LPS: XML en Java

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

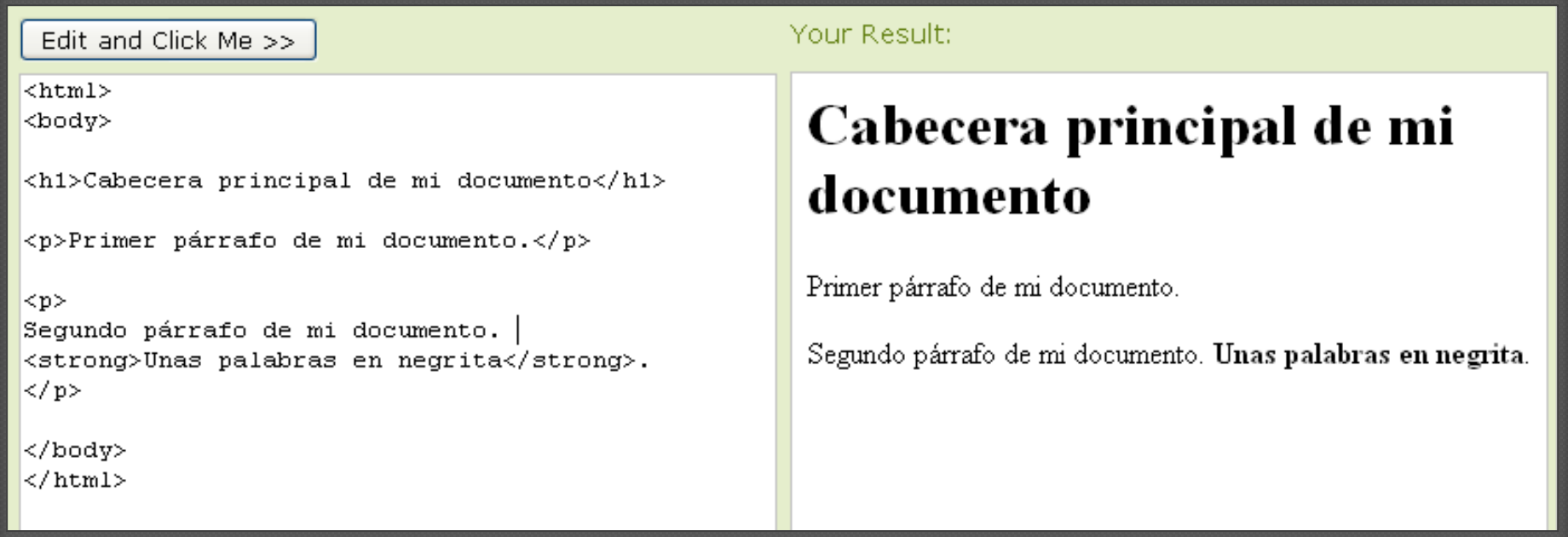
Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Lenguajes de marcado

- En principio, nada que ver con DOO ni POO
- No son lenguajes para programar, sino para estructurar documentos de manera explícita, añadiendo ciertas "marcas" en determinados puntos del documento
- Probablemente el más popular es HTML (*HyperText Markup Language*)
  - Sirve para definir el contenido de una página web, la disposición de los elementos que debe visualizar un navegador web
  - Las "marcas" en este caso se conocen como *etiquetas*
    - Por ejemplo, se usan estas dos para delimitar un párrafo: `<p>Texto ...</p>`

Ejemplo de uso



http://www.w3schools.com/html/

XML

XML

- XML (*eXtensible Markup Language*) es un estándar para lenguajes de marcado del W3C (*World Wide Web Consortium*)
- Diseñado para describir documentos estructurados y cualquier información en forma de texto
  - Los documentos llevan contenidos con marcas
  - Las marcas aquí también se denominan *etiquetas*
    - Son identificadores encerrados entre **<** y **>**
    - Crean una estructura jerárquica, equivalente a un árbol
- En realidad se trata de un meta-lenguaje
  - Permite definir lenguajes de marcado *específicos* para una aplicación concreta
    - No tiene etiquetas predefinidas, hay que definirlas según la aplicación
  - Ejemplo: XHTML es una versión de HTML, pero definida mediante el meta-lenguaje estándar XML

Ventajas

- Ofrece una sintaxis estándar para todos los lenguajes de marcado
  - Permite simplificar el tratamiento automático de este tipo de información
- Existen varios lenguajes asociados que lo hacen aún más potente
  - DTD
  - XML Schema
  - XSLT
  - …
- Disponemos de muchas herramientas y software ya creado
  - Analizadores (*Parsers*)
  - Generadores
  - Intérpretes
  - Editores
  - …
- Permite comprobar *formalmente* si un documento es "correcto"
- …
- ¡Ojo! XML no es ninguna "bala de plata": no tiene porque ser la mejor solución para todos nuestros problemas

Sintaxis básica

- Las etiquetas definen cada uno de los elementos
  - Pueden servir para marcar unos ciertos contenidos:

```
<nombre_etiqueta>    Marca el inicio de este elemento
  Contenido  (puede contener a su vez más etiquetas)
</nombre_etiqueta>   Marca el final de este elemento
```

  - Pueden estar vacías de contenidos:

```
<nombre_etiqueta />
```

- Las etiquetas pueden llevar atributos asociados

```
<nombre_etiqueta nombre_atributo1="valor_atributo1"
  nombre_atributo2="valor_atributo2" ...>
...
</nombre_etiqueta>
```

Ejemplo de documento XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited by XMLSpy® -->

<notes>
 <note type="Post-it">
   <to>Paco</to>
   <from>Juana</from>
   <heading>Lista de la compra</heading>
   <body>¡Acuérdate de la leche!</body>
 </note>

 <note type="Correo electrónico">
   <to>Aurora</to>
   <from>Manuel Esteban</from>
   <heading>Re: Felicitación</heading>
   <body>Gracias por tu felicitación...</body>
 </note>
...
</notes>
```

Más sobre sintaxis

- Los comentarios se delimitan mediante las etiquetas `<!--` y `-->`
- Los documentos XML son sensibles a minúsculas y mayúsculas
- Para poder usar caracteres reservados en XML como son `<`, `>` y `&` hay que utilizar *combinaciones especiales de caracteres* en el contenido como `&lt;`, `&gt;`, `&amp;` y otras
- Los valores de los atributos deben ir siempre *entrecomillados*, para lo que puede usarse la comilla doble o la simple
- Todo documento debe tener un único elemento como raíz del árbol de la jerarquía de elementos

DTD

Corrección de un documento XML

- Un documento XML es "correcto" si está bien formado y es válido
- Bien formado: que cumple la sintaxis básica que impone XML en general, en cuanto a apertura y cierre de etiquetas, uso de atributos, etc.
  - Esto sería un documento *mal formado*:

```
<notes>
 <note type="Post-it">
   </to> <to>Paco
   <from>Juana</from>
...
```

- Válido: que cumple con las *normas semánticas* establecidas para el lenguaje de marcado específico que hayamos creado
  - Esto, según las normas de XHTML sería un documento *inválido*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head> <title>Título</title> ... </head>
<p>Párrafo fuera de su sitio.</p>
<body> ... </body>
</html>
```

DTD

- Para definir un lenguaje derivado de XML es necesario definir una *gramática* que especifique:
  - Etiquetas válidas
  - Atributos válidos
  - Jerarquía existente entre las distintas etiquetas
- DTD (Document Type Definition) es un lenguaje específico con el que se puede definir dicha gramática
  - Una DTD proporciona los criterios con los que podremos *validar* una serie de documentos XML

Sintaxis básica

- Las declaraciones en una DTD tienen esta forma:

  `<!keyword parámetro1 parámetro2 ... parámetroN>`
- Hay 4 palabras reservadas básicas
  - ELEMENT: Declara el nombre de un elemento y a continuación sus posibles *subelementos*
  - ATTLIST: Declara los nombres de los atributos de un elemento, así como sus posibles *valores* y/o *valor por defecto*
  - ENTITY: Declara referencias a caracteres especiales o a bloques de texto (similar a un **`#define`** de C++) o también a contenido que va a ser repetido y que puede estar en un recurso externo (similar a un **`#include`** de C++).
  - NOTATION: Declara contenido externo "no-XML" (por ejemplo, ficheros con imágenes), indicando la aplicación externa que es capaz de gestionar dicho contenido

Sintaxis básica: Elementos

- La declaración de un elemento tiene esta forma:
  `<!ELEMENT nombre_elemento contenido>`
  - *nombre_elemento* es el nombre de la etiqueta que corresponde al elemento que estamos definiendo
- Hay 5 tipos posibles de contenido
  - ANY: El elemento puede contener cualquier XML bien formado
  - EMPTY: El elemento no puede contener nada (aunque puede tener *atributos*)
  - Texto: El elemento sólo puede contener *texto*, pero sin subelementos (se indica con `(#PCDATA)`)
  - Subelementos: El elemento sólo contiene los subelementos que se mencionen
  - Mixto: El elemento puede contener *tanto texto como subelementos*

Organización de subelementos

- Cuando el contenido es del tipo *Subelementos* o *Mixto*, se utiliza una expresión regular que especifica cómo debe organizarse dicho contenido
- Ejemplos de expresiones regulares:
  - Una colección que contiene uno o más libros
    ```
    <!ELEMENT coleccion (libro)+>
    ```
  - Un libro que contiene un título, cero o más autores, y cero o una ediciones
    ```
    <!ELEMENT libro (titulo, autor*, edicion?)>
    ```
  - Una edición tiene una editorial, una colección y un año (todos opcionales)
    ```
    <!ELEMENT edicion (editorial?, coleccion?, año?)>
    ```

Sintaxis básica: Atributos

- Los atributos se utilizan para asociar pares nombre-valor a los elementos
- La declaración comienza con la palabra reservada **ATTLIST** seguida por el nombre del elemento al que pertenecen los atributos y por la definición de cada uno de los atributos individuales
  - El orden en que se presentan los atributos es indiferente
- Cada atributo puede tener un nombre, un tipo, una definición de característica y un valor por defecto.
- Ejemplo

```
<!ATTLIST nombreElemento
  nombreAtributo1 tipo1 caracteristica1 valorPorDefecto1
  ...
  nombreAtributoN tipoN caracteristicaN valorPorDefectoN>
```

Tipos de atributos

- Hay 4 tipos básicos para los atributos
  - CDATA: Datos formados únicamente por caracteres (es decir, *cadenas de texto*)
  - Valores enumerados
    - Se proporciona el conjunto de todos los valores permitidos
    - Opcionalmente puede darse un valor por defecto
  - ID: Identificador único por cada ejemplar del elemento
    - El analizador debe comprobar que efectivamente el valor de este atributo sea único para cada ejemplar en el documento
  - IDREF: Una referencia al identificador de un elemento
    - El analizador debe comprobar que efectivamente hay un ejemplar del elemento con ese identificador en el documento

Características de los atributos

- Las características indican cómo debe comportarse un analizador si un determinado atributo *no aparece* en un documento XML
- Hay 4 posibles características
  - #REQUIRED: El atributo es necesario, por lo que debería estar siempre presente en los ejemplares del elemento en el documento
  - #IMPLIED: El atributo es opcional
  - #FIXED: El atributo es opcional y además:
    - Si aparece debe coincidir con el valor por defecto
    - Si no aparece el analizador puede darle el valor por defecto
  - Por defecto (sin palabra clave): El atributo es opcional y además:
    - Si aparece debe tener un valor adecuado para su tipo
    - Si no aparece el analizador puede darle el valor por defecto

Prólogo de un documento XML

- Todo documento XML (tenga o no DTD) debe empezar con esta línea:

```
<?xml version="num_versión" encoding="codificación" ?>
```

  - *num_versión* = Número de versión del estándar XML
  - *codificación* = Sistema de codificación de los caracteres del documento (ISO-8859-1, UTF-8, etc.)
- Además, a continuación se puede añadir una referencia a la DTD que lo valida:

```
<!DOCTYPE nombre SYSTEM "ruta" >
```

  - *nombre* = Nombre lógico de la gramática del DTD
  - *ruta* = Ruta que lleva al fichero DTD

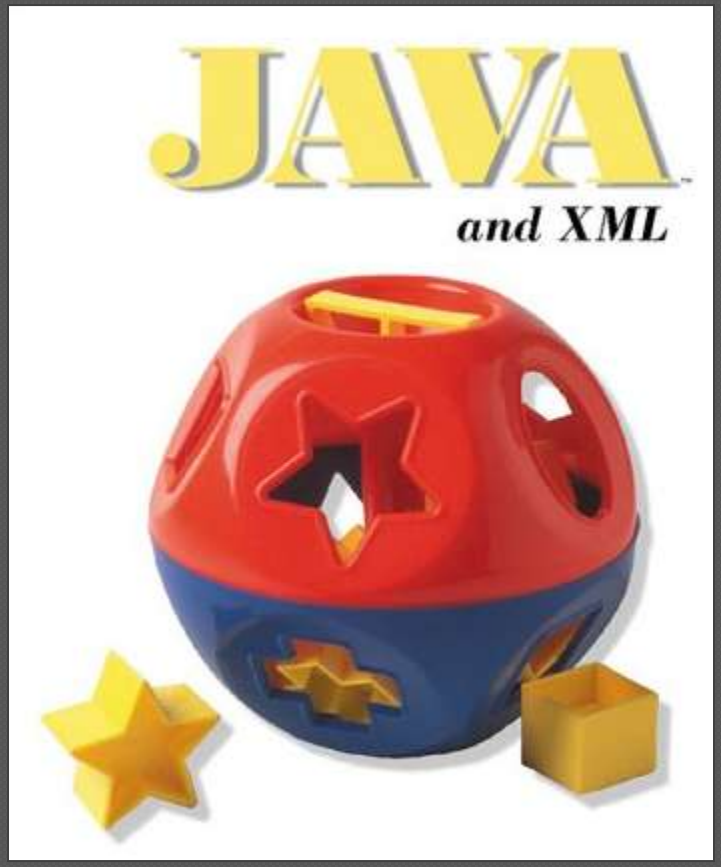
Ejemplo: XML con DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tablero SYSTEM "barcos.dtd">
<tablero ancho="12" alto="12">
    <barco tipo="portaaviones">
        <posicion x="1" y="2"/>
        <posicion x="1" y="3"/>
        <posicion x="1" y="4"/>
        <posicion x="1" y="5"/>
    </barco>
    <barco tipo="submarino">
        <posicion x="5" y="8"/>
    </barco>

    <!-- Definición del resto de barcos... -->

    <casilla-especial tipo="tierra">
        <posicion x="2" y="0"/>
    </casilla-especial>
</tablero>
```

Ejemplo: DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT tablero (barco+, casilla-especial*)>
<!ATTLIST tablero
        ancho CDATA #REQUIRED
        alto CDATA #REQUIRED >

<!ELEMENT barco (posicion+)>
<!ATTLIST barco
   tipo (portaaviones|destructor|fragata|submarino)
   #REQUIRED >

<!ELEMENT casilla-especial (posicion)>
<!ATTLIST casilla-especial
   tipo (tierra) #REQUIRED >

<!ELEMENT posicion EMPTY>
<!ATTLIST posicion
   x CDATA #REQUIRED
   y CDATA #REQUIRED >
```

XML en Java



XML en Java

- Existen diversas APIs para manejar XML desde una aplicación Java
  - JAXP (*Java API for XML Processing*) es probablemente la más popular
    - SAX (Analizador basado en eventos)
    - DOM (Analizador tipo árbol)
    - Transformer (transformador de documentos XML)
    - …

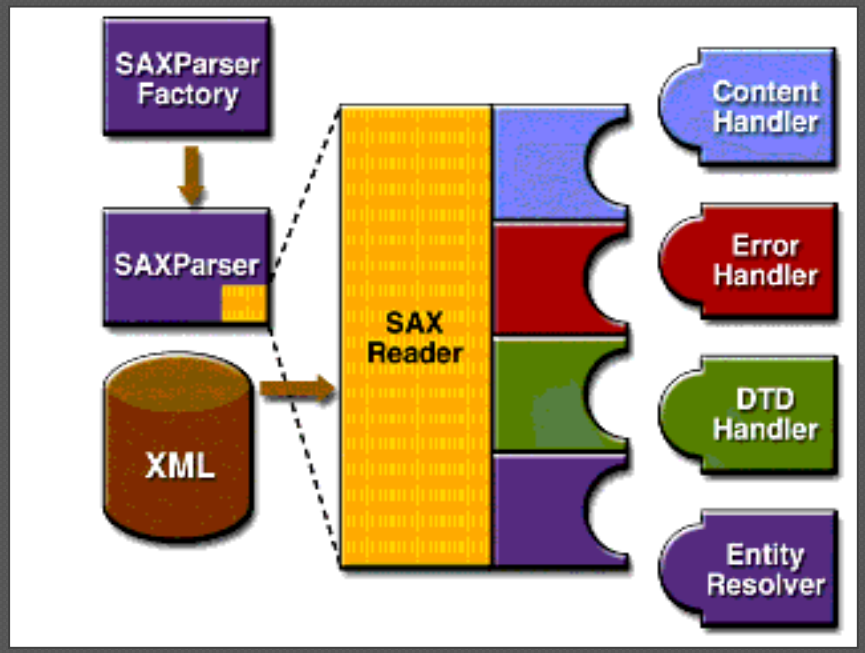
SAX versus DOM

- Representan dos filosofías distintas de procesar documentos XML
  - SAX realiza rápidamente *una única pasada* por todo el documento
    - Requiere poca memoria principal
    - Es necesario disponer del documento íntegro para terminar de analizarlo
    - Muy usado para operaciones de sólo lectura sobre un documento XML
  - DOM copia el árbol de elementos XML (*total o parcialmente*) en memoria principal usando objetos Java y nos permite trabajar sobre ellos
    - Requerirá más memoria cuanto mayor sea el documento XML
    - Permite crear y modificar elementos dentro del documento XML
    - Normalmente no se usa para leer un fichero ya existente

SAX

Uso de SAX

- El uso de SAX (*Simple API for XML*) consiste en los siguientes pasos:
  1. Crear un *SAXParser* (a través de una Factoría)
  2. El *SAXParser* contiene un *SAXReader*
  3. El *SAXReader* procesa el documento completo y va lanzando eventos según el contenido los elementos que va encontrando
  4. Habrá una clase Oyente que va "escuchando" esos eventos y actúa en consecuencia



"Escuchando" los eventos SAX

- Nosotros debemos construir esa clase Oyente, para lo que hay 4 interfaces que implementar
  - ContentHandler
    - **`startDocument`** y **`endDocument`**: Métodos llamados al empezar y al terminar de procesar el documento
    - **`startElement`** y **`endElement`**: Métodos llamados al comenzar y al terminar de procesar cualquier elemento XML
    - **`characters`**: Método llamado al encontrar texto dentro de un elemento
  - ErrorHandler
    - **`error`**, **`fatalError`** y **`warning`**: Métodos para tratar distintos problemas que pueden producirse durante el análisis
  - DTDHandler
    - Se usa si queremos definir un analizador de DTDs
  - EntityResolver
    - **`resolveEntity`**: Método llamado al encontrar una referencia en el XML que deba ser resuelta (como una URI, por ejemplo)

DefaultHandler

- Habitualmente no interesa implementar *todos los métodos* de dichas 4 interfaces
- DefaultHandler da una implementación por defecto para todos ellos
  - Implementaciones básicas de operaciones "de servicio"
  - Implementaciones vacías para muchas operaciones
- Podemos crear nuestra clase Oyente heredando de DefaultHandler para aprovechar todas esas implementaciones

Ejemplo de análisis

```
// Fichero a procesar
File archivoXML = new File("barcos.xml");

// Creamos el parser empleando la Factoría
  (que es un Ejemplar Único)
SAXParserFactory factory =
  SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();

// Nuestro oyente heredará de DefaultHandler
DefaultHandler oyente = new Oyente();

// Lanzamos el proceso de parseo, siendo
  nuestro oyente uno de los argumentos
parser.parse(archivoXML, oyente);
```

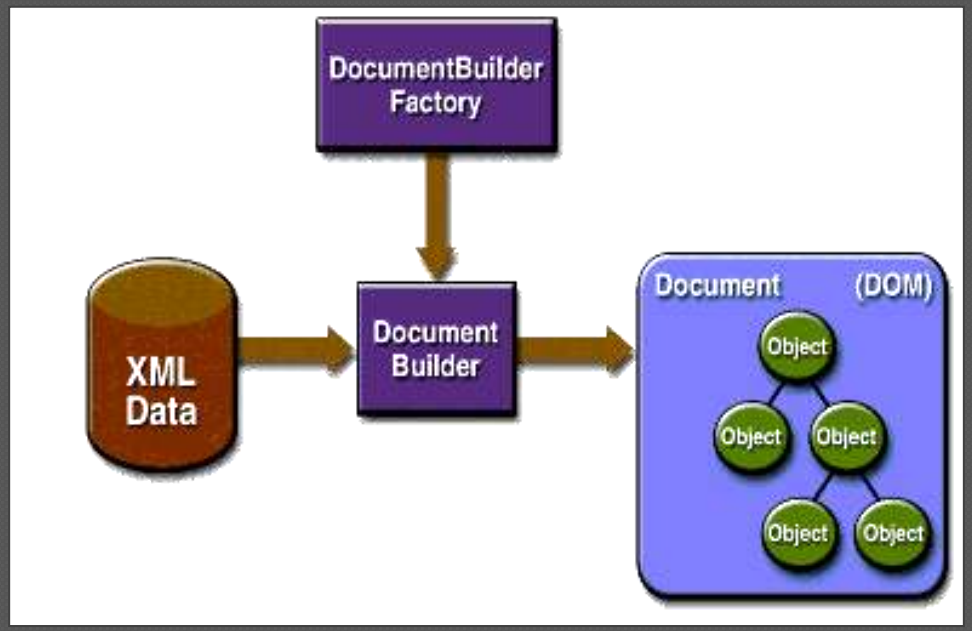
Ejemplo de Oyente

```
public class Oyente extends DefaultHandler {

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

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

    public void startElement(String namespace, String sName,
                  String qName, Attributes atrs) throws SAXException {
      System.out.println("Elemento: " + qName);
      if (atrs != null) {
          for(int i=0; i < atrs.getLength(); i++) {
              System.out.println("Atributo: " + atrs.getQName(i)
                      + " = " + atrs.getValue(i);}
      }
    }

    public void characters(char buf[], int offset, int len) {
      String aux = new String(buf, offset, len);
      System.out.println("Texto: " + aux);
    }
}
```

DOM

Uso de DOM

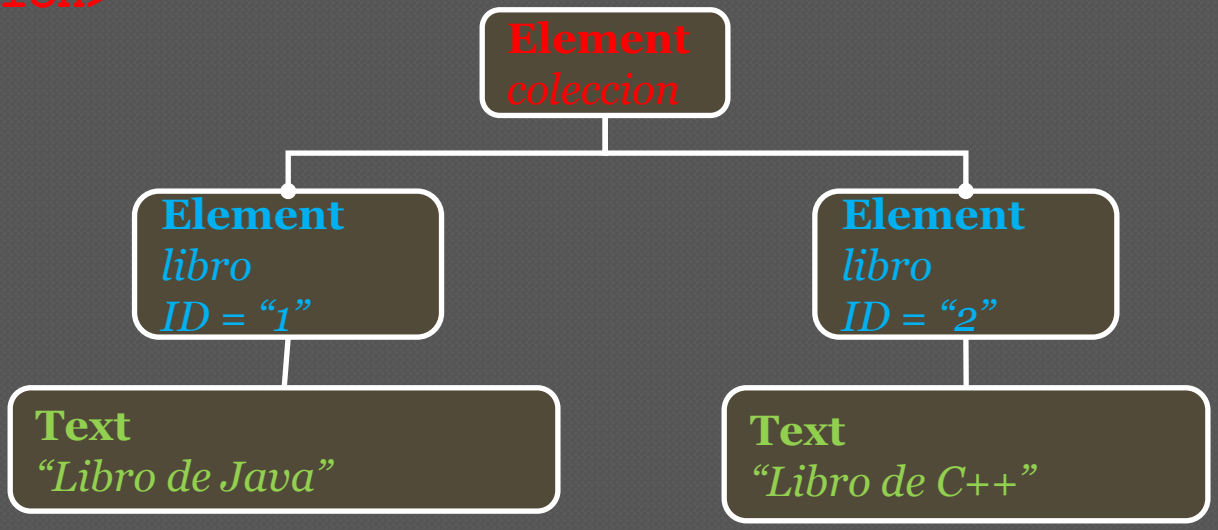
- El uso de DOM (*Document Object Model*) consiste en los siguientes pasos:
  - Crear un *DocumentBuilder* mediante una Factoría
  - *DocumentBuilder* lee un documento XML y crea en memoria principal un árbol de objetos Java que se corresponde con la estructura del mismo
    - *DocumentBuilder* también puede crear árboles vacíos si hiciera falta
  - Usar dicho árbol para navegarlo (pudiendo *añadir*, *eliminar* o *modificar* elementos si hiciera falta)



Estructura del árbol DOM

- Curiosamente, los fragmentos de texto del documento XML se consideran nodos del árbol, como los propios elementos XML
  - DOM no los sitúa *dentro* directamente de los propios nodos de los elementos
- Sin embargo, los atributos de los elementos *sí están dentro* de los nodos

```
<?xml version="1.0"?>
<coleccion>
  <libro ID="1">Libro de Java</libro>
  <libro ID="2">Libro de C++</libro>
</coleccion>
```



Ejemplos de construcción

- Construcción de un árbol DOM vacío

```
try {
    DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.newDocument();
}
catch (ParserConfigurationException e) { ... }
```

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

```
try {
    File fichero = new File("coleccion.xml");
    DocumentBuilderFactory factory =
                    DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse(fichero);
}
catch (ParserConfigurationException e) { ... }
```

Manejo del árbol DOM

- **`org.w3c.dom.Node`** es una interfaz con métodos para organizar un árbol
  - **`getParent()`**: Devuelve el nodo padre
  - **`getChildNodes()`**: Devuelve una lista de hijos
  - **`appendChild(Node newChild)`**: Añade un hijo al nodo actual
  - **`removeChild(Child hijo)`**: Elimina el nodo hijo indicado
- Generalmente, se emplean interfaces que extienden **`Node`**
  - **`Element`**: Para representar cualquier elemento XML
  - **`Text`**: Para representar un fragmento de texto
  - **`Attr`**: Para representar los atributos de un elemento
  - **`Document`**: Para representar el árbol completo y crear más nodos

Elementos DOM

- Métodos más comunes de **`org.w3c.dom.Element`**
  - **`String getAttribute(String s)`**: Devuelve el valor del atributo llamado "s"
  - **`NodeList getElementsByTagName(String s)`**: Devuelve una lista con todos los subelementos con la etiqueta "s"
  - **`String getTagName()`**: Devuelve la etiqueta del propio elemento
  - **`boolean hasAttribute(String s)`**: Pregunta si hay un atributo llamado "s"
  - **`void setAttribute(String nombre, String valor)`**: Añade un nuevo atributo con un cierto "nombre" y "valor"
  - **`void removeAttribute(String nombre)`**: Elimina el atributo llamado "nombre"

Texto y atributos DOM

- **`org.w3c.dom.Text`**
  - **`String getWholeText()`**: Devuelve el contenido textual del nodo
  - **`Text setWholeText(String texto)`**: Cambia el contenido textual del nodo
- **`org.w3c.dom.Attr`**
  - **`String getName()`**: Devuelve el nombre del atributo
  - **`String getValue()`**: Devuelve el valor del atributo
  - **`void setValue(String valor)`**: Cambia el valor del atributo

Documentos DOM

- **`org.w3c.dom.Document`**
  - Por un lado representa al documento XML completo
  - Por otro lado actúa como Factoría para crear nuevos nodos
    - Los nodos sólo pueden colocarse directamente en aquel documento que los creó
  - ¡Ojo! Sólo puede tener un único hijo (= el nodo raíz del documento)
- Métodos más habituales
  - **`Element createElement(String nombre)`**: Crea un nodo de tipo **`Element`**
  - **`Attr createAttribute(String nombre)`**: Crea un nodo de tipo **`Attr`**
  - **`Text createTextNode(String texto)`**: Crea un nodo de tipo **`Text`**
  - **`Node adoptNode(Node fuente)`**: Intenta "adoptar" un nodo creado en un documento distinto, colocándolo dentro de este

Transformaciones

Transformaciones

- La clase **`javax.xml.transform.Transformer`** implementa un motor para realizar transformaciones a documentos XML
  - Los ejemplares se obtienen de Factorías de este tipo: **`javax.xml.transform.TransformerFactory`**
- Dispone del método **`transform(fuente, destino)`**
  - La fuente debe ser un objeto de tipo: **`javax.xml.transform.Source`**
  - El destino debe ser un objeto de tipo: **`javax.xml.transform.Result`**
- Ambas interfaces (*Source* y *Result*) disponen de implementaciones para representar a un documento XML basado en DOM, SAX o en un flujo de datos (*Stream*) genérico

Implementaciones de Source

- DOMSource
  - Se construye pasándole como parámetro el nodo raíz del subárbol DOM a tratar
- SAXSource
  - Se construye a partir de un **`InputStream`** conectado al fichero que queremos tratar
  - **`getInputSource`** y **`setOutputSource`** nos permiten acceder y modificar el flujo de datos
- StreamSource
  - Se construye a partir de un objeto de tipo **`File`**, de un **`InputStream`** (flujo de datos *binarios*) o un **`Reader`** (lector de caracteres)

Implementaciones de Result

- DOMResult
  - Se construye pasándole como parámetro un objeto de tipo nodo, que actúa como raíz del árbol donde queremos colocar la salida de la transformación
  - Con `setNextSibling(Node node)` se puede especificar en que punto del árbol queremos colocar la salida de la transformación
- SAXResult
  - La salida del transformador se analiza directamente como un flujo SAX
  - Se construye a partir de un `ContentHandler`, que será quién escuche los eventos emitidos al analizar la salida del transformador
- StreamResult
  - La salida se escribe directamente en un flujo de datos
  - Se pueden construir a partir de un objeto de tipo `File`, de un `OutputStream` (flujo de datos *binarios*) o un `Writer` (escritor de caracteres)

Ejemplo de transformación

- Usamos un transformador para pasar un árbol DOM (con *DOMSource*) a un fichero normal (con *StreamResult*)

```
//Creación del transformador a partir de una factoría
TransformerFactory factoria =
   TransformerFactory.newInstance();
Transformer transformer = factoria.newTransformer(); //
   Podría llevar como argumento un documento XSLT

//Creación de un Source a partir del árbol DOM
DOMSource origen = new DOMSource(arbolDOM);

//Creación de un Result a partir del fichero de destino
File ficheroDestino = new File("barcos.xml");
StreamResult destino = new StreamResult(ficheroDestino);

transformer.transform(origen, destino);
```

- En realidad a partir de Java 5, hay métodos para cargar y guardar árboles DOM en el paquete `org.w3d.dom.loader`

Críticas, dudas, sugerencias…

Federico Peinado

www.federicopeinado.es