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*[94] 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*)[95] 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

94 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*

95 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 *FileWriter*[96] 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 *true* indica 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 IOException
```

96 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 *FileReader*[97] 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 *FileOutputStream*[98] 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 *FileInputStream*[99] 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 *RandomAccessFile*[100] permite acceder directamente a cualquier posición dentro del fichero. Proporciona dos constructores básicos:

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

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

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

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

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

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).[101] 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.[102]

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.

101 *http://www.w3.org/DOM/*

102 *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 *Libro*, dejando los otros dos sin detallar.

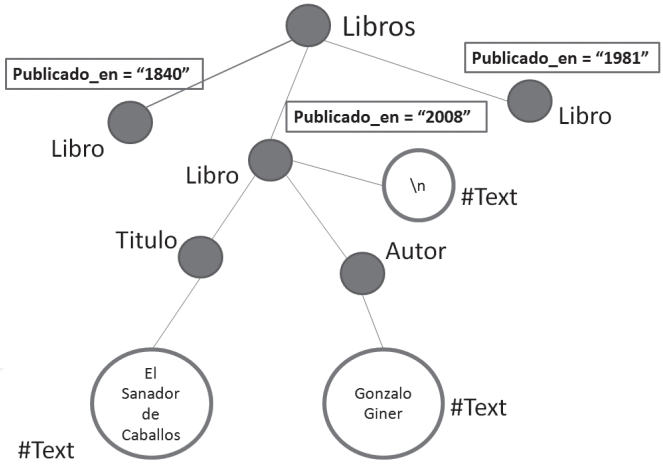

Figura 1.2. *Ejemplo DOM*

La generación del árbol DOM a partir de un documento XML se hace de la siguiente manera:[103]

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 *Libro* descendiente 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.

103 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).

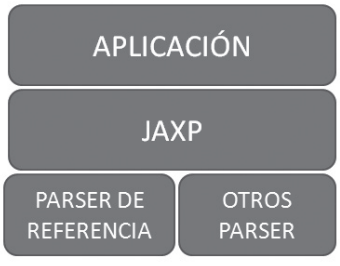

Figura 1.3. *Relación entre JAXP y parsers*

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).

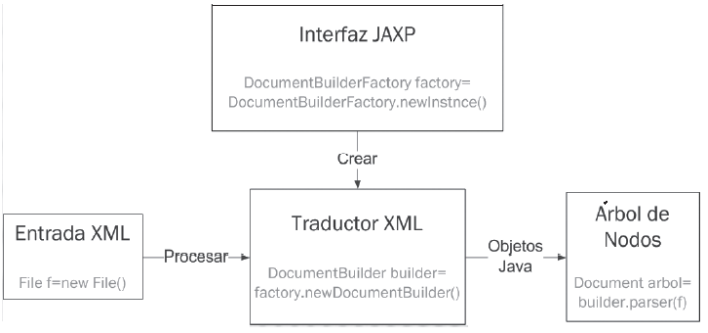

Figura 1.4. *Esquema de acceso a DOM desde JAXP*

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:[104]

- *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*:[105]

- 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

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

105 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 *XMLSerializer* 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[106] (*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.

106 *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*[107] y *SAXParser*[108] proporcionan el acceso desde JAXP. La clase *DefaultHandler*[109] 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 secuencialmente
    sh=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*.

107 Más información sobre SAXParserFactory es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/SAXParserFactory.html*

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

109 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 *DefaultHandler* (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.

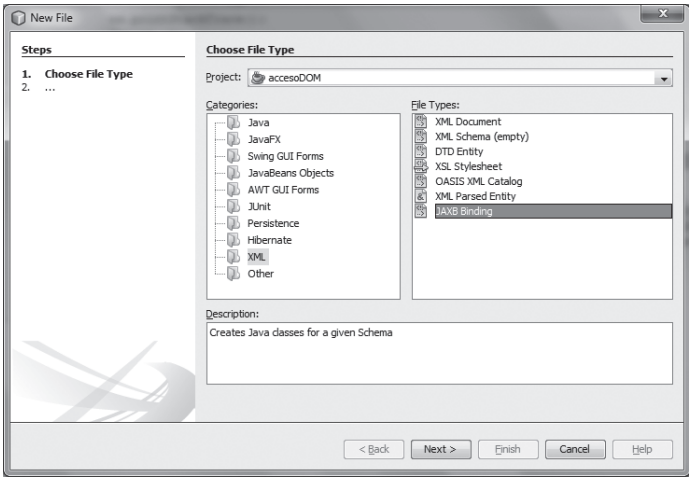

Figura 1.6. *JAXB desde NetBeans - nuevo fichero*

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*).

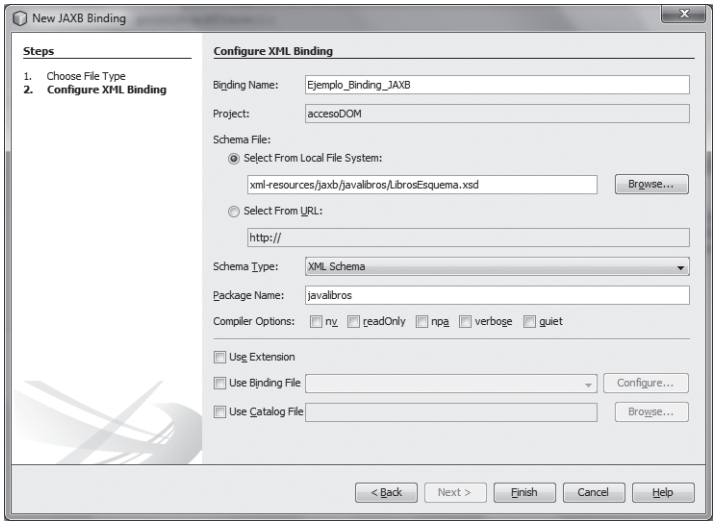

Figura 1.7. *JAXB desde NetBeans - Enlace*

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: *JAXBContext* y *Unmarshaller* que pertenecen al paquete *javax.xml.bind.**.[110]

- 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.

110 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.[111]
  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).

111 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.[112]

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*

112 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*[113], 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*[114], 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;
```

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

114 Más información sobre *XPathExpression* es mostrada en: *http://docs.oracle.com/javase/7/docs/api/javax/xml/xpath/XPathExpression.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 jDOM[115] 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.

115 *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, editorial, 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.