

TEMA

Tema 1. Persistencia en ficheros

Desarrollo de aplicaciones
multiplataforma

Acceso a datos

Autora: Silvia Macho



Tema 1: Persistencia en ficheros

¿Qué aprenderás?

- Acceder a ficheros desde una aplicación Java.
- Gestionar la persistencia en ficheros XML.

¿Sabías que...?

- XML es el estándar hoy en día para guardar datos en documentos sin darles formato.
- DOM (Document Object Model) es una interfaz de programación de aplicaciones (API) para documentos HTML y XML, multiplataforma e independiente del lenguaje de programación que se utiliza para representar documentos HTML, XHTML y XML e interaccionar con ellos a través de objetos.



1.1. Introducción

En los sistemas informáticos actuales, en los cuales puedes existir millones de ficheros, es imprescindible un sistema que permita una gestión eficaz de localización, de forma que los usuarios puedan moverse de forma cómoda entre tantos archivos. La mayoría de los sistemas de ficheros, se organizan de forma jerárquica, en forma de directorios, que facilitan la clasificación, identificación y localización de los archivos. Podemos hablar de indistintamente de directorios o carpetas y de ficheros o archivos.

Los sistemas Linux/Unix, para poder gestionar la variedad de ficheros, utilizan la estrategia de unificar todos los sistemas en uno único, para conseguir una forma de acceso unificada y con una única jerarquía que facilite el acceso a cualquiera de sus componentes, con independencia del sistema de ficheros donde esté ubicado.

```
/camino_donde_esta_el_fichero/fichero.txt
```

En cambio, sistemas como Windows. Utiliza la estrategia de mantener bien diferenciados cada uno de los sistemas y dispositivos a los que tiene acceso. Para distinguir a cada uno de esos elementos, utilizar una letra de unidad que es la raíz de la ruta del objeto a referenciar.

```
F:\camino_donde_esta_el_fichero\fichero.txt  
\Servidor\camino_donde_esta_el_fichero\fichero.txt
```



1.2. La clase File

En java, para gestionar el sistema de ficheros, se utiliza la clase File. Es la clase que debe entenderse como la referencia a la ruta o localización de ficheros del sistema. No representa el contenido del fichero, sino la ruta donde se ubica. Como se trata de una ruta, la clase representa tanto ficheros como directorios.

Si utilizamos una clase para representar rutas, conseguimos una independencia total respecto de la notación que cada sistema operativo pueda utilizar. La estrategia que utiliza cada sistema operativo no afecta a la clase File.

Los objetos de la clase File están directamente vinculados con la ruta especificada. Esto significa que estos objetos, durante todo su ciclo de vida, sólo representarán una única ruta, la que se les asocia en el momento de crearlos. La clase File, no dispone de ningún método que permita modificar la ruta asociada. En caso de necesitar nuevas rutas, siempre tendremos que crear un nuevo objeto y no será posible reutilizar los anteriores.

La clase File, encapsula prácticamente toda funcionalidad necesaria para gestionar un sistema de ficheros organizado en árbol de directorios. Esta gestión incluye:

- Funciones de manipulación y consulta de la propia estructura jerárquica: creación, eliminación, obtención de la ubicación, etc. de ficheros y carpetas.
- Funciones de manipulación y consulta de las características particulares de los elementos: nombres, tamaño, capacidad, etc.
- Funciones de manipulación y consulta de los atributos específicos de cada sistema operativo y que, por tanto, sólo será funcional si el sistema operativo anfitrión soporta también la funcionalidad. Nos referimos, por ejemplo, a los permisos de escritura, ejecución, atributos de ocultación, etc.



La clase **File** reside en el paquete `java.io`, junto con muchas otras clases relacionadas con la gestión, manipulación y acceso a archivos.

La forma más sencilla de uso es así:

```
File f = new File("ruta/a/archivo");
```

De este modo instanciamos un nuevo objeto de esta clase y que permitirá el acceso a las funciones de manipulación de archivos sobre este archivo concreto, el especificado como argumento en el constructor del objeto.

También podemos invocar el constructor de otros modos.

El primero sirve para indicar la ruta separando el directorio y el archivo:

```
File f = new File("ruta/a/directorio", "archivo");
```

El segundo permite reutilizar un objeto **File** que apunte al directorio donde resida el archivo:

```
File padre = new File("ruta/a/directorio");
File f = new File(padre, "archivo");
```

A partir de aquí podremos llamar a las diferentes funciones, algunas de ellas mencionadas en el libro del curso, pero que aquí ampliamos en forma de tabla:

Función	Descripción
Funciones generales	
boolean exists()	Devuelve un valor cierto/falso en función de si el archivo o directorio existe en el disco.
String getPath()	Devuelve el camino que contiene este objeto de disco.
String getName()	Devuelve el nombre del archivo (eliminando la parte del camino).

String getAbsolutePath()	Devuelve el camino absoluto (convirtiendo la parte relativa según convenga).
String getCanonicalPath()	Devuelve el camino absoluto, ahora resolviendo los “.” y “..” según convenga. Este método puede generar una excepción java.io.IOException .
String getParent()	Devuelve el objeto directorio padre, si es conocido.
boolean isHidden()	Devuelve un valor cierto/falso en función de si el objeto tiene el atributo de oculto o no.
boolean delete()	Elimina este objeto del directorio.
Funciones de archivo	
boolean isFile()	Devuelve un valor cierto/falso en función de si el objeto es de tipo archivo o no (por ejemplo un directorio).
long length()	Devuelve el tamaño en bytes del archivo.
long lastModified()	Devuelve la fecha-hora de la última modificación del archivo.
boolean canRead()	Devuelve si el usuario tiene derecho de lectura sobre el archivo.
boolean canWrite()	Devuelve si el usuario tiene derecho de escritura sobre el archivo.
boolean canExecute()	Devuelve si el usuario tiene derecho de ejecución sobre el archivo.
boolean renameTo(File dest)	Cambia el nombre del archivo al especificado por el objeto destino.
boolean createNewFile()	Crea el archivo especificado por este objeto.
File createTempFile(String prefijo, String sufijo)	Crea un archivo temporal a partir del prefijo y sufijo indicado con el cuerpo del nombre aleatorio.



Funciones de directorio	
boolean isDirectory()	Retorna un valor cierto/falso en función de si el objeto es de tipo directorio o no.
boolean mkdir()	Crea el directorio a que hace referencia este objeto.
String[] list()	Devuelve una lista de nombres de objetos de disco que residen en este directorio.
String[] list(FileFilter filter)	Como el anterior, pero aplicando un filtro sobre los archivos.
File[] listFiles()	Hace la lista de objetos en forma de objetos File y no String .
File[] listFiles(FileFilter filter)	Como antes, aplicando un filtro sobre los archivos.



Como muestra de uso de estas funciones, aquí tenemos un pequeño programa que demuestra el uso del filtro para obtener una lista de archivos que cumplan un patrón concreto de nombre en forma de expresión regular:

```
public class Filtrar {  
  
    public static void main(String[] args) {  
  
        // Montamos un File a partir del directorio actual  
        File f = new File(".");  
        if (!f.isDirectory()) {  
            System.err.println("No estamos en un directorio!");  
            System.exit(-1);  
        }  
  
        // Construimos un filtro para obtener archivos con  
        // extensión .java  
        RegExpFileFilter filt = new FileFilter() {  
            @Override public boolean accept(File arg0) {  
                return (arg0.getName().matches(".*\.\.java$"));  
            }  
        };  
  
        // Obtenemos la lista de archivos contenidos en el  
        // directorio y que cumplan el filtro creado  
        File[] lista = f.listFiles(filt);  
  
        // Recorremos los archivos obtenidos y mostramos su  
        // nombre y tamaño  
        // Para los directorios se añade un carácter  
        // "/" al final  
        for (int i=0; i<lista.length; i++) {  
            File item = lista[i];  
            String text = item.getName();  
            if (item.isDirectory()) text += "/";  
            System.out.printf("%10d ", item.length());  
            System.out.println(text);  
        }  
    }  
}
```

VERSIÓN

El siguiente ejemplo detecta la existencia de un fichero:

```
import java.io.File;
import java.io.IOException;

// Ejemplo para detectar la existencia de un fichero

public class Ejemplo01 {

    public static void main(String[] args) throws IOException {
        File ruta = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01");
        File f = new File(ruta, "datos.txt");
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getParent());
        System.out.println(ruta.getAbsolutePath());
        System.out.println(ruta.getParent());
        if (!f.exists()) { //se comprueba si el fichero existe o no
            System.out.println("Fichero " + f.getName() + " no existe");
            if (!ruta.exists()) { //se comprueba si la ruta existe o no
                System.out.println("El directorio " + ruta.getName() + " no existe");
                if (ruta.mkdir()) { //se crea la ruta. Si se ha creado correctamente
                    System.out.println("Directorio creado");
                    if (f.createNewFile()) { //se crea el fichero. Si se ha creado correctamente
                        System.out.println("Fichero " + f.getName() + " creado");
                    } else {
                        System.out.println("No se ha podido crear " + f.getName());
                    }
                } else {
                    System.out.println("No se ha podido crear " + ruta.getName());
                }
            } else //si la ruta existe creamos el fichero
            if (f.createNewFile()) {
                System.out.println("Fichero " + f.getName() + " creado");
            } else {
                System.out.println("No se ha podido crear " + f.getName());
            }
        } else { //el fichero existe. Mostramos el tamaño
            System.out.println("Fichero " + f.getName() + " existe");
            System.out.println("Tamaño " + f.length() + " bytes");
        }
    }
}
```

VERSIÓN IMPRIMIR

El siguiente ejemplo elimina y cambia el nombre de un fichero:

```
import java.io.File;
import java.util.Scanner;

// Ejemplo para eliminar y cambiar el nombre de un fichero

public class Ejemplo02 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String nombre;

        //Eliminar un fichero
        System.out.println("Introduce el nombre del fichero a eliminar: ");
        nombre = sc.nextLine();
        File ruta = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01");
        File f = new File(ruta, nombre);
        if (f.exists()) {
            System.out.println(f.getAbsolutePath());
            if (f.delete()) {
                System.out.println("Fichero eliminado");
            } else {
                System.out.println("No se ha podido eliminar");
            }
        } else {
            System.out.println("El fichero " + f.getAbsolutePath() + " no existe");
        }

        //Cambiar el nombre de un fichero
        String nuevoNombre;
        System.out.println("Introduce el nombre del fichero a renombrar: ");
        nombre = sc.nextLine();
        File f1 = new File(ruta, nombre);
        if (f1.exists()) {
            System.out.println(f1.getAbsolutePath());
            System.out.println("Introduce nuevo nombre: ");
            nuevoNombre = sc.nextLine();
            File f2 = new File(ruta, nuevoNombre);
            if (f1.renameTo(f2)) {
                System.out.println("Se ha cambiado el nombre");
            } else {
                System.out.println("No se ha podido cambiar el nombre");
            }
        } else {
            System.out.println("El fichero " + f1.getAbsolutePath() + " no existe");
        }
    }
}
```

VERSIÓN IMPRESA



1.3. Familias de clases para el acceso a ficheros

La librería de clases `java.io` contiene multitud de clases relacionadas con el acceso a los datos contenidos en archivos.

Estas clases pueden clasificarse en dos familias: la familia de clases de tipo Stream y la familia de clases de tipo Reader/Writer.

- La primera (con nombres de clase que contienen el texto Stream) son clases definidas en Java 1.0 para obtener métodos de acceso a archivos con contenido interpretado en forma binaria (de byte en byte o en bloque de bytes).
- La segunda (con nombres de clase que contienen el término Reader o Writer), son clases definidas en Java 1.1 para ampliar la base anterior y hacer que los métodos de acceso a archivos ahora interpreten los datos en forma de caracteres con codificación Unicode. Por lo tanto, la lectura y escritura será ahora de carácter en carácter o en bloques de caracteres. Debemos tener en cuenta que un carácter puede ocupar más de un byte.

Como necesitamos realizar operaciones de lectura y escritura sobre los archivos, de cada una de las familias aparecerán las versiones de lectura y de escritura separadamente. Así pues:

	Binario	Carácter
Lectura	InputStream	Reader
Escritura	OutputStream	Writer

Aunque parezca una duplicidad innecesaria, resulta muy práctico como se verá más adelante.

Parece que al ser clases diferentes, también aparecerán métodos distintos, pero no es así. Los métodos presentados por ambas familias son parecidos. Incluso mirando la documentación podremos descubrir que tanto `InputStream` como `Reader` disponen del método `read()` y que `OutputStream` y `Writer` disponen del método `write()`.



La parte más sorprendente y que normalmente acostumbra a desorientar a quien se inicia en Java: de cada una de estas 4 clases base, aparecen multitud de derivadas que aportan alguna mejora o funcionalidad a las de base. De hecho, nadie codifica el acceso a los datos de los archivos con las clases base indicadas hasta ahora.

1.3.1. Clases derivadas de InputStream

Se recogen en la siguiente tabla:

Clase	Descripción
ByteArrayInputStream	Lectura de bloques de bytes desde un array de bytes ubicado en memoria. En el constructor se suministra el buffer del cual se quiere extraer los bytes.
StringBufferInputStream	Considera un String como fuente de datos. En el constructor se suministra el objeto String que será fuente de información. La implementación interna utiliza un StringBuffer.
FileInputStream	Permite el acceso a datos que estén ubicados en un archivo. En el constructor se suministra el nombre del archivo o un objeto de tipo File o FileDescriptor y que represente el archivo.
PipedInputStream	Hace la lectura de datos que han estado escritos en un PipedOutputStream asociado a este. En el constructor se suministrará el objeto PipedOutputStream que se le deseé asociar. Implementa y permite el paso de datos a través de tubería entre dos procesos o threads.

SequenceInputStream	Permite encadenar más de un InputStream en una única secuencia de lectura. En el constructor se suministran los dos InputStream que se quieren encadenar; o bien una enumeración que contenga la lista de InputStream a gestionar.
FilterInputStream	Clase abstracta que permite construir otras funcionalidades en otras clases InputStream.

De FilterInputStream, que es una clase abstracta, se derivan varias clases que son bastante útiles en la práctica. Las más importantes son:

Clase	Descripción
DataInputStream	Permite la lectura de primitivas (int, char, long, etc). Se utiliza en conjunción con DataOutputStream. En el constructor se suministra el objeto InputStream que será fuente de datos para este filtro.
BufferedInputStream	Esta implementación añade el uso de un buffer de lectura para evitar el acceso repetitivo al dispositivo físico. En el constructor se suministra el objeto InputStream que será fuente de datos para este filtro.

¿Por qué tantas clases? Pensemos en la siguiente situación: Tenemos un archivo “datos.dat” que contiene valores enteros de tipo long almacenados uno detrás de otro. El archivo está en un dispositivo de cinta de lectura muy lenta y se nos recomienda utilizar un buffer de lectura para mejorar la eficiencia de acceso. Una solución inteligente será combinar las clases y filtros adecuados:

```
FileInputStream fileIS = new FileInputStream("dades.dat");
BufferedInputStream bufferedIS = new BufferedInputStream(fileIS);
DataInputStream dataIS = new DataInputStream(bufferedIS);
...
long dada = dataIS.readLong();
...
```

O de forma más elegante:

```
DataInputStream in
    = new DataInputStream(
        new BufferedInputStream(
            new FileInputStream("dades.dat")));
...
long dada = in.readLong();
// Método suministrado por DataInputStream
...
```

1.3.2. Clases derivadas de OutputStream

Las clases derivadas de OutputStream son las parejas de sus correspondientes clases derivadas de InputStream, con la excepción de StringBufferInputStream y SequenceInputStream (que no tienen aquí sentido), y podemos resumir así:

Clase	Descripción
ByteArrayOutputStream	Crea un buffer de bytes en memoria. Los datos que se envíen serán alojados en este espacio. En el constructor se suministra el tamaño inicial del buffer de manera opcional.
FileOutputStream	Permite la escritura de datos en archivo. En el constructor se suministra el nombre del archivo o un objeto de tipo File o FileDescriptor y que represente el archivo.
PipedOutputStream	Hace la escritura de datos sobre una tubería que tendrá asociada un PipedInputStream en el otro lado. En el constructor se suministrará el objeto PipedInputStream que se le desee asociar. Implementa y permite el paso de datos a través de tubería entre dos procesos o threads.

FilterOutputStream	Clase abstracta que permite construir otras funcionalidades en otras clases OutputStream.
---------------------------	---

Como antes, de la clase abstracta FilterOutputStream derivan varias clases que son bastante útiles en la práctica. Las más importantes son:

Clase	Descripción
DataOutputStream	Pareja natural de DataInputStream. Permite la escritura de primitivas (int, char, long, etc). En el constructor se suministra el objeto OutputStream que será destino de datos para este filtro.
PrintStream	Sirve para formatear el resultado que se envía de salida. En el constructor se suministra el objeto OutputStream que será destino de datos para este filtro. Si se utiliza, debe ser el último filtro a aplicar.
BufferedOutputStream	Esta implementación añade el uso de un buffer de escritura para evitar el acceso repetitivo al dispositivo físico. Destaca la función flush() para vaciar el buffer temporal en el dispositivo de destino. En el constructor se suministra el objeto OutputStream que será destino de datos para este filtro.

1.3.3. Clases derivadas de Reader

A partir de Java 1.1 se incluyen Reader y sus clases derivadas y que aplican el esquema de métodos de acceso a datos considerando que ahora la unidad de información es el carácter.

Recordemos que un carácter no es necesariamente un byte, porque en Unicode se codifican los caracteres con más de un byte para poder conseguir todo el juego de caracteres y símbolos tipográficos que se utilizan universalmente. Así pues, encontramos codificaciones como UTF-8, ISO-8859-X, UTF-16, CP437, CP850, MacRoman, etc.

Estas clases gestionarán los archivos teniendo en cuenta que una unidad de datos puede ocupar más de un byte.



Clase en Java 1.0	Clase en Java 1.1
InputStream	Reader
FileInputStream	FileReader
StringBufferInputStream	StringReader
ByteArrayInputStream	CharArrayReader
PipedInputStream	PipedReader

Como vemos, existe un equivalente Reader para cada clase base comentada anteriormente con InputStream.

La clase base, Reader, es la base de los otros lectores de caracteres, y presenta como origen de datos diferentes fuentes: un archivo, una cadena String, un array de caracteres y una tubería, respectivamente.

Además, también disponemos de una clase intermedia que permite la conversión de una familia a otra: InputStreamReader nos permite indicar en el constructor un objeto derivado de InputStream y obtener así un Reader equivalente.

También disponemos de filtros:

Clase en Java 1.0	Clase en Java 1.1
FilterInputStream	FilterReader
BufferedInputStream	BufferedReader
DataInputStream	NO EXISTE
LineNumberInputStream	LineNumberReader



1.3.4. Clases derivadas de Writer

Del mismo modo que existen las clases Reader, también disponemos de las correspondientes para escritura de caracteres:

Clase en Java 1.0	Clase en Java 1.1
OutputStream	Writer
FileOutputStream	FileWriter
NO EXISTE EQUIVALENTE	StringWriter
ByteArrayOutputStream	CharArrayWriter
PipedOutputStream	PipedWriter

También disponemos de clase intermedia que permite convertir de una familia a otra: OutputStreamWriter permite indicar en el constructor un objeto derivado de OutputStream y obtener así un Writer.

También disponemos de filtros:

Clase en Java 1.0	Clase en Java 1.1
FilterOutputStream	FilterWriter
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter

1.3.5. Ejemplos de uso de las clases

Ejemplo para escribir en un fichero:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Scanner;

// Ejemplo para escribir en un fichero

public class Ejemplo03 {

    public static void main(String[] args) {
        FileWriter fichero = null;
        PrintWriter pw = null;
        File f = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01", "ejemplo03.txt");
        System.out.println(f.getAbsolutePath());
        try {
            fichero = new FileWriter(f);
            pw = new PrintWriter(fichero);

            for (int i = 0; i < 10; i++) {
                pw.println("Linea " + i);
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                // Nuevamente aprovechamos el finally para
                // asegurarnos que se cierra el fichero.
                if (null != fichero) {
                    fichero.close();
                }
            } catch (Exception e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

Ejemplo para mostrar por pantalla el contenido de un fichero:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

// Ejemplo para mostrar por pantalla el contenido de un fichero

public class Ejemplo04 {

    public static void main(String[] args) {
        File archivo = null;
        FileReader fr = null;
        BufferedReader br = null;

        try {
            // Apertura del fichero y creacion de BufferedReader para poder
            // hacer una lectura comoda (disponer del metodo readLine()).
            archivo = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01", "ejemplo04.txt");
            fr = new FileReader(archivo);
            br = new BufferedReader(fr);

            // Lectura del fichero
            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // En el finally cerramos el fichero, para asegurarnos
            // que se cierra tanto si todo va bien como si salta
            // una excepcion.
            try {
                if (null != fr) {
                    fr.close();
                }
            } catch (Exception e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

Ejemplo para leer el contenido de un fichero y copiarlo en otro fichero:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;

// Ejemplo para leer el contenido de un fichero y copiarlo en otro fichero

public class Ejemplo05 {
    public static void main(String [] arg) {
        File fin = null;
        FileReader fr = null;
        BufferedReader br = null;

        File fout = null;
        FileWriter fw = null;
        PrintWriter pw = null;

        try {
            fin = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01", "ejemplo04.txt");
            fr = new FileReader (fin);
            br = new BufferedReader(fr);

            fout = new File("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01", "ejemplo05.txt");
            fw = new FileWriter(fout);
            pw = new PrintWriter(fw);

            String linea;
            while((linea=br.readLine())!=null){
                pw.println(linea);
            }
        } catch(Exception e){
            e.printStackTrace();
        }finally{
            // En el finally cerramos el fichero, para asegurarnos que se cierra tanto si todo va bien como si salta una excepcion.
            try{
                if( null != fr ){
                    fr.close();
                    fw.close();
                }
            }catch (Exception e2){
                e2.printStackTrace();
            }
        }
    }
}
```

VERSIÓN IMPRIMIBLE



Finalmente, el último ejemplo realiza una operación compleja sobre un archivo de texto. Lee el contenido de dicho archivo línea a línea y genera un archivo de salida donde se añade el número de línea al principio:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Ejemplo06 {
    public static void main(String[] args) {
        // Asignamos los dos archivos de datos (el origen y el destino)
        File fIn = new File("./entrada.txt");
        File fOut = new File("./salida.txt");

        // Comprobamos que exista el archivo de origen
        if (!fIn.exists()) {
            System.err.println("El archivo no existe. Acabamos");
            System.exit(-1);
        }

        try {
            // Definimos los dos canales (entrada y salida)
            BufferedReader in = new BufferedReader(new FileReader(fIn));
            PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(fOut)));

            int nl = 1;
            String linea;

            // Leemos del archivo fuente línea a línea
            while((linea = in.readLine()) != null) {
                // Grabamos la línea con el número de línea añadido
                String lineaOut = nl + " " + linea;
                out.println(lineaOut);
                nl++;
            }

            // Cerramos educadamente los canales antes de acabar
            in.close();
            out.close();
            System.out.println("Programa finalizado");
            System.exit(0);
        }
        catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            System.exit(-1);
        }
    }
}
```

VE



1.4. Persistencia de objetos en ficheros

Demás de ficheros de texto, a veces nos puede interesar almacenar objetos enteros para poderlos recuperar en cualquier momento. Básicamente, disponemos de 3 mecanismos para conseguir que un objeto sea persistente.

- Podemos serializarlo, utilizando herramientas específicas que tiene Java para convertir cualquier objeto en una serie de bytes.
- Podemos implementar algún formato binario específico de conversión de objetos a datos primitivos que permita el almacenamiento y la recuperación de los datos básicos.
- Podemos convertir los objetos persistentes en una jerarquía XML, capaz de almacenar los datos básicos y las relaciones entre objetos.

En este documento solo comentamos la primera. Al ser la opción nativa en Java.

1.4.1. Serialización de objetos

La técnica de serialización es seguramente la más sencilla de todas, pero también la más problemática. Java dispone de un sistema genérico de serialización de cualquier objeto, un sistema recursivo que se repite para cada objeto contenido en la instancia que se está serializando. Este proceso, se para al llegar a los tipos simples, los cuales se almacenan en una serie de bytes. Además de los tipos simples, Java serializa también información adicional o metadatos específicos de cada clase. Gracias a estos metadatos, es posible la automatización de la serialización de forma genérica con garantías de recuperar un objeto tal y como se almacenó. Este es un proceso específico para Java, por lo que no serviría para recuperar objetos serializados desde otros lenguajes de programación.



Para que un objeto pueda ser serializado con esta técnica, la clase a la que pertenece debe implementar la interfaz Serializable. Se trata de una interfaz sin métodos, porque su único objetivo es actuar de marcador para indicar qué clases se pueden serializar y qué clases no.

Todas las clases equivalentes a los tipos simples, ya implementan la interfaz Serializable, junto con la clase String y todos los contenedores y Arrays.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP



1.5. Trabajo con ficheros XML

El lenguaje de marcas XML (eXtended Markup Language) ofrece la posibilidad de representar la información de forma neutra, independiente del lenguaje de programación y del sistema operativo. Su utilidad en el desarrollo de aplicaciones es indiscutible.

Desde un punto de vista de bajo nivel, un documento XML no es otra cosa que un fichero de texto. Si embargo, desde un punto de vista de alto nivel, un documento XML no es un fichero de texto. Su uso intensivo en el desarrollo de aplicaciones hace necesarias herramientas específicas para acceder y manipular este tipo de archivos de manera potente, flexible y fiable. XML nunca hubiera tenido la importancia que tiene en el desarrollo de aplicaciones si permitiera almacenar datos, pero luego no 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 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.

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 quieren 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 de 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.

1.5.1. Acceso a datos con DOM

La tecnología DOM (Document Object Model) es una interfaz de programación que permite analizar y manipular dinámicamente y de forma global e contenido, estilo y estructura de un documento. Tiene su origen en el World Wide Web (W3C).

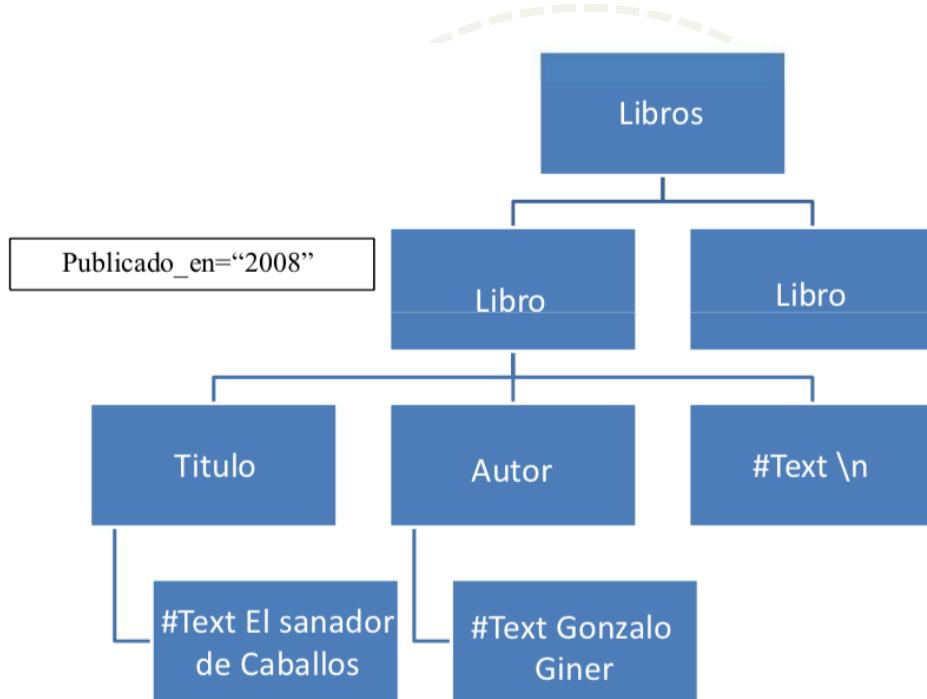
Para trabajar con un documento XML, primero se almacena en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales que son aquellos que no tienen descendientes. Siguiendo este modelo, todas las estructuras de datos del documento se transforman en algún tipo de nodo, y luego esos nodos se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

Ejemplo de documento XML:

```
<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="LibrosESquema.xsd">
    <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>
```

Es un ejemplo de documento que almacena datos de libros. Cada libro está definido por un atributo publicado_en, un texto que indica el año de publicación y por dos elementos hijo: Título y Autor.

Ejemplo DOM del documento XML anterior:



Una vez creada esta estructura en memoria, los métodos 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 generación del árbol DOM a partir del documento XML, se hace de la siguiente forma:

1. Aunque el documento XML tenga asociado un esquema, la librería DOM no necesita hacer la validación, a no ser que se diga lo contrario. Solo tiene en cuenta que el documento esté bien formado.
2. El primero nodo que se crea es el nodo Libros que representa al elemento <Libros>
3. De <Libros> cuelgan tres hijos <Libro> de tipo elemento, por lo tanto el árbol crea 3 nodos Libro descendientes de Libros.

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 parejas (nombre, valor) como atributos haya en el elemento. En ese caso, solo hay un atributo para 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.
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 tipo #text.

Un error muy común cuando se empieza a trabajar con árboles DOM es pensar que, por ejemplo, el valor del nodo Titulo 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 ejemplo anterior, el documento queda mucho más claro y legible así que si se pone todo en una única línea sin espacios entre las etiquetas.

DOM no distingue cuando un espacio en blanco o un salto de línea 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 forma, creando un nodo de tipo #text y poniéndole el valor dentro de ese nodo.



Eso es lo que ocurre en la imagen del árbol DOM anterior, el nodo #text que desciende de Libro y tiene como valor “\n” es creado por DOM ya que, por estética, se ha hecho un salto de línea en el documento XML, para diferenciar que la etiqueta <Titulo> es descendiente de <Libro>.

8. Por último, un documento XML, tiene muchas más “cosas” que las mostradas en el ejemplo anterior, como pueden ser comentarios, encabezados, espacios en blanco, etc. 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.

1.5.1.1. DOM y Java

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

DOM al ser una propuesta de 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.

Java y DOM han evolucionado mucho desde que aparecieron. Para resumir, actualmente la propuesta principal se reduce al uso de JAXP (Java API for XML Processing). A través de JAXP, los diferentes parsers garantizan la interoperabilidad de Java. JAXP está incluido en todo JDK.

Una aplicación que desea manipular documentos XML, accede a la interfaz JAXP porque le ofrece una forma transparente de utilizar los diferentes parsers que hay que gestionar XML.



En los ejemplos de este documento, 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 las clases DocumentBuilder y DocumentBuilderFactory.
- Org.w3c.dom.* , que representa el modelo DOM según la W3C.

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

- setIgnoringComments(boolean ignore)
Sirve para ignorar los comentarios que tenga el documento XML.
- setIgnoringElementContentWhitespace(boolean ignore)
Es útil para eliminar espacios en blanco que no tienen significado.
- setNamespaceAware(boolean aware)
Usado para interpretar el documento usando el espacio de nombres.
- setValidating(boolean validate)
Para que valide 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:

- Todos los nodos contienen los métodos getChild() y getNextSibling() que permiten obtener uno a uno los nodos descendientes 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 asociadas son especialmente importantes a la hora de recorren el árbol ya que permiten ignorar aquellos tipos que no interesan.
- El método `getAttributes()`, devuelve un objeto `NamedNodeMap` (una lista de atributos) si el nodo es de 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 (como 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 éste sobre el que hay que aplicar el método `getNodeValue()` para obtener el título.

1.5.1.2. Abrir DOM desde Java

Para abrir un documento XML desde Java crear un árbol DOM, se utilizan las clases `DocumentBuilderFactory`, `DocumentBuilder` y `Document`. La clase `Document` representa un documento DOM.

```
public int abrir_XML_DOM(File fichero) {
    doc = null;
    try {
        //Se crea un objeto DocumentBuiderFactory.
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        //Indica que el modelo DOM no debe contemplar los comentarios que tenga el XML. Es decir, cuando se convierte
        //el XML al modelo DOM los comentarios que tenga serán ignorados.
        factory.setIgnoringComments(true);
        //Ignora los espacios en blanco. Si no se hace esto entonces los espacios en blanco aparecen como nodos.
        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;
    }
}
```



Siguiendo los comentarios del propio código, se puede entender que hace el método. Primero se crea el Factory y se prepara el parser para interpretar un fichero XML, en el cual ni los espacios en blanco ni los comentarios se tienen en cuenta. El método parse() de DocumentBuilder, recibe como entrada un objeto 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.

1.5.1.3. Recorrer un árbol DOM

Para recorrer un árbol DOM, se utilizan las clases Node y NodeList. En los ejemplos de este apartado, se ha recorrido el árbol DOM creado del documento XML anterior. El resultado de procesar este documento es:

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

```

public String recorrerDOMMyMostrar() {
    String datos_nodo[] = null;
    String salida = "";
    Node node;
    //Obtiene el primero nodo del DOM
    Node raiz = doc.getFirstChild();
    //Obtiene una lista de nodos con todos los nodos hijo.
    NodeList nodelist = raiz.getChildNodes();
    for (int i = 0; i < nodelist.getLength(); i++) //Proceso los nodos hijo
    {
        node = nodelist.item(i);
        //Al ejecutar paso a paso este código, se observa como los nodos que encuentra son
        //de tipo 1 (ELEMENT_NODE) y tipo 3 (TEXT_NODE). Esto es porque en DOM todo elemento tiene un nodo
        //hijo TEXT aunque esté en blanco.
        if (node.getNodeType() == Node.ELEMENT_NODE) { //Es un nodo libro que hay que procesar si es de tipo Elemento
            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;
}

```

Este código, partiendo del objeto Document (doc), que contiene el árbol DOM, recorre dicho árbol para sacar los valores del atributo <Libro> y de los elementos <Titulo> y <Autor>. En DOM, tenemos que comprobar el tipo del nodo que se está trabajando en cada momento, ya que DOM tiene muchos tipos de nodos que no siempre tienen información relevante para la aplicación. En el código del ejemplo, solo se utiliza la información de los nodos de tipo Elemento (ELEMENT_NODE) y de tipo #Text (TEXT_NODE).

El último trozo de código que nos queda por comentar, es el del método ProcesarLibro:

```

protected String[] procesarLibro(Node n) {
    String datos[] = new String[3];
    Node ntemp = null;
    int contador = 1;
    //De la lista de atributos que tiene el nodo selecciona el primero (en nuestro ejemplo solo hay un atributo)
    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);
        //Se debe comprobar el tipo de nodo que se quiere tratar por que posible que haya
        //nodos tipo TEXT que contengan retornos de carro al cambiar de línea en el XML.
        //En este ejemplo, cuando detecta un nodo que no es tipo ELEMENT_NODE es porque es tipo TEXT
        //y contiene los retornos de carro "\n" que se incluyen en el fichero de texto al crear el XML.
        if (ntemp.getNodeType() == Node.ELEMENT_NODE) {
            //IMPORTANTE: para obtener el texto con el título accede al nodo TEXT hijo de ntemp y saca su valor.
            datos[contador] = ntemp.getChildNodes().item(0).getNodeValue();
            contador++;
        }
    }
    return datos;
}

```



En este código, podemos destacar:

- Al conocer la estructura del documento XML, sabemos que el elemento <Libro> solo tiene un atributo, por lo que accedemos directamente a su valor
`(n.getAttributes().item(0).getNodeValue()).`
- Una vez detectado que estamos en el nodo de tipo Elemento (que puede ser el título o el autor), obtenemos su hijo (tipo #text) y consultamos su valor
`(ntemp.getChildNodes().item(0).getNodeValue()).`

1.5.1.4. Modificar y serializar

Además de recorrer un árbol DOM en modo “solo lectura”, como hacen los ejemplos anteriores, éste también puede ser modificado y guardado en un fichero y hacerlo persistente. En los siguientes ejemplos, tenemos el código para añadir un nuevo elemento al árbol DOM y luego guardarlo todo en un documento XML.

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

```
public int añadirDOM(String titulo, String autor, String anno) {  
    try {  
        //Se crean los nodos en la estructura DOM apuntada por la variable doc.  
        //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 para 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 un nodo tipo texto con un retorno de carro (\n) para que al abrirlo con  
        //un editor de texto cada nodo salga en un linea diferente.  
        nlibro.appendChild(doc.createTextNode("\n"));  
        //Se añade a libro el nodo título  
        nlibro.appendChild(ntitulo);  
        //Se añade también un nodo retorno de carro \n  
        nlibro.appendChild(doc.createTextNode("\n"));  
        //Se añade a libro el nodo autor.  
        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;  
    }  
}
```

Para añadir nuevos nodos a un árbol DOM, tenemos que conocer cómo de diferente es el modelo DOM respecto al documento XML. La prueba es la necesidad de crear nodos de texto que sean hijos de los nodos de tipo Elemento para almacenar valores XML.

Después de modificar en memoria un árbol DOM, éste puede guardarse en un fichero y hacerlo persistente. Esto puede hacerse de varias formas, una de ellas es usando las clases `XMLSerializer` y `OutputFormat`. Estas clases se encargan de serializar un documento XML.



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

La clase `XMLSerializer` se encarga de serializar un árbol DOM y guardarlo en un fichero XML bien formado. Esta parte, la realiza la clase `OutputFormat` porque permite asignar diferentes propiedades de formato al fichero XML de salida, como que esté indentado, es decir, que los elementos hijos aparezcan con una 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("/Volumes/Dades/linkiaFP-MAC/DAM-M06/Clase01", "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);

        // Se muestra el contenido del archivo por pantalla
        /* DOMImplementationRegistry registry = DOMImplementationRegistry.newInstance();
        DOMImplementationLS impl = (DOMImplementationLS) registry.getDOMImplementation("XML 3.0 LS 3.0");
        if (impl == null) {
            System.out.println("No DOMImplementation found !");
            System.exit(0);
        }

        System.out.printf("DOMImplementationLS: %s\n", impl.getClass().getName());

        LSParser parser = impl.createLSParser(
            DOMImplementationLS.MODE_SYNCHRONOUS,
            "http://www.w3.org/TR/REC-xml");
        // http://www.w3.org/2001/XMLSchema
        System.out.printf("LSParser: %s\n", parser.getClass().getName());
        LSSerializer lsSerializer = impl.createLSSerializer();
        LSSOutput output = impl.createLSSOutput();
        output.setEncoding("UTF-8");
        output.setByteStream(System.out);
        lsSerializer.write(doc, output);*/
        return 0;
    } catch (Exception e) {
        return -1;
    }
}
```



Mirando el código de este último ejemplo, para serializar un árbol DOM necesitamos:

- Un objeto File que representa al fichero resultado.
- Un objeto OutputFormat que permite indicar pautas de formato para la salida.
- Un objeto XMLSerializer que se crea con el objeto File de salida y el formato marcado por OutputFormat.
- Un método serialize() de XMLSerializer que recibe como parámetro el Document (doc) que quiere llevar a fichero y lo escribe.

1.5.2. Acceso a datos con SAX

SAX (Simple API for XML) es una tecnología para poder acceder a XML desde lenguajes de programación. SAX tiene el mismo objetivo que DOM, pero desde un enfoque diferente. Normalmente usamos SAX cuando la información guardada en los documentos XML es clara, está bien estructurada y no se necesitan hacer modificaciones.

Las características de SAX son:

- SAX ofrece una alternativa para leer documentos XML de forma secuencial. El documento solo se lee una vez. A diferencia de DOM, no nos podemos mover por el documento como queramos. Al abrir el documento, éste se recorre de forma secuencial hasta llegar al final. Cuando llega al final, finaliza el proceso.
- SAX, a diferencia de DOM, no carga el documento en memoria, sino que lo lee directamente del fichero. Eso lo hace especialmente útil cuando el fichero es muy grande.

SAX sigue los siguientes pasos:

1. Le decimos al parser de SAX qué fichero queremos leer de forma secuencial.
2. El documento XML se traduce a una serie de eventos.
3. Los eventos generados, los podemos controlar con métodos de control llamados callbacks.
4. Para implementar los callbacks, es necesario implementar la interfaz ContentHandler o su implementación por defecto, la clase DefaultHandler.

El proceso se puede resumir de la siguiente forma:

- SAX abre un fichero XML y ubica un puntero al inicio del documento.
- Cuando empieza a leer el fichero, el puntero avanza de forma secuencial.
- Cuando SAX detecta un elemento propio de XML, genera un evento. Un evento puede deberse a:
 - Inicio del documento
 - Final del documento
 - Etiqueta de inicio de un elemento, como por ejemplo <Libro>
 - Etiqueta de final de un elemento, como por ejemplo </Libro>
 - Atributo
 - Cadena de caracteres que puede ser un texto
 - Error (en el documento, I/O, etc.)
- Cuando SAX genera un evento, éste puede ser manejado con la clase DefaultHandler (callbacks). Esta clase puede ser extendida y sus métodos sobreescritos para realizar la acción que queramos. Por ejemplo, se puede sobreescribir el método startElement() (que es el método que se llama cuando se genera un evento de inicio de elemento), de forma que compruebe el nombre del nuevo documento, y si es un nombre determinado, muestre por pantalla un mensaje.
- Cuando SAX genera un evento de error o final de documento, finaliza el proceso.



1.5.2.1. Abrir XML con SAX desde Java

Para abrir un documento XML desde Java con SAX, utilizamos las clases SAXParserFactory, SAXParser y DefaultHandler. Además, también será necesaria la clase File, para poder indicar el fichero XML con el que vamos a trabajar.

- Las clases SAXParserFactory y SAXParser, nos proporcionan el acceso desde JAXP.
- La clase DefaultHandler, es la clase base que da respuesta a los eventos generados por el parser. Nuestra aplicación extenderá esta clase para personalizar la respuesta del parser ante la generación de los diferentes eventos.

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

En el código del ejemplo, vemos que primero creamos los objetos factory y parser. Esta parte es parecida a como se hace en DOM. Una diferencia es que en SAX, creamos un objeto de la clase ManejadorSAX. Esta clase es la que extiende la clase DefaultHandler y sobreescribe los métodos callbacks. De forma resumida, la preparación del proceso que realiza SAX, requiere inicializar las siguientes variables, que son las que se utilizarán cuando se inicie el proceso de recorrido del fichero XML:

- Un objeto parser: en el código la variable llamada parser.
- Un objeto de la clase que extiende a DefaultHandler, que en el ejemplo se llama ManejadorSAX. La variable se llama sh.



1.5.2.2. Recorrer XML con SAX

Para recorrer un documento XML una vez inicializado el parser, lo único que necesitamos es lanzar el parser. No es necesario haber creado la clase que extiende a DefaultHandler. Esta clase tiene la lógica a ejecutar para cada uno de los eventos generados a los que queremos dar respuesta.

```
class ManejadorSAX extends DefaultHandler {  
    int ultimoelement;  
    //String publicado_en,titulo,autor;  
    String cadena_resultado = "";  
  
    public ManejadorSAX() {  
        ultimoelement = 0;  
    }  
  
    //A continuación se sobrecargan los eventos de la clase DefaultHandler para recuperar el documento XML.  
    //En la implementación de estos eventos se indica qué se hace cuando se encuentre el comienzo de un elemento(startElement),  
    //el final de un elemento(endElement) y caracteres texto(characters)  
    //Este handler detecta comienzo de un elemento, final y cadenas string(texto).  
    @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 un linea discontinua en la salida.  
    @Override  
    public void endElement(String uri, String localName, String qName) throws SAXException {  
        if (qName.equals("Libro")) {  
            System.out.println("He encontrado el final de un elemento, pero en este ejemplo no hago nada");  
            cadena_resultado = cadena_resultado + "\n -----";  
        }  
    }  
  
    //Cuando se detecta una cadena de texto posterior a uno de los elementos <titulo> o <autor> entonces guarda  
    //ese texto en la variable correspondiente.  
    @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];  
            }  
        }  
    }  
}
```

En este código de ejemplo, creamos una clase que extiende a DefaultHandler y sobreescribe los métodos startElement, endElement e characters, que son los callbacks que queremos que se ejecuten como respuesta a los eventos generados, que son inicio de elemento, final de elemento y cadena de caracteres respectivamente.



En el ejemplo, cada uno de los métodos realiza lo siguiente:

- **startElement():**

Cuando se detecta un evento de inicio de elemento, SAX llama a este método. Lo que hace es comprobar de qué tipo de elemento se trata.

- Si es <Libro>, obtiene el valor de su atributo y lo concatena a una cadena (cadena_resultado) que tendrá la salida final que se mostrará por pantalla.
- Si es <Titulo>, se concatena a la cadena de salida el texto “El título es:”.
- Si es <Autor>, se concatena a la cadena de salida el texto “El autor es:”.
- Si es otro tipo de elemento, no hará nada.

- **endElement():**

Cuando se detecta un evento de final de elemento, SAX llama a este método. Lo que hace es comprobar si es el final de un elemento <Libro>. Si es así, concatena a la cadena de salida el texto “\n-----”

- **characters():**

Cuando se detecta un evento de detección de cadena de texto, SAX llama a este método. Lo que hace es concatenar a la cadena de salida cada uno de los caracteres de la cadena detectada.



La salida del ejemplo anterior, aplicado al documento XML del ejemplo inicial de este documento, será la siguiente:

¡Ya se ha creado el SAX!

He encontrado el final de un elemento, pero en este ejemplo no hago nada
He encontrado el final de un elemento, pero en este ejemplo no hago nada
He encontrado el final de un elemento, pero en este ejemplo no hago nada

MOSTRAMOS LOS DATOS DE LA ESTRUCTURA 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

En el siguiente ejemplo, se lanza el parser SAX para obtener el resultado de la imagen anterior.

```
public String recorrerSAXyMostrar() {  
    //Se da la salida al parser para que comience a manejar el documento XML que se le pasa como parámetro  
    //con el manejador que también se le pasa. Esto recorrera secuencialmente el documento XML y cuando detecte  
    //un comienzo o fin de elemento o un texto entonces lo tratará (según la implementación hecha del manejador)  
    try {  
        parser.parse(ficheroXML, 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 utiliza el parser inicializado (parser), el objeto de la clase que gestionará los eventos (sh) y el File con el fichero XML que recorremos (ficheroXML). El método parse() lanza SAX para el fichero XML seleccionado y con el manejador deseado. Se pueden implementar tantas extensiones de DefaultHandler como manejadores diferentes queramos utilizar). La excepción que de captura es SAXException.

1.5.3. Acceso a datos con JAXB

De las opciones que existen para acceder a un documento XML desde Java, JAXB (Java Architecture for XML Binding) es la más potente.

No tenemos que confundir JAXB con la interfaz de acceso JAXP, que es una librería de un-marshalling. La serialización o marshalling, es el proceso de guardar un conjunto de objetos en un fichero. Justo el proceso contrario, un-marshalling, es convertir en objetos el contenido de un fichero.

De forma 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, para que así el contenido del XML será válido según ese esquema.
- JAXB crea la estructura de clases que guardará el contenido XML en base a su esquema. El esquema es la referencia de JAXB para saber la estructura de clases que contendrá el documento XML.
- JAXB crea la estructura de clases en tiempo de diseño, después el proceso de un-marshalling y marshalling es sencillo y rápido, y se puede hacer en tiempo de ejecución.

El siguiente esquema XML, se corresponde con el documento XML que hemos estado tratando en los apartados anteriores. El esquema XML indica que existe un elemento <Libros> que contiene uno o varios elementos <Libro>. Cada elemento <Libro> tiene un atributo publicado_en cuyo valor tiene que ser de tipo String,y dos elementos hijos: <Titulo> y <Autor>.

```
<?xml version="1.0"?>

<xsd:schema version="1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <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>
```

VERSIÓN IMPRIMIBLE

JAXB es capaz de obtener de este esquema una estructura de clases que le dé soporte en Java. Siguiendo el ejemplo, 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;  
        }  
    }  
}
```

En el proceso de un-marshalling, JAXB carga el contenido de cualquier documento XML, que sea válido respecto al esquema, en una estructura de objetos de las clases Libros y Libro. En el proceso contrario, marshalling, JAXB convierte una estructura de objetos de las clases Libros y Libro, en un documento XML válido con respecto del esquema.



1.5.3.1. Crear las clases Java desde un esquema XML

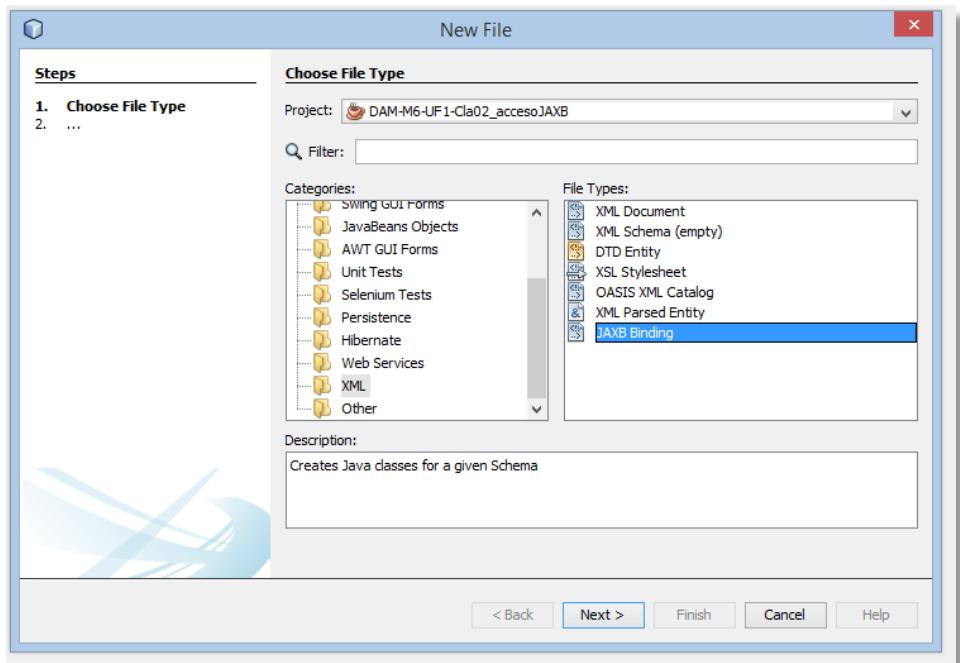
El punto de partida es un esquema XML y el proceso para crear la estructura de clases Java es muy sencillo, ya sea directamente desde un JDK en Java o usando el IDE NetBeans.

Con un JDK, podemos 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 ejecución. Por ejemplo, suponiendo que el fichero que contiene el código del esquema anterior se llama librosEsquema.xsd, la creación de las clases la haríamos con el siguiente comando:

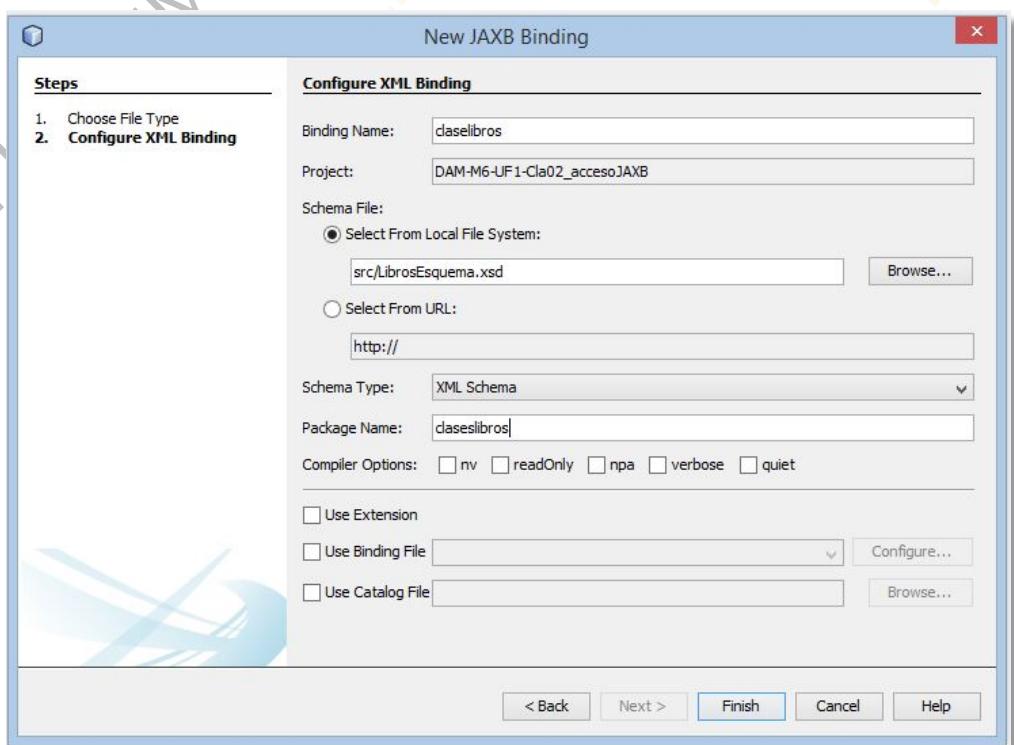
```
xjc librosEsquema.xsd
```

Con el IDE NetBeans, podemos obtener las clases asociadas a un esquema XML. Para ello, tenemos 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, tenemos que seleccionar el tipo de fichero que queremos añadir XML->JAXB Binding.

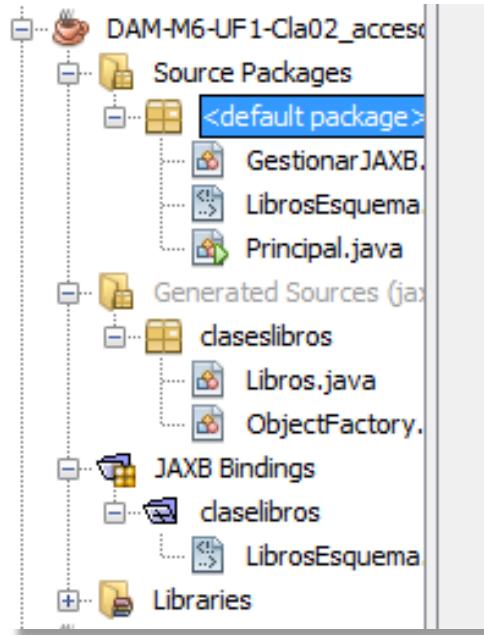


3. En la siguiente pantalla, configuraremos una serie de campos que definirán el tipo de enlace (binding) que realizaremos. Los campos más importantes son la localización del esquema XML sobre el que crearemos las clases Java (llamado librosEsquema.xsd) y el paquete en el que queremos que se creen las nuevas clases (llamado claseslibros).





4. Una vez hemos rellenado estos datos, se crea una nueva carpeta en el árbol del proyecto con el paquete (claseslibros) y dentro las clases que se enlazan con el esquema XML (librosEsquema.xsd).



1.5.3.2. Abrir XML con JAXB

Un documento XML con JAXB se abre utilizando las clases JAXBContext y Unmarshaller.

- La clase JAXBContext crea un contexto con un nuevo objeto de JAXB para la clase principal obtenida del esquema XML.
- La clase Unmarshaller se utiliza para obtener del documento XML los datos que son guardados en la estructura de objetos en 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.

```

Libros misLibros;
public int abrir_XML_JAXB(File fichero)
{
    JAXBContext contexto;
    try {
        //Crea una instancia JAXB
        contexto = JAXBContext.newInstance(Libros.class);
        //Crea un objeto Unmarshaller.
        Unmarshaller u=contexto.createUnmarshaller();
        //Deserializa (unmarshal) el fichero y crea el arbol de objetos
        misLibros=(Libros) u.unmarshal(fichero);

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

```

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 queremos abrir. El objeto misLibros de tipo Libros tendrá los libros obtenidos del fichero.

1.5.3.3. Recorrer un XML desde JAXB

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

```

public String recorrerJAXBMostrar(){
    String datos_nodo[]={};
    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 podemos ver en el código, no usamos 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 los ejemplos de DOM y SAX.

MOSTRAMOS LOS DATOS DE LA ESTRUCTURA JAXB:

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

1.5.3.4. Guardar un XML con JAXB

Para guardar en un documento XML una estructura de objetos Java con JAXB, utilizamos las clases JAXBContext y Marshaller.

- La clase JAXBContext crea un contexto con un nuevo objeto de JAXB para la clase principal obtenida del esquema XML.
- La clase Marshaller se utiliza para guardar en un documento XML una estructura de objetos Java que siguen la estructura de clases creadas a partir de la clase principal del esquema XML en el que se basará el documento XML que estamos creando. Esta clase tiene métodos para establecer propiedades al fichero donde vamos a guardar los objetos de la estructura Java, como por ejemplo que el fichero esté indentado.

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

```
// Creamos el contexto JAXBContext para nuestra clase modulos
JAXBContext contexto = JAXBContext.newInstance(Modulos.class);
// Declaramos el serializador
Marshaller m = contexto.createMarshaller();
// Fichero que vamos a generar
File f = new File("modulos.xml");
// Con esta propiedad hacemos que escriba el texto con formato xml, en vez de todo en una linea
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
// Escribimos el xml en el fichero
m.marshal(modulos, f);
```

Siguiendo los comentarios del código, primero creamos el objeto que representa al contexto JAXB basándonos en la clase principal de la estructura que contiene los objetos que queremos guardar. Después, creamos el objeto marshaller que será encargado de guardar los objetos en el fichero. A este objeto, le establecemos la propiedad para que se muestre intentado (Marshaller.JAXB_FORMATTED_OUTPUT). Finalmente, el método Marshall se encarga de hacer el guardado de los objetos utilizando el fichero donde los vamos a guardar y el objeto que representa la raíz de esa estructura de objetos que queremos guardar.

En este caso, la estructura de clases creadas en Java y que se ha utilizado para crear la estructura de objetos que después guardamos en el archivo, se basa en el siguiente esquema XML:

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsi:element name="modulos">
<xs:complexType>
<xs:sequence>
<xsi:element name="modulo" maxOccurs="unbounded" minOccurs="0">
<xs:complexType>
<xs:sequence>
<xsi:element name="alumno" maxOccurs="unbounded" minOccurs="0">
<xs:complexType>
<xs:sequence>
<xsi:element type="xs:string" name="nombre"/>
<xsi:element type="xs:float" name="UF1"/>
<xsi:element type="xs:float" name="UF2"/>
<xsi:element type="xs:float" name="UF3"/>
</xs:sequence>
</xs:complexType>
</xsi:element>
</xs:sequence>
<xsi:attribute type="xs:string" name="m" use="optional"/>
</xs:complexType>
</xsi:element>
</xs:sequence>
</xs:complexType>
</xsi:element>

```



Un ejemplo que cómo crear esa estructura de objetos Java para después guardarlos, podría ser el siguiente:

```
Modulos modulos = new Modulos();  
  
float uf1 = (float)1.0;  
float uf2 = (float)2.0;  
float uf3 = (float)3.0;  
  
Alumno alumno = new Alumno();  
alumno.setUF1(uf1);  
alumno.setUF2(uf2);  
alumno.setUF3(uf3);  
  
Alumno alumno2 = new Alumno();  
alumno2.setUF1(uf1+1);  
alumno2.setUF2(uf2+2);  
alumno2.setUF3(uf3+3);  
  
Modulo modulo = new Modulo();  
modulo.getAlumno().add(alumno);  
modulo.getAlumno().add(alumno2);  
modulo.setM("M01");  
  
Modulo modulo1 = new Modulo();  
modulo1.getAlumno().add(alumno2);  
modulo1.getAlumno().add(alumno);  
modulo1.setM("M02");  
  
modulos.getModulo().add(modulo);  
modulos.getModulo().add(modulo1);
```

Aunque también podemos crear estructura a partir de datos que leemos de un documento XML, tal y como hemos visto en el punto “Abrir XML con JAXB”.



1.6. Procesamiento de XML: XPATH

La opción más sencilla para consultar la información dentro de un documento XML es mediante el uso de XPath (XML Path Language). Con XPath, podemos seleccionar y hacer referencia a texto, elementos, atributos y cualquier otra información contenida dentro de un fichero XML.

Por otro lado, XQuery es un lenguaje potente para la consulta de XML em bases de datos nativas XML. XQuery está basado en XPath 2.0, por lo que es necesario tener nociones básicas que XPath para entender el funcionamiento de XQuery.

1.6.1. Empezando con XPath

XPath empieza 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:

- (./) 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 empezar con XPath, es con ejemplos. Para ello, vamos a basarnos en el siguiente documento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Base de datos de libros en Castellano -->
<Libros>
    <Libro publicado_en="1840">
        <Autor>Nikolai Gogol</Autor>
        <Titulo>El Capote</Titulo>

    </Libro>
    <Libro publicado_en="2008">
        <Autor>Gonzalo Giner</Autor>
        <Titulo>El Sanador de Caballos</Titulo>

    </Libro>
    <Libro publicado_en="1981">
        <Autor>Umberto Eco</Autor>
        <Titulo>El Nombre de la Rosa</Titulo>
    </Libro>
</Libros>
```

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

/Libros/Libro/Autor

Si queremos obtener todos los elementos Autor, la expresión sería la siguiente:

//Autor

También podemos utilizar comodines en cualquier nivel del árbol. La siguiente expresión selecciona todos los nodos Autor que son nietos de Libro:

/Libros/*/*/Autor

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

Para identificar un conjunto de atributos, utilizaremos el carácter @ delante del nombre del atributo. La siguiente expresión selecciona todos los atributos `publicado_en` de un elemento Libro:

```
/Libros/Libro/@publicado_en
```

Como en el ejemplo en el que nos estamos basando, solo los elementos Libro tienen un atributo `publicado_en`, la misma consulta la podemos hacer con la siguiente expresión:

```
//@publicado_en
```

También podemos seleccionar múltiples atributos con el operador `@*`. Por ejemplo, para seleccionar todos los atributos del elemento Libro:

```
//Libro/@*
```

Además, XPath, ofrece la posibilidad de hacer expresiones para concretar los nodos buscados dentro del árbol XML. Es una opción parecida a la cláusula WHERE de SQL. Por ejemplo, para encontrar todos los nodos Título con el valor `El Capote`, podemos utilizar la siguiente expresión:

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

Aquí, el operador `[]` marca un filtro y el operador `'.'` establece que ese filtro se 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, también podríamos buscar todos los elementos Libro con Título '`El Capote`'.

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

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

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



1.6.2. XPath desde Java

En Java existen librerías con sus correspondientes clases y métodos, que nos permiten ejecutar consultas XPath sobre documentos XML. Las clases más importantes de estas librerías son:

- **XPathFactory**: el método compile de esta clase, comprueba si la sintaxis de la consulta XPath es correcta y crea una expresión XPath (**XPathExpression**).
- **XPathExpression**: el método evaluate de esta clase, ejecuta una expresión XPath.
- **DocumentBuilderFactory** y **Document**: los métodos de estas clases crean la estructura DOM del documento XML sobre el que se va a ejecutar la consulta XPath.

El siguiente ejemplo muestra la ejecución de una consulta XPath sobre un documento XML. Para ello hemos creado dos métodos:

- **abrir_file_DOM**: abre el documento XML sobre el que se va a ejecutar la consulta y crea el árbol DOM.
- **Ejecutar_XPath**: ejecuta una consulta XPath pasada como parámetro y devuelve el resultado en formato String.

```
XPathExpression exp;
Element element;
Document XMLDoc;
XPath xpath;

public int abrir_file_DOM()
{
    //Abre un fichero XML para crear un DOM
    try {
        //El fichero XML que se abre es LibrosXML.xml
        xpath = XPathFactory.newInstance().newXPath();
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        XMLDoc = factory.newDocumentBuilder().parse(new InputSource(new
        FileInputStream("LibrosXML.xml")));
        //Al llegar aquí ya se ha creado la estructura DOM para ser consultada
        return 0;
    }
    catch (Exception ex) {
        System.out.println("Error: " + ex.toString());
        return -1;
    }
}

public String Ejecutar_XPath(String txtconsulta)
{
    //Ejecuta la consulta txtconsulta y devuelve el resultado como un String.
    String salida="";
    try {
        //Compila la consulta
        exp = xpath.compile(txtconsulta);
        //Evaluate evalua la expresión devuelta por compile y devuelve el resultado
        //(Lista de nodos)
        Object result= exp.evaluate(XMLDoc, XPathConstants.NODESET);
        NodeList nodeList = (NodeList) result;
        for (int i = 0; i < nodeList.getLength(); i++) {
            salida = salida + "\n" + nodeList.item(i).getChildNodes().item(0).getNodeValue();
        }
        return salida;
    }
    catch (Exception ex) {
        return "Error: " + ex.toString();
    }
}
```

VERSIÓN II



Recursos y Enlaces

- [Java](#)



- [API Java 11](#)



- [NetBeans](#)



- [Tutorial de XPath](#)



VERSO



Conceptos clave

- **Java**: lenguaje de programación orientado a objetos.
- **NetBeans**: IDE de programación para diferentes lenguajes.
- **DOM**: Document Object Model. Define un estándar para acceder y manipular documentos.
- **XPath**: expresiones que permiten acceder a contenido de un documento XML.

VERSIÓN IMPRIMIBLE ALUMNO LINKIAFP



Test de autoevaluación

1. Pon el significado de cada una de las siglas de JAXB:
 - a. J
 - b. A
 - c. X
 - d. B

2. ¿Cómo se llama la API que traduce un documento XML a eventos para ser gestionado?
 - a. DOM
 - b. JAXB
 - c. SAX
 - d. XPath

3. ¿Con qué tecnología podemos hacer consultas sobre un documento XML?
 - a. XPath
 - b. DOM
 - c. SAX
 - d. JAXB

VERSIÓN 1

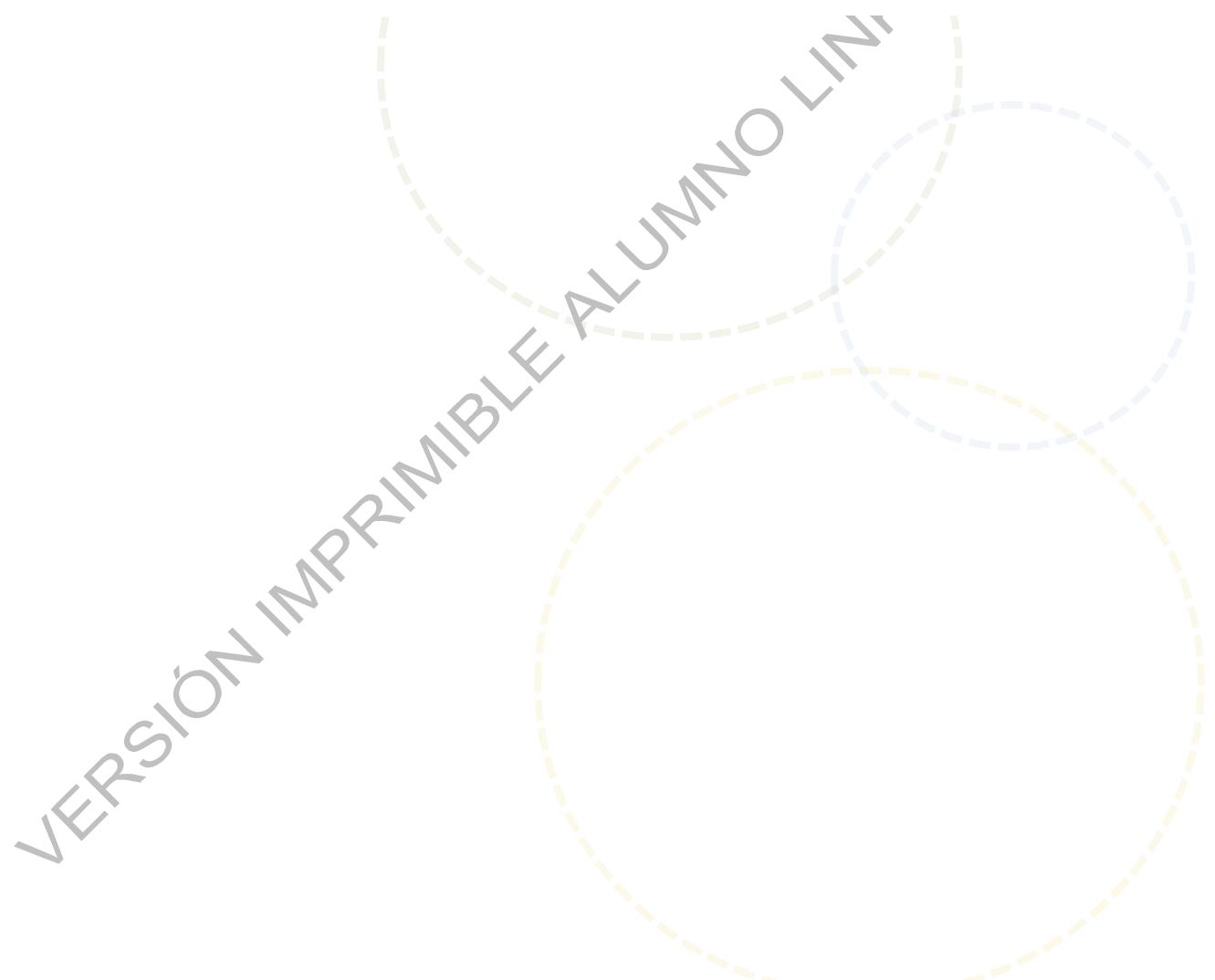


Ponlo en práctica

Actividad 1

Crea un programa en Java que:

- a. Acceda a un documento XML. Este documento XML tendrá el mismo esquema que los ejemplos comentados en este documento.
- b. Realice la siguiente consulta XPath: /Libros/*[Autor
- c. Muestre el resultado por pantalla





SOLUCIONARIOS

Test de autoevaluación

1. Pon el significado de cada una de las siglas de JAXB:
 - a. J – **Java**
 - b. A – **Architectura**
 - c. X – **XML**
 - d. B – **Binding**

2. ¿Cómo se llama la API que traduce un documento XML a eventos para ser gestionado?
 - a. DOM
 - b. JAXB
 - c. **SAX**
 - d. XPath

3. ¿Con qué tecnología podemos hacer consultas sobre un documento XML?
 - a. **XPath**
 - b. DOM
 - c. SAX
 - d. JAXB

VER



Ponlo en práctica

Actividad 1

Crea un programa en Java que:

- a. Acceda a un documento XML. Este documento XML tendrá el mismo esquema que los ejemplos comentados en este documento.
- b. Realice la siguiente consulta XPath: /Libros/*/Autor
- c. Muestre el resultado por pantalla

```
XPathExpression exp;
Element element;
Document XMLDoc;
XPath xpath;

//Abre un fichero XML para crear un DOM
try {
    //El fichero XML que se abre es LibrosXML.xml
    xpath = XPathFactory.newInstance().newXPath();
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    XMLDoc = factory.newDocumentBuilder().parse(new InputSource(new
    FileInputStream("LibrosXML.xml")));
    //Al llegar aquí ya se ha creado la estructura DOM para se consultada

    //Ejecuta la consulta txtconsulta y devuelve el resultado como un String.
    String salida="";
    //Compila la consulta
    exp = xpath.compile("/Libros/*/Autor");
    //Evaluate evalua la expresión devuelta por compile y devuelve el
    resultado (Lista de nodos)
    Object result= exp.evaluate(XMLDoc, XPathConstants.NODESET);
    NodeList nodeList = (NodeList) result;
    for (int i = 0; i < nodeList.getLength(); i++) {
        salida      =      salida      +      "\n";
        nodeList.item(i).getchildNodes().item(0).getNodeValue();
    }
    System.out.println(salida);
} catch (Exception ex) {
    return "Error: " + ex.toString();
}
```

VE