TRABAJO CON FICHEROS XML

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

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.

Un fichero XML sencillo tiene la siguiente estructura:

```
<Alumno>
      <alumno>
            <id>1</id>
            <apellido>GARCIA</apellido>
            <nombre>María</nombre>
            <modulo>BBDD</modulo>
            <nota>5</nota>
            <modulo>Acceso a Datos</modulo>
            <nota>8</nota>
      </alumno >
      <alumno >
            <id>2</id>
            <apellido>Pérez</apellido>
            <nombre>Juan</nombre>
            <modulo>BBDD</modulo>
```

```
            <nota>8</nota>
      </alumno >
      <alumno >
            <id>3</id>
            <apellido>ALVAREZ</apellido>
            <nombre>Marta</nombre>
            <modulo>BBDD</modulo>
            <nota>8</nota>
            <modulo>Programación</modulo>
            <nota>7</nota>
            <modulo>Interfaces</modulo>
            <nota>6</nota>
      </alumno >
</Alumno >
```

Los ficheros XML se pueden utilizar:

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

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un procesador de XML o **parser**. El procesador lee los documentos y proporciona acceso a su contenido y estructura. Algunos de los procesadores más empleados son: **DOM**: *Modelo de Objetos de Documento* y **SAX**: *API Simple para XML*.

- **DOM**: un procesador XML que utilice este planteamiento almacena la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos (de arriba abajo y también se puede volver atrás) y se analiza a qué tipo particular pertenecen. Podemos modificar cualquier nodo del árbol. Tiene su origen en el W3C[1]. Este tipo de procesamiento necesita más recursos de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos.

  Con ficheros XML pequeños no tendremos problemas, pero si tuviéramos un árbol muy muy grande entonces tendríamos una falta de **heap space**[2].

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

[1] El Word Wide Web Consortium (W3C) es una comunidad internacional que desarrolla estándares que aseguran el crecimiento de la Web a largo plazo.
[2] Espacio de memoria que utilizan los objetos.

Al contrario que con DOM, al procesar en SAX no vamos a tener la representación completa del árbol en memoria, pues SAX funciona con eventos. Esto implica:

- Al no tener el árbol completo no puede volver atrás, pues va leyendo secuencialmente.
- La modificación de un nodo y la inserción de nuevos nodos son mucho más complejas.
- Como no tiene el árbol en memoria es mucho más **memory friendly**, de modo que es la única opción viable para casos de ficheros muy grandes, pero demasiado complejo para ficheros pequeños.
- Al ser orientado a eventos, el procesado se vuelve bastante complejo.

Acceso a Ficheros XML con DOM

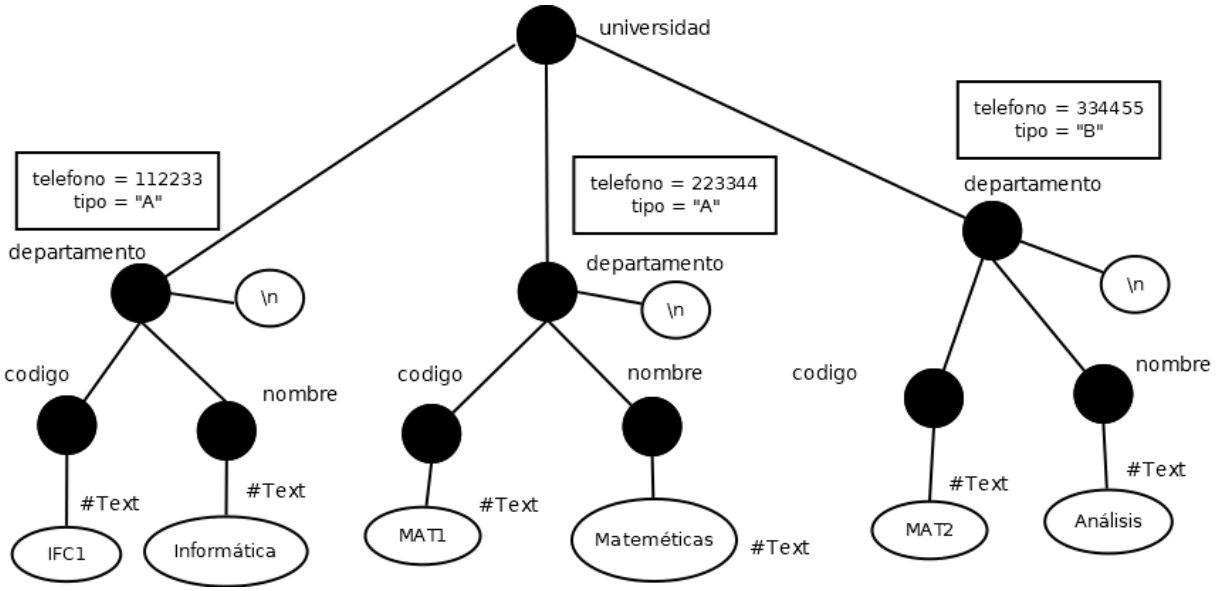
Para poder trabajar con **DOM** en Java necesitamos las clases e interfaces que componen el paquete ***org.w3c.dom*** (contenido en el JSDK) y el paquete ***javax.xml.parsers*** del API estándar de Java que proporciona un par de clases abstractas que toda implementación **DOM** para Java debe extender.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

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

Ejemplo de un documento XML para representar departamentos. Cada departamento está definido por un atributo teléfono , un texto que indica el nº de teléfono, otro atributo tipo con un texto que indica el tipo de departamento y dos elementos hijo: código y nombre.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<universidad>
<espacio xmlns=http://www.misitio.com
xmls:prueba=http://www.misitio.com/pruebas/>
      <!–DEPARTAMENTO 1 -->
      <departamento telefono=”112233” tipo=”A”>
            <codigo>IFC1</codigo>
            <nombre>Informática</nombre>
      </departamento>
      <!–DEPARTAMENTO 2 -->
      <departamento telefono=”223344” tipo=”A”>
            <codigo>MAT1</codigo>
            <nombre>Matemáticas</nombre>
      </departamento>
      <!–DEPARTAMENTO 3 -->
      <departamento telefono=”334455” tipo=”B”>
            <codigo>MAT2</codigo>
            <nombre>Análisis</nombre>
      </departamento>
</universidad>
```

Un esquema del árbol DOM que representaría internamente este documento sería el siguiente:



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

1. 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 *universidad* que representa el elemento <universidad>.
3. De <universidad> cuelgan en el documento tres hijos <departamento> de tipo elemento, por tanto el árbol crea 3 nodos *departamento* descendiente de *universidad*.
4. Cada elemento <departamento> en el documento tiene asociado dos atributos: *teléfono y tipo*. 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 hay dos atributos en el elemento <departamento>.
5. Cada <departamento> tiene dos elementos que descienden de él y que son <codigo> y <nombre>. Al ser elementos, estos son representados en DOM como nodos descendientes de *departamento*, al igual que se ha hecho con *departamento* al descender de *universidad*.
6. Cada elemento <codigo> y <nombre> 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 *codigo* es el texto que contiene el elemento <codigo> 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.

   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 es lo que ha ocurrido en el ejemplo. El nodo #text que desciende de *departamento* 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, para diferenciar claramente que la etiqueta <codigo> es descendiente de <departamento>.

8. Por último, un documento XML tiene muchas más "cosas" que las mostradas en el 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.

Los programas Java que utilicen **DOM** necesitan interfaces, algunas son:

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

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[3]:

- *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.

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

- *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*:[4]

Todos los nodos contienen los métodos *getFirstChild()* y *getNextSibling()* que permiten obtener uno a uno losnodos 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 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, <codigo>) 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 código del departamento y es sobre él sobre el que hay que aplicar el método *getNodeValue()* para obtener el código.

Ejemplo de creación de un fichero XML, en el ejemplo vamos a partir del fichero Personas.txt.

Lo primero que tenemos que hacer es importar los paquetes necesarios:

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

Después crearemos una instancia de ***DocumentBuilderFactory*** para construir el parser (necesario para crear el árbol), se debe encerrar entre **try-catch** porque se puede producir la excepción ***ParserConfigurationException***.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
     try{
          DocumentBuilder db = dbf.newDocumentBuilder();
```

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

```
DOMImplementation implementacion = db.getDOMImplementation();
// crea el documento con el nodo raíz de nombre Personas
Document documento = implementacion.createDocument(null, "Personas",
null);
```

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

```
documento.setXmlVersion("1.0"); // asignamos la versión de nuestro XML
```

El siguiente paso será recorrer el fichero con los datos de las personas y por cada registro crear un nodo persona con 3 hijos (clave, nombre, edad). Cada nodo hijo tendrá su valor (por ejemplo: 1. Isabel, 42). Para crear un elemento usamos el método ***createElement(String)*** llevando como parámetro el nombre que se pone entre las etiquetas menor que y mayor que. El siguiente código crea y añade el nodo <persona> al documento:

```
Element raiz = documento.createElement("persona");    //creamos el nodo persona
documento.getDocumentElement().appendChild(raiz);// lo pegamos a la raíz del documento
```

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

```
CrearElemento("clave", Integer.toString(clave),raiz,documento); // añadir clave
CrearElemento("nombre", nombre.trim(),raiz,documento); // añadir nombre
CrearElemento("edad",       Integer.toString(edad),raiz,documento);// añadir edad
```

La función recibe el nombre del nodo hijo (clave, nombre, edad) y sus textos o valores que tienen que estar en formato String (1. Isabel, 42), el nodo al que se va a añadir (raíz) y el documento (documento). Para crear el nodo hijo (<clave> o <nombre> o <edad>) se escribe:

```
Element elemento = documento.createElement(datoPersona); // creamos hijo
```

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

```
Text texto = documento.createTextNode(valor); // damos valor
```

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

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

Al final se generaría algo similar a esto:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Personas>
	<persona>
		<clave>3</clave>
		<nombre>Beatriz</nombre>
		<edad>25</edad>
	</persona>
	<persona>
		<clave>5</clave>
		<nombre>Isabel</nombre>
		<edad>30</edad>
	</persona>
	<persona>
```

```
            <clave>7</clave>
            <nombre>Iciar</nombre>
            <edad>16</edad>
      </persona>
</Personas>
```

El método para crear el fichero es el siguiente:

```
// metodo de insercion de los datos de la persona
static void CrearElemento(final String datoPersona, final String
valor, final Element raiz, final Document documento){
      Element elemento = documento.createElement(datoPersona); //
creamos hijo
      Text texto = documento.createTextNode(valor); // damos valor
      raiz.appendChild(elemento); // pegamos el elemento hijo a la
raiz
      elemento.appendChild(texto); // pegamos el valor
}// fin del método
```

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

```
      Source fuente = new DOMSource(documento);
```

Se crea el resultado en el fichero **Personas.xml**:

```
      Result resultado = new StreamResult(new
java.io.File("Personas.xml"));
```

Se obtiene su TransformerFactory:

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

Se realiza la transformación del documento a fichero.

```
      transformer.transform(fuente, resultado);
```

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

```
      Result consola = new StreamResult(System.out);
      transformer.transform(fuente, consola);
```

El código completo es:

```
package ejemplos11FicherosXML;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
```

```
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;

public class CrearFicheroDOM {
	private static final long tamanhoRegistro = 35;
	public static void main(final String[] args) throws IOException {
		File fichero = new File ("NombresEdades.dat");
		RandomAccessFile raf = new RandomAccessFile(fichero, "r");
		int clave, edad;
		long posicion = 0; // para situarnos al principio del fichero
		String nombre, aux;

		DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
		try{
			DocumentBuilder db = dbf.newDocumentBuilder();
			DOMImplementation implementacion =
db.getDOMImplementation();
			Document documento = implementacion.createDocument(null,
"Personas", null);// crea el documento con el nodo raíz de nombre Personas
			documento.setXmlVersion("1.0"); // asignamos la versión
de nuestro XML

			for(;;){
				raf.seek(posicion); // nos posicionamos al comienzo
del fichero
				clave=raf.readInt();    // leemos los datos del
fichero
				nombre = raf.readUTF();
				edad=raf.readInt();

				if(clave > 0){    // clave valida
					Element raiz =
documento.createElement("persona");//creamos el  nodo persona

	documento.getDocumentElement().appendChild(raiz);// lo pegamos a la
raíz del documento
					CrearElemento("clave",
Integer.toString(clave),raiz,documento); // añadir clave
					CrearElemento("nombre",
nombre.trim(),raiz,documento); // añadir nombre
					CrearElemento("edad",
Integer.toString(edad),raiz,documento);// añadir edad

					// el método trim() elimina los espacios en
blanco al pricipio y al final de la cadena
				}// fin if clave
```

```
                        posicion = posicion+tamanhoRegistro; // se
posiciona para el siguiente registro
                        if(raf.getFilePointer() == raf.length())
                              break;
                  }// fin del for que recorre el fichero

                  // recorremos el fichero XML para ver su contenido
                  Source fuente = new DOMSource(documento);
                  Result resultado = new StreamResult(new
java.io.File("Personas.xml"));
                  Transformer transformer =
TransformerFactory.newInstance().newTransformer();
                  transformer.transform(fuente, resultado);
                  // para mostrar el documento por pantalla,podemos
especificar como resultado el canal de salida System.out
                  Result consola = new StreamResult(System.out);
                  transformer.transform(fuente, consola);
            }catch(Exception e){
                  System.err.println("Error: " +e);
            }
            raf.close();
      }// fin del main

      // metodo de insercion de los datos de la persona
      static void CrearElemento(final String datoPersona, final String
valor, final Element raiz, final Document documento){
            Element elemento = documento.createElement(datoPersona); //
creamos hijo
            Text texto = documento.createTextNode(valor); // damos valor
            raiz.appendChild(elemento); // pegamos el elemento hijo a la
raiz
            elemento.appendChild(texto); // pegamos el valor
      }// fin del metodo
}// fin de la clase
```

Leer un documento XML con DOM

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

```
DocumentBuilder db = dbf.newDocumentBuilder();
Document documento = db.parse(new File("Personas.xml"));
```

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

```
NodeList personas = documento.getElementsByTagName("persona");
```

Se realiza un bucle para recorrer la lista de nodos. Por cada nodo se obtienen sus etiquetas y sus valores llamando a la función **getNodo().**

El código es el siguiente:

```
package ejemplos11FicherosXML;

import java.io.File;
import javax.xml.parsers.DocumentBuilder;
```

```
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class LecturaFicheroDOM {

	public static void main(String[] args) {
		DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
		try{
			DocumentBuilder db = dbf.newDocumentBuilder();
			Document documento = db.parse(new File("Personas.xml"));
			//el método parse devuelve el documento DOM que se va a
crear para el fichero XML
			documento.getDocumentElement().normalize();
			/*
			 * el método normalize transforma el texto Unicode que
nosotros le enviemos en texto
			 * normalizado o bien, texto bajo una misma norma
estándar, basado en la estandarización
			 * de normalización Unicode descrita en Unicode Standard
Annex #15 – Unicode Normalization
			 *  Forms.
			 */
			System.out.println("Elemento raiz: "
+documento.getDocumentElement().getNodeName());
			// getNodeName() imprime el nombre de la raiz

			//crea una lista con todos los nodos de personas
			NodeList personas =
documento.getElementsByTagName("persona");
			/*
			 * Por medio de este método lo que se selecciona es una
lista de nodos cuyo elemento es el especificado como parámetro; a cada uno
de los nodos se le asigna un índice, de acuerdo al orden en el que
aparecen en el marcado del documento.  */

			// recorre la lista
			for(int i=0; i< personas.getLength(); i++){
				Node persona = personas.item(i); // obtener un nodo
				if(persona.getNodeType() == Node.ELEMENT_NODE){ //
tipo de nodo
					Element elemento = (Element) persona;
//obtener los elementos del nodo

					System.out.println("Clave: "
+getNodo("clave", elemento));
					System.out.println("Nombre: "
+getNodo("nombre", elemento));
					System.out.println("Edad: " +getNodo("edad",
elemento));
```

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

        // obtener la información de un  nodo
        private static String getNodo(final String etiqueta, final Element
elemento){
                NodeList nodo =
elemento.getElementsByTagName(etiqueta).item(0).getChildNodes();
                /*
                 * getChildNodes() para obtener una lista de los nodos hijos de
un elemento
                 */
                Node valorNodo = nodo.item(0);
                //metodo item() devuelve el valor del nodo indicado como
parametro
                return valorNodo.getNodeValue(); // devuelve el valor del nodo
        }
}// fin clase
```

Acceso a Ficheros XML con SAX

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

La lectura de un documento XML produce eventos que ocasiona la llamada a métodos, los eventos son encontrar la etiqueta de inicio y fin de documento (***startDocument()*** y ***endDocument()***), la etiqueta de inicio y fin de un elemento (***startElement()*** y ***endElement()***), los caracteres entre etiquetas (***characters()***), etc.

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 *DefaultHandler)*.

   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 <departamento>.
    - Que se haya detectado una etiqueta de final de un elemento, por ejemplo </departamento>.
    - 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.

Documento XML (Personas.xml)	**Métodos asociados a eventos del documento**
<?xml version="1.0" encoding="UTF-8" standalone= "no"?>	startDocument()
<Personas>	startElement()
<persona>	startElement()
<clave>	startElement()
3	characters()
</clave>	endElement()
</persona>	endElement()
<persona>	startElement()
<clave>	startElement()
5	characters()
</clave>	endElement()
</persona>	endElement()
. . . . .	. . . . .
</Personas>	endElement()
	endDocument()

Ejemplo en Java en el que se muestran los pasos básicos necesarios para hacer que se puedan tratar los eventos.

En primer lugar se incluyen las clases e interfaces de SAX:

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

Se crea un objeto procesador de XML, es decir un ***XMLReader***[5], durante la creación de este objeto se puede producir una excepción (***SAXException***) que es necesario capturar:

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

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

Interface	Función
ContentHandler	Recibe las notificaciones de los eventos que ocurren en el documento.
DTDHandler[6]	Recoge eventos relacionados con la DTD (Declaración de tipo de documento
ErrorHandler	Define métodos de tratamiento de errores
EntityResolver	Sus métodos se llaman cada vez que se encuentra una referencia a una entidad.
DefaultHandler	Clase que provee una implementación por defecto para todos sus métodos, el programador definirá los métodos que sean utilizados por el programa. Esta clase es de la que extenderemos para poder crear nuestro parser de XML.

DefaultHandler es una clase que va a procesar cada evento que lance el procesador SAX. Basta con heredar del manejador por defecto de **SAX *DefaultHandler*** y sobreescribir los métodos correspondientes a los eventos deseados. Los más comunes son:

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

Para indicar al procesador XML los objetos que realizarán el tratamiento se utiliza alguno de los siguientes métodos incluidos dentro de los objetos ***XMLReader***: ***setContentHandler(), setDTDHandler(), setEntilyResolver(), setErrorHandler()***; cada uno trata un tipo de evento y está asociado con una interfaz determinada.

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

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

[5] La extensión XMLReader es un analizador de XML. El lector actúa como un cursor yendo hacia delante en el flujo del documento y deteniéndose en cada nodo del camino.

[6] Definición de tipo de documento

```
InputSource ficheroXML = new InputSource("Personas.xml");
```

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

```
procesadorXML.parse(ficheroXML);
```

El ejemplo completo sería:

```
package ejemplos11FicherosXML;
/*
 * Ejemplo que crea un fichero XML con SAX a partir del fichero aleatorio
Personas.txt
 */
import java.io.IOException;

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

public class LeerFicheroXMLconSAX {

	public static void main(final String[] args) {

		try{/*
		 * La extensión XMLReader es un analizador de XML.
		 * El lector actúa como un cursor yendo hacia delante en el
flujo del documento y
		 * deteniéndose en cada nodo del camino.
		 */
			XMLReader procesadorXML =
XMLReaderFactory.createXMLReader();

			// esta clase es la que extiende a DefaultHandler
			GestionContenido gestor = new GestionContenido();

			// le pasa al procesador XML los eventos que van pasando
en el fichero XML
			procesadorXML.setContentHandler(gestor);

			/*
			 * InputSource
			 * Esta clase permite una aplicación SAX para encapsular
información acerca de una fuente de entrada
			 * en un solo objeto, que puede incluir un identificador
público, un identificador de sistema,
			 * un flujo de bytes (posiblemente con una codificación
especificada), y / o un flujo de caracteres.
			 * El analizador SAX utilizará el objeto InputSource para
determinar cómo leer la entrada XML
			 * según sean caracteres, bytes.
			 */
```

```
                InputSource ficheroXML = new InputSource("Personas.xml");
                //le pasamos al procesador el fichero a leer
                procesadorXML.parse(ficheroXML);
            }catch(SAXException se){
                System.out.println("Error SAX");
            }catch(IOException io){
                System.out.println("Error de L/E");
            }

        }// fin main
}// fin clase EjemploFicheroXMLSAX01

class GestionContenido extends DefaultHandler{
        public GestionContenido(){
            super();
        }
        @Override
        public void startDocument(){
            System.out.println("Comienzo del documento XML");
        }
        @Override
        public void endDocument(){
            System.out.println("Final del documento XML");
        }
        public void startElement(final String uri, final String nombre, final
String nombreC, final Attributes aatts){
            System.out.println("Principio Elemento: " +nombre);
        }
        @Override
        public void endElement(final String uri, final String nombre, final
String nombreC){
            System.out.println("Fin Elemento: " +nombre);
        }
        @Override
        public void characters(final char[] ch, final int inicio, final int
longitud){
            String car = new String(ch, inicio, longitud);
            car = car.replaceAll("[\t\n]", ""); //quitar saltos de linea
            System.out.println("\tTexto: " +car);
        }
}
```

Serialización de Objetos XML

Para serializar objetos Java a XMl o viceversa necesitamos la librería **XStream**. Para poder utilizarla debemos descargar los fichero JAR desde la web: http://xstream.codehaus.org/download.html, para el ejemplo descargaremos el fichero ***Binary distribution*** (*xstream-distribution-1.4-2-bin.zip*) lo descomprimimos y buscamos el fichero JAR ***xstream-1.4.7.jar*** que está en la carpeta ***lib.*** También necesitamos el fichero ***kxml2-2.3.0.jar*** que se puede descargar desde el apartado ***Optional Dependencies***.

Partimos del fichero *"Alumnos.Dat"* que contiene objetos *Alumno*. El proceso es crear una lista de objetos *Alumno* y la convertiremos en un fichero de datos XML. Necesitamos la clase

Alumno y la clase *ListaAlumno* en la que definimos una lista de objetos *Alumno* que pasaremos al fichero XML:

```
package ejemplos12SerializacionObjetosXML01;
/*
 * Ejemplo que recorre el fichero Alumnos.Dat para crear una lista de
alumnos
 * que después se insertarán en el fichero Alumnos.xml
 */

import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import com.thoughtworks.xstream.XStream;
public class CrearFicheroXMLconObjetos {
      public static void main(final String[] args) throws IOException,
ClassNotFoundException {
            File fichero = new File("Alumnos.DAT");
            FileInputStream lectura = new FileInputStream(fichero); //
flujo de entrada
            // conecta el flujo de bytes al flujo de datos
            ObjectInputStream datos = new ObjectInputStream(lectura);
            System.out.println("Comienza el proceso de creación del fichero
XML....");

            // Creamos un objeto Lista de alumnos
            ListaAlumnos listaalu = new ListaAlumnos();
            try{
                  while(true){ // lectura del fichero
                        Alumno alumno = (Alumno)datos.readObject();// leer
un alumno
                        listaalu.add(alumno); //añadir un alumno a la lista
                  }// fin while
            }catch(EOFException eo){}
            datos.close();
            try{
                  XStream xstream = new XStream();
                  //cambiar de nombre a las etiquetas XML
                  xstream.alias("ListadoAlumnos", ListaAlumnos.class);
                  xstream.alias("DatosAlumno", Alumno.class);
                  //quitar etiqueta lista (atributo de la clase ListaAlumno
                  xstream.addImplicitCollection(ListaAlumnos.class,
"lista");
                  //Insertar los objetos en el XML
                  xstream.toXML(listaalu, new
FileOutputStream("Alumnos.xml"));
                  System.out.println("Creado el fichero xml");
            }catch(Exception e){
                  e.printStackTrace();
            }
```

```
        }// fin main
}// fin clase
```

El fichero generado tiene el siguiente aspecto:

```
<Listado Alumnos>
      <DatosAlumno>
            <dni>11111A</dni>
            <nombre>Marta Aguirre</nombre>
            <telefono>986141414</telefono>
      </DatosAlumno>
       <DatosAlumno>
            <dni>222222B</dni>
            <nombre>Ana Sánchez</nombre>
             <telefono>627323232</telefono>
      </DatosAlumno>
      <DatosAlumno>
            <dni>333333C</dni>
            <nombre>Pedro García</nombre>
            <telefono>615545454</telefono>
      </DatosAlumno>
</Listado Alumnos>
```

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

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

En general las etiquetas XML se corresponden con el nombre de los atributos de la clase, pero pueden cambiarse utilizando el método ***alias***(). En el ejemplo se ha dado un alias a la clase ListaAlumnos que en el fichero XML aparecerá con el nombre Listado Alumnos.

```
xstream.alias("ListadoAlumnos", ListaAlumnos.class);
```

También se ha dado un alias a la clase *Alumno*, en el XML aparecerá con el nombre *DatosAlumno*

```
xstream.alias("DatosAlumno", Alumno.class);
```

Para que no aparezca el atributo *lista* de la clase *ListaAlumnos* en el XML generado se utiliza el método ***addImplicitCollection();***

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

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

```
xstream.toXML(listaalu, new FileOutputStream("Alumnos.xml"));
```

El proceso para leer del fichero XML generado es el siguiente:

```
package ejemplos12SerializacionObjetosXML01;
/*
 * Lectura de un fichero XML a objetos
 */
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.thoughtworks.xstream.XStream;

public class LeerFicheroXML {
      public static void main(final String[] args) throws
FileNotFoundException {

            //crear una instancia de la clase XStream
            XStream xstream = new XStream();
            //cambiar de nombre a las etiquetas XML
            xstream.alias("ListadoAlumnos", ListaAlumnos.class);
            xstream.alias("DatosAlumno", Alumno.class);
            //quitar etiqueta lista (atributo de la clase ListaAlumnos
            xstream.addImplicitCollection(ListaAlumnos.class, "lista");
            ListaAlumnos listadoTodas = (ListaAlumnos)
                        xstream.fromXML(new
FileInputStream("Alumnos.xml"));
            System.out.println("Número de alumnos: "
+listadoTodas.getListaAlumno());
            List<Alumno> listaAlumnos = new ArrayList<Alumno>();
            listaAlumnos = listadoTodas.getListaAlumno();
            Iterator iterator = listaAlumnos.listIterator(); //recorrer los
elementos
            while(iterator.hasNext()){
                  Alumno alu = (Alumno) iterator.next(); //obtenemos el
elemento
                  System.out.println("DNI: "+alu.getDni() +"\tNombre: "
+alu.getNombre() +"\tTeléfono: "
                              +alu.getTelefono());
            }// fin del while
            System.out.println("\n\nFin del listado... ");
      }// fin main
}//fin clase
```

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

```
ListaAlumnos listadoTodas = (ListaAlumnos)
xstream.fromXML(new FileInputStream("Alumnos.xml"));
```

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

Conversión de Ficheros XML a otro formato

XSL (*Extensible Stylesheet Language*) son recomendaciones del Word Wide Web Consortium (http://www.w3.org/Style/XSL/) para expresar hojas de estilo en lenguaje XML. Una hoja de estilo **XSL** describe el proceso de presentación a través de un pequeño conjunto de elementos XML. Esta hoja puede contener elementos de reglas que representan a las reglas

de construcción y elementos de reglas de estilo que representan a las reglas de mezclas de estilos.

En el ejemplo vamos a ver como a partir de un fichero XML que contiene datos y otro XSL que contiene la presentación de esos datos se puede generar un fichero HTML usando el lenguaje Java.

LIBROS XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ListaLibrosBiblioteca>
	<libro>
		<titulo>El Quijote</titulo>
		<autor>Cervantes</autor>
		<precio>32</precio>
	</libro>
	<libro>
		<titulo>Patria</titulo>
		<autor>Fernando Aranguren</autor>
		<precio>27</precio>
	</libro>
	<libro>
		<titulo>Los documentos electrónicos</titulo>
		<autor>Jordi Serra Serra</autor>
		<precio>15</precio>
	</libro>
</ListaLibrosBiblioteca>
```

LIBROSPLANTILLA.XLS

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
	<xsl:template match="/">
		<html> <xsl:apply-templates/> </html>
	</xsl:template>
	<xsl:template match='ListaLibrosBiblioteca'>
<head><title>LISTADO DE LIBROS</title></head>
<body>
	<h1>LISTA DE LIBROS</h1>
	<table border="1" width="50%">
		<tr><th>Título</th><th>Autor</th><th>Precio</th></tr>
		<xsl:apply-templates select='libro'/>
	</table>
</body>
	</xsl:template>
		<xsl:template match='libro'>
			<tr><xsl:apply-templates /></tr>
	</xsl:template>
		<xsl:template match='titulo|autor|precio'>
		<td><xsl:apply-templates /></td>
	</xsl:template>
</xsl:stylesheet>
```

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

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

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

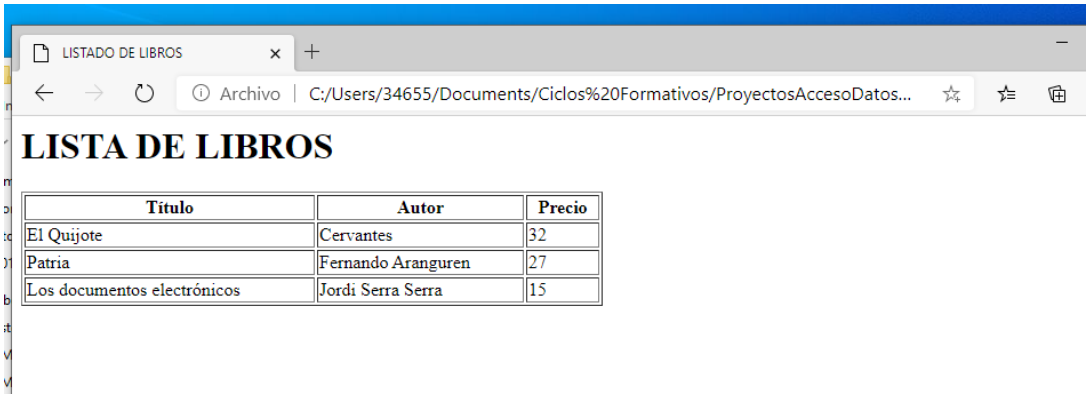
```
    transformer.transform(datos, resultado);
```

El programa completo podría ser:

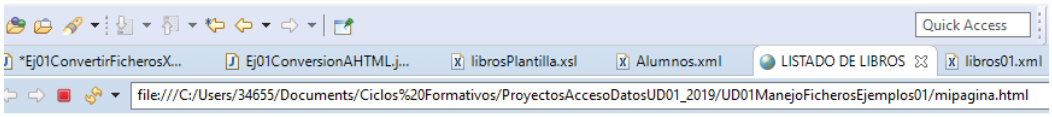
```
package ejemplos13ConversionFicheroXMLOtrosFormatos01;

import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class Ej01ConvertirFicherosXML {
      public static void main(String[] args) {
            String hojaEstilo = "librosPlantilla.xsl";
            String datosLibros = "Libros01.xml";
            File paginaHTML = new File("mipagina.html");
            FileOutputStream fos = null;
            // fuentes XSL "librosPlantilla.xsl";
            Source estilos = null;
            // fuente XML "Libros.xml";
            Source datos = null;
            //resultado de la transformacion
            Result resultado = null;
            // hacer la transformacion
            Transformer transformer = null;
            try {
                  //crea el fichero HTML "mipaginaLibros.html"
                  fos = new FileOutputStream(paginaHTML);
                  // fuentes XSL "librosPlantilla.xsl";
                  estilos = new StreamSource(hojaEstilo);
                  // fuente XML "Libros.xml";
                  datos = new StreamSource(datosLibros);
                  //resultado de la transformacion
                  resultado = new StreamResult(fos);
                  //Transformamos el resultado
                  transformer =
TransformerFactory.newInstance().newTransformer(estilos);
                  //obtiene el HTML
                  transformer.transform(datos, resultado);
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
```

```
            }catch(Exception e){
                  e.printStackTrace();
            }
            try {
                  fos.close();
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
            System.out.println("Fin del programa");
      }
}
```

El fichero HTML generado, visto desde el navegador, es el siguiente:



El fichero miPagina.html



LISTA DE LIBROS

Título	Autor	Precio
El Quijote	Cervantes	32
Patria	Fernando Aranguren	27
Los documentos electrónicos	Jordi Serra Serra	15