Tutorato Progettazione, Algoritmi e Computabilità

Linguaggi per lo scambio di dati in rete

Dott. Michele Beretta

Prof.ssa Patrizia Scandurra

Università degli Studi di Bergamo

2023 – 2024

Outline

XML

Linguaggio XML

L'acronimo **XML** sta per *eXtensible Markup Language*.

In realtà XML non è un vero e proprio linguaggio, ma un insieme di regole utilizzate per costruire particolari linguaggi. Si parla infatti di *metalinguaggio*.

Un esempio di linguaggio realizzato con XML è *HTML* che utilizza un particolare vocabolario con tag predefiniti, come `table`, `body`, etc.

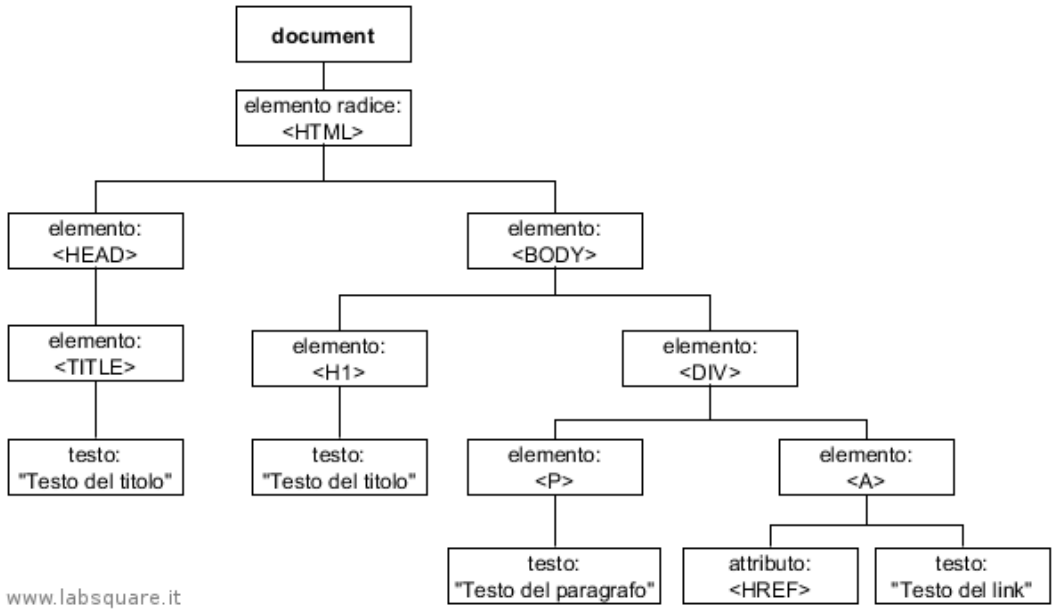
Linguaggio XML

XML consente una rappresentazione basata su testo per descrivere contenuti (serializzare) e scambiare dati strutturati:

- Tra applicazioni, anche di sistemi diversi, in quanto XML ha un formato standard, facilmente interpretabile con un parser.
- Tra umani e calcolatori

Linguaggio XML

La rappresentazione di XML è una tipica rappresentazione ad **albero** con un unico **nodo padre** come radice.



Parser XML

Un parser XML ha il compito di interpretare la sintassi XML di un qualunque documento *ben formato* (cioè sintatticamente corretto) al fine di estrarne delle informazioni.

Le informazioni lette possono poi essere:

- Mappate su oggetti
- Salvate in strutture dati

Parser XML

Un parser XML si occupa, in generale, di:

- Recuperare il documento XML.
- Caricare i dati estratti in memoria.
- Offrire all'applicazione un'interfaccia di alto livello per l'accesso ai dati.

Opzionalmente, possono essere offerti servizi di:

- Validazione dei dati
- Ricerca di dati

Struttura di un file XML

Un file XML normalmente inizia con la riga seguente:

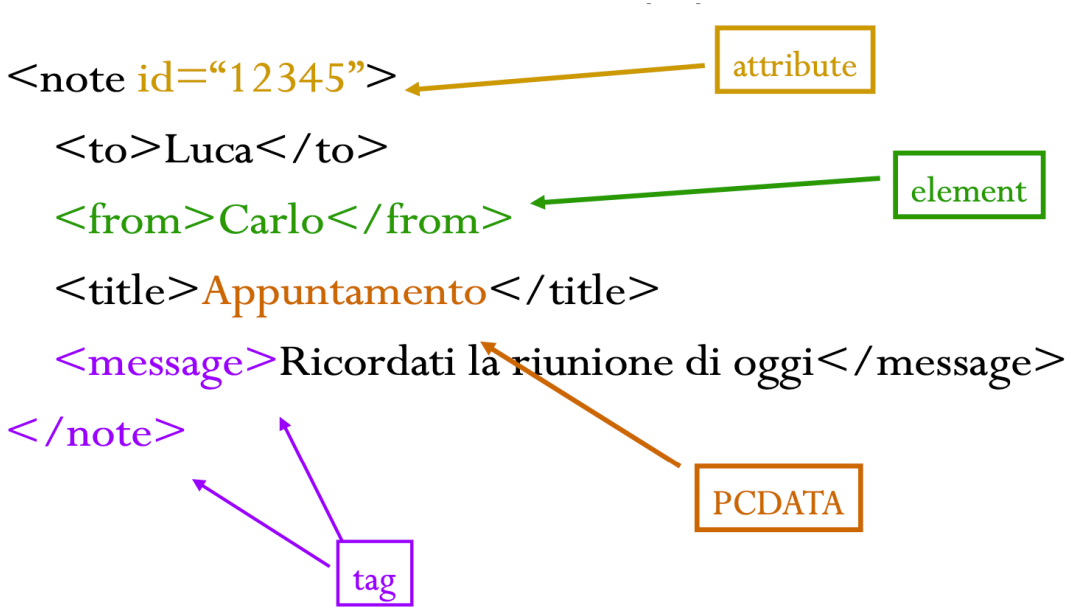
```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

in cui:

- `version`: indica la versione delle specifiche XML a cui il documento è conforme.
- `encoding`: indica il tipo di codifica utilizzata per i caratteri (ASCII-7, ASCII-8, UTF-8, UTF-16).
- `standalone`: indica se il documento si basa su altri documenti.

Struttura di un file XML

Un **elemento XML** è tutto ciò che è compreso fra un tag di apertura ed il corrispondente tag di chiusura, *tag compresi*.



Struttura di un file XML

Tra i due tag si trova il contenuto dell'elemento. In base al tipo di contenuto si parla di:

- *Element content*: quando il contenuto è costituito da altri elementi (ad esempio l'elemento `<note>` della slide precedente).
- *Simple content*: se il contenuto è semplice testo.
- *Mixed content*: se contiene testo alternato ad altri elementi.
- *Empty content*: se il contenuto è vuoto, ad esempio il tag `<img>` di HTML (che ha attributi ma non contenuto).

Per gli elementi vuoti la coppia apertura/chiusura può essere sostituita dal tag vuoto `<nometag />`.

Attributi

- Gli attributi sono informazioni aggiuntive che possono essere inserite negli elementi XML (caratteristica presente anche in HTML, ad esempio l'attributo `src` del tag `<img>`).
- Vengono aggiunti solo nei tag di apertura o nei tag vuoti.
- Devono essere racchiusi tra apici (singoli o doppi).

```
<img src="logo.gif" />
<message language="IT">Remember today's meeting</message>
```

Regole sintattiche di XML

Nonostante XML sia molto flessibile, è comunque necessario rispettare alcune regole sintattiche:

- Tutti i tag aperti devono essere chiusi.
- I tag devono essere correttamente annidati.
- Ogni documento XML deve avere uno ed un solo elemento radice.
- Gli attributi devono essere racchiusi tra apici (singoli o doppi).
- XML è case sensitive.
- Gli spazi nel testo sono preservati (`<tag> text </tag>`).
- I commenti hanno la notazione `<!- comment ->`.

XML e Java

In questo corso ci occuperemo di come interagire con dati XML utilizzando il linguaggio di programmazione Java.

In particolar modo, come utilizzare queste 2 API:

- DOM (Document Object Model)
- SAX (Simple API for XML)

XML e Java

	APPROCCIO AD EVENTI (SAX)	APPROCCIO DEL MODELLO (DOM)
PRO	• Molto leggero • Permette di implementare solo le funzionalità necessarie (efficienza)	• Fornisce all'applicazione un modello ricco del documento • Rappresentazione persistente e completa del documento in memoria
CONTRO	• Interfaccia a volte tropo semplice, più codice richiesto nell'applicazione • Nessun supporto per operare sul documento	• Occorre memorizzare tutto il documento (spazio)

XML – Approccio SAX

Approccio SAX

Con questa tipologia di approccio, il parser:

- Scandisce l'intero file.
- Per ogni elemento informa l'applicazione tramite la tecnica del callback.

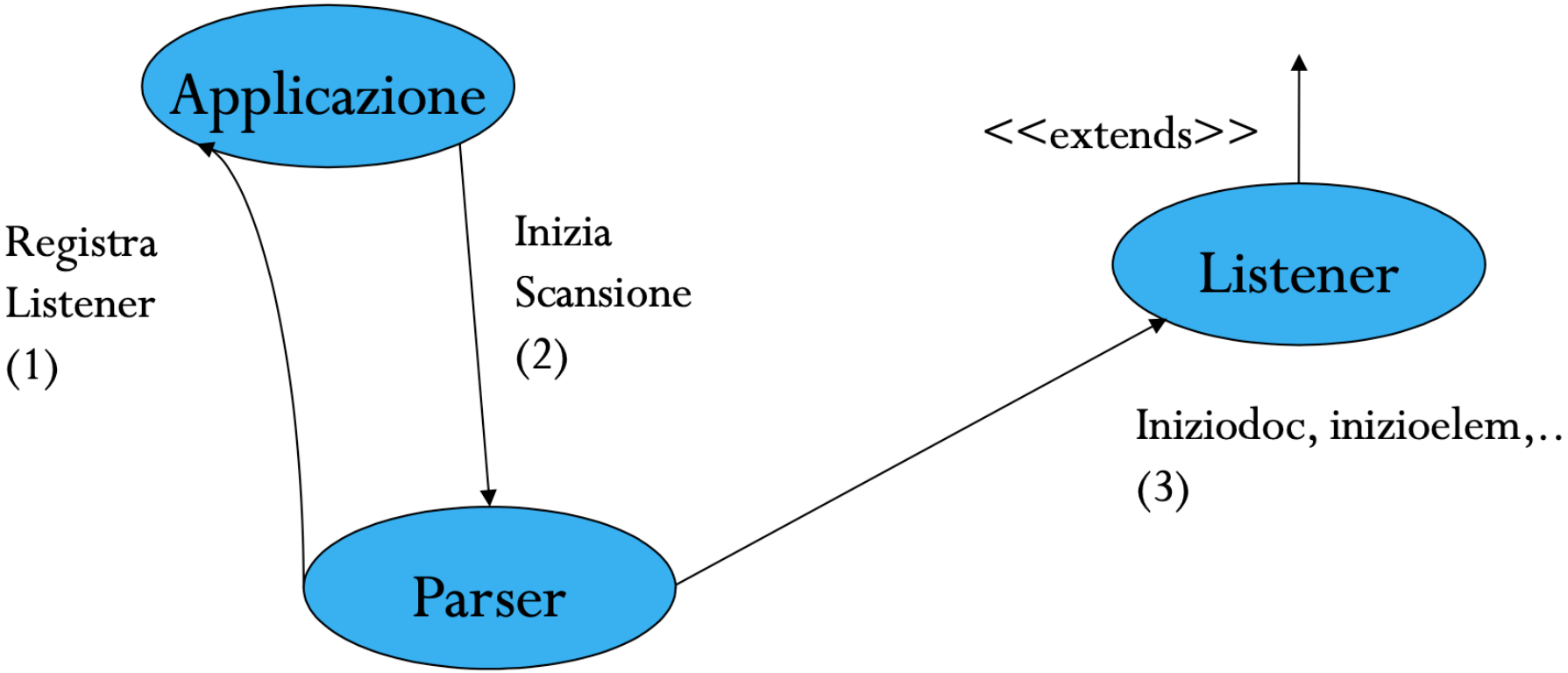
Importante è ricordare che con SAX è solamente possibile leggere file XML e non crearli/modificarli.

Approccio SAX

Il sistema è basato su eventi che vengono generati in varie occasioni durante la scansione del testo:

- Inizio e fine di un documento.
- Apertura e chiusura di un elemento.
- Apertura e chiusura di un blocco CDATA.
- Ingresso e uscita dallo scope di un namespace.
- Caratteri, Process Instruction, spazi non significativi.

Approccio SAX



Approccio SAX

SAX adotta il *source-listener-delegate* model per il parsing:

- Il source è formato dai dati XML (elementi).
- Un listener viene associato al documento per catturare gli eventi.
- Quando un evento si verifica, viene notificato il listener, e un opportuno metodo (callback) viene richiamato.

È compito del programmatore definire i metodi di callback associati agli eventi.

Gestione degli eventi con SAX

Per ricevere le notifiche occorre registrare nel parser un oggetto che implementa l'interfaccia: `org.xml.sax.helpers.ContentHandler`.

- Essa dichiara i metodi di callback.
- Ciascun metodo riceve, attraverso i parametri, tutte le informazioni necessarie a trattare correttamente l'evento associato.

Ad esempio, il metodo `startElement(...)` riceve il nome dell'elemento, il namespace e la lista degli attributi.

Interfaccia ContentHandler

```
interface ContentHandler {
  void setDcoumentLocator(Locator locator);
  void startDocument();
  void startElement(String namespaceURI,
    String localName, String qName, Attributes atts);
  void startPrefixMapping(String prefix, String uri);
  void characters(char[] ch, int start, int length);
  void endDocument();
  void endElement(String namespaceURI,
    String localName, String qName);
  void endPrefixMapping(String prefix);
  void ignorableWhitespace(char[] ch,
    int start, int length);
  void processingInstruction(String target,
    String data);
  void skippedEntity(String name);
}
```

Classe `DefaultHandler`

La classe `DefaultHandler` è predefinita e implementa l'interfaccia, permettendo di ridefinire solo i metodi desiderati.

Utilizzo dell'handler

Il client utilizza la classe factory `SAXParserFactory` per recuperare un'istanza di un parser. Viene chiesto al parser di parsare il file XML.

```
public class Ex1 {
   public static void main(String args[]) throws Exception {
      // crea un parser XML
      SAXParserFactory spf = SAXParserFactory.newInstance();
      SAXParser saxParser = spf.newSAXParser();
      XMLReader parser = saxParser.getXMLReader();
      // crea un handler
      ContentHandler handler = new MyHandler();
      // assegna l'handler al parser
      parser.setContentHandler(handler);
      // parsing del documento, verranno chiamati i callback
      parser.parse("test.xml");
   }
}
```

Gestione degli errori

Gli errori generati dal parser SAX sono incapsulati in eccezioni che non vengono sollevate direttamente ma passate ad un oggetto che implementa l'interfaccia `ErrorHandler`.

Tale oggetto deve essere registrato nel parser, se questo non viene fatto, gli errori di parsing non vengono segnalati.

Anche DOM utilizza lo stesso approccio.

Gestione degli errori

La classe `org.xml.sax.helpers.DefaultHandler` fornisce un'implementazione di default anche per `ErrorHandler` (e anche per altre interfacce di SAX). È possibile estendere questa classe per implementare il proprio ErrorHandler.

```
interface ErrorHandler {
  void error(SAXParseException exception);
  void fatalError(SAXParseException exception);
  void warning(SAXParseException exception);
}
class SAXParseException {
  int getColumnNumber();
  int getLineNumber();
  String getPublicId();
  String getSystemId();
}
```

Esempio

```
public void saxTest(String path) {
   File file = new File(path);
   SAXParserFactory factory =
     SAXParserFactory.newInstance();
   factory.setValidating(true);

   SAXParser sxp = factory.newSAXParser();
   DefaultHandler handler = new DefaultHandler();
   sxp.parse(file, handler);
}
```

Parsing SAX

- L'approccio utilizzato da SAX è quello di generare degli eventi nel momento in cui il reader incontra i tag del documento xml.
- Per intercettare questi eventi, occorre creare una classe parser che implementa alcune interfacce (`ContentHandler`, `DTDHandler`, `ErrorHandler`) o estender il `DefaultHandler`.
- In genere basta concentrarsi su questi tre metodi:
  - `startElement();`
  - `characters();`
  - `endElement();`

Parsing SAX

```
public void startElement(String uri, String localName,
    String gName, Attributes attrs)
```

Il metodo viene chiamato in automatico dal SAX parser, il quale in automatico passa i parametri:

- Nome del namespace
- Nome locale (*più importante per il parsing*)
- Nome completo dell'elemento
- Eventuali attributi

```
public void characters (char[] ch, int start, int length)
```

Vengono passati sia l'array di caratteri contenuti nell'elemento corrente che gli indici `start` e `length`.

Esercizio guidato 1 – Parsing SAX

Vediamo insieme come si può implementare, praticamente, il parsing di un file XML `employees.xml` che contiene i dati dei dipendenti di un'azienda. Il codice deve stampare il contenuto del file XML dopo averlo parsato e salvato in oggetti Java.

Trovate il codice ed i file XML su GitHub nel progetto.

Esercizio guidato 2 – Parsing e Lookup SAX

Vediamo insieme come si può implementare, praticamente, il parsing di un file XML `corso.xml` che contiene i dati dei corsi di un'università. Il codice deve stampare solamente i nomi dei docenti, senza salvarli all'interno di oggetti Java.

Esercizio – Parsing SAX (1)

Sia dato il linguaggio "libro" in un file XML contenente le seguenti informazioni:

- Un elemento radice chiamato `libro` con attributi obbligatori `titolo`, `autore`, `editore` tutti di tipo CDATA.
- L'elemento `libro` potrà avere uno o nessun elemento `prefazione`, un elemento `indice` (obbligatorio) e uno o più elementi `capitolo`.
- L'elemento `prefazione` è di tipo PCDATA (ossia contiene valori alfanumerici) ed ha un attributo `autore` non obbligatorio.
- L'elemento `indice` non ha attributi ma ha elementi figli `titolo` (uno o più).
- Gli elementi `capitolo` (figlio di `libro`) e `titolo` (figlio di `indice`) sono semplici elementi contenenti del testo.

Esercizio – Parsing SAX (2)

La definizione rigorosa del contenuto del file XML viene data attraverso il file `libro.dtd`.

```
<?xml encoding="UTF-8" ?>
<!ELEMENT libro (prefazione?, indice, capitolo+)>
<!ATTLIST libro titolo CDATA #REQUIRED>
<!ATTLIST libro autore CDATA #REQUIRED>
<!ATTLIST libro editore CDATA #REQUIRED>

<!ELEMENT prefazione (#PCDATA)>
<!ATTLIST prefazione autore CDATA #IMPLIED>
<!ELEMENT indice (titolo+)>
<!ELEMENT capitolo (#PCDATA)>
<!ELEMENT titolo (#PCDATA)>
```

Esercizio – Parsing SAX (3)

Un esempio del possibile contenuto del file XML è il seguente:

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

<libro titolo="1984" autore="G. Orwell" editore="Mondadori">
  <prefazione autore="Bill Gates">...</prefazione>
  <indice>
    <titolo>Parte prima</titolo>
    <titolo>Parte seconda</titolo>
  </indice>
  <capitolo>Era una notte buia e tempestosa...</capitolo>
  <capitolo>...</capitolo>
  <capitolo>...</capitolo>
</libro>
```

Esercizio – Parsing SAX (4)

Vogliamo realizzare un processore di documenti `libri.xml`, modificando l'esempio `sax_employees`:

- Implementare un gestore eventi generico
- Impostare il parser per la validazione del file XML rispetto al suo DTD
- Stampare a video le informazioni lette dal documento XML, gestendo gli eventi `startElement()` e `characters()`

XML – Approccio DOM

Approccio DOM – Document Object Model (1)

Permette operazioni di *load* o *save* di un documento XML da e su risorse esterne (file, stringhe, stream generici).

Contrariamente a SAX consente di creare e modificare i documenti XML:

- Creare un nuovo DOM
- Modificare la struttura di un albero DOM
- Modificare il contenuto di un albero DOM
- DOM carica tutto il contenuto del documento in memoria (attenzione allo spazio necessario per documenti XML di grandi dimensioni)

Approccio DOM (2)

DOM è un modello ad oggetti per rappresentare documenti strutturati, definisce:

- Gli oggetti usati per rappresentare e manipolare il documento
- Le interfacce usate per interagire con gli oggetti definiti
- La semantica richiesta dagli oggetti e dalle interfacce definite
- Le relazioni e le interazioni tra le interfacce e gli oggetti definiti

Approccio DOM (3)

Come da standard XML, DOM lavora rappresentando i documenti XML con una *struttura ad albero*, incapsulando ogni nodo del grafico in un oggetto specifico dotato della propria interfaccia di manipolazione.

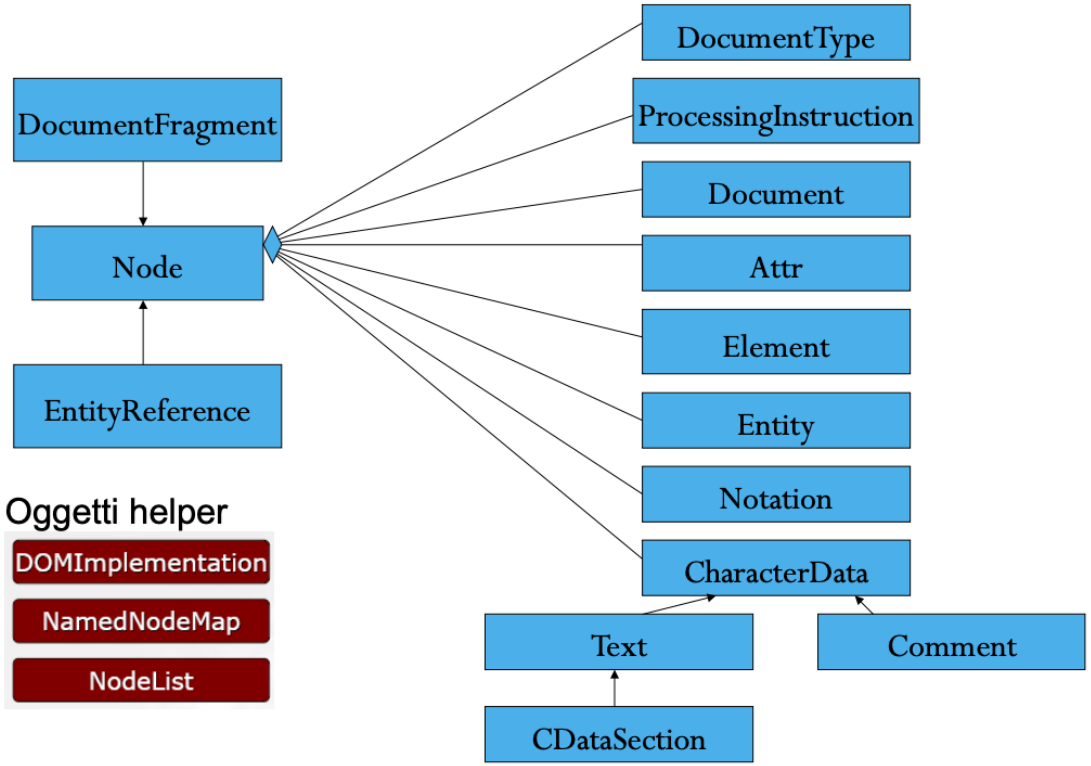
Utilizzando le API del DOM, un programmatore può creare documenti XML, navigarli e modificarne la struttura (aggiungere, modificare o cancellare nodi) in modo molto semplice in quanto DOM fornisce un'interfaccia generica.

Livelli di DOM XML

Le specifiche di DOM sono suddivise in livelli:

- *Livello 1*: oggetti DOM base con interfacce e metodi di uso più comune
- *Livello 2*: supporto ai namespace XML e clonazione di nodi
- *Livello 3*: nuovi metodi e interfacce per navigare più rapidamente il documento, supporto ai tipi di nodo e serializzazione

Gli oggetti del DOM



Gli oggetti `Node`

DOM rappresenta i documenti come una struttura gerarchica di oggetti di tipo `Node`, ciascuno dei quali può essere specializzato (*Element*, *Document*, etc...):

- I `Node` di tipo `Element` e `Document` possono avere zero o più figli
- Ogni `Node`, tranne `Document`, ha un genitore
- `Document` rappresenta l'intero documento XML

L'interfaccia di `Node` include operazioni di base eseguibili su ogni tipo di oggetto, indipendentemente dal tipo specifico. Ciascun oggetto implementa poi un'interfaccia più specifica, sempre derivata da `Node`, che definisce operazioni particolari.

Interfaccia degli oggetti Node

```
interface Node {
      const unsigned short ELEMENT_NODE = 1;    // ... altre costanti di tipo: vedi dopo
      readonly attribute DOMString nodeName;
      attribute DOMString nodeValue;
      readonly attribute unsigned short nodeType;
      readonly attribute Node parentNode;
      readonly attribute NodeList childNodes;
      readonly attribute Node firstChild;
      readonly attribute Node lastChild;
      readonly attribute Node previousSibling;
      readonly attribute Node nextSibling;
      readonly attribute NamedNodeMap attributes;
      readonly attribute Document ownerDocument;
    attribute DOMString prefix;                    // Level 2
    readonly attribute DOMString localName;              // Level 2
    attribute DOMString textContent;               // Level 3
    boolean isEqualNode(in Node other);                  // Level 3
    Node insertBefore(in Node newChild,in Node refChild) raises(DOMException);
      Node replaceChild(in Node newChild, in Node oldChild) raises(DOMException);
      Node removeChild(in Node oldChild) raises(DOMException);
      Node appendChild(in Node newChild) raises(DOMException);
      boolean hasAttributes();                           //Level 2
    boolean hasChildNodes();
      Node cloneNode(in boolean deep);
};
```

Tipi di nodo – nodeType

I tipi specifici per i nodi sono identificati dalle costanti dell'interfaccia Node (valori tra parentesi):

- `ELEMENT_NODE`: Il nodo è un elemento (1)
- `ATTRIBUTE_NODE`: Il nodo è un attributo (2)
- `TEXT_NODE`: Il nodo è del testo (3)
- `CDATA_SECTION_NODE`: Il nodo è una sezione CDATA (4)
- `ENTITY_REFERENCE_NODE`: Il nodo è un riferimento ad entità (5)
- `ENTITY_NODE`: Il nodo è un'entità (6)
- `PROCESSING_INSTRUCTION_NODE`: Il nodo è una processing instruction (7)
- `COMMENT_NODE`: Il nodo è un commento (8)
- `DOCUMENT_NODE`: Il nodo è un documento (9), ma non è la radice del documento.
- `DOCUMENT_TYPE_NODE`: Il nodo è un DOCTYPE (10)
- `DOCUMENT_FRAGMENT_NODE`: Il nodo è un frammento (11)
- `NOTATION_NODE`: Il nodo è una notation (12)

Semantica degli attributi di Node

Tipo di nodo	*nodeName*	*nodeValue*	*attributes*
Element	Nome del tag	null	*NamedNodeMap*
Attr	Nome dell'attributo	Valore dell'attributo	null
Text	"#text"	Testo associato	null
CDATASection	"#cdata-section"	Testo associato	null
EntityReference	Nome dell'entità	null	null
Entity	Nome dell'entità	null	null
ProcessingInstruction	Valore dell'attributo target	Contenuto escluso l'attributo target	null
Comment	"#comment"	Testo associato	null
Document	"#document"	null	null
DocumentType	Nome del tipo di documento	null	null
DocumentFragment	"#document-fragment"	null	null
Notation	Nome della NOTATION	null	null

Navigare l'albero con `Node`

L'interfaccia `Node` mette a disposizione diversi attributi e metodi per navigare l'albero DOM:

- `prefix` e `localname`: per leggere prefisso del namespace e nome del nodo (insieme formano il `NodeName`)
- `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`, etc...: consentono di muoversi nella struttura ad albero
- `attributes`: restituisce la lista degli attributi del nodo corrente (`NamedNodeMap`)
- `textContent`: leggere e scrivere il contenuto testuale del nodo (concatenazione dei nodi `Text` in esso nidificati)
- `isEqualNode`: verifica se due nodi sono uguali testando ricorsivamente nome, namespace, attributi e contenuto

Modificare l'albero con `Node`

I metodi di Node per la manipolazione dei nodi fili sono:

- `appendChild`
- `removeChild`
- `replaceChild`
- `insertBefore`
- `renameNode` (dell'ownerDocument)
- Etc...

L'effettiva possibilità di applicare tali metodi dipende dal tipo di nodo, ad esempio `appendChild` ad un nodo `Text`. Nel caso l'operazione non sia disponibile, viene generata un'eccezione di tipo `DOMException`.

Interfaccia Document

```
interface Document : Node {
        readonly attribute DocumentType doctype;
        readonly attribute DOMImplementation implementation;
        readonly attribute Element documentElement;
        Element createElement(in DOMString tagName) raises(DOMException);
        DocumentFragment createDocumentFragment();
        Text createTextNode(in DOMString data);
        Comment createComment(in DOMString data);
        CDATASection createCDATASection(in DOMString data) raises(DOMException);
        ProcessingInstruction createProcessingInstruction(in DOMString target,
                                in DOMString data) raises(DOMException);
        Attr createAttribute(in DOMString name) raises(DOMException);
        EntityReference createEntityReference(in DOMString name) raises(DOMException);
        NodeList getElementsByTagName(in DOMString tagname);
};
```

`documentElement` rappresenta l'elemento radice del documento. Un oggetto `Document`, inoltre, fornisce i metodi `createX()` per creare i nodi del documento.

Interfaccia `Element`

```
interface Element : Node {
        readonly attribute DOMString tagName;
        DOMString getAttribute(in DOMString name);
        void setAttribute(in DOMString name, in DOMString value) raises(DOMException);
        void removeAttribute(in DOMString name) raises(DOMException);
        Attr getAttributeNode(in DOMString name);
        Attr setAttributeNode(in Attr newAttr) raises(DOMException);
        Attr removeAttributeNode(in Attr oldAttr) raises(DOMException);
        NodeList getElementsByTagName(in DOMString name);
        void normalize();
};
```

- `Element` dispone di attributi e metodi per manipolare gli attributi (trattati come stringhe o oggetti `Attr`)
- L'attributo `tagName` restituisce il nome del tag
- Il metodo `getElementsByTagName` restituisce i soli figli del nodo che siano elementi con uno specifico nome (filtra i `childNodes`)
- `normalize` fonde nodi `Text` adiacenti nel sottoalbero dell'elemento

Interfaccia `Attr`

```
interface Attr : Node {
      readonly attribute DOMString name;
      readonly attribute boolean specified;
      attribute DOMString value;
};
```

- Gli `Attr` sono semplici oggetti attributi con nome e valore.
- I valori name e value sono accessibili anche con i metodi `getName` e `getValue` dell'interfaccia `Node`.
- L'attributo `specified` è `false` se l'attributo non era presente nel documento, ma è stato inserito dal parser con il suo valore di default specificato nel DTD associato al documento stesso. In caso contrario, l'attributo è `true`.
- I metodi ereditati da `Node` per la manipolazione di nodi (e.g. `appendChild`) non sono validi per `Attr`! (Generano eccezioni)

Interfacce `NodeList` e `NamedNodeMap`

```
interface NodeList {
        Node item(in unsigned long index);
        readonly attribute unsigned long length;
};
```

```
interface NamedNodeMap {
        Node getNamedItem(in DOMString name);
        Node setNamedItem(in Node arg, in DOMString value) raises(DOMException);
        Node removeNamedItem(in DOMString name) raises(DOMException);
        Node item(in unsigned long index);
        readonly attribute unsigned long length;
};
```

- `NodeList` gestisce una lista ordinata di nodi. Con i metodi `length()` e `item(i)` si ottengono la lunghezza della lista e l'i-esimo elemento.
- La `NamedNodeMap` contiene elementi accessibili sia per indice (`item(i)`) che attraverso il nome (attributo `nodeName` di `Node`) (si tratta di una tabella hash).

DOM in Java: JAXP

JAXP (Java API for XML Processing) è un'interfaccia astratta che deve sfruttare un'implementazione concreta di DOM fornita da terze parti.

Il parser per JAXP più conosciuto, più completo e probabilmente migliore è Xerces di Apache (pacchetto Xerces 2 dell'Apache XML Project) `http://xerces.apache.org/xerces2-j`.

Supporta DOM level 3 dalla versione 2.9.0, ed è già integrato in Eclipse da 3.0 in poi.

I/O di XML con DOM

Per "scrivere" il contenuto di un DOM (rappresentazione serializzata del documento su file, stringa o stream) si ricorre al metodo `save` definito da DOM level 3.

Per leggere il contenuto di un DOM:

- Si usa il metodo `load` di DOM level 3
- Si utilizza l'approccio tradizionale del `DocumentBuilder`

DOM – Creazione di un documento

```
import org.w3c.dom.Document;  //Elementi DOM
import org.w3c.dom.Element;   //Elementi DOM
import javax.xml.parsers.*;   //Elementi JAXP
public Document newDocument(){ //Crea un documento vuoto
   DocumentBuilderFactory factory =
               DocumentBuilderFactory.newInstance();
   factory.setValidating(true);
   factory.setNamespaceAware(true);
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.newDocument();
        return doc;
    } catch (javax.xml.parsers.ParserConfigurationException) { }
        return null;
}
```

Supponiamo di avere un documento inizialmente vuoto. Con i metodi `createX()` dell'oggetto `Document` è possibile creare sia la struttura che il contenuto del documento.

DOM – Parsing

```
import org.w3c.dom.Document;      //Elementi DOM
import javax.xml.parsers.*;       //Elementi JAXP
import  java.io.*;                //Java i/o
public Document loadDocument(String path){
   DocumentBuilderFactory factory =
           DocumentBuilderFactory.newInstance();
   factory.setValidating(true);
   factory.setNamespaceAware(true);
   try {
          DocumentBuilder builder = factory.newDocumentBuilder();
         Document doc = builder.parse(new File(path));
         return doc;
   }
   catch (javax.xml.parsers.ParserConfigurationException) {}
   catch (java.io.IOException ioe) {}
   catch (org.xml.sax.SAXParseException spe) {}
   catch (org.xml.sax.SAXException se) {}
   return null;
}
```

In questo caso il codice è molto simile al caso precedente ma si invoca il metodo `parse` su una risposta esterna (in questo caso un file).

Visita in pre-order dell'albero DOM (1)

Una volta caricato in memoria il DOM, esso può essere attraversato con una banale visita per alberi:

- Pre-order
- In-order
- Post-order

Nel caso di una visita pre-order:

- Visita la root
- Visita ricorsivamente ogni sotto-albero in pre-order

Visita in pre-order dell'albero DOM (1)

```
static void printPreorder(String indent, Node node) {
        printNode(indent, node);
        if(node.hasChildNodes()) {
            Node child = node.getFirstChild();
            while (child != null) {
                printPreorder(indent + "  ", child);
                child = child.getNextSibling();
            }
        }
    }

static void printNode(String indent, Node node) {
        System.out.print(indent);
        System.out.print(node.getNodeType() + " ");
        System.out.print(node.getNodeName() + " ");
        System.out.print(node.getNodeValue() + " ");
        System.out.println(node.getAttributes());
    }
```

Serializzare il DOM – save (1)

Le classi JAXP per serializzare si trovano nel namespace `org.w3c.dom.ls`.

Per serializzare un documento XML:

- Si ottiene un'istanza di `DOMImplementationLS` attraverso il metodo `getImplementation` del documento.
- Si crea un oggetto `LSOutput` (stream di output astratto) attraverso `DOMImplementationLS` e si imposta lo stream di output su cui scrivere.
- Si crea un oggetto `LSSerializer` attraverso `DOMImplementationLS`.
- Si fa serializzare il documento (o un nodo qualsiasi del documento) su `LSOutput` invocando il metodo `write` di `LSSerializer`.
- `writeToString` per serializzare su una stringa.

Serializzare il DOM – save (2)

```
import org.w3c.dom.*;
import org.w3c.dom.ls.*;
import  java.io.*;
public void saveDocument(Document doc, Writer w){
  DOMImplementationLS ls = (DOMImplementationLS)
                  doc.getImplementation(); //Nota il cast!
  LSOutput o = ls.createLSOutput();
  LSSerializer s = ls.createLSSerializer();
  try {
    o.setCharacterStream(w); //possibile connettere anche stream binari
                             //con setByteStream
   s.write(doc,o);
  }
  catch (IOException ioe) { ioe.printStackTrace();}
  catch (LSException lse) { System.err.println(lse.getMessage());}
}
//Esempio: stampa a video del DOM XML
public void printDocument(Document doc){
    saveDocument(doc, new PrintWriter(System.out));
}
```

Esercizio Guidato 1 – Parsing DOM

Vediamo insieme come si può implementare, la creazione e la ricerca all'interno di un file XML utilizzando DOM.

JSON

JSON

JavaScript Object Notation (JSON) è un formato di scambio dati progettato per essere il più leggero possibile.

JSON ha i seguenti valori:

- `null`
- Stringhe
- Numeri (IEEE754)
- `true` e `false`
- *Oggetti*, ovvero collezioni di coppia chiave e valore JSON
- *Array*, ovvero liste ordinate di valori JSON

JSON	XML equivalente
`{ "company": Volkswagen, "name": "Vento", "price": 800000 }`	`<car> <company>Volkswagen</company> <name>Vento</name> <price>800000</price> </car>`

XML vs JSON

XML	JSON
Più ridondante (*verbose*)	Meno ridondante (*verbose*) -> più veloce da scrivere
Usato per descrivere dati strutturati, che non includono array.	Include la possibilità di definire array
Parsing *relativamente* pesante e complesso	Parsing più leggero. Anche *eval()* in Javascript supporta JSON
Supporta DTD, linguaggio auto-descrivente. Ideale in contesti applicativi specifici (es. trasmissione dei dati fra pazienti fra ospedali, WS SOAP con imbustamento e crittografia)	Scopo *limitato.* Non c'è uno standard per la auto-documentazione, da realizzare a parte. Formato diffuso per open-data, WS REST

JSON e Java

Una delle librerie più usate è `org.json.*`, scaricabile da `http://mvnrepository.com/artifact/org.json/json`.

Per rappresentare gli oggetti e gli array si usano rispettivamente:

- `org.json.JSONObject`
- `org.json.JSONArray`
- Per il parsing da `String` a `JSONObject` si usa il costruttore `JSONObject(String s)`
- Il metodo `toString()` di un oggetto `JSONObject` consente di ottenerne la sua rappresentazione come stringa di caratteri.