Leer y escribir ficheros XML con Java

Dos formas de leer un fichero XML

Para **leer un fichero XML** existen dos formas básicas (dos APIs): **SAX** y **DOM**.

Por medio de **SAX**, una clase standard de java se va encargando de leer el fichero y nos va avisando según va leyendo tags de **XML**. Nosotros debemos hacer unas clases con unos métodos concretos que son los que van recibiendo estos tags.

Por medio de **DOM**, una clase estándar de java se encarga de leer todo el fichero **XML** de golpe. Luego nos lo da en forma de **Document** para que nosotros lo vayamos analizando y haciendo lo que debamos con él.

Ventajas e inconvenientes de SAX y DOM

SAX es más complejo de programar, pero no carga todo el fichero en memoria como **DOM**.

DOM permite además escribir en el fichero **XML**, mientras que **SAX** no permite escribir.

Lectura de un fichero XML con DOM

El siguiente código lee un fichero **XML** y lo carga como un **Document**

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance
();
Document documento = null;

try
{
   DocumentBuilder builder = factory.newDocumentBuilder();
   documento = builder.parse( new File(nombreFicheroXML) );
}
catch (Exception spe)
{
   // Algún tipo de error: fichero no accesible, formato de XML
incorrecto, etc.
}
```

Una vez ejecuta esto, si todo ha ido bien, en *documento* tendremos los datos del fichero **XML**. Para analizar este documento, debemos hacer cosas como esta:

```
// Nos devuelve el nodo raíz del documento XML.
Node nodoRaiz = documento.getFirstChild();
```

Si nuestro fichero **XML** fuera

```
<nodo_raiz>
   <nodo_hijo nombre_atributo="valor">
   </nodo_hijo>
</nodo_raiz>
```

En *nodoRaiz* tendríamos algo que representa a "nodo_raiz" del fichero **XML**.

Dado un nodo (raíz o no), podemos obtener sus nodos hijo así

```
// Devuelve nodos hijos de un nodo dado
NodeList listaNodosHijos = nodo.getChildNodes();
for (int i=0; i<listaNodosHijo.getLength(); i++)
   Node unNodoHijo = listaNodosHijo.item(i);
```

En el ejemplo anterior, nuestra lista de nodos hijos de "nodo_raiz" sería un único hijo "nodo_hijo".

Para obtener los atributos de un nodo, usamos los siguientes métodos

```
// Obtener los atributos de un nodo
NamedNodeMap atributos = unNodo.getAttributes(  );
Node unAtributo = atributos.getNamedItem( "nombre_atributo" );
String valorAtributo = unAtributo.getNodeValue();
```

En nuestro ejemplo **XML**, los atributos de "nodo_hijo" sería un sólo atributo de nombre "nombre_atibuto", cuyo valor es "valor".

Escritura de un fichero XML con DOM

Supongamos que ya tenemos creado nuestro **Document**, bien porque hemos leído un fichero **XML** y lo hemos modificado, bien porque lo hemos creado con código, a base de añadir los nodos y atributos uno a uno. Vamos ahora a escribir este **Document** en un fichero **XML**.

En primer lugar, la clase **javax.xml.transform.TransformerFactory** nos permite conseguir instancias de **Transformer**. Esta clase, a su vez, nos permite transformar una fuente **XML** (en concreto nuestro **Document**) en otra cosa (en nuestro caso, un fichero **XML**). Como fuente **XML** usaremos la clase **DOMSource**, que se puede instanciar pasándole nuestro **Document**. Como destino de la transformación, usaremos un **StreamResult**, al que pasaremos un File (la veremos en detalle más adelante). El código de todo esto puede ser el siguiente:

```
// Document de XML
Document documento = ....;

// Obtención del TransfomerFactory y del Transformer a partir de él.
TransformerFactory transformerFactory =
TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
```

```
// Creación de la fuente XML a partir del documento.
DOMSource source = new DOMSource(documento);

// Creación del resultado, que será un fichero.
StreamResult result = new StreamResult(new File("otro.xml"));

// Se realiza la transformación, de Document a Fichero.
transformer.transform(source, result);
```

En el siguiente ejemplo completo vemos todo junto. Leemos un fichero **XML** usando **DOM** para obtener el **Document** y lo escribimos en otro fichero **XML**, usando también **DOM**. Es similar a un *copy* de ficheros, pero leyéndolo y escribiéndolo como **XML**.

```
package com.chuidiang.ejemplos.xml;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
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.Document;

public class EjemploLecturaEscrituraXML {

	/**
	 * Ejemplo de lectura y escritura de un fichero xml
	 * @param args
	 */
	public static void main(String[] args) {
		try {
			// Lectura de fichero_origen.xml
			DocumentBuilderFactory factory =
DocumentBuilderFactory
					.newInstance();
			DocumentBuilder builder =
factory.newDocumentBuilder();
			Document documento = builder.parse(new
File("fichero_origen.xml"));

			// Ahora documento es el XML leido en memoria.

			// Escritura de fichero_destino.xml
			TransformerFactory transformerFactory =
TransformerFactory
					.newInstance();
			Transformer transformer =
transformerFactory.newTransformer();
			DOMSource source = new DOMSource(documento);
			StreamResult result = new StreamResult(new
File("fichero_destino.xml"));
			transformer.transform(source, result);
		} catch (Exception e) {
			e.printStackTrace();
```

```
                    }
            }
}
```

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

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

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

➔ *javax.xml.parsers.*, e*n concreto *javax.xml.parsers.DocumentBuilder* y *javax.xml.parsers.DocumentBuilderFactory*, para el acceso desde JAXP.

La clase *DocumentBuilderFactory* tiene métodos importantes para indicar qué interesa y qué no 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:

- ➔ *setIgnoringComments(boolean ignore)*: Sirve para ignorar los comentarios que tenga el fichero XML.
- ➔ *setIgnoringElementContentWhitespace(boolean ignore)*: Es útil para eliminar los espacios en blanco que no tienen significado.
- ➔ *setNamespaceAware(boolean aware)*: Usado para interpretar el documento usando el espacio de nombres.
- ➔ *setValidating(boolean validate)*: 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 étodos asociados a la clase *Node*:

- ➔ Todos los nodos contienen los métodos *getFirstChild()* y *getNextSibling()* que permiten obtener uno a uno los nodos descendientes de un nodo y sus hermanos.
- ➔ El método *getNodeType()* devuelve una constante para poder distinguir entre los diferentes tipos de nodos: nodo de tipo Elemento (ELEMENT_NODE), nodo de tipo #text (TEXT_NODE), etc. Este método y las constantes asociadas son especialmente importantes a la hora de recorrer el árbol ya que permiten ignorar aquellos tipos de nodos que no interesan (por ejemplo, los #text que tengan saltos de línea).
- ➔ El método *getAttributes()* devuelve un objeto *NamedNodeMap()* (una lista con sus atributos) si el nodo es del tipo Elemento.
- ➔ Los métodos *getNodeName()* y *getNodeValue()* devuelven el nombre y el valor de un nodo de forma que se pueda hacer una búsqueda selectiva de un nodo concreto. El error típico es creer que el método *getNodeValue()* aplicado a un nodo de tipo Elemento (por ejemplo, <Titulo>) devuelve el texto que contiene. En realidad, es el nodo de tipo #text (descendiente de un nodo tipo Elemento) el que tiene el texto que representa el título del libro y es sobre él sobre el que hay que aplicar el método *getNodeValue()* para obtener el título.

La clase Document

La clase Document es una representación Java que almacena en memoria un archivo XML. Mediante esta clase y otras clases compañeras podremos recorrer cualquier punto del archivo XML.

Este recorrido se basa siempre en la visita de nodos hijo o nodos hermano. No todos los nodos son iguales y se debe tener presente que en un nodo podríamos encontrar que los saltos de línea pueden ser un problema a la hora de recorrer el árbol DOM.

Por ejemplo, dado un documento, se debe empezar obteniendo la raíz. Este elemento se llama también el "elemento documento" y podemos obtenerlo así:

```
documento=constructor.parse(archivoXML);
Element raiz=documento.getDocumentElement();
System.out.println(raiz.getNodeName());
```

La clase principal que nos interesa es la clase `Node`, siendo `Document` su clase Hija. Algunos métodos de interés son estos:

- `getDocumentElement()` obtiene el elemento raíz, a partir del cual podremos empezar a «navegar» a través de los elementos.
- `getFirstChild()` obtiene el primer elemento hijo del nodo que estemos visitando.
- `getParentNode()` nos permite obtener el nodo padre de un cierto nodo.
- `getChildNodes()` obtiene todos los nodos hijo.
- `getNextSibling()` obtiene el siguiente nodo hermano.
- `getChildNodes()` devuelve un `NodeList` con todos los hijos de un elemento. Esta `NodeList` se puede recorrer con un `for`, obteniendo el tamaño de la lista con `getLength()` y extrayendo los elementos con el método `item(posicion)`
- `getNodeType()` es un método que nos indica el tipo de nodo (devuelve un `short`). Podemos comparar con `Node.ELEMENT_NODE` para ver si el nodo es realmente un elemento.
- Otro método de utilidad es `getElementsByTagName` que extrae todas los subelementos que tengan un cierto nombre de etiqueta.

Ejercicio 1

Dado el siguiente archivo XML crear un programa que muestre toda la información de cada coche:

Ejemplo de salida

(Propiedad: matricula, Valor: 1111AAA

Propiedad: marca, Valor: AUDI

Propiedad: precio, Valor: 30000…)

```
<concesionario>
    <coches>
        <coche>
            <matricula>1111AAA</matricula>
            <marca>AUDI</marca>
            <precio>30000</precio>
        </coche>
        <coche>
            <matricula>2222BBB</matricula>
            <marca>SEAT</marca>
            <precio>10000</precio>
        </coche>
        <coche>
            <matricula>3333CCC</matricula>
            <marca>BMW</marca>
            <precio>20000</precio>
        </coche>
        <coche>
            <matricula>4444DDD</matricula>
            <marca>TOYOTA</marca>
            <precio>10000</precio>
```

```
        </coche>

    </coches>

</concesionario>
```

Ejercicio 2

Dado el siguiente archivo XML crear un programa que muestre todos los nombres:

```
<listaempleados>
        <empleado edad="27">
                <nombre>Pepe Perez</nombre>
                <categoria>Empleado</categoria>
        </empleado>
        <empleado edad="34">
                <nombre>Juan Sanchez</nombre>
                <categoria>Gerente</categoria>
        </empleado>
</listaempleados>
```