1. TRATAMIENTO DE DOCUMENTOS XML

XML es un mecanismo extraordinariamente sencillo para estructurar, almacenar e intercambiar información entre sistemas informáticos.

XML define un lenguaje de etiquetas, muy fácil de entender pero con unas reglas muy estrictas, que permite encapsular información de cualquier tipo para posteriormente ser manipulada. Se ha extendido tanto que hoy día es un estándar en el intercambio de información entre sistemas.

La información en XML va escrita en texto legible por el ser humano, pero no está pensada para que sea leída por un ser humano, sino por una máquina. La información va codificada generalmente en unicode, pero estructurada de forma que una máquina es capaz de procesarla eficazmente. Esto tiene una clara ventaja: si necesitamos modificar algún dato de un documento en XML, podemos hacerlo con un editor de texto plano. Veamos los elementos básicos del XML:

Elemento	Descripción	Ejemplo
Cabecera o declaración del XML.	Es lo primero que encontramos en el documento XML y define cosas como, por ejemplo, la codificación del documento XML (que suele ser ISO-8859-1 o UTF-8) y la versión del estándar XML que sigue nuestro documento XML.	`<?xml version="1.0" encoding="ISO-8859-1"?>`

Etiquetas.	Una etiqueta es un delimitador de datos, y a su vez, un elemento organizativo. La información va entre las etiquetas de apertura (“<pedido>”) y cierre (“</pedido>”). **Fíjate en el nombre de la etiqueta (“pedido”), debe ser el mismo tanto en el cierre como en la apertura, respetando mayúsculas.**	`<pedido>`
		`  información del pedido`
		`</pedido>`

Atributos.	Una etiqueta puede tener asociado uno o más atributos. Siempre deben ir detrás del nombre de la etiqueta, en la etiqueta de apertura, poniendo el nombre del atributo seguido de igual y **el valor encerrado entre comillas**. Siempre debes dejar al menos un espacio entre los atributos.	`<articulo cantidad="20">`
		`  información`
		`</articulo>`

Texto.	Entre el cierre y la apertura de una etiqueta puede haber texto.	`<cliente>`
		`  Muebles Bonitos S.A.`
		`</cliente>`

Etiquetas sin contenido.	Cuando una etiqueta no tiene contenido, no tiene porqué haber una etiqueta de cierre, pero no debes olvidar poner la barra de cierre (“/”) al final de la etiqueta para indicar que no tiene contenido.	`<fecha entrega="1/1/2012" />`
Comentario.	Es posible introducir comentarios en XML y estos van dirigidos generalmente a un ser humano que lee directamente el documento XML.	`<!-- comentario -->`

El nombre de la etiqueta y de los nombres de los atributos no deben tener espacios. También es conveniente evitar los puntos, comas y demás caracteres de puntuación. En su lugar se puede usar el guión bajo (“<pedido_enviado> … </pedido_enviado>”).

1.1. ¿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.

Un documento XML está compuesto desde el punto de vista de programación por nodos, 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). 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), 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.
- Un documento 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.

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 sería 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”.

Ahora veamos el conjunto, un documento estará formado, por algunos elementos opcionales, y obligatoriamente por un único elemento (es decir, por 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.

1.2. Librerías para procesar documentos XML (paso de XML a DOM)

El W3C o Wold Wide Web Consortium es la entidad que establece las bases del XML. Dicha entidad, además de describir como es el XML internamente, define diversas tecnologías estándar adicionales para verificar, convertir y manipular documentos XML. Nosotros no vamos a explorar todas las tecnologías de XML aquí (son muchísimas), solamente vamos a usar dos de ellas, aquellas que nos van a permitir manejar de forma simple un documento XML:

- Procesadores de XML. Son librerías para leer documentos XML y comprobar que están bien formados. En Java, el procesador más utilizado es SAX.
- XML DOM. Permite transformar un documento XML en un modelo de objetos (de hecho DOM significa Document Object Model), accesible cómodamente desde el lenguaje de programación. DOM almacena cada elemento, atributo, texto, comentario, etc. del documento XML en una estructura tipo árbol compuesta por nodos fácilmente accesibles, sin perder la jerarquía del documento. A partir de ahora, la estructura DOM que almacena un XML la llamaremos árbol o jerarquía de objetos DOM.

En Java, estas y otras funciones están implementadas en la librería JAXP (Java API for XML Processing), y ya van incorporadas en la edición estándar de Java (Java SE). En primer lugar vamos a ver como convertir un documento XML a un árbol DOM, y viceversa, para después ver cómo manipular desde Java un árbol DOM.

Para cargar un documento XML tenemos que hacer uso de un procesador de documentos XML (conocidos generalmente como parsers) y de un constructor de documentos DOM. Las clases de Java que tendremos que utilizar son:

- *javax.xml.parsers.DocumentBuilder*: será el procesador y transformará un documento XML a DOM, se le conoce como constructor de documentos.
- *javax.xml.parsers.DocumentBuilderFactory*: permite crear un constructor de documentos, es una fábrica de constructores de documentos.
- *org.w3c.dom.Document*: una instancia de esta clase es un documento XML pero almacenado en memoria siguiendo el modelo DOM. Cuando el parser procesa un documento XML creará una instancia de esta clase con el contenido del documento XML.

Para pasar de XML a DOM hay que seguir los siguientes pasos:

```
// 1º Creamos una nueva instancia de una fabrica de constructores de documentos.
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
// 2º A partir de la instancia anterior, fabricamos un constructor de documentos, que procesará el XML.
DocumentBuilder db = dbf.newDocumentBuilder();
// 3º Procesamos el documento (almacenado en un archivo) y lo convertimos en un árbol DOM.
Document doc=db.parse(CaminoAArchivoXml);
```

En caso de crear un documento desde cero habría que sustituir el tercer paso por:

```
Document doc=db.newDocument();
```

1.2. Librerías para procesar documentos XML (paso de DOM a XML)

Pasar la jerarquía o árbol de objetos DOM a XML requiere el uso de un varias clases del paquete java.xml.transform.

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 permite leer y escribir en un archivo almacenado en disco. El archivo será obviamente el documento XML que vamos a escribir en el disco.

Veamos el ejemplo:

```
// 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, intermediaria entre el transformador y el archivo de destino.
StreamResult result = new StreamResult(f);
// 5º Creamos el DOMSource, intermediaria entre el transformador y el árbol DOM.
DOMSource source = new DOMSource(doc);
//6º Realizamos la transformación.
transformer.transform(source, result);
```

1.3. Manipulación de documentos XML

Un árbol DOM es una estructura en árbol jerárquica 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 para manipular un árbol DOM se encuentran en el paquete org.w3c.dom (paquete que tendrás que importar).

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*. Todos los objetos contenidos en el árbol son nodos. La clase Document es también un nodo, considerado el nodo principal.
- *org.w3c.dom.Element*. Corresponde con cualquier par de etiquetas (“<pedido></pedido>”) y todo su contenido (atributos, texto, subetiquetas, etc.). .
- *org.w3c.dom.Attr*. Corresponde con cualquier atributo.
- *org.w3c.dom.Comment*. Corresponde con un comentario.
- *org.w3c.dom.Text*. Corresponde con el texto que encontramos dentro de las etiquetas.

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, “doc” corresponde con una instancia de la clase Document.

Obtener el elemento raíz del documento.

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=doc.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 n1=doc.getElementsByTagName(“cliente”);
Element cliente;
if(n1.getLength()==1) cliente=(Element)n1.item(0);
```

El método getLength() de la clase NodeList, permite obtener el número de elementos 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. Fijate 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.

Obtener una lista de hijos de un elemento y procesarla

Se trata de obtener una lista con los nodos hijo de un elemento cualquiera. Para sacar la lista de nodos hijo se puede usar el método getChildNodes:

```
NodeList nl=doc.getDocumentElement().getChildNodes();
for (int i=0; i<nl.getLength();i++) {
        Node n=nl.item(i); ()){
        switch (n.getNodeType()){
                case Node.ELEMENT_NODE:
                        Element e=(Element)n;
                        System.out.println("Etiqueta:" + e.getTagName());
                        break;
                case Node.TEXT_NODE:
                        Text t=(Text)n;
                        System.out.println("Texto:" + t.getWholeText();
                        break;
                }
}
```

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

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 dirTag=doc.createElement(“DireccionEntrega”);
Text dirTxt=doc.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:

```
dirTag.appendChild(dirTxt);
doc.getDocumentElement().appendChild(dirTag);
```

En el ejemplo anterior, el texto se añade como hijo de la etiqueta “Direccion_entrega”, y a su vez, la etiqueta “Direccion_entrega” 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 arbol DOM (todos se usan sobre la clase Element):

- *insertBefore* (Node nuevo, Node referencia). Insertará un nodo nuevo antes del nodo de referencia.
- *replaceChild* (Node nuevo, Node anterior). Sustituye un nodo (anterior) por uno nuevo.

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 nl3=doc.getElementsByTagName("Direccion_entrega");
for (int i=0;i<nl3.getLength();i++){
        Element e=(Element)nl3.item(i);
        Element parent=(Element)e.getParentNode();
        parent.removeChild(e);
}
```

En el ejemplo anterior se eliminan todos las etiquetas, estén donde estén, que se llamen “Direccion_entrega”. Para ello ha sido necesario buscar todos los elementos cuya etiqueta sea esa,

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:
doc.getDocumentElement().removeChild(e);

Cambiar el contenido de un elemento cuando sólo es texto

Los métodos getTextContent y setTextContent, aplicado a un elemento, permiten respectivamente acceder al texto contenido dentro de un elemento o etiqueta. Tienes que tener cuidado, porque utilizar setTextContent significa eliminar cualquier hijo que previamente tuviera la etiqueta.
Ejemplo:

```
Element nuevo=doc.createElement("direccion_recogida").setTextContent("C/Del Medio S/N");
System.out.println(nuevo.getTextContent());
```

Atributos de un elemento

Cualquier elemento puede contener uno o varios atributos. Acceder a ellos es sencillo, sólo necesitamos tres métodos: setAttribute para establecer o crear el valor de un atributo, getAttribute, para obtener el valor de un atributo, y removeAttribute, para eliminar el valor de un atributo.

```
doc.getDocumentElement().setAttribute("urgente","no");
System.out.println(doc.getDocumentElement().getAttribute("urgente"));
```

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

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

Funciones útiles para trabajar con el documento

```
//Obtener el Element raíz:
Element raiz=doc.getDocumentElement();

//Obtener una lista de nodos
NodeList n1=doc.getElementsByTagName(“persona”);

//Obtener una lista de los hijos de un Element
NodeList n1=doc.getChildNodes();

//Obtener el valor de un atributo de un Element
String id=raiz.getAttribute(String s);

//Obtener el nombre de un Element
String nombre=raiz.getTagName();
```

```
//Obtener el tipo de un Nodo
nodo.getNodeType();

//Obtener un nodo de una lista
Node nodo=nList.item(i);

//Comprobar si un elemento tiene un atributo
Boolean b=raiz.hasAttribute(“id”);

//Obtener el texto de un nodo Text
String contenido=n1.getWholeText();

//Obtener el texto de dentro de una etiquetado
String contenido=e.getTextContent();

//Obtener nombre y valor de un atributo
atributo.getName();
atributo.getValue();

//Obtener todo el texto de un nodo (incluye el texto de los hijos)
nodo.getTextContent();

//Funciones para crear nodos
doc.createTextNode();
doc.createComment();
doc.createElement();
doc.createAttribute();

//Insertar un Nodo después de otro
Node insertBefore(Node nuevoHijo, Node nodoReferencia);

//Eliminar el Node hijo que se pasa como referencia (devuelve ese Nodo)
Node removeChild(Node antiguoHijo;

//Remplazo NodoS hijos (devuelve el nodo reemplazado)
Node replaceChild(Node nuevoHijo, Node nodoReemplazado);

//Añadir un nuevo nodo como último hijo
Node appendChild(Node hijoNuevo);
```

2. SAX

El objetivo de SAX es posibilitar la construcción de analizadores (*parsers*) de documentos XML. En realidad, SAX no es más que un API para analizar los documentos (como su propio nombre, *Simple API for XML*, indica). Existen otros APIs, como DOM, pero con aplicaciones distintas: SAX es recomendable para extraer información del documento, mientras que DOM lo es para manipular su estructura. En este tema describiremos el funcionamiento de un analizador SAX y las tareas para las que es más adecuado.

2.1. ¿Cómo funciona SAX?

Los analizadores SAX están basados en un modelo de eventos: a medida que el *parser* recorre el documento éste informa de la ocurrencia de eventos, tales como el comienzo de un elemento XML o el final del documento, a un manejador de eventos (*event handler*). Se sigue una filosofía parecida a la forma en que se implementa el modelo de eventos en AWT: existe un objeto, una componente gráfica como un botón o un selector, que puede emitir eventos y existen objetos oyentes (*listeners*) que pueden manejarlos. Estos oyentes implementan una interfaz que el objeto emisor entiende y por ello deben registrarse como oyentes autorizados para ser capaces de manejar estos eventos. Cuando se produce un evento el emisor informa solo a los oyentes registrados para que estos lo traten invocando al método correspondiente de la interfaz.

Por ejemplo, veamos qué hace un analizador SAX cuando procesa un documento XML como el siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<documentoXML>
<cabecera> Esto es un documento XML </cabecera>
Eso es todo amigos
</documentoXML>
```

Se producirían los siguientes eventos como salida:
startDocument()
startElement(): documentoXML
startElement(): cabecera
characters(): Esto es un documento XML
endElement(): cabecera
characters(): Eso es todo amigos
endElement(): documentoXML
endDocument()

Lo que demuestra que SAX realiza un análisis secuencial y no hay manera de determinar relaciones padre/hijo: es responsabilidad del programador descubrir si el elemento `<cabecera>` debe encontrarse justo a continuación de `<documentoXML>` o no.

2.2. Estructura de un analizador SAX

El API está formado por un conjunto de interfaces y clases:

- La interfaz `ContentHandler` es la más importante desde el punto de vista del desarrollador. Especifica los manejadores de eventos que debemos implementar en nuestro analizador y que veremos en el siguiente apartado. En caso de que no nos interesen todos los eventos, sino solo algunos concretos, puede ser más sencillo extender la clase

`DefaultHandler`, que ya incorpora manejadores por defecto (vacíos) para todos los eventos.

•Las interfaces `ErrorHandler`, `EntityResolver` y `DTDHandler` desempeñan un papel parecido a `ContentHandler`, pero para tratar errores, entidades y DTDs, respectivamente. Afortunadamente, la clase `DefaultHandler` también implementa estos interfaces.

•La interfaz `XMLReader` es el "corazón" del API, aquí vienen los métodos que analizan el XML y generan los eventos. No obstante, el código de estos métodos es fijo e independiente de la aplicación, por lo que desde el punto de vista del desarrollador nos limitaremos a usar clases ya implementadas en su totalidad.

•Hay tres clases descendientes de `SAXException` que representan los distintos tipos de error que se pueden producir en el análisis.

2.3. Trabajar con SAX en Java

En la actualidad hay diversas implementaciones de analizadores SAX en Java. La más conocida y usada es Xerces, del proyecto Apache (no sólo implementa SAX, sino también DOM). No obstante, el problema de SAX es que deja algunos "cabos sueltos" en cuanto a su uso real. Por ejemplo, la forma de instanciar un parser SAX no está contemplada en el estándar, y es dependiente de la implementación empleada. Este problema se solucionó con la aparición del API JAXP (Java API for XML Processing), desarrollado por Sun, que añade una capa de abstracción sobre SAX y sobre otros estándares, y permite trabajar con XML y Java independientemente de la implementación de parser que estemos usando.

JAXP está dividido en dos partes, el API de análisis y el de transformación (para aplicar XSLT), por lo que las diferentes implementaciones existentes pueden cubrir uno, otro o ambos:

•Implementación de referencia de JAXP (JAXP RI): desarrollada por Sun. La versión actual es la 1.2.0. En el J2SE 1.4 se incluye la versión anterior, la 1.1 (que no soporta schemas). Cubre todo el API JAXP.

•Xerces 2: el propio Xerces implementa también JAXP, pero solo el API de análisis. Soporta schemas.

Instanciar un analizador SAX con JAXP

JAXP encapsula en la clase `SAXParser` una implementación de un analizador SAX. Podemos utilizar la que viene en la distribución de JAXP o bien cualquier otra, siempre que sea compatible con él. Como se muestra en el ejemplo, los objetos `SAXParser` se construyen a partir de la clase `SAXParserFactory`. Una vez instanciado el analizador, se efectúa el análisis llamando al método `parse`:

```
import java.io.FileReader;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.DefaultHandler;
public class EjemploSax {
        public static void main(String[] args) throws Exception {
                SAXParserFactory factory = SAXParserFactory.newInstance();
                SAXParser parser = factory.newSAXParser();
                parser.parse(args[0], new DefaultHandler());
        }
}
```

Al método `parse` hay que pasarle una entrada de donde obtener un documento (en el ejemplo, un nombre de fichero) y un manejador de eventos (en el ejemplo, el manejador por defecto, la clase `DefaultHandler`, que no hace nada salvo que haya un error en el documento).

2.4. Eventos SAX

Como hemos visto en el apartado anterior, el código correspondiente a nuestro analizador SAX debe extender la clase `DefaultHandler` (o implementar los interfaces `ContentHandler`, `ErrorHandler`, `EntityResolver` y `DTDHandler`).

El siguiente ejemplo muestra un manejador de eventos que imprime un mensaje en la salida cuando se produce algún evento SAX, mostrando los parámetros del evento.

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class SAXParserHandler extends DefaultHandler {

//comienzo del documento
public void startDocument() throws SAXException {
        System.out.println("startDocument");
}

//fin del documento
public void endDocument() throws SAXException {
        System.out.println("endDocument");
}

//texto dentro de etiquetas
public void characters(char[] ch, int start, int length throws SAXException {
        String charString = new String(ch, start, length);
        System.out.println("caracteres: " + charString);
}

//etiqueta de apertura, puede contener atributos
public void startElement(String namespaceURI, String localName, String qName,
Attributes atts) throws SAXException {
        System.out.println("startElement: " + qName);
        // Lista atributos y sus valores
        for (int i=0; i<atts.getLength(); i++) {
                System.out.println("Atributo: " + atts.getLocalName(i));
                System.out.println("\tValor: " + atts.getValue(i));
        }
}

//etiqueta de cierre
public void endElement(String nameSpaceURI, String localName, String qName)
throws SAXException {
        System.out.println("endElement: " + qName);
}
```

```
//espacio en blanco que se puede ignorar
public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException {
        System.out.println(length + " caracteres en blanco ignorables");
}
//comienzo de espacio de nombres
public void startPrefixMapping(String prefix, String uri) throws SAXException {
        System.out.println("Comienza prefijo de namespace: " + prefix);
}

//fin de espacio de nombres
public void endPrefixMapping(String prefix) throws SAXException {
        System.out.println("Termina prefijo de namespace: " + prefix);
}

//instrucción de procesamiento
public void processingInstruction(String instruction, String data) throws SAXException {
        System.out.println("Instrucción: " + instruction + ", datos: " + data);
}

//entidad que no se desea resolver, por lo que se ignora
public void skippedEntity(String name) throws SAXException {
        System.out.println("Entidad saltada: " + name);
}

}
```

Como puede verse en el listado anterior, un manejador de eventos SAX puede gestionar los siguientes eventos:

- `startDocument()` y `endDocument()`: se producen cuando se empieza y se termina de analizar el documento XML, respectivamente.
- `startElement()`: Se activa en el momento en que el se encuentra una etiqueta de apertura. Los dos primeros argumentos, `namespaceURI` y `localName`, se usan solamente si el parser contempla espacios de nombres. El argumento `qName` contiene el nombre del elemento. Por otro lado, el argumento `atts` contiene los atributos asociados al elemento (consultar la interfaz `org.xml.sax.Attributes` para ver los métodos de acceso como `getLocalName()` y `getValue()` que en este caso se acceden via índices en lugar de por valor lo cual es mejor para evitar dependencias del orden de aparición).
- `endElement()`: Por cada `startElement()` hay un par `endElement()` siendo los argumentos iguales pero omitiendo los atributos en este último. Cuando tenemos un elemento vacío se dispararán ambos eventos.
- `characters()`: Se dispara cuando se encuentran datos tipo carácter, incluyendo las secciones CDATA. Los argumentos incluyen un array de caracteres, el comienzo del array y la longitud del mismo.

3. PROCESAMIENTO DE XML: XPATH (XML PATH LENGUAGE)

La alternativa más sencilla para consultar información dentro de un documento XML es mediante el uso de XPath, una especificación de la W3C para la consulta de XML. Con XPath se puede seleccionar y hacer referencia a texto, elementos, atributos y cualquier otra información contenida dentro de un fichero XML.

3.1. Lo básico de XPath

XPath comienza con la noción del 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 un consulta XPath:

- (./) usa el nodo en el que se encuentra actualmente como contexto.
- (/) 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.

Dado el siguiente esquema:

```
<Libros>
	<Libro publicado_en=”1840”>
		<Titulo>El capote</Titulo>
		<Autor>Nikolai Gogol</Autor>
	</Libro>
	<Libro publicado_en=”2008”>
		<Titulo>El Sanador de Caballos</Titulo>
		<Autor>Gonzalo Giner</Autor>
	</Libro>
	<Libro publicado_en=”1981”>
		<Titulo>El nombree de la Rosa</Titulo>
		<Autor>Umberto Eco</Autor>
	</Libro>
</Libros>
```

Para seleccionar elementos de un XML se realizan avances hacia dentro de la jerarquía del documento. Por ejemplo, para seleccionar todos los elementos Autor del documento XML Libros:

/Libros/Libro/Autor

Si se desea obtener todos los elementos Autor del documento se puede usar la siguiente expresión:

//Autor

De esta forma no es necesaria la trayectoria completa.

También se pueden usar comodines en cualquier nivel del árbol. Por ejemplo, la siguiente expresión selecciona todos los nodos Autor que son nietos de Libros:

/Libros//Autor*

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

Para identificar un conjunto de atributos se utiliza el carácter @ delante del nombre del atributo. Por ejemplo. La siguiente expresión selecciona todos los atributos publicado_en de un elemento Libro:

/Libros/Libro/@publicado_en

Ya que en el documento de ejemplo sólo los elementos Libro tienen el atributo publicado_en, la misma consulta se puede realizar de la siguiente manera.

//@publicado_en

También se pueden seleccionar múltiples atributos con el operador @*. Para almacenar todos los atributos del elemento Libro en cualquier lugar del documento se puede usar:

*//Libro/@**

Además, XPath ofrece la posibilidad de hacer predicados para concretar los nodos deseados dentro del árbol XML. Por ejemplo, para encontrar todos los nodos Titulo con el valor El Capote:

/Libros/Libro/Titulo[.=”El Capote”]

El operador “[]” especifica un filtro y operador ”.” establece que ese filtro sea aplicado sobre el nodo actual que ha dado la selección previa. Los filtros son siempre evaluados con respecto al contexto actual. Alternativamente, se pueden encontrar todos los elementos Libro con título El Capote:

/Libros/Libro[./Titulo=”El Capote”]

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

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

3.2. XPath desde Java

Las clases necesarias para ejecutar consultas XPath son:

- *XpathFactory*. Esta clase contiene un método compile(), que comprueba si la sintaxis de una consulta es correcta y crea una expresión XPath.
- *XpathExpression*. Esta clase contiene un método evaluate(), que ejecuta un XPath.
- *DocumentBuilderFactory* y *Document*.

El siguiente código comentado muestra un ejemplo de cómo abrir un documento XML con DOM para ejecutar sobre el una consulta.

```
//Crea un DocumentBuilderFactory y un árbol DOM
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

```
Document documento=factory.newDocumentBuilder().parse(new
File(“Libros.xml”);

//Crea el objeto XpathFactory
Xpath xpath=XpathFactory.newInstance().newPath();

//Crea un XpathExpression con la consulta deseada
String xpathExpression=“/Libros/*/Autor”;

//Consultas
NodeList nodos=(NodeList) xpath.evaluate(xPathExpression, documento,
XpathConstants.NODESET);

for(int i=0;i<nodos.getLength();i++){
        System.out.println(nodos.item(i).getNodeName()+” : “+
        nodos.item(i).getTextContent());
}
```