IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma

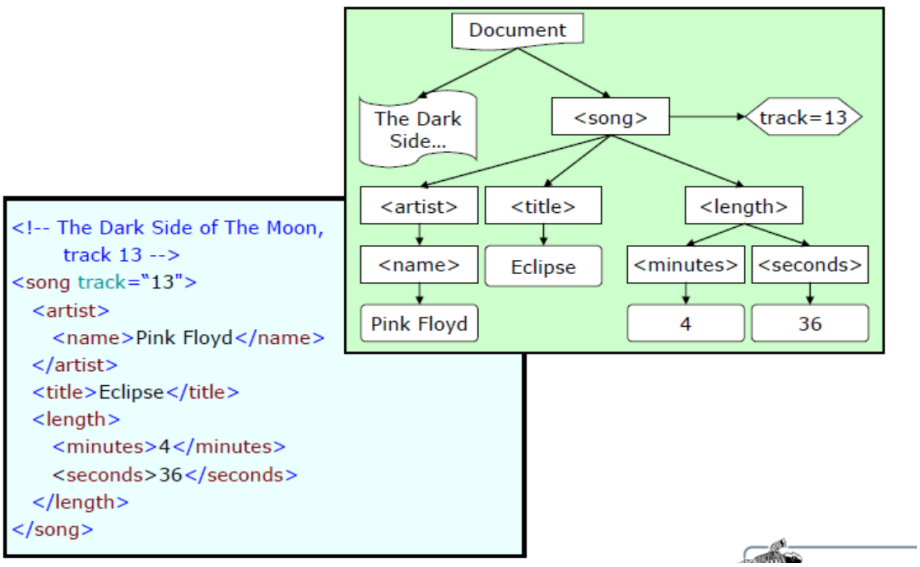
UNIDAD 1 Parte 2: Gestión de Ficheros XML

Índice

1. Acceso a ficheros XML con DOM

El Java **API DOM** para analizar XML está diseñado para trabajar con XML como **un árbol de objetos en la memoria** - un "Document Object Model (DOM)". El analizador accede al archivo XML y crea los correspondientes objetos DOM. Estos objetos DOM están unidos entre sí en una estructura de árbol. Una vez que el analizador se crea, se obtiene una estructura de objetos DOM en la memoria. A continuación, se puede acceder y recorrer esta estructura DOM para realizar las operaciones deseadas.

Ejemplo de vista de un objeto DOM en memoria:



Para **utilizar el API DOM** es necesario importar ciertas bibliotecas:

- Paquete **javax.xml.parsers** contiene las clases de utilidades que nos permiten crear, acceder, y analizar documentos XML.
- Paquete **org.w3c.dom** contiene las clases que se encarga de los componentes DOM (document, element, Attr, node…).
- Paquete **org.xml.sax.SaxException**, aunque estemos utilizado analizador DOM, este paquete es necesario para la implementación del analizador.

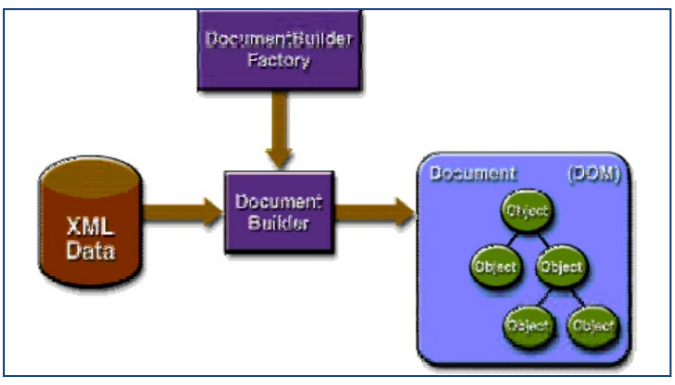
Todos estos paquetes son parte de la versión de java estándar.

1.1 El paquete javax.xml.parsers

Este paquete es el punto de partida para usar DOM en Java. Contiene dos clases fundamentales: **DocumentBuilderFactory y DocumentBuilder**.

En DOM, un documento XML se representa como un árbol. Un analizador DOM genera automáticamente dicho árbol a partir de un documento XML

Crea la estructura completa en memoria, de forma que puedan utilizarla las aplicaciones que trabajen con el documento DOM proporciona los métodos, clases e interfaces necesarios para acceder al árbol y manipular su estructura o modificar sus elementos



Obteniendo una instancia de DocumentBuilderFactory

Lo primero que tenemos que hacer es analizar el documento XML, construyendo su representación DOM y almacenándolo en memoria de forma que el API DOM pueda explorarlo.

La **clase DocumentBuilderFactory** posee un **método estático** **newInstance()** que permite obtener una implementación del analizador.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

newInstance

```
public static DocumentBuilderFactory newInstance()
                                        throws FactoryConfigurationError
```

La factoría DocumentBuilderFactory nos permite **crear analizadores (parses) con características de comportamiento determinadas**. Estos comportamientos son: validación, soporte para espacios de nombres, etc.

Lanza la excepción ***FactoryConfigurationError*** , si la implementación no está disponible y no se puede instanciar.

Podemos **obtener información sobre estos comportamientos** a través de métodos **isXXX()** y podemos **establecerlos** a través de métodos **setXXX().**

Después de establecer estas propiedades, la factoría devolverá **objetos DocumentBuilder** que poseen un analizador con los comportamientos deseados

Estableciendo el comportamiento: La clase DocumentBuilder

Obtener una instancia de DocumentBuilder es bien sencillo. Basta invocar el método newDocumentBuilder() de la factoria.

Si la factoría no es capaz de crear objetos **DocumentBuilde**r con dichos comportamientos se lanzará una excepción del tipo: **javax.xml.parsers.ParserConfigurationException**.

```
try{
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder documentBuilder = dbf.newDocumentBuilder();
    System.out.println("Obtenido DocumentBulider con exito");
}catch(javax.xml.parsers.ParserConfigurationException e){
    System.err.println("No se ha podido crear una instancia de DocumentBuilder");
}catch (javax.xml.parsers.FactoryConfigurationError e){
System.err.println("No se ha podido crear una instancia
                    de DocumentBuilderFatory");

}
```

Ejemplo de comprobación y establecimientos de comportamiento de los analizadores construidos.

```
package ejaccesoficherosxml;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

/**@author Maria Jose Galán López */
public class EjAccesoFicherosXML {

public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

//Posibles comportamientos que se pueden establecer:
//comprueba si la factoría esta configurada para producir parsers que validen el
//documento
   System.out.println("isValidating: \t\t\t"+ dbf.isValidating());
```

```
//comprueba si los parsers producidos soportaran espacios de nombres
   System.out.println("isNamespaceAware: \t\t"+ dbf.isNamespaceAware());
//comprueba si los parsers producidos por la factoría convertirán bloques CDATA a
//nodos Text
   System.out.println("isCoalescing: \t\t\t"  + dbf.isCoalescing());
//comprueba si los parsers producidos ignoraran los comentarios
   System.out.println("isIgnoringComments\t\t"+ dbf.isIgnoringComments());
//Podemos establecerlos con metodos setXXX()
   dbf.setValidating(false);
   System.out.println("validacion establecida a false");
   dbf.setNamespaceAware(true);
   System.out.println("establecido soporte para espacios de nombres");
//etc.
 try {
       DocumentBuilder constructor = dbf.newDocumentBuilder();
       System.out.println("Obtenido DocumentBulider con exito");
     } catch (javax.xml.parsers.ParserConfigurationException e) {
        System.err.println("No se ha podido crear una instancia
                           de DocumentBuilder");
       }
    }
}
```

Creando documentos

Para **un nuevo documento** basta con invocar al método **newDocument()** de **DocumentBuilder**.

newDocument	
public abstract **Document** newDocument()	Obtiene una nueva instancia de un objecto DOM Document para construir un arbol DOM

Ejemplo:

```
Import org.w3c.dom.Document;

Document documento = constructor.newDocument();
```

Así obtendremos un documento vacío. Un documento vacío no es un documento bien formado ya que un documento debe contener al menos un, y no más de un, elemento raíz. Más adelante veremos una forma más adecuada para crear documentos a través de la interfaz org.w3c.dom.DOMImplementation. Esta interfaz nos permitirá crear documentos con un elemento raíz y opcionalmente con un DTD asociado.

Construcción de un árbol DOM vacío:

```
try {
DocumentBuilderFactory dbf =DocumentBuilderFactory.newInstance();
DocumentBuilder constructor = dbf.newDocumentBuilder();
Document documento = constructor.newDocument();
}
catch (ParserConfigurationException e) { ... }
```

Cargando documentos

Para **cargar documentos ya creados** se utiliza el método **parse** de **DocumentBuilder**.

Podemos cargar un documento desde un archivo, indicando una URI, un objeto file o a través de un InputStream. También podemos usar la clase org.xml.sax.InputSource que encapsula cualquier fuente de datos.

Se han aprovechado algunas clases del API SAX como InputSource y SAXException para crear documentos DOM.

parse	
public **Document** parse(**InputStream is**) throws **SAXException, IOException**	Carga y analiza el documento XML proporcionado como un objeto InputStream y retorna un nuevo objecto DOM Document. Una IllegalArgumentException puede lanzarse si el objecto InputStream es null.
public **Document** parse(String uri) throws **SAXException, IOException**	Carga y analiza el documento XML proporcionado como una URI y retorna un nuevo objecto DOM Document. Una IllegalArgumentException puede lanzarse si la URI es null.
public **Document** parse(File f) throws **SAXException, IOExceptionParse**	Carga y analiza el documento XML proporcionado como un objeto File y retorna un nuevo objecto DOM Document. Una IllegalArgumentException puede lanzarse si el objecto File es null.
public **Document** parse(**InputSource is**) throws **SAXException, IOException**	Carga y analiza el documento XML proporcionado como un objeto InputSource y retorna un nuevo objecto DOM Document. Una IllegalArgumentException puede lanzarse si el objecto InputStream es null.

Construcción de un árbol DOM a partir de un fichero XML:

```
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

/** @author Maria Jose Galán López */
public class Ej2AccesoaFicherosXML {
    public static void main(String[] args) {
        DocumentBuilderDbf dbf = DocumentBuilderFactory.newInstance();
        String XMLDocument="C:\\FicherosJava\\XLM\\archivo.xml";
        Document documento =null;
        try {
            DocumentBuilder constructor= dbf.newDocumentBuilder();
            documento =  constructor.parse(new File(XMLDocument));
            System.out.println("Documento cargado con exito");
        } catch (ParserConfigurationException e) {
            System.err.println("No se ha podido crear una instancia de
                                DocumentBuilder");
        } catch (SAXException e) {
            System.err.println("Error SAX al parsear el archivo");
        } catch (IOException e) {
            System.err.println("Se ha producido un error de entrada salida");
        }
    }
}
```

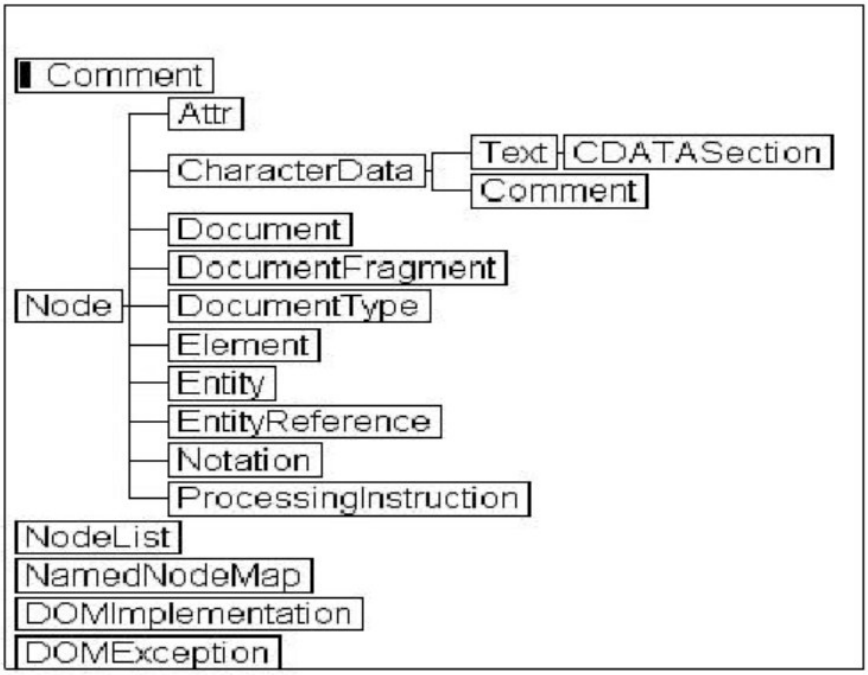
Con esto hemos terminado con lo básico del paquete javax.xml.parsers. Básicamente estos son los pasos que se han de seguir:

- Obtener una instancia de DocumentBuilderFactory.
- Indicar a la factoría el comportamiento que se desea tenga el parser.
- Obtener un DocumentBuilder que internamente usará un parser con el comportamiento deseado.
- Cargar o crear documentos a través del DocumentBuilder.

1.2 El paquete org.w3c.dom

Es el paquete fundamental de DOM. Proporciona **las interfaces para el modelo del objeto del documento (DOM)** que es un componente del Java API para el proceso de XML.

Jerarquias de interfaces:



El DOM presenta los documentos como una jerarquía de objetos **`Node`** (nodos) que a su vez implementan otras interfaces más especializadas.

Interfaces del paquete org.w3c.dom	
Attr	Representa un **atributo** de un **`elemento(Element)`**, en combinación con su valor.
CDATASection	Representa una **sección <[!CDATA[]]>.**
CharacterData	Es una **superinterfaz** para los componentes **CDATASection, Text y Comment**. Proporciona métodos para tener acceso a datos de tipo carácter en el DOM.
Comment	Esta interfaz hereda de `CharacterData` y representa el **contenido de un comentario**.
Document	Representa **el documento entero XML**.
DocumentFragment	Representa un fragmento del modelo total del árbol del documento.
DocumentType	Representa **la referencia al DTD del documento**, es decir, la línea del <!DOCTYPE>.
DOMImplementation	Proporciona métodos para realizar operaciones que son independiente de cualquier caso particular del modelo DOM, por ejemplo, permite crear un documento xml
Element	Representa un **elemento** en un documento XML.
Entity	Este interfaz representa **una entidad** en un documento de XML.
EntityReference	Representa una **referencia de entidad**.
NamedNodeMap	Se utiliza para representar **colecciones no ordenadas de nodos** que se pueden acceder por nombre.
Node	Esta es la **interfaz base** ya que todos los objetos de un árbol DOM son nodos.
NodeList	Proporciona una **colección ordenada de nodos**, que se puede acceder a través de un índice, a partir de 0
Notación	Representa una **notación declarada en el DTD**.
ProcessingInstruction	Representa **una instrucción de proceso**.
Text	Esta interfaz hereda de `CharacterData` y representa el **contenido textual** de los elementos.

Excepción	
DOMException	Las operaciones de DOM lanzan solamente excepciones en circunstancias “excepcionales”, es decir, cuando una operación es imposible realizarse (cualquiera por razones lógicas, porque se pierden los datos, o porque la puesta en práctica ha llegado a ser inestable)

Algunos tipos de nodos pueden tener **nodos hijos de varios tipos**, mientras que otros **son nodos hoja** que **no pueden tener nada bajo ellos** en la estructura del documento.

Los tipos de nodo, y los tipos de nodo que éstos pueden tener como hijos, son los siguientes:

- **Document** -- **Element** (uno como máximo), ProcessingInstruction, Comment, DocumentType
- DocumentFragment -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- DocumentType -- sin hijos
- EntityReference -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- **Element** -- Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
- **Attr** -- **Text**, **EntityReference**
- ProcessingInstruction -- sin hijos
- **Comment** -- sin hijos
- **Text** -- sin hijos
- CDATASection -- sin hijos
- **Entity** -- Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
- Notation -- sin hijos

El DOM especifica:

- Una interfaz ***NodeList*** **para manejar listas ordenadas de Nodes**, como los hijos de un Node o los elementos devueltos por el método Element.getElementsByTagName,
- Una interfaz ***NamedNodeMap*** para **manejar listas no ordenadas de nodos referenciados por su nombre**, como los atributos de un Element

1.2.1 Acceso a los nodos en el DOM

Hay dos formas de acceder a un nodo en el DOM:

- **Recorriendo el árbol**:

  Partiendo del nodo raiz, se accede a cualquier parte del árbol del documento y se utilizan los siguientes atributos de node:

  – **parentNode**: accede al padre.
  – **firstChild**: accede al primer hijo.
  – **lastChild**: accede al último hijo.
  – **nextSibling**: accede a su siguiente "hermano".
  – **previousSibling**: accede a su "hermano" anterior.
  – **childNodes**: es una lista de hijos accesible mediante un índice.

- **Por el nombre del elemento**:

  Se emplea el método **getElementsByTagName(nombre de elemento),** éste retorna una **lista de nodos** del mismo tipo y, mediante un índice, se accede al deseado.

  En la práctica, se accede al documento mediante una combinación de ambos métodos.

  Vamos a partir del siguiente fichero Actores.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Actores>
    <Actor id="51">
        <Nome>Eliabeth Shue</Nome>
        <Sexo>muller</Sexo>
        <DataNacemento formato="dd/mm/aaaa"> 06/10/1963</DataNacemento>
    </Actor>
    <Actor id="139">
        <Nome>Jhonny Depp</Nome>
        <Sexo>home</Sexo>
        <DataNacemento formato="dd/mm/aaaa">09/06/1963</DataNacemento>
    </Actor>
    <Actor id="100">
        <Nome>Harrison Ford</Nome>
        <Sexo>home</Sexo>
        <DataNacemento formato="dd/mm/aaaa">13/07/1943</DataNacemento>
    </Actor>
</Actores>
```

La siguiente introducción retorna una lista NodeList con los nodos que tengan como elemento **Actor**

```
NodeList Actores = documento.getElementsByTagName("Actor");
//Se puede recorrer la lista
for (int i = 0; i < Actores.getLength(); i++) {
    Node actor = Actores.item(i); //devuelve el nodo que está en la posicion i
```

Las siguientes instrucciones retornan una lista de NodeList de los nodos que contenga como elemento **Nome** y visualiza su texto por pantalla

```
NodeList nombres = documento.getElementsByTagName("Nome");//
 for (int i = 0; i < nombres.getLength(); i++) {
    Node nom = nombres.item(i);
    System.out.println("nombre:" + nom.getTextContent());
                                    //visualiza el texto del nodo
            }
```

```
nombre:Eliabeth Shue
nombre:Jhonny Depp
nombre:Harrison Ford
BUILD SUCCESSFUL (total time: 1 second)
```

- **Por ID (si el documento se valida con DTD o XSD)**

DOM permite acceder a un nodo **por su atributo ID**, **solo si** el documento ha sido validado y el atributo correspondiente está **declarado como tipo ID en el DTD** o **como type="ID" en el XSD**.

Se utiliza **getElementById().** Si lo encunetra devuleve el nodo y de lo contrario devolverá null.

Ejemplo con DTD

```
<!ELEMENT Actores (Actor+)>
<!ELEMENT Actor (Nome, Sexo, DataNacemento)>
<!ATTLIST Actor id ID #REQUIRED>
```

Con este DTD, el atributo id de <Actor> está declarado como tipo ID. Entonces en Java puedes acceder así:

```
Element actor = documento.getElementById("A139");
System.out.println("Actor con id A139: " +
    actor.getElementsByTagName("Nome").item(0).getTextContent());
```

Importante sobre **los valores del atributo tipo ID**:

- ✓ El valor de un atributo declarado como ID debe comenzar obligatoriamente por una letra (A–Z o a–z).
- ✓ Después puede contener letras, dígitos, guiones, puntos, guiones bajos o dos puntos ([A-Za-z][A-Za-z0-9._:-]*).
- ✓ Los valores ID deben ser únicos dentro de todo el documento XML.

El parser DOM debe estar configurado como validante (dbf.setValidating(true)) y conocer el DTD y la declaración DOCTYPE en el XML. Solo así reconocerá los atributos tipo ID y permitirá buscar por ellos.

Ejemplo con XSD

```
<xs:attribute name="id" type="xs:ID" use="required"/>
```

En este caso también se podrá usar:

```
Element actor = documento.getElementById("A51");
```

1.2.2 Manejando excepciones: DOMException

DOMException es la **única excepción definida en DOM**. Una excepción DOMException es lanzada cuando una operación es imposible de llevar a cabo.

Una excepción DOMException tiene un **número y una cadena de caracteres** que identifican el tipo de excepción. Los códigos de tipos de excepciones están definidos como variables estáticas de esta misma clase.

```
exception DOMException {
  unsigned short   code;
};

// ExceptionCode
const unsigned short      INDEX_SIZE_ERR     = 1;
const unsigned short      DOMSTRING_SIZE_ERR = 2;
const unsigned short      HIERARCHY_REQUEST_ERR = 3;
const unsigned short      WRONG_DOCUMENT_ERR = 4;
const unsigned short      INVALID_CHARACTER_ERR = 5;
const unsigned short      NO_DATA_ALLOWED_ERR = 6;
const unsigned short      NO_MODIFICATION_ALLOWED_ERR = 7;
const unsigned short      NOT_FOUND_ERR      = 8;
const unsigned short      NOT_SUPPORTED_ERR  = 9;
const unsigned short      INUSE_ATTRIBUTE_ERR = 10;
```

Constantes Definidas

INDEX_SIZE_ERR	Si el índice o el tamaño son negativos, o mayores que el valor permitido
DOMSTRING_SIZE_ERR	Si el tamaño del texto especificado no cabe en un DOMString
HIERARCHY_REQUEST_ERR	Si se inserta un nodo en algún sitio al que no pertenece
WRONG_DOCUMENT_ERR	Si se usa un nodo en un documento diferente del que lo creó (que no lo soporte)
INVALID_CHARACTER_ERR	Si se especifica un carácter inválido, como por ejemplo en un nombre.
NO_DATA_ALLOWED_ERR	Si se especifican datos para un nodo que no soporta datos
NO_MODIFICATION_ALLOWED_ERR	Si se intenta modificar un objeto que no admite modificaciones
NOT_FOUND_ERR	Si se intenta hacer referencia a un nodo en un contexto en que no existe
NOT_SUPPORTED_ERR	Si la implementación no soporta el tipo de objeto requerido
INUSE_ATTRIBUTE_ERR	Si se intenta añadir un atributo que ya está siendo usado en algún otro lugar

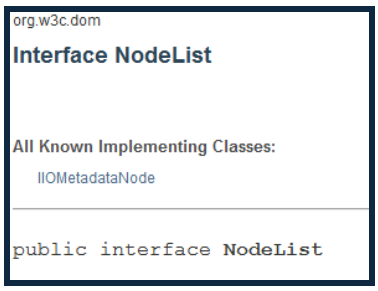
1.2.3 Tipos de lista

La API DOM proporciona dos tipos de lista -**NodeList** y **NamedNodeMap**

Los nodos que forman parte de estos dos tipos de lista DOM son dinámicos, es decir, los cambios realizados en los hijos de un elemento se reflejan automáticamente en su lista de hijos, aunque la lista haya sido obtenida antes de que se produjera el cambio.

Interfaz NodeList:

La interfaz NodeList proporciona la abstracción de un conjunto ordenado de nodos. A los elementos de la lista de nodos se puede acceder a través de un índice, a partir de 0.



Métodos

Modificador y Tipo	Método y descripción
`int`	**`getLength`** `()` El número de nodos en la lista.
`Node`	**`Item`** `(int index)` Devuelve el nodo que ocupa la posición `índice`. Si índice es mayor que o igual al número de nodos en la lista, esto devuelve `null`.

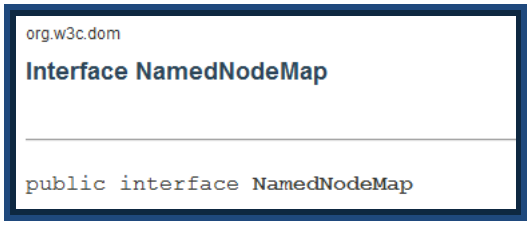
```
//Nodos Element que tenga la etiqueta Actor
NodeList Actores = documento.getElementsByTagName("Actor");
System.out.println("Actores ");
for (int i = 0; i < Actores.getLength(); i++) {
      Node actor = Actores.item(i); //Accedemos a cada nodo

}
```

Interface NamedNodeMap

La interfaz NamedNodeMap se utiliza para representar **conjuntos de nodos que se puede acceder por su nombre**.

Hay que tener en cuenta que NamedNodeMap no hereda de NodeList; Los nodos de NamedNodeMaps no se mantienen en un orden particular. Aunque también se pueden acceder por un índice ordinal, pero esto es simplemente para permitir la enumeración conveniente de los contenidos de un NamedNodeMap , y no implica que el DOM especifica un orden para estos nodos.



Metodos de la interface NamedNodeMap

Modificador y tipo	Método y descripción
`int`	**`getLength()`** El número de nodos en el mapa.
`Node`	**`getNamedItem(String name)`** Recupera un nodo especificado por nombre.
`Node`	**`getNamedItemNS`**`(String namespaceURI, String localName)` Recupera un nodo especificado por el nombre local y el URI de espacio de nombres.
`Node`	**`item`**`(int index)` Devuelve el nodo que ocupa la posición `índice`. Si índice es mayor que o igual al número de nodos en la lista, esto devuelve `null`.
`Node`	**`removeNamedItem`**`(String name)` Elimina un nodo especificado por nombre.
`Node`	**`removeNamedItemNS`**`(String namespaceURI, String localName)` Elimina un nodo especificado por el nombre local y URI de espacio de nombres.

Node	**setNamedItem**(**Node** arg) Agrega un nodo utilizando su nodeName atributo.
Node	**setNamedItemNS**(**Node** arg) Agrega un nodo utilizando su namespaceURI y localName .

Queremos visualizar los atributos del elemento Actor:

```
NodeList Actores = documento.getElementsByTagName("Actor");
System.out.println("Actores ");
for (int i = 0; i < Actores.getLength(); i++) {
   Node actor = Actores.item(i);
   NamedNodeMap atributos = actor.getAttributes();
   for (int j = 0; j < atributos.getLength(); j++) {
        System.out.println(atributos.item(j).getNodeName() + ":" +
                           atributos.item(j).getNodeValue());
       //visualizamos nombre del atributo y valor
   }
}
```

Salida

```
Actores
id:51
id:139
id:100
BUILD SUCCESSFUL (total time: 0 seconds)
```

1.2.4 El nodo: Interfaz Node

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

Representa un solo nodo en el árbol del documento

En el modelo DOM, todo elemento del XML es un nodo (Node):

- ✓ El propio documento (**Document**)
- ✓ Las etiquetas (**Element**)
- ✓ Los textos dentro de las etiquetas (**Text**)
- ✓ Los atributos (**Attr**)
- ✓ Los comentarios (**Comment**)
- ✓ Las instrucciones de proceso (**ProcessingInstruction**)
- ✓ El DTD (**DocumentType**) etc.

Por eso, **la interfaz Node es la base de todas las demás interfaces del DOM**.

- Mientras que todos los **objetos que implementan la interfaz Node proveen métodos para tratar con hijos, no todos los objetos que implementan la interfaz Node pueden tener hijos**. Por ejemplo, los nodos Text no pueden tener hijos, y al añadir hijos a tales nodos provoca una excepción **DOMException.**
  - La interfaz Node provee métodos para **manipular los nodos hijos de cualquier nodo**. 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().***
  - Para **modificar (insertar y eliminiar) los hijos de un nodo tenemos varios métodos**: ***appendChild(Node newChild)*** que inserta un nodo al final de la lista, ***removeChild(Node oldChild***) que elimina el nodo especificado, ***replaceChild(Node newChild, NoderefChild***) que sustituye un nodo por otro e ***insertBefore(Node newChild, Node refChild)*** que inserta el nodo newChild antes de refChild.
  - Cualquier intento de modificar una lista de hijos de un nodo que no admite nodos hijos provocará que se lance una excepción **DOMException.HIERARCHY_REQUEST_ERR**.

- Todos los nodos tienen un campo **ownerDocument** que referencia al documento que lo contiene, a excepción de los documentos, cuyo ownerDocument es null.

- 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.

  Los atributos nodeName, nodeValue y attributes se han incluido como un mecanismo para obtener información del nodo sin destruir la interfaz específica derivada. En los casos donde no hay una correspondencia de esos atributos para un tipo de nodo específico´-nodeType- (Por Ejemplo, nodeValue para un Element o attributes para un Comment), **estos devuelven null**.

Los valores de nodeName, nodeValue, y attributes varían de acuerdo con el tipo de nodo como sigue:

Interface	nodeName	nodeValue	attributes
Attr	Nombre del atributo	Valor del atributo	null
CDATASection	"#cdata-section"	Contenido de CDATA Section	null
Comment	"#comment"	Contenido del comentario	null
Document	"#document"	null	null
DocumentFragment	"#document-fragment"	null	null
DocumentType	Nombre del documento tipo	null	null
Element	nombre de la etiqueta	null	**NamedNodeMap**
Entity	Nombre de la entidad	null	null
EntityReference	Nombre de la entidad referenciada	null	null
Notation	Nombre de la notacion	null	null
ProcessingInstruction	El destino de esta instrucción de procesamiento. XML define a este como la primera palabra ("token") que sigue al código que inicia la instrucción de procesamiento.	El contenido de esta instrucción de procesamiento. Este va desde el primer carácter después del destino que no es espacio en blanco hasta el carácter que precede inmediatamente a ?>.	null
Text	"#text"	Contenido del nodo texto	null

El **valor de un nodo se puede modificar a través del método setNodeValue(String s);** este método lanzará una excepción DOMException.NO_MODIFICATION_ALLOWED_ERR en caso de ser un nodo de sólo lectura.

```
   NodeList Actores = documento.getElementsByTagName("Actor");
   System.out.println("Actores ");
    for (int i = 0; i < Actores.getLength(); i++) {
      Node actor = Actores.item(i);
//como es un nodo element para getNodeName() devuelve la etiqueta y para getNodeValue()
//devuelve null
        System.out.println(actor.getNodeName() + ":" + actor.getNodeValue());
        NamedNodeMap atributos = actor.getAttributes();
        for (int j = 0; j < atributos.getLength(); j++) {
//como es un nodo atributo para getNodeName() devuelve el nombre del atributo y para getNodeValue()
//devuelve el valor del atributo

            System.out.println(atributos.item(j).getNodeName() + ":" +
                               atributos.item(j).getNodeValue());
        }
```

Salida:

```
Actores
Actor:null
id:51
Actor:null
id:139
Actor:null
```

Ejemplo 2:

```
System.out.println("nombre:"+documento.getNodeName()+ "
                    valor:"+documento.getNodeValue()+" Primer hijo:"+
                     documento.getFirstChild().getNodeName())
```

Salida

```
nombre:#document   valor:null Primer Hijo:Actores
```

- 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**.

  Cada nodo DOM tiene un tipo numérico (de tipo short) que indica su naturaleza. Estas constantes te permiten saber de qué tipo es un nodo usando:

Campos de la interfaz org.w3c.dom.Node	
static short	**ATTRIBUTE_NODE** El nodo es un Attr.
static short	**CDATA_SECTION_NODE** El nodo es de tipo CDATASection.
static short	**COMMENT_NODE** El nodo es de tipo Comment.
static short	**DOCUMENT_FRAGMENT_NODE** El nodo es un DocumentFragment.
static short	**DOCUMENT_NODE** El nodo es un Document.
static short	**DOCUMENT_POSITION_CONTAINED_BY** El nodo está contenido por el nodo referenciado.
static short	**DOCUMENT_POSITION_CONTAINS** El nodo contiene la referencia del nodo especificado.
static short	**DOCUMENT_POSITION_DISCONNECTED** Los dos nodos están desconectados.
static short	**DOCUMENT_POSITION_FOLLOWING** El nodo sigue al nodo referenciado.
static short	**DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC** La determinación de "anterior" y "siguiente" depende de la implementación.
static short	**DOCUMENT_POSITION_PRECEDING** el segundo nodo precede al nodo referenciado.
static short	**DOCUMENT_TYPE_NODE** El nodo es de tipo DocumentType.
static short	**ELEMENT_NODE** el nodo es un Element.
static short	**ENTITY_NODE** El nodo es una Entity.
static short	**ENTITY_REFERENCE_NODE** El nodo es de tipo EntityReference.
static short	**NOTATION_NODE** El nodo es de tipo Notation.
static short	**PROCESSING_INSTRUCTION_NODE** El nodo es de tipo ProcessingInstruction.
static short	**TEXT_NODE** El nodo es de tipo Text.
Metodos de la interfaz org.w3c.dom.Node	
Node	**appendChild**(**Node** newChild) Añade el nodo newChild al final de la lista de hijos de este nodo.
Node	**cloneNode**(boolean deep) Devuelve un duplicado de este nodo
short	**compareDocumentPosition**(**Node** other) Compara el nodo referenciado.

NamedNodeMap	**getAttributes**() Un NamedNodeMap que contiene los atributos de este nodo (si es un Element) o null en cualquier otro caso.
java.lang.String	**getBaseURI**() La URI absoluta de este nodo o null si a la implementación no le fue posible obtenerla.
NodeList	**getChildNodes**() Un NodeList que contiene todos los hijos de este nodo.
java.lang.Object	**getFeature**(java.lang.String feature, java.lang.String version) Devuelve un objeto especializado que implementa una API especializada que maneja una característica específica de una versión específica..
Node	**getFirstChild**() El primer hijo de este nodo.
Node	**getLastChild**() El último hijo de este nodo.
java.lang.String	**getLocalName**() Devuelve la parte local del nombre "qualified" de este nodo.
java.lang.String	**getNamespaceURI**() El namespace URI de este nodo, o null si no se ha especificado.
Node	**getNextSibling**() El nodo siguiente primero a este nodo.
java.lang.String	**getNodeName**() El nombre de este nodo, dependiendo de su tipo.
short	**getNodeType**() Un código representando el tipo del objeto que subyace, como se define anteriormente.
java.lang.String	**getNodeValue**() El valor de este nodo, dependiendo de su tipo.
Document	**getOwnerDocument**() El objeto Document asociado con este nodo.
Node	**getParentNode**() El padre de este nodo.
java.lang.String	**getPrefix**() El prefijo del namespace de este nodo o null si no lo tiene especificado.
Node	**getPreviousSibling**() El nodo inmediatamente anterior a este.
java.lang.String	**getTextContent**() Este atributo devuelve el contenido texto de este nodo y sus descendientes.
java.lang.Object	**getUserData**(java.lang.String key) Proporciona el objeto asociado a una clave en este nodo.
boolean	**hasAttributes**() Devuelve si este nodo (si es un elemento) tiene atributos.
boolean	**hasChildNodes**() Devuelve si el nodo tiene algún hijo.
Node	**insertBefore**(Node newChild, Node refChild) Inserta el nodo newChild antes del hijo existente, refChild.
boolean	**isDefaultNamespace**(java.lang.String namespaceURI) Comprueba si el namespaceURI es el namespace por defecto.
boolean	**isEqualNode**(Node arg) Comprueba si dos nodos son iguales.
boolean	**isSameNode**(Node other) Devuelve si los nodos son iguales.
boolean	**isSupported**(java.lang.String feature, java.lang.String version) Testea tanto si la implementación incluye una característica específica como si dicha característica está soportada en este nodo.
java.lang.String	**lookupNamespaceURI**(java.lang.String prefix) Busca el namespace URI asociado al prefijo especificado comenzando en este nodo.
java.lang.String	**lookupPrefix**(java.lang.String namespaceURI) Busca el prefijo asociados al namespace URI, comenzando desde este nodo.
void	**normalize**() El método normalize() recorre todo el árbol de ese nodo (y sus descendientes) y: Fusiona todos los nodos Text adyacentes en uno solo. Elimina los nodos Text vacíos (sin contenido visible).
Node	**removeChild**(Node oldChild) Elimina el nodo hijo indicado por oldChild de la lista de nodos hijos y lo devuelve.
Node	**replaceChild**(Node newChild, Node oldChild) Reemplaza el nodo hijo oldChild con newChild in the list of children, and returns the oldChild node.

void	**setNodeValue**(java.lang.String nodeValue) El valor de este nodo, dependiendo de su tipo. (ver la tabla al principio de esta sección)
void	**setPrefix**(java.lang.String prefix) El prefijo del namespace de este nodo o null si no está especificado.
void	**setTextContent**(java.lang.String textContent) Este atributo devuelve el contenido texto del nodo y sus descendientes.
java.lang.Object	**setUserData**(java.lang.String key,java.lang.Object data, **UserDataHandler** handler) Asocia un objeto a una clave de este nodo.

1.2.5 El documento: interfaz Document

```
public interface Document
extends Node
```

La interfaz de **Document** representa al **documento completo XML**. Conceptualmente, es la **raíz** del árbol de documentos, y **proporciona el acceso primario a los datos del documento**.

Un documento **sólo puede tener** un elemento hijo que es el **elemento raíz** del documento.

Dado que los **elementos, nodos de texto, comentarios, instrucciones de procesamiento**, etc, no pueden existir fuera del contexto de un documento, la interfaz de documento también **contiene los métodos necesarios para crear estos objetos** (posee varios métodos del tipo createXXX() que devuelven nodos de tipos específicos).

Los objetos creados nodo tiene un atributo **ownerDocument** que los asocia con el documento en cuyo contexto se han creado. Los nuevos nodos sólo se pueden añadir al documento que los creó, si se intenta añadir un nodo creado por un documento a otro documento se lanza una excepción del tipo DOMException.WRONG_DOCUMENT_ERR.

Un documento también puede tener asociado o no un objeto DocumentType y varios hijos ProcessingInstruction.

De esta forma se **obtiene el elemento raíz del documento**:

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

El objeto **DocumentType** nos permite obtener información sobre el tipo de documento (DTD). A través de este objeto podemos acceder a las Entities y Notations.

De la siguiente forma se obtiene el DocumentType de un documento:

```
DocumentType docType = documento.getDoctype();
```

El objeto Document permite obtener listados de elementos por tagName y/o por espacio de nombres y también podemos buscar un elemento por su identificador.

Los **listados de elementos** se devuelven en forma de **NodeList**.

NodeList es una interfaz que sólo define dos métodos:

- ✓ **getLength() que devuelve la longitud de la lista**
- ✓ **e item(int index) que devuelve un Node de la lista por su índice.**

Method Summary	
Node	**adoptNode**(**Node** source) Adopta un nodo de otro documento como perteneciente a este documento.
Attr	**createAttribute**(java.lang.String name) Crea un Attr con el nombre que le damos como parámetro.
Attr	**createAttributeNS**(java.lang.String namespaceURI, java.lang.String qualifiedName) Crea un atributo con el "qualified" nombre y el namespace que se le indica como parámetro.
CDATASection	**createCDATASection**(java.lang.String data)

	Crea un nodo del tipo `CDATASection` con el valor especificado en la cadena que se le pasa como parámetro.
Comment	**createComment**(java.lang.String data) Crea un nodo de tipo `Comment` con el valor especificado en la cadena que se le pasa.
DocumentFragment	**createDocumentFragment**() Crea un objeto `DocumentFragment` vacio.
Element	**createElement**(java.lang.String tagName) Crea un elemento del tipo especificado.
Element	**createElementNS**(java.lang.String namespaceURI, java.lang.String qualifiedName) Crea un elemento con el "qualifed" nombre y el "namespace" que se le pasa.
EntityReference	**createEntityReference**(java.lang.String name) Crea un objeto referencia a una entidad, `EntityReference`.
ProcessingInstruction	**createProcessingInstruction**(java.lang.String target, java.lang.String data) Crea un nodo de tipo `ProcessingInstruction` con el nombre y el contenido especificado.
Text	**createTextNode**(java.lang.String data) Crea un nodo `Text` con el texto que se le pasa.
DocumentType	**getDoctype**() La declaración de "Document Type" (DTD) asociada con este documento.
Element	**getDocumentElement**() Este es un método que permite acceder al elemento raíz. A partir de aquí se puede acceder a los nodos hijos .
java.lang.String	**getDocumentURI**() La localización del documento o `null` si no está definida o si el documento fue creado utilizando `DOMImplementation.createDocument`.
DOMConfiguration	**getDomConfig**() La configuración utilizada cuando se llama al método `Document.normalizeDocument()`.
Element	**getElementById**(java.lang.String elementId) Devuelve el `Element` cuyo ID conincide con el valor que se le pasa como parámetro.
NodeList	**getElementsByTagName**(java.lang.String tagname) Devuelve un `NodeList` de todos los `Elements` del documento cuyo nombre coincide con el que se le pasa.
NodeList	**getElementsByTagNameNS**(java.lang.String namespaceURI, java.lang.String localName) Devuelve un `NodeList` de todos los `Elements` con el "local name" y el "namespace" especificados en el orden en el que aparecen en el documento.
DOMImplementation	**getImplementation**() El objeto `DOMImplementation` que maneja este documento.
java.lang.String	**getInputEncoding**() Un Atributo especificando el tipo de codificación utilizada por este documento en el momento de utilizar el parser.
boolean	**getStrictErrorChecking**() Devuelve un atributo especificando si el chequeo de errores es obligatorio o no.
java.lang.String	**getXmlEncoding**() Un atributro especificando, como parte de la declaración de XML, la codificación de este documento.
boolean	**getXmlStandalone**() Un Atributo especificando, como parte de la declaración de XML, si este documento es autónomo, no depende de maś ficheros.
java.lang.String	**getXmlVersion**() Un atributo especificando, como parte de la declaración XML, la versión de este documento.
Node	**importNode**(**Node** importedNode, boolean deep) Importa un nodo de otro documento a este win alterar o borrar el nodo origen del documento original; Este método crea una copia del nodo origen.
Node	**renameNode**(**Node** n, java.lang.String namespaceURI, java.lang.String qualifiedName) Renombra un nodo existente del tipo `ELEMENT_NODE` o `ATTRIBUTE_NODE`.
void	**setDocumentURI**(java.lang.String documentURI) La localización del documento o `null` si no está definida o si el documento fue creado utilizando `DOMImplementation.createDocument`.
void	**setStrictErrorChecking**(boolean strictErrorChecking) Un Atributo especificando si el chequeo de errores es obligatorio o no.
void	**setXmlStandalone**(boolean xmlStandalone) Permite establece si este documento es autónomo, es decir, no hay más ficheros XML que formen parte de este documento.
void	**setXmlVersion**(java.lang.String xmlVersion) Permite establecer la versión de este documento.

1.2.6 Elemento: El interfaz Element

El interfaz Element va a servir para manejar cualquier elemento e incluye, su "tag" de comienzo, el "tag" de salida y el contenido.

Este contenido puede ser otro elemento, un nodo de instrucciones de procesamiento, nodos comentario, nodos de texto, secciones CDATA o entidades. Su definición es la siguiente:

campos heredados del interfaz org.w3c.dom.Node
ATTRIBUTE_NODE, **CDATA_SECTION_NODE**, **COMMENT_NODE**, **DOCUMENT_FRAGMENT_NODE**, **DOCUMENT_NODE**, **DOCUMENT_POSITION_CONTAINED_BY**, **DOCUMENT_POSITION_CONTAINS**, **DOCUMENT_POSITION_DISCONNECTED**,

DOCUMENT_POSITION_FOLLOWING, DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC, DOCUMENT_POSITION_PRECEDING, DOCUMENT_TYPE_NODE, ELEMENT_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, NOTATION_NODE, PROCESSING_INSTRUCTION_NODE, TEXT_NODE

Method Summary	
java.lang.String	**getAttribute**(java.lang.String name) Recupera el valor de un atributo identificandolo con su nombre.
Attr	**getAttributeNode**(java.lang.String name) Devuelve un nodo atributo utilizando su nombre.
Attr	**getAttributeNodeNS**(java.lang.String namespaceURI, java.lang.String localName) Devuelve un nodo de tipo Attr identificándolo por su nombre local y su namespace URI.
java.lang.String	**getAttributeNS**(java.lang.String namespaceURI, java.lang.String localName) Devuelve el valor de un atributo identificándolo por su nombre local y su namespace URI.
NodeList	**getElementsByTagName**(java.lang.String name) Devuelve un NodeList de todos los Elementos descendientes con el "tag name", en el mismo orden en el que aparecen en el documento.
NodeList	**getElementsByTagNameNS**(java.lang.String namespaceURI, java.lang.String localName) Devuelve un NodeList de todos los Elements descendientes con el nombre "local" y el "namespace" URI en el orden en el que aparecen en el documento.
TypeInfo	**getSchemaTypeInfo**() El tipo de información asociada con este elemento.
java.lang.String	**getTagName**() El nombre del elemento.
boolean	**hasAttribute**(java.lang.String name) Devuelve true cuándo un atributo con el nombre que se especifica está presente en este elemento o tiene un valor por defecto, y devuelve false en cualquier otro caso.
boolean	**hasAttributeNS**(java.lang.String namespaceURI, java.lang.String localName) Devuelve true cuando un atributo con el nombre local y el namespace especificado existe en este elemento o tienen un valor por defecto, y false en cualquier otro caso.
void	**removeAttribute**(java.lang.String name) Elimina un atributo identificado por el nombre.
Attr	**removeAttributeNode**(**Attr** oldAttr) Elimina el atributo especificado.
void	**removeAttributeNS**(java.lang.String namespaceURI, java.lang.String localName) Elimina un atributo identificado por su nombre local y su namespace URI.
void	**setAttribute**(java.lang.String name, java.lang.String value) Añade un nuevo atributo.
Attr	**setAttributeNode**(**Attr** newAttr) Añade un nuevo atributo
Attr	**setAttributeNodeNS**(**Attr** newAttr) Añade un nuevo atributo.
void	**setAttributeNS**(java.lang.String namespaceURI, java.lang.String qualifiedName, java.lang.String value) Añade un nuevo atributo.
void	**setIdAttribute**(java.lang.String name, boolean isId) Si es parámetro isId es true, este método declara el atributo especificado para ser utilizado por el usuario como atributo ID.
void	**setIdAttributeNode**(**Attr** idAttr, boolean isId) Si el parámetro isId es true, este método declara el atributo especificado para ser utilizado por el usuario como atributo ID .
void	**setIdAttributeNS**(java.lang.String namespaceURI, java.lang.String localName, boolean isId) Si el parámetro isId es true, este método declara el atributo especificado para ser utilizado por el usuario como atributo ID. .

Methods inherited from interface org.w3c.dom.Node
appendChild, cloneNode, compareDocumentPosition, getAttributes, getBaseURI, getChildNodes, getFeature, getFirstChild, getLastChild, getLocalName, getNamespaceURI, getNextSibling, getNodeName, getNodeType, getNodeValue, getOwnerDocument, getParentNode, getPrefix, getPreviousSibling, getTextContent, getUserData, hasAttributes, hasChildNodes, insertBefore, isDefaultNamespace, isEqualNode, isSameNode, isSupported, lookupNamespaceURI, lookupPrefix, normalize, removeChild, replaceChild, setNodeValue, setPrefix, setTextContent, setUserData

1.2.7 Atributos: Interfaz Attr

La interfaz **Attr** representa un atributo en un elemento de objeto.

Son **un tipo de nodo muy característico**. **Sólo existen dentro de elementos y no forman parte del árbol**, en realidad, no son los nodos secundarios del elemento que describen, ya que DOM no los considera parte de la estructura del documento.

```
public interface Attr
extends Node
```

Attr heredan de la interfaz Node.

- 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.
- 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.

Los atributos se crean de forma casi idéntica a los elementos:

```
Attr unatributo = documento.createAttribute(String name);
```

Campos heredados del interfaz org.w3c.dom.Node
ATTRIBUTE_NODE, CDATA_SECTION_NODE, COMMENT_NODE, DOCUMENT_FRAGMENT_NODE, DOCUMENT_NODE, DOCUMENT_POSITION_CONTAINED_BY, DOCUMENT_POSITION_CONTAINS, DOCUMENT_POSITION_DISCONNECTED, DOCUMENT_POSITION_FOLLOWING, DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC, DOCUMENT_POSITION_PRECEDING, DOCUMENT_TYPE_NODE, ELEMENT_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, NOTATION_NODE, PROCESSING_INSTRUCTION_NODE, TEXT_NODE

Métodos de Atrrib

Modificador y Tipo	Método y descripción
String	getName () Devuelve el nombre de este atributo.
Elemento	getOwnerElement () Devuelve el elemento que pertenece este atributo o null si este atributo no está en uso.
TypeInfo	getSchemaTypeInfo () La información del tipo asociado a este atributo.
booleano	getSpecified () Verdadero si este atributo se le dio un valor explícitamente en el documento o falso de lo contrario.
String	getValue () Devuelve el valor del atributo como una cadena.
booleano	isId () Devuelve true si este atributo es de tipo ID.
void	setValue (String valor) Añade un valor para el atributo

Metodos heredados de la interface org.w3c.dom.Node
appendChild, cloneNode, compareDocumentPosition, getAttributes, getBaseURI, getChildNodes, getFeature, getFirstChild, getLastChild, getLocalName, getNamespaceURI, getNextSibling, getNodeName, getNodeType, getNodeValue, getOwnerDocument, getParentNode, getPrefix, getPreviousSibling, getTextContent, getUserData, hasAttributes, hasChildNodes, insertBefore, isDefaultNamespace, isEqualNode, isSameNode, isSupported, lookupNamespaceURI, lookupPrefix, normalize, removeChild, replaceChild, setNodeValue, setPrefix, setTextContent, setUserData

1.2.8 Nodos de texto: interfaz CharacterData

La interfaz **CharacterData** es superinterfaz de las interfaces: Text y Comment.

org.w3c.dom

Interface CharacterData

All Superinterfaces:

Node

All Known Subinterfaces:

CDATASection, Comment, Text, Text

```
public interface CharacterData
extends Node
```

Campos heredados del interfaz org.w3c.dom.Node

ATTRIBUTE_NODE, CDATA_SECTION_NODE, COMMENT_NODE, DOCUMENT_FRAGMENT_NODE, DOCUMENT_NODE, DOCUMENT_POSITION_CONTAINED_BY, DOCUMENT_POSITION_CONTAINS, DOCUMENT_POSITION_DISCONNECTED, DOCUMENT_POSITION_FOLLOWING, DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC, DOCUMENT_POSITION_PRECEDING, DOCUMENT_TYPE_NODE, ELEMENT_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, NOTATION_NODE, PROCESSING_INSTRUCTION_NODE, TEXT_NODE

Métodos

Modificador y Tipo	Método y descripción
void	**appendData** (**String** arg) Añade la cadena al final de l texto en el nodo.
void	**deleteData** (int desplazamiento, int count) Eliminar count caracteres del texto del nodo empezando en desplazamiento
String	**getData** () Devuelve el texto del nodo.
int	**getLength** () Devuelve el número de caracteres (16 bit) del texto .
void	**insertData** (int desplazamiento, **String** arg) Inserta una cadena de 16 bits a partir de desplazamiento.
void	**replaceData** (int desplazamiento, int count, **String** arg) Replaza el texto por arg a partir del desplazamiento
void	**setData** (String datos) Cambia el texto
String	**substringData** (int desplazamiento, int count) Extrae una count de caracteres (de 16 bits) del texo del nodo a partir de desplazamiento

Metodos heredados de la interface org.w3c.dom.Node

appendChild, cloneNode, compareDocumentPosition, getAttributes, getBaseURI, getChildNodes, getFeature, getFirstChild, getLastChild, getLocalName, getNamespaceURI, getNextSibling, getNodeName, getNodeType, getNodeValue, getOwnerDocument, getParentNode, getPrefix, getPreviousSibling, getTextContent, getUserData, hasAttributes, hasChildNodes, insertBefore, isDefaultNamespace, isEqualNode, isSameNode, isSupported, lookupNamespaceURI, lookupPrefix, normalize, removeChild, replaceChild, setNodeValue, setPrefix, setTextContent, setUserData

Todas estas interfaces representan **nodos que contienen texto**. Debido a ello tienen métodos comunes que son definidos en la interfaz CharacterData. Estos métodos **permiten obtener y manipular el texto**

- **Text**: representa el **texto contenido entre elementos**. La interfaz Text es a su vez superinterfaz de CDATA Section.

org.w3c.dom

Interface Text

All Superinterfaces:

CharacterData, Node

All Known Subinterfaces:

CDATASection, Text

Métodos	
Modificador y Tipo	**Método y descripción**
`String`	**`getWholeText`** `()` Devuelve todo el texto de `texto` nodos nodos de texto lógicamente adyacentes a este nodo, concatenado en el orden del documento.
`booleano`	**`isElementContentWhitespace`** `()` Devuelve si este nodo de texto contiene **elementos de contenido los espacios en blanco** , a menudo abusivamente llamado "ignorable espacios en blanco".
`String`	**`replaceWholeText`** `( String contenido)` Sustituye el texto del nodo actual y todos los nodos de texto lógicamente adyacentes con el texto especificado.
`String`	**`splitText`** `(int offset)` Este método recorta el texto del objeto Text hasta la posicion offset y devuelve otro objeto con el resto del texto. Esto está pensado para insertar nodos entre el texto.

- **Comment**: representa **los comentarios de un documento**. No añade ningún método adicional.

org.w3c.dom

Interface Comment

All Superinterfaces:

CharacterData, Node

```
public interface Comment
extends CharacterData
```

1.2.9 Interfaz CDATASection

Las secciones CDATA se usan para escapar bloques de texto que contienen caracteres que de otro modo serían considerados como código.

El delimitador único que se reconoce en una sección CDATA es la "]]>", cadena que termina la sección CDATA. Las secciones CDATA no se pueden anidar.

org.w3c.dom

Interface CDATASection

All Superinterfaces:

CharacterData, Node, Text

```
public interface CDATASection
extends Text
```

1.3 Validar un documento XML utilizando DOM

La API JAXP permite validar documentos XML utiliando DTD o esquema (s).

1.3.1 Validar un documento XML usando un DTD

Para la validación de un XML utilizando un archivo DTD se necesita primero un archivo xml bien formado y su documento DTD asociado.

- En el archivo DTD se define todos los elementos a tener en el archivo xml. Por ejemplo:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--- El elmento actores contiene cero o más actores. -->
<!ELEMENT Actores (Actor)*>
<!--- El elemento Actor debe tener nome, sexo y fecha de nacimiento. -->
<!ELEMENT Actor (Nome,Sexo,DataNacemento)>
<!---  Actor debe tener obligatorio el atributo id -->
<!ATTLIST Actor
    id CDATA #REQUIRED
 >
<!--- (#PCDATA) - contenido textual, un conjunto de caracteres -->
<!ELEMENT Nome (#PCDATA)>
<!ELEMENT Sexo (#PCDATA)>
<!ELEMENT DataNacemento (#PCDATA)>
<!--- #IMPLIED: permite no especificar el atributo (es opcional).-->
<!ATTLIST DataNacemento
    formato CDATA #IMPLIED
```

- En el **documento XML se indica el DTD asociado** que va a validar el documento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Actores SYSTEM "Actores1.dtd">
<Actores>
    <Actor id="59">
```

- Se activa que el analizador pueda validar documentos. Se utiliza el método **setValidating (true)** del objeto **DocumentBuilderFactory**.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
System.out.println("valida:"+dbf.isValidating());
dbf.setValidating(true);
System.out.println("cambio valida:"+dbf.isValidating());
```

Salida:

```
run:
valida:false
cambio valida:true
```

- Informar de los posibles errores al validar los documentos. Para **informar de errores**, es necesario proporcionar un **ErrorHandler** a la aplicación subyacente.

Interface ErrorHandler

ErrorHandler es la **interface básica para manejar errores SAX**, pero también es utilizada por DOM para el control de errores cuando se lleva a cabo el análisis de un documento XML.

Para **los errores de procesamiento de XML**, se debe utilizar la interfaz **ErrorHandler, en lugar de lanzar una excepción.** A continuación se debe registrar una instancia con el parser XML mediante el método **setErrorHandler**(). El analizador dará cuenta de los posibles errores y advertencias a través de esta interfaz.

La interface ErrorHandler define los sigueintes **tres métodos** para el tratamiento de los distintos tipos de errores:

Métodos

Modificador y tipo	Método y descripción
void	**error(SAXParseException exception) throws SAXException** Recibe notificación de un **error recuperable**. El analizador puede continuar con el análisis después de invocar este método y puede que sea posible que se procese el documento hasta el final. Si la aplicación no puede hacerlo, entonces el analizador debería informar de un error fatal, aunque la recomendación XML no le obliga a hacerlo.

void	**fatalError(SAXParseException** exception) throws SAXException Recibe notificación de un error **no recuperable**. El analizador no puede continuar con el análisis después de invocar este método y el documento XML no se puede usar.
void	**warning(SAXParseException** exception) throws SAXException Recibir notificación de un aviso. El analizador utiliza este método para informar de las condiciones que no son errores o errores fatales. El analizador debe continuar con el análisis después de invocar este método procesando el documento hasta el final.

- La mayoría de los avisos y los errores normales tienen que ver con la validación de los documentos.
- Los errores fatales suelen ser relativos a la mala formación del documento o con la versión del XML.

Ejemplo:

```
public class SimpleErrorHandler implements ErrorHandler {
    public void warning(SAXParseException e) throws SAXException {
        System.out.println(e.getMessage());
    }
    public void error(SAXParseException e) throws SAXException {
        System.out.println(e.getMessage());
   //se puede lanzar la excepción e para que no siga con el procesam del doc XML
        throw e;
    }

    public void fatalError(SAXParseException e) throws SAXException {
        System.out.println(e.getMessage());
        throw e;
    }
}
```

Y en nuestro programa:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(true);
DocumentBuilder analizador = factory.newDocumentBuilder();
analizador.setErrorHandler(new SimpleErrorHandler());
Document documento = analizador.parse(new File ("document.xml"));
```

También se puede tratar los errores de validación en el propio documento:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(true);
DocumentBuilder analizador = dbf.newDocumentBuilder();
analizador.setErrorHandler(
 new ErrorHandler() {
 @Override
 public void warning(SAXParseException e) throws SAXException {
       System.out.println("WARNING : " + e.getMessage());
 }
  @Override
  public void error(SAXParseException e) throws SAXException {
  System.out.println("ERROR : " + e.getMessage());
   //se puede lanzar la excepción e para que no siga con el procesam del doc XML
   throw e;
   }
    @Override
    public void fatalError(SAXParseException e) throws SAXException {
    System.out.println("FATAL : " + e.getMessage());
     throw e;
     }
   }
 );
Document documento = analizador.parse(new File ("document.xml"));
```

Ejemplo de errores de validación utlizando el DTD creado anteriormente:

Ej: Si borramos un elemento de actor por ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Actores SYSTEM "Actores1.dtd">
<Actores>
    <Actor id="59">
        <Nome>Eliabeth Shue</Nome>
        <DataNacemento formato="dd/mm/aaaa"> 06/10/1963</DataNacemento>
    </Actor>
```

```
ERROR : El contenido del tipo de elemento "Actor" debe coincidir con "(Nome
,Sexo,DataNacemento)".
Error SAX al parsear el archivo
BUILD SUCCESSFUL (total time: 0 seconds)
```

1.3.2 La validación de un documento XML utilizando Schemas.

Dado el siguiente archivo XSD;

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Actores">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Actor" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Actor">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Nome"/>
        <xs:element ref="Sexo"/>
        <xs:element ref="DataNacemento"/>
      </xs:sequence>
      <xs:attribute name="id"  type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:element name="Nome" type="xs:string"/>
  <xs:element name="Sexo" type="xs:string"/>
  <xs:element name="DataNacemento">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute name="formato" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

La **validación de un documento XML utilizando Schemas** requiere dos pasos, al igual que cuando son utilizados DTD's:

Forma Clásica: con la declaración en el Documento XML

- Debe ser definido en el elemento raíz (el primero del documento XML), el Schema que será empleado a través de un Namespace/atributo :

```
<?xml version="1.0" encoding="UTF-8"?>
<Actores xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
                  xsi:noNamespaceSchemaLocation="Actores.xsd">
```

La primer declaración indica el Namespace utilizado para documentos que serán validados a través de Schema;

El parámetro xsi:noNamespaceSchemaLocation="Actores.xsd", indica que el Schema utilizado para validar el documento es el archivo llamado Actores.xsd el cual se encuentra en el mismo directorio del documento en cuestión.

Modificación del Parser para utilizar Schemas

Las modificaciones al parser son llevadas acabo a través de parámetros al momento de ser inicializado.

- Es necesario **definir dos constantes** utilizadas para validar XML con "Schemas":

```
//Especifica el lenguaje utilizado por el parser en el análisis
static final String JAXP_SCHEMA_LANGUAGE =
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
//especifica el espacio de nombres
static final String W3C_XML_SCHEMA =
    "http://www.w3.org/2001/XMLSchema";
```

- Para activar la validación completa es necesario definir los siguientes parámetros:

```
String XMLDocument = "Actores1.xml";
//Implementamos el analizador XML que trae el API JAXP de Java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(true);
dbf.setNamespaceAware(true);
```

- Después, hay que configurar el objeto DocumentBuilderFactory para generar un namespace, validando el programa de análisis que utiliza el esquema de XML:

```
dbf.setAttribute( JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

- Informar de los posibles errores al validar los documentos. Para **informar de errores**, es necesario proporcionar un **ErrorHandler** a la aplicación subyacente igual que se hizo con la validación de DTD.

```
DocumentBuilder analizador = dbf.newDocumentBuilder();
analizador.setErrorHandler(
          new ErrorHandler() {
              @Override
            public void warning(SAXParseException e) throws SAXException {
              System.out.println("WARNING : " + e.getMessage()); //
            }
              @Override
            public void error(SAXParseException e) throws SAXException {
              System.out.println("ERROR : " + e.getMessage());
              //si no se quiere que se siga procesando se lanza esta excepción
              throw e;
            }
              @Override
            public void fatalError(SAXParseException e) throws SAXException {
              System.out.println("FATAL : " + e.getMessage());
              throw e;
            }
          }
```

Ej: Ejemplo de errores de validación utlizando del schema creado anteriormente:

Si borramos el atributo id de actor por ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<Actores xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="Actores1.xsd">
    <Actor>
        <Nome>Eliabeth Shue</Nome>
        <Sexo>muller</Sexo>
        <DataNacemento formato="dd/mm/aaaa"> 06/10/1963</DataNacemento>
    </Actor>
```

```
run:
Error SAX al parsear el archivo
ERROR : cvc-complex-type.4: El atributo 'id' debe aparecer en el elemento 'Actor'.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Forma moderna con SchemaFactory y setSchema()

A partir de Java 5, se introdujo la clase SchemaFactory que permite cargar y asociar un esquema XSD de forma más segura y estándar.

- ✓ **SchemaFactory** crea una instancia de la validación estándar W3C.
- ✓ **schemaFactory.newSchema(new File(XSDDocument))** carga el XSD.
- ✓ **dbf.setSchema(schema)** asocia directamente el esquema al parser.
- ✓ **Ya no hace falta definir atributos ni namespaces** manualmente en el XML
- ✓ Si hay errores, el ErrorHandler los informa de forma controlada.

```
   String XMLDocument = "Actores.xml";
           String XSDDocument = "Actores.xsd";

     try {
          // Crear la fábrica de validación
       SchemaFactory schemaFactory
     = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
                Schema schema = schemaFactory.newSchema(new File(XSDDocument));

                // Crear el parser DOM con el esquema
                DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
                dbf.setNamespaceAware(true);
                dbf.setSchema(schema);

                DocumentBuilder analizador = dbf.newDocumentBuilder();
                analizador.setErrorHandler(new SimpleErrorHandler());

                Document documento = analizador.parse(new File(XMLDocument));
                System.out.println("Documento    validado    correctamente    con    el
esquema.");

            } catch (SAXException e) {
                System.out.println("Error de validación: " + e.getMessage());
            } catch (Exception e) {
                System.out.println("Error general: " + e.getMessage());
            }}
```

1.4 Ignorar los nodos de texto de Salto de línea y tabulaciones al generar el árbol DOM

- Los saltos de línea y las tabulaciones al crear el árbol DOM se convierten en nodos de Texto. Se puede hacer que al cargar el documento XML no se generen estos nodos de Texto.
- Primero, tenemos que hacer que cuando se implemente el parse **valide el contenido del documento XML utilizando un DTD o un esquema XDS** (Tal como vimos anteriormente).
- A continuación se estable la propiedad IgnoringElementContentWhitespace a true utilizando el método **setIgnoringElementContentWhitespace(true)** de la interfaz DocumentBuilderFactory.

Ejemplo:

```
String XMLDocument = "Actores1.xml";
        Document documento = null;
        //Implementamos el analizador XML que trae el API JAXP de Java
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setValidating(true);
            dbf.setNamespaceAware(true);
            dbf.setAttribute("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
              "http://www.w3.org/2001/XMLSchema");
            dbf.setIgnoringElementContentWhitespace(true);
```

En actores.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<Actores xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:noNamespaceSchemaLocation="Actores1.xsd">
```

1.5 Añadir nodos a un árbol DOM

El API DOM nos ofrece una serie de métodos para añadir nodos al árbol de un documento. Los más básicos son:

- **createElement (String name):** que crea un nuevo nodo elemento.
  createElementNS(String namespaceURI, java.lang.String Name): Crea un elemento con el "qualifed" nombre y el "namespace" que se le pasa.
- **createAttribute(String name):** Crea un atributo con el nombre que le damos como parámetro.
  createAttributeNS(String namespaceURI, String Name) : Crea un atributo con el "qualified" nombre y el namespace que se le indica como parámetro.
- **createTextNode**(String texto): Crea un nodo Text con el texto que se le pasa.

También se pueden crear otros tipos de nodos con:

- createCDATASection(String texto): Crea un nodo del tipo CDATASection con el valor especificado en la cadena que se le pasa como parámetro.
- createComment(String texto) : Crea un nodo de tipo Comment con el valor especificado en la cadena que se le pasa.
- createDocumentFragment():Crea un objeto DocumentFragment vacio.
- createEntityReference(String name):Crea un objeto referencia a una entidad.
- createProcessingInstruction(String target,String data): Crea un nodo de tipo ProcessingInstruction con el nombre y el contenido especificado.

El proceso para crear un nuevo nodo es muy sencillo. Se llama al método apropiado para **crear el nuevo nodo en el nodo document**.

Una vez, creado el nodo:

- **Se puede añadir al final de los demás nodos:**
  Con el método **appendChild** se añade el nuevo nodo al final de los demás nodos hijos del nodo deseado.
  Ejemplo: Vamos a añadir un nuevo actor al nodo raíz Actores.

```
Element mielem = (Element) documento.createElement("Actor");
        //se lo añadimos al docuemento raiz
documento.getDocumentElement().appendChild(mielem);
```

Para añadir un atributo al elemento Actor:

```
Attr atrib = documento.createAttribute("id"); //creamos el atributo id
 atrib.setValue("2"); //se le da el valor 2
 mielem.setAttributeNode(atrib); //y el atributo se lo ponemos a nodo Actor que hemos creado
 //o también
mieelem.appendChild(atrib)
```

Vamos a añadir un nodo Nome, Sexo y el campo DataNacemento con sus nodos de Texto correspondiente.

```
 Element minombre = documento.createElement("Nome");
 Text texto = documento.createTextNode("Sergio Rodriguez");
 mielem.appendChild(minombre);
 minombre.appendChild(texto);
 Element Sexo = documento.createElement("Sexo");
 Texto = documento.createTextNode("Home");
 mielem.appendChild(Sexo);
 Sexo.appendChild(texto);
 Element Data = documento.createElement("DataNacemento");
 text = documento.createTextNode("19/12/2000");
          mielem.appendChild(Data);
          Data.appendChild(text);
```

- **Para añadir el nodo antes de otro nodo**:

  El método del **insertBefore ()** inserta un nuevo nodo antes de un nodo existente.

  Sintaxis

  ```
  elementNode.insertBefore (nuevo nodo, nodo existente)
  ```

  - **nuevo nodo** : Es el nodo a insertar
  - **nodo existente**: El nuevo nodo se inserta antes de este nodo.

  Ejemplo:

```
 Element raiz = documento.getDocumentElement();
 Element nodo=(Element) raiz.getFirstChild();
 Element mielem = documento.createElement("Actor");
 raiz.insertBefore(mielem, nodo);
 Attr atrib = documento.createAttribute("id"); //creamos el atributo id
 atrib.setValue("2"); //se le da el valor 2
 mielem.setAttributeNode(atrib);
…….
…….
```

- **Remplazar un nodo por otro:**

  El método del replaceChild () substituye un nodo especificado.

  Sintaxis

  ```
  elementNode.replaceChild (nuevo nodo, nodo existente)
  ```

  - **nuevo nodo** : Es el nodo a remplazar
  - **nodo existente**: El nuevo nodo se remplaza por este nodo.

  Ejemplo:

```
 Element raiz = documento.getDocumentElement();
 Element nodo=(Element) raiz.getFirstChild();
 Element mielem = documento.createElement("Actor");
 raiz.replaceChild(mielem, nodo);
 Attr atrib = documento.createAttribute("id"); //creamos el atributo id
 atrib.setValue("2"); //se le da el valor 2
 mielem.setAttributeNode(atrib);
…….
…….
```

1.6 Eliminar nodos de un árbol DOM

El método del removeChild () elimina un nodo especificado del nodo padre.

Sintaxis

```
elementNode.removeChild (nodo)
```

- **nodo** : Es el nodo a eliminar

Ejemplo:

```
Element raiz = documento.getDocumentElement();
Element nodo=(Element) raiz.getFirstChild();
Element borrado= raiz.removeChild(nodo);
```

1.7 Clonar un nodo

El método del cloneNode () crea una copia de un nodo especificado.

Sintaxis

```
Element node=elementNode.cloneNode (Boolean valor)
```

- **nodo** : Es la variable que recibirá el nodo clonado.
- **ElementNode** : Es el elemento a ser copiado.
- **Valor**: indica si se va a copiar también todos los nodos descendientes (valor a true) o solo el nodo especificado.

Ejemplo:

```
Element raiz = documento.getDocumentElement();
Element nodo=(Element) raiz.getFirstChild();
Element clon=(Element) nodo.cloneNode(true);
raiz.appendChild(clon);
```

2. El paquete javax.xml.transform

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM. Pero podemos utilizar la clase javax.xml.transform.Transformer para realizar transformaciones. Esta clase, a su vez, nos permite transformar una fuente XML (en concreto nuestro Document (nuestro árbol DOM) en otra cosa (en nuestro caso, un fichero XML).

El paquete **javax.xml.transform** está diseñado para realizar modificaciones a documentos. Es usado mayoritariamente para manejar plantillas XLST. Permite especificar **una fuente y un resultado**. La fuente y el resultado pueden ser archivos, flujos de datos o nodos DOM entre otros.

2.1 Generar un fichero XML a partir de un árbol DOM

Para grabar en un archivo XML nuestro árbol DOM cargado en memoria sería crear una plantilla que no haga ninguna transformación y especificar como fuente un documento DOM y como destino un archivo. Adicionalmente mostraremos el documento por pantalla especificando como resultado el canal de salida (System.out).

Se necesita importar los siguientes paquetes:

```
//Para implementar el motor de transformación
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
//para tratar las posibles excepciones
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;
//necesaria para utilizar DOM como fuente
```

```
import javax.xml.transform.dom.DOMSource;
//para utilizer un flujo de salida como destino
import javax.xml.transform.stream.StreamResult;
//Para escribir el resultado en un fichero
import java.io.*;
```

Obteniendo una instancia de TransformerFactory

Lo primero que tenemos que hacer es implementar el motor de transformación. La clase javax.xml.transform.**TransformerFactory** posee un método estático **newInstance()** que permite obtener una implementación del transformador.

javax.xml.transform

Class TransformerFactory

java.lang.Object
    javax.xml.transform.TransformerFactory

Direct Known Subclasses:

SAXTransformerFactory

```
public abstract class TransformerFactory
extends Object
```

Método para instanciar un objeto TransformerFactory:

newInstance

```
public static TransformerFactory newInstance()
                                    throws TransformerFactoryConfigurationError
```

Lanza la excepción:

TransformerFactoryConfigurationError – Si la implementación del transformador no está disponible o no puede ser instanciado.

Podemos establecer los espacios de identación o sangrado mediante el método **SetAttribute**.

setAttribute

```
public abstract void setAttribute(String name,
                                  Object value)
```

- El atributo para identar es **"indent-number".**

```
TransformerFactory transFact = TransformerFactory.newInstance();
// añadimos sangrado cada tres espacios
transFact.setAttribute("indent-number", 3);
```

Construimos la implementación del motor de transformación.

La clase javax.xml.transform.**Transformer** implementa un motor para realizar transformaciones de un documento XML que puede provenir de una variedad de fuentes y escribir la transformación en una variedad de destinos.

Para instanciarlo se utiliza el método **newTransformer()** de la clase TransformerFactory.

newTransformer

```
public abstract Transformer newTransformer()
                                  throws TransformerConfigurationException
```

Lanza la excepción:

TransformerConfigurationException – Cuando no es posible crear una instancia Transformer.

Ej:

```
TransformerFactory transFact;
try{
  transFact=TransformerFactory.newInstance();
  // añadimos sangrado
  transFact.setAttribute("indent-number", 3);
  Transformer trans;
  try {
      trans = transFact.newTransformer();

      } catch (TransformerConfigurationException ex) {
          System.out.println("ERROR al construir el motor de transformación ");
      }
}catch(TransformerFactoryConfigurationError e){
  System.out.println("ERROR a la hora de implementar la transformación");
 }
```

Podemos **especificar propiedades para el transformador**. Uno de los métodos para especificar esas propiedades es **setOutputProperty();**

setOutputProperty

```
public abstract void setOutputProperty(String name,
                       String value)
                                throws IllegalArgumentException
```

Podemos **obtener las constantes que se pueden utilizar para fijar las características** de la salida para un transformador mediante la clase **OutputKeys**.

javax.xml.transform

Class OutputKeys

java.lang.Object
  javax.xml.transform.OutputKeys

```
public class OutputKeys
extends Object
```

Estas constantes son:

Modificador y Tipo	Campo y descripción
static String	**CDATA_SECTION_ELEMENTS** cdata-section-elements = *expanded names*.
static String	**DOCTYPE_PUBLIC** doctype-public = *string*.
static String	**DOCTYPE_SYSTEM** doctype-system = *string*.
static String	**ENCODING** encoding = *string*.
static String	**INDENT** indent = "yes" | "no". permite salto de lineas
static String	**MEDIA_TYPE** media-type = *string*.
static String	**METHOD** method = "xml" | "html" | "text" | *expanded name*.
static String	**OMIT_XML_DECLARATION** omit-xml-declaration = "yes" | "no".
static String	**STANDALONE** standalone = "yes" | "no".
static String	**VERSION** version = *nmtoken*.

Ej:

```
trans = transFact.newTransformer();
trans.setOutputProperty(OutputKeys.INDENT, "yes");
trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
```

Especificar la fuente XML

La fuente XML puede provenir de varios sitios: como por ejemplo, de un árbol DOM en memoria, de un fichero XML o de un analizador SAX.

- **Especificar la fuente de un árbol DOM:**

Como **fuente XML** usaremos la clase DOMSource, que se puede instanciar pasándole nuestro objeto **Document.**

```
DOMSource domSource = new DOMSource(documento);
```

Constructor
DOMSource() Constructor por defecto.
DOMSource(**Node** n) Crea una fuente de entrada desde un nodo DOM

Especificar el destino

Como destinos de la transformación podemos especificar varios, como un fichero, otro árbol DOM en memoria, etc...

- **Especificar un fichero como destino de la transformación**

Como destino de la transformación, usaremos un objeto **StreamResult**:

Constuctor
StreamResult() Constructor por defecto
StreamResult(**File** f) Construye un StreamResult desde un fichero.
StreamResult(**OutputStream** outputStream) Construye un StreamResult desde un flujo de bytes
StreamResult(**String** systemId) Construye un StreamResult desde una URL.
StreamResult(**Writer** writer) Construye un StreamResult desde un flujo de caracteres

Ej:

```
DOMSource domSource = new DOMSource(documento);
// creamos fichero para escribir en modo texto
FileWriter sw= new FileWriter("mificheroxml.xml");
 // escribimos todo el arbol en el fichero
StreamResult sr = new StreamResult(sw);
//También podemos especificar el destino en la consola
 StreamResult consola=new StreamResult(System.out);
```

Realizar la transformación

Para transformar la fuente XML al destino, se utiliza el método **transform** de la clase Transformer:

transform

```
public abstract void transform(Source xmlSource,
                               Result outputTarget)
                        throws TransformerException
```

Lanza la excepción:

TransformerException – Si ocurre un error irrecuperable durante el proceso de transformación.

```
    TransformerFactory transFact;
    try{
        transFact=TransformerFactory.newInstance();
       // añadimos sangrado y la cabecera de XML
        transFact.setAttribute("indent-number", 3);
        Transformer trans;
        try {
             trans = transFact.newTransformer();
             trans.setOutputProperty(OutputKeys.INDENT, "yes");
             trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
             DOMSource domSource = new DOMSource(documento);
             // creamos fichero para escribir en modo texto
             FileWriter sw= new FileWriter("mificheroxml1.xml");
             // escribimos todo el arbol en el fichero
              StreamResult sr = new StreamResult(sw);
             //TAmbién podemos especificar el destino en la consola
             StreamResult consola=new StreamResult(System.out);
              try {
                    trans.transform(domSource, sr);
                    trans.transform(domSource, consola);
                } catch (TransformerException ex) {};
             }
          } catch (TransformerConfigurationException ex) {
               System.out.println("ERROR al construir el motor de transformación ");
                 }
     }catch(TransformerFactoryConfigurationError e){
             System.out.println("ERROR a la hora de implementar la transformación");
     }
}
```

3. Creación de un fichero XML con DOM

3.1 DOMImplementation

La API de DOM define la interfaz **DOMImplementation** que permite **crear un objeto Document**.

Métodos de DomImplementatation	
Document	**createDocument**(String namespaceURI, String Nombre ,DocumentType doctype) trows **DOMException** Crea un objeto **document** de DOM .
DocumentType	**createDocumentType**(String Nombre, String publicId, String systemId) trows DOMException Crea un DocumentType nodo vacío.
boolean	**hasFeature**(String característica, String versión) Comprueba si la versión del API DOM implementa una característica específica.
Object	**getFeature**(String característica, String versión) Este método devuelve un objeto si la versión del API soporta una característica.

Creación del documento:

```
public Document createDocument(String namespaceURI, String Nombre,
                               DocumentType doctype)
                            throws DOMException
```

Parámetros:

namespaceURI - El URI del namespace del documento o null.

qualifiedName - El nombre del elemento raíz del documento a ser creado.

doctype – Referencia al DTD asociado o null

Ejemplo:

```
DOMImplementation domImpl = new DOMImplementationImpl();
Document doc =domImpl.createDocument(null, "Empleados", null);
```

3.2 Crear un fichero XML

- **1ª.- Forma**: Construcción de un árbol DOM vacío:

Crear un nuevo documento es algo trivial. Basta con invocar al método **newDocument()** de **DocumentBuilder:**

```
try {
DocumentBuilderFactory dbf =DocumentBuilderFactory.newInstance();
DocumentBuilder constructor = dbf.newDocumentBuilder();
Document documento = constructor.newDocument();
}
catch (ParserConfigurationException e) { ... }
```

2ª.-Forma:

En la primera forma obtendremos un documento vacío. Un documento vacío no es un documento bien formado ya que un documento debe contener al menos un, y no más de un, elemento raíz.

Una forma más adecuada para crear documentos a través de la interfaz org.w3c.dom.**DOMImplementation**. Esta interfaz nos permitirá crear documentos con un **elemento raíz y opcionalmente con un DTD asociado.**

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
DOMImplementation implementacion=db.getDOMImplementation();
Document doc=implementacion.createDocument(null, "Empleados", null);
//Se pueden establecer características al documento xml
 doc.setXmlStandalone(false);
 doc.setXmlVersion("1.1");
// creamos los nodos de nuestro documento */
  Element nodo = doc.createElement("Empleado");
  Text texto = doc.createTextNode("Pepito Pérez Pérez");

// añadimos cada nodo a su padre
   nodo.appendChild(texto);
 //si se quiere añadir un atributo
  nodo.setAttribute("id", "1");
  doc.getDocumentElement().appendChild(nodo);
```

El árbol DOM creado en memoria corresponde al siguiente document XML

```
<?xml version="1.1" encoding="UTF-8" standalone="no"?>
<Empleados>
    <Empleado id="1">Pepito Pérez Pérez</Empleado>
</Empleados>
```