3.- Acceso a documentos XML: SAX y DOM

1 Introducción

1.1 XML

XML, siglas en inglés de e**X**tensible **M**arkup **L**anguage ('lenguaje de marcas extensible'), es un lenguaje de marcas desarrollado por el ***World Wide Web Consortium (W3C)***.

XML no ciñe su utilización al ámbito de Internet, sino que se propone como un estándar para el **intercambio de información** estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

XML es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un **papel muy importante en la actualidad** ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

1.2 Analizadores sintácticos o "parsers"

Un **analizador sintáctico** (***parser*** en inglés) es una herramienta que convierte un texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Durante el análisis se crean *tokens* formados por secuencias de caracteres y son estos tokens los que procesa el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol.

Por ejemplo, una de las partes de un compilador es un analizador sintáctico. Éste nos avisa si el programa analizado no cumple con las reglas del lenguaje (errores de sintáxis).

1.3 Parsers XML

Cuando tenemos que acceder a documentos XML desde un programa, surge la necesidad de utilizar herramientas que faciliten la tarea de acceder a la información que el documento contiene. Se trata de no ver el documento como una mera secuencia de caracteres, sino de tener la posibilidad de acceder a la información estructurada y de reconocer las partes y elementos que lo componen.

En este sentido han surgido distintas API's, que aportan distintas características, ventajas e inconvenientes dependiendo del tipo de aplicación, las características del documento, el lenguaje de programación utilizado, …

Estas API (parsers XML) proporcionan mecanismos para acceder a la información contenida en el documento XML.

1.3.1 SAX - Simple API for XML Processing.

Fue el primero en llegar a la escena y requiere una buena comprensión del funcionamiento de XML. Inicialmente se desarrolló para Java pero su uso se ha extendido y, en la actualidad, hay API's de SAX para otros lenguajes de programación.

Existen muchos parsers XML que implementan el API SAX, incluyendo Xerces, Crimson, Oracle XML Parser y Ælfred. Eclipse por defecto incorpora el parser Xerces, de Apache.

Se trata de una tecnología basada en eventos. El documento se lee de manera secuencial y se generan eventos cada vez que se lee alguno de los elementos del documento.

La lectura secuencial implica que no es posible volver atrás sobre alguna de las partes leídas anteriormente. Si fuera necesario tendríamos que almacenar o gestionar los datos por programa.

El análisis con SAX tiene la ventaja de ser rápido y eficiente en cuanto a recursos consumidos (memoria) debido a que, por un lado, no soporta realizar cambios sobre el documento y, por otro, el documento no se almacena en memoria, simplemente se lee y se procesa.

1.3.2 DOM - Document Object Model

Su característica principal es que se pasa de un modelo basado en eventos a otro en el que el documento xml se lee y se almacena en memoria con una representación orientada a objetos con estructura de árbol.

Esto permite el acceso aleatorio a las partes del documento y posibilita la modificación del documento subyacente. En contrapartida, el procesamiento es más lento y también es mayor la cantidad de memoria necesaria. Esto lo hace poco útil cuando se trata de documentos de tamaño grande.

1.3.3 JAXP – (Java API for XML Processing)

Se trata de un API específico para manejar documentos XML desde Java. Es una capa que se sitúa entre el programa Java y el parser XML y trata de hacer al programa independiente de la tecnología subyacente que se esté utilizando.

1.3.4 JDOM

JDOM es una API de código abierto para manipular de datos XML optimizados para Java. A pesar de su similitud con DOM, es una alternativa que no está incluido en DOM. La principal diferencia es que mientras que DOM fue creado para ser un lenguaje neutral, JDOM se creó específicamente para usarse con Java y por lo tanto

beneficiarse de las características de Java, incluyendo sobrecarga de métodos, colecciones, etc.

1.3.5 SAX vs DOM

La siguiente tabla muestra, de forma resumida las principales diferencias entre SAX y DOM.

	SAX	DOM
Tipo de interface	Basado en eventos	Orientado a objetos
Modelo de objetos	Debe ser creado por la aplicación	Se crea automáticamente
Orden de los elementos	Se pierde	Se conserva el orden de los elementos
Uso de memoria	Bajo	Alto
Tiempo necesario para obtener el primer elemento	Bajo	Alto
Posibilidad de validación	Si	Si
Posibilidad de actualizar el XML	No	Si (en memoria)

2 SAX

2.1 Procedimiento resumido

De forma resumida, podemos decir que en un programa que realiza el análisis de un documento XML intervienen **dos elementos**:

- Un lector de xml (XMLReader).
- Un gestor de contenido (ContentHandler).

El gestor de contenido (ContentHandler) se asocia al lector (XMLReader) y, cada vez que lee un elemento relevante del documento, ejecuta el método correspondiente del gestor.

En el siguiente ejemplo, reproducimos (parcialmente) por pantalla el contenido de un fichero xml (ciclos.xml)

En el programa principal …

```
public class LeerXML {
   public static void main(String[] args) throws IOException, SAXException {
      XMLReader p = XMLReaderFactory.createXMLReader();
      p.setContentHandler(new CiclosHandler());
```

```
            p.parse("Ciclos.xml");
        }
    }
```

… vemos como

- se obtiene un XMLReader a partir de XMLReaderFactory
- se crea un objeto de CiclosHandler (nuestro ContentHandler)
- se asocia CiclosHandler para que maneje los eventos del XMLReader.
- se analiza el docuemento ciclos.xml

El gestor de contenido será el siguiente:

```
public class CiclosHandler extends DefaultHandler {

    @Override
    public void startDocument() throws SAXException {
        System.out.println("Comienza el analisis");
    }

    @Override
    public void endDocument() throws SAXException {
        System.out.print("\nFinaliza el analisis");
    }

    @Override
    public void startElement(String uri, String localName, String qName,
            Attributes attributes) throws SAXException {
        System.out.print("<"+qName+">");
    }

    public void endElement(String uri, String localName, String qName)
            throws SAXException {
        System.out.print("</"+qName+">");
    }

    @Override
    public void characters(char[] ch, int start, int length)
            throws SAXException {
        System.out.print(new String(ch,start,length));
    }

}
```

La clase hereda de DefaultHandler y sobreescribe sus métodos. Los métodos originales de DefaultHandler no realizan ninguna acción.

Los métodos que se han reescrito son:

- startDocument(): Se ejecuta cuando comienza el documento xml.
- endDocument(): Se ejecuta cuando finaliza el documento xml.
- startElement(): Se ejecuta al comienzo de cada elemento. Observa que hay un parámetro attributes que no aparece en endElement()
- endElement(): Se ejecuta al final de cada elemento.

- Characters() Se ejecuta cada vez que se lee alguna secuencia de caracteres. La secuencia leída queda determinada por una posición y longitud dentro del array ch[]

2.2 Clases e interfaces que intervienen

- **Interface org.xml.sax.XMLReader:**

Esta es la interfaz que debe implementar un lector de XML.

Cada vez que el XMLReader encuentra el principio del archivo, el final, un elemento, un caracter especial, un espacio, etc. notifica esto al ContentHandler asociado. Cada vez que encuentra un error se lo notifica al ErrorHandler asociado.

- **Interface org.xml.sax.ContentHandler:**

La clase que la implementa obtiene la capacidad de recibir todas las notificaciones de contenido de un XMLReader.

La forma de asociar este a un XMLReader es mediante el método setContentHandler.

- **Interface org.xml.sax.ErrorHandler:**

La clase que la implementa obtiene la capacidad de recibir todas las notificaciones de error producidas por un XMLReader.

La forma de asociar este a un XMLReader es mediante el método setErrorHandler. En caso de que no exista ningun ErrorHandler asociado al XMLReader, este último no reportara ningun error, salvo las excepciones SAXParseException.

- **Clase org.xml.sax.helpers.DefaultHandler:**

Clase que implementa tanto a ContentHandler como a ErrorHandler (además de DTDHandler y EntityResolver que no veremos por ahora), proveiendo implementaciones por defecto para todos sus métodos.

Esta clase es de la que extenderemos para poder crear nuestro propio parser de XML.

- **Clase org.xml.sax.helpers.XMLReaderFactory:**

Clase que provee métodos estáticos para la creación de XMLReaders. Utilizaremos el método estático XMLReaderFactory.createXMLReader() que nos devuelve el XMLReader un por defecto para nuestro sistema.

2.3 Manejo de errores (ErrorHandler)

Además de disponer del interface ContentHandler para manejar los eventos que suceden durante el análisis, SAX proporciona un interface ErrorHandler que se puede implementar para tratar las condiciones de error que se producen. El interface ErrorHandler define tres métodos que es posible sobreescribir para manejar los errores:

- public abstract void warning (SAXParseException e) throws SAXException;
- public abstract void error (SAXParseException e) throws SAXException;
- public abstract void fatalError (SAXParseException e) throws SAXException;

Cada método recibe información del error o el warning que se ha producido a través de una SAXParseException. Este objeto contiene el número de línea donde se ha producido el problema, la URI del documento que se está analizando y otra información relativa a la excepción. Al mismo tiempo, cada método puede lanzar una SAXException.

La clase DefaultHandler implementa tanto a ContentHandler como a ErrorHandler, por lo que podemos implementar nuestro manejador de errores heredando también de de DefaultHandler, como lo hemos hecho para implementar el manejador de contenido.

2.3.1 Warnings

Cuando se produce un warning, se ejecuta el método ***warning*** del gestor de errores. Un warning notifica alguna condición que no es grave y la gestión del error debería consistir en informar de la situación y permitir continuar con el análisis del documento.

¿Qué situaciones producen un warning? Las descritas en el ***xml 1.0 Recommendation*** (http://www.w3.org/TR/2000/REC-xml-20001006#sec-terminology) pero, en general, están relacionadas con el DTD y la validez del documento.

2.3.2 Errores no fatales

Son errores que suceden durante el análisis del documento de los que es posible recuperarse, aunque constituyen una violación de alguna parte de la especificación del XML. También están relacionados con la validación del docuemento.

Una gestión adecuada consistiría en informar o hacer un log del error y continuar con el análisis.

En el apartado 1.2 de las ***xml 1.0 recommendation*** se define lo que es un error.

Cuando se produce un error no fatal, se ejecuta el método ***error*** del gestor de errores.

2.3.3 Errores fatales

Son los errores que requieren detener el análisis. Generalmente suceden cuando el documento no está bien formado y sería una pérdida de tiempo tratar de analizarlo. La gestión del error consistiría en informar y lanzar una SAXException para que el análisis se detenga.

2.3.4 Ejemplo 1

En el siguiente ejemplo se muestra un manejador de errores sencillo. Se reescriben los tres métodos para mostrar un mensaje por pantalla. **Observa** como el método error lanza **SAXException** para finalizar el proceso de análisis.

Nota: Aunque no lanzaramos la SAXException, cuando se produce un error fatal, el parser (por defecto) abandona el análisis. Sin embargo es posible configurar el parser de Xerces para que continúe analizando después de un error fatal.

```
public class ErroresHandler extends DefaultHandler{

	@Override
	public void warning(SAXParseException e) throws SAXException {
		System.out.println("WARNING: " + e.toString());
		//El proceso continua
	}

	@Override
	public void error(SAXParseException e) throws SAXException {
		System.out.println("ERROR NO FATAL: " + e.toString());
		//El proceso se continua
	}

	@Override
	public void fatalError(SAXParseException e) throws SAXException {
		System.out.println("ERROR FATAL: " + e.toString());
		//El proceso se detiene
		throw new SAXException("Error de parsing");
	}

}
```

En el programa principal tendremos que asociar el manejador de errores al lector de xml:

```
public class LeerXML {
   public static void main(String[] args) throws IOException, SAXException {
      XMLReader p = XMLReaderFactory.createXMLReader();
      p.setContentHandler(new CiclosHandler());
      p.setErrorHandler(new ErroresHandler());
      p.parse("Ciclos.xml");
   }
}
```

2.3.5 Ejemplo 2

En este ejemplo, el gestor de errores está un poco más elaborado. Guarda en un fichero de log los warning, errores y errores fatales que se producen. **Observa** como si se produce algún error durante la escritura del log, se lanza SAXException. He aquí la razón por la que warning y error propagan dicha excepción.

```
public class ErroresHandlerLog extends DefaultHandler{

	@Override
	public void warning(SAXParseException e) throws SAXException {
		FileWriter f = null;
		try {
			f = new FileWriter("ParsingCiclos.log",true);;
			f.write("WARNING: " + e.toString());
		} catch (IOException ex) {
			throw new SAXException("Error al escribir log");
		} finally{
			if(f!=null)
				try {
					f.close();
				} catch (IOException e1) {
					throw new SAXException("Error al cerrar log");
				}
		}
	}
	@Override
	public void error(SAXParseException e) throws SAXException {
		FileWriter f = null;
		try {
			f = new FileWriter("ParsingCiclos.log",true);;
			f.write("ERROR NO FATAL: " + e.toString());
		} catch (IOException ex) {
			throw new SAXException("Error al escribir log");
		} finally{
			if(f!=null)
				try {
					f.close();
				} catch (IOException e1) {
					throw new SAXExce    ption("Error al cerrar
log");
				}
		}
	}
	@Override
	public void fatalError(SAXParseException e) throws SAXException {
		FileWriter f = null;
		try {
			f = new FileWriter("ParsingCiclos.log",true);;
			f.write("ERROR FATAL: " + e.toString());
			throw new SAXException("Error de parsing");
		} catch (IOException ex) {
			throw new SAXException("Error al escribir log");
		} finally{
			if(f!=null)
				try {
					f.close();
				} catch (IOException e1) {
					throw new SAXException("Error al cerrar log");
				}
		}
	}
}
```

2.4 Validación

Un documento xml se puede validar contra un DTD o un esquema.

Para ello:

1. Se activa la característica de validación del parser que estemos utilizando.
2. Se analiza el documento realizando el manejo de errores.

Ejercicio paso a paso:

1. Crea el siguiente ciclosformativos.dtd, que se corresponde con la estructura que debe tener el fichero ciclos.xml. Almacénalo en la misma ruta que ciclos.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT ciclosformativos (familia*)>
<!ELEMENT familia (nombre,ciclo+)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT ciclo (titulo,curso+)>
<!ELEMENT titulo (#PCDATA)>
<!ELEMENT curso (modulo+)>
<!ELEMENT modulo (#PCDATA)>

<!ATTLIST ciclo grado  ( medio | superior ) #REQUIRED>
<!ATTLIST curso numero  ( 1 | 2 ) #REQUIRED>

<!ATTLIST modulo horas CDATA "">
```

2. En ciclos.xml añade la siguiente línea

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ofertaformativa SYSTEM "ofertaformativa.dtd">
```

3. Confecciona un ContentHandler sencillo, que muestre, simplemente, el contenido del fichero (como el que veíamos en el punto 2.1).
4. Ejecútalo y **observa que no aparecen en pantalla los saltos de línea ni las tabulaciones.**

   Cuando un documento xml está asociado a un dtd, las tabulaciones y saltos de línea no llegan al método characters sino al método DefaultHandler.ignorableWhitespaces. Si quisiéramos tratar esta información tendríamos que sobreescribir este método en el nuestro manejador de contenido:

```
@Override
public void ignorableWhitespace(char[] ch, int start, int length)
            throws SAXException {
      …
}
```

5. Introduce algún cambio en ciclos.xml que haga que no cumpla con el dtd. Por ejemplo, elimina el atributo número (que es obligatorio) de algún curso. Ejecuta el programa y observa: **No se produce ningún error, a pesar de que el documento no se corresponde con el dtd.** Esto es debido a que el parser tiene la validación desactivada por defecto.

   Con la validación desactivada se producen errores cuando el xml no está bien formado (se abre un elemento y no se cierra, faltan unas comillas, etc). No se producen errores cuando el documento no es válido (no cumple con el dtd)

6. Activa la validación en el parser. Para ello, añade la siguiente línea al programa principal:

```
public class LeerXML {
    public static void main(String[] args) throws IOException, SAXException {
            XMLReader p = XMLReaderFactory.createXMLReader();
            p.setContentHandler(new CiclosHandler());
            p.setErrorHandler(new ErroresHandler());

            p.setFeature("http://xml.org/sax/features/validation", true);

            p.parse("Ciclos.xml");

    }
}
```

   Al ejecutar, **se observa** que el manejador de errores recibe una notificación de error porque no aparece el atributo ***numero*** del elemento curso. A pesar de ello, como se trata de un error no fatal, el análisis continúa.

3 DOM

En primer lugar, insistimos en cuál es la forma de trabajo con DOM: El documento xml se procesa y se carga por completo en memoria, utilizando una jerarquía de elementos que ahora veremos. Una vez que el documento está en memoria podemos recorrerlo arriba y abajo tantas veces como queramos.

DOM es una especificación del W3C. Es la especificación de un API que guarda la información de documentos HTML y XML en forma de árbol.

DOM ha sido diseñado para que pueda utilizarse desde cualquier lenguaje de programación. Para ello, DOM se ha definido en OML IDL, un lenguaje “neutral” que permite definir interfaces de forma independiente al lenguaje de programación. Además el W3C ha realizado una definición específica de DOM para Java. En definitiva, DOM, es un conjunto de interfaces que, los *parsers* de distintos distribuidores se encargarán de implementar. En java, el parser DOM por defecto es el de Xerces.

Aunque no entraremos en detalle de momento, esa *independencia el lenguaje,* es la causa de que los elementos con los que trabajamos en DOM presenten algunas características que llaman, como mínimo, la atención.

- En ocasiones, varios métodos de un objeto devuelven hacen lo mismo o proporcionan la misma información.
- En ocasiones, un objeto tiene métodos *inútiles,* que no hacen nada o no devuelven nada para ese tipo de objeto.

Esto es debido a que, en DOM, distintos tipos de elementos xml, como un comentario, un elemento, un atributo, … se almacenan en memoria de la misma forma: todos son *nodos (*Objetos pertenecientes al interface Node).

Por último, decir que DOM se apoya a su vez en SAX por lo que los paquetes o intefaces que podemos estar manejando en ocasiones serán los que ya hemos trabajado con SAX.

3.1 Cargar un documento xml en memoria.

El proceso de parsear el documento xml y cargarlo en memoria es sencilloy se realiza en un solo paso. Se corresponde con la instrucción *parse* del siguiente programa:

```
public class LeerXML {
    public static void main(String[] args){
        DocumentBuilderFactory factory = null;
        DocumentBuilder builder= null;
        try {
          factory = DocumentBuilderFactory.newInstance();
          builder = factory.newDocumentBuilder();

          Document doc = builder.parse("ciclos.xml");
          doc.normalize();

          //Elemento raiz
          Element raiz = doc.getDocumentElement();
          System.out.println("Elemento raiz: " + raiz.getNodeName());

          System.out.println("Tipo de documento: " + doc.getDoctype().getName());
          System.out.println("Encoding: " + doc.getXmlEncoding());
          System.out.println("Versión de xml: " + doc.getXmlVersion());

        } catch (ParserConfigurationException e) {
          System.out.println("Problema al crear el DocumentBuilder");
          e.printStackTrace();
        } catch(SAXException e){
          System.out.println("Error al parsear");
          e.printStackTrace();
        } catch(IOException e){
          System.out.println("Error de E/S");
          e.printStackTrace();
        }
    }
}
```

El paquete javax.xml.parsers contiene dos clases:

- DocumentBuilderFactory

    - Tiene el método estático newInstance(), que se encarga de averiguar cuál es nuestra implementación de la clase *DocumentBuilderFactory* y crear un objeto de dicha clase.
- DocumentBuilder
    - DocumentBuilderFactory permite, usando newDocumentBuilder() crear objetos de tipo DocumentBuilder.
    - La idea es que, manipulando ciertas propiedades del DocumentBuilderFactory, como por ejemplo el soporte para validación, el soporte para espacios de nombres, etc, crearíamos DocumentBuider capaces de manejar documentos que incorporan o no esas características (validación, espacios de nombres, …).

El método ***parse*** está sobrecargado. Permite pasarle como parámetro el nom**bre de un fichero** pero también, por ejemplo, un ***FileInputStream***. El método parse devuelve un objeto perteneciente al interface ***Document***.

El método normalice() elimina nodos de texto vacios y une nodos de texto adyacentes. Evita que se generen elementos de texto consecutivos que no estén separados por otros como etiquetas, comentarios, etc … (Ver *http://stackoverflow.com/questions/13786607/normalization-in-dom-parsing-with-java-how-does-it-work*)

3.2 El interface DOCUMENT.

Como hemos dicho en varias ocasiones el análisis de un fichero xml usando DOM carga en memoria el contenido del fichero en forma de estructura jerárquica. ***Document*** es el elemento principal de dicha estructura.

Como hemos visto en el ejemplo anterior, existe cierta información que podemos consultar del documento:

```
System.out.println("Tipo de documento: " + doc.getDoctype().getName());
System.out.println("Encoding: " + doc.getXmlEncoding());
System.out.println("Versión de xml: " + doc.getXmlVersion());
```

- El DocumentType, que permitiría obtener información sobre el DTD (tipo de documento)
- La página de códigos Unicode utilizada: getXmlEnconding
- La versión de xml
- …

Un Document contiene un único elemento (Element): su elemento raíz. Par obtenerlo usamos el método getDocumentElement(). Todos los demás nodos del árbol partirán del nodo raíz del docuemento. Como vemos si ejecutamos el programa, el nodo raíz se corresponde con el elemento del fichero xml que contiene a todos los demás, en nuestro caso, las etiquetas <ciclosformativos> </ciclosformativos>

Además Document dispone de una serie de métodos **createXXX()** que permiten crear y añadir al docuemto nodos de tipos específicos (tags, atributos, comentarios,

elementos de texto, etc). Con esto, queremos llamar la atención sobre una característica de DOM: los documentos resultantes del *parsing* son **editables.**

Document, permite obtener **listas de elemento filtrando por su tagName**. Como se ve en el siguiente ejemplo, las listas de elementos se devuelven en forma de objetos NodeList.

Un NodeList dispone de dos métodos

- getLength(), que devuelve el número de elementos de la lista
- ítem(idex), que devuelve el elemento que se solicita (el primero es el cero):

```
public class MostrarImagenesEnHtml {
      public static void main(String[] args){
            DocumentBuilderFactory factory = null;
            DocumentBuilder builder= null;
            try {
                  factory = DocumentBuilderFactory.newInstance();
                  builder = factory.newDocumentBuilder();
                  Document doc = builder.parse("index.xhtml");

                  NodeList imagenes = doc.getElementsByTagName("img");
                  for(int i=0 ; i<imagenes.getLength();i++){
                      Element e = (Element) (imagenes.item(i));
                      System.out.println(e.getAttribute("src"));
                  }

            } catch (ParserConfigurationException e) {
                  …
                  …

      }
}
```

3.3 El interface NODE

Esta es la interfaz base ya que **todos los objetos de un árbol DOM son nodos**.

Todos los nodos tienen un campo ownerDocument que referencia al documento que lo contiene (a excepción de los documentos, cuyo ownerDocument es null). La mayoría de los nodos tienen un nodo padre, y pueden tener un hermano anterior y un hermano siguiente (previousSibling y nextSibling). Para obtener estos objetos llamamos a los métodos ***getOwnerDocument(), getParentNode(), getPreviousSibling() y getNextSibling()*** respectivamente. Estos métodos permiten recorrer todo el árbol de nodos.

Hay distintos tipos de nodos. En tipo de un nodo se obtiene con getNodeType(). Es un valor entero y en el interface Node se definen constantes que representan a los posibles valores

NodeType	Constante
1	ELEMENT_NODE
2	ATTRIBUTE_NODE
3	TEXT_NODE
4	CDATA_SECTION_NODE
5	ENTITY_REFERENCE_NODE
6	ENTITY_NODE
7	PROCESSING_INSTRUCTION_NODE
8	COMMENT_NODE
9	DOCUMENT_NODE
10	DOCUMENT_TYPE_NODE
11	DOCUMENT_FRAGMENT_NODE
12	NOTATION_NODE

Todos los nodos tienen un nombre y un valor. Para obtener estas propiedades usamos ***getNodeName()*** y ***getNodeValue()*** respectivamente. Realmente la mayoría de los nodos devuelven null en getNodeValue() y la mayoría de los nodos devuelven un valor constante en getNodeName() tales como #document, #comment, etc. En la siguiente tabla se indica que devuelven éstos métodos dependiendo del tipo de nodo de que se trate.

Tipo de nodo	nodeName devuelve	nodeValue devuelve
Document	#document	null
DocumentFragment	#document fragment	null
DocumentType	doctype name	null
EntityReference	entity reference name	null
Element	element name	null
Attr	attribute name	attribute value
ProcessingInstruction	target	content of node
Comment	#comment	comment text
Text	#text	content of node
CDATASection	#cdata-section	content of node
Entity	entity name	null
Notation	notation name	null

No todos los nodos pueden tener hijos pero, debido a que la interfaz Node provee métodos de todos los tipos de nodos, Node tiene métodos para manipular los nodos hijos de cualquier nodo (pueda tener hijos o no):

- Podemos obtener el primer nodo hijo mediante getFirstChild();
- podemos obtener el último nodo hijo mediante getLastChild()
- y podemos obtener una lista (NodeList) de hijos mediante getChildNodes().

En esta tabla se muestra una descripción de los distintos tipos de nodos y cuáles son sus posibles hijos

Node type	Description	Children
Document	Represents the entire document (the root-node of the DOM tree)	Element (max. one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Represents a "lightweight" Document object, which can hold a portion of a document	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	Provides an interface to the entities defined for the document	None
ProcessingInstruction	Represents a processing instruction	None
EntityReference	Represents an entity reference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Represents an element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Represents an attribute	Text, EntityReference
Text	Represents textual content in an element or attribute	None
CDATASection	Represents a CDATA section in a document (text that will NOT be parsed by a parser)	None
Comment	Represents a comment	None
Entity	Represents an entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	Represents a notation declared in the DTD	None

En el siguiente ejemplo se recorre el árbol utilizando getFirstChild() y getLastChild()

```
public class MostrarRecursivoConFirstChild {
   public static void main(String[] args){
        DocumentBuilderFactory factory = null;
        DocumentBuilder builder= null;
        try {
            factory = DocumentBuilderFactory.newInstance();
            builder = factory.newDocumentBuilder();
            Document doc = builder.parse("ciclos.xml");

            Element raiz = doc.getDocumentElement();

            mostrarHijos(raiz,0);
        } catch (ParserConfigurationException e) {
            System.out.println("Problema al crear el DocumentBuilder");
            e.printStackTrace();
        } catch(SAXException e){
            System.out.println("Error al parsear");
            e.printStackTrace();
        } catch(IOException e){
            System.out.println("Error de E/S");
            e.printStackTrace();
        }
   }
   public static void mostrarHijos(Node n, int identacion){
        identar(identacion);
        mostrarNodo(n);
        for(Node hijo = n.getFirstChild(); hijo!= null; hijo= hijo.getNextSibling()){
            mostrarHijos(hijo, identacion+1);
        }
   }
   public static void mostrarNodo(Node n){
        switch(n.getNodeType()){
        case Node.TEXT_NODE:
               System.out.println(n.getNodeValue().trim());
               break;
        case Node.COMMENT_NODE:
               System.out.println("Comentario: " + n.getNodeValue().trim());
               break;

        default: System.out.println(n.getNodeName());
        }
   }
   public static  void identar(int veces){
        for(int i = 1; i<=veces; i++){
            System.out.print("  ");
        }
   }
}
```

El método mostrarHijos recorre los hijos del nodo que recibe y los muestra. Es un método recursivo.

Éste es el mismo ejemplo utilizando getChildNodes(). Sólo cambia el método mostrarHijos, por lo que el resto de código se ha omitido.

```
public class MostrarRecursivoConChildNodes {
   public static void main(String[] args){
    …
   }
   public static void mostrarHijos(Node n, int identacion){
      identar(identacion);
```

```
      mostrarNodo(n);
      NodeList hijos = n.getChildNodes();
      for(int i = 0; i<hijos.getLength();i++){
         mostrarHijos(hijos.item(i), identacion+1);
      }
   }
   public static void mostrarNodo(Node n){
      …

   }
   public static  void identar(int veces){
      …
   }
}
```

getChildNodes devuelve una lista de nodos que se puede recorrer. No se trata de un objeto de tipo Collection, por lo que no se puede usar un iterator.

3.4 El interface Element

Los **elementos** son quizá los nodos más comunes dentro de un árbol DOM. Se corresponden con los ***tags*** de XML y HTML. Los elementos tienen atributos y pueden tener nodos hijos.

Además los elementos pueden tener un espacio de nombres asociados y un prefijo. Junto a los atributos son los únicos nodos que soportan espacios de nombre. En caso de tener un espacio de nombres asociado el elemento tendrá cuatro propiedades: el prefijo, el nombre local, la URI del espacio de nombres y el qualifiedName que es prefijo+":"+localName. Todos ellos son objetos String.

Los atributos de un elemento se pueden obtener mediante ***getAttributes()*** que devuelve una lista de NamedNodeMap. Sólo los tipos de nodo Element pueden poseer atributos, aunque el método aparezca en el interface Node. El método ***hasAttributes()*** devuelve si un elemento tiene atributos o no.

El tipo de lista NamedNodeMap es una lista desordenada en la que los nodos se pueden acceder a través de su nombre. La interfaz NamedNodeMap, al igual que NodeList, tiene un método getLength() y un método item(int index) que permite hacer un recorrido de los nodos que forman parte de la lista.

Ejercicio: Ampliar el ejemplo anterior para que se muestren también los atributos de cada elemento.

3.5 El interface Attr

Los atributos son un tipo de nodo muy característico. Sólo existen dentro de elementos y **no forman parte del árbol**. Estrictamente no tienen un nodo padre, es decir, getParentNode() devuelve null, y tampoco tienen nodos hermanos (getNextSibling() y getPreviousSibling() devuelven null). Sin embargo sí que pueden tener nodos hijo (solamente de los tipos Text y EntityReference, y sólo en documentos XML).

Junto a los elementos son los únicos nodos que tienen soporte para espacios de nombre. El soporte de espacios de nombre es idéntico que en la interfaz Element.

Los atributos son de los pocos tipos de nodo que contienen algún valor. Existen dos métodos redundantes: getName() y getValue() que devuelven lo mismo que getNodeName() y getNodeValue() respectivamente.

Un atributo no tiene un nodo padre, sin embargo, sí tiene un elemento propietario. Este elemento se puede obtener mediante el método getOwnerElement().

Los atributos pueden haber adquirido su valor explícitamente a través del método setValue() o setNodeValue(), o pueden tener un valor por defecto definido en el DTD del documento. En caso de tener un valor explícito, el método getSpecified() devolverá true, y devolverá false en caso contrario.

3.6 XPath

XPath (XML Path Language) es un lenguaje que permite construir expresiones que recorren y procesan un documento XML.

Básicamente se trata de describir una ruta a través del árbol de nodos como si cada etiqueta fuera un directorio. Por ejemplo, /etiquetaPrincipal/etiquetaHija. Esta ruta XPath seleccionaría la etiquetaHija de nuestro XML.

A esto se añade la posibilidad de utilizar funciones, caracteres comodín, condicionales, etc …

Usar XPath desde Java:

```
public class NombreDeFamilias {
   public static void main(String[] args){
      DocumentBuilderFactory factory = null;
      DocumentBuilder builder= null;
      try {
         factory = DocumentBuilderFactory.newInstance();
         builder = factory.newDocumentBuilder();
         XPath xpath = XPathFactory.newInstance().newXPath();

         Document doc = builder.parse("ciclos.xml");
         doc.normalize();

          // Buscamos nombes de familia mediante XPath.
         NodeList  familias = (NodeList)
             xpath.evaluate("/ciclosformativos/familia/nombre",
                         doc, XPathConstants.NODESET);
```

```
            for(int i=0; i<familias.getLength();i++){
                System.out.println(familias.item(i).getTextContent());
            }

        } catch (ParserConfigurationException e) {
            System.out.println("Problema al crear el DocumentBuilder");
            e.printStackTrace();
        } catch(SAXException e){
            System.out.println("Error al parsear");
            e.printStackTrace();
        } catch(IOException e){
            System.out.println("Error de E/S");
            e.printStackTrace();
        } catch (XPathExpressionException e){
            System.out.println("Error de expresión XPath");
            e.printStackTrace();
        }
    }

}
```

En el ejemplo, **observa**:

- Se crea un objeto de tipo **XPath**, y con su método **evaluate** se realizan las consultas XPath
- evaluate recibe
    - El path a consultar
    - El documento sobre el que se trabaja.
    - El tipo a devolver en forma de XPathConstants. En el ejemplo se devuelve un NODESET(Conjunto de nodos), que correspondería con un NodeList. Fíjate en que hay que hacer casting.
- getTextContent devuelve el texto hijo de la etiqueta nombre
- Se ha añadido un nuevo catch para capturar las excepciones de xpath.

Aquí se indican algunos enlaces a tutoriales de xPath.

- Explicado con ejemplos:
    - http://zvon.org/xxl/XPathTutorial/Output_spa/examples.html
- Uno en inglés
    - http://www.w3schools.com/xpath/default.asp
- Este es más flojo
    - http://geneura.ugr.es/~victor/cursillos/xml/XPath/
- Uno, más formal, en pdf
    - http://www.uhu.es/josel_alvarez/xNvasTecnProg/material/tutorial_XPATH.pdf