CF GRADO SUPERIOR

CICLOS FORMATIVOS

R.D. 1538/2006

Acceso a Datos

JOSE EDUARDO CÓRCOLES TENDERO
FRANCISCO MONTERO SIMARRO

Ra-Ma®
www.ra-ma.es/cf

ACCESO A DATOS©
José Eduardo Córcoles Tendero, Francisco Montero Simarro

ISBN de Edición en Papel: **978-84-9964-239-0**

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-390-8

E-Book desarrollado en España en Septiembre de 2014

Acceso a Datos

JOSÉ EDUARDO CÓRCOLES TENDERO
FRANCISCO MONTERO SIMARRO

Ra-Ma®

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

A mi madre: te quiero.

Dormía y soñaba que mi vida era alegre, desperté y advertí que mi vida era justo así gracias a Nieves, Hugo y Marco.

A Yolanda, por su apoyo y cariño incondicional.

Índice

Introducción

Este libro surge con el propósito de acercar al lector a los aspectos más importantes que encierra el acceso a los datos por parte de las aplicaciones informáticas (software). Este trabajo puede servir de apoyo a estudiantes del Ciclo Formativo de Grado Superior de **Desarrollo de Aplicaciones Multiplataforma** y para estudiantes universitarios del Grado de **Informática**.

Lo primero antes de empezar con los capítulos es delimitar adecuadamente qué se entiende por acceso a datos. Según el IEEE *(Institute of Electrical and Electronics Engineers*) software es:

> "El conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación".

Una definición más conocida asumida por el área de la ingeniería del software es:

> "El software es programas + **datos**".

En ambas definiciones se ha resaltado la palabra *datos*. Los *datos* son, sencillamente, lo que necesitan los programas para realizar la misión para la que fueron programados. Desde esta perspectiva, los datos pueden entenderse como persistentes o no persistentes.

- Los datos persistentes son aquellos que el programa necesita que sean guardados en un sitio "seguro" para que en posteriores ejecuciones del programa se pueda recuperar su estado anterior. Por ejemplo, en un teléfono móvil, la agenda con los números de teléfono de los contactos representa un conjunto de datos persistentes. De poca utilidad sería la agenda si cada vez que el móvil se apagara y se volviera a encender hubiese que introducir los datos de los contactos.
- Los datos no persistentes son aquellos que no es necesario que el programa guarde entre ejecución y ejecución ya que solo son necesarios mientras la aplicación se está ejecutando. Por ejemplo, los teléfonos móviles llevan un registro de las aplicaciones que se están ejecutando en un momento dado: agenda, navegador, apps de todo tipo, etc. Sin embargo, cuando el móvil se apaga esos datos no son necesarios, ya que cuando el móvil se vuelva a encender todas esas aplicaciones ya no estarán cargadas. Por tanto, los datos sobre qué aplicaciones se están ejecutando cuando un móvil está encendido son datos no persistentes.

En resumen, cuando se necesitan en una aplicación datos que vivan más allá de una sesión de ejecución del programa entonces es necesario hacer esos datos persistentes. Si los datos no interesan más allá de una sesión de ejecución, entonces esos datos no es necesario que sean persistentes.

Desde un punto de vista lógico, con independencia del sistema operativo, la persistencia se consigue con bases de datos y sistemas de ficheros (los llamados sistemas de almacenamiento). Las bases de datos relacionales (Oracle, MySQL, etc.) son ejemplos de sistemas que permiten la persistencia de datos, aunque no son los únicos. Existen muchas otras tecnologías más allá de las bases de datos relacionales que permiten la persistencia y que se basan en otros modelos diferentes.

El objetivo de este trabajo es ofrecer una visión de diferentes sistemas de almacenamiento destinados a la persistencia de datos y mostrar de manera práctica (con Java) cómo las aplicaciones informáticas pueden acceder a esos datos, recuperarlos e integrarlos. Ficheros XML, bases de datos

orientadas a objetos, bases de datos objeto-relacionales, bases de datos XML nativas, acceso a datos con conectores JDBC, frameworks de mapeo objeto-relacional
(ORM), etc., son algunas de las tecnologías que se trabajan en este libro. Todas ellas son referencias en el desarrollo de aplicaciones multiplataforma profesionales.

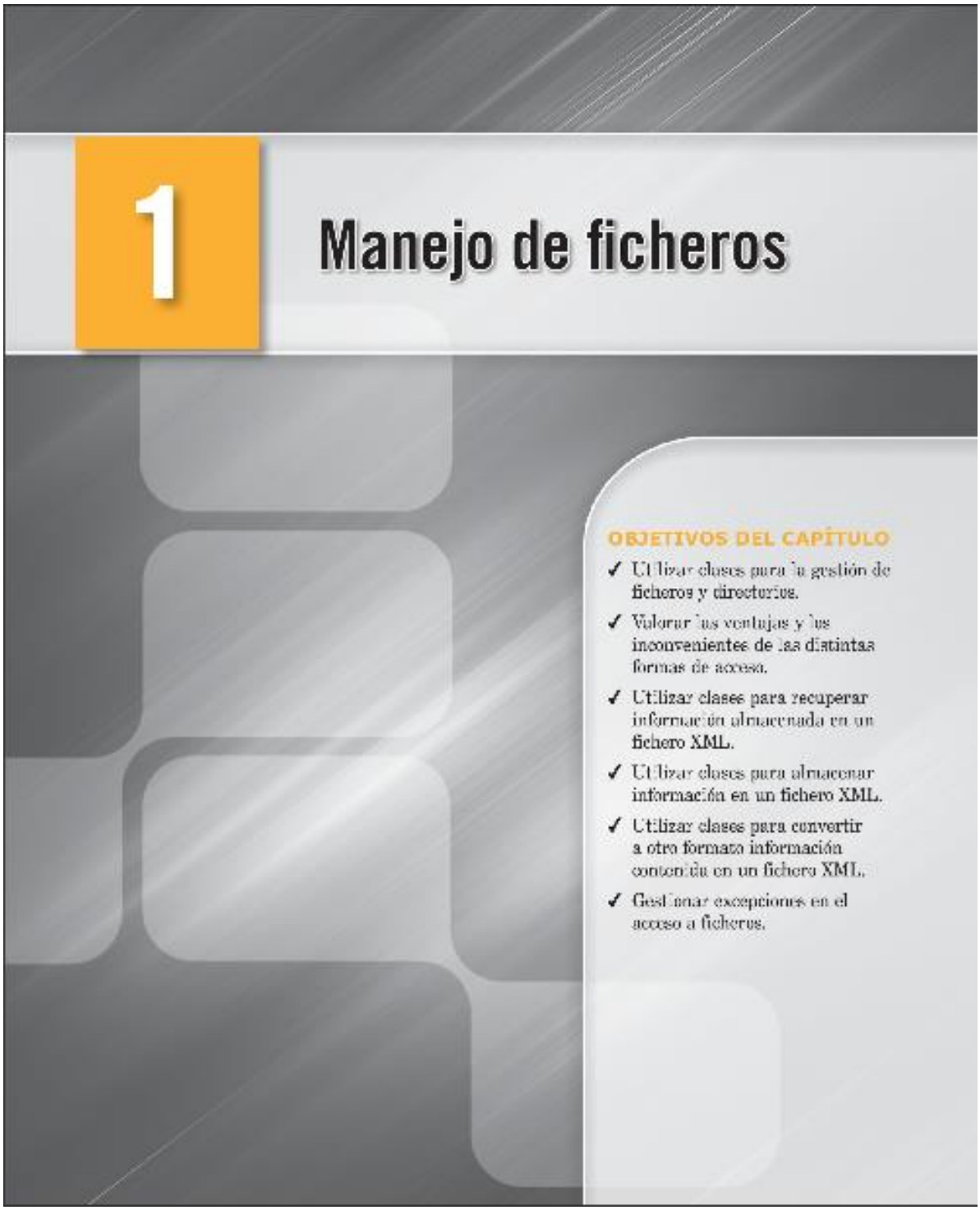
Profesores y alumnos del Ciclo Formativo de Grado Superior de Desarrollo de Aplicaciones Multiplataforma deben tener claros los conocimientos mínimos necesarios para seguir este trabajo con solvencia:

1. Se debe saber generar y manejar ficheros XML y esquemas XML (lenguajes de marcas y sistemas de gestión de información).
2. Se deben manejar sistemas gestores de bases de datos relacionales y SQL como lenguaje de consulta y modificación (bases de datos).
3. Se debe manejar Java, programación básica y entornos de desarrollo (programación).

Las actividades resueltas integradas en los contenidos, los ejercicios propuestos y los test de evaluación sirven para desarrollar y afianzar conocimientos en cualquier secuencia didáctica llevada a cabo en el aula. También las sugerencias y enlaces web para ampliar conocimientos incluidos en cada capítulo pueden servir de gran ayuda para desarrollar actividades de ampliación de conocimientos o concretar más específicamente un tema de interés. Por su parte, los estudiantes pueden utilizar la clara y concreta exposición de los contenidos, las actividades resueltas y los enlaces para ampliar conocimientos como guía para afianzar su aprendizaje.

Ra-Ma pone a disposición de los profesores una guía didáctica para el desarrollo del tema que incluye las soluciones a los ejercicios expuestos en el texto. Puede solicitarla a editorial@ra-ma.com, acreditándose como docente y siempre que el libro sea utilizado como texto base para impartir las clases.

Así mismo, pone a disposición de los alumnos una página web para el desarrollo del tema que incluye las presentaciones de los capítulos, un glosario, bibliografía y diversos recursos para suplementar el aprendizaje de los conocimientos de este módulo.



1 Manejo de ficheros

Objetivos del Capítulo

- ✓ Utilizar clases para la gestión de ficheros y directorios.
- ✓ Valorar las ventajas y los inconvenientes de las distintas formas de acceso.
- ✓ Utilizar clases para recuperar información almacenada en un fichero XML.
- ✓ Utilizar clases para almacenar información en un fichero XML.

- ✓ Utilizar clases para convertir a otro formato información contenida en un fichero XML.
- ✓ Gestionar excepciones en el acceso a ficheros.

Cuando se quiere conseguir persistencia de datos en el desarrollo de aplicaciones los ficheros entran dentro de las soluciones catalogadas como más sencillas. En este capítulo se muestran algunas alternativas diferentes para trabajar con ficheros. No son todas, solo son algunas. Sin embargo, estas soluciones sí pueden considerarse una referencia dentro de las soluciones actuales respecto al almacenamiento de datos en ficheros. Aunque las diferentes alternativas de acceso a ficheros mostradas en el capítulo se han trabajado desde la perspectiva de Java, la mayoría de entornos de programación dan soporte a estas mismas alternativas, aunque con otra sintaxis (por ejemplo, Microsoft .NET).

Las primeras secciones del capítulo se centran en el acceso a ficheros desde Java (flujos). Para comprender adecuadamente estos contenidos, es recomendable que el lector esté familiarizado con conceptos básicos de programación en Java.

Las últimas secciones se centran en el manejo de XML como tipo especial de fichero. Alternativas como DOM, SAX y JAXB serán trabajadas en el capítulo junto con XPath como lenguaje de consulta de datos XML. Para comprender adecuadamente el contenido de estas secciones, es recomendable que el lector esté familiarizado con el lenguaje XML: su formato y la definición de esquemas XML (XSD) para validar su estructura.

1.1 FORMAS DE ACCESO A UN FICHERO.CLASES ASOCIADAS

En Java, en el paquete java.io, existen varias clases que facilitan trabajar con ficheros desde diferentes perspectivas: ficheros de acceso secuencial o acceso aleatorio, ficheros de caracteres o ficheros de bytes (binarios). Los dos primeros son una clasificación según el tipo de contenido que guardan. Los dos últimos son clasificados según el modo de acceso.

Criterios según el tipo de contenido:

- Ficheros de caracteres (o de texto): son aquellos creados exclusivamente con caracteres, por lo que pueden ser creados y visualizados utilizando cualquier editor de texto que ofrezca el sistema operativo (por ejemplo: Notepad, Vi, Edit, etc.).
- Ficheros binarios (o de bytes): son aquellos que no contienen caracteres reconocibles sino que los bytes que contienen representan otra información: imágenes, música, vídeo, etc. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que entiendan cómo están dispuestos los bytes dentro del fichero, y así poder reconocer la información que contiene.

Criterios según el modo de acceso:

- Ficheros secuenciales: en este tipo de ficheros la información es almacenada como una secuencia de bytes (o caracteres), de manera que para acceder al byte (o carácter) i-ésimo, es necesario pasar antes por todos los anteriores (i-1).
- Ficheros aleatorios: a diferencia de los anteriores el acceso puede ser directamente a una posición concreta del fichero, sin necesidad de recorrer los datos anteriores. Un ejemplo de acceso aleatorio en programación es el uso de arrays.

En las siguientes secciones se muestran alternativas para el acceso a ficheros según sean de un tipo u otro. Sin embargo, con independencia del modo, Java define una clase dentro del paquete java.io que representa un archivo o un directorio dentro de un sistema de ficheros. Esta clase es **File**

Un objeto de la clase *File*[1] representa el nombre de un fichero o de un directorio que existe en el sistema de ficheros. Los métodos de File permiten obtener toda la información sobre las características del fichero o directorio. Un ejemplo de código para la creación de un objeto File con un fichero llamado libros.xml es el siguiente:

```
File f = new File ("proyecto\\libros.xml");
```

Si sobre ese nuevo objeto File creado se aplica el siguiente código:

```
System.out.println ("Nombre : + f.getName());
System.out.println ("Directorio padre : + f.getParent());
System.out.println ("Ruta relativa : + f.getPath());
System.out.println ("Ruta absoluta : + f.getAbsolutePath());
```

El resultado será:

```
Nombre: libros.xml
Directorio padre: proyecto
Ruta relativa: proyecto\libros.xml
Ruta absoluta: c:\accesodatos\proyecto\libros.xml
```

A lo largo de este capítulo, y del resto de capítulos, se utilizará mucho la clase File para representar ficheros o directorios del sistema operativo.

1.2 GESTIÓN DE FLUJOS DE DATOS

En Java, el acceso a ficheros es tratado como un flujo (stream)2 de información entre el programa y el fichero. Para comunicar un programa con un origen o destino de cierta información (fichero) se usan flujos de información. Un flujo no es más que un objeto que hace de intermediario entre el programa y el origen o el destino de la información. Esta abstracción proporcionada por los flujos hace que los programadores, cuando quieren acceder a información, solo se tengan que preocupar por trabajar con los objetos que proporcionan el flujo, sin importar el origen o el destino concreto de donde vengan o vayan los datos.

Por ejemplo, Java ofrece la clase *FileReader,* donde se implementa un flujo de caracteres que lee de un fichero de texto. Desde código Java, un programador puede manejar un objeto de esta clase para obtener el flujo de datos texto

1 Para tener más información sobre los métodos de la clase File se puede consultar:
http://docs.oracle.com/javase/7/docs/api/java/io/ File.html

2 Aunque este capítulo se centre en ficheros, los flujos (stream) en Java se utilizan para acceder también a memoria o para leer/escribir datos en dispositivos de entrada/salida.

sacados del archivo que elija. Otro ejemplo es la clase FileWriter. Esta clase implementa un flujo de caracteres que se escriben en un fichero de texto. Un programador puede crear un objeto de esta clase para escribir datos texto en un archivo que elija.

Si lo que se desea es acceder a ficheros que almacenen información binaria (bytes) en vez de texto, entonces Java proporciona otras clases para implementar flujos de lectura o escritura con ficheros binarios: FileInputStream y FileOutputStream.

Las clases anteriores son para acceso secuencial a ficheros. Sin embargo, si lo que se desea es un acceso aleatorio, Java ofrece la clase RandomAccessFile, que permite acceder directamente a cualquier posición dentro del fichero vinculado con un objeto de este tipo.

Como se ha comentado antes, la utilización de flujos facilita la labor del programador en el acceso a ficheros ya que, con independencia del tipo de flujo que maneje, todos los accesos se hacen más o menos de la misma manera:

1. Para leer: se abre un flujo desde un fichero. Mientras haya información se lee la información. Una vez terminado, se cierra el flujo.
2. Para escribir: se abre el flujo desde un fichero. Mientras haya información se escribe en el flujo. Una vez terminado, se cierra el flujo.

Los problemas de gestión del archivo, por ejemplo, cómo se escribe en binario o cómo se hace cuando el acceso es secuencial o es aleatorio, quedan bajo la responsabilidad de la clase que implementa el flujo. El programador se abstrae de esos inconvenientes y se dedica exclusivamente a trabajar con los datos que lee y recupera. A continuación se concretan los pasos básicos para abrir ficheros según el modo de acceso o el tipo de fichero.

1.2.1 CLASE FILEWRITER

El flujo FileWriter3 permite escribir caracteres en un fichero de modo secuencial. Esta clase hereda los métodos necesarios para ello de la clase Writer. Los constructores principales son:

FileWriter (String ruta, boolean añadir)
FileWriter(File fichero)

El parámetro ruta indica la localización del archivo en el sistema operativo. El parámetro añadir igual a trueindica que el fichero se usa para añadir datos a un fichero ya existente.

El método más popular de FileWriter, heredado de Writer, es el método write() que puede aceptar un array de caracteres (buffer), pero también puede aceptar un string:

public void write(String str) throws IOException3

Más información sobre FileWriter es mostrada en:
http://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html

1.2.2 CLASE FILEREADER

El flujo FileReader4 permite leer caracteres desde un fichero de modo secuencial. Esta clase hereda los métodos de la clase Reader. Los constructores principales son:

FileReader(String ruta)
FileReader(File fichero)

El fichero puede abrirse con una ruta de directorios o con un objeto de tipo File. El método más popular de FileReader, heredado de Reader, es el método read() que solo puede aceptar un array de caracteres (buffer):

public int read(char[] cbuf) throws IOException

1.2.3 CLASE FILEOUTPUTSTREAM

El flujo FileOutputStream5 permite escribir bytes en un fichero de manera secuencial. Sus constructores tienen los mismos parámetros que los mostrados para FileWriter: el fichero puede ser abierto vacío o listo para añadirle datos a los que ya contenga.

Ya que este flujo está destinado a ficheros binarios (bytes) todas las escrituras se hacen a través de un buffer (array de bytes).

public void write(byte[] b) throws IOException

1.2.4 CLASE FILEINPUTSTREAM

El flujo FileInputStream6 permite leer bytes en un fichero de manera secuencial. Sus constructores tienen los mismos parámetros que los mostrados para FileReader.

El método más popular de FileInputStream es el método read() que acepta un array de bytes (buffer):

public int read(byte[] cbuf) throws IOException

1.2.5 RANDOMACCESSFILE

El flujo RandomAccessFile7 permite acceder directamente a cualquier posición dentro del fichero. Proporciona dos constructores básicos:

RandomAccessFile(String ruta, String modo)
RandomAccessFile(File fichero, String modo)

4 Más información sobre FileReader es mostrada en:
http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html
5 Más información sobre FileOutputStream es mostrada en:
http://docs.oracle.com/javase/7/docs/api/java/io/FileOutputStream.html
6 Más información sobre FileInputStream es mostrada en:
http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html
7 Más información sobre RandomAccessFile es mostrada en:
http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.htm

El parámetro modo especifica para qué se abre el archivo, por ejemplo, si modo es “r” el fichero se abrirá en modo “solo lectura”, sin embargo, si modo es “rw” el fichero se abrirá en modo “lectura y escritura”.

1.2.6 EJEMPLO DE USO DE FLUJOS

Como se puede apreciar por la descripción de los flujos anteriores, todos son muy parecidos respecto a los parámetros que reciben los constructores y los nombres de los métodos que manejan. Esto hace más fácil su utilización. De manera resumida, el uso de flujos comparte los siguientes pasos:

Se abre el fichero: se crea un objeto de la clase correspondiente al tipo de fichero que se quiere manejar y el modo de acceso. De manera general la sintaxis es:

```
TipoDeFichero obj = new TipoDeFichero(ruta);
```

El parámetro ruta puede ser una cadena de texto con la ruta de acceso al fichero en el sistema operativo, o puede ser un objeto de tipo File (esta opción es la más elegante).

La sintaxis anterior es válida para los objetos de tipo FileInputStream, FileOutpuStream, FileReader y FileWriter. Sin embargo, para objetos RandomAccessFile (acceso aleatorio) es necesario indicar también el modo en el que se desea abrir el fichero: “r”, solo lectura; o “rw”, lectura y escritura:

```
RandomAccessFile obj = new RandomAccessFile(ruta,modo);
```

Se utiliza el fichero: se utilizan los métodos específicos de cada clase para leer o escribir. Ya que cada clase representa un tipo de fichero y un modo de acceso diferente, los métodos son diferentes también.

Gestión de excepciones: todos los métodos que utilicen funcionalidad de java.io deben tener en su definición una cláusula throws IOException. Los métodos de estas clases pueden lanzar excepciones de esta clase (o sus hijas) en el transcurso de su ejecución, y dichas excepciones deben ser capturadas y debidamente gestionadas para evitar problemas.

Se cierra el fichero y se destruye el objeto: una vez se ha terminado de trabajar con el fichero lo recomendable es cerrarlo usando el método close() de cada clase.

```
obj.close();
```

El siguiente código muestra un ejemplo de acceso a un fichero siguiendo los pasos anteriores.

1. El código escribe en un fichero de texto llamado libros.xml una cadena de texto con un fragmento de XML:

   ```
   <Libros><Libro><Titulo>El Capote</Titulo></Libro></Libros>
   ```

2. Se crea un flujo FileWriter, se escrite la cadena con write() y se cierra con close().

   ```
   import java.io.FileWriter;
   public void EscribeFicheroTexto() {
   //Crea el String con la cadena XML
   ```

```
String texto =
"<Libros><Libro><Titulo>El Capote</Titulo></Libro></Libros>";

//Guarda en nombre el nombre del archivo que se creará.
String nombre = "libros.xml";

try{

//Se crea un Nuevo objeto FileWriter
FileWriter fichero = new FileWriter(nombre);

//Se escribe el fichero
fichero.write(texto + "\r\n");

//Se cierra el fichero
fichero.close();

 }catch(IOException ex){
        System.out.println("error al acceder al fichero");
        }
}
```

ACTIVIDADES 1.1

- (Modifica el código anterior para escribir un fichero XML bien formado y leerlo posteriormente, mostrando la salida por pantalla.

1.3 TRABAJO CON FICHEROS XML (EXTENDED MARKUP LANGUAGE)

El lenguaje de marcas extendido (eXtended Markup Language [XML]) ofrece la posibilidad de representar la información de forma neutra, independiente del lenguaje de programación y del sistema operativo empleado. Su utilidad en el desarrollo de aplicaciones software es indiscutible actualmente. Muchas son las tecnologías que se han diseñado gracias a las posibilidades ofrecidas por XML, un ejemplo de ellas son los servicios web.

Desde un punto de vista a "bajo nivel", un documento XML no es otra cosa que un fichero de texto. Realmente nada impide utilizar librerías de acceso a ficheros, como las vistas en la sección anterior, para acceder y manipular ficheros XML.

Sin embargo, desde un punto de vista a "alto nivel", un documento XML no es un mero fichero de texto. Su uso intensivo en el desarrollo de aplicaciones hace necesarias herramientas específicas (librerías) para acceder y manipular este tipo de archivos de manera potente, flexible y eficiente. Estas herramientas reducen los tiempos de desarrollo de aplicaciones y permiten optimizar los propios accesos a XML. En esencia, estas herramientas permiten manejar los documentos XML de forma simple y sin cargar innecesariamente el sistema. XML nunca hubiese tenido la importancia que tiene en el desarrollo de aplicaciones si permitiera almacenar datos pero luego los sistemas no pudiesen acceder fácilmente a esos datos.

Las herramientas que leen el lenguaje XML y comprueban si el documento es válido sintácticamente se denominan analizadores sintácticos o parsers. Un parser XML es un módulo, biblioteca o programa encargado de transformar el fichero de texto en un modelo interno que optimiza su acceso. Para XML existen un gran número de parsers o analizadores sintácticos disponibles para dos de los modelos más conocidos: DOM y SAX, aunque existen muchos otros. Estos parsers tienen implementaciones para la gran mayoría de lenguajes de programación: Java, .NET, etc.

Una clasificación de las herramientas para el acceso a ficheros XML es la siguiente:

- Herramientas que validan los documentos XML. Estas comprueban que el documento XML al que se quiere acceder está bien formado (well-formed) según la definición de XML y, además, que es válido con respecto a un esquema XML (XML-Schema). Un ejemplo de este tipo de herramientas es JAXB (Java Architecture for XML Binding).

- Herramientas que no validan los documentos XML. Estas solo comprueban que el documento está bien formado según la definición XML, pero no necesitan de un esquema asociado para comprobar si es válido con respecto a ese esquema. Ejemplos de este tipo de herramientas son DOM y SAX.

A continuación se describe el acceso a datos con las tecnologías más extendidas: DOM, SAX y JAXB.

1.4 ACCESO A DATOS CON DOM (*DOCUMENT OBJECT MODEL*)

La tecnología DOM (Document Object Model) es una interfaz de programación que permite analizar y manipular dinámicamente y de manera global el contenido, el estilo y la estructura de un documento. Tiene su origen en el Consorcio World Wide Web (W3C).[8] Esta norma está definida en tres niveles: Nivel 1, que describe la funcionalidad básica de la interfaz DOM, así como el modo de navegar por el modelo de un documento general; Nivel 2, que estudia tipos de documentos específicos (XML, HTML, CSS); y Nivel3, que amplía las funcionalidades de la interfaz para trabajar con tipos de documentos específicos.[9]

Para trabajar con un documento XML primero se almacena en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales que son aquellos que no tienen descendientes. En este modelo todas las estructuras de datos del documento XML se transforman en algún tipo de nodo, y luego esos nodos se organizan jerárquicamente en forma de árbol para representar la estructura descrita por el documento XML.

[8] http://www.w3.org/DOM/
[9] http://www.w3.org/DOM/DOMTR

Una vez creada en memoria esta estructura, los métodos de DOM permiten recorrer los diferentes nodos del árbol y analizar a qué tipo particular pertenecen. En función del tipo de nodo, la interfaz ofrece una serie de funcionalidades u otras para poder trabajar con la información que contienen.

La Figura 1.1 muestra un documento XML para representar libros. Cada libro está definido por un atributo publicado_en, un texto que indica el año de publicación del libro y por dos elementos hijo: *Título y Autor*.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Figura 1.1. Documento XML

Un esquema del árbol DOM que representaría internamente este documento es mostrado en la Figura 1.2. El árbol DOM se ha creado en base al documento XML de la Figura 1.1. Sin embargo, para no enturbiar la claridad del gráfico solo se ha desarrollado hasta el final uno de los elementos L*ibro*, dejando los otros dos sin detallar.

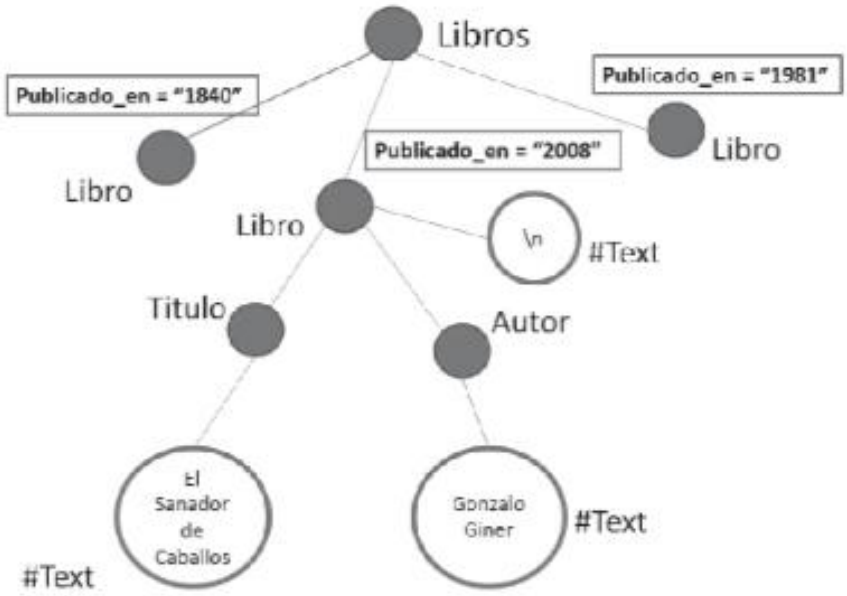


Figura 1.2. Ejemplo DOM

La generación del árbol DOM a partir de un documento XML se hace de la siguiente manera:[10]

1. Aunque el documento XML de la Figura 1.1 tiene asociado un esquema XML, la librería de DOM, a no ser que se le diga lo contrario, no necesita validar el documento con respecto al esquema para poder general el árbol. Solo tiene en cuenta que el documento esté bien formado.

2. El primer nodo que se crea es el nodo Libros que representa el elemento <Libros>.

3. De <Libros> cuelgan en el documento tres hijos <Libro> de tipo elemento, por tanto el árbol crea 3 nodos Librodescendiente de Libro.

4. Cada elemento <Libro> en el documento tiene asociado un atributo publicado_en. En DOM, los atributos son listas con el nombre del atributo y el valor. La lista contiene tantas tuplas (nombre, valor) como atributos haya en el elemento. En este caso solo hay un atributo en el elemento <Libro>.

5. Cada <Libro> tiene dos elementos que descienden de él y que son <Titulo> y <Autor>. Al ser elementos, estos son representados en DOM como nodos descendientes de Libro, al igual que se ha hecho con Libro al descender de Libros.

6. Cada elemento <Titulo> y <Autor> tiene un valor que es de tipo cadena de texto. Los valores de los elementos son representados en DOM como nodos #text. Sin duda esta es la parte más importante del modelo DOM. Los valores de los elementos son nodos también, a los que internamente DOM llama #text y que descienden del nodo que representa el elemento que contiene ese valor. DOM ofrece funciones para recuperar el valor de los nodos #text. Un error muy común cuando se empieza por primera vez a trabajar con árboles DOM es pensar que, por ejemplo, el valor del nodo Título es el texto que contiene el elemento <Titulo> en el documento XML. Sin embargo, eso no es así. Si se quiere recuperar el valor de un elemento, es necesario acceder al nodo #text que desciende de ese nodo y de él recuperar su valor.

7. Hay que tener en cuenta que cuando se edita un documento XML, al ser este de tipo texto, es posible que, por legibilidad, se coloque cada uno de los elementos en líneas diferentes o incluso que se utilicen espacios en blanco para separar los elementos y ganar en claridad. Eso mismo se ha hecho con el documento de la Figura 1.1. El documento queda mucho más legible así que si se pone todo en una única línea sin espacios entre las etiquetas. DOM no distingue cuándo un espacio en blanco o un salto de línea (\n) se hace porque es un texto asociado a un elemento XML o es algo que se hace por “estética”. Por eso, DOM trata todo lo que es texto de la misma manera, creando un nodo de tipo #text y poniéndole el valor dentro de ese nodo. Eso mismo es lo que ha ocurrido en el ejemplo de la Figura 1.2. El nodo #text que desciende de Libro y que tiene como valor “\n” (salto de línea) es creado por DOM ya que, por estética, se ha hecho un salto de línea dentro del documento XML de la Figura 1.1, para diferenciar claramente que la etiqueta <Titulo> es descendiente de <Libro>. Sin duda, en el documento XML hay muchos más “saltos de línea” que se han empleado para dar claridad al documento, sin embargo, en el ejemplo del modelo DOM no se han incluido ya que tantos nodos con “saltos de línea” dejarían el esquema ilegible.

[10] Además del ejemplo mostrado aquí, en http://www.youtube.com/watch?v=c9YSuTg7Sg0&feature=plcp se ofrece un vídeo con una explicación de generación de DOM para otro ejemplo.

8. Por último, un documento XML tiene muchas más “cosas” que las mostradas en el ejemplo de la Figura 1.1, por ejemplo: comentarios, encabezados, espacios en blanco, entidades, etc., son algunas de ellas. Cuando se trabaja con DOM, rara vez esos elementos son necesarios para el programador, por lo que la librería DOM ofrece funciones que omiten estos elementos antes de crear el árbol, agilizando así el acceso y modificación del árbol DOM.

1.4.1 DOM Y JAVA

DOM ofrece una manera de acceder a documentos XML tanto para ser leído como para ser modificado. Su único inconveniente es que el árbol DOM se crea todo en memoria principal, por lo que si el documento XML es muy grande, la creación y manipulación de DOM sería intratable.

DOM, al ser una propuesta W3C, fue muy apoyado desde el principio, y eso ocasionó que aparecieran un gran número de librerías (parsers) que permitían el acceso a DOM desde la mayoría de lenguajes de programación. Esta sección se centra en Java como lenguaje para manipular DOM, pero dejando claro que otros lenguajes también tienen sus librerías (muy similares) para gestionarlo, por ejemplo Microsoft .NET.

Java y DOM han evolucionado mucho en el último lustro. Muchas han sido las propuestas para trabajar desde Java con la gran cantidad de parsers DOM que existen. Sin embargo, para resumir, actualmente la propuesta principal se reduce al uso de JAXP (Java API for XML Processing). A través de JAXP los diversos parsers garantizan la interoperabilidad de Java. JAXP es incluido en todo JDK (superior a 1.4) diseñado para Java. La Figura 1.3 muestra una estructura de bloques de una aplicación que accede a DOM. Una aplicación que desea manipular documentos XML accede a la interfaz JAXP porque le ofrece una manera transparente de utilizar los diferentes parsers que hay para manejar XML. Estos parsers son para DOM (XT, Xalan, Xerces, etc.) pero también pueden ser parsers para SAX (otra tecnología alternativa a DOM, que se verá en la siguiente sección).

Figura 1.3. Relación entre JAXP y parsers

En los ejemplos mostrados en las siguientes secciones se utilizará la librería Xerces para procesar representaciones en memoria de un documento XML considerando un árbol DOM. Entre los paquetes concretos que se usarán destacan:

javax.xml.parsers.*, en concreto *javax.xml.parsers.DocumentBuilder* y *javax.xml.parsers.DocumentBuilderFactory*, para el acceso desde JAXP.

org.w3c.dom.* que representa el modelo DOM según la W3C (objetos *Node, NodeList, Document,* etc. y los métodos para manejarlos). Por lo tanto, todas las especificaciones de los diferentes niveles y módulos que componen DOM están definidas en este paquete.

La Figura 1.4 muestra un esquema que relaciona JAXP con el acceso a DOM. Desde la interfaz JAXP se crea un *DocumentBuilderFactory*. A partir de esa factoría se crea un *DocumentBuilder* que permitirá cargar en él la estructura del árbol DOM (Árbol de Nodos) desde un fichero XML (Entrada XML).

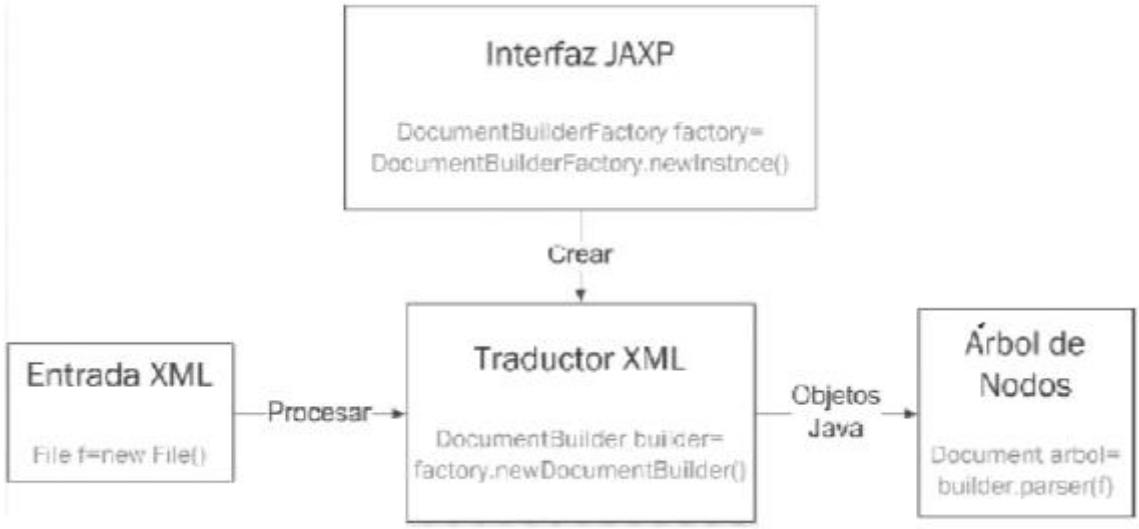


Figura 1.4. Esquema de acceso a DOM desde JAXP

La clase *DocumentBuilderFactory* tiene métodos importantes para indicar qué interesa y qué no interesa del fichero XML para ser incluido en el árbol DOM, o si se desea validar el XML con respecto a un esquema. Algunos de estos métodos son los siguientes:[11]

- *setIgnoringComments(boolean ignore*) sirve para ignorar los comentarios que tenga el fichero XML.
- *setIgnoringElementContentWhitespace(boolean ignore)* es útil para eliminar los espacios en blanco que no tienen significado.
- *setNamespaceAware(boolean aware)* usado para interpretar el documento usando el espacio de nombres.
- *setValidating(boolean validate)* que valida el documento XML según el esquema definido.

Con respecto a los métodos que sirven para manipular el árbol DOM, que se encuentran en el paquete org.w3c.dom, destacan los siguientes métodos asociados a la clase Node:[12]

- Todos los nodos contienen los métodos *getFirstChild()* y *getNextSibling()* que permiten obtener uno a uno los nodos descendientes de un nodo y sus hermanos.
- El método *getNodeType()* devuelve una constante para poder distinguir entre los diferentes tipos de nodos: nodo de tipo Elemento (ELEMENT_NODE), nodo de tipo #text (TEXT_NODE), etc. Este método y las constantes

[11] Más información sobre los métodos que ofrece DocumentBuilderFactory es mostrada en: http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/DocumentBuilderFactory.html

[12] Más información sobre la interfaz Node es mostrada en: http://www.w3.org/2003/01/dom2-javadoc/org/w3c/dom/Node.html

asociadas son especialmente importantes a la hora de recorrer el árbol ya que permiten ignorar aquellos tipos de nodos que no interesan (por ejemplo, los #text que tengan saltos de línea).

- El método *getAttributes()* devuelve un objeto *NamedNodeMap* (una lista con sus atributos) si el nodo es del tipo *Elemento*.
- Los métodos *getNodeName()* y *getNodeValue()* devuelven el nombre y el valor de un nodo de forma que se pueda hacer una búsqueda selectiva de un nodo concreto. El error típico es creer que el método *getNodeValue()* aplicado a un nodo de tipo Elemento (por ejemplo, <Titulo>) devuelve el texto que contiene. En realidad, es el nodo de tipo #text (descendiente de un nodo tipo Elemento) el que tiene el texto que representa el título del libro y es sobre él sobre el que hay que aplicar el método *getNodeValue()* para obtener el título.

En las siguientes secciones se usarán ejemplos para mostrar diferentes accesos a XML con DOM.

En el material adicional incluido en este libro se puede encontrar la carpeta AccesoDOM, que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con DOM (SAX y JAXB).

1.4.2 ABRIR DOM DESDE JAVA

Para abrir un documento XML desde Java y crear un árbol DOM con él se utilizan las clases *DocumentBuilderFactory* y *DocumentBuilder*, que pertenecen al paquete *javax.xml.parsers*, y *Document*, que representa un documento en DOM y pertenece al paquete *org.w3c.dom*. Aunque existen otras posibilidades, en el ejemplo mostrado seguidamente se usa un objeto de la clase *File* para indicar la localización del archivo XML.

```
public int abrir_XML_DOM (File fichero) {
doc=null; //doc es de tipo Document y representa al árbol DOM
        try {
        //Se crea un objeto DocumentBuiderFactory.
        DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
        //Indica que el modelo DOM no debe contemplar los comentarios que tenga el XML.
        factory.setIgnoringComments(true);
        //Ignora los espacios en blanco que tenga el documento
        factory.setIgnoringElementContentWhitespace(true);
        //Se crea un objeto DocumentBuilder para cargar en él la estructura de
        //árbol DOM a partir del XML seleccionado.
        DocumentBuilder builder=factory.newDocumentBuilder();
        //Interpreta (parsea) el documento XML (file) y genera el DOM equivalente.
        doc=builder.parse(fichero);
        //Ahora doc apunta al árbol DOM listo para ser recorrido.
        return 0;
         }
        catch(Exception e) {
        e.printStackTrace();
        return -1;
        }
}
```

Como se puede entender siguiendo los comentarios del código, primeramente se crea factory y se prepara el *parser* para interpretar un fichero XML en el cual ni los espacios en blanco ni los comentarios serán tenidos en cuenta a la hora de generar el DOM. El método *parse()* de *DocumentBuilder* (creado a partir de un *DocumentoBuilderFactory*) recibe como entrada un *File* con la ruta del fichero XML que se desea abrir y devuelve un objeto de tipo *Document*. Este objeto (doc) es el árbol DOM cargado en memoria, listo para ser tratado.

1.4.3 RECORRER UN ÁRBOL DOM

Para recorrer un árbol DOM se utilizan las clases *Node* y *NodeList* que pertenecen al paquete org.w3c.dom. En el ejemplo de esta sección se ha recorrido el árbol DOM creado del documento XML mostrado en la Figura 1.1. El resultado de procesar dicho documento origina la siguiente salida:

Publicado en: 1840
El autor es: Nikolai Gogol
El título es: El Capote

Publicado en: 2008
El autor es: Gonzalo Giner
El título es: El Sanador de Caballos

Publicado en: 1981
El autor es: Umberto Eco
El título es: El Nombre de la Rosa

El siguiente código recorre el árbol DOM para dar la salida anterior con los nombres de los elementos que contiene cada <Libro> (<Titulo>, <Autor>), sus valores y el valor y nombre del atributo de <Libro> (publicado_en).

```
public String recorrerDOMyMostrar (Document doc) {
        String datos_nodo[]=null;
        String salida="";
        Node node;
        //Obtiene el primero nodo del DOM (primer hijo)
        Node raiz=doc.getFirstChild();
        //Obtiene una lista de nodos con todos los nodos hijo del raíz.
        NodeList nodelist=raiz.getChildNodes();
        //Procesa los nodos hijo
        for (int i=0; i<nodelist.getLength(); i++)
        {
```

```
        node = nodelist.item(i);
                if(node.getNodeType()==Node.ELEMENT_NODE){
                        //Es un nodo libro
                        datos_nodo=procesarLibro(node);
                salida=salida + "\n " + "Publicado en: " + datos_nodo[0];
                salida=salida + "\n " + "El autor es: " + datos_nodo[2];
                salida=salida + "\n " + "El título es: " + datos_nodo[1];
                salida=salida + "\n ------------------";
                }
        }
        return salida;
}
```

El código anterior, partiendo del objeto tipo *Document* (doc) que contiene el árbol DOM, recorre dicho árbol para sacar los valores del atributo de <Libro> y de los elementos <Titulo> y <Autor>. Es una práctica común al trabajar con DOM comprobar el tipo del nodo que se está tratando en cada momento ya que, como se ha comentado antes, un DOM tiene muchos tipos de nodos que no siempre tienen información que merezca la pena explorar. En el código, solo se presta atención a los nodos de tipo Elemento (ELEMENT_NODE) y de tipo #Text (TEXT_NODE).

Por último, queda ver el código del método *ProcesarLibro*:

```
protected String[] procesarLibro(Node n){

        String datos[]= new String[3];
        Node ntemp=null;
        int contador=1;
        //Obtiene el valor del primer atributo del nodo (solo uno en este ejemplo)
        datos[0]=n.getAttributes().item(0).getNodeValue();

        //Obtiene los hijos del Libro (titulo y autor)
        NodeList nodos=n.getChildNodes();
        for (int i=0; i<nodos.getLength(); i++)
        {
                ntemp = nodos.item(i);
                if(ntemp.getNodeType()==Node.ELEMENT_NODE){
                        //IMPORTANTE: para obtener el texto con el título y autor se accede al
                        //nodo TEXT hijo de ntemp y se saca su valor.
                        datos[contador]= ntemp.getChildNodes().item(0).getNodeValue();
                        contador++;
                }
        }
        return datos;
        }
```

En el código anterior lo más destacable es:

- Que el programador sabe que el elemento <Libro> solo tiene un atributo, por lo que accede directamente a su valor *(n.getAttributes().item(0).getNodeValue())*.
- Una vez detectado que se está en el nodo de tipo Elemento (que puede ser el título o el autor) entonces se obtiene el hijo de este (tipo #text) y se consulta su valor *(ntemp.getChildNodes().item(0).getNodeValue())*.

1.4.4 MODIFICAR Y SERIALIZAR

Además de recorrer en modo "solo lectura" un árbol DOM, este también puede ser modificado y guardado en un fichero para hacerlo persistente. En esta sección se muestra un código para añadir un nuevo elemento a un árbol DOM y luego guardar todo el árbol en un documento XML. Es importante destacar lo fácil que es realizar modificaciones con la librería DOM. Si esto mismo se quisiera hacer accediendo a XML como si fuera un fichero de texto "normal" (usando FileWriter, por ejemplo), el código necesario sería mucho mayor y el rendimiento en ejecución bastante más bajo.

El siguiente código añade un nuevo libro al árbol DOM (doc) con los valores de publicado_en, título y autor, pasados como parámetros.

```
public int annadirDOM(Document doc,  String titulo, String autor, String anno){
        try{
                //Se crea un nodo tipo Element con nombre 'titulo'(<Titulo>)
                Node ntitulo=doc.createElement("Titulo");
                //Se crea un nodo tipo texto con el título del libro Node
                ntitulo_text=doc.createTextNode(titulo);
                 //Se añade el nodo de texto con el título como hijo del elemento Titulo
                ntitulo.appendChild(ntitulo_text);
                //Se hace lo mismo que con título a autor (<Autor>)
                Node nautor=doc.createElement("Autor");
                Node nautor_text=doc.createTextNode(autor);
                nautor.appendChild(nautor_text);

                //Se crea un nodo de tipo elemento (<libro>)
                Node nlibro=doc.createElement("Libro");
                //Al nuevo nodo libro se le añade un atributo publicado_en
                ((Element)nlibro).setAttribute("publicado_en",anno );

                //Se añade a libro el nodo autor y titulo creados antes
                nlibro.appendChild(ntitulo);
                nlibro.appendChild(nautor);

                //Finalmente, se obtiene el primer nodo del documento y a él se le
                //añade como hijo el nodo libro que ya tiene colgando todos sus
                //hijos y atributos creados antes.
                Node raiz=doc.getChildNodes().item(0);

                raiz.appendChild(nlibro);
```

```
            return 0;
            }catch(Exception e){
            e.printStackTrace();
        return -1;}
        }
```

Revisando el código anterior se puede apreciar que para añadir nuevos nodos a un árbol DOM hay que conocer bien cómo de diferente es el modelo DOM con respecto al documento XML original. La prueba es la necesidad de crear nodos de texto que sean descendientes de los nodos de tipo Elemento para almacenar los valores XML.

Una vez se ha modificado en memoria un árbol DOM, éste puede ser llevado a fichero para lograr la persistencia de los datos. Esto se puede hacer de muchas maneras. Una alternativa concreta se recoge en el siguiente código.

En el ejemplo se usa las clases *XMLSerializer* y *OutFormat* que están definidas en los paquetes *com.sun.org.apache.xml.internal.serialize.OutputFormat* y *com.sun.org.apache.xml.internal.serialize.XMLSerializer*.
Estas clases realizan la labor de serializar un documento XML.

Serializar (en inglés *marshalling*) es el proceso de convertir el estado de un objeto (DOM en nuestro caso) en un formato que se pueda almacenar. Serializar es, por ejemplo, llevar el estado de un objeto en Java a un fichero o, como en nuestro caso, llevar los objetos que componen un árbol DOM a un fichero XML. El proceso contrario, es decir, pasar el contenido de un fichero a una estructura de objetos, se conoce por *unmarshalling*.

La clase *XMLSerializer* se encarga de serializar un árbol DOM y llevarlo a fichero en formato XML bien formado. En este proceso se utiliza la clase *OutputFormat* porque permite asignar diferentes propiedades de formato al fichero resultado, por ejemplo que el fichero esté indentado (anglicismo, indent) es decir, que los elementos hijos de otro elemento aparezcan con más tabulación a la derecha para así mejorar la legibilidad del fichero XML.

```
public int guardarDOMcomoFILE() {
        try{
                //Crea un fichero llamado salida.xml
                File archivo_xml = new File("salida.xml");
                //Especifica el formato de salida
                OutputFormat format = new OutputFormat(doc);
                //Especifica que la salida esté indentada.
                format.setIndenting(true);
                //Escribe el contenido en el FILE
                XMLSerializer serializer = new XMLSerializer(new FileOutputStream(archivo_xml), format);
                serializer.serialize(doc);
                return 0;
        }catch(Exception e) {
                return -1;
                }
        }
```

Según el código anterior, para serializar un árbol DOM se necesita:

- Un objeto *File* que representa al fichero resultado (en el ejemplo será salida.xml).
- Un objeto *OutputFormat* que permite indicar pautas de formato para la salida.
- Un objeto X*MLSerializer* que se construye con el File de salida y el formato definido con un *OutputFormat*. En el ejemplo utiliza un objeto *FileOutputStream* para representar el flujo de salida.
- Un método *serialize()* de *XMLSerializer* que recibe como parámetro el Document (doc) que quiere llevar a fichero, y lo escribe.

ACTIVIDADES 1.2

- Utiliza el código anterior para hacer un método que permita modificar los valores de un libro:
  a. A la función se le pasa el título de un libro y se debe cambiar por un nuevo título que también se le pasa como parámetro.
  b. El resultado debe ser guardado en un nuevo documento XML llamado "modificación.xml".

AYUDA
Se puede extender el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2 para facilitar el trabajo.

1.5 ACCESO A DATOS CON SAX (SIMPLE API FOR XML)

SAX[13] (*Simple API for XML*) es otra tecnología para poder acceder a XML desde lenguajes de programación. Aunque SAX tiene el mismo objetivo que DOM esta aborda el problema desde una óptica diferente. Por lo general, se usa SAX cuando la información almacenada en los documentos XML es clara, está bien estructurada y no se necesita hacer modificaciones.

[13] http://www.saxproject.org

Las principales características de SAX son:

- SAX ofrece una alternativa para leer documentos XML de manera secuencial. El documento solo se lee una vez. A diferencia de DOM, el programador no se puede mover por el documento a su antojo. Una vez que se abre el documento, se recorre secuencialmente el fichero hasta el final. Cuando llega al final se termina el proceso (parser).
- SAX, a diferencia de DOM, no carga el documento en memoria, sino que lo lee directamente desde el fichero. Esto lo hace especialmente útil cuando el fichero XML es muy grande.

SAX sigue los siguientes pasos básicos:

1. Se le dice al *parser* SAX qué fichero quiere que sea leído de manera secuencial.
2. El documento XML es traducido a una serie de eventos.
3. Los eventos generados pueden controlarse con métodos de control llamados *callbacks*.
4. Para implementar los *callbacks* basta con implementar la interfaz *ContentHandler* (su implementación por defecto es *DafaultHandler*).

El proceso se puede resumir de la siguiente manera:

- SAX abre un archivo XML y coloca un puntero en al comienzo del mismo.
- Cuando comienza a leer el fichero, el puntero va avanzando secuencialmente.
- Cuando SAX detecta un elemento propio de XML entonces lanza un evento. Un evento puede deberse a:
  - Que SAX haya detectado el comienzo del documento XML.
  - Que se haya detectado el final del documento XML.
  - Que se haya detectado una etiqueta de comienzo de un elemento, por ejemplo <libro>.
  - Que se haya detectado una etiqueta de final de un elemento, por ejemplo </libro>.
  - Que se haya detectado un atributo.
  - Que se haya detectado una cadena de caracteres que puede ser un texto.
  - Que se haya detectado un error (en el documento, de I/O, etc.).
- Cuando SAX devuelve que ha detectado un evento, entonces este evento puede ser manejado con la clase *DefaultHandler* (*callbacks*). Esta clase puede ser extendida y los métodos de esta clase pueden ser redefinidos (sobrecargados) por el programador para conseguir el efecto deseado cuando SAX detecta los eventos. Por ejemplo, se puede redefinir el método *public void startElement()*, que es el que se invoca cuando SAX detecta un evento de comienzo de un elemento. Como ejemplo, la redefinición de este método puede consistir en comprobar el nombre del nuevo elemento detectado, y si es uno en concreto entonces sacar por pantalla un mensaje con su contenido.
- Cuando SAX detecta un evento de error o un final de documento entonces se termina el recorrido.

En las siguientes secciones se muestra el acceso a SAX desde Java.

En el material adicional incluido en este libro se puede encontrar la carpeta AccesoDOM que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con SAX (DOM y JAXB).

1.5.1 ABRIR XML CON SAX DESDE JAVA

Para abrir un documento XML desde Java con SAX se utilizan las clases: *SAXParserFactory* y *SAXParser* que pertenecen al paquete *javax.xml.parsers*. También es necesario extender la clase *DefaultHandler* que se encuentra en el paquete *org.xml.sax.helpers.DefaultHandler*. Además, en el ejemplo mostrado a continuación se usa la clase *File* para indicar la localización del archivo XML. Las clases *SAXParserFactory*[14] y *SAXParser*[15] proporcionan el acceso desde JAXP. La clase *DefaultHandler*[16] es la clase base que atiende los eventos devueltos por el *parser*. Esta clase se extiende en las aplicaciones para personalizar el comportamiento del *parser* cuando se encuentra un elemento XML.

```
public int abrir_XML_SAX (ManejadorSAX sh, SAXParser parser ){
        try{
                SAXParserFactory factory=SAXParserFactory.newInstance ();
                //Se crea un objeto SAXParser para interpretar el documento XML.
                parser=factory.newSAXParser();
                //Se crea una instancia del manejador que será el que recorra el documento
                //XML secuencialmentesh=new ManejadorSAX();
                return 0;
                } catch (Exception e) {
                        e.printStackTrace();
                        return -1;
                        }
                }
```

Como se puede entender siguiendo los comentarios del código, primeramente se crean los objetos *factory* y *parser*. Esta parte es similar a como se hace con DOM. Una diferencia con DOM es que en SAX se crea una instancia de la clase *ManejadorSAX*. Esta clase extiende *DefaultHandler* y redefine los métodos (*callbacks*) que atienden a los eventos. En resumen, la preparación de SAX requiere inicializar las siguientes variables, que serán usadas cuando se inicie el proceso de recorrido del fichero XML:

- Un objeto *parser*: en el código la variable se llama *parser*.
- Una instancia de una clase que extienda *DefaultHandler*, que en el ejemplo es *ManejadorSAX*. La variable se llama *sh*.

[14] Más información sobre SAXParserFactory es mostrada en:
http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParser-Factory.html

[15] Más información sobre SAXParser es mostrada en:
http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParser.html

[16] Más información sobre DefaultHandler es mostrada en:
http://docs.oracle.com/javase/7/docs/api/org/xml/sax/helpers/DefaultHandler.html

En la sección siguiente se muestra el código de la clase *ManejadorSAX* que es el código que gestiona cómo interpretar los elementos de un documento XML.

1.5.2 RECORRER XML CON SAX

Para recorrer un documento XML una vez inicializado el parser lo único que se necesita es lanzar el *parser*. Evidentemente, antes es necesario haber definido la clase que extiende *DefaultHandle*r (en el ejemplo anterior era *ManejadorSAX*). Esta clase tiene la lógica de cómo actuar cuando se encuentra algún elemento XML durante el recorrido con SAX (*callbacks*).

Un ejemplo de clase *ManejadorSAX* es el siguiente:

```
class ManejadorSAX extends DefaultHandler {

    int ultimoelement;
    String cadena_resultado= “”;
    public ManejadorSAX () {
    ultimoelement=0;
    }

    //Se sobrecarga (redefine) el método startElement
    @Override
    public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException {
            if(qName.equals(“Libro”)) {
             cadena_resultado=cadena_resultado + “Publicado en: “ + atts.getValue(atts.getQName(0))+ “\n “;
            ultimoelement=1;
            }
            else if (qName.equals(“Titulo”)) {
            ultimoelement=2;
            cadena_resultado= cadena_resultado + “\n “ +”El título es: “;
            }
            else if (qName.equals(“Autor”)) {
            ultimoelement=3;
            cadena_resultado= cadena_resultado + “\n “ +”El autor es: “;
            }
      }
      //Cuando en este ejemplo se detecta el final de un elemento <libro>, se pone una línea
      //discontinua en la salida.

      @Override
      public void endElement(String uri, String localName, String qName) throws SAXException {
            if (qName.equals(“Libro”)) {
            cadena_resultado = cadena_resultado + “\n ----------”;
            }
    }
      @Override
      public void characters (char[] ch, int start, int length) throws SAXException {
            if(ultimoelement==2) {
            for(int i=start; i<length+start; i++)
              cadena_resultado=cadena_resultado+ch[i];
            }
```

```
            else if (ultimoelement==3) {
                    for(int i=start; i<length+start; i++)
                    cadena_resultado= cadena_resultado+ch[i];
            }
    }
}
```

Esta clase extiende el método *startElement*, *endElement* y *characters*. Estos métodos (*callbacks*) se invocan cuando, durante el recorrido del documento XML, se detecta un evento de comienzo de elemento, final de elemento o cadena de caracteres. En el ejemplo, cada método realiza lo siguiente:

- *startElement()* : cuando se detecta con SAX un evento de comienzo de un elemento, entonces SAX invoca a este método. Lo que hace es comprobar de qué tipo de elemento se trata.
  - Si es <Libro> entonces saca el valor de su atributo y lo concatena con una cadena (*cadena_resultado*) que tendrá toda la salida después de recorrer todo el documento.
  - Si es <Titulo> entonces a *cadena_resultado* se le concatena el texto “El título es:”.
  - Si es <Autor> entonces a *cadena_resultado* se le concatena el texto “El autor es:”.
  - Si es otro tipo de elemento no hará nada.

- *endElement()* : cuando se detecta con SAX un evento de final de un elemento, entonces SAX invoca a este método. El método comprueba si es el final de un elemento <Libro>. Si es así, entonces a *cadena_resultado* se le concatena el texto “\n ----------”.

- *characters()* : cuando se detecta con SAX un evento de detección de cadena de texto, entonces SAX invoca a este método. El método lo que hace es concatenar a *cadena_resultado* cada uno de los caracteres de la cadena detectada.

Si se aplica el código anterior al contenido del documento XML de la Figura 1.1, el resultado sería el siguiente:

```
Publicado en: 1840

El título es: El Capote

El autor es: Nikolai Gogol

----------

Publicado en: 2008

El título es: El Sanador de Caballos

El autor es: Gonzalo Giner

----------

Publicado en: 1981

El título es: El Nombre de la Rosa

El autor es: Umberto Eco

----------
```

El código que lanza el parser SAX para obtener el resultado anterior es el siguiente:

```
public String recorrerSAX(File fXML,ManejadorSAX sh,SAXParser parser ){
        try{
                parser.parse(fXML,sh);
                return sh.cadena_resultado;
        } catch (SAXException e) {
                 e.printStackTrace();
                return "Error al parsear con SAX";
        } catch (Exception e) {
                e.printStackTrace();
                return "Error al parsear con SAX";
        }
}
```

Este método recibe como parámetro el *parser* inicializado (*parser*), la instancia de la clase que manejará los eventos (*sh*) y el *File* con el fichero XML que se recorrerá (fXML). El método *parse()* lanza SAX para el fichero XML seleccionado y con el manejador deseado (se podrían implementar tantas extensiones de *DefaultHandler* como manejadores diferentes se quisieran usar).

En este método la excepción que se captura es *SAXException* que se define en el paquete *org.xml.sax.SAXException*.

ACTIVIDADES 1.3

- Modifica el código anterior para que:

    a. Cuando SAX encuentre un elemento <Libros> aparezca un mensaje que diga "Se van a mostrar los libros de este documento"

AYUDA

Se puede extender el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2. para facilitar el trabajo.

1.6 ACCESO A DATOS CON JAXB (BINDING)

De las alternativas vistas en las secciones anteriores para acceder a documentos XML desde Java, JAXB (*Java Architecture for XML Binding*) es la más potentes y actual de las tres. JAXB (no confundir con la interfaz de acceso JAXP) es una librería de (*un)-marshalling*. El concepto de serialización o *marshalling*, que ha sido introducido en la Sección 1.4.4, es el proceso de almacenar un conjunto de objetos en un fichero. *Unmarshalling* es justo el proceso contrario: convertir en objetos el contenido de un fichero. Para el caso concreto de XML y Java, *unmarshalling* es convertir el contenido de un archivo XML en una estructura de objetos Java.

De manera resumida, JAXB convierte el contenido de un documento XML en una estructura de clases Java:

- El documento XML debe tener un esquema XML asociado (fichero.xsd), por tanto el contenido del XML debe ser válido con respecto a ese esquema.
- JAXB crea la estructura de clases que albergará el contenido del XML en base a su esquema. El esquema es la referencia de JAXB para saber la estructura de las clases que contendrán el documento XML.
- Una vez JAXB crea en tiempo de diseño (no durante la ejecución) la estructura de clases, el proceso de unmarshalling (creación de objetos de las clases creadas con el contenido del XML) y marshalling (almacenaje de los objetos como un documento XML) es sencillo y rápido, y se puede hacer en tiempo de ejecución.

La Figura 1.5 muestra un esquema XML para el documento de la Figura 1.1. El esquema XML indica que existe un elemento <Libros> que contiene uno o varios elementos <Libro> en su interior. Cada elemento <Libro> tiene un atributo *publicado_en* cuyo valor debe ser de tipo *string*, y dos elementos hijos: <Titulo> y <Autor>.

```
▼<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="1.0"
  ▼<xsd:element name="Libros">
    ▼<xsd:complexType>
      ▼<xsd:choice maxOccurs="unbounded">
        ▼<xsd:element name="Libro" minOccurs="0" maxOccurs="unbounded">
          ▼<xsd:complexType>
            ▼<xsd:sequence>
                <xsd:element name="Titulo" type="xsd:string"/>
                <xsd:element name="Autor" type="xsd:string"/>
              </xsd:sequence>
              <xsd:attribute name="publicado_en" type="xsd:string"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
```

Figura 1.5. Esquema XML

JAXB es capaz de obtener de este esquema una estructura de clases que le dé soporte en Java. De manera simplificada, JAXB obtendría las siguientes clases Java:

```
public class Libros {
        protected List<Libros.Libro> libro;

        public List<Libros.Libro> getLibro() {
                if (libro == null) {
                        libro = new ArrayList<Libros.Libro>();
                }
                return this.libro;
         }
}

public static class Libro {
        protected String titulo;
        protected String autor;
        protected String publicadoEn;

        public String getTitulo() {
                return titulo;
        }

        public void setTitulo(String value) {
        this.titulo = value;
        }

        public String getAutor() {
        return autor;
        }

        public void setAutor(String value) {
        this.autor = value;
        }

        public String getPublicadoEn() {
        return publicadoEn;
        }

        public void setPublicadoEn(String value) {
        this.publicadoEn = value;
        }
}
```

Cuando se realiza un *unmarshalling*, JAXB carga el contenido de cualquier documento XML que satisfaga el esquema mostrado en la Figura 1.5 en una estructura de objetos de las clases *Libros* y *Libro*. En un *marshalling*, JAXB convierte una estructura de objetos de las clases *Libros* y *Libro* en un documento XML válido con respecto al esquema de la Figura 1.5.

En las siguientes secciones se muestra cómo se puede implementar el uso de JAXB en Java.

En el material adicional incluido en este libro se puede encontrar la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que muestra el acceso a documentos XML con JAXB (DOM y SAX).

1.6.1 ¿CÓMO CREAR CLASES JAVA DE ESQUEMAS XML?

Partiendo de un esquema XML como el mostrado en la Figura 1.5, el proceso para crear la estructura de clases Java que le de soporte es muy sencillo:

Directamente con un JDK de Java

Con JDK 1.7.0 se pueden obtener las clases asociadas a un esquema XML con la aplicación *xjc*. Para ello, solo es necesario pasar al programa el esquema XML que se quiere emplear en la ejecución. Por ejemplo, suponiendo que el ejemplo de la Figura 1.5 es un fichero llamado *LibrosEsquema.xsd*, la creación de las clases que le den soporte en JAXB sería con el comando:

```
xjc LibrosEsquema.xsd
```

Usando el IDE NetBeans

Con el IDE NetBeans 7.1.2 se pueden obtener las clases asociadas a un esquema XML. Para ello, solo hay que seguir los siguientes pasos:

1. Añadir el fichero con el esquema XML al proyecto en el que se quiere usar JAXB.

2. Seleccionar **Archivo->Nuevo Fichero** para añadir un nuevo elemento al proyecto. En la ventana que aparece para indicar el tipo de fichero que se quiere añadir seleccionar **XML->JAXB Binding**. La Figura 1.6 muestra la ventana con esas opciones seleccionadas.

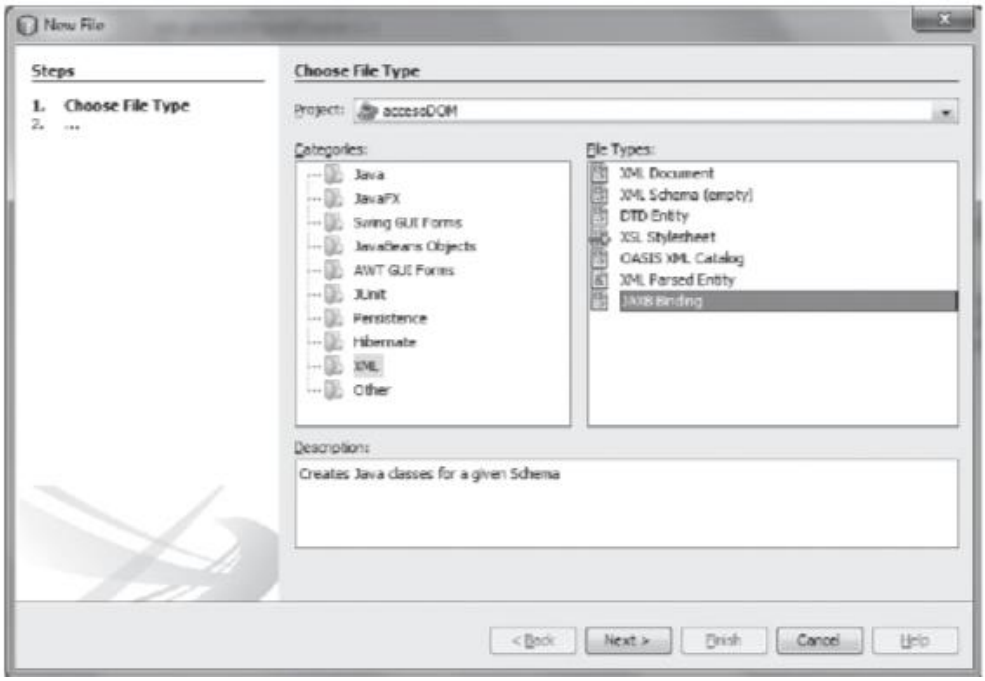


Figura 1.6. JAXB desde NetBeans - nuevo fichero

3. La siguiente ventana se corresponderá con la Figura 1.7 y muestra una serie de campos que definirán el tipo de enlace (*binding*) que se realizará. Los campos más importantes son la localización del esquema XML sobre el que se crearán las clases Java (llamado *LibrosEsquema.xsd*) y el paquete en el que se recogerán las clases nuevas creadas (llamado *javalibros*).

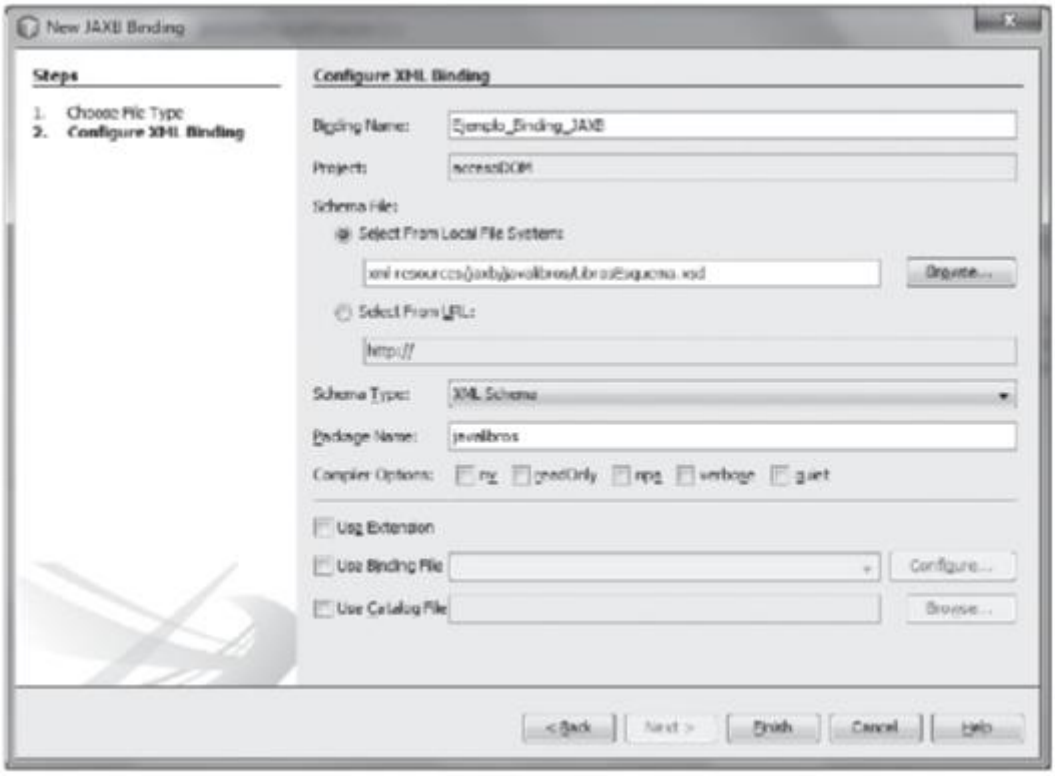


Figura 1.7. JAXB desde NetBeans – Enlace

4. Una vez rellenados esos datos (y tras haber pulsado en **Finish**) se crea una nueva carpeta en el árbol del proyecto con el paquete (*javalibros*) y dentro las clases que enlazan con el esquema XML (*LibrosEsquema.xsd*).

Esas clases creadas ya se pueden utilizar en el proyecto abriendo el enlace JAXB y cargando el documento XML seleccionado en esas estructuras de objetos. Las siguientes secciones muestran el proceso con código.

1.6.2 ABRIR XML CON JAXB

Un documento XML con JAXB se abre utilizando las clases: J*AXBContext* y *Unmarshaller* que pertenecen al paquete *javax.xml.bind*.*.[17]

- La clase *JAXBContext* crea un contexto con una nueva instancia de JAXB para la clase principal obtenida del esquema XML.

- La clase *Unmarshaller* se utilizar para obtener del documento XML los datos que son almacenados en la estructura de objetos Java.

Para poder abrir un documento XML con estas clases es necesario que previamente esté creada la estructura de clases a partir de un esquema XML.

[17] Más información sobre este paquete es mostrada en:
http://docs.oracle.com/javaee/6/api/javax/xml/bind/package-summary.html

El siguiente código muestra cómo se carga un documento XML en una estructura de clases Java previamente obtenida del esquema.

```
public int abrir_XML_JAXB (File fichero, Libros misLibros) {
        JAXBContext contexto;
                try {
                        //Crea una instancia JAXB
                        contexto = JAXBContext.newInstance(Libros.class);
                        //Crea un objeto Unmarsheller.
                        Unmarshaller u=contexto.createUnmarshaller();
                        //Deserializa (unmarshal) el fichero
                        misLibros=(Libros) u.unmarshal(fichero);


                        return 0;

                } catch (Exception ex) {
                        ex.printStackTrace();
                        return -1;
                }
}
```

Siguiendo los comentarios incluidos en el código anterior es fácil entender que el proceso de creación de los objetos Java a partir del contenido de XML lo hace el método *unmarshal* (fichero), donde fichero es un *File* que contiene los datos del documento XML que se quiere abrir. El objeto misLibros de tipo *Libros* contendrá los libros obtenidos del fichero.

1.6.3 RECORRER UN XML DESDE JAXB

Recorrer un XML desde JAXB se reduce a trabajar con los objetos Java que representan al esquema XML del documento. Una vez que el XML es cargado en la estructura de objetos (*unmarshalling*) solo hay que navegar por ellos para obtener los datos que se necesiten.

El siguiente método muestra un ejemplo.

```
public String recorrerJAXByMostrar (Libros misLibros) {
        String datos_nodo[]=null;
        String cadena_resultado="";

        //Crea una lista con objetos de tipo libro.
        List<Libros.Libro> lLibros=misLibros.getLibro();

        //Recorre la lista para sacar los valores.
        For (int i=0; i<lLibros.size(); i++) {
           cadena_resultado= cadena_resultado + "\n " +"Publicado en: " + lLibros.get(i).getPublicadoEn();
           cadena_resultado= cadena_resultado + "\n " +"El Título es: " + lLibros.get(i).getTitulo();
           cadena_resultado= cadena_resultado + "\n " +"El Autor es: " + lLibros.get(i).getAutor();
           cadena_resultado = cadena_resultado +"\n ----------------------------";
        }
        return cadena_resultado;
}
```

Como se puede observar en el código, no se usa ninguna clase o método que no sea el propio manejo de los objetos Java *Libros* y *Libro*. El resultado de ejecutar este método es una salida similar a la mostrada con los ejemplos anteriores de DOM y SAX.

```
Publicado en: 1840
El Título es: El Capote
El Autor es: Nikolai Gogol
---------------------------
Publicado en: 2008
El Título es: El Sanador de Caballos
El Autor es: Gonzalo Giner
---------------------------
Publicado en: 1981
El Título es: El Nombre de la Rosa
El Autor es: Umberto Eco
---------------------------
```

ACTIVIDADES 1.4

- Sobre el código disponible en la carpeta *AccesoDOM* que contiene un proyecto hecho en NetBeans 7.1.2:

a. Modifica el esquema XML llamado LibrosEsquema.xsd para que permita un nuevo elemento <editorial>. Comprueba que un documento XML con un nuevo elemento editorial es válido para ese nuevo esquema creado.[18]

b. Crea las clases JAXB asociadas con ese esquema nuevo creado.

c. Modifica la aplicación y comprueba que funciona mostrando todos los elementos de un libro (incluido la editorial).

[18] La validación de documentos XML no se trata en el capítulo. Sin embargo, se supone que el lector tiene conocimientos de XML y herramientas para manejarlos (lenguajes de marcas). Una herramienta de validación es NetBeans 7.1.2, que comprueba si un esquema incluido en un proyecto es correcto, y si un documento XML, también incluido en un proyecto, es válido con respecto a un esquema dado.

1.7 PROCESAMIENTO DE XML: XPATH (XML PATH LANGUAGE)

La alternativa más sencilla para consultar información dentro de un documento XML es mediante el uso de XPath (XML Path Language), una especificación de la W3C para la consulta de XML. Con XPath se puede seleccionar y hacer referencia a texto, elementos, atributos y cualquier otra información contenida dentro de un fichero XML. En 1999, W3C publicó la primera recomendación de XPath (XPath 1.0), y en 2010 se publicó la segunda recomendación XPath 2.0.[19]

En la Sección 5.8.1 se muestra XQuery como el lenguaje más destacado y potente actualmente para la consulta de XML en bases de datos XML nativas. XQuery está basado en XPath 2.0, por tanto, es necesario conocer las nociones básicas de XPath para entender el funcionamiento de XQuery.

1.7.1 LO BÁSICO DE XPATH

XPath comienza con la noción contexto actual. El contexto actual define el conjunto de nodos sobre los cuales se consultará con expresiones XPath. En general, existen cuatro alternativas para determinar el contexto actual para una consulta XPath. Estas alternativas son las siguientes:

- ✓ (./) usa el nodo en el que se encuentra actualmente como contexto actual.
- ✓ (/) usa la raíz del documento XML como contexto actual.
- ✓ (.//) usa la jerarquía completa del documento XML desde el nodo actual como contexto actual.
- ✓ (//) usa el documento completo como contexto actual.

La mejor forma de dar los primeros pasos con XPath es mediante ejemplos. Para las siguientes explicaciones se parte del documento mostrado en la Figura 1.8.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Figura 1.8. XML Libros

[19] Se puede encontrar más información sobre XPath 2.0 en http://www.w3.org/TR/xpath20/

Para seleccionar elementos de un XML se realizan avances hacia abajo dentro de la jerarquía del documento. Por ejemplo, la siguiente expresión XPath selecciona todos los elementos Autor del documento XML Libros.

/Libros/Libro/Autor

Si se desea obtener todos los elementos Autor del documento se puede usar la siguiente expresión:

//Autor

De esta forma no es necesario dar la trayectoria completa.

También se pueden usar comodines en cualquier nivel del árbol. Así, por ejemplo, la siguiente expresión selecciona todos los nodos Autor que son nietos de Libros:

/Libros/*/Autor

Las expresiones XPath seleccionan un conjunto de elementos, no un elemento simple. Por supuesto, el conjunto puede tener un único miembro, o no tener miembros.

Para identificar un conjunto de atributos se usa el carácter @ delante del nombre del atributo. Por ejemplo, la siguiente expresión selecciona todos los atributos publicado_en de un elemento Libro:

/Libros/Libro/@publicado_en

Ya que en el documento de la Figura 1.8 solo los elementos Libro tienen un atributo publicado_en, la misma consulta se puede hacer con la siguiente expresión:

//@publicado_en

También se pueden seleccionar múltiples atributos con el operador @*. Para seleccionar todos los atributos del elemento Libro en cualquier lugar del documento se usa la siguiente expresión:

//Libro/@*

Además, XPath ofrece la posibilidad de hacer predicados para concretar los nodos deseados dentro del árbol XML. Esta es una posibilidad similar a la cláusula where de SQL. Así, por ejemplo, para encontrar todos los nodos Titulo con el valor El Capote se puede usar la siguiente expresión:

/Libros/Libro/Titulo[.='El Capote']

Aquí, el operador "[]" especifica un filtro y el operador "." establece que ese filtro sea aplicado sobre el nodo actual que ha dado la selección previa (/Libros/Libro/Titulo). Los filtros son siempre evaluados con respecto al contexto actual. Alternativamente, se pueden encontrar todos los elementos Libro con Título 'El Capote':

/Libros/Libro[./Titulo='El Capote']

De la misma forma se pueden filtrar atributos y usar operaciones booleanas en los filtros. Por ejemplo, para encontrar todos los libros que fueron publicados después de 1900 se puede usar la siguiente expresión:

/Libros/Libro[./@publicado_en>1900]

En el material adicional incluido en este libro se puede encontrar la carpeta Acceso_XPath que contiene un proyecto hecho en NetBeans 7.1.2. Este proyecto es una aplicación que ejecuta consultas XPath sobre el documento mostrado en la Figura 1.8. Se puede usar ese entorno para probar las consultas anteriores (aunque teniendo cuidado al escribir las comillas simples que tienen algunas consultas). Hay que tener presente que las dos últimas consultas no devolverán nada ya que es necesario modificar el código para que incluya la posibilidad de devolver elementos Libro. Esta modificación se propone para su realización en la Actividad 1.5.

1.7.2 XPATH DESDE JAVA

En Java existen librerías que permiten la ejecución de consultas XPath sobre documentos XML. En esta sección se muestra un ejemplo de cómo se puede abrir un documento XML y ejecutar consultas XPath sobre él usando DOM. Las clases necesarias para ejecutar consultas XPath son:

- ✓ *XPathFactory*[20]*, disponible en el paquete javax.xml.xpath.*: esta clase contiene un método* compile(), que comprueba si la sintaxis de una consulta XPath es correcta y crea una expresión XPath (XPathExpression).
- ✓ *XPathExpression*[21], disponible también en el paquete javax.xml.xpath.*: esta clase contiene un método evaluate() que ejecuta un XPath.
- ✓ *DocumentBuilderFactory*, disponible en el paquete javax.xml.parsers.*, y Document del paquete org.w3c.xml.*. Ambas clases han sido trabajadas con DOM en la Sección 1.4.2.

El siguiente código comentado muestra un ejemplo de cómo abrir un documento XML con DOM para ejecutar sobre él una consulta (/Libros/*/Autor).

```
public int EjecutaXPath(){
    try {
    // Crea el objeto XPathFactory
    xpath = XPathFactory.newInstance().newXPath();

    // Crea un objeto DocumentBuilderFactory para el DOM (JAXP)
    DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();

    // Crear un árbol DOM (parsear) con el archive LibrosXML.xml
    Document XMLDoc = factory.newDocumentBuilder().parse(new InputSource(new
    FileInputStream ("LibrosXML.xml")));

    // Crea un XPathExpression con la consulta deseada
    exp = xpath.compile ("/Libros/*/Autor");

    // Ejecuta la consulta indicando que se ejecute sobre el DOM y que devolverá
    // el resultado como una lista de nodos.
    Object result= exp.evaluate (XMLDoc, XPathConstants.NODESET);
    NodeList nodeList = (NodeList) result;
```

[20] Más información sobre XPathFactory es mostrada en:
http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/XPathFactory.html

[21] Más información sobre XPathExpression es mostrada en:
http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/XPathExpres-sion.html

```
//Ahora recorre la lista para sacar los resultados
        for (int i = 0; i < nodeList.getLength(); i++) {
         salida = salida + "\n" + nodeList.item(i).getChildNodes().item(0).getNodeValue();
         }
System.out.println(salida);
return 0;
} catch (Exception ex) {
System.out.println("Error: " + ex.toString());
return -1;
}
}
```

ACTIVIDADES 1.5

- Sobre el código disponible en la carpeta Acceso_XPath, que contiene un proyecto hecho en NetBeans 7.1.2, se propone:

a. Modificar el código para que se puedan ejecutar consultas que devuelvan objetos de tipo Libro, como por ejemplo: /Libros/Libro.

1.8 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se han mostrado diferentes formas de acceso a ficheros. Lejos de pretender profundizar en todas las posibilidades de cada acceso, lo que se ha buscado ha sido dar una visión global sobre el tratamiento de ficheros con Java.

El lector interesado en profundizar en las tecnologías expuestas en el capítulo puede hacerlo en las siguientes líneas de trabajo.

1. Trabajar más en profundidad los flujos para el tratamiento de archivos en Java. Para ello, el título Java 2. Curso de programación de Fco. Javier Ceballos, de la editorial RA-MA, es una buena referencia.
2. Conocer otros modos de acceso a documentos XML, como jDOM22 que ofrece un modelo más natural para trabajar con XML desde Java que el ofrecido por DOM. Es una alternativa diferente al DOM de W3C visto en el capítulo, y muy aceptada en el terreno profesional.

En cualquier caso, un amplio conocimiento en el manejo de ficheros y en el acceso a XML, junto con todo lo que XML ofrece (esquemas XML, herramientas de validación de documentos, XPath, etc.) es necesario para desarrollar aplicaciones de acceso a datos solventes, así como para entender parte de los capítulos siguientes.

[22] http://www.jdom.org/

RESUMEN DEL CAPÍTULO

En este capítulo se ha abordado el acceso a ficheros como técnica básica para hacer persistentes datos de una aplicación. El capítulo tiene dos partes bien diferenciadas.

La primera parte abarca el acceso a ficheros con las clases que ofrece java.io. Esta alternativa es la de más bajo nivel de todas las soluciones que se dan en este y en el resto de capítulos para el acceso a datos. Sin embargo, sí es cierto que conocer bien el acceso a fichero (tipos y modos) es obligado cuando se trabaja con persistencia.

La segunda parte abarca el acceso a un tipo concreto de fichero de texto secuencial: XML. Que el lector conozca y maneje las diferentes alternativas de acceso a ficheros XML desde Java ha sido el objetivo perseguido. Los conocimientos en esta tecnología serán muy útiles para entender algunos de los siguientes capítulos de este libro, sobre todo lo relacionado con el marshalling, XPath y DOM

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone una aplicación de gestión de una librería. Los pasos que la describen son:

- 1. Crear un esquema XML para soportar los libros de una librería. Los campos que debe tener son título, autor, ISBN, número de ejemplares, edito-rial, número de páginas, año de edición. El diseñador del esquema puede libremente elegir cuáles de esos campos son atributos XML o elementos XML. Una vez creado el esquema hay que crear un documento XML válido para el esquema.
- 2. La aplicación debe permitir mostrar todo el documento XML creado anteriormente (usando SAX).
- 3. Crear una estructura de objetos con JAXB. Para ello debe usar el esquema creado en el ejercicio 1.
- 4. La aplicación debe permitir modificar el título de un libro. El usuario proporciona el ISBN del libro que se quiere modificar y el nuevo título. Hay que utilizar la estructura de objetos creada en el ejercicio 3 para cargar con JAXB el documento XML con los libros de la librería, y sobre esos objetos hacer las modificaciones.
- 5. La aplicación debe permitir guardar la estructura de objetos JAXB en un fichero XML (marshalling-serialización).
- 6. La aplicación debe permitir al usuario consultar los libros de la librería usando XPath (y DOM).

TEST DE CONOCIMIENTOS

1. Entre DOM y SAX es verdadero que:

   a) DOM carga el documento en memoria principal al igual que SAX.
   b) SAX es más recomendable que DOM con ficheros pequeños.
   c) DOM es adecuado para ficheros pequeños que requieran modificación.

2. Sobre la clase File:

   a) Permite abrir ficheros en modo aleatorio.
   b) Permite flujo de caracteres, pero no binario.
   c) Representa un fichero o directorio y tiene métodos para conocer sus características.

3. Unmarshalling con XML es:

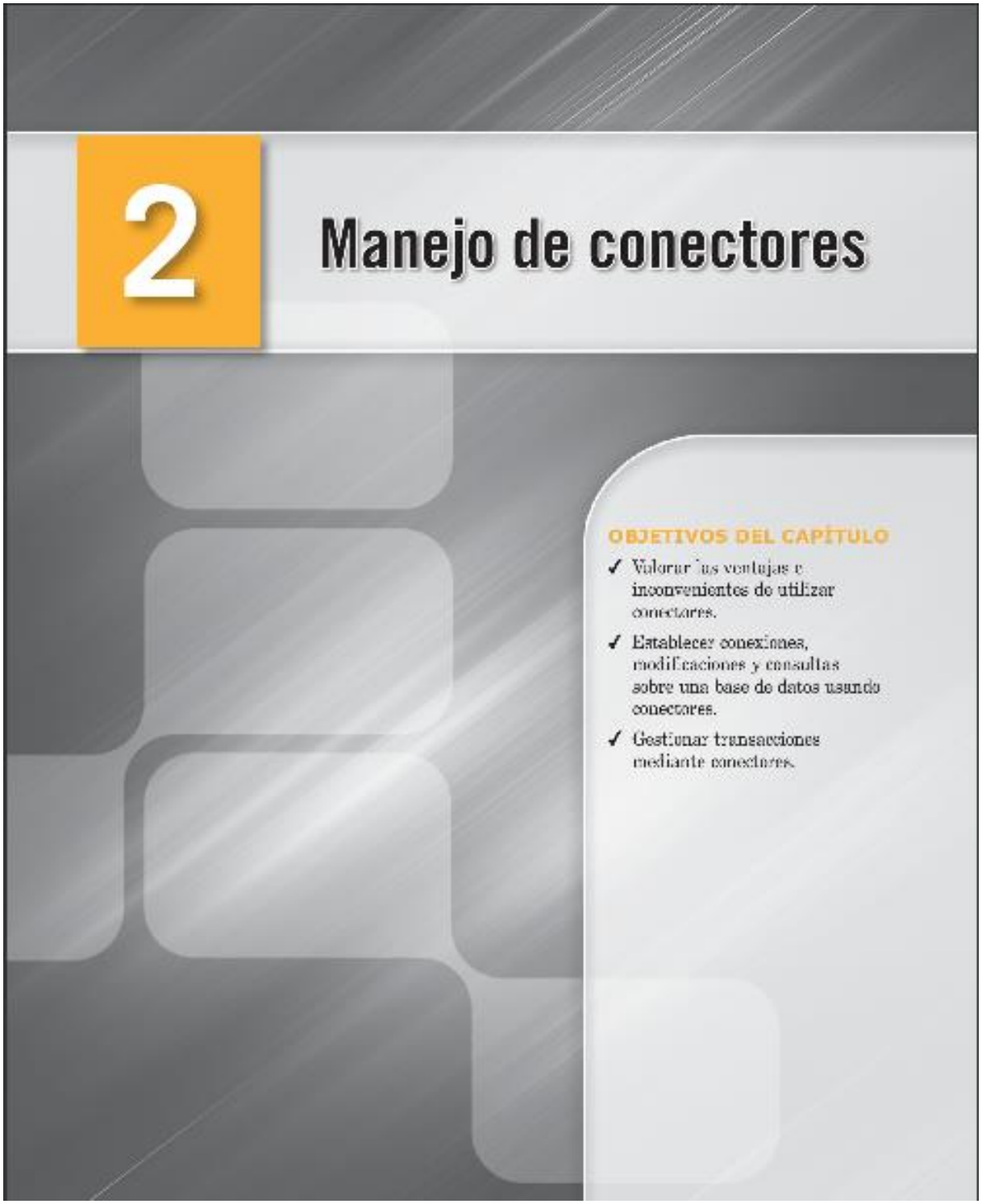
   a) El proceso por el cual se puede consultar documentos XML
   b) Crear a partir de un fichero XML una estructura de objetos que contenga sus datos.
   c) Crear un fichero XML desde una estructura de objetos previamente creada.

4. De SAX es cierto que:

   a) Es un acceso a XML que permite la modificación de los contenidos.
   b) Es un acceso a XML aleatorio que permite navegar por cualquier parte del documento.
   c) Es un acceso secuencial que no permite modificaciones.

5. De SAX, el manejador es una clase que:

   a) Extiende DefaultHandler y sirve para redefinir los métodos (callbacks) que atienden los eventos.
   b) Se utiliza para recorrer el árbol de resultados.
   c) Crea una estructura de objetos para manejar el contenido del XML y así poder modificarlo.



2 Manejo de conectores

Objetivos del Capítulo

- ✓ Valorar las ventajas e inconvenientes de utilizar conectores.
- ✓ Establecer conexiones, modificaciones y consultas sobre una base de datos usando conectores.
- ✓ Gestionar transacciones mediante conectores.

Se llama conector al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Desde los lenguajes propios de los sistemas gestores de bases de datos se pueden gestionar los datos mediante lenguajes de consulta y manipulación propios de esos sistemas. Sin embargo, cuando se quiere acceder a los datos desde lenguajes de programación de una misma manera con independencia del sistema gestor que contenga los datos, entonces es necesario utilizar conectores que faciliten estas operaciones. Los conectores dan al programador una manera homogénea de acceder a cualquier sistema gestor (preferiblemente relacional u objeto-relacional).

En este capítulo se hace una primera introducción a los conceptos básicos que hay detrás de los conectores. Seguidamente se muestran ejemplos sobre cómo realizar las operaciones básicas sobre una base de datos MySQL usando conectores con Java.

2.1 EL DESFASE OBJETO-RELACIONAL

El problema del desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la programación orientada a objetos, con la que se desarrollan aplicaciones, y la base de datos, con las que se almacena la información. Estos aspectos se pueden presentar relacionados cuando:

- ✓ Se realizan actividades de programación, donde el programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.

- ✓ Se especifican los tipos de datos. En las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, suelen ser tipos simples, mientras que en la programación orientada a objetos se utilizan tipos de datos complejos.

- ✓ En el proceso de elaboración del software se realiza una traducción del modelo orientado a objetos al modelo entidad-relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que el desarrollador tenga que diseñar dos diagramas diferentes para el diseño de la aplicación.

La discrepancia objeto-relacional surge porque en el modelo relacional se trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, en el modelo de programación orientada a objetos se trabaja con objetos y las asociaciones entre ellos. Al problema se le denomina desfase objeto-relacional, o sea, el conjunto de dificultades técnicas que aparecen cuando una base de datos relacional se usa conjuntamente con un programa escrito con lenguajes de programación orientada a objetos.

En el Capítulo 3 se volverá a hacer hincapié sobre esta idea, destacando los problemas del desfase objeto-relacional en programación y mostrando la solución que son los sistemas gestores orientados a objetos (SGBDOO) para solventar este problema.

2.2 PROTOCOLOS DE ACCESO A BASES DE DATOS: CONECTORES

Muchos servidores de bases de datos utilizan protocolos de comunicación específicos que facilitan el acceso a los mismos, lo que obliga a aprender un lenguaje nuevo para trabajar con cada uno de ellos. Es posible reducir esa diversidad de protocolos mediante alguna interfaz de alto nivel que ofrezca al programador una serie de métodos para acceder a la base de datos (véase la Figura 2.1).

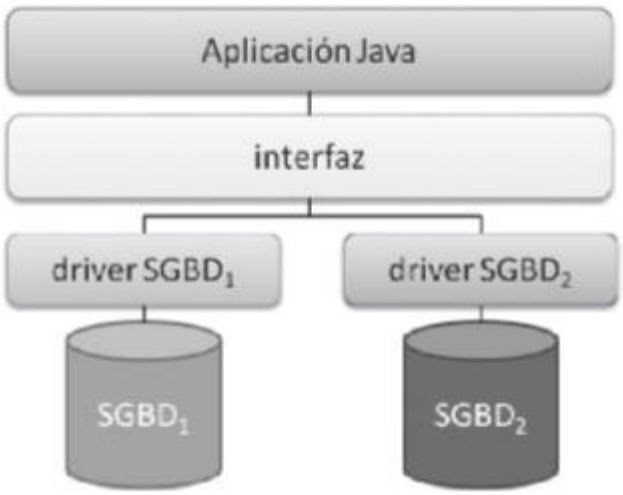


Figura 2.1. Estructura general de acceso a distintas bases de datos desde Java

Estas interfaces de alto nivel ofrecen facilidades para:

- ✓ Establecer una conexión a una base de datos.
- ✓ Ejecutar consultas sobre una base de datos.
- ✓ Procesar los resultados de las consultas realizadas.

Las tecnologías disponibles, aunque muy diversas, abstraen la complejidad subyacente de cada producto y proporcionan una interfaz común, basada en el lenguaje de consulta estructurado (SQL), para el acceso homogéneo a los datos. Algunos ejemplos representativos son JDBC (Java DataBase Connectivity) de Sun y ODBC (Open DataBase Connectivity) de Microsoft.

Al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos se le denomina conector o driver. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Cuando se construye una aplicación de base de datos, el conector oculta los detalles específicos de cada base de datos, de modo que el programador solo debe preocuparse de los aspectos relacionados con su aplicación, olvidándose de otras consideraciones. La mayoría de los fabricantes ofrecen conectores para acceder a sus bases de datos.

Un ejemplo de conector muy extendido es el mencionado con anterioridad, el conector JDBC. Este conector es una capa software intermedia (véase la Figura 2.2) situada entre los programas Java y los sistemas de gestión de bases de datos relacionales que utilizan SQL. Dicha capa es independiente de la plataforma y del gestor de bases de datos utilizado.

Con el conector JDBC no hay que escribir un programa para acceder, por ejemplo, a una base de datos Access y otro programa distinto para acceder a una base de datos Oracle, etc., sino que se puede escribir un único programa utilizando el API JDBC, y es ese programa el que se encarga de enviar las consultas a la base de datos utilizada en cada caso.

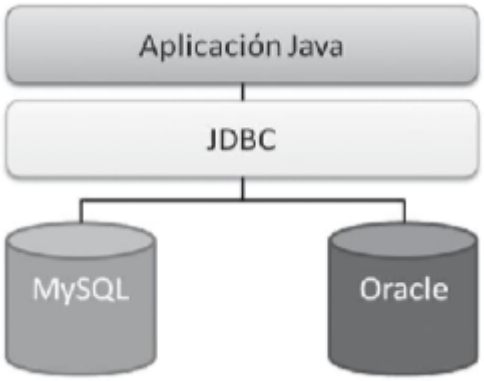


Figura 2.2. Localización de un conector JDBC en el acceso a bases de datos

Otro ejemplo de conector es el conector de Microsoft ODBC. La diferencia entre JDBC y ODBC está en que ODBC tiene una interfaz C. En este sentido, ODBC es simplemente otra opción respecto a JDBC, ya que la mayoría de sistemas gestores de bases de datos disponen de drivers para trabajar con ODBC y JDBC. Además, también existe en Java un driver JDBC-ODBC, para convertir llamadas JDBC a ODBC y poder acceder a bases de datos que ya tienen un driver ODBC y todavía no tienen un conector JDBC.[23]

Seguidamente, y para ser coherentes con el resto de contenidos de este libro, el capítulo se centrará en el conector JDBC, sus componentes y principales características.

2.2.1 COMPONENTES JDBC

El conector JDBC incluye cuatro componentes principales:

- La propia API JDBC, que facilita el acceso desde el lenguaje de programación Java a bases de datos relacionales y permite que se puedan ejecutar sentencias de consulta en la base de datos. Dicha API está disponible en los paquetes java.sql y javax.sql, Java Standard Edition (Java SE) / Java Enterprise Edition (Java EE) respectivamente.
- El gestor del conector JDBC (drivermanager), que conecta una aplicación Java con el driver correcto de JDBC. Se puede realizar por conexión directa (DriverManager) o a través de un pool de conexiones, vía DataSource.
- La suite de pruebas JDBC, encargada de comprobar si un conector (driver)24 cumple con los requisitos JDBC.
- El driver o puente JDBC-ODBC, que permite que se puedan utilizar los drivers ODBC como si fueran de tipo JDBC.

[23] Esta solución, aunque muy versátil, es la que peor rendimiento ofrece, ya que obliga a varias conversiones entre API.

[24] A lo largo del capítulo, conector o driver se usarán como sinónimos indistintamente.

2.2.2 TIPOS DE CONECTORES JDBC

En función de los componentes anteriores, en un conector JDBC existen cuatro tipos de controladores JDBC. La denominación de estos controladores está asociada a un número de 1 a 4 y viene determinada por el grado de independencia respecto de la plataforma, prestaciones, etc. Seguidamente se muestran cada uno de estos tipos de conectores JDBC.

Driver tipo 1: utilizan una API nativa estándar, donde se traducen las llamadas de JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo (Figura 2.3).

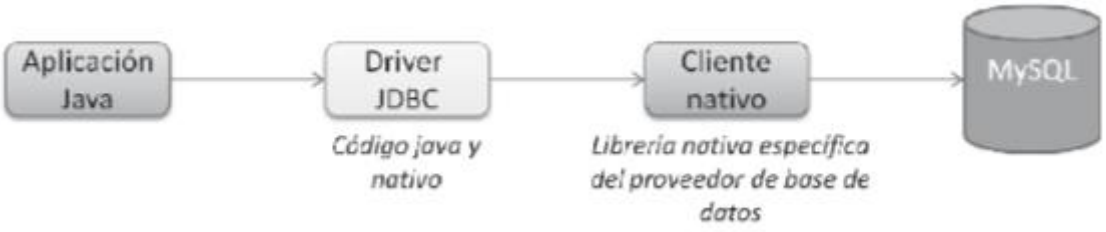


Figura 2.3. Driver tipo 1

Driver tipo 2: utilizan una API nativa de la base de datos, es decir son drivers escritos parte en Java y parte en código nativo. El driver usa una librería cliente nativa, específica de la base de datos con la que se desea conectar. No es un driver 100 % Java. La aplicación Java hace una llamada a la base de datos a través del driver JDBC y este traduce la petición a invocaciones a la API del fabricante de la base de datos (Figura 2.4).

Figura 2.4. Driver tipo 2

Driver tipo 3: utilizan un servidor remoto con una API genérica, es decir son drivers que usan un cliente Java puro que se comunica con un middleware server usando un protocolo independiente de la base de datos (por ejemplo, TCP/IP). Este tipo de drivers convierte las llamadas en un protocolo que puede utilizarse para interactuar con la base de datos (Figura 2.5).

Figura 2.5. Driver tipo 3

Driver tipo 4: es el método más eficiente de acceso a base de datos. Este tipo de drivers son suministrados por el fabricante de la base de datos y su finalidad es convertir llamadas JDBC en un protocolo de red comprendido por la base de datos. Este tipo de driver es el que se trabajará en los ejemplos incluidos en este capítulo (Figura 2.6).

Figura 2.6. Driver tipo 4

2.2.3 MODELOS DE ACCESO A BASES DE DATOS

A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple.

En el modelo de dos capas, la aplicación que accede a la base de datos reside en el mismo lugar que el driver de la base de datos. Sin embargo, la base de datos puede estar en otra máquina distinta, con lo que el cliente se comunica por red. Esta es la configuración llamada cliente-servidor, y en ella toda la comunicación a través de la red con la base de datos será manejada por el conector de forma transparente a la aplicación Java. Este modelo de acceso se muestra gráficamente en la Figura 2.7.

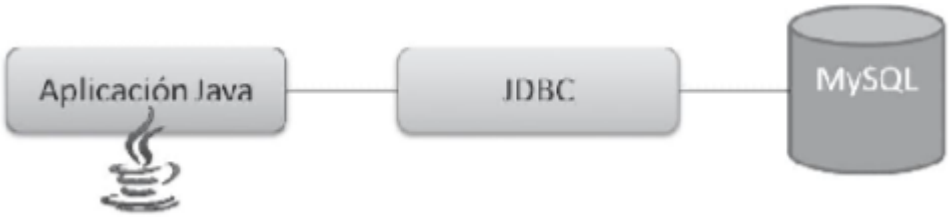


Figura 2.7. Modelo de acceso a base de datos cliente-servidor (dos capas)

Alternativamente al modelo de dos capas, en el modelo de tres capas los comandos se envían a la capa intermedia de servicios, que envía las consultas a la base de datos (Figura 2.8). Esta las procesa y envía los resultados de vuelta a la capa intermedia, para que más tarde sean enviados al cliente. En este modelo una aplicación o applet de Java se está ejecutando en una máquina y accediendo a un driver de base de datos situado en otra máquina. Ejemplos de puesta en práctica de este modelo de acceso se dan en los siguientes casos:

- Cuando se tiene un applet accediendo al driver a través de un servidor web.
- Cuando una aplicación accede a un servidor remoto que comunica localmente con el driver.
- Cuando una aplicación, qué está en comunicación con un servidor de aplicaciones, accede a la base de datos por nosotros.

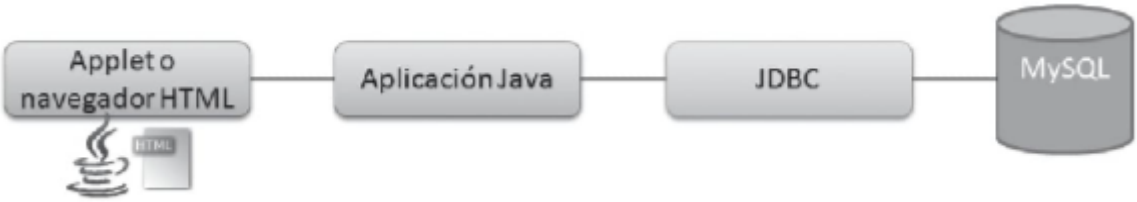


Figura 2.8. Modelo de acceso basado en la existencia de una capa intermedia de servicio (tres capas)

2.2.4 ACCESO A BASES DE DATOS MEDIANTE UN CONECTOR JDBC

Las dos ventajas que ofrece JDBC pasan por proveer una interfaz para acceder a distintos motores de base de datos y por definir una arquitectura estándar con la que los fabricantes puedan crear conectores que permitan a las aplicaciones Java acceder a los datos. Este apartado se centra en la primera de esas ventajas.

A continuación se muestra cómo acceder a una base de datos utilizando JDBC. Lo primero que se debe hacer para poder realizar consultas en una base de datos es, obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas ellas, así que se optará por una en concreto. La elegida es una base de datos MySQL (véase el esquema mostrado en la Figura 2.9). Se ha elegido este gestor de bases de datos porque es gratuito y por funcionar en diferentes plataformas.

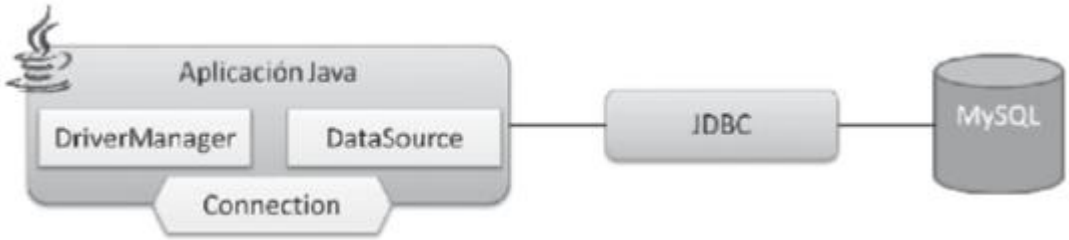


Figura 2.9. Estructura general de acceso a una base de datos MySQL desde una aplicación Java

Para acceder a MySQL con JDBC se deben seguir los pasos que se detallan a continuación (véase la Figura 2.10):

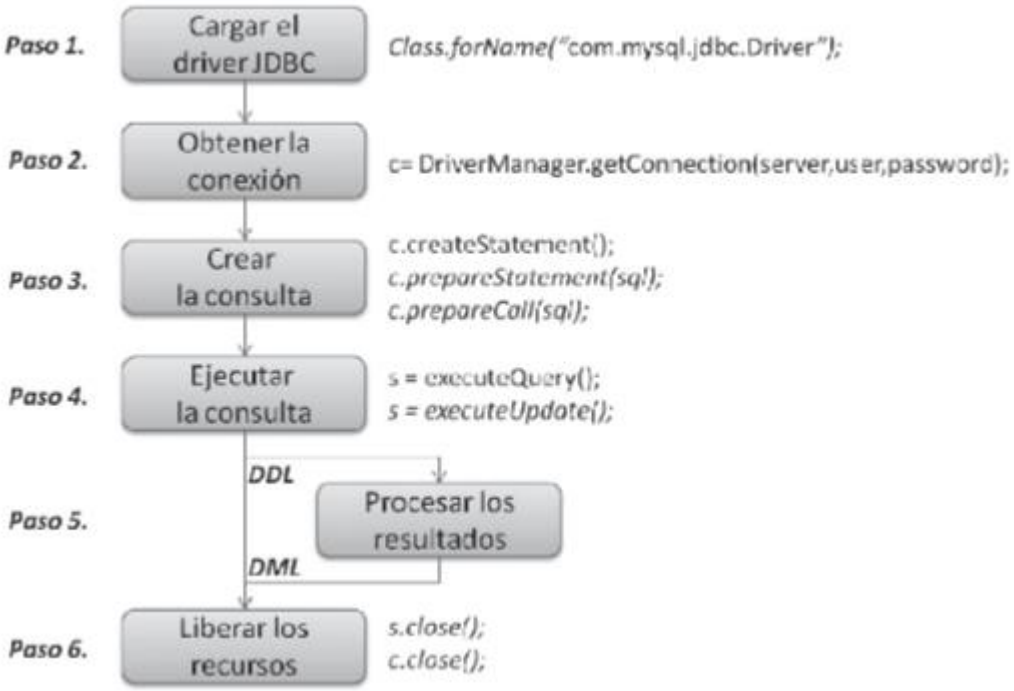


Figura 2.10. Pasos para acceder a una base de datos MySQL

Antes de cargar el driver JDBC y obtener una conexión con la base de datos se deben hacer dos pasos:

Lo primero que se necesita para conectar a una base de datos es un objeto conector. Ese conector es el que sabe cómo interactuar con la base de datos. El lenguaje Java no viene con todos los conectores de todas las posibles bases de datos del mercado. Por tanto, se debe recurrir a Internet para obtener el conector que se necesite en cada caso.

En los ejemplos de este capítulo, se necesitará el conector de MySQL.25 Una vez descargado el fichero mysql-connector-java-5.1.xx.zip, se descomprime y se localiza el fichero mysql-connector-java-5.1.21-bin.jar (donde xx hace referencia a la versión más actual del mismo), que viene incluido en el fichero .zip. En ese otro archivo un fichero con extensión .jar ofrece la clase conector que nos interesa.

Para incluir el fichero mysql-connector-java-5.1.21-bin.jar en cada proyecto se deberán seguir los siguientes pasos:

1. En la carpeta raíz del proyecto, crear la carpeta /lib.
2. Copiar el fichero mysql-connector-java-5.1.21-bin.jar en la carpeta /lib que se acaba de crear.
3. Desde NetBeansIDE 7.1.2,26 pulsar con el botón derecho del ratón en el nombre del proyecto y seleccionar la opción de menú propiedades (Properties).
4. En el árbol lateral pulsar en Libraries.
5. En el botón de la izquierda pulsar en Add JAR/Folder.
6. Seleccionar el fichero mysql-connector-java-5.1.21-bin.jar que se encuentra en la carpeta /lib del proyecto y pulsar Abrir.
7. Pulsar OK. La Figura 2.11 muestra cómo quedarían las propiedades del proyecto con la librería mysql-connector-java-5.1.21-bin.jar incorporada en tiempo de compilación.

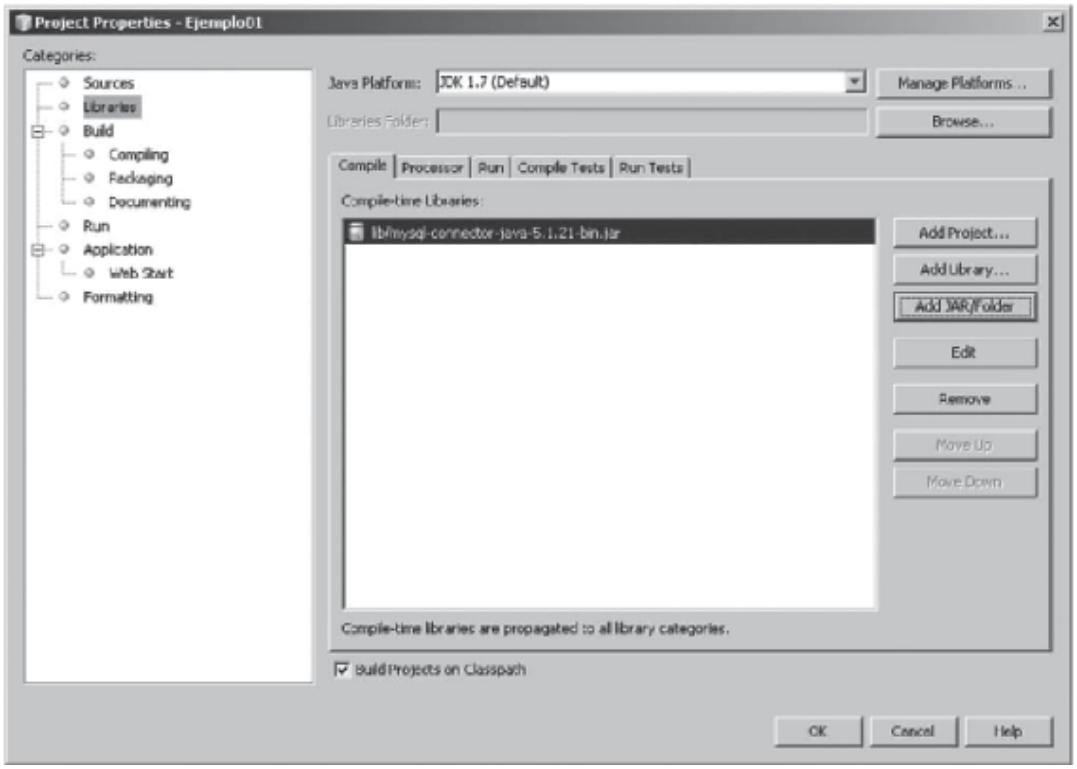


Figura 2.11. Librería añadida al proyecto

[25] http://dev.mysql.com/downloads/connector/j/

[26] Netbeans IDE 7.1.2 es la versión usada en los proyectos de este capítulo.

La segunda acción para poder utilizar el conector sin problemas es localizarlo en el sistema operativo identificando su ruta en la variable de entorno CLASSPATH, siempre que nuestro IDE (Eclipse, Netbeans, etc.) utilice esa variable. Desde consola el comando para lograr este propósito sería el siguiente:

```
$ set CLASSPATH-<PATH_DEL_JAR>\ mysql-conector-java-5.1.21-bin-jar
```

Una vez localizado el driver conector en el sistema será posible cargarlo desde cualquier aplicación Java (Figura 2.10, pasos 1 y 2)

Un ejemplo es el siguiente código. Lo que hace es acceder a una base de datos llamada discográfica que se ha creado en MYSQL Esta base de datos tiene una tabla ALBUMES. Se han creado en ALBUMES tres campos: 1D, clave primaria tipo numérico, TITULO, tipo VARCHAR (30), y AUTOR, tipo VARCHAR (30).

```
public Gestor_conexion() {        //Constructor
        // Crea una conexión
        Connection conn1 = null;
try {
        String url1 = "jdbc:mysql://localhost:3306/discografica";
        String user = "root";
        String password = "";    //no tiene clave
        Conn1 = DriverManager.getConnection (url1, user, password);
        if (conn1 = null) {
                System.out.println("Conectado a discográfica…");
        }

} catch (SQLException ex) {
        System.out.println("Error: dirección no válida o usuario/clave");
        ex.printStackTrace();
}
```

El procedimiento de conexión con el controlador de la base de datos, independientemente de la arquitectura, es siempre muy similar. En primer lugar se carga el conector. Cualquier *driver* JDBC, independientemente de su tipo, debe implementar la interfaz *java.sql.Driver*. La carga de *driver* se realizaba originalmente con *Class.forName(Driver).* Sin embargo al poner el *jar* del driver (MySQL en nuestro caso) en la carpeta *lib* del proyecto (o en el *classpath* del programa), cuando la clase *DriverManager* se inicializa, busca esta propiedad en el sistema y detecta que se necesita el *driver* elegido.

Una vez cargado el *driver*, el programador puede crear una conexión (paso 2 en la Figura 2.10). El objetivo es conseguir un objeto del tipo *java.sql.Connection* a través del método *DriverManagerConnection* (*String url*). En el código anterior se muestra el uso de este método.

La línea **url1="jdbc:mysql://localhost:1306/discografica";** indica que sucede acceder a una base de datos MYSQL mediante JDBC, que la base de datos está localizable en el localhost (127.0.0.1) por el puerta 3306 y que su nombre en *discografica* (sin tilde).

Si todo va bien, cuando se ejecute la sentencia donde el objeto *DriverManager* invoca al método *getConnection()* se crea una conexión a una base de datos MySQL. Si esa invocación fuera mal, se informará de una excepción gracias al uso de las sentencias try-catch utilizadas (captura de excepciones).

No hay que olvidar que, después de usar una conexión, ésta debe ser cerrada con el método close() de Connection. El siguiente código muestra un ejemplo de cómo hacerlo.

```
public void cerrar_Conexion (Connection conn1) {
        try {
                conn1.close();
        } catch (SQLException ex) {
                System.out.println("ERROR: al cerrar la conexión");
                ex.printStackTrace();
        }
}
```

Pool de conexiones

La manera mostrada de obtener una conexión está bien para aplicaciones sencillas, donde únicamente se establece una conexión con la base de datos. Sin embargo, hay un pequeño problema con esta alternativa: varios hilos de ejecución no pueden usar una misma conexión física con la base de datos simultáneamente, ya que la información enviada o recibida por cada uno de los hilos de ejecución se entremezcla con la de los otros, haciendo imposible una escritura o lectura coherente en dicha conexión. Hay varias posibles soluciones para este problema:

- Abrir y cerrar una conexión cada vez que la necesitemos. De esta forma, cada hilo de ejecución tendrá la suya propia. Esta solución en principio no es eficiente, puesto que establecer una conexión real con la base de datos es un proceso costoso. El hecho de andar abriendo y cerrando conexiones con frecuencia puede hacer que el programa vaya más lento de lo debido.

- Usar una única conexión y sincronizar el acceso a ella desde los distintos hilos. Esta solución es más o menos eficiente, pero requiere cierta disciplina al programar, ya que es necesario poner siempre synchronized antes de hacer cualquier transacción con la base de datos. También tiene la pega de que los hilos deben esperar entre ellos.

- Finalmente, también existe la posibilidad de tener varias conexiones abiertas (pool de conexiones), de forma que cuando un hilo necesite una, la pida, y cuando termine, la deje para que pueda ser usada por los demás hilos, todo ello sin abrir y cerrar la conexión cada vez. De esta forma, si hay conexiones disponibles, un hilo no tiene que esperar a que otro acabe. Esta solución es en principio la ideal y es la que se conoce como pool de conexiones.

Apostando por el tercero de los escenarios anteriores, en Java, un pool de conexiones es una clase que tiene abiertas varias conexiones a bases de datos. Cuando alguien necesita una conexión a base de datos, en vez de abrirla directamente con *DriverManager.getConnection(),* se pide al pool usando su método p*ool.getConnection().* El pool coge una de las conexiones que ya tiene abierta, la marca para saber que está asignada y la devuelve. La siguiente llamada a este método *pool.getConnection()* buscará una conexión libre para marcarla como ocupada y ofrecerla.

El código Java asociado a esta forma de trabajar es el siguiente:

```
public Connection crearConexion() {
        BasicDataSource bdSource = new BasicDataSource();
        bdSource.setUrl ("jdbc:mysql://localhost:3306/discografica");
        bdSource.setUsername("root");
        bdSource.setPassword("");
        Connection con = null;
        try {
                if (con != null) {
                        System.out.println("No se puede crear la conexión");
                } else {
                //DataSource reserva una conexión y la devuelve para ser usada
                con = bdSource.getConnection();
                System.out.println("Conexión creada ");
                }
        } catch (Exception e) {
                System.out.println("Error: " + e.toString());
        }
        return con;
}
```

Esta aplicación necesita de la librería *commons-dbcp-all-1.3.jar*. Para usar esta librería se necesita seguir el mismo proceso que el mostrado para el driver MySQL, es decir, incluir el *commons-dbcp-all-1.3.jar* en la carpeta */lib* del proyecto, y seguidamente añadirla a la librería de NetBeans.[27]

En la librería *commons-dbcp-all-1.3.*jar se dispone de una implementación sencilla de un pool de conexiones, ya que al ser *DataSource* una interfaz se debe facilitar una implementación para poder instanciar objetos de esa clase. Esta librería necesita a su vez la librería *commons-pool*, por lo que también es necesario descargarla si se quiere usar este pool. Una vez configurado, para usar *BasicDataSource* solo es necesario hacer un new de esa clase y pasarle los parámetros adecuados de nuestra conexión con los métodos *set()* disponibles para ello.

ACTIVIDADES 2.1

- Utilizar el contenido de la sección para configurar el entorno de desarrollo (IDE Netbeans si es posible) para incluir las librerías JDBC que dan acceso a MySQL.

PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado.

[27] En el código asociado a este proyecto está disponible la librería *apache commons-dbcp* completa, dentro de la cual se encuentra *commons-dbcp-1.4.jar*.

2.2.5 CLASES BÁSICAS DEL API CONECTOR JDBC

En la sección anterior se ha mostrado cómo hacer una conexión con una base de datos MySQL. Como se ha comentado anteriormente, la interfaz del conector JDBC reside en los paquetes *java.sql* y *javax.sql*. Lo que se ofrece en esos paquetes son en su mayoría interfaces, ya que la implementación específica de cada una de ellas es fijada por cada proveedor según su protocolo de bases de datos. En cualquier caso, en la interfaz hay distintos tipos de objetos que se deben tener presentes, por ejemplo: *Connection*. *Statement* y *ResultSet*. El resto de objetos necesarios se mostrarán en próximas secciones

Los objetos de la clase Connection[28] ofrecen un enlace activo a una base de datos a través del cual un programa en Java puede leer y escribir datos, así como explorar la estructura de la base de datos y sus capacidades. Se crea con una llamada a *DriverManager.getConnection()* o a *DataSource.getConnection()* (en JDBC 2.0). En la sección anterior se han mostrado ejemplos asociados donde se utilizan ambas llamadas.

La interfaz DriverManager[29], complementaria de la clase *Connection*, Con ella se registran los controladores JDBC y se proporcionan las conexiones que permiten manejar las URL específicas de JDBC. Se consigue con el método *getConnection()* de la propia clase.

La clase *Statement*[30] proporciona los métodos para que las sentencias, utilizando el lenguaje de consulta estructurado (SQL), sean ejecutadas sobre la base de datos y se pueda recuperar el resultado de su ejecución. Hay tres tipos de sentencias Statement[31], cada una especializa a la anterior. Estas sentencias se verán en las siguientes secciones.

Además de las clases anteriores, el API JDBC ofrece también la posibilidad de gestionar excepciones con la clase SQLException. Dicha clase es la base de las excepciones de JDBC. La mayor parte de las operaciones que proporciona el API JDBC lanzarán la excepción *java.sql.SQLException* en caso de que se produzca algún error en la base de datos (por ejemplo, errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc.). Por este motivo es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques *try/catch.*

ACTIVIDADES 2.2

- Crear en MySQL una base de datos de ejemplo para una discográfica: una tabla con canción (titulo (Varchar()), duración (Varchar()), letra (Varchar()) y álbum (id (Int), titulo (Varchar()), año en el que se publicó (Varchar()). La relación entre las tablas álbum y canción es uno a muchos: un álbum está compuesto por muchas canciones.

[28] Más información sobre esta clase se puede encontrar en la siguiente dirección: http://docs.oracle.com/javese/7/docs/api/java/sql/Connection.html

[29] Más información sobre esta clase se puede encontrar en la siguiente dirección: http://docs.oracle.com/javase/7/docs/api/java/sql/DriverManager.html

[30] Más información sobre esta clase se puede encontrar en la siguiente dirección: http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html

[31] Estos tres tipos serán detallados en las siguientes secciones.

2.2.6 CLASES ADICIONALES DEL API JDBC

Además de las clases básicas anteriores, el API JDBC también ofrece la posibilidad de acceder a los metadatos de una base de datos. Con ellos se puede obtener información sobre la estructura de la base de datos y, gracias a ello, se pueden desarrollar aplicaciones independientemente del esquema que tenga la base de datos. Las principales clases asociadas a metadatos de una base de datos son las siguientes: *DatabaseMetaData* y *ResultSetMetaData*.

- Los objetos de la clase *DatabaseMetaData* ofrecen la posibilidad de operar con la estructura y capacidades de la base de datos. Se instancian con *connection.getMetaData().* Los metadatos son datos acerca de los datos, es decir, datos que explican la naturaleza de otros datos. La interfaz *DatabaseMetaData* contiene más de 150 métodos para recuperar información de una base de datos (catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, procedimientos almacenados, vistas etc.), así como información sobre algunas características del controlador JDBC que se esté utilizando. Estos métodos son útiles cuando se implementan aplicaciones genéricas que pueden acceder a diversas bases de datos.

- Los objetos de la clase *ResultSetMetaData* son el *ResultSet* que se devuelve al hacer un *executeQuery()* de un objeto *DatabaseMetaData*. Los métodos de *ResultSetMetaData* permiten determinar las características de un objeto *ResultSet*. Por ejemplo, con un objeto de la clase *ResultSetMetaData* se puede determinar el número de columnas; información sobre una columna, tal como el tipo de datos o la longitud; la precisión y la posibilidad de contener nulos, e información sobre si una columna es de solo lectura, etc.

ACTIVIDADES 2.3

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que conecte con la base de datos.

PISTA

En el código asociado a este capítulo hay un proyecto llamado *accesoJDBC* para NetBeans 7.1.2 que contiene el código de ejemplo mostrado. Sin embargo, no tiene asociada la base de datos, que se creó con XAMPP[32] en local.

[32] http://www.apachefriends.org/es/xampp.html

2.3 EJECUCIÓN DE SENTENCIAS DE DEFINICIÓN DE DATOS

El lenguaje de definición de datos (*Data Definition Language* o *Data Description Language* [DDL] según autores) es la parte de SQL dedicada a la definición de una base de datos. Dicho lenguaje consta de sentencias para definir la estructura de la base de datos y permite definir gran parte del nivel interno de la misma. Por este motivo, estas sentencias serán utilizadas normalmente por el administrador de la base de datos.

Las principales sentencias asociadas con el lenguaje DDL son CREATE, ALTER y DROP. Siempre se usan estas sentencias junto con el tipo de objeto y el nombre del objeto. Dichas sentencias permiten:

- **CREATE** sirve para crear una base de datos o un objeto.
- **ALTER** sirve para modificar la estructura de una base de datos o de un objeto.
- **DROP** permite eliminar una base de datos o un objeto.

Para enviar comandos SQL a la base de datos con JDBC se usa un objeto de la clase Statement. Este objeto se obtiene a partir de una conexión a base de datos, de esta forma:

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

Statement tiene muchos métodos, pero hay dos especialmente interesantes: executeUpdate() y executeQuery().

- **executeUpdate(): se usa para sentencias SQL que impliquen modificaciones en la base de datos (INSERT, UPDATE, DELETE, etc.).**
- **executeQuery(): se usa para consultas (SELECT y similares).**

El siguiente ejemplo muestra cómo se modifica una tabla desde una aplicación Java:

```
// Añadir una nueva columna a una tabla ya existente
Statement sta = con.createStatement();
int count = sta.executeUpdate("ALTER TABLE contacto ADD edad");
```

Por último, el siguiente código elimina la tabla llamada contacto de una base de datos previamente abierta:

```
st.executeUpdate("DROP TABLE contacto");
```

2.4 EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

Las sentencias de manipulación de datos (*Data Manipulation Language* [DML]) son las utilizadas para insertar, borrar, modificar y consultar los datos que hay en una base de datos. Las sentencias DML son las siguientes:

La sentencia **SELECT** sirve para recuperar información de una base de datos y permite la selección de una o más filas y columnas de una o muchas tablas.

- La sentencia **INSERT** se utiliza para agregar registros a una tabla.
- La sentencia **UPDATE** permite modificar la información de las tablas.
- La sentencia **DELETE** permite eliminar una o más filas de una tabla.

El siguiente ejemplo muestra un método para insertar valores a la tabla álbum creada en la Actividad 2.2.

```
public void Insertar() {
        try {
                // Crea un statement
                Statement sta = conn1.createStatement();
                // Ejecuta la inserción
                sta.executeUpdate("INSERT INTO album " + "VALUES (3, 'Black Album', 'Metallica')");
                // Cierra el statement
                sta.close();
                } catch (SQLException ex) {
                        System.out.println("ERROR: al hacer un Insert");
                        ex.printStackTrace();
                }
}
```

En el ejemplo, partiendo de una conexión previa (*conn1*) se crea un objeto *Statement* llamado *sta*. Sobre ese objeto se ejecuta una consulta *Insert into.SQLException*, que se encarga de capturar los errores que se cometan en la sentencia. Por último, al terminar de usar el *Statement*, este debe cerrarse. (*close()*) para evitar errores inesperados.

ACTIVIDADES 2.4

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita modificar la tabla álbum para incluir un nuevo campo que contenga las imágenes de las carátulas de cada álbum.

2.5 EJECUCIÓN DE CONSULTAS

La ejecución de consultas sobre bases de datos desde aplicaciones Java descansa en dos tipos de clases disponibles en el API JDBC y en dos métodos. Las clases son *Statement* y *ResultSet*, y los métodos, *executeQuery* y *executeUpdate*.

2.5.1 CLASE STATEMENT

Como se ha comentado en varias ocasiones, las sentencias *Statement* son las encargadas de ejecutar las sentencias SQL estáticas con *Connection.createStatement()*.

El método *executeQuery()* de *Statement* está diseñado para sentencias que devuelven un único resultado (ResultSet),[33] como es el caso de las sentencias SELECT.

```
ResultSet res = sta.executeQuery();
```

Los objetos de la clase *ResultSet* son los utilizados para representar la respuesta a las peticiones que se hacen a una base de datos. Esta clase no es más que un conjunto ordenado de filas de una tabla. Asociados a la clase *ResultSet* existen métodos como *next() y getXXX()* para iterar por las filas y obtener los valores de los campos deseados. Después de invocar al método next(), el resultado recién traído está disponible en *el ResulSet*. La forma de recoger los campos es pedirlos con algún método *getXXX()*. Si se sabe de qué tipo es el dato, se puede pedir con *getInt()*, *getString()*, etc. Si no se sabe o da igual el tipo (como en el ejemplo), bastará con un *getObject()*, que es capaz de traer cualquier tipo de dato.

El siguiente código muestra un ejemplo de acceso a la base de datos con una consulta SELECT que obtiene todos los álbumes cuyo título empieza por "B"

```
public void Consulta_Statement(){
        try {
                Statement stmt = conn1.createStatement();
                String query = "SELECT * FROM album WHERE titulo like 'B%'";
                ResultSet rs = stmt.executeQuery(query);
                while (rs.next()) {
                        System.out.println("ID - " + rs.getInt("id") +
                        ", Título " + rs.getString("titulo") +
                        ", Autor " + rs.getString("autor") );
        }
        rs.close();
        stmt.close();
        } catch (SQLException ex) {
                System.out.println("ERROR:al hacer un Insert");
                ex.printStackTrace();
        }
}
```

[33] Más información sobre esta clase se puede encontrar en:
http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html

El código ejecuta la consulta deseada con *executeQuery()* y el resultado lo devuelve en un *ResultSet* (llamado *rs*). El método *next()* de *Resultset* permite recorrer todas las filas para ir sacando cada uno de los valores devueltos. Como el atributo *id* es de tipo *Int* se usa un método *getInt()* para recuperarlo. Sin embargo, como *título* y *autor* es de tipo *VARCHAR()* se usa un *getString()*.

Por último, al terminar de usar *ResultSet* y *Statement*, estos deben cerrarse, *close()* para evitar errores inesperados:

2.5.2 CLASE PREPAREDSTATEMENT

Una primera variante de la sentencia *Statement* es la sentencia *PreparedStatement*. Se utiliza para ejecutar las sentencias SQL precompiladas. Permite que los parámetros de entrada sean establecidos de forma dinámica ganando eficiencia.

El siguiente ejemplo muestra la ejecución de la consulta de la sección anterior, pero parametrizando el criterio de búsqueda. La diferencia principal con respecto a los *Statement* es que las consultas pueden tener valores indefinidos que se establecen con el símbolo interrogación (?). En la consulta se ponen tantas interrogaciones como parámetros se quieran usar. En el siguiente ejemplo solo hay un parámetro.

```
public void Consulta_preparedStatement () {
        try {
                String query - "SELECT * FROM album WHERE titulo like? ";
                PreparedStatement pst = conn1.prepareStatement(query);
                pst.setString (1, "B%");
                ResultSet rs = pst.executeQuery();
                while( rs.next() ) {
                        System.out.println(“ID - “ + rs.getInt(“id”) +
                        “, Título “ + rs.getString(“titulo”) +
                        “, Autor “ + rs.getString(“autor”) );
        }
        rs.close();
        stmt.close();
        } catch (SQLException ex) {
                System.out.println(“ERROR:al hacer un Insert”);
                ex.printStackTrace();
        }
}
```

Al crear el *prepareStatement* se precompila la consulta para dejarla preparada para recibir los valores de los parámetros. Si el parámetro que se quiere colocar es de tipo *Int* se usa *setInt()*. Sin embargo, en el ejemplo anterior se quiere dar un valor de texto por lo que se usa *setString* (1, "B%"). El primer valor (1) indica que la "B%" se asocia con la primera interrogación que se encuentra. Si hubiese más parámetros (?) entonces habría que indicar la posición que ocupa para que el *preparedStatement* sepa a cuál asociarle el valor (setString(2,"), setString(3, ", etc.).[34]

[34] Más información sobre métodos para asignar valores a los parámetros de *preparedStatement* son mostrados en:
http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/PreparedStatement.html

2.5.3 CLASE CALLABLESTATEMENT

Otro tipo de sentencias son las asociadas a los objetos de la clase *CallableStatement*, que son sentencias *preparedStatement* que llaman a un procedimiento almacenado, es decir, métodos incluidos en la propia base de datos. No todos los gestores de bases de datos admiten este tipo de procedimientos. La manera de proceder es la misma que la mostrada en el ejemplo *prepareStatement* aunque lo que se le da como parámetro es el nombre del procedimiento almacenado junto con los valores de los parámetros del procedimiento puestos como parámetros del *Statement*.

En el siguiente código se muestra un ejemplo de llamada para un supuesto procedimiento almacenado llamado *DameAlbumes(titulo,autor)*, que devuelve todos los álbumes cuyo título y autor coincida con los valores dados.

```
CallableStatement cs = conn1.prepareCall("CALL DameAlbumes(?,?)");

// Se proporcionan valores de entrada al procedimiento
cs.setString(1, "Black%");
cs.setString(2, "Metallica");
```

ACTIVIDADES 2.5

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita consultar con consultas SELECT las tablas álbum y canciones. Las consultas deben ser parametrizadas (Statement) y no parametrizadas (preparedStatement). El resultado debe mostrarse en forma de lista de resultados.

2.6 GESTIÓN DE TRANSACCIONES

Una transacción en un sistema de gestión de bases de datos (SGBD) es un conjunto de órdenes que se ejecutan como una unidad de trabajo, es decir, de forma indivisible o atómica. Una transacción se inicia cuando se encuentra una primera sentencia DML y finaliza cuando se ejecuta alguna de las siguientes sentencias:

- Un **COMMIT** o un **ROLLBACK**.
- Una sentencia DDL, por ejemplo **CREATE**.
- Una sentencia DCL (lenguaje de control de datos), dentro de las que se incluyen las sentencias que permiten al administrador controlar el acceso a los datos contenidos en una base de datos (**GRANT** o **REVOKE**).

Las transacciones consisten en la ejecución de bloques de consultas manteniendo las propiedades ACID (*Atomicy-Consistency-Isolation-Durability*), es decir, permiten garantizar integridad ante fallos y concurrencia de transacciones.

Después de que una transacción finaliza, la siguiente sentencia ejecutada automáticamente inicia la siguiente transacción. Una sentencia DDL o DCL es automáticamente completada y por consiguiente implícitamente finaliza una transacción.

Una transacción que termina con éxito se puede confirmar con una sentencia COMMIT, en caso contrario puede abortarse utilizando la sentencia ROLLBACK. En JDBC por omisión cada sentencia SQL se confirma tan pronto se ejecuta, es decir, una conexión funciona por defecto en modo auto-commit. Para ejecutar varias sentencias en una misma transacción es preciso deshabilitar el modo auto-commit, después se podrán ejecutar las instrucciones, y terminar con un COMMIT si todo va bien o un ROLLBACK en otro caso.

El siguiente código muestra el uso de transacciones con el ejemplo de insertar valores en la tabla *álbum*.

```
public void Insertar_con_commit(){
        try {
                conn1.setAutoCommit(false);
                Statement sta = conn1.createStatement();
                sta.executeUpdate("INSERT INTO album " + "VALUES (5, 'Black Album', 'Metallica')");
                sta.executeUpdate("INSERT INTO album " + "VALUES (6, 'A kind of magic', 'Queen')");
                conn1.commit();
        } catch (SQLException ex) {
                System.out.println("ERROR:al hacer un Insert");
                try{
                        if (conn1!=null) conn1.rollback();
                } catch(SQLException se2) {
                        se2.printStackTrace();
                }       //end try
                ex.printStackTrace();
        }
}
```

Como se puede ver en el ejemplo, si las dos operaciones *Insert into* se ejecutan correctamente, entonces se aplica un *commit* antes de salir. Sin embargo, si no es así, y salta una excepción, hay que hacer un *rollback()*. El *rollback()* es obligado ponerlo dentro de un *try/catch*.

ACTIVIDADES 2.6

- Utilizar las tablas creadas en la Actividad 2.2 para crear una aplicación Java que permita realizar varias inserciones de datos en las tablas canciones y álbum de manera atómica. Si falla la inserción en una de las tablas entonces todo el proceso se debe anular.

2.7 CONCLUSIONES Y PROPUESTAS PARA AMPLIAR

En este capítulo se ha mostrado JDBC como alternativa para conectar aplicaciones Java con bases de datos. El uso de conectores es muy habitual en el desarrollo de aplicaciones ya que facilita enormemente el acceso a datos y la ejecución de persistencias, al independizar el código del sistema gestor de bases de datos subyacentes. Un programador de aplicaciones multiplataforma debe tener unos conocimientos avanzados sobre el uso de conectores.

El lector interesado en profundizar en las tecnologías expuestas en el capítulo puede hacerlo en las líneas siguientes:

- ✓ Optimización de conexiones (pool de conexiones en aplicaciones multihilo) y de acceso a datos relacionales con SQL.
- ✓ Otras alternativas para el acceso con conectores, por ejemplo Java Blend y SQLJ.

Para profundizar en estas líneas de trabajo, el título SQL y Java: *Guía para SQLJ, JDBC y tecnologías relacionadas*, de Jim Melton y Andrew Eisenberg, de la editorial RA-MA, es una buena referencia.

RESUMEN DEL CAPÍTULO

En este capítulo se ha abordado el acceso a datos con conectores. En concreto, siendo coherentes con el resto de contenidos de este libro, se ha trabajado con JDBC, la alternativa más extendida para el acceso a datos almacenados en sistemas gestores relacionales desde Java.

La primera parte se ha centrado en una introducción a JDBC y sus posibilidades.

La segunda parte muestra con ejemplos de código el uso de las principales interfaces Java que permiten ejecutar operaciones de modificación y consulta con SQL.

EJERCICIOS PROPUESTOS

Como ejercicio para aplicar lo visto en este capítulo se propone hacer una aplicación para gestionar una biblioteca. Los pasos que se deben realizar son:

1. Crear una base de datos en MySQL con la siguiente estructura:

   - Libros (Título, Número de ejemplares, Editorial, Número de páginas, Año de edición).
   - Socios de la biblioteca (Nombre, Apellidos, Edad, Dirección, Teléfono).
   - Préstamos entre libros y socios (Libros, Socio, Fecha inicio préstamo y Fecha fin de préstamo).

2. Hacer una aplicación (*back-end*) que permita a un administrador:

   - Dar de alta, dar de baja y modificar libros.
   - Dar de alta, dar de baja y modificar socios.

3. Completar la aplicación *back-end* para que el administrador pueda consultar socios y libros por diferentes criterios: por nombre, por apellidos, por título y por autor. Utilizar para las consultas *preparedStatement*, pasando los valores de consulta como parámetros.

4. Completar la aplicación *back-end* para que el administrador pueda realizar préstamos de un libro a un socio.

5. Completar la aplicación *back-end* para que el administrador pueda realizar las siguientes consultas:

   - Listado de libros prestados actualmente.
   - Número de libros prestados a un socio determinado.
   - Libros que han superado la fecha de fin de préstamo.
   - Socios que tienen libros que han superado la fecha de fin de préstamo.

TEST DE CONOCIMIENTOS

1. Con JDBC el resultado obtenido de una consulta SQL ejecutada con executeQuery() se almacena en un objeto de tipo:

   a) Connection.
   b) Resultset.
   c) SQLException.

2. ¿Los objetos de qué clase del JDBC permiten la posibilidad de operar con la estructura de la base de datos?

   a) Pool_de_conexiones.
   b) DatabaseMetaData.
   c) Resultset.

3. ¿Qué objeto del JDBC se usa para sentencias SQL que impliquen modificaciones en la base de datos?

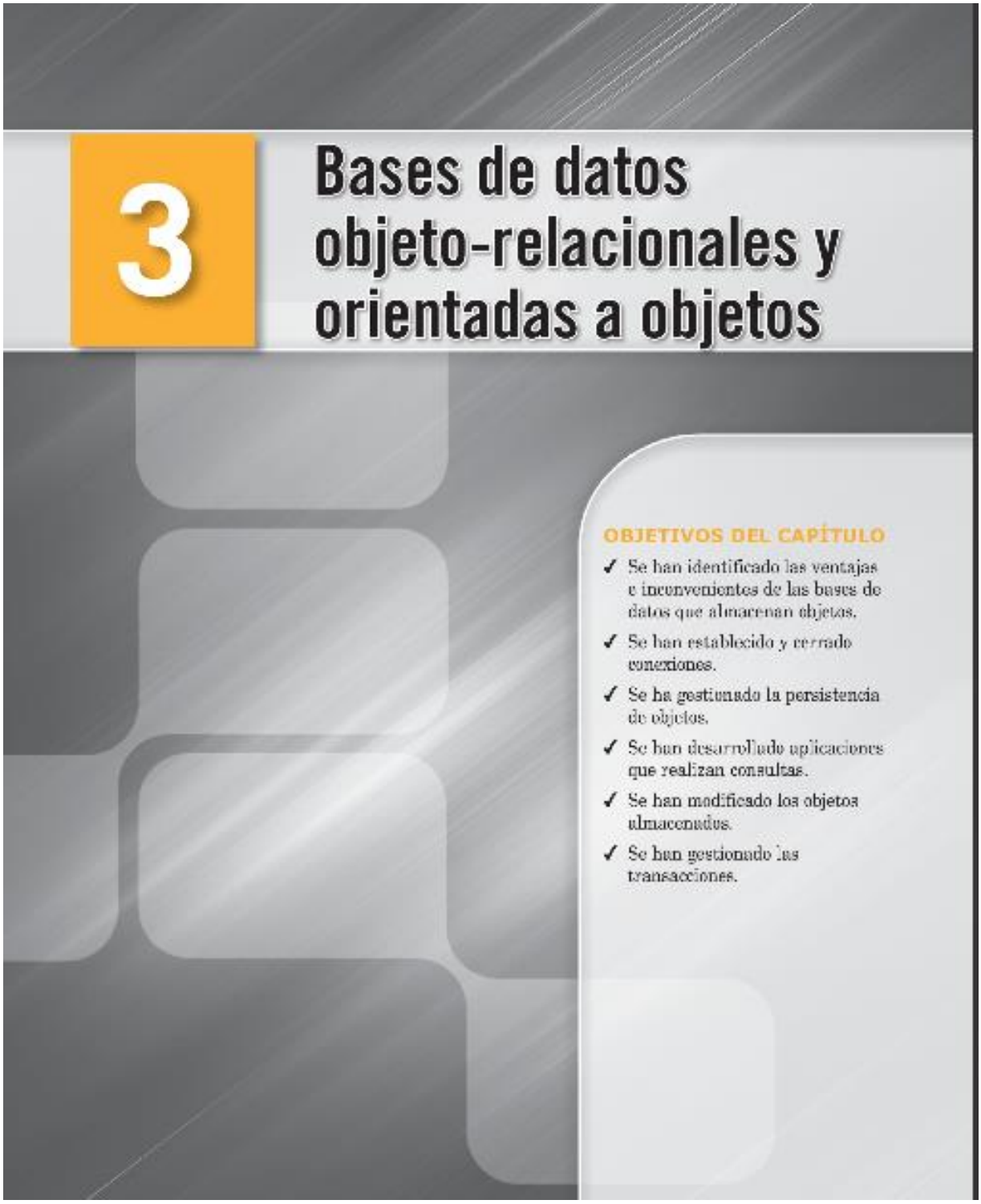
    a) executeQuery().
    b) executeUpdate().
    c) executePool().

4. En JDBC, los datos de un ResultSet (rs) que son de tipo int se recogen con:

    a) rs.getInt().
    b) rs.getFloat().
    c) rs.get().

5. En JDBC la variante de Statement que se usa para ejecutar las sentencias SQL precompiladas es:

    a) ParameterStatement.
    b) CallableStatement
    c) PreparedStatement.



3 Bases de datos objeto-relacionales y orientadas a objetos

OBJETIVOS DEL CAPÍTULO

- ✓ Se han identificado las ventajas e inconvenientes de las bases de datos que almacenan objetos.
- ✓ Se han establecido y cerrado conexiones.4Se ha gestionado la persistencia de objetos.
- ✓ Se han desarrollado aplicaciones que realizan consultas.
- ✓ Se han modificado los objetos almacenados.4Se han gestionado las transacciones.

Hace ya más de cuatro décadas que Edgar F. Codd desarrolló el modelo relacional para la gestión de bases de datos. Sin lugar a dudas, este modelo es referencia a la hora de almacenar y recuperar información. Tanto es así que es el modelo más extendido e implementado. Para darse cuenta de que el modelo relacional es indiscutiblemente válido solo hace falta fijarse en la cantidad de aplicaciones software implementadas sobre él.

Como es sabido, los sistemas gestores de bases de datos relacionales (SGBDR, en inglés RDBMS) son las aplicaciones software que gestionan los datos según el modelo relacional (no hay que confundir este con el software que lo implementa y gestiona). Arquitecturas privativas (como Oracle, Microsoft SQLServer) o basadas en software libre (como MySQL y PostgreSQL) son ejemplos actuales del gran rendimiento que este tipo de sistemas ofrece a los desarrollos software, sea cual sea el entorno y el contexto.

Sin embargo, pese a su eficacia, el modelo relacional se vio abocado en la década de los 90 a una actualización, a una mejora para adaptarse a los nuevos tiempos, al fantástico futuro que ofrecía el paradigma de Programación Orientada a Objetos (POO), impulsado de nuevo en estos años 90 por la imparable y creciente irrupción del lenguaje de programación Java. Destacando las diferencias, el modelo relacional hace hincapié en los datos y sus relaciones, mientras que el modelo orientado a objetos no se centra en los datos en sí, sino en las operaciones realizadas en esos datos. La aceptación recibida por la POO dio paso a una visión más amplia del problema, es decir, a un modelo orientado a objetos (modelo OO) como herramienta para diseñar software, crear código y almacenar datos.

Para satisfacer este último punto, el del almacenamiento de datos, en el mercado aparecieron nuevas propuestas de sistemas gestores de bases de datos orientadas a objetos (SGBD-OO, en inglés OODBMS) como ObjectStore u O2. Los SGBD-OO permiten almacenar objetos (persistencia) y recuperarlos según el modelo orientado a objetos, lo cual simplifica enormemente el desarrollo de software ya que, en teoría, con ellos no es necesaria una conversión entre el modelo de POO (por ejemplo con Java) y el modelo de base de datos (por ejemplo O2). Esta conversión sí es necesaria si se utiliza POO y un modelo relacional para el almacén de objetos (persistencia) ya que en este caso es obligada una conversión (mapeo) entre los datos estructurados en objetos y las propiedades de los objetos a las tablas y atributos del modelo relacional. Hay que entender que este mapeo siempre ofrece muchos problemas debido a que la conversión entre modelos no siempre es posible al no haber siempre equivalentes semánticos entre el modelo OO y el modelo relacional.

Pese a sus ventajas, el mayor problema de los SGBD-OO y el modelo OO es la odiosa comparación con el más-que-adecuado modelo relacional. En resumen, el principal escollo que ofrece el modelo OO como sistema para almacenar y recuperar datos es que es un modelo no formal. Frente a la matemática que subyace al modelo relacional (lógica de predicados y teoría de conjuntos), que garantiza su óptima implementación, el modelo OO ofrece una alternativa no formal, lo que siempre ha puesto en duda la implementación óptima de este tipo de sistemas gestores OO, y los relega a una posición inferior al compararlos con los sistemas relacionales. En términos generales, los SGBD-OO, por las características del propio modelo OO, no consiguen unos resultados tan buenos (espacio requerido para el almacenamiento, eficiencia en consultas, escalabilidad, etc.) a la hora de almacenar y recuperar información de ellos como sí ofrecen los SGBDR.

Esta puesta en duda de la eficiencia de los SGBD-OO provocó a finales de los 90 el desarrollo de sistemas híbridos que combinaran la eficiencia del modelo relacional con la simplicidad de utilizar el mismo modelo tanto en POO como en la persistencia de los objetos. Estas soluciones de compromiso se llamaron sistemas gestores objeto-relacionales (SGBD-OR, en inglés OR-DBMS). Los sistemas

gestores objeto-relacionales ofrecen una interfaz que simula ser OO, pero internamente los objetos se almacenan como en los sistemas relacionales clásicos. Con esta estructura "engañosa" lo que se pretendía (y pretende) es conseguir con los SGBD-OR las ventajas que ofrecen ambos modelos. Hasta la fecha se puede afirmar que lo híbrido funciona, puesto que grandes empresas como Oracle siguen apostando por soluciones SGBD-OR.

En este capítulo se estudiarán los SGBD objeto-relacionales y orientados a objetos, aunque se tratarán más en profundidad los SGBD orientados a objetos, mostrando alternativas de acceso a datos para su gestión y recuperación. En todo el capítulo se supone que el lector está familiarizado con los sistemas gestores relacionales y con SQL como lenguaje de modificación y consulta.

3.1 CARACTERÍSTICAS DE LAS BASES DE DATOS ORIENTADAS A OBJETOS

Como se ha comentado anteriormente, los SGBDR no son una alternativa óptima para almacenar objetos tal y como se entienden en la POO (Programación Orientada a Objetos). Esto es debido a que el modelo OO atiende a unas características que no son contempladas por el modelo relacional ni, por tanto, por la implementación que los SGBDR hacen de este modelo. Usar un SGBDR para almacenar objetos (persistencia) incrementa significativamente la complejidad de un desarrollo software ya que obliga a una conversión de objetos a tablas relacionales. Esta conversión implica:

- Mayor tiempo de desarrollo. El tiempo empleado en generar el código para la conversión de objetos a tablas y viceversa.
- Mayor posibilidad de errores debidos a la traducción, ya que no siempre es posible traducir la semántica de la OO a un modelo relacional.
- Mayor posibilidad de inconsistencias debidas a que el proceso de paso de modelo relacional a OO y viceversa puede realizarse de forma diferente en las distintas aplicaciones.
- Mayor tiempo de ejecución debido a la obligada conversión.

Por el contrario, si se usa un SGBD-OO los objetos se almacenan directamente en la base de datos, empleando las mismas estructuras y relaciones que los lenguajes de POO. Así, el esfuerzo del programador y la complejidad del desarrollo software se reducen considerablemente y se mejora el flujo de comunicación entre todos los implicados en un desarrollo (usuarios, ingenieros software y desarrolladores).

Un SGBD-OO debe contemplar las siguientes características:

- ✓ Características propias de la OO. Todo sistema OO debe cumplir características como Encapsulación, Identidad, Herencia y Polimorfismo, junto con Control de tipos y Persistencia.

- ✓ Características propias de un SGBD. Todo sistema gestor de bases de datos debe permitir 5 características principales: Persistencia, Concurrencia, Recuperación ante fallos, Gestión del almacenamiento secundario y facilidad de Consultas.

Todas estas características están ampliamente explicadas en el Manifiesto de las bases de datos OO propuesto en diferentes etapas por los expertos en bases de datos más prestigiosos de los años 80 y 90:

- Atkinson en 1989 propuso el Manifiesto de los sistemas de bases de datos orientadas a objetos puras.
- Stonebraker en 1990 propuso el Manifiesto de los SGBD de tercera generación, que propone las características que deben tener los sistemas relacionales para almacenar objetos. Esta propuesta es justamente la que contemplan los actuales SGBD-OR. Estas características fueron ampliadas en 1995 por Darwen y Date.

El manifiesto de Atkinson expone las siguientes características que todo SGBD-OO debe implementar:

1. Almacén de Objetos complejos: los SGBD-OO deben permitir construir objetos complejos aplicando constructores sobre objetos básicos.
2. Identidad de los objetos: todos los objetos deben tener un identificador que sea independiente de los valores de sus atributos.
3. Encapsulación: los programadores solo tendrán acceso a la interfaz de los métodos, de modo que sus datos e implementación estén ocultos.
4. Tipos o clases: el esquema de una BDOO incluye únicamente un conjunto de clases (o un conjunto de tipos).
5. Herencia: un subtipo o una subclase heredará los atributos y métodos de su supertipo o superclase, respectivamente.
6. Ligadura dinámica: los métodos deben poder aplicarse a diferentes tipos (sobrecarga). La implementación de un método dependerá del tipo de objeto al que se aplique. Para proporcionar esta funcionalidad, el sistema deberá asociar los métodos en tiempo de ejecución.
7. Completitud de cálculos usando el lenguaje de manipulación de datos (Data Management Language [DML]).
8. El conjunto de tipos de datos debe ser extensible. Además, no habrá distinción en el uso de tipos definidos por el sistema y tipos definidos por el usuario.
9. Persistencia de datos: los datos deben mantenerse (de forma transparente) después de que la aplicación que los creó haya finalizado. El usuario no tiene que hacer ningún movimiento o copia de datos explícita para ello.
10. Debe ser capaz de manejar gran cantidad de datos: debe disponer de mecanismos transparentes al usuario, que proporcionen independencia entre los niveles lógico y físico del sistema.
11. Concurrencia: debe poseer un mecanismo de control de concurrencia similar al de los sistemas convencionales.
12. Recuperación: debe poseer un mecanismo de recuperación ante fallos similar al de los sistemas relacionales (igual de eficientes).
13. Método de consulta sencillo: debe poseer un sistema de consulta de alto nivel, eficiente e independiente de la aplicación (similar al SQL de los sistemas relacionales).

En resumen, los SGBD-OO nacen de la necesidad de proporcionar persistencia a los desarrollos hechos con lenguajes de programación OO. Hay varias alternativas para proporcionar la persistencia, sin embargo, la alternativa natural es utilizar un sistema gestor que permita conservar y explotar todas las posibilidades que la POO permite. Esta alternativa la constituyen los SGBD-OO. Estos

sistemas, al igual que ocurre con los SGBDR, han sido ampliamente estudiados para dar con la mejor solución posible y la más estandarizada.

El manifiesto de Atkinson detalla muy bien ese estudio ya que define qué debe tener obligatoriamente todo sistema gestor que quiera llamarse SGBD-OO. El manifiesto de Atkinson es básicamente una declaración de intenciones. Para garantizar que la industria de las bases de datos siga unas pautas comunes (estándares) para desarrollar SGBD-OO son necesarias organizaciones o grupos que completen las características que todo SGBD-OO debe tener y además exija su cumplimiento para favorecer la interoperabilidad entre sistemas de diferentes fabricantes. En los SGBDR, ISO y ANSI velan por el estándar SQL. En los SGBD-OO es ODMG (Object Data Management Group) la organización encargada de estandarizar todo lo relacionados con los SGBD-OO.

3.1.1 ODMG (OBJECT DATA MANAGEMENT GROUP)

El ODMG[35] (Object Data Management Group) es un consorcio industrial de vendedores de SGBD-OO que después de su creación se afilió al OMG[36] (Object Management Group). El ODMG no es una organización de estándares acreditada en la forma en que lo es ISO o ANSI pero tiene mucha influencia en lo que a estándares sobre SGBD-OO se refiere. En 1993 publicó su primer conjunto de estándares sobre el tema: el ODMG-93, que en 1997 evolucionó hacia el ODMG 2.0. En enero de 2000 se publicó el ODMG 3.0. La última aportación referente a los SGBD-OO es la realizada por el OMG según ODMG 3.0 llamada "base de datos de 4.ª generación" [37], publicada en 2006.

Entre muchas otras especificaciones el estándar ODMG define el modelo de objetos que debe ser soportado por el SGBD-OO. ODMG se basó en el modelo de objetos del OMG (Object Management Group) que sigue una arquitectura de núcleo-componentes. Por otro lado, el lenguaje de base de datos es especificado mediante un lenguaje de definición de objetos (ODL) que se corresponde con el DDL de los SGBD relacionales, un lenguaje de manipulación de objetos(OML) y un lenguaje de consulta (OQL), que equivale al archiconocido SQL de ANSI-ISO. La arquitectura propuesta por ODMG incluye además un método de conexión con lenguajes tan populares como Smalltalk, Java y C. En las siguientes secciones se definirán con más precisión el modelo ODMG y los lenguajes ODL, OML y OQL.

3.1.2 EL MODELO DE DATOS ODMG

El modelo de objetos ODMG permite que los diseños OO y las implementaciones usando lenguajes OO sean portables entre los sistemas que lo soportan. El modelo de datos dispone de unas primitivas de modelado. Estas primitivas subyacen en la totalidad de los lenguajes orientados a objetos puros (como Eifflel, Smalltalk, etc.) y en mayor o menor medida en los híbridos (por ejemplo: Java, C, etc.).

Las primitivas básicas de una base de datos orientada a objetos son los objetos y los literales.

- Un objeto es una instancia de una entidad de interés del mundo real. Los objetos necesitan un identificador único (Identificador de Objeto [OID]).
- Un literal es un valor específico. Los literales no tienen identificadores. Un literal no tiene que ser necesariamente un solo valor, puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre (por ejemplo, enumeraciones).

[35] http://www.odbms.org/odmg/

[36] http://www.omg.org/

[37] http://www.odbms.org/About/News/20060218.aspx

Los objetos se dividen en tipos. Sin ser estrictos en la definición, un tipo se puede entender como una clase en POO. Los objetos de un mismo tipo tienen un mismo comportamiento y muestran un rango de estados común:

- El comportamiento se define por un conjunto de operaciones que pueden ser ejecutadas por un objeto del tipo (métodos en POO).
- El estado de los objetos se define por los valores que tienen para un conjunto de propiedades. Las propiedades pueden ser:
  - Atributos. Los atributos toman literales por valores y son accedidos por operaciones del tipo get_value y set_value (como exige la OO pura, y nunca se accede a ellos directamente).
  - Relaciones entre el objeto y uno o más objetos. Son propiedades que se definen entre tipos de objetos, no entre instancias. Las relaciones pueden ser uno-a-uno, uno-a-muchos o muchos-a-muchos.

Un tipo tiene una interfaz y una o más implementaciones. La interfaz define las propiedades visibles externamente y las operaciones soportadas por todas las instancias del tipo. La implementación define la representación física de las instancias del tipo y los métodos que implementan las operaciones definidas en la interfaz.

Los tipos pueden tener las siguientes propiedades:

- Supertipo. Los tipos se pueden jerarquizar (herencia simple). Todos los atributos, relaciones y operaciones definidas sobre un supertipo son heredadas por los subtipos. Los subtipos pueden añadir propiedades y operaciones adicionales para proporcionar un comportamiento especializado a sus instancias. El modelo contempla también la herencia múltiple, y en el caso de que dos propiedades heredadas coincidan en el subtipo, se redefinirá el nombre de una de ellas.
- Extensión: es el conjunto de todas las instancias de un tipo dado. El sistema puede mantener automáticamente un índice con los miembros de este conjunto incluyendo una declaración de extensión en la definición de tipos. El mantenimiento de la extensión es opcional y no necesita ser realizado para todos los tipos.
- Claves: propiedad o conjunto de propiedades que identifican de forma única las instancias de un tipo (OID). Las claves pueden ser simples (constituidas por una única propiedad) o compuestas (constituidas por un conjunto de propiedades).

3.1.3 ODL (LENGUAJE DE DEFINICIÓN DE OBJETOS)

ODL (lenguaje de definición de objetos) es un lenguaje para definir la especificación de los tipos de objetos en sistemas compatibles con ODMG. ODL es el equivalente al DDL (lenguaje de definición de datos) de los SGBD relacionales. ODL define los atributos y las relaciones entre tipos y especifica la signatura de las operaciones. ODL se utiliza para expresar la estructura y condiciones de integridad sobre el esquema de la base de datos: mientras que en una base de datos relacional,DDL define las tablas, los atributos en la tabla, el dominio de los atributos y las restricciones sobre un atributo o una tabla, en una base de datos orientada a objetos ODL define los objetos, métodos, jerarquías, herencia y el resto de elementos del modelo OO.

Una característica importante que debe cumplir (según ODMG) un ODL es ofrecer al diseñador de bases de datos un sistema de tipos semejantes a los de otros lenguajes de programación OO. Los tipos permitidos son:

- Tipos básicos: incluyen los tipos atómicos (Boolean, Float, Short, Long, Double, Chart, etc.) y las enumeraciones.
- Tipos de interfaz o estructurados: son tipos complejos obtenidos al combinar tipos básicos por medio de los siguientes constructores de tipos:
  - Conjunto (Set<tipo>) denota el tipo cuyos valores son todos los conjuntos finitos de elementos del tipo.
  - Bolsa (Bag<tipo>) denota el tipo cuyos valores son bolsas o multiconjuntos de elementos del tipo. Una bolsa permite a un elemento aparecer más de una vez, a diferencia de los conjuntos, por ejemplo {1, 2, 1} es una bolsa pero no un conjunto.
  - Lista (List<tipo>) denota el tipo cuyos valores son listas ordenadas finitas conteniendo 0 o más elementos del tipo. Un caso especial lo constituye el tipo String que es una abreviatura del tipo List<char>.
  - Array (Array<tipo,i>) denota el tipo cuyos elementos son arrays de i elementos del tipo.

Por tanto, con la ayuda de ODL se puede crear el esquema de cualquier base de datos en un SGBD-OO que siga el estándar ODMG. Una vez creado el esquema, usando el propio gestor o un lenguaje de programación se pueden crear, modificar, eliminar y consultar objetos que satisfagan ese esquema.

El siguiente ejemplo muestra la definición de un esquema usando ODL para el SGBD-OO Matisse. En el ejemplo mostrado abajo se definen dos tipos complejos llamados Libro y Autor:

- Un Libro tiene como atributos título de tipo básico String, año y páginas de tipo básico Integer.
- Un Autor tiene como atributos apellidos, nombre y nacionalidad de tipo String y edad de tipo Short.
- Entre ambos tipos hay relaciones definidas como conjuntos Set: un Libro es escrito_por un conjunto de autores y un Autorescribe un conjunto de libros.

```
interface Libro
{
        /* Definición de atributos
        attribute string título;
        attribute integer año;
        attribute integer paginas;
        attribute enum PosiblesEncuadernaciones (Dura,Bolsillo) tipo;

        /* Definición de relaciones */
        relationship Set<Autor> escrito_por inverse Autor::escribe;
}

interface Autor
{
        /* Definición de atributos */
         attribute string apellidos;
        attribute string nombre;
```

```
        attribute string nacionalidad;
        attribute short edad;

        /* Definición de relaciones */
        relationship Set<Libro> escribe inverse Libros::escrito_por;
}
```

ACTIVIDADES 3.1

Analizar las secciones anteriores para responder a las siguientes preguntas:

- ¿Hay alguna diferencia significativa entre el modelo OO que sigue un lenguaje de programación como Java y el modelo OO propuesto por ODMG para bases de datos OO?
- ¿Todos los elementos definidos en el modelo de datos ODMG se pueden encontrar en Java?
- ¿Se podría decir que, con lo visto, Java es compatible al 100 % con el modelo ODMG?

3.1.4 OML (LENGUAJE DE MANIPULACIÓN DE OBJETOS)

Una importante peculiaridad de ODMG es que no define ningún lenguaje de manipulación de objetos (OML). El motivo es claro: dejar descansar esta tarea en los propios lenguajes de programación, es decir, que sean los lenguajes de programación los que puedan acceder a los objetos y modificarlos, cada uno con su sintaxis y sus posibilidades. El objetivo es no diferenciar en la ejecución de un programa entre objetos persistentes almacenados en una base de datos y objetos no persistentes creados en memoria.

Lo que formalmente sugiere ODMG es definir un OML que sea la extensión de un lenguaje de programación de forma que se puedan realizar las operaciones típicas de creación, eliminación, modificación e identificación de objetos desde el propio lenguaje como se haría con objetos que no fueran persistentes.

Para mostrar cómo se pueden modificar objetos directamente con un lenguaje de programación, en la Sección 3.3 se trabajará con Java para realizar modificaciones en una base de datos gestionada con Matisse.

ACTIVIDADES 3.2

Analizar las conclusiones obtenidas en esta sección:

- ¿Siguen las bases de datos relacionales la misma filosofía respecto a OML que sigue ODMG?
- Pon algunos ejemplos de cómo se modifican atributos y tablas usando el OML de un SGBDR.
- ¿Es posible que la ausencia de un OML en los SGBD-OO diferencie mucho la modificación de objetos a como se hace en los SGBDR con OML?

3.1.5 OQL (LENGUAJE DE CONSULTAS DE OBJETOS)

OQL (lenguaje de consultas de objetos) es un lenguaje declarativo del tipo de SQL que permite realizar consultas sobre bases de datos orientadas a objetos, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras.

Tanto la definición de las bases de datos OO como de OQL fue posterior a las bases de datos relaciones y a SQL. De hecho, SQL ya estaba más que extendido y aceptado por los desarrolladores y clientes de bases de datos cuando apareció OQL. Por este motivo OQL no quiso reinventar la rueda e intentó definirse lo más parecido a la sintaxis usada en SQL (Select-From-Where) dentro de las posibilidades. Así los nuevos usuarios potenciales del OQL no apreciarían en este lenguaje diferencias significativas con respecto a SQL y obtendrían una curva de aprendizaje más rápida.

En los sistemas relacionales SQL es el estándar de consulta, sin embargo la implementación que los sistemas comerciales hacen de SQL puede variar de unos a otros (por ejemplo, MS-Access utiliza * para usarlo como comodín mientras que Oracle utiliza %). En OQL ocurre igual, en algunas implementaciones comerciales el estándar OQL no se corresponde exactamente con la implementación realizada. En esta sección mostraremos el OQL de la base de datos Matisse. Éste, básicamente, cumple con todas las características de OQL (ODMG) pero no satisface todo estrictamente.

La Figura 3.2 muestra un ejemplo de diagrama de clases para una biblioteca. Las consultas OQL mostradas se ejemplarizarán sobre ese mismo diagrama. Un ejemplo de OQL que devuelva el título de todos los artículos cuyo número de páginas sea mayor de 30 sería:

OQL_1
```
select a.titulo
from articulo a
where a.paginas>30;
```

En el siguiente ejemplo se muestra una consulta para obtener todos los nombres y apellidos (sin repetir) de los autores cuyos apellidos empiezan por “Mura”.

OQL2
```
select distinct a.nombre, a.apellidos
from autor a
where apellidos like ‘Mura%’;
```

En la consulta anterior, para cada objeto de la colección que cumple la condición se muestra el valor de los atributos incluidos en el select. El resultado por defecto es un tipo bag de tipo string y muestra todos los valores aunque estén duplicados. Sin embargo, si se utiliza distinct el resultado es de tipo set ya que se eliminan los duplicados.

Además, OQL permite hacer reuniones entre objetos. La siguiente consulta obtiene el título de los libros escritos por todos los autores cuyo apellido empieza por “Mura%”.

OQL3
```
select l.titulo, a.nombre, a.apellidosfrom autor a, Libro l
where a.escribe=l.OID and apellidos like ‘Mura%’;
```

La consulta anterior reúne los objetos autor con los objetos libro usando la relación escribe y el identificador único de objeto (OID).

Como se puede observar en la consulta, la sintaxis de OQL es idéntica a SQL. Sin embargo, esto ocurre solo en consultas sencillas, cuando se necesita explotar las características concretas de OO las consultas no tienen una sintaxis tan similar a SQL.

Por ejemplo, la siguiente consulta obtiene como resultado el título de los libros escritos por un autor y el nombre y apellidos. Sin embargo, el nombre y apellidos del autor no se obtienen invocándolos directamente, sino a través del método dameNombreyApellidos() definido en la clase Autor.

OQL4

```
select l.titulo, a.dameNombreyApellidos()
from autor a, Libro l
where a.escribe=l.OID;
```

Esta manera tan natural de incluir los métodos de los objetos en la propia consulta es de gran utilidad en OQL ya que no es necesario que el lenguaje posea primitivas de modificación, con la simple invocación de los métodos se puede consultar y modificar el estado (valores de las variables) de los objetos.

OQL tiene una sintaxis más rica que permite explotar todas las posibilidades de la OO. Sin embargo no es objeto de este capítulo tratar OQL en profundidad. Para más información sobre OQL (con Matisse) se puede consultar la guía para desarrolladores de Matisse.[38]

ACTIVIDADES 3.3

Diseñar en OQL las siguientes consultas sobre el modelo dado en la Figura 3.2:
Obtener el título de todas las obras almacenadas.
Obtener el título de todos los artículos cuyo autor tenga por nombre “Nikolai” y por apellido, “Gogol”.
Obtener el título y revista de todos los artículos cuyo resultado de invocar al método dameNombreyApellidos() sea “Nikolai Gogol”.

[38] http://www.matisse.com/pdf/developers/sql_pg.pdf

3.2 SISTEMAS GESTORES DE BASES DE DATOS ORIENTADAS A OBJETOS

Aunque la oferta no es tan extensa como ocurre con los SGBD relacionales también hay una oferta significativa de SGBD-OO en el mercado. Como en el caso de los sistemas relacionales existen:

- Sistemas privativos, como ObjectStore,[39] Objectivity/DB[40] o Versant.[41]
- Sistemas bajo licencias de software libre como Matisse[42] (versión para desarrolladores) y db4o[43] (versión liberada bajo licencia por Versant).

En esta sección se pretende mostrar los pormenores de un SGBD-OO pero desde la perspectiva de una solución como Matisse. Este SGBD-OO es una alternativa que respeta en gran medida el estándar ODMG, por lo que es una buena referencia para probar los diferentes lenguajes de definición, manipulación y consulta de objetos descritos en la sección anterior.

Matisse, de ADB Inc., es un SGBD-OO que da soporte para ser manejado con C, Eiffel, Java y .NET. Según sus autores, Matisse está orientado a trabajar con gran cantidad de datos con una rica estructura semántica. Además del control de transacciones, acceso, etc., propio de cualquier sistema gestor relacional y no relacional, Matisse tiene ventajas para la gestión propias de la OO. Algunas de ellas son:

- Técnicas para fragmentar objetos grandes en varios discos para optimizar así el tiempo de acceso.
- Una ubicación optimizada de los objetos en los discos.
- Un mecanismo automático de duplicación que proporciona una solución software a los fallos de tolerancia del hardware: los objetos (en lugar de los discos en sí) se pueden duplicar especularmente en varios discos, con recuperación automática en caso de fallo del disco.
- Un mecanismo de versiones de objetos incorporado.
- Soporte para una arquitectura cliente-servidor en la cual un servidor central gestiona los datos para un número posiblemente elevado de clientes, que mantienen una "reserva" de objetos a los que se haya accedido recientemente.
- Con respecto a la implementación del modelo OO, Matisse ofrece un mecanismo de optimización de acceso a objetos relacionados. Por ejemplo, si una clase como Libro posee como atributo Autor, Matisse mantendrá, si así se solicita, los enlaces inversos de forma automática de modo que será posible no solo acceder a los autores de un libro sino también a los libros de un autor determinado.

[39] http://www.progress.com/es/objectstore/
[40] http://objectivity.com/products/objectivitydb/overview
[41] http://www.versant.com/products/versant-object-database
[42] http://www.fresher.com/
[43] http://www.versant.com/products/db4o-object-database

3.2.1 INSTALACIÓN DE MATISSE

Matisse puede descargarse de su web oficial[44] para diferentes sistemas. La instalación es sencilla, un ejecutable que no necesita de ninguna configuración salvo la básica de cada sistema. Todo lo que ofrece Matisse está bien documentado en el sitio oficial. La versión con la que se trabaja en los ejemplos de este capítulo es Matisse9.0.5 para Windows 7.

Una vez arrancado Matisse se accede a un sencillo entorno de gestión. La Figura 3.1 muestra el entorno una vez arrancado. La ventana principal tiene tres partes bien diferenciadas:

- El árbol de la izquierda muestra las bases de datos creadas. Desplegando cada una se puede acceder a los elementos (espacio de nombres, clases, métodos, atributos, etc.) de cada base de datos.
- La parte de la derecha sirve para ejecutar los lenguajes ODL y OQL sobre las bases de datos y mostrar los resultados de estas y otros comandos que Matisse permite.

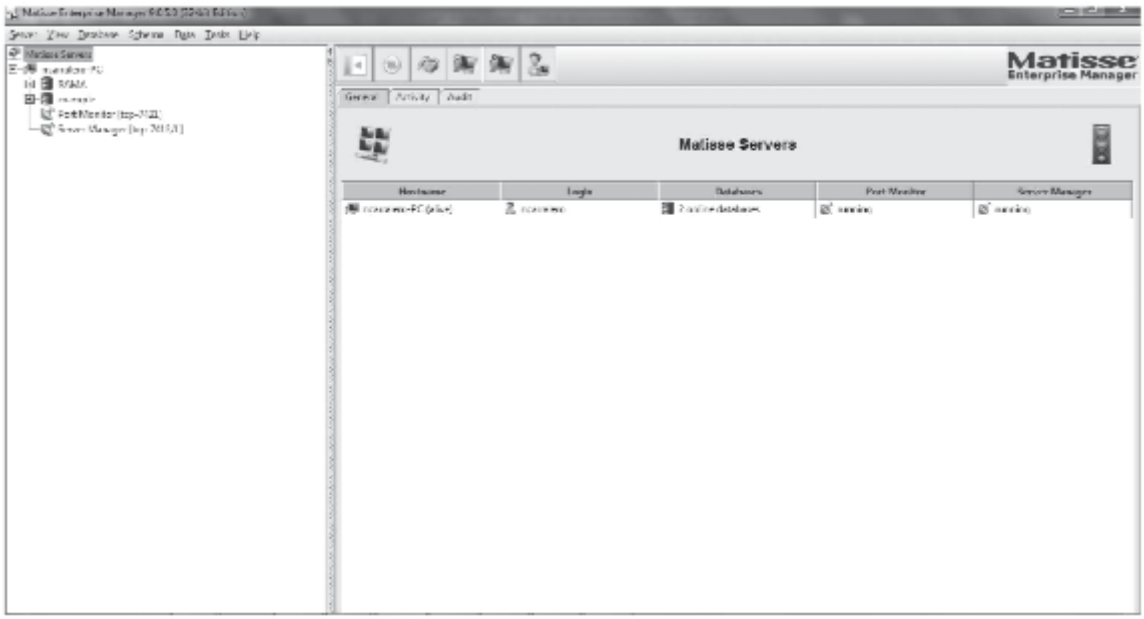


Figura 3.1. Entorno de Matisse

Al igual que ocurre con muchos de los sistemas gestores empleados en este libro no se pretende explicar en detalle el funcionamiento de Matisse, sino únicamente los aspectos esenciales para entender el acceso a datos desde un lenguaje de programación (Java). Por tanto, la siguiente sección da a conocer cómo se puede crear un esquema de bases de datos nuevo sobre el que más adelante se trabajará con el acceso mediante Java. Si el lector está interesado en los pormenores de este sistema gestor, puede acceder a la web oficial para recabar información más detallada.

[44] www.fresher.com

3.2.2 CREANDO UN ESQUEMA CON MATISSE

En las secciones siguientes se trabajará sobre un ejemplo concreto para mostrar una aplicación de acceso a SGBD-OO. Sin embargo, para ello es necesario previamente utilizar el entorno de Matisse para crear el esquema básico de base de datos. A continuación se muestra cómo crear ese esquema.

El modelo de ejemplo que se muestra en la Figura 3.2 es con el que se trabajará en las siguientes secciones y representa una simplificación de una Biblioteca. El modelo contiene:

- Una clase Autor, que representa a todos los posibles autores de una obra literaria. Sus atributos son nombre, apellidos y edad. Además tiene un método dameNombreyApellidos() que devuelve la concatenación del nombre y la edad en una única cadena.
- Una clase Obra, que representa a los diferentes tipos de creaciones que puede hacer un autor. Tiene como atributos título y páginas.
- Una clase Libro, que hereda de Obra y añade un atributo más llamado editorial que representa a la editorial que publica el libro.
- Una clase Artículo, que también hereda de Obra y añade otro atributo llamado revista que representa a la revista que publica el artículo.

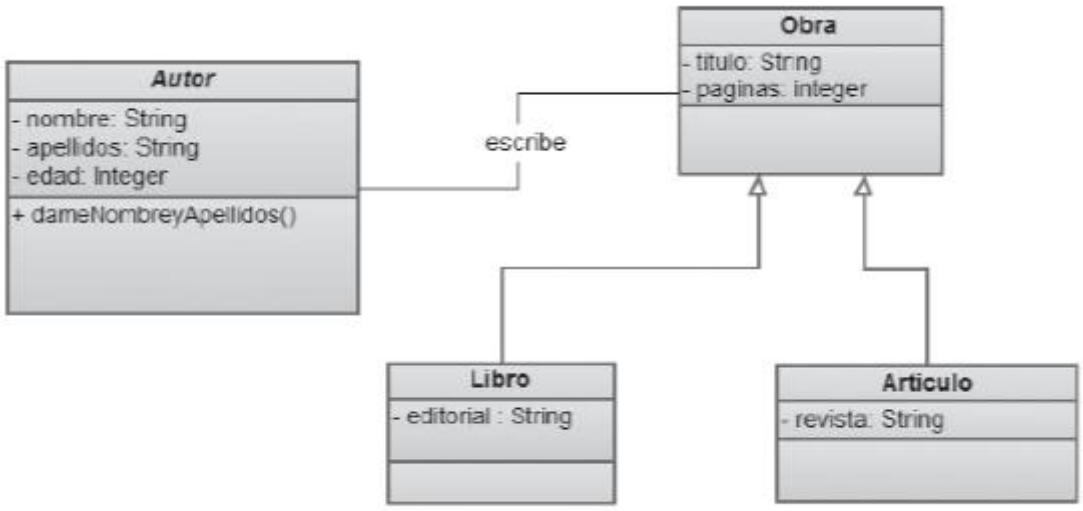


Figura 3.2. Modelo de datos ejemplo - Biblioteca

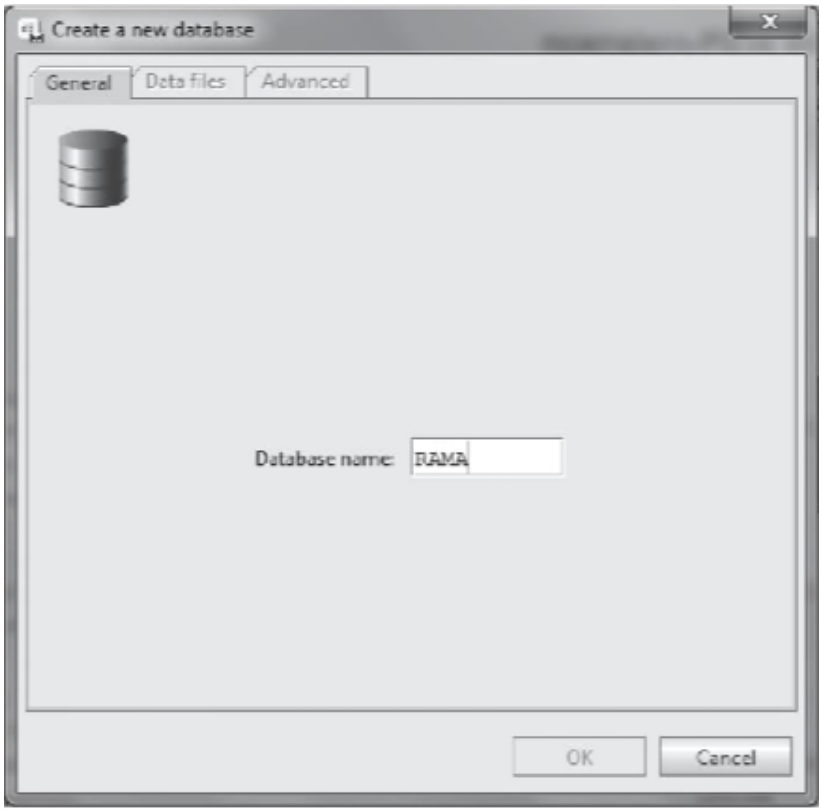


Figura 3.3. Crear una BBDD

Para crear este esquema en Matisse se deben seguir los siguientes pasos:

1. Crear una nueva base de datos (Database). Para ello se selecciona la opción de menú **Server -> New Database**. Entonces saldrá una ventana que pedirá el nombre de la nueva base de datos. En este ejemplo, como muestra la Figura 3.3, la base de datos se llamará RAMA.
2. Creada la base de datos, esta aparece en la estructura de árbol pero con una cruz blanca sobre fondo rojo. Eso indica que hay que arrancar la base de datos para poder utilizarla. Esto se hace pulsando con el botón derecho del ratón sobre el nombre de la base de datos y seleccionando la opción Start en el menú contextual que aparece.
3. Arrancada la base de datos aparecerá una estructura en el árbol similar a la de la Figura 3.4.

Figura 3.4. Estructura Matisse

- Namespaces representa el espacio de nombre en el cual se crearán las clases asociadas. Para una buena organización de la información es interesante definir un espacio de nombre que no cree ambigüedad sobre dónde están situados los elementos de la base de datos.[45]
- Classes albergará la estructura propia de tipos definidos. En el ejemplo será (Autor, Libro, Revista y Obra).
- Methods contendrá los métodos definidos en cada clase. Hay que recordar que en un SGBD-OO (Atkinson en la Sección 3.1) se deben poder almacenar objetos con sus atributos y sus métodos asociados. Indexes contiene los índices creados sobre los objetos (creados por el usuario) para optimizar las consultas.[46]

4. Crear una Namespace para la estructura Biblioteca. Para ello se pulsa con el botón derecho del ratón sobre **Namespaces** y se selecciona la opción **New Namespace**, que aparece en el menú contextual. En la parte derecha aparecerá una plantilla de ayuda para escribir la sentencia ODL create namespace. La Figura 3.5 muestra cómo crear el nuevo espacio de nombres (biblioteca).

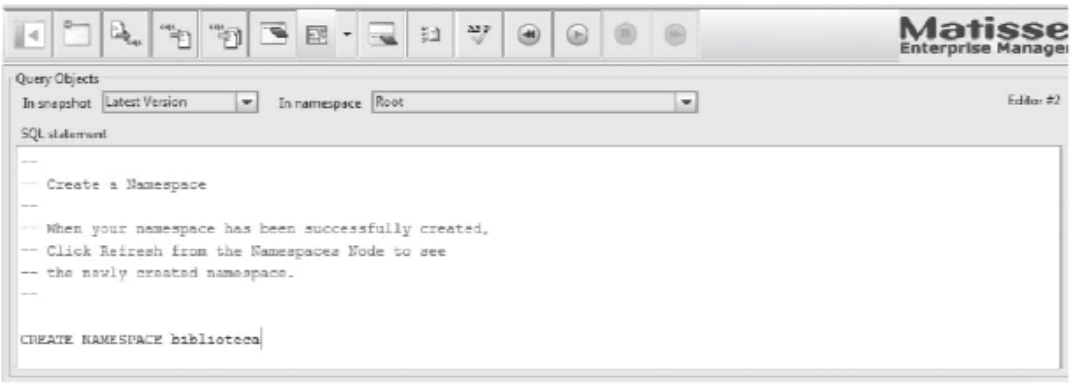


Figura 3.5. Namespace Biblioteca

5. Creado el espacio de nombres, lo siguiente es crear la estructura definida en la Figura 3.2. Para ello se pulsa en **Classes** y se selecciona la opción de menú **New Class**. Al igual que con los namespaces, en la parte derecha saldrá una plantilla con la sintaxis para crear una nueva clase. Esta plantilla utiliza un lenguaje propio de Matisse pero que luego puede ser exportado a ODL (ODMG). La Figura 3.6 muestra el código necesario para la creación de la clase Autory se puede observar en ella cómo es necesario seleccionar de la lista (In namespace) el namespacebiblioteca para que la clase se cree en ese espacio de nombres y no en la raíz (root).

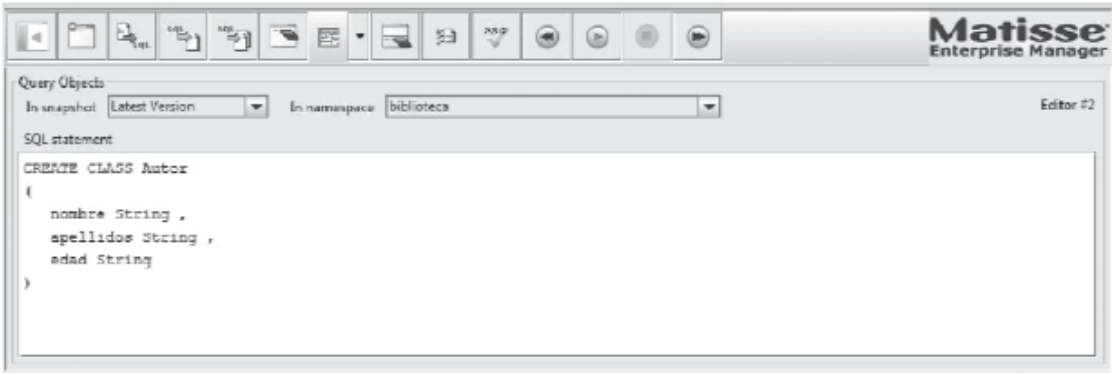


Figura 3.6. Clases – Biblioteca

[45] Aquí, el concepto de Namespace es sinónimo del utilizado en Java para los paquetes (package).
[46] Los índices en SGBD-OO son necesarios, al igual que ocurre en los SGBD relacionales.

Las otras clases del ejemplo se crean de la misma manera con el siguiente código. Debemos observar que en el código no se especifica la relación entre Obra y Autor ni se define el método dameNombreyApellidos(),estos elementos se definirán en el siguiente paso.

```
CREATE CLASS Obra
(
        titulo string,
        paginas Integer
)

CREATE CLASS Libro
        INHERIT Obra
(
        editorial String
)

CREATE CLASS Articulo
        INHERIT Obra
(
        revista String
)
```

6. Por último, falta por crear las relaciones entre Obra y Autor (escribe) y el método dameNombreyApellidos() que devuelve ambos atributos de Autor concatenados. Para ello se usa el mismo procedimiento que para crear las clases.

    - En el árbol se selecciona la clase a la que se quiere crear una nueva relación, por ejemplo Autor, y se pulsa con el botón derecho del ratón. En el menú contextual se selecciona Alter **Class -> Add Relationship**. Entonces saldrá una plantilla en la parte derecha con la sintaxis para añadir una nueva relación. El siguiente código es el necesario para crear primero una relación entre Autor y Obra (escribe) y segundo la inversa entre Obray Autor (escrita_por). Esto optimizará la recuperación de objetos. Es importante recordar que se debe poner el namespaces en Biblioteca para que el sistema sepa que las clases que se quieren relacionar están bajo ese espacio de nombres.

```
ALTER CLASS Autor
        ADD RELATIONSHIP escribe
        RELATIONSHIP SET( Obra)

        INVERSE Obra.escrito_por;

ALTER CLASS Obra
        ADD RELATIONSHIP escrito_por
        RELATIONSHIP SET( Autor)

        INVERSE Autor.escribe;
```

De la misma manera se añade un método, sin embargo en este caso se selecciona la clase Autor y pulsando en el botón derecho se selecciona la opción de menú Alter Class -> Add Method. El código para añadir un método a Autor[47] que concatene el nombre y el apellido será:

```
CREATE METHOD dameNombreyApellidos ()
  RETURNS String
  FOR Autor
  --
  -- Describe your method here
  --
  BEGIN
  return CONCAT (nombre, apellidos);
END;
```

ACTIVIDADES 3.4

- Instalar en local el sistema gestor Matisse. Una vez instalado, y siguiendo los pasos descritos en esta sección, crear una base de datos según el modelo de la Figura 3.2.

3.3 INTERFAZ DE PROGRAMACIÓN DE APLICACIONES DE LA BASE DE DATOS

Una vez el esquema de la base de datos se ha creado, el siguiente paso es preparar el sistema para que pueda ser accedido desde código Java. Como se ha comentado, ODMG no define un lenguaje de manipulación de objetos (OML) y Matisse tampoco, por lo que la única manera de añadir, eliminar, modificar y consultar objetos en el esquema creado en la sección anterior es usando código fuente. En esta sección se tratará esta forma de acceso mediante código fuente. Aunque Matisse ofrece alternativas para varios lenguajes (como por ejemplo Eiffel, C y Microsoft .NET), en los ejemplos mostrados se usará Java.

3.3.1 PREPARANDO EL CÓDIGO JAVA

Para poder entender el proceso para acceder a los objetos almacenados en las bases de datos OO hay que tener siempre presente que lo que se busca es tener la sensación de trabajar solo con objetos, sin tener que pensar en si están almacenados en una base de datos o están creados en memoria. Es decir, se busca un acceso a objetos totalmente transparentes.

[47] Los métodos de las clases son como procedimientos almacenados (típicos de los SGBDR). Por ello, su ejecución siempre se hará en el servidor de bases de datos.

Por lo tanto, cuando se trabaja con SGBD-OO hay que olvidarse de buscar algo similar a las típicas API de acceso a datos de sistemas relacionales como ODBC o JDBC. Lo que se tiene que buscar es una manera de que el SGBD-OO genere en un lenguaje de programación (Java en este caso) las clases que componen el esquema de base de datos. Esta es la clave del problema, lo que beneficia el trabajo con SGBD-OO desde lenguajes de programación OO y lo que lo diferencia a su vez del acceso a datos en sistema relacionales.

La librería de clases creada por el SGBD-OO en el lenguaje de programación seleccionado (en nuestro caso Java) puede ser integrada en un proyecto en el cual, usando la librería adecuada, se puede abordar la persistencia de un objeto con solo invocar a un método del mismo. El mecanismo por el cual el contenido de un objeto se almacena en la base de datos es totalmente transparente al programador. Solo tiene que pedir que se haga persistente y el objeto se hará, solo hay que pedir que se recupere y el objeto aparecerá en memoria como si de cualquier otro objeto se tratase.

A continuación se creará con Matisse la estructura de clases en Java que representa el esquema Biblioteca de la base de datos RAMA.

1. Seleccionar la opción de menú Schema -> Generate Code. En el siguiente menú hay que seleccionar RAMA para que su esquema se convierta en clases Java. La Figura 3.7 muestra la ventana de generación:

Figura 3.7. Crear código Java

Las opciones de generación son las siguientes:

- Project directory (directorio de proyecto): se utiliza para indicar dónde está ubicada la carpeta del proyecto Java en el que se integrará el código Java generado. Esto es útil para que las clases creadas se coloquen en la carpeta deseada y dentro del paquete seleccionado y así no tener que hacer importaciones posteriores.
- Lenguage (lenguaje destino): indica el lenguaje en el que se quiere generar el código, en este caso Java.

- Database namespace: se utiliza para indicar de qué espacio de nombres almacenados en la base de datos se quieren sacar las clases de las que se generará el código. En este caso será del namespaceBiblioteca.
- Language Package: sirve para indicar el paquete Java en el que se quieren agrupar las clases que se generarán. En el ejemplo de la Figura 3.7 todas las clases se colocarán en un paquete llamado biblioteca.
- SQL method call: indica si se desea hacer llamadas a SQL48 para invocar a los métodos definidos en la base de datos. En el ejemplo Biblioteca se creó un método en la clase Autor llamado dameNombreyApellidos().Seleccionando la opción SQL method call, Matisse genera el código necesario para que cuando el usuario quiera invocar este método de la clase Autor se llame directamente a su código ODL almacenado en la base de datos. De alguna manera, los métodos creados en la base de datos son como procedimientos almacenados (típicos de los SGBDR), que siempre deben ser ejecutados en el servidor de base de datos por razón de optimización. Por esto es por lo que el código generado en el método no se traduce a Java en la generación de las clases Java, sino que solo se pone el código para conectar a la base de datos y ejecutarlo desde el propio Matisse. En las siguientes secciones se concretará este código y su acceso desde Java.

2. Una vez generadas las clases, todas se ubicarán en el paquete seleccionado dentro del directorio del proyecto seleccionado. Esas clases ya estarán listas para ser incorporadas a un proyecto y poder realizar operaciones de modificación y consulta.

La Figura 3.8 muestra un proyecto Java (hecho en NetBeans IDE 7.1.2) en el que se muestra un paquete biblioteca (matisseoo.biblioteca) que contiene las clases Java creadas desde Matisse.

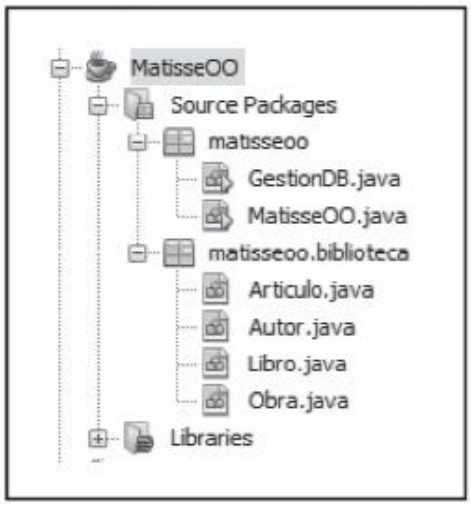


Figura 3.8. Estructura de clases en Java

3. Para que el proyecto reconozca las clases y métodos creados por Matisse es necesario incluir en el proyecto la librería matisse.jar. En el caso de que la instalación de Matisse se haga en la unidad C:\, bajo sistema Windows, la ruta donde se localiza la librería sería: C:\Products\Matisse\lib\.

[48] Desafortunadamente, Matisse utiliza SQL para referirse al lenguaje que usa para interactuar con las bases de datos OO que almacena. Sin embargo, no hay que confundirlo con SQL de ANSI ya que no sigue la misma s