguiente:

```
INSERT INTO NombreEntidad (lista de 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).

Ejemplo: desde SQL creo la siguiente tabla e inserto varias filas:

```
CREATE TABLE nuevos (
 dept_no  TINYINT(2) NOT NULL PRIMARY KEY,
 dnombre  VARCHAR(15),
 loc      VARCHAR(15)
) ENGINE=InnoDB;

INSERT INTO nuevos VALUES (51,'PERSONAL','MADRID');
INSERT INTO nuevos VALUES (52,'NÓMINAS','TOLEDO');
INSERT INTO nuevos VALUES (53,'OCIO','BARCELONA');
```

A continuación añado la siguiente línea al fichero **hibernate.reveng.xml**:

```
<table-filter match-catalog="ejemplo" match-name="nuevos"/>
```

A continuación generamos la nueva clase desde **Hibernate Code Generation Configurations**. Se tiene que generar la clase y el fichero **nuevos.hbm.xml**. Por último, ejecuto el código Java para insertar los datos de esa tabla en *Departamentos*:

```
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Departamentos (deptNo, dnombre, loc)"
                 + " select n.deptNo, n.dnombre, n.loc from Nuevos n";

Query cons = session.createQuery( hqlInsert );
int filascreadas = cons.executeUpdate();

tx.commit(); // valida la transacción

System.out.printf("FILAS INSERTADAS: %d%n",filascreadas);
```

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, una clase, etc. 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[1]:

- 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.*

[1]Fuente: http://docs.jboss.org/hibernate/orm/3.5/reference/es-ES/html/queryhql.html#queryhql-expressions

- Concatenación de cadenas ...||... o *concat(...,...)*.
- *current_date(), current_time()* y *current_timestamp()*.
- *second(...), minute(...), hour(...), day(...), month(...),* y *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 índices, 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.

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 los mapeos realizados sobre las tablas las asociaciones de claves ajenas se generan de forma automática. A continuación vamos a ver cómo se realizan asociaciones de forma manual sobre tablas que no tienen clave ajena definida. Por ejemplo, la tabla *empleados* tiene la columna *dir* que representa el director del empleado y es un número de empleado. Entonces podemos decir que un empleado que es director puede tener a cargo otros empleados, tenemos pues, una relación de uno a muchos (**one-to-many**) entre dos clases persistentes *Empleados*.

Para mapear esta relación añadimos las siguientes líneas al fichero XML **Empleados.hbm.xml** antes de la finalización de la definición de la clase (etiqueta **</class>**):

```
<set name="empleacargo" table="empleados" >
       <key column="dir" />
       <one-to-many class="primero.Empleados" />
</set>
```

En donde la colección se indica mediante la etiqueta **<set> </set>**, que se le da un nombre (*empleacargo*), 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 (*dir*), 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 *Empleados.java* la colección (*empleacargo*) con los métodos *set* y los *get*:

```
private Set<Empleados> empleacargo = new HashSet<Empleados>(0);

public Set<Empleados> getEmpleacargo() {
      return empleacargo;
}

public void setEmpleacargo(Set<Empleados> empleacargo) {
      this.empleacargo = empleacargo;
}
```

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)*. Aunque la forma de utilizarlos es diferente a la usada en SQL. 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 donde se define la asociación. El siguiente ejemplo:

```
from Empleados as emp join emp.empleacargo
```

Devuelve tantas instancias de dos objetos *Empleados* como resulte de combinar la tabla *empleados* consigo misma mediante las columnas *dir* y *emp_no*. El primer objeto resultante representa el director del empleado y el segundo el empleado. La orden anterior se corresponde con la siguiente orden en SQL:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire, empleados em
WHERE dire.emp_no = em.dir
```

En la consulta anterior faltan los empleados que no tienen director, para que se muestren ejecutamos la consulta con *right join*, además la salida se puede ordenar:

```
from Empleados as emp right join emp.empleacargo order by emp.empNo
```

En SQL quedaría así:

```
SELECT dire.emp_no, dire.apellido, em.emp_no, em.apellido
FROM empleados dire
RIGHT JOIN empleados em ON dire.emp_no = em.dir
```

El siguiente ejemplo muestra los datos de los empleados (número de empleado y apellido) y los de su director, la salida se ordena por director:

```
String hql = "from Empleados as emp right join emp.empleacargo
              order by emp.empNo";
Query cons = session.createQuery(hql);
Iterator q = cons.iterate();

while (q.hasNext()) {
      Object[] par = (Object[]) q.next();
      Empleados dir = (Empleados) par[0];//director
      Empleados em = (Empleados) par[1]; //empleado

      if(dir!=null)
         System.out.printf("Empleado: %d, %s, DIRECTOR: %d, %s %n",
                  em.getEmpNo(), em.getApellido(),
                  dir.getEmpNo(), dir.getApellido());
      else
         System.out.printf("Empleado %d, %s, SIN DIRECTOR.%n",
                  em.getEmpNo(), em.getApellido());

}
```

El siguiente ejemplo muestra los empleados, si el empleado es director muestra los que tiene a su cargo:

```
String hql = "from Empleados ";
Query cons = session.createQuery(hql);

List<Empleados> lis = cons.list();
Iterator<Empleados> ite = lis.iterator();
System.out.println("========================================");

while (ite.hasNext()) {
```

```
    Empleados emple = (Empleados) ite.next();
    if (emple != null) {
       Set acargo = emple.getEmpleacargo(); //empleados a cargo
       if (acargo.size() == 0) {// no son directores
          System.out.printf("EMPLEADO: %d, %s %n",
                  emple.getEmpNo(), emple.getApellido());
          System.out.println("=====================================");
       } else {
          System.out.printf("DIRECTOR: %d, %s %n",
                   emple.getEmpNo(), emple.getApellido());

          System.out.println("A cargo: " + acargo.size());

          Iterator it = acargo.iterator();
          while (it.hasNext()) { // recorrer los empleados a cargo
             Empleados em = (Empleados) it.next();
             System.out.printf("\t %d, %s %n",
                   em.getEmpNo(), em.getApellido());
          }
          System.out.println("=====================================");
       }
    }
}//
```

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*

COMPRUEBA TU APRENDIZAJE

1. ¿Qué ventajas aportan las herramientas de mapeo objeto-relacional?
2. Cita alguno de los inconvenientes del uso de herramientas de mapeo objeto-relacional.
3. 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.
4. Disponemos de la siguiente base de datos de nombre ***baloncesto*** y usuario y clave con el mismo nombre. Las tablas se muestran en la Figura 3.33. Son las siguientes:

Figura 3.33.Base de datos *baloncesto*.

EQUIPOS - Contiene información de los equipos que participan en la liga de baloncesto.

```
CREATE TABLE equipos (
  Nombre      varchar(20) NOT NULL,
  Ciudad      varchar(20) DEFAULT NULL,
  Conferencia varchar(4)  DEFAULT NULL,
  Division    varchar(9)  DEFAULT NULL,
  PRIMARY KEY (Nombre) )engine=innodb;
```

JUGADORES - Contiene información de los datos de los jugadores de los equipos. La altura viene en pies y el peso en libras. Al lado se indica la conversión a metros y gramos respectivamente.

```
CREATE TABLE jugadores (
  codigo int NOT NULL,
  Nombre              varchar(30) DEFAULT NULL,
  Procedencia         varchar(20) DEFAULT NULL,
  Altura  varchar(4) DEFAULT NULL, -- en pies, 1 pie 0,3048 metros
  Peso    int DEFAULT NULL, -- en libras 1 libra 453.59 gramos
  Posicion            varchar(12) DEFAULT NULL,
  Nombre_equipo       varchar(20) DEFAULT NULL,
  PRIMARY KEY (codigo),
  FOREIGN KEY (Nombre_equipo) References equipos(Nombre)
)engine=innodb;
```

ESTADÍSTICAS – Esta tabla contiene la información de las estadísticas de los jugadores. La media de puntos por partido, las asistencias realizadas, los tapones, los rebotes, etc.

```
CREATE TABLE estadisticas (
  temporada              varchar(5) NOT NULL ,
  jugador                int NOT NULL ,
  Puntos_por_partido     number(5,2) DEFAULT NULL,
```

```
  Asistencias_por_partido number(5,2) DEFAULT NULL,
  Tapones_por_partido     number(5,2) DEFAULT NULL,
  Rebotes_por_partido     number(5,2) DEFAULT NULL,
  PRIMARY KEY (temporada,jugador),
  FOREIGN KEY (jugador) REFERENCES Jugadores(Codigo)
)engine=innodb;
```

PARTIDOS – Esta tabla contiene la información de los partidos disputados, los equipos y los puntos. Las columnas son:

```
CREATE TABLE partidos (
  codigo           int NOT NULL,
  equipo_local     varchar(20) DEFAULT NULL,
  equipo_visitante varchar(20) DEFAULT NULL,
  puntos_local     number(5) DEFAULT NULL,
  puntos_visitante number(5) DEFAULT NULL,
  temporada        varchar(5) DEFAULT NULL,
  PRIMARY KEY (codigo),
  FOREIGN KEY (equipo_local) REFERENCES equipos(nombre),
  FOREIGN KEY (equipo_visitante) REFERENCES equipos(nombre)
)engine=innodb;
```

Mapea las tablas de la base de datos y estudia las clases generadas.

5. Realiza un programa Java que admita un argumento desde *main()*, este argumento es el código de un jugador. El programa debe mostrar las estadísticas del jugador. Se deben controlar situaciones de error: si no se introduce ningún parámetro o el parámetro no es correcto (debe ser numérico) se debe mostrar un mensaje de error; si el jugador no existe muestra un mensaje indicándolo. Por ejemplo, si el código de jugador es 227, se debe mostrar la siguiente información:

```
DATOS DEL JUGADOR: 227
Nombre : Kirk Hinrich
Equipo : Bulls
Temporada  Ptos   Asis    Tap     Reb
=========================================
06/07      16.6     6.3   0.3     3.4
03/04      12.0     6.8   0.3     3.4
05/06      15.9     6.3   0.3     3.6
07/08      12.0     6.0   0.3     3.4
04/05      15.7     6.4   0.3     3.9
=========================================
Num de registros: 5
=========================================
```

6. Realiza un programa Java que muestre por cada equipo la lista de sus jugadores con la media de los puntos por partido. El listado debe aparecer ordenado por equipo. Debe mostrar también el número de equipos que hay. Ejemplo de salida del programa:

```
Número de Equipos: 30
=====================================================
Equipo: 76ers
120, Louis Amundson:  1,45
125, Willie Green:  9,04
```

```
. . . . . .
==================================================
Equipo: Bobcats
181, Derek Anderson:  11,20
184, Jermareo Davidson:  3,20
191, Adam Morrison:  11,80
. . . . . .
==================================================
Equipo: Bucks
204, Royal Ivey:  3,93
. . . . . .
```

7. Realiza un programa Java que inserte estadísticas para el jugador 123. Los datos a insertar son los siguientes: temporada 05/06: puntos por partido 7, rebotes 5; temporada 06/07 puntos por partido 10, tapones 3. Los valores no indicados tendrán valor 0. Transforma después el programa para que todos los valores a insertar se introduzcan a partir de los argumentos de *main()*. Controlar posibles errores, número de argumentos correctos, que el jugador exista, que la estadística no exista, etc.

8. Realiza un programa Java para visualizar los datos de los empleados. La pantalla debe presentar 5 botones que nos permitirán iniciar la visualización de empleados y movernos entre los empleados. Los campos que se muestran en la pantalla no son editables. El botón *Ejecutar Consulta* y *Primer Reg* deben mostrar el primer empleado (el que tiene menor número de empleado). Si no hay empleados visualizar mensaje indicándolo. El botón *Último Reg* muestra el último empleado (el que tiene el mayor número de empleado). El botón *Siguiente* muestra el siguiente empleado al que se está visualizando en este momento. Si se pulsa y estamos ante el último empleado se debe visualizar un mensaje indicándolo (por ejemplo: "No hay más empleados"). El botón *Anterior* muestra 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"). Al hacer clic en el botón de cierre de la ventana finalizará el programa. La pantalla es la siguiente:

Figura 3.34. Ejercicio 8.

ACTIVIDADES DE AMPLIACIÓN

1. En una compañía de metro se dispone de una base de datos que contiene las tablas con la información necesaria para su gestión. Las tablas son las siguientes:
   - Tabla **T_Lineas**, contiene la información de las distintas líneas de metro que gestiona la compañía. La clave es el ***cod_linea***.
   - Tabla **T_Estaciones**, contiene la información de las distintas estaciones de metro que existen y son gestionadas por la compañía. La clave es ***cod_estacion***.
   - Tabla **T_Linea_estacion,** contiene la información de las estaciones que tiene cada línea y el orden que hace cada estación dentro de la línea, este número no se puede repetir dentro de la misma línea. La clave está formada por el ***cod_linea*** y el ***cod_estacion***.
   - Tabla **T_Accesos**, contiene la información sobre los distintos accesos de entrada por los que se puede llegar a cada estación. Una estación puede tener distintos puntos de acceso. La clave es ***cod_acceso.***
   - Tabla **T_Trenes**, contiene la información sobre los distintos trenes que circulan por cada línea. En una línea circulan muchos trenes. Y un tren pertenece a una línea. La clave es ***cod_tren***. Los trenes se guardan en cocheras.
   - Tabla **T_Cocheras**, contiene la información sobre las cocheras y depósitos donde aparcan los trenes al final de la jornada. La clave es ***cod_cochera.*** En una cochera se guardan muchos trenes, y un tren se guarda en una cochera.
   - Tabla **T_Viajes**, contiene la información sobre distintos viajes que ofrece la compañía. Cada viaje tiene una estación de destino y una estación de origen. La clave es cod_viaje.
   -

Las tablas y sus relaciones se muestran en la siguiente Figura 3.35:

Figura 3.35. Modelo Actvidad de ampliación.

Mapea las tablas utilizando Hibernate y realiza un proyecto Java con los siguientes métodos (si prefieres puedes hacer cada ejercicio en una clase):

2. Crea un método que reciba un número de línea, un número de estación, el orden y los inserte en la tabla T_LINEA_ESTACION. Antes de insertar hay que comprobar que la línea y la estación existan en las tablas correspondientes, que no exista el registro en la tabla T_LINEA_ESTACION, y que el orden sea correcto. Visualiza los mensajes de error que correspondan (línea inexistente, estación inexistente, registro ya existe…)

3. Crea un método para actualizar los campos *numaccesos*, *numlineas*, *numviajesdestino* y *numviajesprocedencia* de las estaciones de la tabla T_ESTACIONES. Estas columnas deben contener el número de accesos que tiene la estación (*numaccesos*), el número de líneas que pasan por la estación (*numlineas*), el número de viajes que la tienen como destino (*numviajesdestino*), y el número de viajes que la tienen como procedencia (*numviajesprocedencia*).

4. Crea un método que visualice por cada estación, el número de líneas que pasan por ella, el número de accesos que tiene, el número de viajes que tienen como destino la estación y los viajes con su nombre y su código. Y lo mismo con los viajes de procedencia. Obtén la siguiente salida por estación:

```
COD ESTACIÓN: xxxxxx  NOMBRE ESTACIÓN: xxxxxxxxxxx
-----------------------------------------------------
Números de líneas que pasan: xxxxxx
Número de accesos que tiene: xxxxxx
NUM-Viajes-DESTINO: xxxxxxxx
COD-VIAJE     NOMBRE-VIAJE-DESTINO
----------   ---------------------
   xxxx       xxxxxxxxxxxxxxxxxxx
   xxxx       xxxxxxxxxxxxxxxxxxx

NUM-Viajes-PROCEDENCIA: xxxxxxxx
COD-VIAJE     NOMBRE-VIAJE-PROCEDENCIA
----------   -------------------------
   xxxx       xxxxxxxxxxxxxxxxxxx
   xxxx       xxxxxxxxxxxxxxxxxxx
-----------------------------------------------------
```

5. Crea un método o una clase para crear el fichero *Viajes.xml* que debe contener la información de todos los viajes: el código, el nombre, y los nombres de la estación de destino y de procedencia. Para ello utiliza JAXB, y crea las clases correspondientes para crear el mapeo a XML. El fichero XML debe de tener la siguiente estructura:

```
<todoslosviajes>
  <viaje>
      <codigoviaje> </codigoviaje>
      <nombreviaje> </nombreviaje>
      <nombreestacionorigen>  </nombreestacionorigen>
      <nombreestaciondestino> </nombreestaciondestino>
  </viaje>
</todoslosviajes>
```

CAPÍTULO 4

BASES DE DATOS OBJETO-RELACIONALES Y ORIENTADAS A OBJETOS

OBJETIVOS

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.

CONTENIDOS

Bases de datos Objeto-Relacionales. Características. Tipos de objetos. Métodos. Tablas de objetos. Tipos colección. Varrays. Tablas anidadas. Referencias.

Bases de datos Orientadas a Objetos. Características. El estándar ODMG. El lenguaje de consultas OQL.

Uso Bases de datos Orientadas a Objetos. Almacenamiento de objetos. Consultas sencillas. Consultas más complejas.

RESUMEN

En este capítulo utilizaremos una base de datos relacional a la que se le ha añadido conceptos del modelo orientado a objetos. Aprenderemos a crear tipos complejos de datos. Posteriormente usaremos una base de datos orientada a objetos pura, aprenderemos a almacenar y recuperar los objetos.

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** (OR REPLACE reemplaza el tipo si ya existe). 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 (desde la línea de comandos de SQL). 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);
  DIR2 DIRECCION;   -- SE INICIA CON NEW
  P2 PERSONA;       -- SE INICIA CON NEW
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' ;
  DBMS_OUTPUT.PUT_LINE('NOMBRE: ' ||P.NOMBRE || ' * CALLE: ' ||
                       P.DIREC.CALLE);
  --
  DIR2 := NEW DIRECCION ('C/Madrid 10', 'Toledo', 45002);
  P2 := NEW PERSONA(2, 'JUAN', DIR2, SYSDATE);
```

```
  DBMS_OUTPUT.PUT_LINE('NOMBRE: ' ||P2.NOMBRE || ' * CALLE: ' ||
                        P2.DIREC.CALLE );
END;
/
```

¡¡INTERESANTE!!

Si utilizamos SQLDEVELOPER, para activar DBMS_OUTPUT seguimos estos pasos: Opción de menú ***Ver->Salida de DBMS***, se abre una nueva ventana. Pulsar el botón + para activar DBMS_OUTPUT y a continuación seleccionar la conexión. Desde la línea de comandos escribimos: SET SERVEROUTPUT ON.

Para borrar un tipo usamos la orden **DROP TYPE** indicando a la derecha el nombre de tipo a borrar: *DROP TYPE nombre_tipo;*

ACTIVIDAD 4.1

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. Estos métodos son operaciones globales que no son de los objetos, sino del tipo.
- **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. Los parámetros del constructor coinciden con los atributos del tipo de objeto, esto es, los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo. No obstante, podemos sobrescribirlo 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**. PL/SQL nunca invoca al constructor implícitamente, por lo que el usuario debe invocarlo explícitamente.

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, un constructor que recibe 2 parámetros, un método **STATIC** y otro **MEMBER**:

```
CREATE OR REPLACE TYPE RECTANGULO AS OBJECT
(
  BASE   NUMBER,
  ALTURA NUMBER,
  AREA   NUMBER,
  STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER),
  MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER),
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
                        RETURN SELF AS RESULT
);
/
```

Una vez creado el tipo con la especificación de los atributos y los métodos crearemos el cuerpo del tipo con la implementación de los métodos, usaremos 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 formato:

```
[STATIC | MEMBER] PROCEDURE nombreProc [(parametro1, parámetro2, ...)]
IS
  Declaraciones;
BEGIN
  Instrucciones;
END;
[STATIC | MEMBER | CONSTRUCTOR] FUNCTION nombreFunc
[(param1, param2, ... )] 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 invocarlo al definir el objeto (también se puede llamar al constructor con el operador **NEW**):

```
DECLARE
  DIR DIRECCION := DIRECCION(NULL, NULL, NULL);--Llamada al constructor
BEGIN
  DIR.SET_CALLE('La Mina, 3');
  DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
  DIR := NEW DIRECCION ('C/Madrid 10', 'Toledo', 45002);
  DBMS_OUTPUT.PUT_LINE(DIR.GET_CALLE);
END;
/
```

La implementación de los métodos del objeto RECTÁNGULO se muestra a continuación; antes se crea la tabla TABLAREC que usarán los métodos para insertar datos. En el constructor para hacer referencia a los atributos del objeto a partir del cual se invocó el método usamos el cualificador **SELF** delante del atributo, en el método **STATIC** no están permitidas las referencias a los atributos de instancia, en los métodos **MEMBER** sí está permitido:

```
CREATE TABLE TABLAREC (VALOR INTEGER);
/

CREATE OR REPLACE TYPE BODY RECTANGULO AS
--
  CONSTRUCTOR FUNCTION RECTANGULO (BASE NUMBER, ALTURA NUMBER)
                                   RETURN SELF AS RESULT IS
  BEGIN
    SELF.BASE := BASE;
    SELF.ALTURA := ALTURA;
    SELF.AREA := BASE * ALTURA;
    RETURN;
  END;
--
  STATIC PROCEDURE PROC1 (ANCHO INTEGER, ALTO INTEGER) IS
  BEGIN
    INSERT INTO TABLAREC VALUES(ANCHO*ALTO);
    --ALTURA := ALTO; --ERROR NO SE PUEDE ACCEDER A LOS ATRIBUTOS DEL TIPO
    DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
    COMMIT;
  END;
--
  MEMBER PROCEDURE PROC2 (ANCHO INTEGER, ALTO INTEGER) IS
  BEGIN
    SELF.ALTURA := ALTO; --SE PUEDE ACCEDER A LOS ATRIBUTOS DEL TIPO
    SELF.BASE := ANCHO;
    AREA := ALTURA*BASE;
    INSERT INTO TABLAREC VALUES(AREA);
    DBMS_OUTPUT.PUT_LINE('FILA INSERTADA');
```

```
    COMMIT;
  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 en el bloque BEGIN 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;
  DBMS_OUTPUT.PUT_LINE('AREA R3: '||R3.AREA);

  --USO DE LOS MÉTODOS DEL TIPO  RECTANGULO
  RECTANGULO.PROC1(10, 20);    --LLAMADA AL MÉTODO STATIC
  --RECTANGULO.PROC2(20, 30); --ERROR, LLAMADA AL MÉTODO MEMBER
  --R1.PROC1(5, 6);           --ERROR, LLAMADA AL MÉTODO STATIC
  R1.PROC2(5, 10);            --LLAMADA AL MÉTODO MEMBER
END;
/
```

En cuanto a los métodos, se produce error al llamar al método **STATIC** usando cualquiera de los objetos instanciados. También se produce error si la llamada a un método **MEMBER** se realiza sin haber instanciado un objeto.

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 4.2

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, etc.) 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 Capítulo.

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'
);
```

El siguiente bloque PL/SQL inserta una fila en la tabla ALUMNOS:

```
DECLARE
  DIR DIRECCION := DIRECCION('C/Sevilla 20', 'GUADALAJARA', 19004);
  PER PERSONA := PERSONA(5, 'MANUEL',DIR, '20/10/1987');
BEGIN
  INSERT INTO ALUMNOS VALUES(PER); --insertar
  COMMIT;
END;
/
```

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 código 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;
/
```

- El siguiente bloque PL/SQL modifica la dirección completa de un alumno:

```
DECLARE
  D DIRECCION := DIRECCION ('C/Galiano 5', 'Guadalajara', 19004);
BEGIN
  UPDATE ALUMNOS
         SET DIREC = D WHERE NOMBRE = 'Juan Pérez';
  COMMIT;
END;
/
```

ACTIVIDAD 4.3

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.

- Alumnos de GUADALAJARA con nota media mayor de 6.
- Nombre de alumno con más nota media.
- Nombre de alumno con nota más alta (cualquiera de sus notas).

Realiza los ejercicios propuestos 2 y 3.

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 a 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 := '*';

    --Recorrer el varray
    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 4.4

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.

La función deberá de controlar si la persona no tiene teléfonos, si no tiene que devuelva un mensaje indicándolo. Controla los posibles errores.

Para obtener información sobre la colección tenemos los siguientes métodos:

Parámetros	Función
COUNT	Devuelve el número de elementos de la colección
EXISTS	Devuelve TRUE si la fila existe
FIRST/LAST	Devuelve el índice del primer y último elemento de la colección.
NEXT/PRIOR	Devuelve el elemento próximo o anterior al actual
LIMIT	Informa del número máximo de elementos que puede contener la colección

Para modificar los elementos de la colección tenemos los siguientes métodos:

Parámetros	Función
DELETE	Elimina todos los elementos de la colección
EXTEND	Añade un elemento nulo a al colección
EXTEND(n)	Añade n elementos nulos
TRIM	Elimina el elemento situado al final de la colección
TRIM(n)	Elimina n elementos del final de la colección

El siguiente ejemplo muestra cómo usar los parámetros:

```
DECLARE
  TEL TELEFONO := TELEFONO(NULL, NULL, NULL);
BEGIN
  SELECT TELEF INTO  TEL FROM AGENDA WHERE NOMBRE = 'MARTA';

  --Visualizar Datos
  DBMS_OUTPUT.PUT_LINE('Nº DE TELÉFONOS ACTUALES:    ' || TEL.COUNT);
  DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL PRIMER ELEMENTO: ' || TEL.FIRST);
  DBMS_OUTPUT.PUT_LINE('ÍNDICE DEL ÚLTIMO ELEMENTO: ' || TEL.LAST);
  DBMS_OUTPUT.PUT_LINE('MÁXIMO Nº DE TLFS PERMITIDO:' || TEL.LIMIT);

  --Añade un número de teléfono a MARTA
  TEL.EXTEND;
  TEL(TEL.COUNT):= '123000000';
  UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MARTA';

  --Elimina un teléfono
  SELECT TELEF INTO TEL FROM AGENDA WHERE NOMBRE = 'MANUEL';
  TEL.TRIM;    --Elimina el último elemento del array
  TEL.DELETE; --Elimina todos los elementos
  UPDATE AGENDA A SET A.TELEF = TEL WHERE NOMBRE = 'MANUEL';
END;
/
```

ACTIVIDAD 4.5

Crea un VARRAY de 5 elementos de tipo PERSONA.

Crea después la tabla GRUPOS, con dos columnas: la primera contiene el nombre de grupo de tipo VARCHAR2(15) y la segunda es del tipo definido anteriormente.

Partiendo de las tablas EMPLEADOS y DEPARTAMENTOS llena la tabla GRUPOS. Como nombre de grupo se pondrá el nombre de departamento, como nombre de persona el apellido del empleado, como código la columna EMP_NO y como calle la localidad del departamento. Puedes realizar un procedimiento para ello. Cada fila de la tabla GRUPOS representa un departamento con hasta 5 empleados.

Realiza un bloque PL/SQL que recorra la tabla GRUPOS mostrando por cada departamento el apellido de sus empleados.

Realiza los ejercicios propuestos 4, 5 y 6.

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**. Sintaxis:

```
CREATE TYPE nombre_tipo AS TABLE OF tipo_de_dato;
```

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)
  )
);
```

- Se inserta el código, el nombre y la tabla anidada vacía:

```
INSERT INTO EJEMPLO_TABLA_ANIDADA
            VALUES (5, 'PEREZ', TABLA_ANIDADA());
```

- Seleccionamos todas las filas de la tabla:

```
SELECT * FROM EJEMPLO_TABLA_ANIDADA;
```

```
  1 RAMOS  EJEMPLO.TABLA_ANIDADA([EJEMPLO.DIRECCION],[EJEMPLO.DIRECCION],[EJEMPLO.DIRECCION],[EJEMPLO.DI
  2 MARTIN EJEMPLO.TABLA_ANIDADA([EJEMPLO.DIRECCION],[EJEMPLO.DIRECCION])
  5 PEREZ  EJEMPLO.TABLA_ANIDADA()
```

- El siguiente ejemplo obtiene el identificador, el apellido y la dirección completa de todas las filas de la tabla. Se obtienen tantas filas como calles tiene cada identificador. El operador **TABLE** con la columna que es tabla anidada entre paréntesis y colocado

a la derecha de FROM se utiliza para acceder a todas las filas de la tabla anidada, es necesario indicar el alias (en este caso se llama DIRECCION); con DIRECCION.* se obtienen todos los campos de la dirección:

```
SELECT  ID, APELLIDOS, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION;
```

	ID	APELLIDOS	CALLE	CIUDAD	CODIGO_POST
1	1	RAMOS	C/Los manantiales 5	GUADALAJARA	19004
2	1	RAMOS	C/Los manantiales 10	GUADALAJARA	19004
3	1	RAMOS	C/Av de Paris 25	CÁCERES	10005
4	1	RAMOS	C/Segovia 23-3A	TOLEDO	45005
5	2	MARTÍN	C/Huesca 5	ALCALÁ DE H	28804
6	2	MARTÍN	C/Madrid 20	ALCORCÓN	28921

En la consulta anterior la columna que es tabla anidada se utiliza como si fuese una tabla normal, incluyéndola en la cláusula FROM.

- El siguiente ejemplo obtiene las direcciones completas del identificador 1:

```
SELECT  ID, DIRECCION.*
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) DIRECCION WHERE ID=1;
```

	ID	CALLE	CIUDAD	CODIGO_POST
1	1	C/Los manantiales 5	GUADALAJARA	19004
2	1	C/Los manantiales 10	GUADALAJARA	19004
3	1	C/Av de Paris 25	CACERES	10005
4	1	C/Segovia 23-3A	TOLEDO	45005

Se pueden usar cursores dentro de una SELECT para acceder o poner condiciones a las filas de una tabla anidada. La sintaxis es la siguiente:

```
CURSOR (SELECT columnas FROM TABLE (columna_tabla_anidada))
```

Donde ***columnas*** son las columnas del tipo de dato de la tabla anidada.

- A continuación obtenemos el identificador, los apellidos y las calles y ciudad de cada fila de la tabla, se obtienen tantas filas como filas hay en la tabla, con el operador **TABLE** se hace referencia a la tabla anidada:

```
SELECT ID, APELLIDOS, CURSOR (SELECT CALLE, CIUDAD FROM TABLE(DIREC))
FROM EJEMPLO_TABLA_ANIDADA ;
```

	ID	APELLIDOS	CURSOR(SELECT CALLE,CIUDAD FROM TANIDADA(DIREC))
1	1	RAMOS	{<CALLE=C/Los manantiales 5,CIUDAD=GUADALAJARA>,<CALLE=C/Los manantiales 10,CIUDAD-
2	2	MARTÍN	{<CALLE=C/Huesca 5,CIUDAD=ALCALA DE H>,<CALLE=C/Madrid 20,CIUDAD=ALCORCÓN>,}
3	5	PEREZ	{}

- Es habitual el uso de alias en las tablas anidadas:

```
SELECT ID, APELLIDOS,
       CURSOR (SELECT T.CALLE, T.CIUDAD FROM TABLE(DIREC) T)
FROM EJEMPLO_TABLA_ANIDADA ;
```

- Las siguientes consultas muestran el número de direcciones de cada identificador, la primera usa la tabla anidada dentro de un **CURSOR**, la segunda como si fuese una

tabla normal utilizando el operador **TABLE**, en este caso es necesario usar GROUP BY, ya que cada identificador puede tener varias direcciones:

```
SELECT ID, APELLIDOS, CURSOR(SELECT count(*) FROM TABLE(DIREC) )
FROM EJEMPLO_TABLA_ANIDADA;
```

	ID	APELLIDOS	CURSOR(SELECT COUNT(*) FROM TABLE(DIREC))
1	1	RAMOS	{<COUNT(*)=4>,}
2	2	MARTÍN	{<COUNT(*)=2>,}
3	5	PEREZ	{<COUNT(*)=0>,}

```
SELECT ID, APELLIDOS, count(*)
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
GROUP BY ID, APELLIDOS;
```

	ID	APELLIDOS	COUNT(*)
1	1	RAMOS	4
2	2	MARTÍN	2

- Las siguientes consultas muestran aquellas filas que tienen 2 direcciones en la CIUDAD de GUADALAJARA, la primera usa CURSOR para acceder a la tabla anidada y la segunda usa la tabla anidada como si fuese una tabla normal, que se combina con otra tabla:

```
SELECT ID, APELLIDOS, CURSOR (SELECT COUNT(*) FROM  TABLE(DIREC)
                              WHERE CIUDAD = 'GUADALAJARA')
FROM EJEMPLO_TABLA_ANIDADA
WHERE
  (SELECT COUNT(*)FROM TABLE(DIREC) WHERE CIUDAD = 'GUADALAJARA') = 2;
```

	ID	APELLIDOS	CURSOR(SELECT COUNT(*) FROM TABLE(DIREC) WHERE CIUDAD='GUADALAJARA')
1	1	RAMOS	{<COUNT(*)=2>,}

```
SELECT ID, APELLIDOS, COUNT(*)
FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
WHERE CIUDAD = 'GUADALAJARA'
GROUP BY ID, APELLIDOS HAVING COUNT(*) = 2;
```

	ID	APELLIDOS	COUNT(*)
1	1	RAMOS	2

Para seleccionar filas de una tabla anidada se puede utilizar la cláusula **THE** con SELECT. La sintaxis es:

```
SELECT ... FROM THE (subconsulta sobre tabla anidada) WHERE ...
```

- El siguiente ejemplo obtiene 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 DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1)
WHERE CIUDAD = 'GUADALAJARA';
```

	CALLE
1	C/Los manantiales 5
2	C/Los manantiales 10

- La siguiente consulta obtiene todos los datos de las direcciones del identificador 2:

```
SELECT * FROM THE
        (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 2);
```

	CALLE	CIUDAD	CODIGO_POST
1	C/Huesca 5	ALCALÁ DE H	28804
2	C/Madrid 20	ALCORCÓN	28921

- La siguiente consulta usa la tabla anidada en FROM y obtiene el mismo resultado que la anterior:

```
SELECT TT.* FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC) TT WHERE ID = 2;
```

ACTIVIDAD 4.6

Obtén el número de direcciones que tiene en cada ciudad el identificador 1.

Obtén la ciudad con más direcciones que tiene el identificador 1.

Realiza un bloque PL/SQL que muestre el nombre de las calles de cada apellido.

- 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', 19004));
```

La cláusula **TABLE** a la derecha de INTO se utiliza para acceder a la fila que nos interesa, en este caso la que tiene ID = 1.

- En el siguiente ejemplo se modifica la primera dirección del identificador 1, se le asigna el valor *'C/Pilón 11', 'TOLEDO', 45589*:

```
UPDATE TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET VALUE(PRIMERA) = DIRECCION ('C/Pilón 11', 'TOLEDO', 45589)
WHERE
VALUE(PRIMERA)= DIRECCION('C/Los manantiales 5', 'GUADALAJARA', 19004);
```

El alias **PRIMERA** recoge los datos devueltos por la SELECT (que debe devolver una fila). Con **SET VALUE (PRIMERA)** se asigna el valor *'C/Pilón 11', 'TOLEDO', 45589* al objeto DIRECCIÓN cuyo valor coincida con *'C/Los manantiales 5', 'GUADALAJARA', 19004*; esto se indica en la cláusula WHERE con la función **VALUE(PRIMERA)**.

- En el siguiente ejemplo se modifican (para el identificador 1) todas las direcciones que tengan la ciudad de GUADALAJARA, se le asigna el valor MADRID. En este caso no se necesita la función **VALUE**, ya que se modifica la columna CIUDAD y no un objeto:

```
UPDATE TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
SET    PRIMERA.CIUDAD = 'MADRID'
WHERE PRIMERA.CIUDAD = 'GUADALAJARA';
```

En el siguiente ejemplo se elimina la segunda dirección del identificador 1, aquella cuyo valor es *'C/Los manantiales 10', 'GUADALAJARA', 19004*:

```
DELETE FROM TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE
VALUE(PRIMERA)=DIRECCION('C/Los manantiales 10', 'GUADALAJARA', 19004);
```

En el siguiente ejemplo se eliminan todas las direcciones del identificador 1 con ciudad igual a 'GUADALAJARA':

```
DELETE FROM TABLE
       (SELECT DIREC FROM EJEMPLO_TABLA_ANIDADA WHERE ID = 1) PRIMERA
WHERE PRIMERA.CIUDAD = 'GUADALAJARA';
```

- 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;
/
--Probando el procedimiento
BEGIN
  VER_DIREC(1);
END;
/
```

La vista **USER_NESTED_TABLES** obtiene información de las tablas anidadas.

- El siguiente ejemplo crea una función almacenada que comprueba si existe una dirección en un identificador concreto. La función recibe el identificador y un tipo DIRECCION, devuelve un mensaje indicando si existe o no la dirección. Primero se comprobará si existe el identificador, si no existe o si existen varias filas con el mismo identificador se devuelve un mensaje indicándolo.

```
CREATE OR REPLACE FUNCTION EXISTE DIREC
                          (IDEN NUMBER, DIR DIRECCION)
          RETURN VARCHAR2 AS
  IDT NUMBER;
  CUENTA NUMBER;
BEGIN
```

```
--COMPROBAR SI EXISTE ID:
  SELECT COUNT(ID) INTO CUENTA
  FROM EJEMPLO_TABLA_ANIDADA WHERE ID = IDEN;

  IF CUENTA = 0 THEN
    RETURN 'NO EXISTE EL ID: '|| IDEN || ', EN LA TABLA';
  END IF;

  IF CUENTA > 1 THEN
    RETURN 'EXISTEN VARIOS REGISTROS CON EL MISMO  ID: ' ||IDEN;
  END IF;

--EL ID EXISTE, COMPROBAR SI LA CALLE EXISTE:
  SELECT ID INTO IDT
         FROM EJEMPLO_TABLA_ANIDADA, TABLE(DIREC)
         WHERE ID = IDEN
         AND UPPER(CALLE) = UPPER(DIR.CALLE)
         AND UPPER(CIUDAD) = UPPER(DIR.CIUDAD)
         AND CODIGO_POST = DIR.CODIGO_POST;

  RETURN ('LA DIRECCIÓN : '||DIR.CALLE || '*' ||DIR.CIUDAD
                           || '*' || DIR.CODIGO_POST
                           || ' YA EXISTE PARA ESE ID: '||IDEN);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
  RETURN 'NO EXISTE LA DIRECCIÓN - '||DIR.CALLE || '*'  ||DIR.CIUDAD
         || '*' || DIR.CODIGO_POST   || ' PARA EL ID: ' ||IDEN;
END EXISTE_DIREC;
/

--Probando la función
BEGIN
  DBMS_OUTPUT.PUT_LINE
    (EXISTE_DIREC(1, DIRECCION('C/Huesca 5', 'ALCALÁ DE H', 28804)));
  DBMS_OUTPUT.PUT_LINE
    (EXISTE_DIREC(2, DIRECCION('C/Huesca 5', 'ALCALÁ DE H', 28804)));
END;
/
```

ACTIVIDAD 4.7

Realiza un procedimiento almacenado para insertar direcciones en la tabla EJEMPLO_TABLA_ANIDADA.

El procedimiento recibe como parámetros un identificador y un objeto DIRECCION. Debe visualizar un mensaje indicando si se ha insertado o no la dirección.

Se deben hacer las siguientes comprobaciones y visualizar los mensajes correspondientes:

- Comprobar si el identificador existe, si no existe es un caso de error, visualizar mensaje.
- Que la tabla anidada no sea null, si es null hay que hacer un UPDATE no un INSERT.
- Que la dirección no exista ya en la tabla, si ya existe visualiza que no se puede insertar.

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. Ejemplos:

- 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';

INSERT INTO EMPLEADO SELECT EMPLEADO_T ('RAMOS', 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:

```
SQL> SELECT NOMBRE, DEREF(P.JEFE) FROM EMPLEADO P;
NOMBRE         DEREF(P.JEFE)(NOMBRE, JEFE)
-------------- ----------------------------------------------------
GIL
ARROYO         EMPLEADO_T('GIL', NULL)
RAMOS          EMPLEADO_T('GIL', NULL)
```

- La siguiente consulta obtiene el identificador del objeto cuyo nombre es GIL:

```
SELECT REF(P) FROM EMPLEADO P WHERE NOMBRE = 'GIL';
```

- La siguiente consulta obtiene nombre del empleado y el nombre de su jefe:

```
SQL> SELECT NOMBRE, DEREF(P.JEFE).NOMBRE FROM EMPLEADO P;
NOMBRE                          DEREF(P.JEFE).NOMBRE
------------------------------- ------------------------------
GIL
ARROYO                          GIL
RAMOS                           GIL
```

- El siguiente ejemplo actualiza el jefe del nombre RAMOS, se le asigna ARROYO:

```
UPDATE EMPLEADO
  SET JEFE = (SELECT REF(E) FROM EMPLEADO E WHERE NOMBRE = 'ARROYO')
WHERE NOMBRE = 'RAMOS '.
```

ACTIVIDAD 4.8

Crea un TIPO_DEP con las siguientes columnas: DEPT_NO NUMBER(2), DNOMBRE VARCHAR2(15), LOC VARCHAR2(15). Crea una tabla del tipo definido anteriormente. Llena la tabla a partir de los datos de la tabla DEPARTAMENTOS.

Crea una tabla con las siguientes columnas, una de ellas es una referencia a un TIPO_DEP: EMP_NO NUMBER(4), APELLIDO VARCHAR2(15), SALARIO NUMBER(6,2) y DEPT REF TIPO_DEP. Llena esta tabla a partir de la tabla EMPLEADOS.

Haz un bloque PL/SQL que recorra esta última tabla y muestre el apellido, salario, número de departamento, nombre y localidad.

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, -- No se puede redefinir
  MEMBER FUNCTION GET_NOMBRE RETURN VARCHAR2,
  MEMBER PROCEDURE VER_DATOS
) NOT FINAL;  -- Se pueden derivar subtipos
/
--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(
                                        --se define un subtipo
  CURSO VARCHAR2(10),
  NOTA_FINAL NUMBER,
  MEMBER FUNCTION NOTA RETURN NUMBER,
  OVERRIDING MEMBER PROCEDURE VER_DATOS  --se redefine ese método
);
/
--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  --se redefine ese método
  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
  --Al asignar datos al alumno escribimos
  --        DNI, NOMBRE, FECHA_NAC, CURSO, NOTA

  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 (al insertar se escriben las columnas del supertipo - dni, nombre, fec_nac - y luego las del subtipo - curso, nota_final -):

```
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 anidada 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);
```

Insertamos 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;
```

Insertamos 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);
```

Insertamos 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;
```

Insertamos 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);
```

La siguiente consulta muestra el total de ventas en cada venta:

```
SELECT  IDVENTA, DEREF(IDCLIENTE).NOMBRE NOMBRE,
DEREF(IDCLIENTE).IDCLIENTE IDCLIENTE, T.TOTAL_VENTA() TOTAL
FROM TABLA_VENTAS T;

IDVENTA NOMBRE                              IDCLIENTE      TOTAL
---------- ------------------------------ ---------- ----------
         1 Luis Gracia                             1        340
         2 Luis Gracia                             1        327
```

La siguiente consulta muestra el detalle de los productos junto con la venta y el cliente; se puede utilizar la tabla anidada como tabla en la consulta poniendo la cláusula **TABLE**:

```
SELECT P.IDVENTA IDV, DEREF(P.IDCLIENTE).NOMBRE NOMBRE,
       DETALLE.NUMEROLINEA LINEA,
       DEREF(DETALLE.IDPRODUCTO).DESCRIPCION PRODUCTO,
       DETALLE.CANTIDAD,
       DETALLE.CANTIDAD * DEREF(DETALLE.IDPRODUCTO).PVP IMPORTE,
       DEREF(DETALLE.IDPRODUCTO).PVP PVP,
       DEREF(DETALLE.IDPRODUCTO).STOCKACTUAL STOCK
FROM TABLA_VENTAS P, TABLE(P.LINEAS) DETALLE;
```

IDV	NOMBRE	LINEA	PRODUCTO	CANTIDAD	IMPORTE	PVP	STOCK
1	Luis Gracia	1	CAJA DE CRISTAL DE MURANO	1	100	100	5
1	Luis Gracia	2	BICICLETA CITY	2	240	120	15
2	Luis Gracia	1	CAJA DE CRISTAL DE MURANO	2	200	100	5
2	Luis Gracia	2	OPERACIONES CON BD	1	25	25	5
2	Luis Gracia	3	APLICACIONES WEB	4	102	25,5	10

El siguiente procedimiento almacenado visualiza los datos de la venta cuyo identificador recibe:

```
CREATE OR REPLACE PROCEDURE VER_VENTA (ID NUMBER) AS
  IMPORTE NUMBER;
  TOTAL_V NUMBER;
  CLI TIP_CLIENTE := TIP_CLIENTE(NULL, NULL, NULL, NULL, NULL);
  FEC DATE;
--cursor para recorrer la tabla anidada del idventa
--que se recibe, recorre las líneas de venta
  CURSOR C1 IS
```

```
  SELECT NUMEROLINEA LIN, DEREF(IDPRODUCTO) PROD, CANTIDAD
  FROM THE
    (SELECT T.LINEAS FROM TABLA_VENTAS T WHERE IDVENTA = ID);

BEGIN
--obtener datos de la venta
  SELECT DEREF(IDCLIENTE), FECHAVENTA, V.TOTAL_VENTA()
         INTO CLI, FEC, TOTAL_V
  FROM TABLA_VENTAS V WHERE IDVENTA = ID;

  DBMS_OUTPUT.PUT_LINE('NÚMERO DE VENTA: ' || ID ||
                       ' * Fecha de venta: ' || FEC);

  DBMS_OUTPUT.PUT_LINE('CLIENTE: ' || CLI.NOMBRE);
  DBMS_OUTPUT.PUT_LINE('DIRECCION: '|| CLI.DIREC.CALLE);
  DBMS_OUTPUT.PUT_LINE('==========================================');

  FOR I IN C1 LOOP
    IMPORTE:= I.CANTIDAD * I.PROD.PVP;
    DBMS_OUTPUT.PUT_LINE(I.LIN|| '*' || I.PROD.DESCRIPCION || '*' ||
                I.PROD.PVP || '*' || I.CANTIDAD || '*' ||IMPORTE);
  END LOOP;
  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;
/

NÚMERO DE VENTA: 2 * Fecha de venta: 10/03/16
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 4.9

Crea una función almacenada que reciba un identificador de venta y retorne el total de la venta. Comprueba si la venta existe, si no existe devuelve -1. Realiza un bloque PL/SQL anónimo que haga uso de la función.

Realiza una consulta que muestre por cada producto el total de unidades vendidas, debe mostrar el identificador del producto, la descripción y la suma de las unidades vendidas.

4.3. BASES DE DATOS ORIENTADA A OBJETOS

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 usados 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 hardware y software.
13. Debe proporcionar un método de consulta sencillo.

¡¡INTERESANTE!!

The Object-Oriented Database System Manifesto: En este trabajo se intenta definir un sistema de base de datos orientada a objetos. En él se describen los principales rasgos y características que un sistema debe tener para calificarle como un sistema de base de datos orientado a objetos:

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**) 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 objeto 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 Linea 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 pertenece 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,* 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

Los valores pueden ser comparados usando los siguientes operadores:

= Igual a

> 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.

Ejemplos:

- 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 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*) y 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 o incluirlo en nuestro proyecto Eclipse o NetBeans. 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.printf("%d Jugadores: %n", objects.size());

      int i = 1;
      // visualizar los objetos
      while(objects.hasNext()){
         Jugadores jug = objects.next();
         System.out.printf("%d: %s, %s, %s %n",
                     i++, jug.getNombre(), jug.getDeporte(),
                     jug.getCiudad(), jug.getEdad());
      }
     odb.close(); // Cerrar BD
    }
```

ACTIVIDAD 4.10

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 inserte 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). El explorador también se puede abrir haciendo doble clic en el fichero **neodatis-odb-1.9.30.689.jar**.

En primer lugar 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*.

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.printf("%s, %s, %s, %d %n", jug.getNombre(),
```

```
                    jug.getDeporte(), jug.getCiudad(), jug.getEdad());
    odb.close(); // Cerrar BD
  }
}
```

Figura 4.3. Explorador de objetos de NeoDatis.

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*. Por 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()***:

```
//obtiene solo el 1º
Jugadores j = (Jugadores) odb.getObjects(query).getFirst();
```

Este método lanza la excepción **IndexOutOfBoundsException** si no localiza ningún objeto con ese criterio. Para capturarla pondríamos lo siguiente:

```
try{
    Jugadores j = (Jugadores) odb.getObjects(query).getFirst();
}catch(IndexOutOfBoundsException e){
    System.out.printf("OBJETO NO LOCALIZADO");
 }
```

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 4.11

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()***. Los cambios que se realicen en la base de datos se validarán utilizando el método ***commit()***, aunque también se validarán cerrando la base de datos con el método ***close()***.

El siguiente ejemplo cambia el deporte de *María* a *vóley-playa*:

```
IQuery 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.commit();     // 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);
```

Consulta la página *http://neodatis-odb.wikidot.com/criteria-queries* para saber más sobre la API **CriteriaQuery**.

Ejemplos:

- El siguiente método visualiza los jugadores de 14 años de los países de IRLANDA, FRANCIA e ITALIA.

```
private static void jugadores14irlandafranciaitalia() {
  ODB odb = ODBFactory.open("EQUIPOS.DB");
  ICriterion criterio = new And().add(Where.equal("edad", 14))
      .add(new Or().add(Where.equal("pais.nombrepais", "IRLANDA"))
                   .add(Where.equal("pais.nombrepais", "ITALIA"))
                   .add(Where.equal("pais.nombrepais", "FRANCIA")));

  IQuery query = new CriteriaQuery(Jugadores.class, criterio);
  Objects jugadores = odb.getObjects(query);

  if (jugadores.size() == 0) {
      System.out.println(" No existen jugadores de 14 años de  IRLANDA,
                          ITALIA, FRANCIA.");
  } else {
      Jugadores jug;
      System.out.println("Jugadores de 14 años de IRLANDA, ITALIA,
                         FRANCIA.");
      while (jugadores.hasNext()) {
       jug = (Jugadores) jugadores.next();
       System.out.printf("Nombre: %s, Edad: %d, Ciudad: %s,País: %s%n",
                        jug.getNombre(), jug.getEdad(),
                        jug.getCiudad(), jug.getPais().getNombrepais());
      }
  }
  odb.close();
}
```

- El siguiente método recibe el nombre de un país y actualiza las edades de los jugadores de ese país. Suma 2 a la edad. Si no hay jugadores del país visualiza un mensaje indicándolo:

```
private static void actualizaredadjugadoresdepais(String pais) {
  ODB odb = ODBFactory.open("EQUIPOS.DB");
  ICriterion crit = Where.equal("pais.nombrepais", pais);
  IQuery query = new CriteriaQuery(Jugadores.class, crit);

  Objects jugadores = odb.getObjects(query);

  if (jugadores.size() == 0) {
      System.out.printf(" No existen Jugadores de %s,
                         no se actualiza la edad %n", pais);
  } else {
     Jugadores jug;
     System.out.printf("ACTUALIZAMOS LA EDAD DE LOS JUGADORES DE %s%n",
                       pais);
```

```
        while (jugadores.hasNext()) {
              jug = (Jugadores) jugadores.next();
              int edad = jug.getEdad() + 2;
              System.out.printf("%s %d, NUEVA: %d %n",
                                  jug.getNombre(), jug.getEdad(), edad);
              jug.setEdad(edad);
              odb.store(jug);
        }
          odb.commit();
    }
    odb.close();
}
```

ACTIVIDAD 4.12

A-Crea un método que reciba un nombre de país y un deporte y visualice los jugadores que son de ese país y practican ese deporte. Si no hay ninguno visualice que no hay jugadores.

B-Crea un método que reciba un nombre de país y lo borre de la BD Neodatis. Comprueba antes de borrar si tiene jugadores. En caso de tener jugadores asigna *null* al país de esos jugadores.

Usa la base de datos EQUIPOS.DB.

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.printf("%s, Ciudad : %s %n",
                        objectValues.getByAlias("nombre"),
                        objectValues.getByIndex(1));
}
```

Mediante los métodos ***getByAlias("alias")*** o ***getByIndex(índice)*** podemos obtener los valores de los atributos de un objeto (el índice comienza en la posición 0). 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", "alias")***; entonces a la hora de recuperar el valor del atributo usamos ***getByAlias("alias")***.

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:

```
//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.printf("Suma de edad : %d %n", value.longValue());
```

Si no se opera con los valores, sino que solo se visualiza, no es necesario convertirlos a *BigDecimal*. Ejemplos:

```
System.out.printf("Suma de edad : %.2f %n", ov.getByAlias("edad"));

//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.printf("Numero de jugadores : %d %n", 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.printf("Edad media : %.2f %n", value3.floatValue());

//MÁXIMO Y MÍNIMO - Obtiene la edad máxima y la edad mínima
//SELECT MAX(edad) edad_max , MIN(edad) edad_min FROM Jugadores
Values val4 = odb.getValues(rew ValuesCriteriaQuery(Jugadores.class)
              .max("edad", "edad_max"));
ObjectValues ov4 = val4.nextValues();
BigDecimal maxima = (BigDecimal) ov4.getByAlias("edad_max");

Values val5 = odb.getValues(new ValuesCriteriaQuery(Jugadores.class)
              .min("edad", "edad_min"));
ObjectValues ov5 = val5.nextValues();
BigDecimal minima = (BigDecimal) ov5.getByAlias("edad_min");

System.out.printf("Edad máxima: %d, Edad minima: %d %n",
                  maxima.intValue(), minima.intValue());
```

Si la media sale redondeada se ejecuta bien la función *avg*, pero si la media no sale redondeada hay que capturar la excepción **Arithmeticexception**, y hacer la media con la suma y el contador. Este ejemplo realiza un método para calcular la media de edad capturando la excepción:

```
private static void visualizarmediadeedad() {
  ODB odb = ODBFactory.open("EQUIPOS.DB");
  Values val;
```

```
  ObjectValues ov;
  try {
      val = odb.getValues(new
              ValuesCriteriaQuery(Jugadores.class).avg("edad"));
      ov = val.nextValues();
      System.out.printf("AVG-La media de edad es: %.2f %n",
                          ov.getByIndex(0));
   } catch (ArithmeticException e) {
      System.out.println(e.getMessage());

      Values val2 = odb.getValues
             (new ValuesCriteriaQuery(Jugadores.class)
                       .sum("edad").count("edad"));

      ObjectValues ov2 = val2.nextValues();
      float media;
      BigDecimal sumaedad = (BigDecimal) ov2.getByIndex(0);
      BigInteger cuenta = (BigInteger) ov2.getByIndex(1);

      media = sumaedad.floatValue() / cuenta.floatValue();
      System.out.printf("La media de edad es: %.2f Contador = %d  "
                  + "suma = %.2f %n", media, cuenta, sumaedad);
   }
   odb.close();
}
```

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.printf("%s, %d%n", 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 4.10), 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 cómo 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.printf("Nombre: %s, Edad: %d, Pais: %s %n",
                objectValues.getByAlias("nombre"),
                objectValues.getByIndex(1),
                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 ciudad de los jugadores cuyo país es ESPAÑA y edad igual a 15 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"));
```

En este método visualizamos el número de jugadores, la edad máxima y la edad media por cada país. Para calcular la edad media utilizamos la suma y el contador, así evitamos el error de la media redondeada:

```
private static void contadorymediaporpais() {
   ODB odb = ODBFactory.open("EQUIPOS.DB");
   System.out.println("Numero de jugadores por país, "+
                      " max de edad y media de edad: ");
   Values groupby = odb.getValues(new ValuesCriteriaQuery(
         Jugadores.class,Where.isNotNull("pais.nombrepais"))
             .field("pais.nombrepais").count("nombre")
              .max("edad").sum("edad").groupBy("pais.nombrepais") );

  if (groupby.size() == 0)
      System.out.println( " La consulta no devuelve datos. ");
  else
  {
    while(groupby.hasNext()){
      ObjectValues objetos= (ObjectValues) groupby.next();
      float media = ((BigDecimal) objetos.getByIndex(3)).floatValue() /
             ((BigInteger) objetos.getByIndex(1)).floatValue();
      System.out.printf("Pais: %-8s Num jugadores: %d,  Edad Máxima:
            %.0f, Suma de Edad: %.0f, Edad media: %.2f %n",
            objetos.getByAlias("pais.nombrepais"),
            objetos.getByIndex(1),
            objetos.getByIndex(2),
            objetos.getByIndex(3), media );
    }
  }
```

```
  odb.close();
}
```

ACTIVIDAD 4.13

Realiza un método que reciba un nombre de país y visualice el número de jugadores que tiene por cada ciudad de ese país, y la media de edad. Si no tiene jugadores que visualice que el país no tiene jugadores. Y si el país no existe que visualice que el país no existe. Evita el error del redondeado de la media.

4.4.2.1 Crear una BD Neodatis a partir de un modelo relacional

En este apartado vamos a ver como crear una BD Neodatis, a partir de los datos de las tablas de un modelo relacional. Disponemos de las siguientes tablas en MySQL:

- ***C1_Centros***: con información de los centros de una red de centros. Cada centro tiene un profesor que es el director del centro. Un centro tiene muchos profesores.
- ***C1_Profesores***: con información de los profesores de los centros. El profesor pertenece a un centro. Los profesores imparten asignaturas.
- ***C1_Asigprof:*** con información de los profesores y las asignaturas que imparten. Un profesor imparte varias asignaturas. Una asignatura puede ser impartida por varios profesores.
- ***C1_Asignaturas:*** con información de las asignaturas y su nombre.El modelo de datos y las relaciones se muestran en la figura:

Figura 4.4. Modelo de datos BD MySql, profesores.

Deseamos crear la BD Neodatis *profesasig.neo* con las siguientes clases:

```
public class C1Asignaturas {
    private String codAsig;
    private String nombreAsi;
    private Set<C1Profesores> setprofesores;
. . . . . . . . . . . . . . . }

public class C1Centros
      private Integer codCentro;
      private String nomCentro;
```

```
        private ClProfesores director;
        private String direccion;
        private String localidad;
        private String provincia;
        private Set<ClProfesores> setprofesores;
. . . . . . . . . . . . . . . }
public class ClProfesores {
        private Integer codProf;
        private String nombreApe;
        private String nombreEspe;
        private Date fechaNac;
        private String sexo;
        private ClCentros clCentros;
. . . . . . . . . . . . . . . }
```

- La clase ***ClAsignaturas*** contendrá un set con los profesores que imparten esa asignatura.
- La clase ***ClCentros*** contendrá un set con los profesores de ese centro y un objeto profesor que es el director.
- La clase ***ClProfesores*** contendrá un objeto con los datos de su centro.

Para la solución se hace lo siguiente:

Se crea el atributo de clase ***static ODB bd.***

Y desde el método *main()*, creamos la conexión con MySQL, abrimos la BD Neodatis y llamaremos a los distintos métodos para crear las clases de Neodatis:

```
public static void main(String[] args) {
try {
        Class.forName("com.mysql.jdbc.Driver");
        Connection conexion = DriverManager.getConnection
            ("jdbc:mysql://localhost/profes", "root", "");
        bd = ODBFactory.open("profesasig.neo");
        // Recorrer Clasignaturas y guardar en Neodatis
        InsertarAsignaturas(conexion);
        // Recorrer ClCentros y guardar en Neodatis
        InsertarCentros(conexior);
        // Recorrer ClProfesores y guardar en Neodatis
        InsertarProfesores(conexion);
        // Llenar el set de profesores de asignaturas, por cada objeto
        // asignatura hacemos la select
        llenarSetProfesAsignaturas(conexion);
        // Llenar el set de profesores de Centros y el director
        llenarSetProfesEnCentrosYDirector(conexion);
        conexion.close();
        bd.close();
        } catch (ClassNotFoundException cn) {
                cn.printStackTrace();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}
```

Los métodos son los siguientes:

- **Métodos para comprobar que los objetos existen ya en la BD Neodatis**, se trata de no duplicar objetos si ejecutamos varias veces el programa. Los métodos recibirán el código de asignatura, de profesor y de centro, y devolverán true si existe y false si no existe, los métodos son estos:

```
private static boolean comprobarasig(String cod) {
  try {
        IQuery consulta = new CriteriaQuery(ClAsignaturas.class,
                        Where.equal("codAsig", cod));
        ClAsignaturas obj = (ClAsignaturas)
               bd.getObjects(consulta).getFirst();
        return true;
      } catch (IndexOutOfBoundsException e) {
        return false;}
}

private static boolean comprobarcentro(int cod) {
  try {
        IQuery consulta = new CriteriaQuery(ClCentros.class,
             Where.equal("codCentro", cod));
        ClCentros obj = (ClCentros) bd.getObjects(consulta).getFirst();
        return true;
      } catch (IndexOutOfBoundsException e) {
            return false;}
}

private static boolean comprobarprofe(int cod) {
try {
      IQuery consulta = new CriteriaQuery(ClProfesores.class,
              Where.equal("codProf", cod));
      ClProfesores obj = (ClProfesores)
             bd.getObjects(consulta).getFirst();
      return true;
      } catch (IndexOutOfBoundsException e) {
                return false;}
}
```

- **Métodos para insertar los objetos**, se van a crear tres métodos, uno para insertar las asignaturas, otro para insertar los centros y otro para insertar los profesores. Antes de insertar en la BD Neodatis se comprobará si los objetos existen en la base de datos. Al insertar las asignaturas y los centros el set de profesores se creará vacío, y una vez que se inserten los profesores, se llenarán éstos *set*. Al insertar los profesores sí que se cargará el objeto centro, ya que los centros se habrán insertado en la BD Neodatis. Los métodos son los siguientes:

```
private static void InsertarAsignaturas(Connection conexion) {
try {
   Statement sentencia = (Statement) conexion.createStatement();
   ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM cl_asignaturas");
   while (resul.next()) {
    if (comprobarasig(resul.getString(1)) == false) {
     HashSet<ClProfesores> setprofesores = new HashSet<ClProfesores>();
     ClAsignaturas ass = new ClAsignaturas(resul.getString(1),
```

```
        resul.getString(2), setprofesores);
     bd.store(ass);
     System.out.println("Asignatura grabada " + resul.getString(1));
     } else
      System.out.println("Asig: "+ resul.getString(1)+ ", EXISTE.");
   }
      bd.commit();
      resul.close();sentencia.close();
} catch (SQLException e) {e.printStackTrace();}
}

private static void InsertarCentros(Connection conexion {
try {
    Statement sentencia = (Statement) conexion.createStatement();
    ResultSet resul = sentencia.executeQuery
        ("SELECT * FROM c1_centros");
    while (resul.next()) {
      if (comprobarcentro(resul.getInt(1)) == false) {
         HashSet<C1Profesores> setprofesores = new
             HashSet<C1Profesores>();
         C1Centros cen = new C1Centros(resul.getInt(1),
            resul.getString(2), null, resul.getString(4),
            resul.getString(5), resul.getString(6), setprofesores);
         bd.store(cen);
         System.out.println("Centro grabado " + resul.getInt(1));
       } else
         System.out.println("Centro: " +rcoul.getInt(1)+ ", EXISTE.");
      }
      bd.commit();
      resul.close();sentencia.close();
} catch (SQLException e) {e.printStackTrace();}
}

private static void InsertarProfesores(Connection conexion) {
try {
      Statement sentencia = conexion.createStatement();
      ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM c1_Profesores");
      while (resul.next()) {
        if (comprobarprofe(resul.getInt(1)) == false)
         {  IQuery consulta = new CriteriaQuery(C1Centros.class,
               Where.equal("codCentro", resul.getInt(6)));
            //Cargamos el centro del profesor
            C1Centros cen = (C1Centros)
              bd.getObjects(consulta).getFirst();
            C1Profesores nueprof = new C1Profesores(
               resul.getInt(1), resul.getString(2), resul.getString(3),
               resul.getDate(4), resul.getString(5), cen);
            bd.store(nueprof);
            System.out.println("Profe grabado " + resul.getInt(1));
          } else
            System.out.println("Profe: "+resul.getInt(1)+", EXISTE."),
      }
      bd.commit();
      resul.close();sentencia.close();
```

```
} catch (SQLException e) {e.printStackTrace();  }
}
```

- **Finalmente se crean los métodos para cargar los set de profesores.** Lo que se hace es recorrer todos los objetos de la clase correspondiente (*ClAsignaturas* o *ClCentros*) y se hace SELECT a la BD relacional, seleccionando los profesores que impartan la asignatura, o que pertenezcan al centro. Y luego se busca el objeto profesor en la BD Neodatis que se corresponda con los códigos de profesor devueltos por la SELECT, y se añade al set de registros. Los métodos son estos:

```
private static void llenarSetProfesAsignaturas(Connection conexion)
throws SQLException {
  Objects<ClAsignaturas> objects = bd.getObjects(ClAsignaturas.class);
  while (objects.hasNext()) {
     ClAsignaturas asi = objects.next();
     HashSet<ClProfesores> setprofesores=new HashSet<ClProfesores>();
     Statement sentencia = conexion.createStatement();
     ResultSet resul = sentencia.executeQuery
         ("SELECT * FROM cl_asigprof where cod_asig = '" +
                                    asi.getCodAsig() + "'");
     while (resul.next()) {
       IQuery consulta = new CriteriaQuery(ClProfesores.class,
            Where.equal("codProf", resul.getInt(2)));
       //Cargo el objeto profesor
       ClProfesores obj = (ClProfesores)
           bd.getObjects(consulta).getFirst();
       //Lo añado al set de profesores
       setprofesores.add(obj);
     }
     //Asigno el set a la asignatura
     asi.setSetprofesores(setprofesores);
     bd.store(asi);
     resul.close();sentencia.close();
  }
  bd.commit();
}

private static void llenarSetProfesEnCentrosYDirector(Connection
conexion) throws SQLException {
      Objects<ClCentros> objectscen = bd.getObjects(ClCentros.class);
      while (objectscen.hasNext()) {
        ClCentros cee = objectscen.next();
        HashSet<ClProfesores> setprofesores = new
             HashSet<ClProfesores>();
        Statement sentencia = conexion.createStatement();
        ResultSet resul = sentencia.executeQuery
           ("SELECT * FROM cl_profesores where cod_centro=" +
              cee.getCodCentro());
        while (resul.next()) {
            IQuery consulta = new CriteriaQuery(ClProfesores.class,
                 Where.equal("codProf", resul.getInt(1)));
            ClProfesores obj = (ClProfesores)
                     bd.getObjects(consulta).getFirst();
            setprofesores.add(obj);
          }
```

```
        //Asigno el set al centro
        cee.setSetprofesores(setprofesores);
        // Localizo al director.
        sentencia = conexion.createStatement();
        resul = sentencia.executeQuery
           ("SELECT director FROM cl_centros where cod_centro=" +
                 cee.getCodCentro());
      if (resul.next()) {
           IQuery consulta = new CriteriaQuery(ClProfesores.class,
                Where.equal("codProf", resul.getInt(1)));
       try {
             ClProfesores obj = (ClProfesores)
                   bd.getObjects(consulta).getFirst();
             cee.setDirector(obj);
           } catch (IndexOutOfBoundsException ee) {
                System.out.println("Centro  " + cee.getCodCentro() +
                    ", Sin Director, es null.");}
      }
      bd.store(cee);
      resul.close();sentencia.close();
   }
   bd.commit();
}
```

ACTIVIDAD 4.14

Dada la BD Neodatis con nombre ARTICVENTAS.DAT (se adjunta código para crear la BD. en los recursos del tema), cuya estructura de clases es la siguiente:

```
public class Articulos {
      private int codarti;
      private String denom;
      private int stock;
      private float pvp;
      private Set<Ventas> Compras;
............. }
```

```
public class Clientes {
private int numcli ;
private String nombre;
private String pobla;
............. }

public class Ventas {
      private int codventa;
      private Clientes numcli ;
      private int univen;
      private String fecha;
............. }
```

Donde: la clase *Artículos* contiene un set con las *Ventas* de ese artículo, y el cliente que compró el artículo. Se pide leer los datos de esa base de datos y obtener un listado con las ventas de cada artículo. Para cada artículo hay que calcular:

SUMA DE UNIDADES VENDIDAS (SUMA_UNIVEN) es la suma de las unidades vendidas del artículo, las unidades vendidas se encuentran en el set de Ventas de cada artículo.

SUMA DE IMPORTE (SUMA_IMPORTE), es el resultado de multiplicar la suma de las unidades vendidas * el PVP del artículo.

NÚMERO DE VENTAS (NUM_VENTAS), es el contador de ventas del artículo, es decir el número de elementos que aparecen en el Set.

También se desea obtener una línea de totales con la suma de todas las columnas numéricas. La salida del informe debe ser similar a la que se muestra en la Figura 4.5.

CODART	DENOMINACION	STOCK	PVP	SUMA_UNIVEN	SUMA_IMPORTE	NUM_VENTAS
1	Mesas	30	100.5	12	1206,00	3
2	Pupitres	10	150.7	9	1356,30	2
6	Cuadernos	100	4.5	3	13,50	1
8	Tabletas	10	175.4	15	2631,00	4
9	Bolígrafos	100	3.5	19	66,50	5
10	Lapiceros	300	2.5	0	0	0
14	Sillas	30	120.5	0	0	0
16	Portátil	25	400.5	0	0	0
17	Espejo baño	20	100.5	0	0	0
18	Reloj cocina	10	20.7	0	0	0
20	Tarjetero	50	18.5	0	0	0
22	Estuches	110	20.4	0	0	0
23	Libro BD	10	23.5	0	0	0
24	Tijeras	30	5.0	0	0	0
25	Cubiertos	130	10.5	0	0	0
26	Teclado	25	40.5	0	0	0
Totales		990	1198.7	58	5273,30	15

Figura 4.5. Visualización de los datos de la BD Neodatis *ARTICVENTAS.DAT*

4.4.3. Modelo cliente/servidor de la base de datos

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, usamos el método ***openServer(puerto)***. 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 *neodatistServer.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/neodatisServer.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, usamos el método ***openClient()***. 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 que forman la base de datos 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 ***commit()*** para confirmar los cambios y ***close()*** para cerrar la base de datos. Para estar seguros de cerrarla, es una buena práctica utilizar un bloque **try / finally** (en este caso tendremos que cerrar el cliente y el servidor):

```
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 referirse a ella
      server.addBase("base", "D:/UNI4/Server/neodatisServer.test");
      // Se ejecuta el servidor
      server.startServer(true);

      // Se abre la BD
      odb = server.openClient("base");

      // Se llama al método que visualiza 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.printf("%2d.  %s, %s, %s, %d %n",
                     (i++), jug.getNombre(), jug.getDeporte(),
                     jug.getCiudad(), jug.getEdad());
        }
    }
}// fin clase
```

Para probarlo creamos un nuevo proyecto Eclipse (o NetBeans) con las clases *EjemploNeodatisCliente* y *Jugadores*, esta última implementando **Serializable**. Creamos de nuevo la base de datos Neodatis (la llamamos *neodatisServer.test*) y la llenamos con varios jugadores. No hemos de olvidar añadir el JAR (**neodatis-odb-1.9.30.689.jar**) a nuestro proyecto. Ejecutamos *EjemploNeodatisCliente*, la salida que se muestra es la siguiente:

```
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]
```

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 EjemploNeodatisServer{
  public static void main(String[] args) {
      ODBServer server = null;
      //Crea el servidor en el puerto 8000
      server = ODBFactory.openServer(8000);
      //Abrir BD
      server.addBase("base1", "D:/UNI4/Server/neodatisServer.test");
      // Se inicia el servidor ejecutándose en segundo plano
      server.startServer(true);
      System.out.println("Servidor iniciado....");
   }
}
```

Al ejecutarlo se muestra la siguiente pantalla:

```
Servidor iniciado....
NeoDatis ODB Server [version=1.9.30 - build=689-10-11-2010-08-21-21]
running on port 8000
```

```
Managed bases: [base1]
```

Por otro lado tenemos el código cliente que se ejecutará en otra máquina virtual, en este ejemplo se insertará un nuevo jugador en la base de datos:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;

public class EjemploClienteInsertar {
  public static void main(String[] args) {
      ODB odb = null;
      try{
        odb = ODBFactory.openClient("localhost", 8000, "base1");
        Jugadores j4 = new Jugadcres("Andrea", "padel",
                                     "Guadalajara", 14);
        odb.store(j4);
        odb.commit();
        System.out.println("Jugador Insertado...");
      } finally {
         if (odb != null) {
             odb.close();
         }
      }
  }// --main
}
```

El siguiente cliente visualiza los datos de la BD:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

public class VerBDNeodatisServer {
   public static void main(String[] args) {
     ODB odb = null;
     try {
         odb = ODBFactory.openClient("localhost", 8000, "base1");
         Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
         System.out.printf("%d Jugadores: %n", objects.size());
         int i = 1;

         // visualizar los objetos
         while (objects.hasNext()) {
            Jugadores jug = objects.next();
            System.out.printf("%d: %s, %s, %s %n", i++,
                   jug.getNombre(), jug.getDeporte(),
                   jug.getCiudad(), jug.getEdad());
         }

     } finally {
         if (odb != null)
            odb.close();
     }
   }//main
}
```

COMPRUEBA TU APRENDIZAJE

1. 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:

e) Extienden el modelo de datos relacional proporcionando tipos de datos complejos y la programación orientada a objetos.
f) No permiten el uso de SQL como lenguaje de consulta, ya que hay tipos complejos de datos.
g) Respuestas a) y b) correctas.
h) Ninguna de las anteriores.

C) ¿Cuál de estas afirmaciones es correcta?

i) Los lenguajes de consulta relacionales como SQL necesitan ser extendidos para trabajar con tipos de datos orientados a objetos.
j) En las BBDD relacionales orientadas a objetos se encuentran extensiones orientadas a objetos, como los tipos de datos definidos por el usuario.
k) En las BBDD relacionales orientadas a objetos no se encuentran extensiones sobre las tablas anidadas.
l) Respuestas a) y b) correctas.

2. Partiendo de las tablas EMPLEADOS Y DEPARTAMENTOS, crea un tipo departamento T_DEP; los atributos para este tipo son como las columnas de la tabla DEPARTAMENTOS. Crea después la tabla DEPART (con clave primaria) del tipo creado anteriormente. Llena esta tabla a partir de los datos de la tabla DEPARTAMENTOS. Crea una tabla llamada EMPLE, las columnas como las de la tabla EMPLEADOS, salvo la columna DEPT_NO que es de tipo T_DEP. Llena la tabla a partir de EMPLEADOS y DEPARTAMENTOS.

3. Crea las siguientes tablas y tipos:

```
CREATE TABLE CUENTAS (
     NUMCTA   NUMBER(5) primary key,
     DATOS   PERSONA,
SALDO NUMBER(7,2)
);
CREATE OR REPLACE TYPE T_MOVIM AS OBJECT (
     IMPORTE NUMBER(7,2),
     TIPOMOV CHAR, -- I ingreso, R reintegro
     FECHA DATE
);
 /
CREATE TABLE MOVIMIENTOS (
```

```
  NUMCTA    NUMBER(5) REFERENCES CUENTAS,
  MOV       T_MOVIM,
  CONSTRAINT PK_MOV PRIMARY KEY(NUMCTA, MOV.FECHA)
);
```

Y realiza los siguientes ejercicios:

1. Inserta datos en las tablas CUENTAS y MOVIMIENTOS. Asigna el valor 0 al saldo.
2. Realiza una consulta que muestre el nombre de la cuenta, la suma de ingresos y la suma de reintegros.
3. Modifica el saldo de la cuenta. Debe contener los ingresos menos los reintegros.

4. Crea un tipo VARRAY para guardar las ventas de artículos por trimestre. Llama al tipo **TOTAL_TRIM**, de 4 elementos de tipo numérico. Crea la tabla TARTÍCULOS, la columna VENTAS es del tipo creado anteriormente:

```
CREATE TABLE TARTICULOS (
   COD_ART NUMBER(4) PRIMARY KEY,
   DENO  VARCHAR2(15),
   PVP NUMBER(5),
   VENTAS TOTAL_TRIM
);
```

Crea la tabla TVENTAS que contiene las ventas de cada artículo en el 2012:

```
CREATE TABLE TVENTAS :
   COD_ART NUMBER(4) REFERENCES TARTICULOS,
   FECHA  DATE,
   UNIDADES NUMBER(4)
);
```

Inserta varios artícubs y varias ventas.

Realiza un procedimiento almacenado que actualice la tabla TARTICULOS, se han de actualizar las ventas trimestrales, es decir, el VARRAY VENTAS. El procedimiento debe calcular la suma de las unidades vendidas en cada trimestre por cada artículo, e ir actualizando la tabla.

Visualiza después el contenido de la tabla artículos actualizada:

```
CODART DENOMINACION PVP  UNI-1T UNI-2T UNI-3T UNI-4T TOTAL UNIS
------ ------------ ---- ------ ------ ------ ------ -----------
```

5. Partimos de las tablas EMPLEADOS, DEPARTAMENTOS y el tipo T_DEP creado en el ejercicio 2:

   1- Crea un tipo empleado T_EMPLE, los atributos son los de la tabla EMPLEADOS. Pero la columna DEPT_NO es de tipo T_DEP.
   2- Crea un tipo VARRAY para 20 elementos de tipo T_EMPLE, le damos el nombre V_EMPLE.
   3- Crea una tabla, de nombre TAB_DEP, donde una columna es del tipo V_EMPLE y la otra es de tipo T_DEP .
   4- Inserta filas en esa tabla a partir de las tablas EMPLEADOS y DEPARTAMENTOS, cada fila de la tabla contendrá el departamento y un VARRAY con sus empleados. Puedes realizar un procedimiento para insertar las filas.

5- Haz un bloque PL-SQL que usando use la tabla TAB_DEP y muestre por cada departamento el APELLIDO y SALARIO de sus empleados. Muestra también el nombre del departamento.

6. 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.
- ¿Cual 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' ) ));
```

7. Disponemos de las siguientes tablas referentes a piezas que suministran los proveedores a determinados proyectos:

Figura 4.6. Modelo de datos Ejercicio 7.

1- Crea un tipo proyectos, piezas y proveedores con el mismo formato que las tablas expuestas.

2- Crea un tipo gestión con tres referencias, una a un tipo proyectos, otra a un tipo piezas y una tercera a un tipo proveedor; y la cantidad.

3- Crea la tabla gestión del tipo gestión definido anteriormente. Llena la nueva tabla con los datos de las tablas del modelo relacional. Crea todo lo que sea necesario para llenar esta nueva tabla.

4- Realiza una SELECT con la nueva tabla gestión en la que se muestre para un código de proveedor concreto las piezas que ha suministrado.

8. ¿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, etc. en su lugar se utilizan los métodos definidos en el objeto.

9. Construye a partir del modelo relacional de la Figura 4.7 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 4.7. Modelo de datos Ejercicio 9.

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, NOTA3EV, NOTAFJUN, NOTASEPT 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 solo 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.

10. 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. Define 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 visualice 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.

ACTIVIDADES DE AMPLIACIÓN

1. A partir de la BD neodatis ARTICULOS.DAT (se adjunta en los recursos de la unidad la clase para crear esta BD) en la que disponemos de las siguientes clases (sin los *getter*, *setter* y constructores):

```
Clase Articulos {
      private int codarti;
      private String denom;
      private int stock;
      private float pvp;
...................
}

Clase Ventas {
      private int codventa;
      private Articulos codarti;
      private Clientes numcli ;
      private int univen;
      private String fecha;
...................
}
```

```
Clase Clientes {
      private int numcli ;
      private String nombre;
      private String pobla;
...................

}
```

Realiza los siguientes métodos:

a. Método que visualice los datos de cada venta:

```
CODVENTA  CODARTI  DENOMINACION  NUMCLI  NOMBRE  FECHA   UNIVEN   IMPORTE
-------------------------------------------------------------------------
 Xxxx      xxxxx    xxxxxxxxxxx   xxx    xxxxxx  xxxx    xxxxxx   xxxxxxxx
 Xxxx      xxxxx    xxxxxxxxxxx   xxx    xxxxxx  xxxx    xxxxxx   xxxxxxxx
```

Donde:

Importe es el PVP del artículo por las unidades vendidas.

b. Método que visualice por cada artículo la suma de unidades vendidas, el importe, y el número de ventas en las que se ha vendido.

```
CODARTI  DENOMINACION  STOCK  PVP  SUMA_UNIVEN   SUMA_IMPORTE   NUM_VENTAS
-------------------------------------------------------------------------
 Xxxx    xxxxxxxxxxx   xxxx   xxx   xxxxxxx       xxxxxxx       xxxxxxxx
 Xxxx    xxxxxxxxxxx   xxxx   xxx   xxxxxxx       xxxxxxx       xxxxxxxx
```

Donde:

SUMA_UNIVEN es la suma de unidades vendidas de ese artículo.

SUMA_IMPORTE es la suma de los importes del artículo (PVP del artículo por las unidades vendidas).

NUM_VENTAS es el número de ventas que se han realizado del artículo.

c. Método que visualice por cada cliente este informe:

```
NUMCLI    NOMBRE     POBLACIÓN    TOTAL IMPORTE  NUM_VENTAS
-------------------------------------------------------------
 Xxxx   xxxxxxxxxxx  xxxxxxxxxxxx      xxxxxxx       xxxxxxxx
```

Donde:

TOTAL_IMPORTE es la suma de los importes del artículo de las ventas de ese cliente (PVP del artículo por las unidades vendidas).

NUM_VENTAS es el número de ventas que ha realizado el cliente.

d. Método que visualice las siguientes estadísticas:

Nombre de artículo más vendido (más número de ventas): _____
Media de importe de ventas por artículo: _____
Nombre de cliente que más ha gastado (total importe de cliente máximo): _____
Nombre de cliente con más ventas (más número de ventas): _____

2. A partir de los datos de las tablas del siguiente modelo relacional. Crea la BD Neodatis *vuelos.neo* con los datos de las tablas del modelo.

Figura 4.8. Modelo de datos de la BD MySQL Vuelos.

Las tablas del modelo son las siguientes:

- **VUELO**: contiene la información de los vuelos, su origen, destino, tipo de vuelo ('CHARTER', 'DIRECTO', 'DOMÉSTICO', 'LARGO RECORRIDO', 'LOW COST', 'REGULAR') y fecha de vuelo. La clave es IDENTIFICADOR.
- **PASAJERO**: contiene la información de los pasajeros que realizan o han realizado vuelos, la clave es COD. Los pasajeros forman el pasaje de los vuelos.
- **PASAJE**: contiene la información del pasaje de los vuelos, el código de pasajero, el número de asiento (NUMASIENTO), el PVP y la CLASE. Un vuelo tiene muchos pasajes.
- **PERSONAL**: contiene los datos del personal de la empresa, la clave es CODIGO.
- **TRIPULACION**: contiene los datos del personal que forma la tripulación de los vuelos. La clave está formada por el identificador de vuelo y código del personal.

La BD Neodatis tendrá las siguientes clases:

```
public class Vuelo {
  private String identificador;
  private String aeropuertoorigen;
  private String aeropuertodestino;
  private Set <Pasajeros> pasajeros;
```

```
public class Pasajeros {
      private short codigo;
      private String nombre;
      private String tlf;
      private String direccion;
```

```
  private Set <Tripulacion> tripulacion;
. . . .
```

CAPÍTULO 5

BASES DE DATOS NoSQL

OBJETIVOS

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

Instalar y utilizar bases de datos NoSQL (eXist y Mongo DB).

Realizar consultas a documentos y colecciones utilizando XPath y XQuery.

Instalar y utilizar la BD MongoDB.

Realizar consultas a bases de datos y sus colecciones en mongoDB.

Desarrollar aplicaciones Java con bases de datos NoSQL, accediendo a su información para consultarla y manipularla.

CONTENIDOS

Bases de datos NoSQL. Características.

Base de datos nativa XML, eXist.

Instalación eXist.

Lenguajes de consultas.

Lenguaje Java y eXist.

Estructuras JSON.

Instalación de MongoDB.

Consultas y manipulación de datos en MongoDb.

Java y MongoDB.

RESUMEN

En este capítulo se estudian bases de datos NoSQL, bases de datos que no utilizan el lenguaje SQL para hacer sus consultas, y no utilizan estructuras fijas de almacenamiento. Aprenderemos a consultar las informaciones, y a crear, eliminar y modificar datos utilizando nuevos lenguajes de consulta y nuevos modelos de almacenamiento de datos basados en documentos como son el XML y JSON.

5.1. INTRODUCCIÓN

Las bases de datos NoSQL son aquellas que no siguen el modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS). Se caracterizan porque no usan el SQL como lenguaje principal de consultas, y además, en el almacenamiento de los datos no se utilizan estructuras fijas de almacenamiento.

El término NoSQL surge con la llegada de la web 2.0, ya que hasta ese momento solo subían contenidos a la red aquellas empresas que tenían un portal, pero con la llegada de aplicaciones como Facebook, Twitter o Youtube, en las que el usuario interactúa en la web, cualquier usuario podía subir contenido, provocando así un crecimiento exponencial de los datos.

Surgen así los problemas para gestionar y acceder a toda esa información almacenada en bases de datos relacionales. Una de las soluciones, propuestas por las empresas para solucionar estos problemas de accesibilidad, fue la de utilizar más máquinas, sin embargo, era una solución cara y no terminaba con el problema.

La otra solución fue la de crear nuevos sistemas gestores de datos pensados para un uso específico, que con el paso del tiempo han dado lugar a soluciones robustas, apareciendo así el movimiento NoSQL.

Así pues, hablar de bases de datos NoSQL es hablar de estructuras que nos permiten almacenar información en aquellas situaciones en las que las bases de datos relacionales generan ciertos problemas, debido principalmente a problemas de escalabilidad y rendimiento de las bases de datos relacionales, donde se dan cita miles de usuarios concurrentes y con millones de consultas diarias. Por tanto, las bases de datos NoSQL intentan resolver problemas de almacenamiento masivo, alto desempeño, procesamiento masivo de transacciones (sitios con alto tránsito) y, en términos generales, ser alternativas NoSQL a problemas de persistencia y almacenamiento masivo (voluminoso) de información para las organizaciones.

SQL vs NoSQL

Las bases de datos relacionales focalizan su interés en la fiabilidad de las transacciones bajo el conocido principio **ACID**, acrónimo de ***Atomicity***, ***Consistency***, ***Isolation*** and ***Durability*** (Atomicidad, Consistencia, Aislamiento y Durabilidad en español):

PRINCIPIO ACID – Bases de datos relacionales	
Atomicity	Asegurar de que la transacción se complete o no, sin quedarse a medias ante fallos
Consistency	Asegurar el estado de validez de los datos en todo momento
Isolation	Asegurar indepencencia entre transacciones
Durability	Asegurar la persistencia de la transacción ante cualquier fallo

El principio ACID aporta una robustez que colisiona con el rendimiento y operatividad a medida que los volúmenes de datos crecen.

Cuando la magnitud y el dinamismo de los datos cobran importancia, el principio ACID de los modelos relacionales queda en segundo plano frente al rendimiento, disponibilidad y escalabilidad, las características más propias de las bases de datos NoSQL. Hoy en día, los modernos sistemas de datos en internet se ajustan más al también conocido principio **BASE**.

acrónimo de ***Basic Availability*** (disponibilidad como prioridad) ***Soft state*** (la consistencia de datos se delega a gestión externa al motor de la base de datos) ***Eventually consistency*** (intentar lograr la convergencia hacia un estado consistente)

PRINCIPIO BASE – Bases de datos NoSQL	
Basic Availability	Prioridad de la disponibilidad de los datos
Soft state	Se prioriza la propagación de datos, delegando el control de inconsistencias a elementos externos
Eventually consistency	Se asume que inconsistencias temporales progresen a un estado final estable

Estas informaciones han sido consultadas en las siguientes URLs:
https://www.certsi.es/blog/bases-de-datos-nosql y
http://www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf

5.2. VENTAJAS DE LOS SISTEMAS NoSQL

La gran diferencia de estas bases de datos es cómo almacenan los datos. Por ejemplo, una factura en el modelo relacional termina guardándose en 4 tablas (con 3 o 4 claves ajenas) y en NoSQL simplemente guarda la factura y no se diseña ninguna estructura por adelantado, se almacena y ya está, por ejemplo, una clave (numero de la factura) y el Objeto (la factura).

La forma de almacenamiento de información en este tipo de bases de datos ofrece ciertas ventajas sobre los modelos relacionales, a destacar las siguientes:

- ***Se ejecutan en máquinas con pocos recursos***: estos sistemas no requieren mucha programación, por lo que se pueden instalar en máquinas de un coste más reducido.
- ***Escalabilidad horizontal:*** para mejorar el rendimiento de estos sistemas simplemente se consigue añadiendo más nodos, con la única operación de indicar al sistema cuáles son los nodos que están disponibles.
- ***Pueden manejar gran cantidad de datos:*** esto es debido a que utiliza una estructura distribuida, en muchos casos mediante tablas *Hash*.
- ***No genera cuellos de botella***: el principal problema de los sistemas SQL es que necesitan transcribir cada sentencia para poder ser ejecutada, y cada sentencia compleja requiere, además, de un nivel de ejecución aún más complejo, lo que constituye un punto de entrada en común, que ante muchas peticiones puede ralentizar el sistema.

5.3. DIFERENCIAS CON LAS BASES DE DATOS SQL

Estas son las diferencias más importantes entre los sistemas NoSQL y los sistemas SQL:

- ***No utilizan SQL como lenguaje de consultas***. La mayoría de las bases de datos NoSQL evitan utilizar este tipo de lenguaje o lo utilizan como un lenguaje de apoyo. Por poner algunos ejemplos, Cassandra utiliza el lenguaje CQL, MongoDB utiliza JSON o BigTable hace uso de GQL.

- ***No utilizan estructuras fijas como tablas para el almacenamiento de los datos***. Permiten hacer uso de otros tipos de modelos de almacenamiento de información como sistemas de clave–valor, objetos o grafos.
- ***No suelen permitir operaciones JOIN.*** Al disponer de un volumen de datos tan extremadamente grande suele resultar deseable evitar los JOIN. Esto se debe a que, cuando la operación no es la búsqueda de una clave, la sobrecarga puede llegar a ser muy costosa. Las soluciones más directas consisten en desnormalizar los datos, o bien, realizar el JOIN mediante software en la capa de aplicación.
- ***Arquitectura distribuida.*** Las bases de datos relacionales suelen estar centralizadas en una única máquina o bien en una estructura máster–esclavo, sin embargo en los casos NoSQL la información puede estar compartida en varias máquinas mediante mecanismos de tablas Hash distribuidas.

5.4. TIPOS DE BASES DE DATOS NOSQL

Según el tipo o modelo escogido para almacenar los datos, las bases de datos **NoSQL** se agrupan en cuatro categorías principales:

- ***Clave/Valor.*** Los datos son almacenados y se localizan e identifican usando una clave única y un valor (un dato o puntero a los datos). Ejemplos de este tipo son: *DynamoDB, Riak, o Redis*. Amazon y Best Buy entre otros utilizan esta implementación. Se caracterizan por ser muy eficientes tanto para las lecturas como para las escrituras.
- ***Columnas.*** Parecido al modelo clave/valor, pero la clave se basa en una combinación de columna, fila y marca de tiempo que se utiliza para referenciar conjuntos de columnas (familias). Es la implementación más parecida a bases de datos relacionales. Ejemplos: *Cassandra, BigTable, Hadoop/HBase*. Compañías como Twitter o Adobe hacen uso de este modelo.
- ***Documentos.*** Los datos se almacenan en documentos que encapsulan la información en formato XML, YAML o JSON. Los documentos tienen nombres de campos auto contenidos en el propio documento. La información se indexa utilizando esos nombres de campos. Este tipo de implementación permite, además de realizar búsquedas por clave–valor, realizar consultas más avanzadas sobre el contenido del documento. Ejemplos: ***MongoDB***, *CouchDB*, ***eXist***. Un caso de uso de esta tecnología lo tenemos con Netflix, empresa que proporciona contenidos audiovisuales online.
- ***Grafos.*** Se sigue un modelo de grafos que se extiende entre múltiples máquinas. En este tipo de bases de datos, la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos para recorrerla. Es un modelo apropiado para datos cuyas relaciones se ajustan a este modelo, como, por ejemplo, redes de transporte, mapas, etc. Ejemplos: *Neo4J, GraphBase o Virtuoso*

En esta unidad estudiaremos dos bases de datos NoSQL, por un lado la base de datos ***eXist*** que almacena documentos XML, y por otro lado, la base de datos ***MongoDB*** que almacena estructuras JSON.

5.5. 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 almacena 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.
- 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 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 cómo 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 archivo ***departamentos.xml*** escribiremos en la misma línea:

```
mysql --xml -u ejemplo -p -e "select * from departamentos;" ejemplo
>departamentos.xml
```

5.5.1. Base de Datos eXist

eXist es 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, las colecciones serían como las carpetas. No es necesario que los documentos tengan una DTD o un XML Schema asociado (XSD), y dentro de una colección pueden almacenarse documentos de cualquier tipo.

En la carpeta ***eXist\webapp\WEB-INF\data*** es donde se guardan los archivos más importantes de la BD, entre ellos están:

- ***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.

5.5.1.1. 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/apps/homepage/index.html podremos descargar la última versión de la BD. En este caso se ha instalado la versión ***eXist 3.0.RC1.jar***, es necesario tener instalada la versión Java 8. Se instala utilizando el fichero JAR proporcionado en la web, que ha de invocarse desde la línea de comandos (***java -jar eXist 3.0.RC1.jar***), o bien, haciendo doble clic sobre el icono correspondiente una vez descargado, y siguiendo el asistente.

El proceso de instalación es bastante intuitivo, simplemente seguir el asistente y estar atento a los pasos 2, donde se indica el directorio para la instalación, y 4 donde se indica la password del administrador, en nuestro caso pondremos ***admin***, para acordarnos. Véanse las Figuras 5.1 y 5.2.

Figura 5.1. Paso 2 de la instalación.

Figura 5.2. Paso 4 de la instalación.

Una vez instalada, para arrancar la base de datos se puede hacer desde el acceso directo que se crea en el escritorio (si así se ha decidido en la instalación), o también desde el menú de la aplicación seleccionando (véase la Figura 5.3) ***eXist-db XML Database***. También se puede lanzar el fichero ***start.jar*** que se encuentra en la carpeta de **eXist**.

Figura 5.3. Menú de eXist.

Al lanzar la BD puede ocurrir que esta no se inicie y nos visualice una pantalla como la de la Figura 5.4. El problema ocurrirá cuando al intentar conectar con la BD, el puerto que utiliza eXist esté ocupado por otra aplicación. Exist se instala en un servidor web (***jetty***) y ocupa el puerto ***8080***, como la mayoría de servidores web (Apache, Tomcat, Lampp...)

Figura 5.4. Error de arranque de la BD.

Para resolver el problema, cambiamos el puerto. Para ello se edita el fichero de configuración: **%HOME_EXIST%/tools/jetty/etc/jetty.xml,** en la etiqueta **<Call name="addConnector">** cambiaremos el puerto, y en la propiedad **SystemProperty** de la etiqueta **<Set name="port">** escribimos un nuevo valor, por ejemplo 8083:

```
<SystemProperty name="jetty.port" default="8080"/>
```

por

```
<SystemProperty name="jetty.port" default="8083"/>
```

Al lanzar la BD también se creará un icono asociado a eXist en la barra de tareas (Figura 5.5), desde allí podremos iniciar y parar el servidor, y abrir las distintas herramientas de trabajo con esta base de datos (*Dashboard*, *eXide* y *Java Admin Client*).

Figura 5.5. Opciones de eXist desde a barra de tareas.

5.5.1.2. Primeros pasos con eXist

La base de datos, una vez arrancada, también se puede abrir desde el navegador escribiendo: http://localhost:8083/exist/.

Para hacer consultas y trabajar con la base de datos lo podemos hacer desde varios sitios:

- Desde el eXide (**Open eXide**): el eXide (véase Figura 5.6) es una de las herramientas para realizar consultas a documentos de la bd, cargar documentos externos a la BD, crear y borrar colecciones, entre otras cosas.

La URL del exide es: http://localhost:8083/exist/apps/eXide/index.html. Desde la pestaña ***directory***, se podrá navegar por las carpetas y documentos de la BD. Dentro de ***/db/apps/demo/data/*** se encuentran algunos documentos XML como *hamlet.xml*, o *macbeth.xml*. Para realizar operaciones se utilizarán los iconos de la barra de herramientas.

Figura 5.6. El eXide.

- Desde el dashboard **(Open dashboard):** el salpicadero o cuadro de instrumentos es el administrador de aplicaciones de la BD (véase Figura 5.7). Soporta aplicaciones y plugins. Las aplicaciones proporcionan su propia interfaz gráfica de usuario web, mientras que los "plugins" se ejecutan dentro del *dashboard* como los diálogos de una sola pantalla. Ejemplos de las aplicaciones son la documentación ***eXist-bd Documentation***, el entorno de consultas ***eXide - XQuery IDE,*** o la aplicación de demostración ***eXist-bd Demo Apps***. Ejemplos de plugins son el gestor de colecciones ***Collections***, o el gestor de Backups ***Backup***, o el gestor de usuarios ***User Manager***, o el gestor de paquetes ***Package Manager.*** Los plugins son más adecuados para las funciones administrativas.

  Más información en http://exist-db.org/exist/apps/doc/dashboard.xml

Figura 5.7. El dashboard.

- Desde el cliente java **(Open Java Admin Client):** el cliente es la herramienta que utilizaremos lo largo del tema para hacer las consultas. El cliente nos pedirá conexión, con el usuario y la contraseña (utilizaremos admin/admin) y hay que asegurarse de poner correctamente el puerto de la URL: ***xmldb:exist://localhost:8083/exist/xmlrpc.***

Figura 5.8. El Cliente Java.

5.5.1.3. El cliente de administración de eXist

Una vez conectado se muestra la ventana del *Cliente de Administración eXist* (Figura 5.9), 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. Además, podremos navegar por las colecciones (las carpetas) y elegir un contexto a partir de donde se ejecutarán las consultas. Igualmente si se hace doble clic en un documento este se abrirá, y también se podrán hacer cambios en los mismos.

Si pulsamos al botón (***Consultar la BD usando XPath***), aparece la ventana de consultas ***Diálogo de consulta***, desde aquí se puede elegir también, 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 *Ejecutar*, y en la inferior se muestra el resultado, también se puede ver la traza de ejecución seguida en la ejecución de la consulta. En la Figura 5.10 se muestra la ventana de consultas.

Figura 5.9. *Cliente de Administración* de eXist.

Figura 5.10. Ventana de consultas *Dialogo de Consultas* de eXist.

SUBIR UNA COLECCIÓN A LA BASE DE DATOS

Lo primero que haremos es subir una colección a la base de datos desde el cliente. Nos conectamos con *Admin*, de momento es el único usuario con el que se va a trabajar. En la ventana

del *Cliente* pulsamos al botón *Almacena uno o más ficheros en la base de datos*, y en la ventana que aparece seleccionamos la carpeta a subir y se pulsa al botón ***Seleccionar los ficheros o directorios***, véase la Figura 5.11.

Figura 5.11. Subir una colección a eXist.

Una vez pulsado, aparece una ventana de transferencia de datos, y al cerrarla, la colección con los documentos se creará en la base de datos, y se creará dentro de la colección (o carpeta) desde donde se llamó a la subida de documentos. En principio se subirán dentro del contexto ***exist:/db>***.

Para hacer consultas sobre esa colección, hacemos doble clic sobre ella para seleccionarla, y una vez dentro pulsamos al botón *Consultar la base de datos usando XPath*. Así nos aseguramos de hacer las consultas dentro de nuestra colección, ese será el contexto a utilizar. Desde el diálogo de consulta también se podrán guardar las consultas y los resultados en ficheros. Compilar y ejecutar las consultas. Y desde el historial obtendremos las consultas ya realizadas. Véase la Figura 5.12. Si la consulta se realiza mal, aparecerá una ventana Java informando del error.

Figura 5.12. Haciendo consultas en eXist.

5.5.2. 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 archivos 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.

5.5.2.1. Expresiones XPath

XPath es un lenguaje que permite seleccionar nodos de un documento XML y calcular valores a partir de su contenido. Existen varias 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 estudiaremos 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 ***ColeccionPruebas***, que contiene los documentos *departamentos.xml* y *empleados.xml*, cuyas estructuras se muestras en la Figura 5.13. Realiza desde el diálogo de consultas las siguientes consultas:

Figura 5.13. Estructuras de *departamentos.xml* y *empleados.xml*.

- /, si ejecutamos esta orden en el (***Diálogo de Consulta***) y se está dentro del contexto **/db/ColeccionPruebas** 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 *Prueba*. 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 23 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, y *departamentosnuevo.xml* con 5 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 23.

  – /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. Se mostrarán todas las etiquetas de todos los documentos, es decir, se mostrarán todos los documentos de la colección.

- ***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 nº 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.

- 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 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/xsl/xsl_functions.asp

ACTIVIDAD 5.1

Dado el documento ***productos.xml*** que está dentro de la colección ***ColeccionPruebas***, con información de datos de productos, y cuya estructura es la siguiente:

```
<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.

5.5.2.2. 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 dentro de la *ColeccionPruebas* subida en los anteriores apartados. El documento está formado por los nodos atributo: teléfono y tipo, que pertenecen al elemento departamento y salario que pertenece al elemento empleado. El documento es el siguiente:

```
<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 García</nombre>
  </empleado>
 </departamento>

</universidad >
```

El árbol del documento se muestra en la Figura 5.14., los atributos se representan con elipses.

Figura 5.14. 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 de consultas utilizando nodos atributo:

- /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 5.2

Dado el documento ***sucursales.xml*** que se encuentra dentro de la colección ***ColeccionPruebas***. 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 número, el saldo haber y el saldo debe. 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:

- Obtener los datos de las cuentas bancarias cuyo tipo sea AHORRO.

- Obtener por cada sucursal la concatenación de su código, y el número de cuentas del tipo AHORRO que tiene.
- Obtener las cuentas de tipo PENSIONES de la sucursal con código SUC3.
- Obtener por cada sucursal la concatenación de los datos, código sucursal, director, y total saldo haber.
- Obtener todos los elementos de las sucursales con más de 3 cuentas.
- Obtener todos los elementos de las sucursales con más de 3 cuentas del tipo AHORRO.
- Obtener los nodos del director y la población de las sucursales con más de 3 cuentas.
- Obtener el número de sucursales cuya población sea Madrid.
- Obtener por cada sucursal, su código y la suma de las aportaciones de las cuentas del tipo PENSIONES.
- Obtener los nodos número de cuenta, nombre de cuenta y el saldo haber de las cuentas con saldo haber mayor de 10000.
- Obtener por cada sucursal con más de 3 cuentas del tipo AHORRO, su código y la suma del saldo debe de esas cuentas.

5.5.2.3. 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 ***Diálogo de consulta*** del *Cliente de Administración de eXist*, se puede ver en la parte de *Resultados*, y dentro de la pestaña *Trace*, la traza de las consultas, y en la traza podemos observar los ejes utilizados. Véase Figura 5.15.

Figura 5.15. Ejes en la traza de ejecución de las consultas XPath.

Ejemplos con Axis XPath :

– /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().

5.5.2.4. 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 5.16. 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 5.16. Procesamiento de una consulta XQuery.

En las consultas XQuery podemos utilizar las siguientes funciones para referirnos a colecciones y documentos dentro de la BD, son las siguientes:

- **collection("/ruta")**, indicamos el camino para referirnos a una colección.
- **doc("/ruta/documento.xml")**, indicamos el camino de un documento de una colección, y el nombre del documento.

Si no indicamos esas funciones la bd busca los elementos en el contexto actual. Así por ejemplo:

1. La consulta ***collection(/ColeccionPruebas),*** devuelve el contenido de la colección de ruta absoluta */ColeccionPruebas*, es decir, visualiza todos los documentos incluidos en esa colección. Todas las consultas parten de la ruta */db*.
2. La consulta ***doc("/ColeccionPruebas/productos.xml")*** devuelve el documento *productos.xml* completo que se encuentra en *ColeccionPruebas*. Esto hace lo mismo: ***doc("/db/ColeccionPruebas/productos.xml").***

Si en una colección se prevé que pueda haber varias etiquetas raíz con el mismo nombre, es muy importante al hacer consultas sobre un documento indicar el nombre del documento utilizando la función **doc()** para referirnos a él.

Otros ejemplos de consultas XQuery utilizando estas funciones:

- La consulta ***collection(/ColeccionPruebas)/departamentos/DEP_ROW*** devuelve los nodos ***DEP_ROW*** que cuelgan de la etiqueta raíz *departamentos*, que aparezcan dentro de la colección. Buscará todos los nodos ***departamentos/DEP_ROW*** que estén dentro de la colección.
- La consulta ***collection(/ColeccionPruebas)/sucursales/sucursal[@codigo='SUC1']*** devuelve el nodo *sucursal* cuyo código es SUC1, y que se encuentra dentro de nodos *sucursales*, y dentro de la colección.
- ***doc("/ColeccionPruebas/productos.xml")/productos/produc[precio>50]/denominación*** esta consulta devuelve los productos cuyo precio sea mayor de 50. Selecciona el documento de la colección con doc, y la consulta solo se realizará para ese documento.
- ***doc("/ColeccionPruebas/universidad.xml")/universidad/departamento[@tipo="A"]***. esta consulta devuelve los departamentos de tipo A, que se encuentran en el documento universidad.xml.
- ***También podemos hacer consultas sobre ficheros XML guardados en disco.*** En este caso escribiremos la ruta completa de la ubicación del archivo. Por ejemplo, si pongo: ***doc("file:///D:/misXMLs/clientes.xml")/clientes/clien/nombre***, obtengo los nombres de los clientes del documento *clientes.xml* que se encuentran en la carpeta *misXMLs* de la unidad *D*.

En XQuery las consultas se pueden construir utilizando expresiones **FLWOR** (leído como flower), que corresponde a las siglas de ***For***, ***Let***, ***Where***, ***Order*** y ***Return***. Permite a diferencia de XPath 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 devuelven 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 salida de esta consulta es:

```
<APE_OFI>GIL ANALISTA</APE_OFI>
<APE_OFI>FERNANDEZ ANALISTA</APE_OFI>
```

En este otro ejemplo se obtienen los nodos:

```
for $emp in /EMPLEADOS/EMP_ROW
let $nom:=$emp/APELLIDO, $ofi :=$emp/OFICIO
order by $emp/OFICIO
return <APE_OFI> {($nom,' ',$ofi)} </APE_OFI>
```

La salida de esta consulta es:

```
<APE_OFI>
    <APELLIDO>GIL</APELLIDO> <OFICIO>ANALISTA</OFICIO>
</APE_OFI>
<APE_OFI>
    <APELLIDO>FERNANDEZ</APELLIDO> <OFICIO>ANALISTA</OFICIO>
</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. Si se usa la condicional, 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 $var:=data(//empleado/@salario)** o si hay que devolver el salario pondríamos **return data($var).**

Ejemplos de consultas XQuery:

`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/pruebaxquery/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>`

`for $dep in /universidad/departamento`
`let $emp:= $dep/empleado`
`let $sal:= $dep/empleado/@salario`
`let $maxi := max($dep/empleado/@salario)`
`let $emplmax:= $dep/empleado[@salario = $maxi]`
`return`
	Obtiene los nombres de departamento, los empleados que tiene y la media del salario entre etiquetas y el empleado con salario máximo:
	`<depart>Informática<numemples>2</numemples>`
	`<medsal>2150</medsal>`
	`<salariomax>2300</salariomax>`
	`<emplemax>Alicia Martín - 2300</emplemax>`
	`</depart>`
	`<depart>Matemáticas<numemples>4</numemples>`
	`<medsal>2200</medsal>`

<depart> {data($dep/nombre)} <numemples>{count($emp)} </numemples> <medsal>{avg($sal)}</medsal> <salariomax>{$maxi}</salariomax> <emplemax>{$emplmax/nombre/text( )} - {data($emplmax/@salario)} </emplemax> </depart>	<salariomax>2500</salariomax> <emplemax>Antonia González - 2500</emplemax> </depart> <depart>Análisis<numemples>2</numemples> <medsal>2050</medsal> <salariomax>2200</salariomax> <emplemax>Mario García - 2200</emplemax> </depart>

5.5.2.5. 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 lista.

- 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>
```

Y saldría así:

```
<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 5.3

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 entre 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, las 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.

5.5.2.6. 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, eso sí generando un documento XML nuevo. 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>
```

La salida sería esta:

```
<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>
```

La salida sería esta:

```
<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 la 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>')
```

La salida sería esta:

```
<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 ()
```

La salida sería esta:

```
<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 5.4

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>
```

La salida sería esta:

```
<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
```

La salida sería esta:

```
<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 ()
```

La salida sería esta:

```
<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 5.5

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. Obtén 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.

5.5.2.7. 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 EXPRESIÓN XPATH
update insert ELEMENTO following EXPRESIÓN XPATH
update insert ELEMENTO preceding EXPRESIÓN XPATH
```

`update insert`
`  <zona><cod_zona>50</cod_zona>`
`  <nombre>Madrid-OESTE </nombre>`
`  <director>Alicia Ramos Martin</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 (véase 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 o 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 xpath**

`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*

Más sobre sentencias de actualización de eXist en: http://exist-db.org/exist/apps/doc/update_ext.xml

ACTIVIDAD 5.6.

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 el 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*.

5.5.3. Acceso a eXist desde Java

Ya se ha estudiado en la UNIDAD 1 cómo 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. Estas son:

- ***API XML:DB***, cuyo objetivo es la definición de un método común de acceso a SGBD XML, permitiendo la consulta, creación y modificación de contenido. La última actualización de este trabajo es del año 2001 y la actividad desde el año 2005 es prácticamente nula. Sin embargo, proporciona clases bastante útiles para el manejo de colecciones y documentos.

- ***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. Es una api similar a JDBC para bases de datos relacionales.

5.5.3.1. La API XML:DB para bases de datos XML

La estructura de XML:DB gira en torno a los siguientes componentes básicos:

- ***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 la 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 subcolecciones. 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 cerrar. Para conectarnos a una colección lo escribimos así (en URI indicamos la colección):

```
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");
```

- ***Resource***: representa un recurso, un archivo de datos, hay dos tipos:
  - ***XMLResource***: representa un documento XML o parte de un documento obtenido con una consulta. Es el recurso que más utilizaremos en nuestros ejercicios.
  - ***BinaryResource***: que representa una secuencia de información binaria, como un mapa de bits.

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() throws XMLDBException  {
  String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
  Collection col = null; // Colección
  //Datos para la connexion a la colección URI, usuario y pasword.
  String URI =
    "xmldb:exist://localhost:8083/exist/xmlrpc/db/ColeccionPruebas";
  String usu="admin"; //Usuario
  String usuPwd="admin"; //Clave
  try {
   Class cl = Class.forName(driver);
   Database database =(Database) cl.newInstance();
   DatabaseManager.registerDatabase(database);
   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");
   System.out.println(" Se han obtenido " + result.getSize()
       + " elementos.");
   // 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();
} catch (Exception e) {
             System.out.println("Error al inicializar la BD eXist");
             e.printStackTrace();     }
}// FIN verempleados10
```

ACTIVIDAD 5.7.

- Realiza los cambios necesarios al ejercicio anterior para leer de teclado un departamento y visualizar sus empleados. Utiliza la entrada estándar.
- 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. Hacer 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 ur 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.

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 cserv = (CollectionManagementService)
        col.getService("CollectionManagementService","1.0");
cserv.createCollection("NUEVA_COLECCION");
```

- ***Borrar una colección:*** para borrar utilizamos el método ***removeCollection***:

```
cserv = (CollectionManagementService)
     col.getService("CollectionManagementService","1.0");
cserv.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 archivo 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); //Asigno el archivo
    col.storeResource(nuevoRecurso); //Lo almaceno en la colección
}
```

- ***Borrar un documento de la colección:*** este ejemplo borra el docuemto 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."); }
```

Este método visualiza el número de colecciones, los nombres de las colecciones de la base de datos, los documentos XML de las colecciones y el contenido de los documentos:

```
public static void verrecursosdelascolecciones() {
String driver = "org.exist.xmldb.DatabaseImpl";
try {
    Class cl = Class.forName(driver);
    Database database = (Database) cl.newInstance();
```

```
        DatabaseManager.registerDatabase(database);
        String URI = "xmldb:exist://localhost:8083/exist/xmlrpc/db/";
        String usu = "admin"; String usuPwd = "admin";
        Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
        System.out.println("Número de colecciones: " +
              col.getChildCollectionCount());
        //Se carga la lista de colecciones en un array de cadenas
        String[] colecciones = col.listChildCollections();
        for (int j = 0; j < colecciones.length; j++) {
          System.out.println("-------------------------- ");
          System.out.println(colecciones[j]);
          //Se extrae una colección
          Collection colecc = col.getChildCollection(colecciones[j]);
          // Se cargan los recursos de la colección en un array
          String[] lista = colecc.listResources();//Lista de recursos
          for (int i = 0; i < lista.length; i++) {
              //Se extrae el recurso y se visualiza
              Resource res = (Resource) colecc.getResource(lista[i]);
              System.out.println("ID del documento: " + res.getId());
              System.out.println("Contenido del documento:\n " +
                 res.getContent());
          }
      }
    } catch (ClassNotFoundException ex) {
       System.out.println(" ERROR EN EL DRIVER. COMPRUEBA CONECTORES.");
    } catch (IllegalAccessException ex) {
        System.out.println("Error Instancia-cl.newInstance()");
    } catch (XMLDBException ex) {ex.printStackTrace();
    } catch (InstantiationException ex) {ex.printStackTrace(); }
}
```

ACTIVIDAD 5.8.

- 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:

    Tipo 1: son actividades de libre horario, el socio no paga cuota adicional por ellas, por ejemplo: aparatos o piscina.

    Tipo 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.

    Tipo 3: representan actividades en las que se alquila un espacio, por ejemplo pádel o tenis. El socio paga una cuota adicional de 4€ por cada hora que dedique a la actividad.
  - *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. Obtén 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 con nombre ***socios_cuotaadicional.xml*** (la raíz del documento llamadla ***socios_cuotaadicional***) que calcule 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 ***socios_cuotaadicional.xml*** a la colección GIMNASIO. Una vez creado y añadido el documento, el programa Java debe recorrer ese documento para obtener por cada socio 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>yyyyyyx</cuota_total>
</datos>
```

Localizar un documento en la bd y bajar el documento a un archivo en disco:

Para localizar si existe un documento en una colección utilizaremos el método ***getResource***. En el método de ejemplo *bajardocumento*, se busca el documento ***zonas.xml*** de la colección ***BDProductosXML*** (previamente hay que subir esta colección). Si no existe, el método devuelve null, y si existe devuelve el documento. El documento devuelto lo almacenamos en un objeto **DOM** (*Modelo de Objetos de Documento*), luego se visualiza en consola y finalmente se copia a un archivo en disco de nombre ***zonas.xml***.

Se va a necesitar los siguientes *import*:

- Del paquete ***javax.xml.transform***, donde se encuentran las clases para transformar el árbol DOM al fichero XML (se utilizaron en la unidad 1):

```
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

- Del paquete ***org.w3c.dom*** la clase *Node* (`import org.w3c.dom.Node;`), necesaria para crear el documento DOM:

```
Node document = (Node) res.getContentAsDOM();
```

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

- Y de la api XMLDB se necesitan:

```
import org.xmldb.api.DatabaseManager;
import org.xmldb.api.base.Collection;
import org.xmldb.api.base.Database;
```

El método es el siguiente:

```
public static void bajardocumento() throws TransformerConfigurationException,
TransformerException {
   //Localizar un documento, extraerlo y guardarlo en disco
   String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXis
   try {
        Class cl = Class.forName(driver); //Cargar del driver
        Database database = (Database) cl.newInstance(); //Instancia de la BD
        DatabaseManager.registerDatabase(database); //Registro del driver
        String URI =
            "xmldb:exist://localhost:8083/exist/xmlrpc/db/BDProductosXML";
        String usu = "admin"; //Usuario
        String usuPwd = "admin"; //Clave
        Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
        XMLResource res = (XMLResource) col.getResource("zonas.xml");
        if (res == null) {
                System.out.println("NO EXISTE EL DOCUMENTO");
        } else {
          System.out.println("ID del documento: " + res.getDocumentId());
          //Volcado del documento a un árbol DOM
          Node document = (Node) res.getContentAsDOM();
          Source source = new DOMSource(document);
          // Volcado del documento de memoria a consola
          Transformer transformer =
            TransformerFactory.newInstance().newTransformer();
          Result console = new StreamResult(System.out);
          transformer.transform(source, console);
          //Volcado del documento a un fichero
          Result fichero = new StreamResult(new java.io.File("./zonas.xml"));
          transformer = TransformerFactory.newInstance().newTransformer();
          transformer.transform(source, fichero);
        }
      } catch (ClassNotFoundException ex) {
          System.out.println(" ERROR EN EL DRIVER. COMPRUEBA CONECTORES.");
      } catch (InstantiationException ex) {
          System.out.println("Error al crear Instancia de la BD " +
            " (Database) cl.newInstance()");
      } catch (IllegalAccessException ex) {
            System.out.println("Error al crear Instancia de la BD " +
              " (Database) cl.newInstance()");
    } catch (XMLDBException ex) {
            ex.printStackTrace();  }
}
```

Ejecutar consultas almacenadas en ficheros

El siguiente método ejecuta una consulta que se encuentra almacenada en un fichero. El fichero se llama *miconsulta.xq*, y se encuentra en la carpeta del proyecto. El método recibe como parámetro el nombre del fichero. El fichero será de texto y lo leemos hasta el final, lo vamos guardando en una cadena y una vez leído el fichero ejecutaremos la consulta.

```
public static void ejecutarconsultafichero(String fichero) {
   String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
```

```
try {
      Class cl = Class.forName(driver);
      Database database = (Database) cl.newInstance();
      DatabaseManager.registerDatabase(database);
      String URI=
         xmldb:exist://localhost:8083/exist/xmlrpc/db/BDProductosXML";
      String usu = "admin"; //Usuario
      String usuPwd = "admin"; //Clave
      Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
      // Leer el fichero y guardarlo en una cadena
      System.out.println("Convirtiendo el fichero a cadena...");
      BufferedReader entrada = new BufferedReader(new FileReader(fichero));
      String linea = null;
      StringBuilder stringBuilder = new StringBuilder();
      String salto = System.getProperty("line.separator"); //El salto linea
      while ((linea = entrada.readLine()) != null) {
             stringBuilder.append(linea);
             stringBuilder.append(salto);
      }
      String consulta = stringBuilder.toString();
      System.out.println("Consulta: " + consulta);
      // Ejecutar consulta
      XPathQueryService servicio = (XPathQueryService)
                    col.getService("XPathQueryService", "1.0");
      ResourceSet result = servicio.query(consulta);
      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("Elemento: " + (String) r.getContent());
      }
     } catch (ClassNotFoundException ex) {
         System.out.println("ERROR EN EL DRIVER.");
     } catch (InstantiationException ex) {
         System.out.println("ERROR AL CREAR LA INSTANCIA.");
     } catch (IllegalAccessException ex) {
         System.out.println("ERROR AL CREAR LA INSTANCIA.");
     } catch (XMLDBException ex) {
         System.out.println("ERROR AL OPERAR CON EXIST.");
     } catch (FileNotFoundException ex) {
         System.out.println("El fichero no se localiza: " + fichero);
     } catch (IOException ex) {      ex.printStackTrace();    }
 }
```

5.5.3.2. 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 es un estándar independiente del fabricante, soporta al estándar XQuery 1.0 y muy fácil de utilizar. El inconveniente es que solo se pueden hacer consultas, no se puede operar con colecciones y documentos.

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/.

Para trabajar con esta API necesitamos los siguientes import:

```
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 una fuente física de datos a partir de la cual crear conexiones; cada implementación definirá las propiedades necesarias para efectuar la conexión, siendo básicas las propiedades user y password.

Por ejemplo, este código realiza la conexión con eXist, hay que poner el nombre del servidor (Localhost), el puerto de la BD (8083), 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", "8083");
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. Se obtiene a través de un XQDataSource, en nuestro ejemplo es ***server***. Ejemplo:

```
XQConnection conn = server.getConnection();
```

También se puede indicar el usuario y password que abre la sesión:

```
XQConnection conn = server.getConnection("admin","admin");
```

La conexión la cerramos escribiendo `conn.close();`

Clases y métodos para procesar los resultados de una consulta:

- **XQExpression**: objeto creado a partir de una conexión para la ejecución de una expresión una vez, retornando un ***XQResultsetSequence*** con los datos obtenidos, podemos decir que en un paso se evalúa el contexto estático o expresión y el dinámico. La ejecución se produce llamando al método ***executeQuery***, y se evalúa teniendo en cuenta el contexto estático en vigor.
- **XQPreparedExpression**: objeto creado a partir de una conexión para la ejecución de una expresión múltiples veces, retornando un ***XQResultsetSequence*** con los datos obtenidos, en este caso la evaluación del contexto estático solo se hace una vez, mientras que el proceso del contexto dinámico se repite. Igual que en ***XQExpression*** la ejecución se produce llamando al método ***executeQuery***, y se evalúa teniendo en cuenta el contexto estático en vigor.
- **XQDynamicContext**: representa el contexto dinámico de una expresión, como puede ser la zona horaria y las variables que se van a utilizar en la expresión.

- **XQStaticContext**: representa el contexto estático para la ejecución de expresiones en esa conexión. Se puede obtener el contexto estático por defecto a través de la conexión. Si se efectúan cambios en el contexto estático no afecta a las expresiones ejecutándose en ese momento, solo en las creadas con posterioridad a la modificación. También es posible especificar un contexto estático para una expresión en concreto, de modo que ignore el contexto de la conexión.
- **XQItem**: representación de un elemento en ***XQuery***. Es inmutable y una vez creado su estado interno no cambia.
- **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.

```
XQPreparedExpression consulta;
XQResultSequence resultado;
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));
}
```

- **XQSecuence**: representación de una secuencia del modelo de datos ***XQuery***, contiene un conjunto de 0 o más ***XQItem***. Es un objeto recorrible.
- **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));
```

Con XQJ no se necesita 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.

Partimos de que se ha subido la colección ***BDProductosXML***, y la consulta la haremos en el contexto ***/db/BDProductosXML***. Recuerda que con XQJ **NO** nos conectamos a la colección, nos conectamos a la base de datos, con lo cual a la hora de hacer las consultas indicaremos la colección o el documneto de la colección.

EJEMPLOS:

- Este método realiza una consulta al documento ***productos.xml*** de la colección ***BDProductosXML*** , y devuelve los nodos.
  Si escribimos esta consulta XQJ busca los productos en esa colección:

```
for $pr in collection('/db/BDProductosXML')/productos/produc
```

```
return $pr
```

Si escribimos esta otra consulta XQJ busca los productos en toda la base de datos y todas las colecciones. Pueden existir varios documentos con nodos ***productos/produc***.

```
for $pr in /productos/produc return $pr
```

```
public static void verproductos() {
  try {
        XQDataSource server = new ExistXQDataSource();
        server.setProperty("serverName", "localhost");
        server.setProperty("port", "8083");
        server.setProperty("user", "admin");
        server.setProperty("password", "admin");
        XQConnection conn = server.getConnection();
        XQPreparedExpression consulta;
        XQResultSequence resultado;
        System.out.println("-- Consulta documento productos.xml --");
       consulta = conn.prepareExpression("for $pr in
     collection('/db/BDProductosXML')/productos/produc return $pr");
       resultado = consulta.executeQuery();
       while (resultado.next()) {
              System.out.println("Elemento " +
              resultado.getItemAsString(null));
        }
         conn.close();
    } catch (XQException ex) {
       System.out.println("Error al operar.");      }
}
```

- Estas instrucciones devuelven el número de productos con precio mayor de 50.

```
XQPreparedExpression consulta = conn.prepareExpression(
   " count(collection('/db/BDProductosXML')
          /productos/produc[precio>50] )" );
XQResultSequence resultado = consulta.executeQuery();
resultado.next();
System.out.println("Número de productos con precio > de 50: "
                        + resultado.getInt());
 conn.close();
```

- El siguiente método ejecuta una consulta almacenada en un fichero. El fichero está en la carpeta del proyecto, y se llama ***miconsulta.xq***.

```
public static void ejecutarconsultadefichero() {
     try {
          XQDataSource server = new ExistXQDataSource();
          server.setProperty("serverName", "localhost");
          server.setProperty("port", "8083");
          server.setProperty("user", "admin");
          server.setProperty("password", "admin");
          XQConnection conn = server.getConnection();
          InputStream query;
          query = new FileInputStream("miconsulta.xq");
          XQExpression xqe = conn.createExpression();;
          XQSequence resultado = xqe.executeQuery(query);
```

```
        while (resultado.next()) {
              System.out.println(resultado.getItemAsString(null));
        }
        conn.close();
      } catch (XQException ex) {
        System.out.println("Error en las propiedades del server.");
      } catch (FileNotFoundException ex) {
        System.out.println("Error fichero.");    }
    }
```

ACTIVIDAD 5.9.

Crea los siguientes métodos utilizando la API XQJ:

1. Realiza una consulta al documento *zonas.xml* para que se visualice el nombre de zona y el número de productos de cada zona. Utiliza los documentos de la colección *BDProductosXML*.
2. Utilizando el documento *universidad.xml*, visualiza los datos de los empleados de los departamentos de tipo A.

- El siguiente método crea un archivo XML en el disco con nombre *NUEVO_EMPLE10.xml*, a partir de los datos extraídos de una consulta. La consulta devuelve los empleados del departamento 10, más el título del documento *empleados.xml*, de la colección *ColeccionPruebas*. La consulta es la siguiente:

```
let $titulo:=
   doc('/db/ColeccionPruebas/empleados.xml')/EMPLEADOS/TITULO
return  <EMPLEADOS>{$titulo}
        {for $em in doc('/db/ColeccionPruebas/empleados.xml')
           /EMPLEADOS/EMP_ROW[DEPT_NO=10]
        return $em}
      </EMPLEADOS>
```

El ejercicio nos queda así:

```
public static void creaemple10() {
 String nom = "NUEVO_EMPLE10.xml";
 File fichero = new File(nom);
 XQDataSource server = new ExistXQDataSource();
 try {
  server.setProperty("serverName", "localhost");
  server.setProperty("port", "8083");
  XQConnection conn = server.getConnection();
  XQPreparedExpression consulta = conn.prepareExpression(
       "let $titulo:= doc('/db/ColeccionPruebas/empleados.xml')"
        + "/EMPLEADOS/TITULO  "
        + " return  <EMPLEADOS>{$titulo} "
        + " {for $em in doc('/db/ColeccionPruebas/empleados.xml'')"
        + "/EMPLEADOS/EMP_ROW[DEPT_NO=10] return $em} "
        + " </EMPLEADOS> ");
  XQResultSequence result = consulta.executeQuery();
  if (fichero.exists()) { // borramos y creamos
     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));
      bw.write("<?xml version='1.0' encoding='ISO-8859-1'?>" + "\n");
      result.next();
      String cad = result.getItemAsString(null);
      System.out.println("Salida: " + cad); // visualizamos
      bw.write(cad + "\n"); // grabamos en el fichero
      bw.close(); // Cerramos el fichero el fichero
      System.out.println("Fichero Creado");
      } catch (IOException ioe) {ioe.printStackTrace();
      }
   conn.close();
 } catch (XQException e) {e.printStackTrace();}
}
```

ACTIVIDAD 5.10.

A partir del los documentòs *productos.xml* y *zonas.xml* de la colección *BDProductosXML*. Realiza un método que realice lo siguiente:

Crea un documento externo con nombre *zonas20.xml* que contenga 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.

5.5.4. Tratamiento de excepciones

En este apartado se prueban las excepciones XMLDBEXCEPTION y XQEXCEPTION, para ver cuando se producen. A lo largo de la unidad se han realizado ejemplos capturando las excepciones o bien a nivel de los métodos utilizado la palabra clave **throws**, para luego ser tratadas en main(), o bien utilizando bloques **try-catch** dentro de los métodos. En este apartado se gestionan las excepciones utilizando bloques **try-catch**.

XMLDBException

XMLDBException 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*** hereda de ***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 cómo 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"+
           "  la 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 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 yo escribo esta consulta:

```
res = service.query(
     "forr $b in /EMPLEADOS/EMP_ROW[APELLIDO='TOVAR'] return $b");
```

Visualiza este mensaje:

```
Error XMLDBException en la query: Failed to invoke method queryP in
class org.exist.xmlrpc.RpcConnection:
org.exist.xquery.StaticXQueryException: exerr:ERROR
org.exist.xquery.XPathException: err:XPST0003 unexpected token: $ [at
line 1, column 6]
```

Para saber más sobre esta excepción acceder al sitio:

http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/base/XMLDBException.html

ACTIVIDAD 5.11.

Carga esta clase en tu ordenador y prueba las distintas situaciones de error.

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.

Para saber más sobre esta excepción acceder al sitio:

http://xqj.net/javadoc/javax/xml/xquery/XQException.html

En el siguiente ejemplo vemos cómo 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 pruebaexcepcionesxqj{
 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");
        server.setProperty("port", "8083");
   }catch (XQException  e) {
   System.out.println("Error en Server. Mensaje:"+e.getMessage());
   System.out.println("Error en Server. 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/produc 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."); }
  }
}
```

ACTIVIDAD 5.12.

Carga esta clase en tu ordenador y provoca situaciones de error, escribiendo mal la consulta, o el nombre del servidor o cerrando la BD.

5.6. BASE DE DATOS MONGODB

MongoDB es un sistema de base de datos multiplataforma orientado a documentos, se podrá almacenar cualquier tipo de contenido sin obedecer a un modelo o esquema. Está escrito en C++, con lo que es bastante rápido a la hora de ejecutar sus tareas. Además, está licenciado como GNU AGPL 3.0, de modo que se trata de un software de licencia libre. Funciona en sistemas operativos Windows, Linux, OS X y Solaris.

Una de sus características principales es la velocidad y la sencilla forma que tiene para hacer consultas a los contenidos. MongoDB se utiliza para cualquier aplicación que necesite almacenar datos semiestructurados, caso de aplicaciones CMS, aplicaciones móviles, de juegos, o plataformas e-commerce. MongoDB no soporta JOINS ni transacciones, aunque posee índices secundarios, un propio lenguaje de consulta muy expresivo, operaciones atómicas en un solo documento (pero no soporta transacciones de múltiples documentos), y lecturas consistentes.

La mayor diferencia entre las bases de datos relacionales y MongoDB es la forma en que se crea el modelo de datos, el modelo relacional es un modelo rígido y estructurado mientras que el modelo MongoDB es un modelo dinámico. En la siguiente tabla se muestra la terminología utilizada en el modelo relacional y el modelo de documento de MongoDB:

Modelo Relacional	**MongoDB**
Base de datos	Base de datos
Tabla	Colección
Fila	Documento
Columna	Campo
Índice	Índice
Join	Documento embebido o referencia

Con el modelo MongoDB se pasa de un modelo de datos rígido basado en estructuras de datos bidimensionales, formado por tablas, filas y columnas a un modelo de datos de documentos rico y dinámico con subdocumentos y matrices embebidas. En MongoDB se pueden crear colecciones sin definir su estructura, también se puede alterar la estructura de los documentos simplemente añadiendo nuevos campos o borrando los ya existentes. Esta característica convierte a Mongo en una BD muy flexible con respecto a las alternativas relacionales.

MongoDB almacena documentos JSON (JavaScript Object Notation) en una representación binaria llamada BSON (Binary JSON). BSON es una serialización codificada en binario de documentos JSON, soporta todas las características de JSON e incluye los tipos de datos int, long, float o arrays. El documento (el registro en el modelo relacional) representa la unidad básica de datos en MongoDB.

5.6.1 Estructuras JSON

JSON (**J**avaScript **O**bject **N**otation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generarlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, *Standard ECMA-262 3rd Edition - Diciembre 1999*. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. (Fuente: http://www.json.org/json-es.html)

JSON está constituido por dos estructuras:

- ***Una colección de pares de nombre/valor***. En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un array asociativo.
- ***Una lista ordenada de valores***. En la mayoría de los lenguajes, esto se implementa como arrays, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON, se presentan de estas formas:

- Como un **objeto**, conjunto desordenado de ***pares nombre/valor***. Un objeto comienza con { (llave de apertura) y termine con } (llave de cierre). Cada nombre es seguido por: (dos puntos) y los pares *nombre/valor* están separados por , (coma). En el ejemplo creo un objeto persona con nombre y oficio, y un objeto zona con su código y su nombre:

```
{ "persona": {"nombre": "Alicia","oficio": "Profesora"  }}
{ "zona": {"codzona": 10,"nombre": "Madrid"  }}
```

- Un **array**, es decir, una colección de valores. Un array comienza con [(corchete izquierdo) y termina con] (corchete derecho). Los valores se separan por, (coma). En el ejemplo creo el objeto persona, un array de dos elementos, no tienen por qué tener los mismos pares ***nombre/valor***, y el objeto zona con dos zonas:

```
{
 "persona": [
  {"nombre":"Alicia","oficio":"Profesora","ciudad": "Talavera"},
  {"nombre": "María Jesús","oficio":"Profesora"}
]}

{
 "zona": [
      {"codzona": 10,"nombre": "Madrid"},
```

```
        {"codzona": 20,"nombre": "Toledo", "tasa": 15}
    ]}
```

- Un **valor** puede ser una *cadena de caracteres* con comillas dobles, o un *número*, o **true** o **false** o **null**, o un *objeto* o un *array*. Estas estructuras pueden anidarse. En el ejemplo se muestra un objeto de nombre ventana con distintos tipos de *nombre/valor*:

```
{     "ventana": {
            "titulo": "Gestión Artículos",
            "alto": 300,
            "ancho": 500,
            "menu": null,
            "modal": true,
            "botones": ["ok", "cancel"]
      }
}
```

- Una **cadena de caracteres** es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres de un único carácter. Una *cadena de caracteres* es parecida a una cadena de caracteres C o Java.
- Un ***número*** es similar a un número C o Java, excepto que **no se usan** los formatos octales y hexadecimales.

Los espacios en blanco pueden insertarse entre cualquier par de símbolos *nombre/valor*.

Ejemplos XML vs JSON:

Para comprobar si un objeto JSON está bien escrito lo validaremos desde algún ***validator***, por ejemplo, ***http://jsonlint.com/***, o ***https://jsonformatter.curiousconcept.com/***.

El documento ***departamentos.xml***

```
<departamentos>
   <TITULO>DATOS DE LA TABLA DEPART</TITULO>
   <DEP_ROW>
       <DEPT_NO>10</DEPT_NO>
       <DNOMBRE>CONTABILIDAD</DNOMBRE>
       <LOC>SEVILLA</LOC>
   </DEP_ROW>
   <DEP_ROW>
       <DEPT_NO>20</DEPT_NO>
       <DNOMBRE>INVESTIGACION</DNOMBRE>
       <LOC>MADRID</LOC>
   </DEP_ROW>
   <DEP_ROW>
       <DEPT_NO>30</DEPT_NO>
       <DNOMBRE>VENTAS</DNOMBRE>
       <LOC>BARCELONA</LOC>
   </DEP_ROW>
   <DEP_ROW>
      <DEPT_NO>40</DEPT_NO>
      <DNOMBRE>PRODUCCION</DNOMBRE>
      <LOC>BILBAO</LOC>
   </DEP_ROW>
</departamentos>
```

Podemos representarlo en JSON así:

```
{"departamentos":
 {"TITULO":"DATOS DE LA TABLA DEPART",
 "DEP_ROW":{"DEPT_NO":10,"DNOMBRE":"CONTABILIDAD","LOC":"SEVILLA"},
 "DEP_ROW":{"DEPT_NO":20,"DNOMBRE":"INVESTIGACIÓN","LOC":"MADRID"},
 "DEP_ROW":{"DEPT_NO":30,"DNOMBRE":"VENTAS","LOC":"BARCELONA"},
 "DEP_ROW":{"DEPT_NO":40,"DNOMBRE":"PRODUCCION","LOC":"BILBAO"}
 }
}
```

O también, como DEP_ROW se repite, podemos representarlo como un array:

```
{"departamentos":
  {"TITULO":"DATOS DE LA TABLA DEPART",
   "DEP_ROW":
    [
     {"DEPT_NO":10, "DNOMBRE":"CONTABILIDAD", "LOC":"SEVILLA"},
     {"DEPT_NO":20, "DNOMBRE":"INVESTIGACIÓN", "LOC":"MADRID"},
     {"DEPT_NO":30, "DNOMBRE":"VENTAS", "LOC":"BARCELONA"},
     {"DEPT_NO":40, "DNOMBRE":"PRODUCCION", "LOC":"BILBAO"}
    ]
  }
}
```

En el siguiente documento (***sucursales.xml***) podemos representar *sucursal* y *cuenta* como arrays, ya que tenemos varias sucursales, y las sucursales tienen varias cuentas. Los atributos se representan como los demás elementos (pares nombre/valor).

```
<sucursales>
   <sucursal telefono="112233" codigo="SUC1">
          <director>Alicia Gómez</director>
          <poblacion>Madrid</poblacion>
          <cuenta tipo="AHORRO">
              <nombre>Antonio García</nombre>
              <numero>123456</numero>
              <saldohaber>21000</saldohaber>
              <saldodebe>200</saldodebe>
          </cuenta>
          <cuenta tipo="AHORRO">
              <nombre>Pedro Gómez</nombre>
              <numero>1111456</numero>
              <saldohaber>12000</saldohaber>
              <saldodebe>0</saldodebe>
         </cuenta>
   </sucursal>
   <sucursal telefono="2023345" codigo="SUC2">
          <director>Fernando Rato</director>
          <poblacion>Talavera</poblacion>
          <cuenta tipo="AHORRO">
              <nombre>Marcelo Saez</nombre>
              <numero>30303036</numero>
              <saldohaber>15000</saldohaber>
              <saldodebe>12000</saldodebe>
          </cuenta>
          <cuenta tipo="AHORRO">
              <nombre>María Jesús Ramos</nombre>
              <numero>4444222</numero>
```

```
                <saldohaber>5000</saldohaber>
                <saldodebe>0</saldodebe>
            </cuenta>
    </sucursal>
</sucursales >
```

Podemos representarlo en JSON así:

```
{
   "sucursales": {
      "sucursal": [{
            "telefono": 112233,"codigo": "SUC1",
            "director": "Alicia Gómez","poblacion":"Madrid",
            "cuenta": [{
                  "tipo": "AHORRO","nombre":"Antonio García",
                  "numero": 123456,"saldohaber": 21000,
                  "saldodebe": 200
               }, {
                  "tipo": "AHORRO","nombre": "Pedro Gómez",
                  "numero": 1111456,"saldohaber": 12000,
                  "saldodebe": 0
               }]
            }, {
            "telefono": 2023345,"codigo": "SUC2",
            "director":"Fernando Rato","poblacion":"Talavera",
            "cuenta": [{
                  "tipo": "AHORRO",
                  "nombre":"Marcelo Saez","numero":30303036,
                  "saldohaber":150000,    "saldodebe": 12000
               }, {
                  "tipo": "AHORRO","nombre":"María Jesús Ramos",
                  "numero": 4444222,"saldohaber": 5000,
                  "saldodebe": 0
               }]
         }]
   }
}
```

ACTIVIDAD 5.13.

Convierte el documento *universidad.xml* de la colección *ColeccionPruebas* a objeto JSON.

5.6.2. Instalación MongoDB

En este apartado instalaremos la base de datos *NoSQL MongoDB*. Descargamos el archivo desde la página https://www.mongodb.com/download, para este libro se ha descargado la versión 3.2.6, *(mongodb-win32-x86_64-2008plus-ssl-3.2.6-signed.msi)* válida para Windows 32 y 64, que es la versión estable a fecha de hoy.

Para la instalación ejecutamos el archivo y seguimos el asistente (Figura 5.17), se acepta la licencia, se selecciona el tipo de instalación completa o custom (personalizada), se elige completa, y se pulsa el botón ***Install***.

Figura 5.17. Instalación de MongoDB.

La base de datos se instala por defecto en (*Archivos de programa*) ***C:\Program Files\MongoDB***. Para arrancar la base de datos buscaremos la carpeta bin de MongoDB (***C:\Program Files\MongoDB\Server\3.2\bin***), dentro de esa carpeta se encuentra el archivo ***mongod.exe***, que es el que arranca la BD.

Importante: antes de iniciar la base de datos crearemos la carpeta ***data*** en la unidad donde se ha instalado la BD, por ejemplo ***C:***\, y dentro de data se crea ***db*** (***C:\data\db***), esa es la carpeta que por defecto MongoDB va a utilizar para almacenar la información.

Si se quiere ***utilizar otro directorio***, por ejemplo, incluir la carpeta ***data*** dentro del propio Mongo DB (***C:\Program Files\MongoDB\data***), primero se crearán las carpetas, y luego ***al arrancar la BD tenemos que indicar dónde se encuentra la carpeta data.*** Por ejemplo, si yo me sitúo dentro de ***bin***, para asignar la carpeta ***data*** escribiré lo siguiente desde la línea de comandos ***mongod.exe --dbpath*** *camino* (con ..\ voy un nivel hacia atrás):

```
C:\Program Files\MongoDB\Server\3.2\bin>mongod.exe --dbpath ..\..\..\data
```

Una vez arrancada la BD, se muestra una pantalla como la de la Figura 5.18, en la que se inicializa la BD, y se queda escuchando por el puerto **27017** las conexiones de los clientes.

Figura 5.18. Server MongoDB iniciado.

Si ahora queremos conectarnos como cliente a la BD, ejecutaremos el programa ***mongo.exe*** de la carpeta ***bin***, también desde la línea de comando este programa inicia un cliente. Y desde el cliente podremos trabajar con la base de datos. Al conectarnos por defecto nos conecta a la BD test, véase la Figura 5.19.

Figura 5.19. Cliente MongoDB.

5.6.3. Operaciones básicas en MongoDB

Todos los comandos para operar con esta base de datos se escriben en minúscula, los más comunes son los siguientes:

- Listar las bases de datos: **`show databases`**
- Mostrar la base de datos actual: **`db`**
- Mostrar las colecciones de la base de datos actual: **`show collections`**
- Usar una base de datos (similar a MySQL): **`use nombrebasedatos`**, si no existe no importa, la creará en el momento que añadamos un objeto JSON, con las funciones ***.save*** o ***.insert***.
- Si queremos saber el número de documentos dentro de las colecciones, utilizaremos la función ***count***, escribiremos: **`db.nombre_colección.count()`**. También se utilizan las funciones ***size()*** y ***length()***.
- Para añadir comentarios utilizamos los caracteres // de comentario de Java.

CREAR REGISTROS

Para añadir datos a la base de datos utilizaremos los comandos **.save o .insert** según este formato:

```
db.nombre_colección.save(dato JSON)
db.nombre_colección.insert(dato JSON)
```

Donde **db** es la base de datos actual, la que estemos usando (la abriremos con **use**) y nombre de colección es la colección donde se van a añadir los registros, si no existe se crea en ese momento.

Ejemplo: creo la base de datos ***mibasedatos***, y dentro de ella la colección ***amigos*** con dos amigos:

```
use mibasedatos;
Amigo1={nombre:'Ana',teléfono:545656885,curso:'1DAM', nota:7};
Amigo2={nombre:'Marleni',teléfono:3446500,curso:'1DAM',nota:8};
db.amigos.save(Amigo1);
db.amigos.save(Amigo2);
```

Añado un amigo más, pero ahora utilizo la orden **insert:**

```
db.amigos.insert({nombre:'Juanito', teléfono:55667788,curso:'2DAM',
nota:6 });
```

IDENTIFICADOR DE OBJETOS, EL ObjectId (campo _id)

Los identificadores de cada documento (registro) son únicos. Se asignan automáticamente al crear el documento, se generan de forma rápida y ordenada. También se pueden crear de forma manual. Es un número hexadecimal que consta de 12 bytes, los primeros 4 son una marca de tiempo, los tres siguientes la identificación de la máquina, 2 bytes de identificador de proceso y un contador de 3 bytes empezando en un número aleatorio. El ***ObjectId*** o ***_id***, es como si fuese la clave del documento, no se repetirá en una colección. Si un documento no tiene ***_id*** *MongoDB* se lo asignará automáticamente, es lo que ocurre cuando insertamos y no indicamos el identificador.

5.6.4. Consultar registros

Para consultar datos de una colección utilizaremos la orden **.find()**, escribiremos:

```
db.nombre_colección.find()
```

En el ejemplo si queremos ver la colección *amigos* escribiremos: `db.amigos.find()`. Se muestran los identificativos **(_id)** de cada objeto JSON, únicos por colección, con el resto de campos. Véase la Figura 5.20.

Si se desea que la salida sea ascendente por uno de los campos, utilizamos el operador **.sort**, por ejemplo, para obtener los datos de la colección ordenados por nombre escribimos:

```
db.amigos.find().sort({nombre:1});
```

El número que acompaña a la orden indica el tipo de ordenación, **1 ascendente y -1 descendente**. Por ejemplo, esto visualiza los datos ordenados descendentemente por nombre:

```
db.amigos.find().sort({nombre:-1});
```

Si se desean hacer búsquedas de documentos que cumplan una o varias condiciones, utilizamos el siguiente formato:

```
db.nombre_colección.find(filtro, campos)
```

- En ***filtro*** indicamos la condición de búsqueda, podemos añadir los pares *nombre:valor* a buscar. Si omitimos este parámetro devuelve todos los documentos, o pasa un documento vacío ({ })
- En ***campos*** se especifican los campos a devolver de los documentos que coinciden con el filtro de la consulta. Para devolver todos los campos de los documentos omitimos este parámetro. Si se desean devolver uno o más campos escribiremos ***{nombre_campo1: 1, nombre_campo2: 1, ...}***. Si no se desean que se seleccionen los campos escribimos. ***{nombre_campo1: 0, nombre_campo2: 0, ...}*** También podemos poner ***true*** o ***false*** en lugar de 1 o 0.

Por ejemplo, para buscar el amigo con nombre Marleni lo escribimos así:

```
db.amigos.find({nombre : "Marleni"})
```

Si solo deseo saber su teléfono escribo:

```
db.amigos.find({nombre : "Marleni"},{teléfono:1})
```

Si deseo buscar el nombre y la nota de los alumnos de 1DAM escribiremos:

```
db.amigos.find({curso : "1DAM"}, {nombre:1, nota:1})
```

Si queremos saber el número de registros que devuelve una consulta pondremos **db.nombre_coleccion.find({filtros}).count()**, por ejemplo, para saber cuántos son del curso 1DAM escribiremos:

```
db.amigos.find({curso : "1DAM"}).count();
```

```
> use mibasedatos;
switched to db mibasedatos
> Amigo1={nombre:'Ana',teléfono:545656885,curso:'1DAM', nota:7};
{ "nombre" : "Ana", "teléfono" : 545656885, "curso" : "1DAM", "nota" : 7 }
> Amigo2={nombre:'Marleni',teléfono:3446500,curso:'1DAM',nota:8};
{
        "nombre" : "Marleni",
        "teléfono" : 3446500,
        "curso" : "1DAM",
        "nota" : 8
}
> db.amigos.save(Amigo1);
WriteResult({ "nInserted" : 1 })
> db.amigos.save(Amigo2);
WriteResult({ "nInserted" : 1 })
> db.amigos.insert({nombre:'Juanito', teléfono:55667788,curso:'2DAM', nota:6 });
WriteResult({ "nInserted" : 1 })
> db.amigos.find()
{ "_id" : ObjectId("5767678e7682441c00e99186"), "nombre" : "Ana", "teléfono" : 545656885, "curso" : "1DAM", "nota" : 7 }
{ "_id" : ObjectId("5767678e7682441c00e99187"), "nombre" : "Marleni", "teléfono" : 3446500, "curso" : "1DAM", "nota" : 8 }
{ "_id" : ObjectId("5767679c7682441c00e99188"), "nombre" : "Juanito", "teléfono" : 55667788, "curso" : "2DAM", "nota" : 6 }
>
```

Figura 5.20. Creación de *mibasedatos* y la colección *amigos*.

Se pueden hacer consultas más complejas añadiendo selectores de búsquedas.

SELECTORES DE BÚSQUEDAS DE COMPARACIÓN

- ***$eq***, igual a un valor. Esta orden obtiene los documentos con nota = 6:

  ```
  db.amigos.find({ nota : { $eq : 6 } })
  ```

- ***$gt***, mayor que y ***$gte*** mayor o igual que. Esta orden obtiene los documentos con nota >= 6:

  ```
  db.amigos.find({ nota : { $gte : 6 } })
  ```

- ***$lt***, menor que y ***$lte***, menor o igual que. El ejemplo muestra los amigos de 1DAM con notas entre 7 y 9 incluidas, preguntamos por un intervalo >=7 y <=9:

  ```
  db.amigos.find({curso : "1DAM", nota : { $gte : 7,  $lte : 9} })
  ```

- ***$ne***, distinto a un valor. El siguiente ejemplo obtiene los documentos con nota distinta de 7:

  ```
  db.amigos.find({ nota : { $ne : 7 } })
  ```

- ***$in***, entre una lista de valores y ***$nin***, no está entre la lista de valores. En el ejemplo se obtienen los documentos cuya nota sea uno de estos valores: 5,7 y 8:

  ```
  db.amigos.find({ nota : { $in : [5, 7, 8] } })
  ```

  Ahora añado el nombre y el curso:

  ```
  db.amigos.find({ nota : { $in : [5, 7, 8] } },{nombre:1, curso:1})
  ```

SELECTORES DE BÚSQUEDAS LÓGICOS

- ***$or***. La siguiente orden obtiene los documentos de los cursos 1DAM , o los que tienen nota > de 7:

  ```
  db.amigos.find({ $or : [ { nota: { $gt: 7 }}, {curso : "1DAM"} ]})
  ```

  Esta consulta obtiene los amigos con nombre Ana o Marleni:

  ```
  db.amigos.find({ $or: [ {nombre : "Ana"}, {nombre: "Marleni"} ] })
  ```

- ***Sand.*** Este operador se maneja implícitamente, no es necesario especificarlo. Las siguientes órdenes hacen lo mismo, obtienen los amigos del curso 2DAM y con nota 6:

```
db.amigos.find({ $and : [{curso : "2DAM"}, {nota : 6} ]})
db.amigos.find({curso : "2DAM", nota : 6})
```

Esta otra consulta devuelve el documento con nombre Marleni y con teléfono 3446500:

```
db.amigos.find( {nombre : "Marleni", teléfono : 3446500} )
db.amigos.find( { $and : [{nombre:"Marleni"},{teléfono:3446500}] })
```

- ***Snot.*** Representa la negación, en el ejemplo obtengo los amigos con nota no mayor de 7.

```
db.amigos.find({ nota : { $not: { $gt: 7 }} })
```

Esta otra consulta visualizará el nombre, el curso y la nota de los que su nota no mayor de 7:

```
db.amigos.find({ nota : { $not: { $gt: 7 }} },{_id:0, nombre:1,
curso:1, nota:1} )
```

- ***Sexists,*** este operador booleano permite filtrar la búsqueda tomando en cuenta la existencia del campo de la expresión. Este ejemplo obtiene los registros que tengan nota.

```
db.amigos.find( { nota : {$exists:true} })
```

5.6.5. Actualizar registros en MongoDB

Para actualizar datos utilizaremos el comando **.update,** según este formato de uso:

```
db.nombre_colección.update(
    filtro_búsqueda,
    cambios_a_realizar,
    {
        upsert: booleano,
        multi:  booleano
    });
```

Donde:

En ***filtro_búsqueda,*** se indica la condición para localizar los registros o documentos a modificar.

En ***cambios_a_realizar***, se especifican los cambios que se desean hacer. Hay que tener cuidado al utilizar esta orden, pues en ***cambios_a_realizar*** se indica cómo quedará el documento que se busca si este existe, es decir: el resultado final del documento es lo que se escriba en *cambios_a_realizar.*

Si no se escriben todos los campos que tenía el documento, estos no los incluye en la modificación, entonces, los elimina. Por ejemplo:

```
db.amigos.update({nombre:"Ana"},{nombre: "Ana María" } );
```

Esta orden cambia el documento con nombre *Ana* por nombre *Ana María*. El resto de campos como no aparecen en ***cambios_a_realizar*** los elimina.

Ahora cambio el teléfono de Marleni por 925666555, y solo mantengo el campo nombre y teléfono, el resto al no aparecer en ***cambios_a_realizar*** se eliminan:

```
db.amigos.update({nombre: 'Marleni'},{ nombre: 'Marleni' ,teléfono:
925666555 } );
```

Nos encontramos con dos tipos de cambios: cambiar el documento completo por otro que indiquemos, o modificar solo los campos especificados, para ello utilizamos los parámetros ***upsert*** y ***multi***, ambos son opcionales y su valor por defecto es ***false***:

- ***upsert*** – si asignamos ***true*** a este parámetro, se indica que si el filtro de búsqueda no encuentra ningún resultado, entonces, el cambio debe ser insertado como un nuevo registro.
- ***multi*** – en caso de que el filtro de búsqueda devuelva más de un resultado, si especificamos este parámetro a ***true***, el cambio se realizará a todos los resultados, de lo contrario solo se cambiará al primero que encuentre, es decir, al que tenga menor identificativo de objeto, `"_id"`.

Ahora cambio el teléfono de Pepita por 12345, y utilizo ***upsert***. Como Pepita no existe lo va a añadir por indicar ***upsert: true.***

```
db.amigos.update({nombre: 'Pepita'},
 { nombre: 'Pepita' ,teléfono: 12345 }, { upsert: true } );
```

El comando ***.update*** cuenta con una serie de operadores para realizar actualizaciones más complejas. Algunos de estos operadores son los siguientes:

OPERADORES DE MODIFICACIÓN

- ***$set,*** permite actualizar con nuevas propiedades a un documento (o conjunto de documentos). Por ejemplo, añado la edad 24 a Ana María y 34 a Marleni:

```
db.amigos.update({nombre:"Ana María"},{ $set: {edad:24} } )
db.amigos.update({nombre:"Marleni"},{ $set: {edad:34} } )
```

  Si el documento ya tiene ese campo, no lo añade, cambiaría el valor si es distinto.

- ***$unset***, permite eliminar propiedades de un documento. Por ejemplo, borro la edad 34 de Marleni:

```
db.amigos.update({nombre:"Marleni"},{ $unset: {edad:34} } )
```

- ***$inc,*** incrementa en una cantidad numérica especificada en el valor del campo a incrementar. Por ejemplo sumo 1 a la edad de Ana María:

```
db.amigos.update({nombre:"Ana María"},{ $inc: {edad:1} } )
```

- ***$rename,*** renombra campos del documento. Por ejemplo, cambiamos los campos nombre y edad de Ana María y los ponemos en inglés:

```
db.amigos.update({nombre:'Ana María'},{ $rename:{edad:'age',
nombre:'name'}})
```

Cargamos de nuevo los datos iniciales de amigos y probamos ahora las siguientes consultas. Subimos la nota 1 punto a los de 1DAM, podemos escribir estas órdenes:

```
db.amigos.update({curso:"1DAM"},{ $inc: {nota:1} } )
db.amigos.update({curso:"1DAM"},{ $inc: {nota:1} },{multi: true} )
```

La primera consulta solo sube la nota al primero que se encuentra, es decir, al primer *_id* que cumpla la condición. La segunda sube la nota a todos los del 1DAM por utilizar ***{multi: true}***.

En este ejemplo añado población Talavera a todos los del curso 1DAM:

```
db.amigos.update({curso:"1DAM"},{ $set: {población: "Talavera"} },
{multi: true})
```

ACTIVIDAD 5.14.

Crea la colección *empleados* dentro de la base de datos *mibasedatos*, y añade los siguientes registros:

```
Emp_no:1,nombre:"Juan",dep:10, salario:1000, fechaalta:"10/10/1999"
Emp_no:2,nombre:"Alicia",dep:10, salario:1400, fechaalta:"07/08/2000",
oficio: "Profesora"
Emp_no:3,nombre:"María Jesús",dep:20, salario:1500, fechaalta:
"05/01/2005", oficio: "Analista", comisión:100
Emp_no:4,nombre:"Alberto",dep:20, salario:1100, fechaalta:"15/11/2001"
Emp_no:5,nombre:"Fernando",dep:30, salario:1400, fechaalta:
"20/11/1999", comisión:200, oficio: "Analista"
```

Realiza las siguientes consultas:

- Visualiza los empleados del departamento 10.
- Visualiza los empleados del departamento 10 y 20.
- Obtén los empleados con salario >1300 y oficio Profesora.
- Sube el salario a los analistas en 100€, a todos los analistas.
- Decrementa la comisión en 20€ (escribir -20), solo a los que tengan comisión.

OPERACIONES CON ARRAYS

En este apartado veamos cómo realizar consultas en documentos que contienen arrays. Creamos la colección libros con tres libros, y un array con temas del libro:

```
db.libros.insert({codigo:1,nombre:"Acceso a datos", pvp: 35,
editorial:"Garceta", temas:["Base de datos", "Hibernate","Neodatis"]})
```

```
db.libros.insert({codigo:2,nombre:"Entornos de desarrollo", pvp: 27,
editorial:"Garceta", temas:["UML", "Subversión", "ERMaster"]})
```

```
db.libros.insert({codigo:3,nombre:"Programación de Servicios", pvp: 25,
editorial:"Garceta", temas:["SOCKET", "Multihilo"]})
```

Para consultar los elementos del array escribimos el array y el elemento a consultar. Ejemplos:

Libros que tengan el tema UML: `db.libros.find({temas:"UML"})`

Libros que tengan el tema UML o Neodatis:

```
db.libros.find( { $or: [ {temas:"UML"},{temas: "Neodatis"}] } )
```

Libros de la editorial Garceta, con pvp > 25 y que tengan el tema UML o Neodatis:

```
db.libros.find( { editorial:"Garceta", pvp: { $gt:25} , $or: [
{temas:"UML"} , {temas: "Neodatis"}] } )
```

OPERACIONES DE MODIFICACIÓN PARA ARRAYS

- ***$push***, añade un elemento a un array. Este ejemplo añade el tema *MongoDB* al libro con código 1:

```
db.libros.update( { codigo:1 },{ $push : {temas: "MongoDB" } })
```

- ***$addToSet***, agrega elementos a un array solo si estos no existen. En el ejemplo se añade el tema *Base de datos* a todos los libros que no lo tengan. Primero preguntamos si el libro tiene el campo *temas*. Para que se añada a todos los libros indicamos ***multi:true***:

```
db.libros.update({ temas : {$exists:true} },{ $addToSet: {temas:
"Base de datos" } }, {multi:true})
```

- ***$each***, se usa en conjunto con ***$addToSet*** o ***$push*** para indicar que se añaden varios elementos al array.

```
db.libros.update({codigo:1},{$push:{temas:{$each:["JSON","XML"]} } })
db.libros.update({codigo:2},{ $addToSet :{temas: { $each: ["Eclipse",
"Developper" ] } } })
```

- ***$pop***, elimina el primer o último valor de un array. Con valor -1 borra el primero, con otro valor el último. En el ejemplo se borra el primer tema del libro con código 3:

```
db.libros.update({codigo:3},{$pop: { temas:-1 } })
```

- ***$pull***, elimina los valores de un array que cumplan con el filtro indicado. En el ejemplo se borran de todos los libros los elementos *"Base de datos"* y *"JSON"*, si los tienen:

```
db.libros.update({}, {$pull:{ temas: { $in: ["Base de datos","JSON" ]
}}}, { multi: true } )
```

ACTIVIDAD 5.15.

Utilizando la colección libros realiza las siguientes consultas:

- Visualiza los libros de la editorial Garceta, con pvp entre 20 y 25 incluidos y que tengan el tema SOCKET.
- Agrega el tema SOCKET a los libros que no lo tengan.
- Baja a 5 el precio de los libros de la editorial Garceta.

5.6.6. Borrar registros

Para borrar datos JSON podemos utilizar las órdenes **.remove** y **.drop.** Se puede eliminar los documentos que cumplan una condición, o todos los documentos de la colección o la colección completa.

- Para borrar un documento que cumpla una condición utilizaremos la orden **remove({** ***nombre: valor*** **})**. Por ejemplo, se borra a Marleni:

```
db.amigos.remove({nombre : "Marleni"});
```

- Si se desea borrar al elemento con nombre Ana y teléfono 545656885 escribimos:

```
db.amigos.remove({nombre : "Ana",  teléfono : 545656885 });
```

- Para borrar todos los elementos de la colección ponemos: `db.amigos.remove({});`

- Para borrar la colección escribiremos: `db.amigos.drop();`

5.6.7. Funciones de agregado

Al igual que en otra base de datos MongoDB dispone de funciones matemáticas y de cadenas para utilizarlas en las consultas. Algunas de ellas son las siguientes:

FUNCIONES ARITMÉTICAS:

Función	Descripción
$abs	Devuelve el valor absoluto de un número
$add	Añade números a una cantidad o a una fecha, en este caso suma milisegundos
$ceil	Devuelve el entero menor, mayor o igual que el número especificado
$divide	Devuelve el resultado de dividir el primer número por el segundo. Tiene 2 argumentos
$floor	Devuelve el entero mayor, menor o igual que el número especificado
$mod	Devuelve el resto de dividir el primer número por el segundo. Tiene 2 argumentos
$multiply	Multiplica varios números, acepta varios argumentos
$pow	Eleva un número a la potencia especificada
$sqrt	Calcula la raíz cuadrada
$subtract	Devuelve el resultado de restar el primer número menos el segundo. Si los dos valores son números, devuelve la diferencia. Si los dos valores son fechas, devuelve la diferencia en milisegundos
$trunc	Trunca un número

FUNCIONES DE CADENAS:

Función	Descripción
$concat	Concatena varias cadenas, las que se pongan en la expresión
$substr	Devuelve una subcadena de una cadena, a partir de una posición indicada hasta una longitud especificada. Lleva 3 argumentos, la cadena, la posición de inicio y la longitud
$toLower	Convierte una cadena a minúsculas
$toUpper	Convierte una cadena a mayúsculas
$strcasecmp	Compara cadenas y devuelve 0 si las dos cadenas son equivalentes, 1 si la primera cadena es mayor que la segunda, y -1 si la primera cadena es menor que la segunda

FUNCIONES DE GRUPO:

Función	Descripción
$sum	Devuelve la suma de valores numéricos. Ignora los valores no numéricos
$avg	Devuelve la media de valores numéricos. Ignora los valores no numéricos
$first	Devuelve el primer valor del grupo
$last	Devuelve el último valor
$max	Devuelve el valor máximo de un grupo o de un array
$min	Devuelve el valor mínimo de un grupo o de un array

FUNCIONES DE FECHA:

Función	Descripción
$dayOfYear	Devuelve el día del año. Un número entre 1 y 366
$dayOfMonth	Devuelve el día del mes. Un número entre 1 y 31
$dayOfWeek	Devuelve el día de la semana. Un número entre 1 (Domingo) y 7 (Sábado)
$year	Devuelve el año, formato yyyy, por ejemplo, 2016
$month	Devuelve el número de mes entre 1 (Enero) y 12 (Diciembre)
$hour	Devuelve la hora entre 0 y 23
$minute	Devuelve los minutos entre 0 y 59
$second	Devuelve los segundos entre 0 y 6
$dateToString	Devuelve la fecha en formato String

USO DE ESTAS FUNCIONES:

Estas funciones se utilizan en las ***operaciones de agregación***, o ***consultas de agregación***, que lo que hacen es procesar los registros y obtener nuevos resultados, calculados o transformados.

La agregación opera con grupos de valores de múltiples documentos y se puede realizar una variedad de operaciones sobre los datos agrupados para devolver un solo resultado. El objetivo es presentar datos calculados, formateados y/o filtrados de manera diferente a como se encuentran en los documentos.

MongoDB ofrece tres formas de realizar la agregación: ***la agregación pipeline***, la función de ***map-reduce*** y la ***agregación de propósito único***. En este tema estudiaremos la agregación más común para hacer consultas complejas, la ***pipeline***.

Consulta esta URL para saber más sobre la agregación:
https://docs.mongodb.com/manual/aggregation/

5.6.8. La agregación pipeline

La ***agregación pipeline*** o ***tuberías de agregación*** se basa en someter una colección a un conjunto de ***operaciones*** o ***etapas,*** estas etapas irán convirtiendo y transformando el conjunto de documentos pertenecientes a la colección, hasta obtener un conjunto de documentos con el resultado deseado.

Se le llama tubería ya que cada etapa irá modificando, moldeando y calculando la estructura de los documentos para pasarlos a la etapa que le sigue. Las etapas son las siguientes:

Etapa	Descripción	Multiplicidad
$project	Cambia la forma del documento. La ***proyección*** permite modificar la representación de los datos, por lo que en general se emplea para darles una nueva forma con la que resulte más cómodo trabajar.	1:1

Etapa	Descripción	Multiplicidad
$match	Filtra los resultados. La etapa match permite ***filtrar*** los documentos para que en el resultado de la etapa solo estén aquellos que cumplen ciertos criterios. Se puede filtrar antes o después de agregar los resultados, en función del orden en que definamos esta etapa	n:1
$group	Agrupación. Permite ***agrupar*** distintos documentos según compartan el valor de uno o varios de sus atributos, y realizar operaciones de agregación sobre los elementos de cada uno de los grupos. Se utilizan las funciones *sum*, *max*, *min*, *avg*, etc.	n:1
$sort	Ordenación de documentos	1:1
$skip	Salta N elementos	n:1
$limit	Elige N elementos para el resultado	n:1
$unwind	Normaliza arrays	1:n
$out	Envía el resultado a una salida, se almacena en la BD como una nueva colección	1:1

La multiplicidad se refiere a cuántos documentos obtenemos como resultado después de aplicar la etapa, por ejemplo, **1:1** se aplica a 1 documento y se obtiene 1. **n:1** se aplica a **n** documentos y se obtiene **1**. **1:n** se aplica a un documento y se obtienen **n**.

Formato para utilizar las etapas:

```
db.mi_coleccion.aggregate([
    {
        $etapa1: {
            .....
        }
    },{
        $etapa2: {
            .....
        }
    },.....
])
```

EJEMPLOS:

Para los ejemplos creamos la colección ***artículos***, que se encuentra en el archivo de colecciones *MongoDB*, de la carpeta de recursos de la unidad. Cada documento artículo está formado por los campos: *código*, *denominación*, *pvp*, *categoría*, *uv*, y *stock*.

- Obtener las denominaciones de los artículos y la categoría convertida a mayúsculas. Se utiliza la etapa ***$project*** pues cambiamos el aspecto del documento. **Para referirnos a los campos del documento los ponemos entre comillas y con el prefijo $**:

```
db.articulos.aggregate(
  [ {
      $project:
        {
          denominación: { $toUpper: "$denominación" },
```

```
            categoría: { $toUpper: "$categoría" }
        }
    } ])
```

Si esta salida se desea almacenar en la base de datos, añadimos la etapa ***out***. Por ejemplo:

```
db.articulos.aggregate(
   [ {
       $project:
         {
           denominación: { $toUpper: "$denominación" },
           categoría: { $toUpper: "$categoría" }
         }
     },
     {    $out: "salidanueva"    }
    ])
```

Obtener la denominación en mayúsculas, el importe de las ventas, que serán las *uv* * el *pvp*, y el stock actual que será *stock* menos *uv*. Se utiliza la etapa ***$project***, la función ***$multiply*** para multiplicar las *uv* por el *pvp* y ***$subtract*** para restar las *uv* del *stock*.

•

```
db.articulos.aggregate(
   [ {
       $project:
         { artículo: { $toUpper: "$denominación" },
           importe: { $multiply: ["$pvp",  "$uv"] },
           stockactual: { $subtract: ["$stock",  "$uv"] }
         }
     } ])
```

Condiciones de agregación:

Podemos añadir las siguientes condiciones a las consultas de agregación:

Name	Descripción
$cond	Este operador evalúa una expresión y dependiendo del resultado, devuelve el valor de una de las otras dos expresiones. Recibe tres expresiones en una lista ordenada o tres parámetros con nombre. Formato: `{ $cond: [ <boolean-expression>, <caso-true>, <caso-false> ] }`
$ifNull	Devuelve o bien el resultado no nulo de la primera expresión o el resultado de la segunda expresión si la primera expresión da como resultado un resultado nulo. Acepta dos expresiones como argumentos. El resultado de la segunda expresión puede ser nulo. `{ $ifNull: [ <expression>, <expresionsiesnull> ] }`

Ejemplos:

- A la consulta anterior, vamos a preguntar si el stock actual es negativo, asignaremos a un campo nuevo llamado ***areponer*** true si es menor que 0 y false si no lo es. La condición que se añade es:

```
{  $cond: [ { $lte: [ { $subtract: ["$stock",  "$uv"] } , 0 ] },
```

```
            true , false ]
}
```

La consulta completa queda así:

```
db.articulos.aggregate(  [ {
      $project:
        { artículo: { $toUpper: "$denominación" },
          importe: { $multiply: ["$pvp", "$uv"] },
          stockactual: { $subtract: ["$stock",  "$uv"] },
          areponer: {
            $cond: [{$lte: [{ $subtract: ["$stock", "$uv"] } , 0]},
              true, false]     }
        }
    } ]  )
```

- En la siguiente consulta obtenemos por cada categoría el número de artículos, el total unidades vendidas de artículos, y el total importe, la suma ce los pvp*unidades. Es como una select con *group by*. En este caso se utiliza la etapa ***$group***, cuando se utiliza esta etapa se debe añadir el identificador de objeto ***_id***, en este caso como agrupamos por categoría lo indicamos en el ***_id***. Que a su vez será el identificador del resultado. Para contar artículos se utiliza la función ***$sum***, sumando 1:

```
db.articulos.aggregate( [ {
  $group: {
      _id: "$categoría",
      contador: { $sum: 1 },
      sumaunidades: { $sum: "$uv"},
      totalimporte: { $sum: { $multiply: ["$pvp",  "$uv"] } }
      }
  } ] )
```

- En la siguiente consulta obtenemos el número de documentos de la categoría ***Deportes***, el total de unidades vendidas de sus artículos, el total importe y la media de unidades vendidas. Se utilizan las etapas ***$match*** para seleccionar la categoría, y luego ***$group*** para obtener resultados agrupados, en este caso en *_id* ponemos cualquier valor:

```
db.articulos.aggregate( [
  { $match: { categoría: "Deportes"  } ],
  { $group: {
          _id: "deportes",
          contador: { $sum: 1 },
          sumaunidades: { $sum: "$uv"},
          media: { $avg: "$uv"},
          totalimporte: { $sum: { $multiply: ["$pvp",  "$uv"] } }
      }    }
] )
```

- En la siguiente consulta obtenemos el precio más caro, se agrupan los registros y obtenemos el máximo del pvp:

```
db.articulos.aggregate( [
  { $group: {
          id: null,
          maximo: { $max: "$pvp" } } }
] )
```

- En la siguiente consulta obtenemos el artículo con el precio más caro. Utilizamos dos consultas:

  1-Primero obtenemos los datos pvp y denominación, de todos los artículos, ordenados descendentemente por precio y denominación. Para ello utilizamos la etapa **Ssort**

  2-El resultado obtenido se agrupa con **Sgroup,** para luego obtener el primero con la función **Sfirst**:

```
db.articulos.aggregate(
   [
     { $sort: { pvp: -1, denominación: -1 } },
     { $group:
         { _id: null,
           mascaro: { $first: "$denominación" },
           precio:  { $first: "$pvp" } }
     }
] )
```

- En la siguiente consulta obtenemos la suma de importe de los artículos cuya denominación empieza por M o P. Para realizar esta consulta pasamos por 3 etapas:

  1-Obtenemos de todos los artículos el primer carácter de la denominación utilizando la función ***Ssubstr*** y el importe de cada artículo:

```
{  $project: {
      primercarac: { $substr: ["$denominación", 0, 1]   }  ,
      impor: {$multiply: ["$pvp",  "$uv"] } }
}
```

  2-En la siguiente etapa se seleccionan, de los datos obtenidos en la primera etapa ***(primercarac, impor)***, los que tienen en ***primercarac*** P o M:

```
{   $match: { "primercarac": { $in: ["M", "P"]   }  } }
```

  3-Y finalmente se agrupa ese resultado, se añade un ***_id: 1***, y se suman los importes.

```
{   $group: { _id: 1,
              totalimporte: { $sum:  "$impor"  }  }
}
```

  La consulta completa quedará así:

```
db.articulos.aggregate([
{  $project: {
      primercarac: { $substr: ["$denominación", 0, 1]   }  ,
      impor: {$multiply: ["$pvp",  "$uv"] } }
},
{   $match: { "primercarac": { $in: ["M", "P"]   }  }
},
{   $group: { _id: 1,
    totalimporte: { $sum:  "$impor"  }  }
} ] )
```

- En la siguiente consulta obtenemos por cada categoría el artículo con el precio más caro. Para ello primero ordenamos descendentemente por pcategoría, pvp y denominación, utilizando la etapa **Ssort**. Y el resultado obtenido se agrupa con **Sgroup** para luego obtener el primero de cada categoría con la función **Sfirst**:

```
db.articulos.aggregate(
```

```
    [
      { $sort: { categoria: -1, pvp: -1, denominación: -1 } },
      { $group:
          { _id: "$categoria",
            mascaro: { $first: "$denominación" },
            precio:  { $first: "$pvp" } }
      }
] )
```

UTILIZACIÓN DE ARRAYS, CAMPOS COMPUESTOS Y AGREGADOS

Carga la colección *Trabajadores* que se encuentra en el archivo de colecciones de los recursos de la unidad. El formato de un trabajador es este:

```
db.trabajadores.insert( {
    nombre: {nomb:"Alicia",apel:"Ramos", ape2:"Martín"},
    direccion: {población: "Madrid", calle : "Avda Toledo 10"},
    salario: 1200,
    oficios:["Profesora", "Analista"],
    primas: [20,30,40],
   edad:50
} )
```

Observa que el trabajador está formado por dos campos compuestos: *nombre* y *dirección*, y dos arrays: *oficios* y *primas*.

- La siguiente consulta de agregado devuelve la población, el nombre descompuesto en nombre, apel y ape2, el primer oficio del array oficios, el segundo oficio y el último. Si no los tiene no devuelve nada. Ordenados por población ascendentemente. Para acceder a los campos compuestos navegamos como si fuesen un objeto, ***nombre.nom***, o ***dirección.población***. Por ejemplo, esta consulta devuelve los que tienen la población Toledo:

  ```
  db.trabajadores.find({ "direccion.población":"Toledo"})
  ```

  Para acceder a los elementos de un array utilizamos la función ***$arrayElemAt: ["$nombre_del_array", posición_del_elemento_a_consultar]***. La posición es 0 para el primer elemento, y -1 para el último.

  ```
  db.trabajadores.aggregate(
     [
       { $sort: { "direccion.población":  1} },
       { $project:
           {
             población: "$direccion.población",
             nombre:  "$nombre.nomb" ,
             apel:  "$nombre.apel" ,
             ape2:  "$nombre.ape2" ,
             oficio1: { $arrayElemAt: [ "$oficios", 0 ] },
             oficio2 : { $arrayElemAt: [ "$oficios", 1 ] } ,
             oficioultimo: { $arrayElemAt: [ "$oficios", -1 ] }
           }
       }
  ])
  ```

  Otras funciones para arrays son:

Nombre	Descripción
$arrayElemAt	Devuelve el elemento especificado en el índice
$concatArrays	Devuelve un array concatenado en una cadena
$filter	Selecciona elementos de un array y devuelve otro array con esos elementos
$isArray	Determina si el operando es una array o no. Devuelve true o false.
$size	Devuelve el número de elementos del array
$slice	Devuelve un sub-set de elementos del array, se especifica el número.

- La siguiente consulta de agregado devuelve los elementos que tienen los arrays de los trabajadores (oficios y primas), y los arrays concatenados. Se utiliza la función ***"$ifNull"*** para comprobar que los arrays existan en los trabajadores, y evitar errores de salida *("The argument to $size must be an Array, but was of type: EOO")*. Se pregunta si el array es null, si lo es devuelve el array vacío, el tamaño del array vacío será 0.

```
db.trabajadores.aggregate( [
 { $project:
    {
     nombre: "$nombre.nomb" ,
     numerooficios: { $size: { "$ifNull": ["$oficios",[] ] } },
     numeroprimas: { $size: { "$ifNull": ["$primas",[] ] } },
     oficiosconcatenados: {$concatArrays: ["$oficios","$primas"] }
    }
 }
] )
```

- La siguiente consulta de agregado devuelve el número de trabajadores y la media de edad de los trabajadores que han tenido el oficio de *Analista*.

```
db.trabajadores.aggregate( [
   { $match: { oficios: "Analista" } },
   { $group: {
           _id: "analista",
           contador: { $sum: 1 },
           media: { $avg: "$edad"} }
    }
])
```

ACTIVIDAD 5.16.

Utilizando la colección trabajadores realiza las siguientes consultas:

- Visualiza la edad media, la media de salario y el número de trabajadores que hayan tenido una prima de 30 o de 80.
- Visualiza por población el número de trabajadores, el salario medio y el máximo salario.
- Visualiza el nombre, ape1 y ape2 del empleado que tiene máximo salario.
- A partir de la consulta anterior, obtén ahora el nombre, ape1, ape2 y salario del empleado que tiene máximo salario por cada población:

Para saber más sobre consultas, consulta las siguientes URLs de MongoDB:
Operadores para agregación: https://docs.mongodb.com/manual/reference/operator/aggregation/
Operadores para consultas: https://docs.mongodb.com/manual/reference/operator/query/
Operadores de actualización: https://docs.mongodb.com/manual/reference/operator/update/

Comandos de MongoDB: https://docs.mongodb.com/manual/reference/command/
Funciones para las colecciones: https://docs.mongodb.com/manual/reference/method/js-collection/

5.6.9. Relaciones entre documentos en MongoDB

MongoDB utiliza 2 métodos o patrones que nos van a permitir establecer la estructura de los documentos y sus relaciones. Vamos a ver cómo son las relaciones entre documentos comparándolas con el modelo relacional. Los métodos son utilizar referencias:

- ***Referencias Manuales:*** en la que se guarda el campo ***_id*** de un documento como referencia en otro documento. Similar al concepto de clave ajena del modelo relacional. En este método la aplicación debe ejecutar una segunda consulta para devolver los datos relacionados. Este es el método más utilizado.

  Ejemplo: se van a crear las colecciones ***emple*** y ***depart***, cada elemento con su **_id** creado manualmente. Y se va a simular una relación 1 a muchos, 1 departamento va a tener a varios empleados.

  Colección *emple*, con 4 empleados.

```
db.emple.insert({_id:'emp1',nombre:"Juan", salario:1000,
fechaalta:"10/10/1999"})
db.emple.insert({_id:'emp2',nombre:"Alicia", salario:1400,
fechaalta:"07/08/2000", oficio: "Profesora"})
db.emple.insert({_id:'emp3',nombre:"María Jesús", salario:1500,
fechaalta: "05/01/2005", oficio: "Analista", comisión:100})
db.emple.insert({_id:'emp4',nombre:"Alberto", salario:1100,
fechaalta:"15/11/2001"})
```

  Colección ***depart*** con dos departamentos, asignamos los dos primeros empleados al primer departamento, y los dos siguientes al segundo. Para asignar los empleados ponemos el nombre de la colección (***emple***) y entre corchetes dentro de un array de referencias, los **_id** de los empleados a incluir, por ejemplo ***emple:['emp1', 'emp2']*** :

```
db.depart.insert({_id:'dep1',nombre:"Informática",loc:'Madrid',
emple:['emp1', 'emp2']})
db.depart.insert({_id:'dep2',nombre:"Gestión",loc:'Talavera',
emple:['emp3', 'emp4']})
db.depart.find()
```

  Para visualizar los datos de la combinación de las colecciones necesitaremos hacer dos consultas, una para obtener el departamento a consultar, y la otra para obtener los empleados de ese departamento, que están dentro del array del departamento. Por ejemplo, se desea visualizar los empleados del departamento con identificativo **_id** igual a **dep1.**

  1º Cargamos el departamento con **_id:dep1** en una variable, utilizamos el método **.findOne**. El método ***findOne ( )*** siempre incluye el campo **_id** incluso si el campo no se especifica explícitamente en el parámetro de consulta:

```
departrabajo = db.depart.findOne({_id:'dep1'})
```

  2º Recuperamos los empleados cuyo **_id** se encuentre enlazado a este departamento (***departrabajo*** en el ejemplo):

```
emplesdep = db.emple.find({_id: { $in : departrabajo.emple } } )
```

Si se añade el método **.toArray** los datos se devuelven en una matriz que contiene todos los documentos de la consulta, es decir, devuelve un array de documentos:

```
emplesdep = db.emple.find({_id: { $in : departrabajo.emple } }
).toArray()
```

La siguiente consulta devuelve los empleados del departamento dep2 que tienen el salario > de 1400:

Se carga el departamento: `departrabajo = db.depart.findOne({_id:'dep2'})`

Y luego los empleados:

```
emplesdep = db.emple.find({_id: { $in : departrabajo.emple },
salario: {$gt:1400 }} ).toArray()
```

ACTIVIDAD 5.17.

Utilizando la transformación del documento ***sucursales.xml*** a JSON, realizada en los apartados anteriores, se pide crear las colecciones ***cuentas*** y ***sucursales***. Para crearlas primero crea la colección *cuentas* y luego crea la colección *sucursales* asignando a las sucursales las cuentas correspondientes.

Una vez creadas las colecciones realiza las siguientes consultas:

- Visualiza las cuentas de las sucursales de Madrid.
- Visualiza las cuentas con *saldohaber* > 10000 cuyo director sea Fernando Rato
- Sube 300 el *saldohaber* de las cuentas de la sucursal con código SUC1 .

- ***DBRefs*** son referencias de un documento a otro utilizando el valor del campo ***_id*** del primer documento, el nombre de la colección, y, opcionalmente, el nombre de base de datos. Con la inclusión de estos nombres, los DBRefs permiten que documentos que se encuentran en varias colecciones sean vinculados para ser documentos de una sola colección. Los ***DBRefs*** proporcionan en esencia una semántica común para la representación de los vínculos entre documentos. Los ***DBRefs*** también requieren consultas adicionales para devolver los documentos de referencia.

5.6.10. Herramienta Robomongo

Robomongo es una herramienta multiplataforma con la que se pueden administrar de forma más visual las bases de datos MongoDB.

Esta herramienta integra la Shell de mongodb en un entorno gráfico con todas sus funcionalidades, además se podrá trabajar con múltiples conexiones a las bases de datos, podremos navegar por las colecciones y a la hora de hacer las consultas contaremos con el resaltado de sintaxis y autocompletado del código, muy útil para detectar los errores.

Robomongo se descarga de la URL https://robomongo.org/. Existe la versión instalable la ***.exe***, o la versión portable ***.zip***. En esta unidad se ha optado por utilizar la versión portable robomongo-0.9.0. Para trabajar con ella descomprimimos el zip en una carpeta y ejecutamos el archivo ***Robomongo.exe***. Pide conexión con la base de datos, debemos tener la base de datos arrancada para que la detecte. El puerto de escucha de MongoDB **es 27017**. Véase Figura 5.21.

Figura 5.21. Conexión a MongoDB desde Robomongo.

Desde la ventana de Robomongo podemos navegar por las bases de datos, las colecciones, y los objetos de las colecciones. Al hacer clic en una colección se abrirá una pestaña donde se podrán ver los documentos de la colección, también podemos ver los campos del documento si hacemos doble clic en el objeto o si lo desplegamos. En la parte superior es donde escribiremos las consultas, véase la Figura 5.22.

Figura 5.22. Ventana de trabajo de Robomongo

Desde los menús contextuales se podrán realizar todo tipo de operaciones:

- Desde el menú contextual de la conexión a la base de datos podremos crear nuevas bases de datos, abrir la *Shell* de MongoDB, que se abrirá en una pestaña para hacer las consultas, obtener información de la base de datos o desconectarnos.
- Desde el menú contextual asociado a una base de datos, además de abrir la *Shell*, se podrán obtener estadísticas, reparar o borrar la base de datos.

- Desde el menú contextual de una colección podremos ver los documentos, insertar, modificar o borrar documentos. Podremos cambiar el nombre de la colección, hacer una copia o borrar la colección.

5.6.11 MongoDB en Java

Para trabajar en Java con MongoDB necesitamos descargar el driver desde la URL de MongoDB https://mongodb.github.io/mongo-java-driver/. La versión que se va a utilizar en este libro es la ***mongodb-java-driver-3.2.2.jar.***

Antes de trabajar con MongoDB definamos BSON: ***BSON*** es un formato de serialización binaria, se utiliza para almacenar documentos y hacer llamadas a procedimientos en MongoDB. La especificación BSON se encuentra en ***bsonspec.org***. BSON soporta los siguientes tipos de datos como valores en los documentos, cada tipo de dato tiene un número y un alias que se pueden utilizar con el operador ***$type*** para consultar los documentos por tipo BSON. Algunos de los tipos BSON son los siguientes:

Tipo	Número	Alias
Double	1	"double"
String	2	"string"
Object	3	"object"
Array	4	"array"
Binary data	5	"binData"
ObjectId	7	"objectId"
Boolean	8	"bool"
Date	9	"date"
Null	10	"null"
Symbol	14	"symbol"
Timestamp	17	"timestamp"

Al comparar los valores de los diferentes tipos BSON, MongoDB utiliza el siguiente orden de comparación, de menor a mayor: *Null, Numbers* (ints, longs, doubles), *Symbol, String, Object, Array, BinData, ObjectId, Boolean, Date, Timestamp*

CONEXIÓN A LA BD

Para conectarnos a la base de datos creamos una instancia ***MongoClient***, por defecto crea una conexión con la base de datos local, y escucha por el puerto 27017. Todos los métodos relacionados con operaciones ***CRUD (Create, Read, Update and Delete)*** en Java se acceden a través de la interfaz ***MongoCollection***. Las instancias de ***MongoCollection*** se pueden obtener a partir de una instancia ***MongoClient*** por medio de una ***MongoDatabase***. Así pues para conectarme a la base de datos *mibasedatos* y a la colección *amigos* escribiré lo siguiente:

```
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("mibasedatos");
MongoCollection <Document> coleccion = db.getCollection("amigos");
```

También se puede escribir el nombre del servidor y el puerto:

```
MongoClient cliente = new MongoClient("localhost",27017);
```

MongoCollection es una interfaz genérica: el parámetro de tipo TDocument es la clase que los clientes utilizan para insertar o modificar los documentos de una colección, y es el tipo predeterminado para devolver búsquedas (*.find*) y agregados (*.aggregate*). El método de un solo argumento ***getCollection*** devuelve una instancia de ***MongoCollection < Document>***, y así es como podemos trabajar con instancias de la clase de documento.

VISUALIZAR LOS DATOS DE UNA COLECCIÓN

Los datos de una colección se pueden cargar en una lista utilizando el método ***find().into()*** de la siguiente manera:

```
List<Document> consulta = coleccion.find().
                                    into(new ArrayList<Document>());
for (int i = 0; i < consulta.size(); i++) {
     System.out.println(" - " + consulta.get(i).toString());
}
```

También podemos recuperar los valores de los campos del documento, utilizando los métodos ***get*** del objeto ***Document***, reciben como parámetro el nombre de la clave. Si se sabe el tipo de dato de la clave elegiremos el método correspondiente, y si no utilizamos ***get()*** que devuelve un objeto. Primero cargamos el elemento de la lista en un *Document*. Si la clave no existe en el documento visualizará null:

```
for (int i = 0; i < consulta.size(); i++) {
       Document amig = consulta.get(i);
       System.out.println(" - " + amig.getString("nombre") + " - " +
       amig.get("teléfono") + " - " +
       amig.getString("curso") + " - " + amig.getDouble("nota") );
      }
```

INSERTAR DOCUMENTOS

Para insertar documentos, creamos un objeto *Document*, con el método ***put*** asignamos los pares *clave-valor*, donde el primer parámetro es el nombre del campo o la clave, y el segundo el valor. Y con el método ***insertOne*** se inserta en la colección. El siguiente código añade un amigo a la colección:

```
Document amigo = new Document();
amigo.put("nombre", "Pepito");
amigo.put("teléfono", 925677);
amigo.put("curso", "2DAM");
amigo.put("fecha", new Date());
coleccion.insertOne(amigo);
```

También se puede insertar documentos utilizando el método ***append*** de ***Document***. Por ejemplo, se va a insertar el siguiente documento, se crea en ***curso*** un nuevo documento con dos pares *clave-valor*:

```
{  "Nombre" : "Pedro",    "teléfono" : 12345,
   "curso" : { curso1: "1DAM", curso2: "2DAM" } }

Document amigo2 = new Document("nombre", "Pedro")
                      .append("teléfono" , 1234)
                      .append("curso", new Document("curso1",
                                  "1DAM").append("curso2","2DAM" ));
coleccion.insertOne(amigo2);
```

A la hora de visualizar el curso utilizaremos el método ***get()*** en lugar de ***getString()***.

Se puede insertar en la base de datos una lista de documentos en una colección utilizando el método ***insertMany***. Por ejemplo:

```
List<Document> listadocs = new ArrayList<Document>();
for (int i = 0; i < 100; i++) {
    listadocs.add(new Document("Valor de i", i));
}
coleccion.insertMany(listadocs);
```

Si se desea saber los documentos de la colección se puede utilizar el método **count**:

```
coleccion.count());
```

CONSULTAR DOCUMENTOS

Anteriormente se ha visto cómo cargar los documentos en una lista utilizando el método ***find.into()***. El método ***find***() devuelve un cursor, devuelve una instancia ***FindIterable***. Podemos utilizar el método ***iterator()*** para recorrer el cursor. En el ejemplo recuperamos todos los documentos de la colección y se visualizan en formato ***Json***:

```
MongoCursor<Document> cursor = coleccion.find().iterator();
while (cursor.hasNext()) {
            Document doc = cursor.next();
            System.out.println(doc.toJson());
}
cursor.close();
```

Si solo se desea obtener el primer documento utilizamos el método ***first()***:

```
Document doc = coleccion.find().first();
```

UTILIZAR FILTROS EN LAS CONSULTAS

El método ***find()*** admite la utilización de filtros. Para utilizar los métodos de la clase *Filters* hacemos un ***import static*** de la clase *Filters* de la siguiente manera:

```
import static com.mongodb.client.model.Filters.*;
```

En el ejemplo se busca el documento con la clave *nombre* igual a *Ana*, devuelve solo el primero, por añadir el método *first*:

```
Document doc = coleccion.find(eq("nombre", "Ana")).first();
try {
      System.out.println("Localizado: " + doc.toJson());
} catch (NullPointerException e) {
      System.out.println("No se encuentra.");
}
```

Si el filtro devuelve varios documentos los recuperamos con un cursor, o bien con una lista como ya vimos al principio. En el ejemplo recuperamos los datos de los amigos con nota >5 y se visualizan en formato Json:

```
MongoCursor<Document> docs = coleccion.find(gt("nota", 5)).iterator();
while (docs.hasNext()) {
            Document docu = docs.next();
            System.out.println(docu.toJson());  }
docs.close();
```

Esta condición recupera los amigos con un 5 o un 8 en nota:

```
coleccion.find( or(eq("nota", 5),  eq("nota", 8)) )
```

Esta condición recupera los amigos de 1DAM y nota 8:

```
coleccion.find( and(eq("curso","1DAM" ), eq("nota", 8)) )
```

Si se desea extraer los objetos ***BSON*** de un documento, utilizaremos los filtros:

```
System.out.println(" - --- Objetos Bson -----------------");
MongoCursor<Document> cursor2 = coleccion.find().iterator();
while (cursor2.hasNext()) {
    Document doc2 = cursor2.next();
    Bson id = eq("_id", doc2.get("_id"));
    Bson nombre = eq("nombre", doc2.get("nombre"));
    Bson curso = eq("curso", doc2.get("curso"));
    System.out.println("Id: " + id + ". Nombre: "
       + nombre.toString() + ". Curso : " + curso.toString());
}
```

ORDENAR RESULTADOS

Para ordenar el resultado de una consulta importamos los métodos de la clase ***Sorts***:

```
import static com.mongodb.client.model.Sorts.*;
```

La siguiente condición consulta los amigos del curso 2DAM ordenados descendentemente por el campo *nombre*:

```
coleccion.find(eq("curso","2DAM" ))
                .sort(descending("nombre")).iterator();
```

UTILIZAR PROYECCIONES

A veces no se necesitan todos los datos contenidos en un documento, se pueden utilizar proyecciones para cambiar las salidas. Se necesitan importar los métodos de la clase ***Projection***, estos métodos devuelven un tipo *BSON*, que podrá ser utilizado en otro método. El ***import*** debe ser el siguiente

```
import static com.mongodb.client.model.Projections.*;
```

El siguiente ejemplo devuelve el nombre y la nota de los amigos del curso 1DAM, se utiliza el método ***include*** para añadir solo nombre y nota:

```
MongoCursor<Document> docs3 = coleccion.find(eq("curso", "1DAM"))
               .sort(ascending("nombre"))
               .projection(include("nombre","nota")).iterator();
while (docs3.hasNext()) {
    Document docu = docs3.next();
    System.out.println(docu.toJson());
}
docs3.close();
```

Si no se desea incluir en la consulta algunos de los campos utilizamos el método ***exclude***, en la consulta no se desea visualizar el _id, la nota y el nombre de *Ana*:

```
Document dd = coleccion.find(eq("nonbre", "Ana"))
      .projection(exclude("_id","nota", "nombre")).first();
```

UTILIZAR AGREGACIONES

Para utilizar los agregados se necesitan importar los métodos de la clase ***Aggregates***. Cada método devuelve una instancia del tipo *BSON*, que a su vez se puede pasar al método de agregado de *MongoCollection*. El ***import*** debe ser el siguiente:

```
import static com.mongodb.client.model.Aggregates.*;
```

Para añadir las etapas de agregado se utiliza un ***Arrays.AsList*** de java.util. Este ejemplo visualiza los amigos del curso 1DAM, utiliza la etapa ***match***:

```
MongoCursor<Document> docs4 = coleccion.aggregate(
        Arrays.asList(match(eq("curso", "1DAM")))).iterator();
while (docs4.hasNext()) {
    Document docu = docs4.next();
    System.out.println(docu.toJson());
}
docs4.close();
```

En el siguiente ejemplo se visualiza el nombre y la nota de los amigos de 1DAM, se utiliza también la la etapa ***project***:

```
MongoCursor<Document> docs5 = coleccion.aggregate(Arrays.asList(
    match(eq("curso", "1DAM")),
    project(fields(include("nombre", "nota"), excludeId())))
).iterator();
```

Calculo ahora la suma y la nota media, agrupada por curso. Para utilizar las funciones de cálculo importamos los métodos de la clase ***Acumulator***:

```
import static com.mongodb.client.model.Accumulators.*;
```

La consulta quedará así:

```
MongoCursor<Document> docs5 = coleccion.aggregate(Arrays.asList(
group("$curso", sum("sumanota", "$nota"), avg("medianota", "$nota"))
    ).iterator();
```

Si se desea calcular una media, o una suma de un campo, global para todos los documentos, el primer parámetro del group lo dejamos vacío:

```
group("", sum("sumanota", "$nota"), avg("medianota", "$nota"))
```

Si se desea que la salida de la consulta se almacene en una nueva colección en la base de datos añadimos la etapa ***out***. En el ejemplo las notas medias y las sumas de las notas se almacenarán en la colección *mediascurso*:

```
MongoCursor<Document> docs7 = coleccion
.aggregate(Arrays.asList(group("$curso", sum("sumanota", "$nota"),
                                         avg("medianota", "$nota")),
                         out('mediascurso"))
).iterator();
```

ACTUALIZAR DOCUMENTOS

Para realizar actualizaciones se necesita importar los métodos de la clase ***Updates***:

```
import static com.mongodb.client.model.Updates.*;
```

En este ejemplo actualizamos la nota de *Ana* y la asignamos la nota 5. Para actualizar un único documento se utiliza el método ***updateOne***, si hay varios con nombre Ana, actualiza el primero. La actualización devuelve un ***UpdateResult***:

```
coleccion.updateOne(eq("nombre", "Ana"), set("nota", 5));
```

Si ahora deseamos actualizar varios registros que cumplan una condición utilizamos el método ***updateMany***, este ejemplo sube la nota a todos los amigos de 1DAM, la sube 1 punto. La actualización devuelve un ***UpdateResult*** que tiene métodos para decir cuántos se seleccionan y cuántos se actualizan:

```
UpdateResult updateResult = coleccion.updateMany(
                            eq("curso", "1DAM"), inc("nota", 1));
System.out.println("Se han modificado: " +
                    updateResult.getModifiedCount());
System.out.println("Se han seleccionado: " +
                   updateResult.getMatchedCount());
```

Si se desean actualizar todos los registros de la colección, utilizamos la función ***exists("_id")*** para que devuelvan todos los documentos que tengan ***_id***:

```
updateResult = coleccion.updateMany(exists("_id"),inc("nota", 2));
```

Para añadir un campo se utiliza también la función ***set***, si el campo no existe lo crea. Esta consulta añade la población a Marleni:

```
coleccion.updateOne(eq("nombre","Marleni"),set("población","Toledo"));
```

Para eliminar un campo utilizamos el método ***unset***, en el ejemplo borro la población a los de 1 DAM, borrará el campo a los que lo tengan:

```
coleccion.updateMany(eq("curso", "1DAM"), unset("población"));
```

BORRAR UN DOCUMENTO DE LA COLECCIÓN

Como en el caso anterior, para borrar un documento de la colección utilizamos el método ***deleteOne***, y para borrar varios ***deleteMany***. También se devuelve ruEn el ejemplo se borra el documento con nombre María, borrará solo el primero. En el segundo borra todos los documentos de la colección. Devuelven un ***DeleteResult***.

```
// Borro un documento
DeleteResult del = coleccion.deleteOne(eq("nombre", "Ana"));
System.out.println("Se han borrado: " + del.getDeletedCount());
//Borro todos
del= coleccion.deleteMany(exists("_id"));
System.out.println("Se han borrado: " + del.getDeletedCount());
```

CREAR Y BORRAR UNA COLECCIÓN

Para crear una colección utilizamos el método ***createCollection***, asociado a la base de datos, y para borrarla utilizamos el método ***drop*** asociado a la colección:

```
//Crear Colección:
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("mibasedatos");
db.createCollection("nuevacoleccion");
```

```
//Borro la colección
MongoCollection<Document> cnueva = db.getCollection("nuevacoleccion");
cnueva.drop();
```

LISTAR LAS COLECCIONES DE LA BASE DE DATOS

El método ***listCollectionNames*** devuelve las colecciones de la base de datos en un *MongoIterable*, para listar la lista podemos elegir una de las siguientes maneras:

```
// Listar las colecciones de la BD
System.out.println("Listado de colecciones: ");
MongoIterable<String> colecciones = db.listCollectionNames();
Iterator col = colecciones.iterator();
while (col.hasNext())
      System.out.println(col.next());

// También se pueden listar así:
for (String name : db.listCollectionNames()) {
            System.out.println(name);
}
```

CREAR, LISTAR Y BORRAR BASES DE DATOS

Para crear una base de datos se llama al método ***getDatabase*** desde un objeto ***MongoClient***, sin embargo, la base de datos no se creará hasta que no se inserte un documento. El siguiente código creará la base de datos ***nueva*** y la colección ***colecnueva*** con un documento:

```
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("nueva");
MongoCollection<Document> clnue = db.getCollection("colecnueva");
Document doc1 = new Document("nombre", "Pedro").append("teléfono",
     1234).append("curso","2DAM");
clnue.insertOne(doc1);
```

El siguiente código obtiene los nombres de las bases de datos:

```
for (String name: cliente.listDatabaseNames()) {
             System.out.println(name);
}
```

Par borrar una base de datos se utiliza la orden:

```
cliente.getDatabase("base_datos_a_borrar").drop();
```

PASAR DATOS DE MONGODB A UN FICHERO DE TEXTO

En este método extraemos los documentos de la colección ***amigos*** y los guardamos en un fichero de texto en formato JSON llamado ***amigos.json***.

```
public static void crearficherojson() {
  MongoClient cliente = new MongoClient();
  MongoDatabase db = cliente.getDatabase("mibasedatos");
  MongoCollection<Document> coleccion = db.getCollection("amigos");

  File fiche = new File("./amigos.json");
  FileWriter fic;
  try {
```

```
        fic = new FileWriter(fiche);
        BufferedWriter fichero = new BufferedWriter(fic);
        // Recorro la colección amigos:
        System.out.println(" ----------------------------------");
        List<Document> consulta = coleccion.find()
                        .into(new ArrayList<Document>());
        for (int i = 0; i < consulta.size(); i++) {
              System.out.println(" Grabo elemento  " + i +
                  ", " + consulta.get(i).toString());
              fichero.write(consulta.get(i).toJson());
              fichero.newLine();
              }
        fichero.close();
    } catch (IOException e) {e.printStackTrace();}
}
```

En este método hacemos lo contrario, leemos el fichero de texto ***amigos.json*** que contiene documentos JSON, y los almacenamos en la BD en una colección llamada ***amigosfile.*** Para convertir una cadena a formato JSON utilizamos el método ***parse*** de la clase ***Document***:

```
private static void leerficheroyguardardatos() {
 try {
      FileReader fr = new FileReader("./amigos.json");
      BufferedReader bf = new BufferedReader(fr);

      MongoClient cliente = new MongoClient();
      MongoDatabase db = cliente.getDatabase("mibasedatos");
      MongoCollection<Document> coleccion =
                      db.getCollection("amigosfile");

      String cadenajson;
      while ((cadenajson = bf.readLine()) != null) {
            System.out.println(cadenajson);
            Document docu = new
                 Document(org.bson.Document.parse(cadenajson));
            coleccion.insertOne(docu);
      }
  } catch (FileNotFoundException e) {e.printStackTrace();
  } catch (IOException e) { e.printStackTrace();}
}
```

ACTIVIDAD 5.18.

Realiza un método que lea los datos de la tabla Empleados de la BD MySQL *ejemplo* (creada en los primeros temas), y guarde los datos en MongoDB en una colección con nombre ***empleadosmysql***. Los nombres de las claves de los documentos deben ser los nombres de las columnas de la tabla, convertidas a minúscula, utiliza la Interfaz ***ResultSetMetaData*** de jdbc.

Crea también los siguientes métodos:

Para visualizar los datos de la colección creada.

Para crear un fichero de texto con los datos de esta colección en formato JSON.

Método para subir 100 al salario de los empleados del oficio EMPLEADO

COMPRUEBA TU APRENDIZAJE

1. A partir del documento ***departamentos.xml*** de la colección ***ColeccionPruebas***. Realiza un programa Java para gestionar dicho documento. Utiliza las clases **Main.java** y **Pantalla.java** que se incluyen en la carpeta de recursos del tema.

El programa debe mostrar la pantalla inicial (Figura 5.23) donde podremos consultar, insertar, borrar y eliminar nodos <DEPT_ROW> del documento *departamentos.xml*.

En todas las operaciones nos aseguraremos de trabajar con el documento ***departamentos.xml***, utilizaremos la función ***doc(documento)*** en las consultas.

Comprobar 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. Visualizar los mensajes informativos en cada caso.

Crear 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. Utilizar las *Sentencias de actualización de eXist*.

Figura 5.23. Pantalla inicial ejercicio Gestión de Departamentos.

A partir de la colección VENTAS

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>
```

Realiza los siguientes ejercicios:

2. 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 Martin</nombre><nufac>10</nufac></facturasclientes>
<facturasclientes> <nombre>Pilar Martin</nombre><nufac>11</nufac></facturasclientes>
<facturasclientes> <nombre>Antonio Reus</nombre><nufac>12</nufac></facturasclientes>
. . . . . .
```

3. 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>
. . . . . .
```

4. Realiza una consulta XQuery para obtener el detalle de las ventas de los productos de cada una de las facturas del documento ***detallefacturas.xml.*** Crea 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">
. . . . . .
```

5. Haz un programa Java que genere un fichero XML a partir de esta consulta, llamarle al nuevo fichero ***totalfacturas.xml***, el elemento raíz se llamará también ***totalfacturas***. Una vez creado subir 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. Crea un método para cada operación.

6. Utilizando el documento creado (***totalfacturas.xml***), 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 (***totalfacturas.xml***).

7. Utilizando los documentos ***facturas.xml*** (ya actualizado) y ***clientes.xml,*** realiza una consulta XQuery para obtener todo lo que tiene que pagar cada cliente. La salida debe ser similar a:

```
<clien>
    <nombre>Alicia Diaz</nombre>
    <total>0</total>
</clien>
<clien>
    <nombre>Pilar Martin</nombre>
    <total>7880</total>
</clien>
. . . . . .
```

8. Utilizando los datos devueltos por la consulta anterior, haz un programa Java para obtener el nombre del cliente que más importe tiene que pagar. Guardar el resultado de la consulta en disco, en la carpeta del proyecto, y luego hacer la consulta sobre el fichero que se encuentra en la carpeta. Recuerda para hacer consultas a ficheros en disco pondremos el camino del documento, en este ejemplo el fichero a consultar se encuentra en D:/Misdatos y se llama prueba.xml:

```
for $pru in doc('file:///D:/Misdatos/prueba.xml')/prueba/datos
return $pru
```

ACTIVIDADES DE AMPLIACIÓN

1. Crea la BD MySQL BDARTICULOSCLIEN, copia y ejecuta el script para crear las tablas y las relaciones que se muestran en la figura. El script se encuentra en la carpeta de recursos de la unidad.

Figura 5.24. Tablas y relaciones del ejercicio.

A partir de estas tablas se pide hacer un programa Java para cargar los datos de estas tablas en la BD ***mibasedatos*** de MongoDB. Hay que crear una colección para cada tabla: *artículos*, *clientes* y *ventas*. Y, además, en las ventas hay que añadir en lugar del *codcli* y *codart*, la referencia al cliente y al artículo de las correspondientes colecciones para simular las claves ajenas. Los nombres de las claves de los documentos deben ser los nombres de las columnas de la tabla, convertidas a minúscula, utiliza la *Interfaz* ***ResultSetMetaData*** de jdbc.

Utiliza como identificativo de cada documento (el ***_id)*** las claves primarias de las tablas MySQL para poder hacer así las referencias.

Una vez creados todos los documentos realiza en Java los siguientes métodos con MongoDB:

- Obtén el detalle de cada venta, los datos a obtener son:

Numventa, codcliente, nombre, codarticulo, denominación, fecha venta, unidades, importe.

Importe es el PVP * UNIDADES

- Visualiza para cada artículo, su código, su denominación, el total de unidades vendidas (suma de unidades), el total de importe (suma de unidades * pvp), y el stock actual (resta de stock menos unidades)

- Visualiza para cada cliente: su código, su nombre, el número de compras y el total de las compras que será la suma de unidades * pvp.

CAPÍTULO 6

PROGRAMACIÓN DE COMPONENTES DE ACCESO A DATOS

OBJETIVOS

Utilizar herramientas para desarrollar componentes de acceso a datos.

Programar componentes para gestionar la información almacenada en una base de datos.

Probar y examinar los componentes desarrollados integrándolos en las aplicaciones.

Utilizar un mismo componente en distintas bases de datos.

Definir modelos para comunicar con la base de datos usando el patrón Modelo-Vista-Controlador.

CONTENIDOS

Componente. Propiedades y atributos.

Eventos; asociación de acciones a eventos.

Persistencia del componente.

Creación de componentes. Patrones DAO y Factory.

Empaquetado de componentes.

Patrón Modelo-Vista-Controlador.

RESUMEN

En este capítulo aprenderemos a desarrollar componentes sencillos para realizar operaciones básicas contra una base de datos. Utilizaremos los patrones DAC y Factory para desarrollar componentes e integrarlos en las aplicaciones. Usaremos el patrón MVC para desarrollar componentes web e integrarlos fácilmente en un servidor de contenedores web.

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 vayan 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; a estas operaciones también se las llama 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 se puede construir un componente y luego mejorarlo.

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 se pueden modificar 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.

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 para la propiedad *dnombre*.
- 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 metodo *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*.

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; además, 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()*):

```
import java.io.Serializable;

public class MiPrimerBean implements Serializable {
      private String propiedad;

      public MiPrimerBean() {
            this.propiedad = "";
      }

      public String getPropiedad() {
            return this.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 this.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 this.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){ this.categorias = valor; }
public int[] getCategorias()           { return this.categorias; }

//métodos get y set para acceder a valores individuales
public void setCategorias(int indice, int valor)
                                { this.categorias[indice] = valor; }
public int  getCategorias(int indice)
                                { return this.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, permitiéndoles 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 siguiente:

```
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 oír 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 más 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 oír 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 palabra clave **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 (*setStockactual()*). 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:

- Las clases que implementan **Serializable** deben tener un constructor sin argumentos.
- 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 mínimo se debe generar un pedido. Crearemos dos clases con los siguientes atributos y que implementarán las siguientes interfaces:

Clase *Producto*, implementa la interfaz **Serializable**, es la fuente de eventos:

```
public class Producto implements Serializable {
    private String descripcion;
    private int idproducto;
    private int stockactual;
    private int stockminimo;
    private float pvp;
    . . . . . . . . . .
    //métodos get y set
}
```

Clase *Pedido*, implementa la interfaz **Serializable** y **PropertyChangeListener**, es la clase receptor de eventos:

```
public class Pedido implements Serializable, PropertyChangeListener {
    private int numeropedido;
    private int idproducto;
    private Date fecha;
    private int cantidad;
    private boolean pedir;
    . . . . . . . . . .
    //métodos get y set
}
```

6.4.1. Crear JavaBeans con NetBeans

Abrimos NetBeans, seleccionamos ***File-> New Project-> Java-> Java Class Library*** y pulsamos el botón *Next*; véase Figura 6.3

Figura 6.3. Nueva librería Java

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 *Finish*, Figura 6.4.

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 ***New-> Other-> JavaBeans Objects-> JavaBean Component***; a continuación escribimos el nombre del Bean (*Producto*) y el nombre del paquete del que formará parte, en el ejemplo *MisBeans*. Pulsamos el botón *Finish*, véase Figura 6.5. Se crea el Bean *Producto* con un código por defecto, véase la Figura 6.6.

Figura 6.5. Creamos 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 ***addPropertyChangeListener()*** y ***removePropertyChangeListener()*** vistos anteriormente en el apartado de propiedades ligadas. El código es el siguiente (sin los comentarios generados automáticamente):

```
package MisBeans;

import java.beans.*;
import java.io.Serializable;

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);
    }
}
```

Figura 6.6. Código generado al crear el JavaBean *Producto*.

Modificamos el código del JavaBean. Elimiramos 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 ***Insert Code ->Getter and Setter***, a continuación marcamos los atributos para los que insertaremos dichos métodos y pulsamos el botón *Generate*, 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()***) y dejar el stock con el valor antiguo, sin actualizarlo:

```
public void setStockactual(int valorNuevo) {
    int valorAnterior = this.stockactual;
    this.stockactual = valorNuevo;

    if (this.stockactual < getStockminimo()) //hay que realizar pedido
    {
        propertySupport.firePropertyChange("stockactual",
                    valorAnterior, this.stockactual);
        //dejamos el stock anterior, no actualizamos
        this.stockactual = valorAnterior;
    }
}
```

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 sobrescribir el método *propertyChange()*. En este método es donde indicamos las acciones a realizar cuando el stock actual es menor que el stock mínimo, 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 pce) {
        throw new UnsupportedOperationException("Not supported yet.");
       //To change body of generated methods, choose Tools | Templates.
    }
}
```

Añadimos los siguientes atributos para esta clase (será necesario añadir *import java.util.Date*):

```
private int numeropedido;
private Producto producto;
private Date fecha;
private int cantidad;
```

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.printf("Stock anterior: %d%n", evt.getOldValue());
    System.out.printf("Stock actual: %d%n", evt.getNewValue());

    System.out.printf("REALIZAR PEDIDO EN PRODUCTO: %s%n",
                                        producto.getDescripcion());
}
```

Añadimos también los siguientes constructores (los usaremos después en los ejemplos que hagamos):

```
public Pedido() { }
public Pedido(int numeropedido, Producto producto,
              Date fecha, int cantidad) {
    this.numeropedido = numeropedido;
    this.producto = producto;
    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 **Build**, véase Figura 6.8. NetBeans mostrará en la ventana *Output* los mensajes de la generación con los directorios creados y si se ha realizado correctamente:

```
ant -f D:\\CLASE\\ADAT\\ProyectoNetBeans\\LibreriaJava1 -
Dnb.internal.action.name=rebuild clean jar
init:
deps-clean:
Updating property file:
D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\built-
clean.properties
Deleting directory D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build
clean:
init:
deps-jar:
Created dir: D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build
Updating property file:
D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\built-jar.properties
Created dir: D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\classes
Created dir: D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\empty
Created dir:
D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\generated-
sources\ap-source-output
Compiling 3 source files to
D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\build\classes
compile:
Created dir: D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\dist
Building jar:
D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\dist\LibreriaJava1.jar
jar:
BUILD SUCCESSFUL (total time: 1 second)
```

Figura 6.8. Generar fichero JAR.

Si todo va bien, se generará el fichero *LibreriaJava1.jar* en la carpeta **dist** del proyecto, en el ejemplo en *D:\CLASE\ADAT\ProyectoNetBeans\LibreriaJava1\dist*. Si posteriormente modificamos los JavaBean hemos de pulsar en la opción **Clean and Build** para construir de nuevo la librería.

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 ***Libraries-> Add JAR Folder***, véase Figura 6.9.

Figura 6.9. Incluir el fichero JAR en nuestro proyecto Java.

Hemos de localizar el fichero JAR (carpeta **dist** del proyecto *LibreriaJava1*) pulsar el botón *Abrir*. Una vez realizado, al abrir el nodo ***Libraries*** del proyecto, podemos ver la librería incluida con el paquete *MisBeans* creado anteriormente y la carpeta META-INF con el fichero MANIFEST.MF que se genera al crear el JAR.

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)*.

El siguiente código crea un objeto *Producto*, inicialmente el stock actual es 10 y el mínimo 3. Posteriormente se cambia el stock actual a 2 que es menor que el mínimo, esto generará un evento que será detectado por el pedido y se mostrará un mensaje en pantalla indicando que hay que realizar pedido del producto:

```
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();
     pedido.setProducto(producto);

     producto.addPropertyChangeListener(pedido); //
     //cambiamos el stock actual, le camos el valor 2 --anterior es 10
     producto.setStockactual(2);
   }
}
```

Y lo ejecutamos. Al ejecutarlo se visualiza:

```
Stock anterior: 10
Stock actual: 2
REALIZAR PEDIDO EN PRODUCTO: Dabber Sur Femme 2011
```

ACTIVIDAD 6.1

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 fichero JAR (vimos anteriormente cómo 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 fichgenerado.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.
- **fichgenerado.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 un nuevo JavaBean y una clase auxiliar al paquete *MisBeans* creado en el apartado anterior, modificaremos el JavaBean *Pedido* para que inserte un pedido en una base de datos. El JavaBean a crear se llama *Venta*. Al crearlo eliminamos todo el código de la clase y añadimos **PropertyChangeListener** a la cláusula **implements**; sobrescribiremos el método ***propertyChange()*** en donde indicaremos la acción a realizar cuando el stock actual del producto sea menor que el stock mínimo. 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;
package MisBeans;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
```

```
import java.io.Serializable;
import java.util.Date;

public class Venta implements Serializable, PropertyChangeListener{
    private int numeroventa;
    private int idproducto;
    private Date fechaventa;
    private int cantidad;
    private String observaciones;

    public Venta() {
    }

    public Venta(int numeroventa, int idproducto,
            Date fechaventa, int cantidad, String observaciones) {
        this.numeroventa = numeroventa;
        this.idproducto = idproducto;
        this.fechaventa = fechaventa;
        this.cantidad = cantidad;
        this.observaciones = observaciones;
    }

    public int getNumeroventa() {
        return this.numeroventa;
    }

    public void setNumeroventa(int numeroventa) {
        this.numeroventa = numeroventa;
    }

    public int getIdproducto() {
        return this.idproducto;
    }

    public void setIdproducto(int idproducto) {
        this.idproducto = idproducto;
    }

    public Date getFechaventa() {
        return this.fechaventa;
    }

    public void setFechaventa(Date fechaventa) {
        this.fechaventa = fechaventa;
    }

    public int getCantidad() {
        return this.cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    public String getObservaciones() {
```

```
        return observaciones;
    }

    public void setObservaciones(String observaciones) {
        this.observaciones = observaciones;
    }

     public void propertyChange(PropertyChangeEvent evt) {
        //el stock actual del producto es < que el stock minimo
        //la venta queda pendiente hasta que haya stock
        this.observaciones = "Pendiente de pedido por falta de stock";
    }
}//Venta
```

Vamos a utilizar una BDOO **Neodatis** para gestionar las ventas y pedidos de una serie de productos. Creamos una nueva clase donde incluiremos las operaciones a realizar con la base de datos, como por ejemplo insertar un producto, insertar un pedido, insertar una venta, calcular el número del pedido, calcular el número de la venta, actualizar el stock, etc. La clase se llama *BaseDatos* y es la siguiente:

```
package MisBeans;

import java.math.BigDecimal;
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.ObjectValues;
import org.neodatis.odb.Values;
import org.neodatis.odb.impl.core.query.values.ValuesCriteriaQuery;

public class BaseDatos {
    static ODB odb;

    public BaseDatos() {
        //Abrir base de datos
        this.odb = ODBFactory.open("Producto_Ped.BD");
    }

    //Devuelve el objeto ODB
    public static ODB getOdb() {
        return BaseDatos.odb;
    }

    //Cerrar base de datos
    public static void closeBD() {
        odb.close();
    }

    //Obtiene el número de pedido para el próximo pedido
    public static int obtenerNumeroPedido() {
        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;
    }//

    //Obtiene el número de venta para la próxima venta
    public int obtenerNumeroVenta() {
        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;
    }//

    //Inserta un Producto
    public void insertaProducto(Producto producto) {
        odb.store(producto); //Almacenar producto
        odb.commit();
    }

    //Inserta un Pedido
    public static void insertaPedido(Producto producto, int cantidad) {
        Pedido pedido = new Pedido (obtenerNumeroPedido(), producto,
                                    getCurrentDate(), cantidad);
        odb.store(pedido); //Almacenar pedido
        System.out.printf("PEDIDO GENERADO para el Producto: %s%n",
                          producto.getDescripcion());
        odb.commit();
    }

    //Inserta una Venta
    public void insertaVenta(Producto producto, int cantidad) {
      int numeroventa = obtenerNumeroVenta();
      Venta ven = new Venta(numeroventa, producto.getIdproducto(),
                        getCurrentDate(), cantidad, "");

      producto.addPropertyChangeListener(ven);//añadir oyente

      actualizarStock(producto, cantidad);
      odb.store(ven); //Almacenar venta
      System.out.printf("VENTA %d INSERTADA, Observaciones: %s %n",
                numeroventa, ven.getObservaciones());
      odb.commit();
    }

    //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());
    }

    //Actualizar stock del producto
    private void actualizarStock(Producto producto, int cantidad) {
        Pedido pedido = new Pedido();
        pedido.setProducto(producto);
```

```
        pedido.setCantidad(cantidad);

        producto.addPropertyChangeListener(pedido);//añadir oyente

        //cálculo de stock
        int nuevostock = producto.getStockactual() - cantidad;
        producto.setStockactual(nuevostock);//cambiamos el stock actual

        odb.store(producto); //almacenar producto actualizado
        odb.commit();
    }
}//BaseDatos
```

El método *obtenerNumeroVenta()* devuelve el número de venta que se asignará a una venta, y se requiere al insertar la venta. Igualmente el método *obtenerNumeroPedido()* devuelve el número de pedido que se asignará al pedido que se va a crear. Ambos valores se obtienen al consultar el máximo valor asignado y sumarle 1.

El método *actualizarStock()* crea un objeto *Pedido* y lo añade a la lista de oyentes. Antes se le asigna valor al atributo *producto* y *cantidad* del pedido. Se calcula el nuevo stock actual y se actualiza. Se generará un pedido si el stock actual es menor que el mínimo y además no se actualizará el stock.

Modificamos el código del método ***propertyChange()*** de la clase Pedido, añadimos la línea para insertar un pedido para el producto:

```
public void propertyChange(PropertyChangeEvent evt) {
        System.out.printf("Stock anterior: %d%n", evt.getOldValue());
        System.out.printf("Nuevo Stock: %d (menor que el minimo) %n",
                          evt.getNewValue());

        //insertar un pedido
        BaseDatos.insertaPedido(producto, cantidad);
}
```

Añadimos al fichero **MANIFEST.MF** las siguientes líneas con las nuevas clases:

```
Name: MisBeans/Venta.class
Java-Bean: True

Name: MisBeans/BaseDatos.class
Java-Bean: False
```

Y compilamos las clases que tendrá el paquete, como se usará una base de datos **Neodatis** necesitamos la librería *neodatis-odb-1.9.30.689.jar*, suponiendo que está en la carpeta *D:/UNI6* escribimos las siguientes líneas desde la línea de comandos del DOS:

```
D:\UNI6>SET CLASSPATH=.;D:\UNI6\MisBeans;D:\UNI6\neodatis-odb-1.9.30.689.jar
D:\UNI6>javac MisBeans/Producto.java
D:\UNI6>javac MisBeans/Venta.java
D:\UNI6>javac MisBeans/BaseDatos.java
D:\UNI6>javac MisBeans/Pedido.java
D:\UNI6>jar cfm MisBeans.jar META-INF/MANIFEST.MF MisBeans/*.class
```

Y generamos de nuevo el paquete JAR con la siguiente orden:

```
D:\UNI6>jar ctm MisBeans.jar META INF/MANIFEST MF MisBeans/*.class
```

Esto no será necesario si creamos la librería JAR desde el entorno gráfico. Sí será necesario incluir la librería de **Neodatis**.

Creo un nuevo proyecto Java de nombre *PROD_PED_VEN* y añado las librerías que voy a necesitar: *MisBeans.jar* y *neodatis-odb-1.9.30.689.jar*. En primer lugar realizo un programa Java para dar de alta los productos, *LLenarProductos.java*, se almacenarán en la base de datos de nombre *Producto_Ped.BD*:

```
import MisBeans.BaseDatos;
import MisBeans.Producto;

public class LLenarProductos {
    public static void main(String[] args) {
     BaseDatos bd = new BaseDatos();

     Producto p1 = new Producto(1, "Duruss Cobalt", 10, 3, 220);
     Producto p2 = new Producto(2, "Varlion Avant Carbon", 5, 2, 176);
     Producto p3 = new Producto(3, "Star Vie Pyramid R50", 20, 5, 193);
     Producto p4 = new Producto(4, "Dunlop Titan", 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
     bd.insertaProducto(p1);
     bd.insertaProducto(p2);
     bd.insertaProducto(p3);
     bd.insertaProducto(p4);
     bd.insertaProducto(p5);
     bd.insertaProducto(p6);

     bd.closeBD(); // Cerrar BD
    }
}
```

Para visualizar los productos creo el programa *VerProductos.java*:

```
import MisBeans.BaseDatos;
import org.neodatis.odb.ODB;
import org.neodatis.odb.Objects;
import MisBeans.Producto;

public class VerProductos {
    public static void main(String[] args) {
      BaseDatos bd = new BaseDatos();
      ODB odb = bd.getOdb();

      //recuperamos todos los objetos
      Objects<Producto> objects = odb.getObjects(Producto.class);
      System.out.printf("Número de Productos: %d%n", objects.size());

      int i = 1;
```

```
        // visualizamos los productos
        while (objects.hasNext()) {
            Producto pro = objects.next();
            System.out.printf("%d: %s, STOCK ACTUAL: %d, MINIMO: %d, Pvp:
               %.2f%n", i++, pro.getDescripcion(), pro.getStockactual(),
                             pro.getStockminimo(), pro.getPvp());
        }
          bd.closeBD(); // Cerrar BD
    }
}
```

Realizamos otros dos programas Java para mostrar los pedidos y las ventas, *VerPedidos.java* y *VerVentas.java* son similares a este, solo que los objetos a recuperar de la base de datos son distintos. 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 y si no hay stock se generará automáticamente un pedido del producto con la cantidad solicitada en la venta, además, en el campo de *observaciones* de la venta se indicará lo que ha ocurrido, esto se realiza en los métodos ***propertyChange()*** de la venta y del pedido. El código es el siguiente:

```
import MisBeans.BaseDatos;
import org.neodatis.odb.ODB;
import org.neodatis.odb.Objects;
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 MisBeans.Producto;

public class LLenarVentas {
    public static void main(String[] args) {
        // Datos de entrada para la venta
        int idproducto = 2;  //idproducto
        int cantidad = 4;    //cantidad

        BaseDatos bd = new BaseDatos();
        ODB odb = bd.getOdb();
        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.printf("ID=> %d: %s, STOCK-ACT: %d, STOCK-MIN:
             %d, PVP:%.2f %n", idproducto, pro.getDescripcion(),
             pro.getStockactual(), pro.getStockminimo(), pro.getPvp());
          System.out.printf("Cantidad: %d%n",cantidad);

          //se inserta la venta
          bd.insertaVenta(pro, cantidad);

        } catch (IndexOutOfBoundsException e) {
            System.out.println("NO EXISTE EL PRODUCTO");
        } finally {
```

```
            bd.closeBD(); // Cerrar BD
        }
    }//
}//fin LLenarVentas
```

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 inserta la venta en la base de datos.

La ejecución del programa muestra la siguiente salida:

```
ID=> 2: Varlion Avant Carbon, STOCK-ACT: 5, STOCK-MIN: 2, PVP:176,00
Cantidad: 4
Stock anterior: 5
Nuevo Stock: 1 (menor que el minimo)
PEDIDO GENERADO para el Producto: Varlion Avant Carbon
VENTA 1 INSERTADA, Observaciones: Pendiente de pedido por falta de
stock
```

ACTIVIDAD 6.2

Como ejercicio se propone que el IDPRODUCTO y la CANTIDAD se acepten desde los argumentos de *main()*.

6.7. PATRÓN DATA ACCESS OBJECT (DAO)

El componente anterior se diseñó para trabajar con una base de datos **Neodatis**, si queremos trabajar con otra base de datos, por ejemplo, con una base de datos SQL, tendríamos que realizar cambios específicos para acceder a esa base de datos.

J2EE proporciona una serie de patrones para ayudarnos a construir aplicaciones Java. El patrón **DAO** nos permite acceder a datos que pueden estar localizados en distintas fuentes: en una base de datos SQL, orientada a objetos, en una base de datos XML, ficheros, etc. El **DAO**, o lo que es lo mismo, el **Objeto de Acceso a Datos** encapsulará el código requerido para localizar y acceder a la fuente de datos. Toda la lógica del negocio usará el **DAO** para recuperar y almacenar datos.

En general, por cada objeto de negocio crearemos un **DAO** distinto, por ejemplo, si queremos realizar la gestión de departamentos y empleados crearemos un **DAO** para departamentos y otro para empleados: *DepartamentoDAO* y *EmpleadoDAO*. Si queremos llevar a cabo la gestión de productos, pedidos y ventas podemos crear un **DAO** para cada uno: *ProductoDAO*, *PedidoDAO* y *VentasDAO*. Estos **DAO** los podremos utilizar para cualquier base de datos, ya sea Oracle, Neodatis, SQlite, XML, etc.

Para crear un **DAO** necesitamos 3 elementos:

- Un **objeto DAO**, es un simple **POJO** (*Plain Old Java Object*) que contiene los atributos, constructores y métodos *set* y *get*. Por ejemplo, para llevar a cabo la gestión de departamentos necesitamos un objeto *Departamento* con datos de departamentos *deptno*, *dnombre* y *loc*.
- Una **interfaz** que defina las operaciones que se pueden realizar en un **objeto DAO**. Por ejemplo, crearemos la interfaz *DepartamentoDAO* que definirá las operaciones a realizar con un objeto *Departamento*: insertar, modificar, eliminar, consultar, etc.
- Una **clase que implemente la interfaz**, esta clase es la responsable de obtener los datos de un origen de datos que puede ser una base de datos o cualquier otro mecanismo de almacenamiento. La clase la llamaremos *DepartamentoImpl* e implementará los métodos definidos en la interfaz para interactuar con una base de datos Neodatis. El diagrama de clases de los 3 elementos se muestra en la Figura 6.10.

Figura 6.10. Diagrama de clases ejemplo del objeto DAO.

6.7.1. Ejemplo de aplicación

Veamos a continuación cómo se desarrollan estos 3 elementos. Creamos desde NetBeans una nueva librería Java, la llamamos *DEP_DAO1*. A continuación creamos un paquete de nombre *Dep* y creamos la clase *Departamento.java* con los siguientes atributos y métodos:

```
package Dep;
import java.io.Serializable;
//Objeto DAO
public class Departamento implements Serializable{
    int deptno;
    String dnombre;
    String loc;

    public Departamento() {  }
    public Departamento(int deptno, String dnombre, String loc) {
        this.deptno = deptno;
        this.dnombre = dnombre;
```

```
        this.loc = loc;
    }

    public int getDeptno() { return deptno; }
    public void setDeptno(int deptno) { this.deptno = deptno; }
    public String getDnombre() { return dnombre; }

    public void setDnombre(String dnombre) {this.dnombre = dnombre;}
    public String getLoc() {return loc;}
    public void setLoc(String loc) {this.loc = loc;}
}
```

Añadimos la interfaz *DepartamentoDAO.java* con las siguientes operaciones:

- ***public boolean InsertarDep(Departamento dep)***: recibe un objeto *Departamento* para insertarlo en la fuente de datos que sea (una base de datos SQL, XML, orientada a objetos, etc.) Devuelve *true* si la operación se ha realizado correctamente, en caso contrario devuelve *false*.
- ***public boolean EliminarDep(int deptno)***: elimina un departamento de la fuente de datos, recibe el número de departamento a eliminar. Devuelve *true* o *false* dependiendo de cómo se haya realizado la operación.
- ***public boolean ModificarDep(int deptno, Departamento dep)***: modifica el departamento indicado en *deptno*, los datos a modificar están en el objeto *dep*. Devuelve *true* o *false* dependiendo de cómo se haya realizado la operación.
- ***public Departamento ConsultarDep(int deptno)***: recibe un número de departamento y devuelve el objeto departamento cuyo número coincida con *deptno*.

```
package Dep;
//interfaz DAO
public interface DepartamentoDAO {
    public boolean InsertarDep(Departamento dep);
    public boolean EliminarDep(int deptno);
    public boolean ModificarDep(int deptno, Departamento dep);
    public Departamento ConsultarDep(int deptno);
}
```

Desarrollamos la clase que implemente la interfaz, la llamamos *DepartamentoImpl.java*, en este caso la implementación se realiza sobre una base de datos Neodatis, es necesario añadir al proyecto la librería JAR de Neodatis. Se desarrollan todos los métodos definidos en la interfaz para una base de datos llamada *Departamento.BD*, también se añade un método para obtener la conexión a la BD que previamente se crea en el constructor. El código es el siguiente:

```
package Dep;
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

//implementación de la interfaz
public class DepartamentoImpl implements DepartamentoDAO {
```

```
static ODB bd;

public DepartamentoImpl() {
   bd = ODBFactory.open("Departamento.BD");
}

public static ODB crearConexion() {
     return bd; //Devuelve el objeto ODB
}

@Override
public boolean InsertarDep(Departamento dep) {
  bd.store(dep);
  bd.commit();
  System.out.printf("Departamento: %d Insertado", dep.getDeptno());
  return true;
}

@Override
public boolean EliminarDep(int deptno) {
  boolean valor = false;
  IQuery query = new
    CriteriaQuery(Departamento.class, Where.equal("deptno", deptno));
  Objects<Departamento> objetos = bd.getObjects(query);
  try {
      Departamento depart = (Departamento) objetos.getFirst();
      bd.delete(depart);
      bd.commit();
      valor = true;
      System.out.printf("Departamento: %d eliminado %n", deptno);
  } catch (IndexOutOfBoundsException i) {
      System.out.printf("Departamento a eliminar: %d No existe %n",
                        deptno);
  }
  return valor;
}

@Override
public boolean ModificarDep(int deptno, Departamento dep) {
  boolean valor = false;
  IQuery query = new CriteriaQuery(Departamento.class,
                     Where.equal("deptno", deptno));
  Objects<Departamento> objetos = bd.getObjects(query);
  try {
      Departamento depart = (Departamento) objetos.getFirst();
      depart.setDnombre(dep.getDnombre());
      depart.setLoc(dep.getLoc());
      bd.store(depart); // actualiza el objeto
      valor = true;
      bd.commit();
  } catch (IndexOutOfBoundsException i) {
      System.out.printf("Departamento: %d No existe%n", deptno);
  }
  return valor;
}
```

```
  @Override
  public Departamento ConsultarDep(int deptno) {
    IQuery query = new CriteriaQuery(Departamento.class,
                       Where.equal("deptno", deptno));
    Objects<Departamento> objetos = bd.getObjects(query);
    Departamento dep = new Departamento();
    if (objetos != null) {
        try {
            dep = (Departamento) objetos.getFirst();
        } catch (IndexOutOfBoundsException i) {
            System.out.printf("Departamento: %d No existe%n", deptno);
            dep.setDnombre("no existe");
            dep.setDeptno(deptno);
            dep.setLoc("no existe");
        }
    }
    return dep;
  }
}//
```

Una vez creadas las 3 clases, generamos el JAR, pulsamos con el botón derecho del ratón sobre el proyecto y a continuación pulsamos sobre la opción **Clean and Build**.

Por último, creamos la clase que probará el **DAO**. La llamamos *DEP_DAO1_PRUEBA.java*. Creamos un nuevo proyecto en NetBeans y añadimos el JAR creado anteriormente y el JAR de Neodatis. Los dos proyectos creados con sus librerías nos deben quedar como se muestra en la Figura 6.11.

Figura 6.11. Proyecto ejemplo de uso del componente *DEP_DAO1.jar*.

El código de la clase de prueba es el siguiente:

```
package dep_dao1_prueba;

import Dep.Departamento;
import Dep.DepartamentoDAO;
import Dep.DepartamentoImpl;
```

```
public class DEP_DAO1_PRUEBA {
    public static void main(String[] args) {
      DepartamentoDAO depDAO = new DepartamentoImpl();

      //INSERTAR
      Departamento dep1 = new Departamento(17, "NÓMINAS", "SEVILLA");
      depDAO.InsertarDep(dep1);

      //CONSULTAR
      Departamento dep2 = depDAO.ConsultarDep(17);
      System.out.printf("Dep: %d, Nombre: %s, Loc: %s %n",
                  dep2.getDeptno(),dep2.getDnombre(), dep2.getLoc());

      //MODIFICAR
      dep2.setDnombre("nuevonom");
      dep2.setLoc("nuevaloc");
      depDAO.ModificarDep(17, dep2);

      //ELIMINAR
      depDAO.EliminarDep(17);
    }
}
```

La ejecución muestra la siguiente salida:

```
Departamento: 17 Insertado
Dep: 17, Nombre: NÓMINAS, Loc: SEVILLA
Departamento: 17 eliminado
```

Hasta aquí parece que esto no nos resuelve el problema de usar un mismo componente para acceder a distintas bases de datos, de momento el componente *DEP_DAO1.jar* solo puede ser usado para una base de datos Neodatis. Para que sea independiente del almacén de datos, es decir, para que se pueda usar en una base de datos distinta a la anterior necesitamos la ayuda del patrón **Factory**.

6.8. PATRÓN FACTORY

Una clase **Factory** (fábrica o factoría) es una clase que se encarga de crear instancias de objetos que pueden ser de esta misma clase o de otras. Existen diferentes "tipos" de fábricas[1]:

- **Simple Factory**: clase utilizada para crear nuevas instancias de objetos.
- **Factory Method**: define una interfaz para crear objetos, pero deja que sean las subclases las que deciden qué clases instanciar.
- **Abstract Factory**: Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

El patrón DAO se puede flexibilizar adoptando los patrones **Abstract Factory** [GoF] y **Factory Method** [GoF]. Cuando el almacenamiento subyacente no está sujeto a cambios de una implementación a otra, este patrón se puede implementar usando el patrón **Factory Method** para

[1]Fuente: https://msdn.microsoft.com/es-es/library/bb972258.aspx.

producir el número de DAOs que necesita la aplicación. Cuando el almacenamiento subyacente puede cambiar de una implementación a otra, podemos utilizar el patrón **Abstract Factory**[2].

Usamos el patrón **Abstract Factory** cuando tenemos varias bases de datos. En el siguiente ejemplo vamos a desarrollar un componente que nos permita acceder a dos bases de datos diferentes, Neodatis y MySQL. El diagrama de clases que se utilizarán se muestra en la Figura 6.12. En la parte superior tenemos la clase abstracta *DAOFactory* que producirá DAOs como *DepartamentoDAO* (o *EmpleadoDAO*) y creará instancias de objetos *NeodatisDAOFactory* y *SqlDbDAOFactory* dependiendo de la base de datos a la que se accederá.

Esta estrategia utiliza además el patrón **Factory Method** en las distintas factorías producidas. Es decir, dentro de las factorías encontraremos los métodos que cada subclase debe sobrescribir (métodos definidos en *DAOFactory*, en este caso *getDepartamentoDAO()*) creando instancias de objetos *NeodatisDepartamntoImpl* y *SqlDbDepartamentoImpl* que implementarán los métodos definidos en el DAO, o sea en *DepartamentoDAO*.

Figura 6.12. Implementación de *Factory* para *DAO* usando la estrategia *Abstract Factory*.

6.8.1. Ejemplo de aplicación

Creamos desde NetBeans un nuevo proyecto de librería y le damos el nombre *DEP_DAO2*. Incluimos las clases *DepartamentoDAO* y *Departamento* del ejemplo anterior. El código para la clase abstracta *DAOFactory.java* se muestra a continuación, se define una constante para cada base de datos, un método abstracto que produce el DAO *DepartamentoDAO*, este método tendrá que ser redefinido por todas las subclases; y un método que devuelve una instancia del objeto de

[2]Fuente: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

base de datos con el que se va a trabajar. El método se llama *getDAOFactory()* y recibe un entero que indica la base de datos con la que trabajaremos (1 para MySQL y 2 para Neodatis), se define como estático:

```
package Dep;

public abstract class DAOFactory {
  // Bases de datos soportadas
  public static final int MYSQL = 1;
  public static final int NEODATIS = 2;

  public abstract DepartamentoDAO getDepartamentoDAO();

  public static DAOFactory getDAOFactory(int bd) {
    switch (bd) {
      case MYSQL    :
           return  new SqlDbDAOFactory();
      case NEODATIS:
           return  new NeodatisDAOFactory();
      default       :
          return null;
    }
  }
}
```

Para usar esta clase en un programa Java y conectarnos a la base de datos Neodatis tendríamos que invocar al método estático *getDAOFactory()* llevando como parámetro la constante definida para Neodatis:

```
DAOFactory bd = DAOFactory.getDAOFactory(DAOFactory.NEODATIS);
```

Si queremos conectarnos con la base de datos MySQL escribimos lo mismo, solo cambia el valor de la constante:

```
DAOFactory bd = DAOFactory.getDAOFactory(DAOFactory.MYSQL);
```

La clase *SqlDbDAOFactory.java* extiende *DAOFactory*, por tanto, tiene que sobrescribir el método *getDepartamentoDAO()*, este método devuelve una instancia del objeto que implementará los métodos definidos en la interface *DepartamentoDAO*, en este caso se llama *SqlDbDepartamentoImpl.java* (es en esta clase donde se desarrollan los métodos para realizar operaciones sobre una base de datos MySQL).

En la clase *SqlDbDAOFactory* se definen los parámetros de la conexión a la base de datos MySQL (en el ejemplo la base de datos se llama *unidad6* al igual que el usuario y la clave) y se crea la conexión en el método *crearConexion()* que devuelve un objeto *Connection*. Tanto el método como la conexión se declaran estáticos:

```
package Dep;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import java.util.logging.Level;
```

```
import java.util.logging.Logger;

public class SqlDbDAOFactory extends DAOFactory {
    static Connection conexion = null;
    static String DRIVER = "";
    static String URLDB = "";
    static String USUARIO = "unidad6";
    static String CLAVE = "unidad6";

    public SqlDbDAOFactory() {
        DRIVER = "com.mysql.jdbc.Driver";
        URLDB = "jdbc:mysql://localhost/unidad6";
    }

    // crear la conexión si no está creada
    public static Connection crearConexion() {
      if (conexion == null) {
        try {
           Class.forName(DRIVER); // Cargar el driver
        } catch (ClassNotFoundException ex) {
             Logger.getLogger(SqlDbDAOFactory.class.getName()).
                 log(Level.SEVERE, null, ex);
        }

        try {
           conexion = DriverManager.getConnection
                          (URLDB, USUARIO, CLAVE);
        } catch (SQLException ex) {
           System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:%n");
           System.out.printf("Mensaje   : %s %n", ex.getMessage());
           System.out.printf("SQL estado: %s %n", ex.getSQLState());
           System.out.printf("Cód error : %s %n", ex.getErrorCode());
        }
      }
      return conexion;
    }
    @Override
    public DepartamentoDAO getDepartamentoDAO() {
        return new SqlDbDepartamentoImpl();
    }
}
```

La clase *NeodatisDAOFactory.java* extiende *DAOFactory*, y por tanto tiene que sobreescribir el método *getDepartamentoDAO()*. En este caso el método devuelve una instancia de un objeto *NeodatisDepartamentoImpl*; en esta clase se implementan los métodos para acceder a la base de datos Neodatis (similar a la clase *DepartamentoImpl* definida anteriormente). En la clase *NeodatisDAOFactory* se definen los parámetros para la conexión a la base de datos y se crea la conexión si no se ha creado anteriormente. Igual que antes se define el método para crear la conexión, la variable de conexión y el método se definen como estáticos:

```
package Dep;

import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
```

```
public class NeodatisDAOFactory extends DAOFactory {
    static ODB odb = null;

    public NeodatisDAOFactory() { }

    public static ODB crearConexion() {
        if (odb == null) {
            odb = ODBFactory.open("Departamento.BD");
        }
        return odb;
    }

    @Override
    public DepartamentoDAO getDepartamentoDAO() {
        return new NeodatisDepartamentoImpl();
    }
}
```

La clase *NeodatisDepartamentoImpl*, es similar a *DepartamentoImpl*, cambia el constructor que obtiene la conexión de *NeodatisDAOFactory*:

```
public class NeodatisDepartamentoImpl implements DepartamentoDAO {
    static ODB bd;

    public NeodatisDepartamentoImpl() {
        bd = NeodatisDAOFactory.crearConexion();
    }

    @Override
    public boolean InsertarDep(Departamento dep) { . . . .
    @Override
    public boolean InsertarDep(Departamento dep) {. . . .
    @Override
    public boolean EliminarDep(int deptno) {. . . .
    @Override
    public boolean ModificarDep(int deptno, Departamento dep) {. . .
}
```

Por último, la clase *SqlDbDepartamentoImpl* implementa las operaciones para acceder a una base de datos MySQL (aunque el código es válido para cualquier base de datos basada en SQL). En el constructor se obtiene la conexión invocando al método *crearConexion()* de *SqlDbDAOFactory*:

```
package Dep;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SqlDbDepartamentoImpl implements DepartamentoDAO {
    Connection conexion;

    public SqlDbDepartamentoImpl() {
```

```
        conexion = SqlDbDAOFactory.crearConexion();
    }

    @Override
    public boolean InsertarDep(Departamento dep) {
        boolean valor = false;
        String sql = "INSERT INTO departamentos VALUES(?, ?, ?)";
        PreparedStatement sentencia;
        try {
            sentencia = conexion.prepareStatement(sql);
            sentencia.setInt(1, dep.getDeptno());
            sentencia.setString(2, dep.getDnombre());
            sentencia.setString(3, dep.getLoc());
            int filas = sentencia.executeUpdate();
            if (filas > 0) {
                valor = true;
                System.out.printf("Departamento %d insertado%n",
                                  dep.getDeptno());
            }
            sentencia.close();

        } catch (SQLException e) { MensajeExcepcion(e); }

        return valor;
    }

    @Override
    public boolean EliminarDep(int deptno) {
        boolean valor = false;
        String sql = "DELETE FROM departamentos WHERE dept_no = ? ";
        PreparedStatement sentencia;
        try {
            sentencia = conexion.prepareStatement(sql);
            sentencia.setInt(1, deptno);
            int filas = sentencia.executeUpdate();
            if (filas > 0) {
              valor = true;
              System.out.printf("Departamento %d eliminado%n",deptno);
            }
            sentencia.close();
        } catch (SQLException e) { MensajeExcepcion(e); }

        return valor;
    }

    @Override
    public boolean ModificarDep(int num, Departamento dep) {
        boolean valor = false;
        String sql = "UPDATE departamentos SET dnombre= ?, loc = ?
                      WHERE dept_no = ? ";
        PreparedStatement sentencia;
        try {
            sentencia = conexion.prepareStatement(sql);
            sentencia.setInt(3, num);
            sentencia.setString(1, dep.getDnombre());
```

```
            sentencia.setString(2, dep.getLoc());
            int filas = sentencia.executeUpdate();
            if (filas > 0) {
                valor = true;
                System.out.printf("Departamento %d modificado%n", num);
            }
            sentencia.close();
        } catch (SQLException e) { MensajeExcepcion(e); }

        return valor;
    }

    @Override
    public Departamento ConsultarDep(int deptno) {
        String sql = "SELECT dept_no, dnombre, loc
                      FROM departamentos WHERE dept_no = ?";
        PreparedStatement sentencia;
        Departamento dep = new Departamento();
        try {
            sentencia = conexion.prepareStatement(sql);
            sentencia.setInt(1, deptno);
            ResultSet rs = sentencia.executeQuery();
            if (rs.next()) {
                dep.setDeptno(rs.getInt("dept_no"));
                dep.setDnombre(rs.getString("dnombre"));
                dep.setLoc(rs.getString("loc"));
            }
            else
                System.out.printf("Departamento: %d No existe%n",
                                  deptno);

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

        } catch (SQLException e) { MensajeExcepcion(e); }
        return dep;
    }

    private void MensajeExcepcion(SQLException e) {
       System.out.printf("HA OCURRIDO UNA EXCEPCIÓN:%n");
       System.out.printf("Mensaje   : %s %n", e.getMessage());
       System.out.printf("SQL estado: %s %n", e.getSQLState());
       System.out.printf("Cód error : %s %n", e.getErrorCode());
    }
}
```

Para probar este componente en un proyecto Java hemos de crear primero el JAR y después importar al proyecto el JAR del componente y la librería o el JAR de la base de datos con la que trabajaremos. En el programa Java primero hay que crear un objeto *DAOFactory* usando el método *getDAOFactory()* al que le enviaremos la base de datos con la que vamos a trabajar, y después un objeto *DepartamentoDAO* para acceder a las distintas operaciones sobre los departamentos.

El siguiente ejemplo inserta un departamento usando el método *InsertarDep()* y a continuación visualiza los datos del departamento cuyo número se introduce por teclado. Se usará el método *ConsultarDep()* que devuelve un objeto *Departamento* para obtener los datos. El proceso finaliza cuando el número introducido sea menor o igual que 0:

```
import Dep.DAOFactory;
import Dep.Departamento;
import Dep.DepartamentoDAO;
import java.util.Scanner;

public class PruebaSQLDB {
    public static void main(String[] args) {
        DAOFactory bd = DAOFactory.getDAOFactory(DAOFactory.MYSQL);
        DepartamentoDAO depDAO = bd.getDepartamentoDAO();

        //crear departamento
        Departamento dep = new Departamento(17, "NÓMINAS", "SEVILLA");
        depDAO.InsertarDep(dep);

        Scanner sc = new Scanner(System.in);
        int entero = 1;
        //Visualizar departamentos leidos por teclado
        while (entero > 0) {
            System.out.println("Departamento: ");
            entero = sc.nextInt();
            dep = depDAO.ConsultarDep(entero);
            System.out.printf("Dep: %d, Nombre: %s, Loc: %s %n",
                    dep.getDeptno(),
                    dep.getDnombre(), dep.getLoc());
        }
    }
}
```

Para probarlo en la base de datos Neodatis solo tendríamos que cambiar la primera línea indicando la base de datos con la que vamos a trabajar:

```
DAOFactory bd = DAOFactory.getDAOFactory(DAOFactory.NEODATIS);
```

ACTIVIDAD 6.3

Partiendo del componente anterior, amplíalo y realiza los cambios necesarios para que dé soporte a una base de datos Oracle y otra SQLite. Crea para ello desde NetBeans un nuevo proyecto de librería, le asignamos el nombre *DEP_DAO3* y copiamos las clases anteriores. Para la base de datos Oracle crea un usuario de nombre y clave UNIDAD6 y crea también las tablas *Empleados* y *Departamentos*. En SQLite usaremos la base de datos creada en el Capítulo 2 que también contiene empleados y departamentos.

Después haz un programa Java para probar cualquiera de las bases de datos soportadas por el componente. El programa pedirá por teclado la base de datos a probar, se introducirá un número 1 para probar MYSQL, 2 para NEODATIS, 3 para ORACLE y 4 para SQLITE. Y una vez comprobado (si el número introducido no es uno de los anteriores el programa finaliza) se pedirá el departamento a consultar. El programa visualizará los datos del departamento.

Recuerda que para probar todas las bases de datos se necesita su conector.

A veces nos interesa empaquetar varios JAR en uno solo JAR. Por ejemplo, en la actividad anterior si se nos olvida incluir el JAR de la base de datos con la que vamos a trabajar, el programa mostrará errores.

Para evitar esto podemos empaquetar dentro del componente todos los JAR de las bases de datos en un único fichero JAR. Partimos del proyecto de librería *DEP_DAO3* que usa 4 bases de datos, véase Figura 6.13.

Figura 6.13. Componente *DEP_DAO3*.

Pulsamos con el botón derecho del ratón sobre el proyecto, y seleccionamos ***Properties->Libraries***, se muestran todas las librerías externas del proyecto. Pulsamos en el botón *Browse* para que se muestre la carpeta de librerías (que por defecto es *.\lib*) Figura 6.14, dejamos el nombre por defecto, a continuación pulsamos el botón *Next* y seguidamente el botón *Finish* para que se copien las librerías externas en la carpeta *lib* del componente.

Figura 6.14. Añadir carpeta de librerías al componente.

Se muestran todas las librerías copiadas a la carpeta *lib*, véase Figura 6.15. A continuación pulsamos sobre *Packaging* y marcamos la casilla *Copy Dependent Libraries*, pulsamos sobre el botón *OK* para finalizar, véase Figura 6.16. Con esto se han añadido las librerías externas al componente, pero aún no se ha creado el fichero JAR.

Figura 6.15. Carpeta *lib* con las librerías externas.

Figura 6.16. Copisr las librerías externas.

A continuación pulsamos sobre la pestaña *Files* de NetBeans para localizar el fichero **build.xml**, véase Figura 6.17. Añadimos el siguiente texto antes de la etiqueta *</project>*, nos fijamos en la segunda línea, en *value* se indica el nombre del fichero JAR que contendrá las librerías incluidas en la carpeta *lib*, en el ejemplo se asigna el nombre de *ConectoresBD*:

```
<target name="-post-jar">
    <property name="store.jar.name" value="ConectoresBD"/>

    <property name="store.dir" value="dist"/>
    <property name="store.jar"
              value="${store.dir}/${store.jar.name}.jar"/>

    <echo message="Packaging ${application.title} into a single JAR at
${store.jar}"/>

    <jar destfile="${store.dir}/temp_final.jar" filesetmanifest="skip">
        <zipgroupfileset dir="dist" includes="*.jar"/>
        <zipgroupfileset dir="dist/lib" includes="*.jar"/>

        <manifest>
            <attribute name="Main-Class" value="${main.class}"/>
        </manifest>
```

```
    </jar>

    <zip destfile="${store.jar}">
        <zipfileset src="${store.dir}/temp_final.jar"
        excludes="META-INF/*.SF, META-INF/*.DSA, META-INF/*.RSA"/>
    </zip>

    <delete file="${store.dir}/temp_final.jar"/>
    <delete dir="${store.dir}/lib"/>
    <delete file="${store.dir}/README.TXT"/>
</target>
```

Por último, volvemos a la pestaña de *Projects*, pulsamos con el botón derecho del ratón sobre nuestro proyecto y a continuación pulsamos sobre **Clean and Build**. El resultado final del contenido de la carpeta *dist* se muestra en la Figura 6.17.

Figura 6.17. Fichero *build.xml* y carpeta *dist* con los dos JAR.

Ahora, si queremos usar este componente y las librerías para acceso a base de datos solo tendremos que incluir los 2 ficheros JAR en el proyecto, véase Figura 6.18.

Figura 6.18. Proyecto Java usando las dos librerías.

Hasta ahora el componente desarrollado solo producía un objeto DAO, *DepartamentoDAO*. Para añadir más objetos, por ejemplo, para gestionar la tabla de empleados, necesitamos añadir una clase *Empleado*, el DAO con las operaciones a realizar, *EmpleadoDAO*, y otra clase o clases que implementen las operaciones. La clase *DAOFactory* tendrá que producir además del objeto DAO de departamento, el nuevo objeto DAO de empleado:

```
package Dep;

public abstract class DAOFactory {
  // Bases de datos soportadas
  public static final int MYSQL = 1;
  public static final int NEODATIS = 2;
  public static final int ORACLE = 3;
  public static final int SQLITE = 4;

  public abstract DepartamentoDAO getDepartamentoDAO();
  public abstract EmpleadoDAO getEmpleadoDAO();

  public static DAOFactory getDAOFactory(int bd) {
    switch (bd) {
      case MYSQL     :
           return new SqlDbDAOFactory(MYSQL);
      case NEODATIS  :
           return new NeodatisDAOFactory();
      case ORACLE    :
           return new SqlDbDAOFactory(ORACLE);
      case SQLITE    :
           return new SqlDbDAOFactory(SQLITE);
      default        :
          return null;
    }
  }
}
```

Las clases **Factory** para cada base de datos *SqlDbDAOFactory()* y *NeodatisDAOFactory()* deberán sobrescribir el método *getEmpleadoDAO()*:

```
package Dep;

import . . . . . .
public class SqlDbDAOFactory extends DAOFactory {
    . . . . . .
    @Override
    public DepartamentoDAO getDepartamentoDAO() {
        return new SqlDbDepartamentoImpl();
    }

    @Override
    public EmpleadoDAO getEmpleadoDAO() {
         return new SqlDbEmpleadoImpl();
    }
}
```

La clase *SqlDbEmpleadoImpl()* implementará las operaciones sobre la tabla *Empleados* para una base de datos SQL.

```
package Dep;

import . . . .
public class NeodatisDAOFactory extends DAOFactory {
```

```
    . . . . . . .
    @Override
    public DepartamentoDAO getDepartamentoDAO() {
        return new NeodatisDepartamentoImpl();
    }

    @Override
    public EmpleadoDAO getEmpleadoDAO() {
       return new NeodatisEmpleadoImpl();
    }
}
```

La clase *NeodatisEmpleadoImpl()* implementará las operaciones sobre la tabla *Empleados* para una base de datos Neodatis.

ACTIVIDAD 6.4

Partiendo del componente anterior amplíalo para que pueda realizar operaciones sobre la tabla de empleados. El diagrama de clases se muestra en la Figura 6.19.

Figura 6.19. Diagrama de clases Actividad 6.4.

Añade las siguientes clases:

- *Empleado* (los atributos son los mismos que los de la tabla empleados usada en capítulos anteriores),

- *EmpleadoDAO*, los métodos a desarrollar son los siguientes:

```
public interface EmpleadoDAO {
    public boolean InsertarEmp(Empleado emp);
    public boolean EliminarEmp(int empno);
    public boolean ModificarEmp(int empno,  Empleado emp);
    public Empleado ConsultarEmp(int empno);
}
```

- *SqlDbEmpleadoImpl()* y *NeodatisEmpleadoImpl()* implementarán las operaciones anteriores.

Realiza los cambios necesarios en las clases **Factory**.

Controla las siguientes situaciones: no se permite insertar un departamento o empleado que ya exista, no se permite eliminar un departamento que tiene empleados, no se permite eliminar un empleado que tiene empleados a cargo, no se permite insertar o modificar un empleado cuyo departamento o director no exista.

Realiza después un programa Java que pruebe todas las operaciones definidas para el empleado.

Nombra al componente como *DEP_DAO4*. Empaqueta dentro del componente todas las librerías de bases de datos utilizadas.

6.9. PATRÓN MODELO VISTA CONTROLADOR (MVC)

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 6.20 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 6.20. Flujo de solicitud en el MVC.

1. Un usuario realiza una solicitud a través de la aplicación (por ejemplo, pulsa un botón de un formulario o un enlace de una página web). 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 6.21. 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 que viene incluido en el paquete de NetBeans con soporte para aplicaciones Java EE (versión usada NetBeans IDE 8.1)

Figura 6.21. Arquitectura Modelo 2.

Antes de empezar con la aplicación introducimos una serie de conceptos sobre los Servlets y las páginas JSP.

6.9.1. Servlets

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, el objetivo es que el Servlet realice cualquier inicialización antes de ser invocado mediante peticiones HTTP. Este método es opcional, se utiliza para leer datos de configuración de recursos en ficheros de configuración, leer parámetros de inicialización mediante el objeto *ServletConfig*, registrar un driver de base de datos, etc.
- Se llama al método **service()** para procesar las peticiones de los clientes web, es el punto de entrada para ejecutar la lógica de la aplicación. A diferencia del anterior este método es obligatorio.
- Se llama al método **destroy()** para eliminar al Servlet y liberar los recursos. El servidor los destruye porque cesan las llamadas desde el cliente o un temporizador del servidor así lo indica o el propio administrador lo decide. Este es opcional.

6.9.2. Páginas JSP

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 éste 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 %>*. Ejemplo, inicializamos un contador a 0:

```
<%! int contador=0; %>
```

- **Expresiones**: dentro de las etiquetas *<%= código Java %>*. Ejemplo, mezclamos etiquetas HTML con expresiones JSP:

```
<tr><td><%=d.getDeptno()%></td>
   <td><%=d.getDnombre()%></td>
   <td><%=d.getLoc()%></td>
</tr>
```

- **Scriptlets**: dentro de las etiquetas *<% código Java %>*. Ejemplo:

```
<%
ArrayList listadep = (ArrayList)request.getAttribute("departamentos");
if(listadep != null)
  for(int i = 0; i<listadep.size(); i++){
   Departamento d = (Departamento)listadep.get(i);
%>
```

- **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 *<%@ ... %>*. Ejemplo:

```
<%@ page import="Dep.Departamento, 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. En el siguiente ejemplo se incluye en tiempo de ejecución el código de una página HTML llamada *ejercicio.html* en la página JSP:

```
<jsp:include page="ejercicio.html" />
```

- *<jsp:forward> ... </jsp:forward>* o *<jsp:forward ... />* si solo tiene atributos. Se utiliza para redirigir la petición a otra página JSP, a un Servlet o a otro recurso. El siguiente ejemplo redirige la petición a un Servlet llamado *Controlador* enviándole un parámetro de nombre *acción* y valor *insertar*:

```
<jsp:forward page="/Controlador?accion=insertar"/>
```

  - Asociadas a los JavaBeans:
    - *<jsp:useBean ... />* si solo tiene atributos o *<jsp:useBean ...> ... </jsp:useBean>*. Se indica el nombre del Bean que se usará en la página JSP.
    - *<jsp:setProperty ... />*. Se utiliza para establecer las propiedades del Bean.
    - *<jsp:getProperty ... />*. Se utiliza para acceder a las propiedades del Bean.

6.9.3. Ejemplo de aplicación

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:

- En primer lugar aparece el nombre de la aplicación Web, en este ejemplo el nombre es *WebApplication2*, es el directorio donde se ubica toda la aplicación.
- Dentro de este directorio nos encontramos con páginas JSP, HTML y una carpeta muy especial de nombre **WEB-INF**, que está formada por dos carpetas especiales **classes** y **lib** y un fichero especial de nombre **web.xml**:

La 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 se muestra un paquete de nombre *control* donde se almacena el Servlet que llevará a

cabo la lógica de la aplicación (puede haber varios Servlets); y otro paquete de nombre *pantalla* donde se almacenarán los JavaBean que usaremos para enviar los datos de los formularios al controlador. El JavaBean *Departamentos* se usará para las pantallas de entrada de datos de departamentos.

- **Carpeta lib**: contiene todos los JAR que necesita la aplicación, en el ejemplo vamos a utilizar el componente *DEP_DAO4* desarrollado en la Actividad 6.4 y ampliado en el Ejercicio 5 y las librerías empaquetadas en el fichero *ConectoresBD* para acceder a las distintas bases de datos.
- 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 se llama *Controlador.java*). Puede contener otros parámetros para componentes adicionales y manejo de errores.

La aplicación llevará a cabo la gestión de departamentos. Visualizará una página Web inicial llamada ***index.html*** con dos enlaces, uno para realizar el alta de un departamento y el otro para realizar el listado de todos los departamentos, véase Figura 6.22.

Figura 6.22. Aplicación Web desde el navegador.

Desde NetBeans creamos un nuevo proyecto. Pulsamos sobre la opción de menú **File-> NewProyect-> Java Web -> Web Aplication**, véase Figura 6.23 y pulsamos el botón *Next*. En la siguiente pantalla damos nombre al proyecto, nos aseguramos de que la casilla *Use Dedicated Folder for Storing Libraries* esté marcada y pulsamos *Next*. Elegimos el servidor Web, en este caso Tomcat y pulsamos *Next*, Figura 6.23. Desde la siguiente pantalla no seleccionamos ningún Frameworks y pulsamos *Finish*.

Figura 6.23. Crear proyecto Web en NetBeans.

El proyecto generado se muestra en la Figura 6.24. En la pantalla de edición se muestra el fichero *index.html* generado.

Figura 6.24. Proyecto Web generado en NetBeans.

A continuación añadimos las librerías al proyecto. Pulsamos con el botón derecho del ratón sobre **Libraries -> Add JAR/Folders** y añadimos las librerías: *ConectoresBD* y *DEP_DAO4*. Hemos de asegurarnos que esté marcada la opción *Copy to Libraries Folder*; pulsamos el botón *Abrir*. Desde **Source Packages** creamos los dos paquetes que tendrá la aplicación, uno llamado *control* y el otro llamado *pantalla*. En el primero incluiremos el Servlet y en el segundo el Bean que recoge los datos de la pantalla de entrada de los departamentos.

Creamos el Bean en el paquete *pantalla* (**File-> New-> Java Class**) le damos el nombre *Departamentos* y definimos los siguientes atributos, dos constructores (uno de ellos vacío y el otro con los atributos del departamento) y los métodos *get* y *set* de cada atributo:

```
public class Departamentos {
    //atributos del departamento
    private int deptno;
    private String dnombre;
    private String loc;

    //botones de los formularios
    String modificar;
    String eliminar;
    String insertar;

    //constructores y métodos get set
}
```

Creamos el Servlet *Controlador* dentro del paquete *control*. Pulsamos sobre el paquete y a continuación con el botón derecho del ratón pulsamos **New-> Other->Web-> Servlet.** Le damos el nombre *Controlador*, pulsamos el botón *Next*, marcamos la casilla *Add information to deployment descriptor (web.xml)* y pulsamos *Finish*, véase Figura 6.25. Se creará el Servlet y el fichero **web.xml**.

Figura 6.25. Creación del Servlet.

El fichero **web.xml** generado se muestra a continuación. Le añadimos las 3 líneas marcadas en negrita antes de la etiqueta **</web-app>**. Las líneas marcadas en cursiva (dentro de la etiqueta **<web-app>**) se generan automáticamente y hacen referencia al espacio de nombre usado en el documento XML y a los esquemas y versión utilizados:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
        <servlet-name>Controlador</servlet-name>
        <servlet-class>control.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.html</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 (*nombrePaquete.fciheroclass*), en el ejemplo el fichero *Controlador.class* que está en el paquete *control*.
- 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 escribimos lo siguiente para invocar al Servlet enviándole el parámetro *acción* con un valor:

```
<a href = "/WebApplication2/Controlador?accion=alta" >
```

Dentro de la etiqueta **<welcome-file-list>** se indican los ficheros de bienvenida o de entrada en la aplicación, puede haber varios ficheros. En el ejemplo se indica un único fichero, *index.html*, que es el punto de entrada en la aplicación.

Todos los ficheros HTML y JSP los creamos dentro de la carpeta **Web Pages** del proyecto. Para crear el fichero *index.html* pulsamos con el botón derecho del ratón sobre **Web Pages** y a continuación pulsamos sobre **New-> Other->Web-> HTML**. Para crear los ficheros JSP hacemos lo mismo pero seleccionamos **JSP** en vez de HTML.

Creamos el fichero *index.html* y escribimos el siguiente código:

```
<!DOCTYPE html>
<html>
    <head>
        <title>GESTI&Oacute;N DE DEPARTAMENTOS</title>
    </head>
    <body>
        <h1 align="center">GESTI&Oacute;N DE DEPARTAMENTOS</h1>
        <p align='center'>
            <a href = "/WebApplication2/Controlador?accion=alta">
             Alta de Departamento</a>
        </p>
        <p align='center'>
            <a href = "/WebApplication2/Controlador?accion=listado">
             Listado de Departamentos</a>
        </p>
    </body>
</html>
```

Desde los enlaces **href** se llama al Servlet *Controlador* con la acción a realizar "alta" o "listado", enviada por medio del parámetro ***accion***. Para poder llevar a cabo esta acción los contenedores web tienen el fichero **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.

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***.

El código del Servlet es el siguiente, todo el proceso se desarrolla en el método ***service()***:

```
package control;

import Dep.DAOFactory;
import Dep.Departamento;
import Dep.DepartamentoDAO;

import java.io.IOException;
import java.util.ArrayList;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Controlador extends HttpServlet {

  //Se usa el component DEP_DAO4
  DAOFactory bd = DAOFactory.getDAOFactory(DAOFactory.MYSQL);

  public void service(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

     //Se obtiene el DAO
     DepartamentoDAO depDAO = bd.getDepartamentoDAO();

     //Se obtiene la acción a realizar, parámetro accion
     String op = request.getParameter("accion");

     //Si es alta se muestra la pantalla de alta de departamento
     if (op.equals("alta")) {
         response.sendRedirect("alta.jsp");
     }

     //Si es listado, se obtienen los datos de los departamentos
     //después se envían a listado.jsp
     if (op.equals("listado")) {
         ArrayList lista = depDAO.ObtenerDepartamentos();

         request.setAttribute("departamentos", lista);
         RequestDispatcher rd =
                    request.getRequestDispatcher("/listado.jsp");
         rd.forward(request, response);
     }

     //Se obtienen los datos de la página JSP
     //y luego se inserta el departamento en la BD
     if (op.equals("insertar")) {
         pantalla.Departamentos dep =  (pantalla.Departamentos)
              request.getAttribute("depart");
```

Usamos el componente DEP_DAO4 para acceder a los datos

```
            Departamento departamento = new Departamento
                    (dep.getDeptno(), dep.getDnombre(), dep.getLoc());

            boolean insertar = depDAO.InsertarDep(departamento);
            String mensaje = "";

            if (insertar) {
                mensaje = "Departamento " + dep.getDeptno() +
                          " insertado";
            } else {
                mensaje = "Error al insertar Departamento " +
                          dep.getDeptno();
            }
            System.out.println(mensaje);

            response.sendRedirect("index.html");
        }
    }
}//
```

Antes de realizar el listado de departamentos se cargan los datos de la tabla Departamentos en un ArrayList, para ello se utiliza el método *depDAO.ObtenerDepartamentos()* desarrollado en el componente *DEP_DAO4* (hemos de realizar antes el Ejercicio 5 propuesto).

Los datos para insertar un departamento los recibe de la página ***alta.jsp*** mediante el atributo *depart*. Se obtiene un objeto *pantalla.Departamentos* con los datos del formulario de alta. La inserción se realiza invocando al método *depDAO.InsertarDep(departamento)* desarrollado en el componente *DEP_DAO4*. El método devuelve *true* o *false* indicando si la operación se ha realizado correctamente.

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. En el ejemplo se ha usado la expresión *request.getParameter("accion")* para leer el parámetro *acción* enviado por la página ***index.html***.
- ***request.setAttribute("nombre", objeto)***: se almacena un objeto en la sesión con el *nombre* indicado. En el ejemplo se ha usado *request.setAttribute("departamentos", lista)* en el controlador para enviar la lista de departamentos a la página ***listado.jsp***.
- ***request.getAttribute("nombre")***: devuelve el atributo asociado al nombre en esta sesión o *null* si no hay ningún objeto asociado bajo el nombre. Este método se usa en la página ***listado.jsp*** para recoger los departamentos que envía el controlador: *ArrayList listadep = (ArrayList) request.getAttribute("departamentos")*.

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. En el ejemplo se ha usado para abrir las

páginas ***index.html*** y ***alta.jsp*** que no necesitan una recarga previa de datos (*response.sendRedirect("alta.jsp")*).

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 /. En el ejemplo se ha usado para referenciar la página ***listado.jsp***. Después de este método se usa el método *forward(request, response)* que permite reenviar la solicitud a otro servlet, página JSP o fichero HTML, este método permite seguir gestionando la misma petición, es como si se recargase la página manteniendo la URL original.

Cuando queramos abrir una página que implique recarga de datos necesitaremos usar estos dos métodos. Por tanto, para ir a la página *listado.jsp* con los datos de los departamentos necesitamos hacer uso de los dos métodos:

```
RequestDispatcher rd = request.getRequestDispatcher("/listado.jsp");
rd.forward(request, response);
```

La diferencia entre los métodos *sendRedirect()* y *forward()* es que con el primero se pierden la petición y la respuesta (*ServletRequest* y *ServletResponse*), con lo cual no podemos enviarle al recurso destino información grabada sobre *ServletRequest* y *ServletResponse*.

La página ***alta.jsp*** muestra en el navegador un formulario para realizar la entrada de datos de un departamento, Figura 6.26.

Figura 6.26. Formulario de entrada de departamentos.

Los datos del formulario se enviarán a través del JavaBean *Departamentos.java* (que se encuentra en el paquete *pantalla*) 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()* de la siguiente manera: *request.getAttribute("depart")*. El atributo *name* de las etiquetas *input* del formulario deben tener el mismo nombre que los atributos del Bean *Departamentos*. El código de ***alta.jsp*** es el siguiente:

```
<!DOCTYPE html>
<html><head><title>ALTA DE DEPARTAMENTOS</title></head>
<!--Formulario de entrada de datos e Inserción en el JavaBean -->
<jsp:useBean id = "depart" scope = "request"
             class = "pantalla.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>
  <form method="post">
        <p>N&uacute;mero de departamento:
        <input name="deptno" required
              type="number" min="1" max="99" /> </p>
        <p>Nombre:
        <input name="dnombre"
              required type="text" size="15" maxlength="15" /> </p>
        <p>Localidad:
        <input name="loc" required type="text"
              size="15" maxlength="15" /> </p>
        <input type="submit" name="insertar"
                              value="Insertar departamento." />
        <input type="reset" name="cancelar"
                             value="Cancelar entrada." />
  </form>
  <a href="index.html">Inicio</a>
  </center>
</body>
</html>
```

Para acceder al JavaBean 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 las siguientes:

- **<jsp:useBean** *id = "depart" scope = "request" class = "pantalla.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* indica 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, loc, insertar*).
- **<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 envía al controlador con el parámetro *accion* con valor *insertar* para que inserte los datos del Bean en la base de datos.

La página ***listado.jsp*** muestra en una tabla HTML el listado de los departamentos, Figura 6.27. Es llamada por el controlador y recibe el atributo *departamentos* conteniendo un ArrayList con los departamentos a listar, cada elemento del array es un Bean de tipo *Dep.Departamento* del componente *DEP_DAO4*, por tanto, al principio de la página hay que importarlo, además de otras librerías necesarias. El código es el siguiente:

```
<%@ page import = "Dep.Departamento, java.util.*"%>
<html><head><title>LISTADO DE DEPARTAMENTOS</title></head>
<body>
<center>
```

Para distribuir la aplicación Web y desplegarla en el servidor Tomcat necesitamos construir el fichero **WAR** (*Web Application Archive*), pulsamos con el botón derecho del ratón sobre el proyecto Web y seleccionamos **Clean and Build**. Se genera un fichero **war** con el mismo nombre que la aplicación Web en la carpeta del proyecto, desde **Files** se pueden ver el directorio ***dist*** con el **war** generado, Figura 6.29.

Para ejecutar y probar la aplicación, pulsamos sobre el nombre del proyecto con el botón derecho del ratón y seleccionamos **Run**. Se abre el navegador Web con la página inicial ***index.html***. Desde la consola de NetBeans se puede ver la consola del servidor Tomcat, Figura 6.30. Se puede detener el servidor pulsando en el botón *Stop the Server*.

Figura 6.30. Consola del servidor Tomcat desde NetBeans.

Para desplegar la aplicación en el servidor Tomcat, fuera del entorno de NetBeans, necesitamos generar el **war** (que ya hemos visto cómo se hace y dónde se almacena). A continuación necesitamos saber la carpeta donde está instalado Tomcat. Por ejemplo, lo tenemos instalado en la carpeta: *D:\Servers\Apache Software Foundation\Apache Tomcat 8.0.27*. Editamos el fichero **server.xml** ubicado en la carpeta **conf** (*D:\Servers\Apache Software Foundation\Apache Tomcat 8.0.27\conf\server.xml*). Modificamos el puerto, asignamos un puerto que no esté ocupado por ninguna aplicación, por ejemplo 8084 (aproximadamente línea 69):

```
<Connector port="8084" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Editamos el fichero **users-xml** ubicado en la carpeta **conf** (*D:\Servers\Apache Software Foundation\Apache Tomcat 8.0.27\conf\users-xml*) para crear un usuario que maneje el entorno Web de Tomcat. Añadimos las siguientes líneas entre las etiquetas *<tomcat-users> </tomcat-users>* para crear el usuario *admin* y clave *admin*:

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
```

Desde la línea de comandos del DOS nos dirigimos a la carpeta **bin** (*D:\Servers\Apache Software Foundation\Apache Tomcat 8.0.27\bin*) y ejecutamos el fichero **startup.bat** (en Linux *startup.sh*), hemos de asegurarnos de que el servidor no está siendo utilizado por NetBeans. Se muestran las variables de entorno de Tomcat, similar a las variables que se mostraron desde la consola en NetBeans.

A continuación se abre una nueva ventana mostrando la consola del servidor. Abrimos el navegador Web y escribimos la URL **http://localhost:8084/.** Se muestra la pantalla inicial de Tomcat. Pulsamos sobre *Manage App*, Figura 6.31. Nos pide nombre de usuario y clave, escribimos el usuario creado anteriormente *admin* clave *admin*.

Figura 6.31. Pantalla de administración de Tomcat.

Aparece la pantalla de gestión de aplicaciones. Desde aquí podemos recargar nuestra aplicación. Movemos la pantalla hacia abajo hasta encontrar el marco para desplegar archivo, seleccionamos el fichero **war** y pulsamos en el botón *Desplegar*, véase Figura 6.32.

Figura 6.32. Desplegar un fichero WAR.

Se muestra nuestra aplicación desplegada, Figura 6.33. Al pulsar sobre ella se muestra página inicial de la aplicación ***index.html***.

Figura 6.33. Aplicación desplegada en Tomcat.

Para parar el servidor Tomcat ejecutamos desde la línea de comandos del DOS el comando **shutdown.bat** (en Linux *shutdown.sh*).

ACTIVIDAD 6.5

Cambia lo necesario en la clase *Controlador* para probar la aplicación web con las distintas bases de datos incluidas en el componente. Para probarla en Neodatis es necesario que la base de datos esté en la carpeta **bin** de Tomcat (*D:\Servers\Apache Software Foundation\Apache Tomcat 8.0.27\bin*).

En el *Controlador* después de insertar el departamento se muestra un mensaje en la consola del servidor indicando si se ha insertado o no el departamento, y se abría de nuevo la página **index.html**:

```
boolean insertar = depDAO.InsertarDep(departamento);
String mensaje = "";
if (insertar)
      mensaje = "Departamento " + dep.getDeptno() + " insertado";
else
      mensaje = "Error al insertar Departamento " + dep.getDeptno();
System.out.println(mensaje);
response.sendRedirect("index.html");
```

Lo lógico es que el mensaje nos aparezca en el navegador Web. Para ello cambiamos el código y hacemos que el mensaje aparezca en una página JSP de nombre *DepartamentoInsertado.jsp*, a esta página le enviamos el mensaje a visualizar mediante el método *request.setAttribute("mensaje",mensaje)*:

```
boolean insertar = depDAO.InsertarDep(departamento);
String mensaje="";
if(insertar)
      mensaje="Departamento "+dep.getDeptno()+" insertado";
else
      mensaje="Error al insertar Departamento " + dep.getDeptno();

request.setAttribute("mensaje",mensaje); //se envia mensaje al jsp
RequestDispatcher rd=
            request.getRequestDispatcher("/DepartamentoInsertado.jsp");
rd.forward(request,response);
```

El código de la página ***DepartamentoInsertado.jsp*** es el siguiente, recoge el mensaje enviado por el controlador y lo muestra en el navegador, muestra también dos enlaces, uno para volver a la pantalla de altas y el otro para volver al índice, Figura 6.34:

```
<html>
  <head><title>INSERCI&Oacute;N DE DEPARTAMENTOS</title></head>
  <body>
    <br>
    <h1 align="center">INSERCI&Oacute;N DE DEPARTAMENTO</h1>
    <%
       String mensaje = (String) request.getAttribute("mensaje");
    %>
     <p align="center"><font color="red"><%=mensaje%></font></p>
     <p align="center">
        <a href="/WebApplication1/controlador?accion=alta">
                    Alta de Departamento</a></p>
    <p align="center">
        <a href="index.html">Inicio</a> </p>
  </body>
</html>
```

Figura 6.34. Página *DepartamentoInsertado.jsp*.

ACTIVIDAD 6.6

Amplía la aplicación Web de departamentos, se deben poder modificar y eliminar los departamentos. La página *index.html* debe mostrar un enlace para la modificación de departamentos. Al pulsar en el enlace se debe mostrar un formulario (página *pantModifBorr.jsp*) que muestre una lista con los departamentos y los botones para modificar y eliminar el departamento, Figura 6.35. La visualización de la página se debe controlar en la clase controlador, esta página al cargarse debe mostrar una lista con los departamentos (el proceso es similar a la carga de la página *listado.jsp*). El enlace *Inicio* vuelve a la página *index.html*.

Figura 6.35. Enlace a *Modificación de departamentos* y página *pantModifBorr.jsp*.

Desde esta página se elige el departamento, y se pulsa en el botón modificar o eliminar. Se usará el Bean *pantalla.Departamentos* para enviar la información al controlador y saber qué botón se ha pulsado (recuerda que en el Bean están definidos los campos del departamento y 3 campos más que hacen referencia a los botones de los formularios *insertar*, *modificar* y *eliminar*). Esta página es similar a la página *alta.jsp*, con la diferencia de que tenemos 2 botones y desde el controlador hemos de averiguar el botón que se ha pulsado para realizar la acción oportuna. Por ejemplo, en el controlador recogemos los datos del Bean y preguntamos por los botones:

```
//obtener datos del formulario
pantalla.Departamentos depPantalla = (pantalla.Departamentos)
                        request.getAttribute("depart");//
```

```
if (depPantalla.getEliminar() == null) {
   //botón  modificar pulsado
   //acciones para modificación
} else {
   //botón eliminar pulsado
   //acciones para borrado
}
```

Si se pulsa el botón modificar se debe mostrar un nuevo formulario, página *modificacionDep.jsp*, que al cargarse mostrará los datos del departamento seleccionado en la lista, Figura 6.36.

Figura 6.36. Página *modificacionDep.jsp*.

El campo de número de departamento no se puede modificar (en la etiqueta *input* añadimos el atributo *readonly="readonly"*). Al pulsar el botón de modificar se enviarán los datos al controlador por medio del Bean *pantalla.Departamentos*. Dentro del controlador se hará el proceso oportuno y se mostrará una página de mensaje *DepartamentoModificado.jsp* indicando si se ha podido modificar o no (similar a la página *DepartamentoInsertado.jsp*), Figura 6.37.

Figura 6.37. Páginas *DepartamentoModificado.jsp* y *DepartamentoEliminado.jsp*.

El enlace de *Modificación de Departamentos* vuelve a la página *pantModifBorr.jsp*, antes debe pasar por el controlador para realizar la carga de los departamentos y que se muestren en la lista. El enlace *Inicio* vuelve a la página *index.html*.

Si se pulsa el botón *Eliminar el Departamento*, desde el controlador se hará la acción oportuna y se mostrará una página de mensaje *DepartamentoEliminado.jsp* indicando si se ha

podido eliminar o no (similar a la página *DepartamentoInsertado.jsp*), Figura 6.37. El proyecto en NetBeans con las nuevas páginas desarrolladas tendrá el aspecto mostrado en la Figura 6.38.

Figura 6.38. Proyecto de la Actividad 6.6.

COMPRUEBA TU APRENDIZAJE

1. ¿Que es un componente? Cita algunas ventajas e inconvenientes del uso de componentes.
2. ¿Qué características debe tener un componente?
3. ¿Cuál es la diferencia entre el modelo de componentes y la plataforma de componentes?
4. ¿Qué características debe cumplir un JavaBean?.
5. Partiendo del componente *DEP_DAO4* creado en la *Actividad 6.4* añade a departamentos un nuevo método que obtenga un ArrayList con todos los departamentos. Añade a empleados dos nuevos métodos, uno que obtenga un ArrayList con todos los empleados y, un segundo que obtenga un ArrayList de los empleados de un departamento.
6. Crea un programa Java que use el componente anterior para mostrar todos los departamentos. Pruébalo en distintas bases de datos.
7. Crea un programa Java que use el componente anterior para mostrar todos los empleados de un departamento. El departamento se pedirá por teclado. Se deben mostrar los datos del departamento. Además, por cada empleado se debe mostrar también el apellido de su director. Realiza el ejercicio con varias bases de datos.
8. Amplía la aplicación Web desarrollada en el Epígrafe 6.9.3 y ampliada en la Actividad 6.6 para que dé soporte a la gestión de empleados. En la página *index.html* se añadirán 3

nuevos enlaces para alta, listado y modificación de empleados. Crea un nuevo Servlet controlador para gestionar los empleados y un nuevo Bean para los formularios con datos de empleados. La página *index.html* mostrará el aspecto de la Figura 6.39.

Figura 6.39. Página *index.html* del Ejercicio 8.

La página de alta de empleados se muestra en la Figura 6.40. El director y el departamento se eligen de una lista, estas listas se envían desde el controlador por medio de un ArrayList, la primera lista contiene todos los empleados y la segunda los departamentos. Se considera como fecha de alta la fecha del sistema. Utiliza tipos de datos HTML5 para la entrada de datos en los formularios.

Figura 6.40. Página de alta de empleados.

La página de listado de empleados se muestra en la Figura 6.41. La página de modificación y borrado de empleados se muestra en la Figura 6.42. En la Figura 6.43 se muestran los datos del empleado que se desea modificar, el número de empleado no se puede modificar.

Figura 6.41. Página de listado de empleados.

Figura 6.42. Página de modificación y borrado de empleados.

Figura 6.43. Página de modificación de un empleado.

Después de modificar, insertar o eliminar un empleado se debe mostrar una pantalla de mensaje indicando si se ha llevado a cabo o no el proceso. El proyecto en NetBeans tiene el aspecto mostrado en la Figura 6.44.

Figura 6.44. Proyecto Web del Ejercicio 8.

ACTIVIDADES DE AMPLIACIÓN

Disponemos de una colección llamada *DepartamentosEmpleados* en una base de datos **eXist** que contiene dos documentos: *departamentos.xml* y *empleados.xml*, su estructura es la siguiente:

departamentos.xml	empleados.xml

<?xml version="1.0" encoding="ISO-8859-1"?> <departamentos>
  <TITULO> ...</TITULO>
  <DEP_ROW>
  <DEPT_NO>..</DEPT_NO>
  <DNOMBRE>..</DNOMBRE>
  <LOC>..</LOC>
  </DEP_ROW>
  <DEP_ROW>
  . . . .
  </DEP_ROW>
  . . . . . .
  . . . . . .
</departamentos>
	<?xml version="1.0" encoding="ISO-8859-1"?>
	<empleados>
	  <TITULO>. . .</TITULO>
	  <EMP_ROW>
	  <EMP_NO>..</EMP_NO>
	  <APELLIDO>..</APELLIDO>
	  <OFICIO>..</OFICIO>
	  <DIR>..</DIR>
	  <FECHA_ALT>..</FECHA_ALT>
	  <SALARIO>..</SALARIO>
	  <COMISION>..</COMISION>
	  <DEPT_NO>..</DEPT_NO>
	  </EMP_ROW>
	  <EMP_ROW>
	  . . . . . .
	  </EMP_ROW>
	  . . . . . .
	  . . . . . .
	</empleados>

La URI para acceder a la base de datos y a la colección específica es la siguiente: *xmldb:exist://localhost:8089/exist/xmlrpc/db/DepartamentosEmpleados*, el nombre de usuario para autenticarse en la base de datos y la contraseña es *admin*.

Se pide crear un nuevo proyecto de librería llamado *DEP_DAO5* e incluir todas las clases desarrolladas en *DEP_DAO4* (o bien ampliar la librería *DEP_DAO4*). Añadir nuevas clases y librerías para dar soporte a la gestión de empleados y departamentos de la base de datos **eXist**.

Se deberán crear las siguientes clases: *EXISTDbDAOFactory.java* que extienda *DAOFactory* para crear la conexión a la base de datos **eXist.** *EXISTDepartamentoImpl.java* que implemente *DepartamentoDAO* desarrollando todos los métodos para gestionar departamentos y *EXISTEmpleadoImpl.java* que implemente *EmpleadoDAO* desarrollando todos los métodos para gestionar empleados. También hay que cambiar la clase *DAOFactory.java* para que soporte la nueva base de datos.

En las operaciones sobre empleados y departamentos se debe tener en cuenta:

- No se permite insertar un departamento o empleado que ya exista.
- No se permite eliminar un departamento que tiene empleados.
- No se permite eliminar un empleado que tiene empleados a cargo.
- No se permite insertar o modificar un empleado cuyo departamento o director no exista.

Una vez desarrollado el componente realiza un programa Java para probar las distintas operaciones sobre empleados y otro para las operaciones sobre departamentos.

Modifica lo necesario en la aplicación Web del Ejercicio 8 para que funcione sobre la base de datos **eXist**.