XML e Java

Elaborazione di documenti XML

- Un elemento importante che ha consentito la diffusione dei linguaggi e delle tecnologie XML è il supporto fornito da strumenti per il parsing (analisi sintattica)
- Ogni applicazione che vuole fruire di XML deve includere un supporto per il parsing
- I parser XML sono diventati strumenti standard nello sviluppo delle applicazioni
- Per esempio JDK1.4 integra un supporto al parsing (specifiche JAXP), e come default dornisce un parser (Apache Crimson)

Compiti del parser

- Decomporre i documenti XML (istanza) nei loro elementi costitutivi
  - elementi, attributi, testo, processing instructions, entità, …
- Eseguire controlli sul documento
  - Controllare che il documento sia ben formato
  - Controllare **eventualmente** che il documento sia valido (DTD, XML Schema)
- Non tutti i parser consentono la validazione dei documenti: si parla quindi di **parser validanti** e **non validanti**
- La validazione è una operazione computazionalmente costosa e quindi è opportuno potere scegliere se attivarla o meno

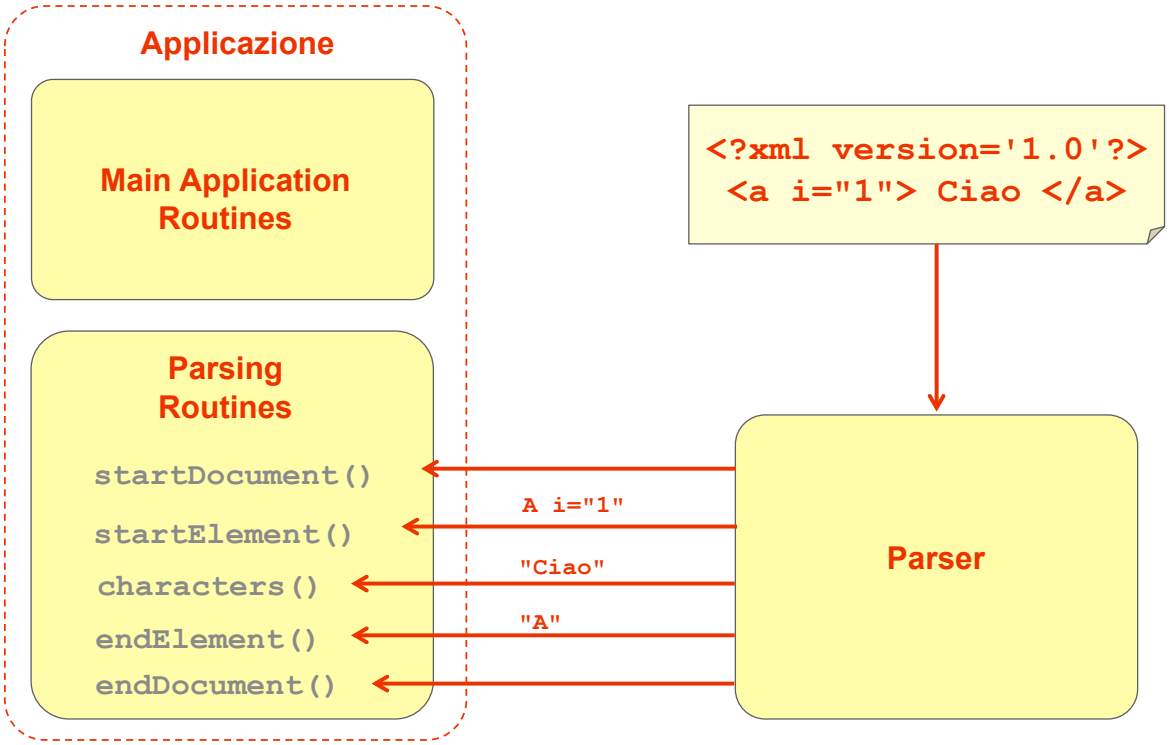
Modelli di parsing per XML

- Dal punto di vista dell'interfacciamento tra applicazioni e parser esistono due grandi categorie di API
- **Interfacce basate su eventi**
  - **Interfacce ESIS** (Element Structure Information Set): interfacce dei parser per documenti SGML
  - **Interfacce SAX** (Simple API for XML): sfruttano un modello a call-back
- **Interfacce Object Model**
  - W3C DOM Recommendation

Interfacce ad eventi

- L'applicazione implementa un insieme di metodi di callback che vengono invocati dal parser mentre elabora il documento
  - Le callback sono invocate al verificarsi di specifici eventi es. start-tag, end-tag,…
- I parametri passati al metodo di callback forniscono ulteriori informazioni sull'evento
  - Nome dell'elemento
  - Nome e valore assunto dagli attributi
  - Valore delle stringhe contenute …
- E' una modalità semplice ed efficiente
- E il modello adottato da SAX (Simple API for XML)
  - è uno standard de facto per il parsing di documenti XML

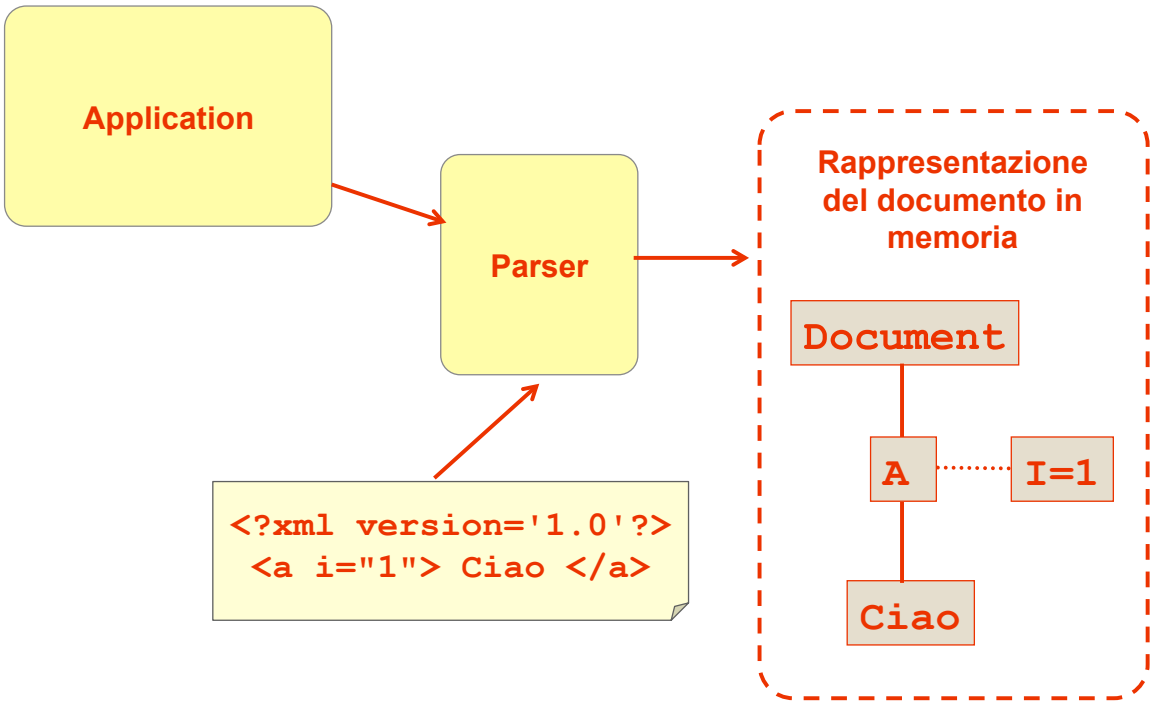
Schema di parsing ad eventi



Interfacce Object Model

- L'applicazione interagisce con una rappresentazione object-oriented del documento
- Il documento è rappresentato da un albero (parse tree) costituito da vari oggetti quali document, element, attribute, text, …
- Il livello di astrazione che le interfacce Obiect Model forniscono al programmatore è maggiore rispetto a quello fornito dalle interfacce basate su modelli basati su eventi
  - Facile accedere ai figli di un elemento, ai fratelli, …
  - Problema: Richiede la disponibilità di maggiori risorse computazionali (particolare memoria perché l'intero documento viene caricato in memoria)

Schema di parsing object model



Ricapitolando

- I parser XML rendono disponibile alle applicazioni la struttura ed il contenuto dei documenti XML e si interfacciano mediante due tipologie di API
- **Event-based API (Es. API SAX)**
  - Notificano alle applicazioni eventi generati nel parsing dei documenti
  - Usano poche risorse ma non sono sempre comodissime da usare
- **Object-model based APIs (es. DOM)**
  - Forniscono accesso al parse tree che rappresenta il documento XML Molto comode ed eleganti r
  - Richiedono però maggiori risorse, soprattutto in termini di memoria
- I parser più diffusi supportano sia SAX sia DOM: spesso i parser DOM sono sviluppati su parser SAX

Parser in Java: JAXP

- JAXP è un framework che ci consente di istanziare ed utilizzare parser XML (sia SAX che DOM) nelle applicazioni Java
- JAXP 1.1 è Inclusa nella JDK a partire dalla vers. 1.4
- Fornisce vari package:
  - `org.xml.sax`: interfacce SAX 2.0
  - `org.w3c.dom`: interfacce DOM Level 2
  - `javax.xml.parsers`: inizializzazione ed uso dei parser
- JAXP permette di operare indipendentemente dall'effettiva implementazione del parser utilizzata
- Utilizza un parser presente all'interno del sistema
- Usa il pattern **factory method** per ottenere questa indipendenza

JAXP: factory e parser

- `javax.xml.parsers` espone due classi factory, una per SAX e una per DOM:

  `SAXParserFactory`
  `DocumentBuilderFactory`

- Sono classi astratte ed espongono il metodo statico `newInstance()` che consente di ottenere un'istanza di una classe discendente concreta:

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

- Una volta ottenuta una factory possiamo invocarla per creare un parser con interfaccia SAX o DOM:

```
SAXParser saxParser = spf.newSAXParser();
DocumentBuilder builder = dbf.newDocumentBuilder();
```


SAX Event Callback API

- SAX stabilisce un insieme di API molto diffuso
- È uno standard de-facto
  - SAX 1.0 Maggio 1998,
  - SAX 2.0 Maggio 2000
- **SAX non è un parser, ma è una interfaccia**
- L'interfaccia SAX è supportata da molti parser:
  - Sun JAXP
  - IBM XML4J,
  - Oracle XML Parser for Java
  - Apache Xerces
  - Microsoft MSXML (in IE 5)
  - …

Interfacce SAX

- Le applicazioni interagiscono con parser conformi a SAX utilizzando interfacce stabilite
- Le interfacce di SAX possono essere divise in 3 categorie:
  - **Interfacce Parser-to-Application (call-back):** consentono di definire funzioni di call-back e di associarle agli eventi generati dal parse mentre questi processa un documento XML
  - **Interfacce Application-to-Parser:** consentono di gestire il parser
  - **Interfacce Ausiliarie:** facilitano la manipolazione delle informazioni fornite dal parser

Interfacce Parser to Application (call-back)

- Vengono implementate dall'applicazione per imporre un preciso comportamento a seguito del verificarsi di un evento:
- Sono quattro:
  - `ContentHandler:` metodi per elaborare gli eventi generati dal parser
  - `DTDHandler:` metodi per ricevere notifiche su entità esterne al documento e loro notazione dichiarata in DTD
  - `ErrorHandler:` metodi per gestire gli errori ed i warning nell'elaborazione di un documento
  - `EntityResolver:` metodi per personalizzare l'elaborazione di riferimenti ad entità esterne
- Se un'interfaccia non viene implementata il comportamento di default è ignorare l'evento.

Interfacce Application-to-Parser

- Sono implementate dal parser
  - `XMLReader:` interfaccia che consente all'applicazione di invocare il parser e di registrare gli oggetti che implementano le interfacce di call-back
  - `XMLFilter:` interfaccia che consente di porre in sequenza vari XMLReaders come una serie di filtri

Interfacce Ausiliarie

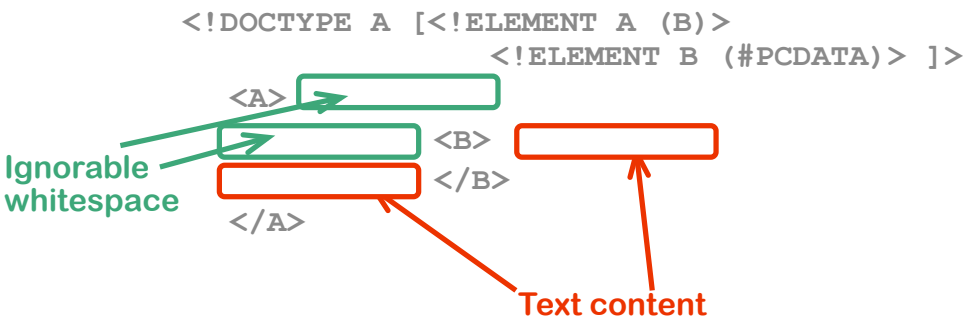
- `Attributes:` Metodi per accedere ad una lista di attributi
- `Locator:` metodi per individuare l'origine degli eventi nel parsing dei documenti (es. systemID, numeri di linea e di colonna…)

ContentHandler

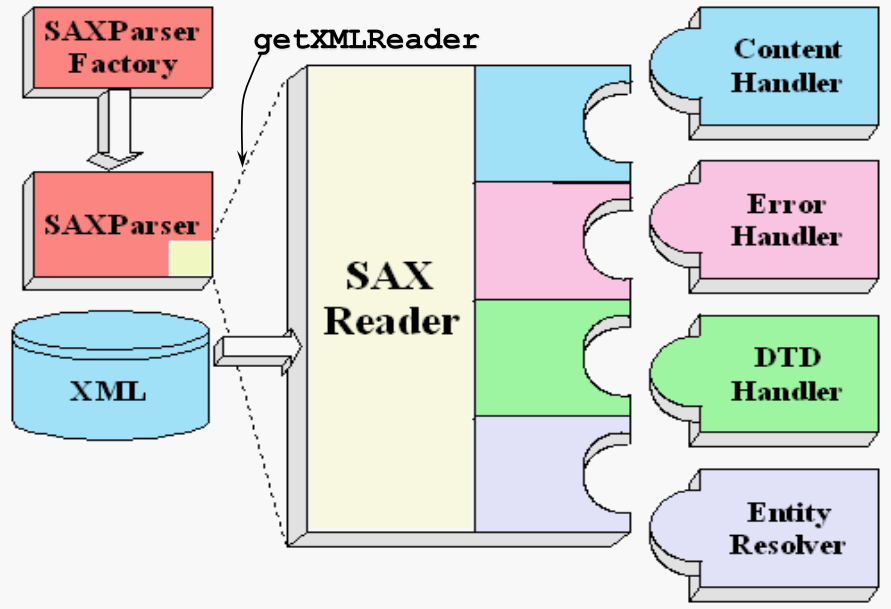
- Metodi per ricevere informazioni sugli eventi generali che si manifestano nel parsing di un documento XML.
  - `setDocumentLocator(Locator locator)`: Riceve un oggetto che determina l'origine di un evento SAX. Per esempio potremmo usare il metodo per fornire informazioni agli utenti.
- Inizio e fine documento:
  - `startDocument(); endDocument()`
- Inizio e fine di un elemento:
  - `startElement(String namespaceURI, String localname, String qName, Attributes atts);`
  - `endElement(String namespaceURI, String localname, String qName):`
- Si noti che i due ultimi metodi supportano i namespaces

Altri metodi di ContentHandler

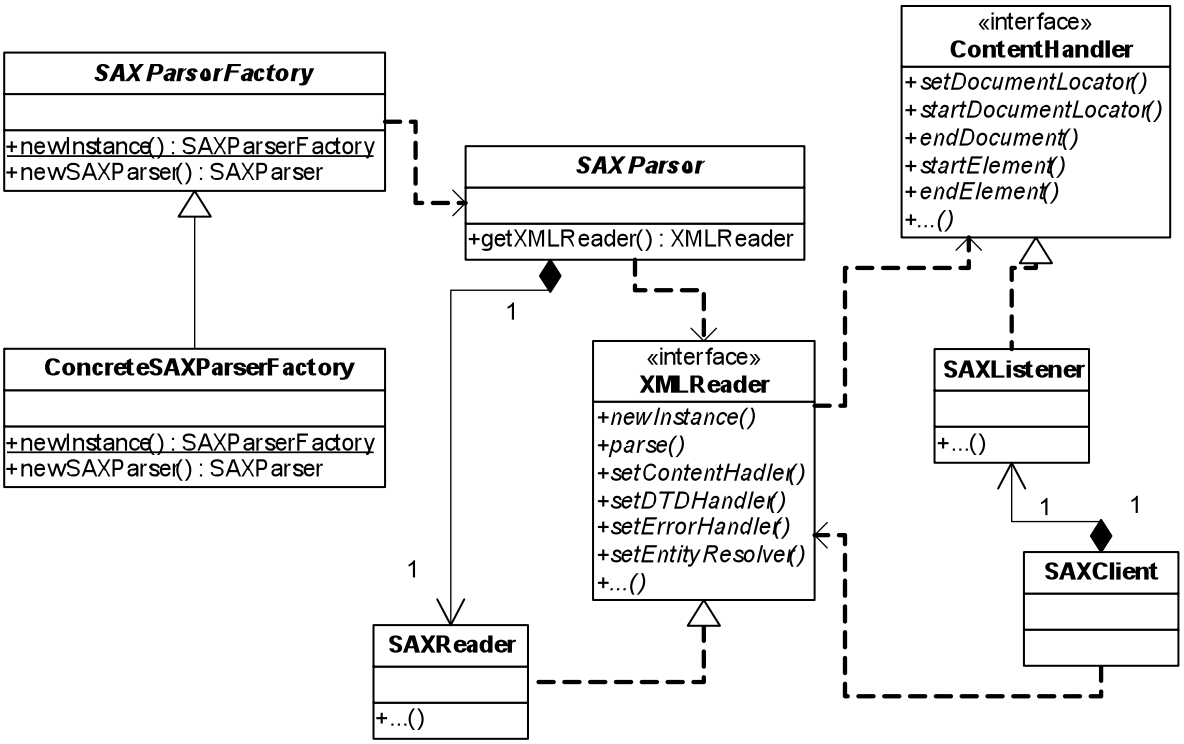
- `characters(char ch[], int start, int length)`
  - Notifica la presenza di character data.
- `ignorableWhitespace(char ch[], int start, int length)`
  - Notifica della presenza di whitespace ignorabili nell'element content.



Schema sintetico di un parser SAX



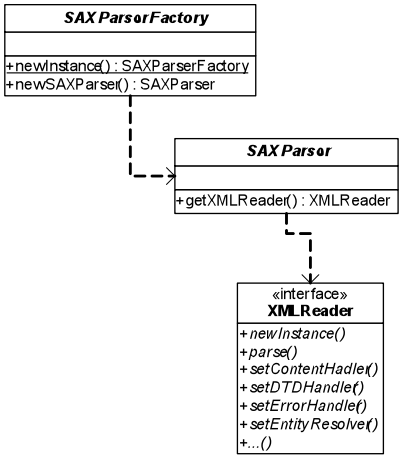
Schema di funzionamento di SAX in UML



Attivazione di un parser SAX

- Si crea prima una factory, poi il parser e infine il reader

```
SAXParserFactory spf =
  SAXParserFactory.newInstance();
try
{
  SAXParser saxParser =
    spf.newSAXParser();
  XMLReader xmlReader =
    saxParser.getXMLReader();
}
catch (Exception e)
{
  System.err.println(e.getMessage());
  System.exit(1);
};
```



SAX e i namespace: esempio 1

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match="/">
  <html>
      <xsl:value-of select="//total"/>
  </html>
</xsl:template>
</xsl:stylesheet>
```

- Una invocazione del metodo di SAX `startElement` per l'elemento `xsl:template` passerebbe i seguenti parametri:

  `namespaceURI="http://www.w3.org/1999/XSL/Transform"`

  `localname=template`

  `qName=xsl:template`

SAX e i namespace: esempio 2

```
<xsl:stylesheet version="1.0" ...
  xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match="/">
  <html> ... </html>
</xsl:template>
</xsl:stylesheet>
```

- Una invocazione del metodo di SAX `startElement` per l'elemento `html` passerebbe i seguenti parametri:

  `namespaceURI= http://www.w3.org/TR/xhtml1/strict`

  `localname = html,`

  `qName = html`

- Infatti `html` è senza prefisso e quindi appartiene al namespace di default

Esempio SAX - 1

- Consideriamo il seguente file XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<db>
  <person idnum="1234">
    <last>Rossi</last><first>Mario</first>
  </person>
  <person idnum="5678">
    <last>Bianchi</last><first>Elena</first>
  </person>
  <person idnum="9012">
    <last>Verdi</last><first>Giuseppe</first>
  </person>
  <person idnum="3456">
    <last>Rossi</last><first>Anna</first>
  </person>
</db>
```

Esempio SAX - 2

- Stampare a video nome, cognome e identificativo di ogni persona:

```
Mario Rossi (1234)
Elena Bianchi (5678)
Giuseppe Verdi (9012)
Anna Rossi (3456)
```

- Strategia di soluzione con un approccio event-based:
  - All'inizio di `person`, si registra `idnum` (p.es. 1234)
  - Si tiene traccia dell'inizio degli elementi `last` e `first`, per capire quando registrare nome e cognome (e.g., "Rossi" and "Mario")
  - Alla fine di `person`, si stampano i dati memorizzati

Esempio SAX - 3

- Incominciamo importando le classi che ci servono:

```
import org.xml.sax.XMLReader;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;

// Implementazione di default di ContentHandler:
import org.xml.sax.helpers.DefaultHandler;

// Classi JAXP usate per accedere al parser SAX:
import javax.xml.parsers.*;
```

Esempio SAX - 4

- Definiamo una classe che discende da `DefaultHandler` e ridefinisce solo i metodi di callback che sono rilevanti nella nostra applicazione:

```
public class SAXDBApp extends DefaultHandler
{
  // Flag per ricordare dove siamo:
  private boolean InFirst = false;
  private boolean InLast = false;
  // Attributi per i valori da visualizzare
  private String FirstName, LastName, IdNum;
```


Esempio SAX - 5

- Implementiamo i metodi di callback che ci servono
- Start element registra l'inizio degli elementi `first` e `last`, ed il valore dell'attributo `idnum` dell'elemento `person`:

```
public void startElement (String namespaceURI,
String localName, String rawName, Attributes atts)
{
  if (localName.equals("first"))
    InFirst = true;
  if (localName.equals("last"))
    InLast = true;
  if (localName.equals("person"))
    IdNum = atts.getValue("idnum");
}
```

Esempio SAX - 6

- Il metodo di callback `characters` intercetta il contenuto testuale
- Registriamo in modo opportuno il valore del testo a seconda che siamo dentro `first` o `last`:

```
public void characters (char ch[], int start,
  int length)
{
  if (InFirst) FirstName =
    new String(ch, start, length);
  if (InLast) LastName =
    new String(ch, start, length);
 }
```


Esempio SAX - 7

- Quanto abbiamo trovato la fine dell'elemento `person`, scriviamo i dati. Quando troviamo la fine dell'elemento `first` o dell'elemento `last` aggiorniamo i flag in modo opportuno

```
public void endElement(String namespaceURI, String
  localName, String qName)
{
  if (localName.equals("person"))
    System.out.println(FirstName + " " +
      LastName + " (" + IdNum + ")" );
  //Aggiorna i flag di contesto
  if (localName.equals("first"))
    InFirst = false;
  if (localName.equals("last"))
    InFirst = false;
}
```

Esempio SAX – Il main

```
public static void main (String args[]) throws Exception
{
  // Usa SAXParserFactory per istanziare un XMLReader
  SAXParserFactory spf =  SAXParserFactory.newInstance();
  try
  {
    SAXParser saxParser = spf.newSAXParser();
    XMLReader xmlReader = saxParser.getXMLReader();
    ContentHandler handler = new SAXDBApp();
    xmlReader.setContentHandler(handler);
    for (int i=0; i<args.length; i++)
      xmlReader.parse(args[i]);
  }
  catch (Exception e)
  {
    System.err.println(e.getMessage());
    System.exit(1);
  };
}
```


DOM

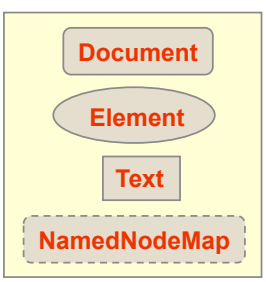
- **DOM** (**D**ocument **O**bject **M**odel) è una API per accedere e manipolare documenti XML (e HTML)
- Consente alle applicazioni di costruire documenti, navigare la loro struttura, aggiungere, modificare o cancellare elementi e contenuto
- DOM è object-based
- DOM è uno standard W3C supportato da numerosi linguaggi che utilizzano le stesse API
- Mentre SAX fornisce un meccanismo sequenziale per il parsing dei documenti DOM permette un accesso diretto senza imporre una particolare sequenza

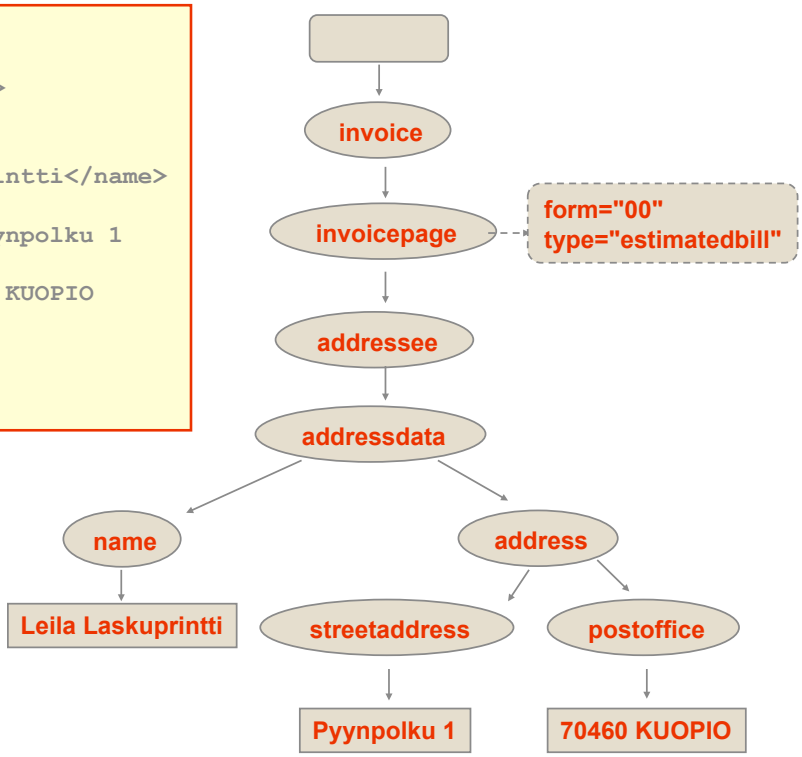
Modello di DOM

- DOM è basato sui concetti tradizionali della programmazione ad oggetti:
  - Metodi: regolano l'accesso e consentono di alterare lo stato degli oggetti
  - Interfacce: consentono di dichiarare un insieme di metodi
  - Oggetti: incapsulano dati e metodi
- Il modello rappresenta i documenti XML come alberi (parse tree)
- La struttura ad albero è determinata dalle interfacce stabilite dalla specifica di DOM.
- Tipicamente la struttura ad albero viene mantenuta nell'implementazione dei parser DOM.

Esempio

```
<invoice>
 <invoicepage form="00"
    type="estimatedbill">
  <addressee>
   <addressdata>
     <name>Leila Laskuprintti</name>
     <address>
       <streetaddress>Pyynpolku 1
       </streetaddress>
       <postoffice>70460 KUOPIO
       </postoffice>
     </address>
   </addressdata>
  </addressee> ...
```





DOM Level 1

- DOM Level 1 fornisce il modello ed i meccanismi di base per rappresentare e manipolare la struttura e il contenuto di un documento
  - Non fornisce però accesso ai contenuti del DTD
- **DOM Core Interface**
  - Fornisce le interfacce di base per accedere ai documenti strutturati
  - Fornisce inoltre le **extended interfaces**, specifiche di XML, per gestire CDATASection, DocumentType, Notation, Entity, EntityReference, ProcessingInstruction
- **DOM HTML Interfaces**
  - Forniscono supporto per accedere ai documenti HTML

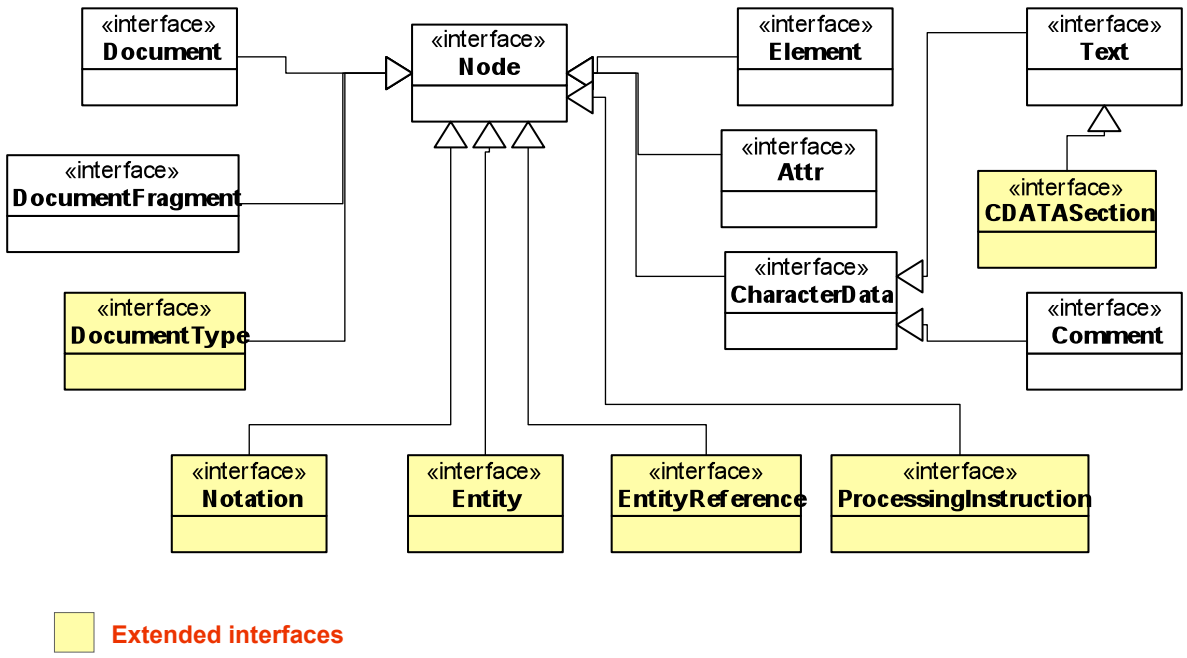
DOM Level 2

- DOM Level 2 aggiunge
  - Supporto per i **namespaces**
  - Accesso agli elementi tramite il valore degli attributi ID
- Funzionalità opzionali:
  - Interfacce per **document views** e **style sheets**
  - **Modello ad eventi** (per gestire ad esempio le azioni degli utenti sugli elementi)
  - Metodi per attraversare il documento e per manipolarne intere regioni (es. regioni selezionate dall'utente tramite un editor)
- Il caricamento e la scrittura dei documenti, benché spesso supportati, non sono specificati in DOM Level 2
  - Saranno inseriti in DOM Level 3

DOM Language Bindings

- DOM non dipende da un linguaggio specifico (language-independence):
- Le interfacce DOM sono definite sulla base della Interface Definition Language (OMG IDL; definito nelle specifiche di Corba)
- Language bindings (implementazioni delle interfacce di DOM) per tutti i principali linguaggi
  - Java
  - C++
  - JavaScript
- Nelle specifiche di DOM sono definiti i language bindings per
  - Java
  - JavaScript

Core interface: la tassonomia di Node



Interfaccia Node: metodi

- `getNodeType()`
- `getNodeValue()`
- `getOwnerDocument()`
- `getParentNode()`
- `hasChildNodes()`
- `getChildNodes()`
- `getFirstChild()`
- `getLastChild()`
- `getPreviousSibling()`
- `getNextSibling()`
- `hasAttributes()`
- `getAttributes()`
- `appendChild(newChild)`
- `insertBefore(newChild,refChild)`
- `replaceChild(newChild,oldChild)`
- `removeChild(oldChild)`

Object Creation in DOM

- DOM specifica diverse classi
- L'oggetto X di DOM è inserito nel contesto di un Document:

  `X.getOwnerDocument()`
- Gli oggetti che implementano l'interfaccia X sono creati da una factory

  `D.createX(…)` ,
  dove D è l'oggetto `Document`.
- Esempi:

  `createElement("A")`,
  `createAttribute("href")`,
  `createTextNode("Hello!")`
- Creazione e salvataggio di `Document` sono demandati alle specifiche implementazioni

Interfacce Document ed Element

- **Document**
  - `getDocumentElement()`
  - `createAttribute(name)`
  - `createElement(tagName)`
  - `createTextNode(data)`
  - `getDocType()`
  - `getElementById(IdVal)`
- **Element**
  - `getTagName()`
  - `getAttributeNode(name)`
  - `setAttributeNode(attr)`
  - `removeAttribute(name)`
  - `getElementsByTagName(name)`
  - `hasAttribute(name)`

Proprietà di Node

- **`Node.getNodeName()`**
  - Se applicato ad `Element` equivale a getTagName()
  - Se applicato ad `Attr` restituisce il nome dell'attributo
- **`Node.getNodeValue()`**
  - Restituisce il contenuto di un text node, il valore di un attributo…;
  - Se applicato ad `Element` restituisce null
- **`Node.getNodeType()`:**
  - Restituisce un valore numerico costante (es. 1, 2, 3, …, 12) che corrisponde a ELEMENT_NODE, ATTRIBUTE_NODE,TEXT_NODE, …, NOTATION_NODE

Manipolazione di Content ed Element

- Manipolazione di `CharacterData`:
  - `substringData(offset, count)`
  - `appendData(string)`
  - `insertData(offset, string)`
  - `deleteData(offset, count)`
  - `replaceData(offset, count, string)`
    `(= delete + insert)`
- Accesso e manipolazione degli attributi di un `Element`:
  - `getAttribute(name)`
  - `setAttribute(name, value)`
  - `removeAttribute(name)`

Interfacce NodeList e NamedNodeMap

- `NodeList` rappresenta liste ordinate di nodi:
  - Per esempio tutti i nodi figli di un nodo:
    `Node.getChildNodes()`
  - Oppure tutti i discendenti di tipo "name" nell'ordine in cui appaiono nel documento.
    `Element.getElementsByTagName("name")`
  - Si può usare la wild-card "*" per ottenere tutti gli elementi
- Esempio: iteriamo su tutti i nodi figli di un nodo:

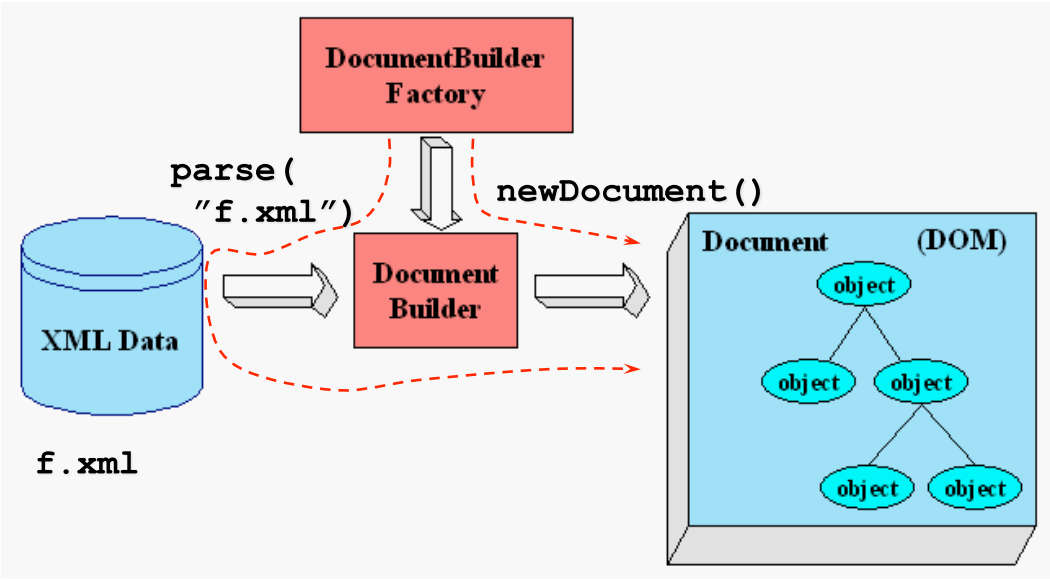
```
for (i=0;i<node.getChildNodes().getLength();i++)
  process(node.getChildNodes().item(i));
```

- `NamedNodeMap` permette di accedere a insiemi non ordinati di nodi individuati per nome:
  - Per esempio tutti gli attributi di un nodo:
    `Node.getAttributes()`

Implementazioni di DOM

- Parser Java
  - IBM XML4J
  - Apache Xerces
  - Apache Crimson
  - SUN JAXP
  - …
- MS IE5 browser:
  - interfacce per la programmazione COM per C/C++ e Visual Basic
  - oggetti ActiveX per i linguaggi di script
- XML::DOM: implementazione Perl di DOM Level 1

JAXP: Usare un Parser DOM



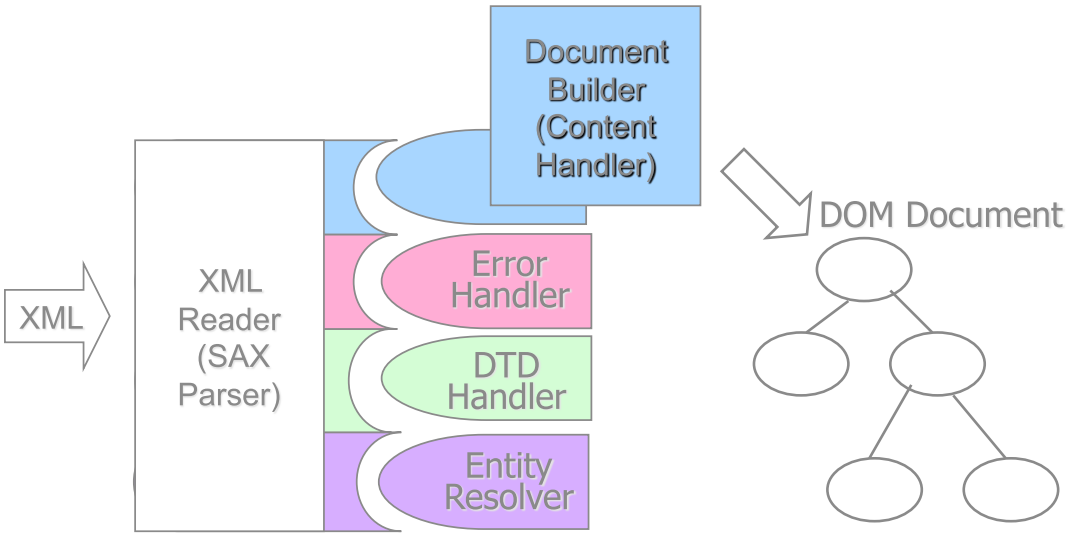
Attivazione e uso di un parser DOM in JAXP

- Si crea prima una factory, poi il document builder
- Si cdrea un oggetto di tipo File passando il nome del file che contiene il documento
- Si invoca il metodo parse del parser per ottenere un istanza di Document che può essere navigata

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

documentBuilder builder =
    dbf.newDocumentBuilder();

File file = new File ("prova.xml");

Document doc = dbf.parse(file)
```


DOM e SAX

- DOM è implementato usando SAX
- Il parser DOM mplementa l'interfaccia ContentHandler
- Ad ogni evento costruisce un pezzo del modello DOM



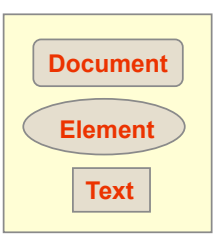
Gestione del Parsing

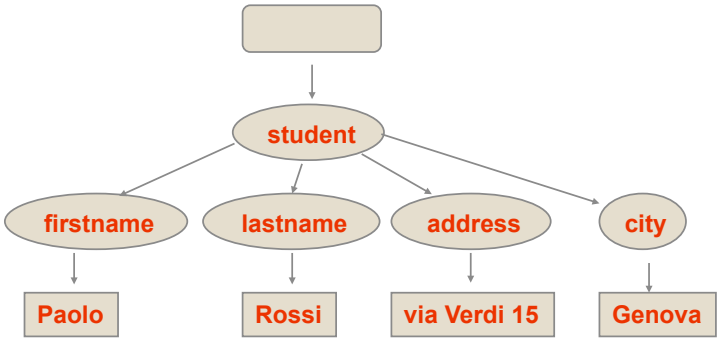
- JAXP Consente la gestione degli errori nel parsing di documenti DOM
- Si crea una classe che implementa l'interfaccia ErrorHandler di SAX (metodi error, fatalError e warning).
- Un'istanza di questa classe viene passata al parser DOM usando il metodo setErrorHandler esposto da DocumentBuilder
- La Validazione ed i namespace vengono gestiti sia per `SAXParserFactories` e per `DocumentBuilderFactories` con
  - `setValidating(boolean)`
  - `setNamespaceAware(boolean)`

Esempio DOM – Il documento XML

- Consideriamo un semplice documento XML:

```
<?xml version="1.0"?>
<student>
  <firstname>Paolo</firstname>
  <lastname>Rossi</lastname>
  <address>via Verdi 15</address>
  <city>Genova</city>
</student>
```





Esempio DOM – Caricamento del documento

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class AccessingXmlFile
{
  public static Document loadDocument(String fileName)
  {
    try
    {
      File file = new File(fileName);
      DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();
      return db.parse(file);
    }
    catch (Exception e)
      e.printStackTrace();
  }
  ...
```


Esempio DOM – Navigazione nel documento

```
  ...
  public static void main(String argv[])
  {
    Document document = loadDocument();
    Element root = document.getDocumentElement();
    root.normalize();
    System.out.println("Root " + root.getNodeName());
    System.out.println("Informazioni sugli studenti");
    NodeList nodes = root.getChildNodes();
    for (int i = 0; i < nodes.getLength(); i++)
    {
      Node el = nodes.items(i);
      String elName= el.getName()
      Node tx = el.getFirstChild();
      String elText=tx.getNodeValue();
      System.out.println(elname+"="”+elText);
    }
}
```

JAXP: processor plugin

- Metodo per selezionare l'implementazione del parser XML che preferiamo a tempo di esecuzione
- Tipicamente sfrutta le system properties per individuare nel sistema il parser da utilizzare
- Interfacce:

```
javax.xml.parsers.SAXParserFactory
javax.xml.parsers.DocumentBuilderFactory
javax.xml.transform.TransformerFactory
```

- Esempio:

```
System.setProperty(
"javax.xml.parsers.DocumentBuilderFactory",
"com.icl.saxon.om.DocumentBuilderFactoryImpl");
```

- A default, l'implementazione di riferimento sfrutta
- Apache Crimson/Xerces per il parsing dei documenti XML
- Apache Xalan per la traformazione di documenti

Altre API Java per il Parsing di XML

- JDOM
  - variante di DOM; maggiormente allineato alla programmazione orientata agli oggetti
  - http://www.jdom.org/
- DOM4J
  - Simile a JDOM ma con maggiori funzionalità
  - http://www.dom4j.org/
- JAXB (Java Architecture for XML Binding)
  - Compila i DTD in classi Java DTD-specific che consentono di leggere, scrivere e manipolare documenti XML validi
  - http://java.sun.com/xml/jaxb/