0. ¿Qué es un documento XML?

Los documentos XML son documentos que solo utilizan los elementos expuestos en el apartado anterior (declaración, etiquetas, comentarios, etc.) de **forma estructurada**. Siguen una estructura de árbol, pseudo-jerárquica, permitiendo agrupar la información en diferentes niveles, que van desde la raíz a las hojas.
Para comprender la estructura de un documento XML vamos a utilizar una terminología afín a la forma en la cual procesaremos los documentos XML. Un documento XML está compuesto desde el punto de vista de programación por nodos, que pueden (o no) contener otros nodos. Todo es un nodo:

- El par formado por la etiqueta de apertura ("<etiqueta>") y por la de cierre ("</etiqueta>"), junto con todo su contenido (elementos, atributos y texto de su interior) es un nodo llamado **elemento** (Element desde el punto de vista de programación). Un elemento puede contener otros elementos, es decir, puede contener en su interior subetiquetas, de forma anidada.
- Un **atributo** es un nodo especial llamado atributo (Attr desde el punto de vista de programación), que solo puede estar dentro de un elemento (concretamente dentro de la etiqueta de apertura).
- El **texto** es un nodo especial llamado texto (Text), que solo puede estar dentro de una etiqueta.
- Un **comentario** es un nodo especial llamado comentario (Comment), que puede estar en cualquier lugar del documento XML.
- Y por último, un **documento** (Document) es un nodo que contiene una jerarquía de nodos en su interior. Está formado opcionalmente por una declaración, opcionalmente por uno o varios comentarios y **obligatoriamente por un único elemento**.

Esto es un poco lioso, ¿verdad? Vamos a clarificarlo con ejemplos.
Primero, tenemos que entender la diferencia entre nodos padre y nodos hijo. Un elemento (par de etiquetas) puede contener varios nodos hijo, que pueden ser texto u otros elementos. Por ejemplo:

```
<padre att1="valor" att2="valor">

  texto 1

  <ethija> texto 2 </ethija>

</padre>
```

En el ejemplo anterior, el elemento padre tendría dos hijos: el texto "texto 1", sería el primer hijo, y el elemento etiquetado como "ethija", el segundo. Tendría también dos atributos, que serían nodos hijo también, pero que se consideran especiales y la forma de acceso es diferente. A su vez, el elemento "ethija" tiene un nodo hijo, que será el texto "texto 2". ¿Fácil no?
Ahora veamos el conjunto, un documento estará formado, como se dijo antes, por algunos elementos opcionales, y obligatoriamente por un único elemento (es decir, por un único par de etiquetas que lo engloba todo) que contendrá internamente el resto de información como nodos hijo. Por ejemplo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

<pedido>

<cliente> texto </cliente>

<codCliente> texto </codCliente>

...

</pedido>

La etiqueta pedido del ejemplo anterior, será por tanto el elemento raíz del documento y dentro de él estará toda la información del documento XML. Ahora seguro que es más fácil, ¿no?
Se puede y se suele usar **indentación** para que visualmente se vea con más claridad qué etiquetas van dentro de qué otras, aunque su uso no es obligado, ya que como dijimos antes, el XML no está pensado para ser leído por humanos, (aunque puedan hacerlo) sino por la aplicación de forma automática. Para el ordenador, la indentación es irrelevante.

1. Transformar el árbol DOM en archivo XML.

En el documento anterior, ya habéis visto cómo convertir el archivo XML a árbol DOM, o cómo parsearlo con varios parsers distintos.

En este sentido (convertir el archivo XML a árbol DOM) no tiene mucha complicación, pero es un pelín más complicado para hacer el camino inverso (pasar el DOM a XML).
Este proceso puede generar hasta tres tipos de excepciones diferentes. La más común, que el documento XML esté mal formado, por lo que tienes que tener cuidado con la sintaxis XML.
¿Y cómo paso la jerarquía o árbol de objetos DOM a XML?
Como hemos dicho, en Java esto es un pelín más complicado que la operación inversa, y requiere el uso de un montón de clases del paquete java.xml.transform, pues la idea es transformar el árbol DOM en un archivo de texto que contiene el documento XML.
Las clases que tendremos que usar son:

- javax.xml.transform.TransformerFactory. Fábrica de transformadores, permite crear un nuevo transformador que convertirá el árbol DOM a XML.
- javax.xml.transform.Transformer. Transformador que permite pasar un árbol DOM a XML.
- javax.xml.transform.TransformerException. Excepción lanzada cuando se produce un fallo en la transformación.
- javax.xml.transform.OutputKeys. Clase que contiene opciones de salida para el transformador. Se suele usar para indicar la codificación de salida (generalmente UTF-8) del documento XML generado.

- javax.xml.transform.dom.DOMSource. Clase que actuará de intermediaria entre el árbol DOM y el transformador, permitiendo al transformador acceder a la información del árbol DOM.
- javax.xml.transform.stream.StreamResult. Clase que actuará de intermediaria entre el transformador y el archivo o String donde se almacenará el documento XML generado.
- java.io.File. Clase que, como posiblemente sabrás, permite leer y escribir en un archivo almacenado en disco. El archivo será obviamente el documento XML que vamos a escribir en el disco.

Esto es un poco lioso, ¿o no? No lo es tanto cuando se ve un ejemplo de cómo realizar el proceso de transformación de árbol DOM a XML, así que veamos ese ejemplo:

```
try {

        // 1º Creamos una instancia de la clase File para acceder al archivo donde guardaremos el XML.

        File f=new File(CaminoAlArchivoXML);

        //2º Creamos una nueva instancia del transformador a través de la fábrica de transformadores.

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

        //3º Establecemos algunas opciones de salida, como por ejemplo, la codificación de salida.

        transformer.setOutputProperty(OutputKeys.INDENT, "yes");

        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

        //4º Creamos el StreamResult, que intermediará entre el transformador y el archivo de destino.

        StreamResult result = new StreamResult(f);

        //5º Creamos el DOMSource, que intermediará entre el transformador y el árbol DOM.

        DOMSource source = new DOMSource(doc);

        //6º Realizamos la transformación.

        transformer.transform(source, result);

} catch (TransformerException ex) {

   System.out.println("¡Error! No se ha podido llevar a cabo la transformación.");

}
```

A continuación te adjuntamos, cortesía de la casa, una de esas clases anti-estrés, que nos permitirá hacer estas operaciones sin tener que reinventar la rueda cada vez, ni tener que acordarnos de tantos detalles... Utilízala todo lo que quieras, con ella podrás convertir un documento XML a árbol DOM y viceversa de forma sencilla. El documento XML puede estar almacenado en un archivo o en una cadena de texto (e incluso en Internet, para lo que necesitas el URI (Identificador de Recursos Unificado)).
Los métodos estáticos DOM2XML() te permitirán pasar el árbol DOM a XML, y los métodos String2DOM() y XML2DOM() te permitirán pasar un documento XML a un árbol DOM .
También contiene el método crearDOMVacio() que permite crear un árbol DOM vacío, útil para empezar un documento XML de cero.
DOMUtil.java

2. Manipulación de documentos XML.

2. 1.- Obtener el elemento raíz, y buscar un elemento.

Bien, ahora sabes cargar un documento XML a árbol DOM y de árbol DOM a XML, pero, ¿cómo se modifica el árbol DOM?
Como ya se dijo antes, un árbol DOM es una estructura en árbol, jerárquica como cabe esperar, formada por nodos de diferentes tipos. El funcionamiento del modelo de objetos DOM es establecido por el organismo W3C, lo cual tiene una gran ventaja: **el modelo es prácticamente el mismo en todos los lenguajes de programación**.
En Java, prácticamente todas las clases que vas a necesitar para manipular un árbol DOM están en el paquete org.w3c.dom. Si vas a hacer un uso muy intenso de DOM es conveniente que hagas una importación de todas las clases de este paquete ("import org.w3c.dom.*;").
Tras convertir un documento XML a DOM lo que obtenemos es una instancia de la clase org.w3c.dom.Document. Esta instancia será el nodo principal que contendrá en su interior toda la jerarquía del documento XML. Dentro de un documento o árbol DOM podremos encontrar los siguientes tipos de clases:

- org.w3c.dom.Node (Nodo). Todos los objetos contenidos en el árbol DOM son nodos. La clase Document es también un tipo de nodo, considerado el nodo principal.
- org.w3c.dom.Element (Elemento). Corresponde con cualquier par de etiquetas ("<> </>") y todo su contenido (atributos, texto, subetiquetas, etc.).
- org.w3c.dom.Attr (Atributo). Corresponde con cualquier atributo.
- org.w3c.dom.Comment (Comentario). Corresponde con un comentario.
- org.w3c.dom.Text (Texto). Corresponde con el texto que encontramos dentro de dos etiquetas.

¿A que te resultan familiares?
Claro que sí. Estas clases tendrán diferentes métodos para acceder y manipular la información del árbol DOM. A continuación vamos a ver las operaciones más importantes sobre un árbol DOM. En todos los ejemplos, "documento" corresponde con una instancia de la clase Document.
Obtener el elemento raíz del documento.
Como ya sabes, los documentos XML deben tener obligatoriamente un único elemento ("<pedido></pedido>" por ejemplo), considerado el elemento raíz, dentro del cual está

el resto de la información estructurada de forma jerárquica. Para obtener dicho elemento y poder manipularlo podemos usar el método getDocumentElement():

```
Element raiz=documento.getDocumentElement();
```

Buscar un elemento en toda la jerarquía del documento.
Para realizar esta operación se puede usar el método getElementsByTagName() disponible tanto en la clase Document como en la clase Element. Dicha operación busca un elemento por el nombre de la etiqueta y retorna una lista de nodos (NodeList) que cumplen con la condición. Si se usa en la clase Element, solo buscará entre las subetiquetas (subelementos) de dicha clase (no en todo el documento).

```
NodeList listaNodos=documento.getElementsByTagName("cliente");
Element cliente;

if (listaNodos.getLength()==1){
   cliente=(Element)listaNodos.item(0);
}
```

El método getLength() de la clase NodeList, permite obtener el número de elementos (longitud de la lista) encontrados cuyo nombre de etiqueta es coincidente. El método item() permite acceder a cada uno de los elementos encontrados, y se le pasa por argumento el índice del elemento a obtener (empezando por cero y acabando por longitud menos uno). Fíjate que es necesario hacer una conversión de tipos después de invocar el método item(). Esto es porque la clase NodeList almacena un listado de nodos (Node), sin diferenciar el tipo.

2. 2- Obtener y procesar la lista de hijos y añadir uno nuevo.

¿Y qué más operaciones puedo realizar sobre un árbol DOM? Veámoslas.

Obtener la lista de hijos de un elemento y procesarla.
Se trata de obtener una lista con los nodos hijo de un elemento cualquiera, estos pueden ser un sub-elemento (sub-etiqueta) o texto. Para sacar la lista de nodos hijo se puede usar el método getChildNodes():

```
NodeList listaNodos=documento.getDocumentElement().getChildNodes();
for (int i=0; i<listaNodos.getLength();i++) {
  Node nodo=listaNodos.item(i);
  switch (nodo.getNodeType()){
     case Node.ELEMENT_NODE:
         Element elemento = (Element) nodo;
         System.out.println("Etiqueta:" + elemento.getTagName());
         break;
     case Node.TEXT_NODE:
         Text texto = (Text) nodo;
         System.out.println("Texto:" + texto.getWholeText());
         break;
  }
}
```

En el ejemplo anterior se usan varios métodos. El método getNodeType() de la clase Node permite saber de qué tipo de nodo se trata, generalmente texto (Node.TEXT_NODE) o un sub-elemento (Node.ELEMENT_NODE). De esta forma podremos hacer la conversión de tipos adecuada y gestionar cada elemento según corresponda. También se usa el método getTagName() aplicado a un elemento, lo cual permitirá obtener el nombre de la etiqueta, y el método getWholeText() aplicado a un nodo de tipo texto (Text), que permite obtener el texto contenido en el nodo.

Añadir un nuevo elemento hijo a otro elemento.
Hemos visto cómo mirar qué hay dentro de un documento XML pero no hemos visto cómo añadir cosas a dicho documento. Para añadir un sub-elemento o un texto a un árbol DOM, primero hay que crear los nodos correspondientes y después insertarlos en la posición que queramos. Para crear un nuevo par de etiquetas o elemento (Element) y un nuevo nodo texto (Text), lo podemos hacer de la siguiente forma:

```
Element direccionTag=documento.createElement("DIRECCION_ENTREGADA");
Text direccionTxt=documento.createTextNode("C/Perdida S/N");
```

Ahora los hemos creado, pero todavía no los hemos insertado en el documento. Para ello podemos hacerlo usando el método appendChild() que añadirá el nodo (sea del tipo que sea) al final de la lista de hijos del elemento correspondiente:

```
direccionTag.appendChild(direccionTxt);
documento.getDocumentElement().appendChild(direccionTag);
```

En el ejemplo anterior, el texto se añade como hijo de la etiqueta "DIRECCION_ENTREGADA", y a su vez, la etiqueta "DIRECCION_ENTREGADA" se añade como hijo, al final del todo, de la etiqueta o elemento raíz del documento. Aparte del método appendChild(), que siempre insertará al final, puedes utilizar los siguientes métodos para insertar nodos dentro de un árbol DOM (todos se usan sobre la clase Element):

- insertBefore (Node nuevoNodo, Node nodoReferencia). Insertará un nodo nuevo antes del nodo de referencia (nodoReferencia).
- replaceChild (Node nuevoNodo, Node nodoAnterior). Sustituye un nodo (nodoAnterior) por uno nuevo.

2. 3.- Eliminar un hijo y modificar un elemento texto.

Seguimos con las operaciones sobre árboles DOM. ¿Sabrías cómo eliminar nodos de un árbol? ¿No? Vamos a descubrirlo.

Eliminar un elemento hijo de otro elemento.
Para eliminar un nodo, hay que recurrir al nodo padre de dicho nodo. En el nodo padre se invoca el método removeChild(), al que se le pasa la instancia de la clase Element con el nodo a eliminar (no el nombre de la etiqueta, sino la instancia), lo cual implica que primero hay que buscar el nodo a eliminar, y después eliminarlo. Veamos un ejemplo:

```
NodeList
listaNodos3=documento.getElementsByTagName("DIRECCION_ENTREGA");
```

```
for (int i=0;i<listaNodos3.getLength();i++){

   Element elemento=(Element) listaNodos3.item(i);

   Element padre = (Element)elemento.getParentNode();

   padre.removeChild(elemento);

}
```

En el ejemplo anterior se eliminan todas las etiquetas, estén donde estén, que se llamen "DIRECCION_ENTREGA". Para ello ha sido necesario buscar todos los elementos cuya etiqueta sea esa (como se explicó en ejemplos anteriores), recorrer los resultados obtenidos de la búsqueda, obtener el nodo padre del hijo a través del método getParentNode(), para así poder eliminar el nodo correspondiente con el método removeChild().
No es obligatorio obviamente invocar al método getParentNode() si el nodo padre es conocido. Por ejemplo, si el nodo es un hijo del elemento o etiqueta raíz, hubiera bastado con poner lo siguiente:
documento.getDocumentElement().removeChild(elemento);

Cambiar el contenido de un elemento cuando solo es texto.
Los métodos getTextContent() y setTextContent(), aplicados a un elemento, permiten respectivamente acceder al texto contenido dentro de un elemento o etiqueta o modificar dicho contenido. Tienes que tener cuidado, porque utilizar setTextContent() significa eliminar cualquier hijo (sub-elemento por ejemplo) que previamente tuviera la etiqueta. Ejemplo:

```
Element
nuevo=documento.createElement(DIRECCION_RECOGIDA").setTextContent("C/Del
Medio S/N");

System.out.println(nuevo.getTextContent());
```

2. 4.- Manejar atributos de un elemento.

Y ya solo queda una cosa: descubrir cómo manejar los atributos en un árbol DOM.
Atributos de un elemento.
Por último, lo más fácil. Cualquier elemento puede contener uno o varios atributos. Acceder a ellos es sencillísimo, solo necesitamos tres métodos:

- setAttribute() para establecer o crear el valor de un atributo.
- getAttribute() para obtener el valor de un atributo.
- removeAttribute() para eliminar el valor de un atributo.

Veamos un ejemplo:

```
documento.getDocumentElement().setAttribute("urgente","no");

System.out.println(documento.getDocumentElement().getAttribute("urgente"));
```

En el ejemplo anterior se añade el atributo "urgente" al elemento raíz del documento con el valor "no", después se consulta para ver su valor. Obviamente se puede realizar en cualquier otro elemento que no sea el raíz. Para eliminar el atributo es tan sencillo como lo siguiente:

```
documento.getDocumentElement().removeAttribute("urgente");
```