XML

a.s. 2023/24

v17.10.23

1: Introduzione - perché utilizzare XML?

L'**XML (Extensible Markup Language)** è un *(meta)*-linguaggio di markup estremamente versatile e diffuso che riveste un ruolo fondamentale nella gestione dei dati strutturati nel mondo dell'informatica.

Le ragioni per cui XML è ampiamente utilizzato sono molteplici, e comprendono:

1. **Interoperabilità e Scambio di Dati:** XML è uno standard aperto e indipendente dalla piattaforma, ciò significa che i dati possono essere scambiati e condivisi tra sistemi eterogenei senza problemi di compatibilità. Ad esempio, è possibile utilizzare XML per trasferire dati tra un'applicazione scritta in Java su un sistema Windows e un'applicazione scritta in Python su un sistema Linux.

2. **Strutturazione dei Dati:** XML offre un modo efficace per strutturare e organizzare dati complessi. È possibile definire la struttura gerarchica dei dati utilizzando tag e attributi, rendendo più facile comprendere e interpretare il significato dei dati.

3. **Leggibilità Umana:** I documenti XML sono progettati per essere facilmente leggibili da esseri umani. La loro struttura basata su tag rende chiaro il significato dei dati, il che è utile per la documentazione, l'analisi e la *comprensione* della struttura dei dati.

4. **Flessibilità:** XML è estensibile e adattabile alle esigenze specifiche di una vasta gamma di applicazioni. Gli sviluppatori possono definire i propri **tag personalizzati per rappresentare dati specifici.**

5. **Supporto Multipli Linguaggi:** XML è supportato da molti linguaggi di programmazione, compresi Java, Python, C#, Ruby, e altri. Questo permette di utilizzare XML in una varietà di contesti di sviluppo.

6. **Standardizzazione e Integrazione:** XML è alla base di molte tecnologie e standard importanti. Ad esempio, è utilizzato per definire la struttura dei documenti HTML, è parte integrante dei servizi Web (utilizzando SOAP), viene utilizzato per la definizione di protocolli di comunicazione (come XML-RPC), e alimenta i feed di notizie RSS.

1.1 - Esempi di Utilizzo di XML Oggi:

- **Servizi Web:** XML è ampiamente utilizzato per la comunicazione tra applicazioni distribuite attraverso i servizi Web. *SOAP (Simple Object Access Protocol)* utilizza XML per la definizione dei messaggi scambiati tra client e server.
- **Documenti Scientifici:** XML è utilizzato per rappresentare documenti scientifici strutturati, come documenti di ricerca, articoli scientifici e dati sperimentali.
- **Basi di Dati XML:** Alcune basi di dati utilizzano XML per archiviare dati in un formato strutturato e consultabile.
- **Configurazioni e Impostazioni:** XML è spesso utilizzato per definire configurazioni e impostazioni di applicazioni e dispositivi, consentendo la personalizzazione delle applicazioni.

XML è molto utilizzato per lo scambio di dati tra applicazioni, il salvataggio di configurazioni e la rappresentazione di dati gerarchici.

2: Sintassi di base di XML

Un documento XML è costituito da elementi, attributi e testo **(valore)**.

Gli elementi XML sono racchiusi tra tag di apertura **<elemento>** e tag di chiusura **</elemento>.**
Gli attributi vengono definiti all'interno dei tag e hanno un nome e un valore:

```
<elemento attributo="valore">    testo    </elemento>
```

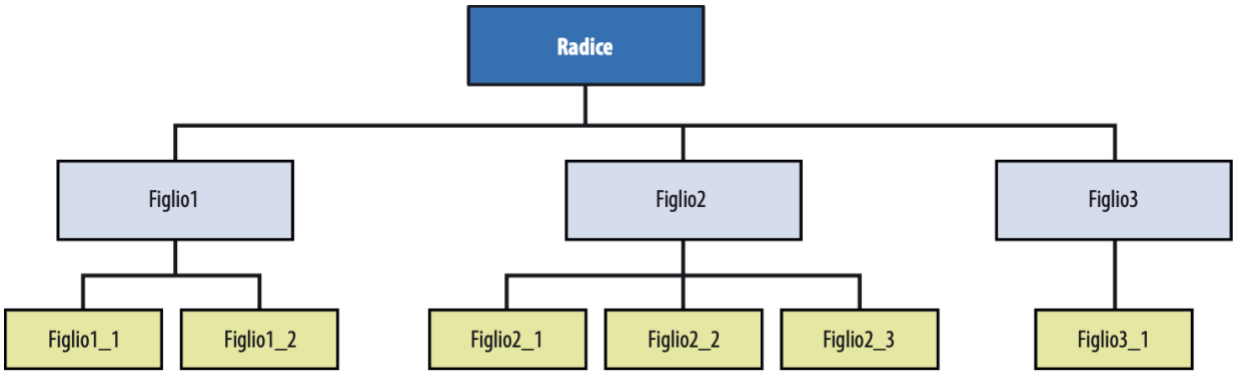
Un documento XML **deve** avere **un elemento radice** che racchiude tutti gli altri elementi.
Questa definizione ricorda un famossismo linguaggio di markup.

Esercitazione 1:
Scrivi un documento XML semplice che rappresenti le informazioni di un libro con titolo, autore ed anno di pubblicazione.

Soluzione 1:

```
<?xml version="1.0" encoding="UTF-8"?>
<libro>
    <titolo> Il grande inverno </titolo>
    <autore> George R. R. Martin</autore>
    <anno_pubblicazione>1991</anno_pubblicazione>
</libro>
```

Gerarchia degli elementi



2.1 - Gli attributi in un documento XML

Gli attributi in un file **XML**: Possono essere dichiarati direttamente all'interno di un elemento XML, come segue:
Esempio:

```
<persona nome="Marco" cognome="Rossi" età="30" />
```

In questo esempio, gli attributi nome, cognome, e età sono definiti direttamente nell'elemento **<persona>**

Questa è la forma più comune di rappresentazione degli attributi in XML ed è spesso preferita quando gli attributi sono strettamente associati a un elemento specifico.

2.1.2 Gli attributi in un file DTD:

Attributi separati dagli elementi - (All'interno dei documenti DTD)
Gli attributi possono essere dichiarati separatamente dagli elementi e associati a un elemento utilizzando una "entità" o una "ref" (riferimento).
Esempio:

```
<!ELEMENT persona EMPTY>
  <!ATTLIST persona
      nome CDATA #REQUIRED
      cognome CDATA #REQUIRED
      età CDATA #IMPLIED
  >

<persona nome="Marco" cognome="Rossi" età="30" />
```

In questo esempio, gli attributi nome, cognome, e età sono definiti separatamente dagli elementi, all'interno di una dichiarazione **ATTLIST** associata all'elemento **<persona>.**

Questo approccio è meno comune rispetto alla dichiarazione degli attributi direttamente all'interno degli elementi ed è spesso utilizzato quando è necessario definire regole di validità complesse o se si desidera raccogliere tutte le dichiarazioni degli attributi in un unico punto nel documento XML.

Entrambi i modi sono corretti per rappresentare attributi, ma **la scelta tra di essi dipende spesso dalle esigenze specifiche del tuo progetto e dalla tua preferenza personale.**

La rappresentazione diretta degli attributi all'interno degli elementi è più comune e leggibile, mentre la dichiarazione separata degli attributi può essere più utile in **scenari di validazione XML complessi o in cui è necessario applicare regole di validità specifiche agli attributi stessi.**

2.2: Esercizi

Esercizio 1: Documento XML Semplice
Traccia: Crea un documento XML che rappresenti i dettagli di un contatto, incluso il nome e l'indirizzo email.

Soluzione:

```
<contatto>
    <nome>Mario Rossi</nome>
```

```
    <email>mario@example.com</email>
</contatto>
```

Esercizio 2: Documento XML Semplice
Traccia: Crea un documento XML che rappresenti i dettagli di un libro, compreso il titolo, l'autore e una breve descrizione.

Soluzione:
```
<libro>
    <titolo>Il Signore degli Anelli</titolo>
    <autore>J.R.R. Tolkien</autore>
    <descrizione>Un epico racconto fantasy.</descrizione>
</libro>
```

Esercizio 3: Documento XML con Attributi
Traccia: Crea un documento XML che rappresenti un prodotto, incluso il nome, il prezzo e la categoria come attributi.

Soluzione:
```
<prodotto nome="Smartphone" prezzo="499.99" categoria="Elettronica" />
```

Esercizio 4: Documento XML con più Elementi
Traccia: Crea un documento XML che rappresenti una lista di studenti, ognuno con nome, cognome e numero di matricola.

Soluzione:
```
<università>
    <studente>
        <nome>Anna</nome>
        <cognome>Bianchi</cognome>
        <matricola>12345</matricola>
    </studente>
    <studente>
        <nome>Luca</nome>
        <cognome>Rossi</cognome>
        <matricola>67890</matricola>
    </studente>
</università>
```

Esercizio 5: Documento XML con Elementi Complessi e Attributi
Traccia: Crea un documento XML che rappresenti un ordine online, con informazioni sul cliente, gli articoli dell'ordine e i dettagli di spedizione. Utilizza attributi per le informazioni principali.

Soluzione:
```
<ordine numero="12345">
    <cliente nome="Maria Rossi" email="maria@example.com" />
    <articoli>
        <articolo codice="A001" quantità="2" />
        <articolo codice="B002" quantità="1" />
    </articoli>
    <spedizione>
        <indirizzo>Via Roma, 123</indirizzo>
        <città>Roma</città>
        <cap>00100</cap>
    </spedizione>
</ordine>
```

Questi esempi rappresentano una serie di sfide crescenti, dalla creazione di documenti XML semplici con elementi di base a documenti più complessi con elementi nidificati, attributi e strutture dati più elaborate

Esercizio 6: Documento XML con Elementi Ripetuti e l'utilizzo di Attributi
Traccia: Crea un documento XML che rappresenti una lista di città con nome e popolazione, utilizzando attributi per la popolazione.

Soluzione:
```
<città>
    <città nome="Roma" popolazione="2873000" />
    <città nome="Parigi" popolazione="2187526" />
    <città nome="Londra" popolazione="8982000" />
</città>
```

Esercizio 7: Documento XML con Elementi Complessi Nidificati
Traccia: Crea un documento XML che rappresenti un curriculum vitae con dettagli personali, istruzione e esperienza lavorativa.

Soluzione:
```
<curriculum>
    <informazioni-personali>
        <nome>Maria Rossi</nome>
        <indirizzo>Via Roma, 123</indirizzo>
        <telefono>123-456-7890</telefono>
    </informazioni-personali>
    <istruzione>
        <scuola nome="Liceo Classico" anno="2010" />
        <università nome="Università di Roma" anno="2014" />
    </istruzione>
    <esperienza-lavorativa>
        <lavoro azienda="ABC Inc." anno="2015-2017" />
        <lavoro azienda="XYZ Corp." anno="2018-oggi" />
    </esperienza-lavorativa>
</curriculum>
```

Esercizio 8: Documento XML con Commenti
Traccia: Crea un documento XML che rappresenti una ricetta culinaria con ingredienti e istruzioni. Inserisci commenti per le istruzioni.

Soluzione:
```
<ricetta>
    <nome>Lasagne al Forno</nome>
    <ingredienti>
        <ingrediente>300g di lasagne</ingrediente>
        <ingrediente>500g di carne macinata</ingrediente>
        <ingrediente>400g di pomodori in scatola</ingrediente>
    </ingredienti>
    <istruzioni>
        <!-- Passo 1 -->
        <passo>Mettere a cuocere la carne macinata in una padella.</passo>
        <!-- Passo 2 -->
        <passo>Aggiungere i pomodori e cuocere per 10 minuti.</passo>
        <!-- Passo 3 -->
        <passo>Cuocere le lasagne in acqua bollente.</passo>
    </istruzioni>
</ricetta>
```

Esercizio 9: Documento XML con Attributi di Spazio dei Nomi
Traccia: Crea un documento XML che rappresenti una raccolta di dipinti con titolo, autore e anno di creazione, utilizzando uno ~~**spacename**~~ per l'arte.

Soluzione:

```
<arte:dipinti xmlns:arte="http://www.example.com/arte">
    <arte:dipinto>
        <arte:titolo>La Gioconda</arte:titolo>
        <arte:autore>Leonardo da Vinci</arte:autore>
        <arte:annoCreazione>1503-1506</arte:annoCreazione>
    </arte:dipinto>
    <!-- Altri dipinti qui -->
</arte:dipinti>
```

Esercizio 10: Documento XML con Utilizzo di CDATA
Traccia: Crea un documento XML che includa una descrizione lunga di un prodotto, utilizzando CDATA per racchiudere il testo.

Soluzione:

```
<prodotto>
    <nome>Smartphone</nome>
    <descrizione><![CDATA[Questo smartphone è dotato di un potente processore, schermo AMOLED da 6 pollici e fotocamera da 16 MP.]]></descrizione>
</prodotto>
```

Questi esempi aggiuntivi aumentano ulteriormente la complessità, includendo elementi ripetuti, elementi complessi nidificati, commenti, attributi di spazio dei nomi e l'uso di CDATA per il testo senza dover trattare i caratteri speciali XML.

Esercizio 11

Dato il seguente documento testuale, in formato ASCII, di un curriculum vitae:

Curriculum Vitae di Mario Rossi

Dati personali

Nome	Mario
Cognome	Rossi
Residenza	Via Milano, 40 - 10100 Roma
e-mail	mario.rossi@email.it
tel.	01-2345678
cell.	111-222333444
Nato il	11-6-1960 a Como (Co)
Nazionalità	Italiana
Stato civile	Coniugato

Esperienze professionali

1990-1995 Computer Software s.r.l – Como
Ruolo: Analista programmatore
Attività: Sviluppo di Sistemi Gestionali per piccole imprese tessili
1995-2020 Docente ITIS Magistri Cumacini – Como
Ruolo: Docente
Attività: Tecnologie e progettazione – Corso informatica 1

effettua la sua codifica in XML.

2.3 XML Namespace

Namespace XML forniscono un metodo per evitare conflitti tra i nomi degli elementi.

In XML, i nomi degli elementi sono definiti dallo sviluppatore. Questo si traduce spesso in un conflitto quando si cerca di mescolare documenti XML da diverse applicazioni XML.

Questo XML trasporta informazioni tabella HTML:

```
<table>
  <tr>
    <td>Elemento 1</td>
    <td>Elemento 2</td>
  </tr>
</table>
```

Questo XML contiene delle informazioni:

```
<table>
  <name>Tavolo</name>
  <width>80</width>
  <length>120</length>
</table>
```

Se questi frammenti XML vengono uniti in un unico elemento, si creerebbe un conflitto di nomi. Entrambi contengono **<table>** come elemento radice, ma gli elementi hanno contenuto e significato diverso.

Un utente o un'applicazione XML non sapranno come gestire queste differenze.

Possiamo risolvere il conflitto causato dal nome utilizzando un prefisso

Conflitti di nomi in XML possono essere facilmente evitati con un prefisso del nome **(NameSapce).**

Questo XML contiene informazioni di una tabella HTML, e di un tavolo:

```
<h:table>
  <h:tr>
    <h:td>Elemento 1</h:td>
    <h:td>Elemento 2</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>Tavolo</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

Nell'esempio precedente, non vi sarà alcun conflitto perché i due elementi **<table>** hanno nomi diversi.

3: I documenti DTD

(Document Type Definitions) sono utilizzati in XML per definire la struttura, la grammatica e le regole di validità di un documento XML.
Sono uno dei mezzi principali per definire il modello dati di un documento XML.

Ecco cosa sono, come funzionano e come assicurarsi che un file XML rispetti un DTD:

3.1 Cos'è un Documento DTD:

Un documento DTD è un file di testo che contiene regole specifiche per definire gli elementi, gli attributi e la struttura di un documento XML. In sostanza, **un DTD fornisce una "schema" per un documento XML, specificando quali elementi possono apparire, in quale ordine, e quali attributi sono ammessi per ciascun elemento.**

Un DTD può essere dichiarato nel documento XML stesso o in un file separato e collegato al documento **<<IMPORT>>.**

3.2 Come si Usa un Documento DTD:

1. **Dichiarazione nel Documento XML:** Per utilizzare un DTD, puoi dichiararlo nel tuo documento XML utilizzando la direttiva `<!DOCTYPE>`. Ad esempio:

```
<!DOCTYPE biblioteca SYSTEM "biblioteca.dtd">
```

   In questo caso, `**biblioteca.dtd**` è il file DTD che definisce la struttura della tua "**biblioteca**" XML.

Nel contesto della dichiarazione `**<!DOCTYPE>**`, la parola chiave `SYSTEM` indica che il documento DTD (Document Type Definition) è presente come un file esterno e viene referenziato tramite un sistema di identificazione di file, come un percorso del file system o un URL.

La sintassi completa della dichiarazione `**<!DOCTYPE>**` con la parola chiave `**SYSTEM**` è la seguente:

```
<!DOCTYPE root_element SYSTEM "system_identifier">
```

Dove:

- `**root_element**` è il nome dell'elemento radice del documento XML.
- `**SYSTEM**` indica che si sta facendo riferimento a un file DTD esterno.
- `**system_identifier**` è il percorso del file DTD nel sistema di file o un URL che punta al DTD esterno.

Nel caso specifico:

```
<!DOCTYPE biblioteca SYSTEM "biblioteca.dtd">
```

- `**biblioteca**` è l'elemento radice del tuo documento XML.
- `**SYSTEM**` indica che stai referenziando un file DTD esterno.
- `**"biblioteca.dtd"**` è il sistema di identificazione del file che indica che il file DTD si trova nella stessa directory del tuo documento XML e ha il nome `**biblioteca.dtd**`.

Quindi, il documento XML `**books.xml**` si riferisce al DTD esterno `**biblioteca.dtd**` utilizzando la dichiarazione `**<!DOCTYPE>**`, consentendo al **parser XML** di comprendere e convalidare la struttura del documento XML rispetto alle regole definite nel DTD.

2. **Definizione degli Elementi:** Nel DTD, si definiscono gli elementi che compongono il documento XML utilizzando la sintassi specifica.

Ad **esempio:**

```
<!ELEMENT libro (titolo, autore, anno_pubblicazione)>
```

Questa dichiarazione indica che un elemento `<libro>` può contenere elementi `<titolo>`, `<autore>` e `<anno_pubblicazione>` in quell'ordine specifico.

3. **Validazione:** Una volta dichiarato il DTD nel tuo documento XML, puoi utilizzare un parser XML per convalidare il documento rispetto alle regole definite nel DTD. ***Se il documento rispetta tutte le regole, è considerato valido.***

3.3 Introduzione: Verificare la Conformità di un File XML a un DTD:

Per verificare che un file XML rispetti le regole di un DTD, puoi fare quanto segue:

1. **Utilizzare un Parser XML:** Usa un parser XML, come ad esempio quello incluso nelle librerie XML di molti linguaggi di programmazione (Java, Python, C#, etc.), per analizzare il tuo documento XML e il DTD.
   Il parser segnalerà eventuali errori di validità.

2. **Validazione:** Il parser dovrebbe eseguire la validazione automaticamente quando si dichiara il DTD nel documento XML. Ad esempio, in Java, puoi utilizzare JAXP (Java API for XML Processing) e specificare l'opzione di validazione:

   ```java
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   factory.setValidating(true);
   ```

   Se il tuo documento XML non rispetta le regole del DTD, il parser genererà un errore di validazione. (vedremo più avanti un esempio pratico)

3. **Strumenti di Validazione Online:** Ci sono strumenti online che consentono di caricare il tuo documento XML e il DTD per verificarne la validità.
   Questi strumenti sono utili per la convalida rapida senza dover scrivere codice.

Assicurarsi che un file XML sia conforme a un DTD è fondamentale per garantire che i dati siano strutturati correttamente e possano essere elaborati in modo affidabile dai sistemi e dalle applicazioni che utilizzano l'XML come formato di dati. **La validazione è un passo critico nella gestione dei dati XML.**

3.3.1. Esempio XML e DTD

Ecco un esempio di un semplice documento XML:

```xml
<!-- books.xml -->
<?xml version="1.0"?>
<!DOCTYPE biblioteca SYSTEM "biblioteca.dtd">
<biblioteca>
    <libro>
        <titolo>Il Codice Da Vinci</titolo>
        <autore>Dan Brown</autore>
        <anno_pubblicazione>2003</anno_pubblicazione>
    </libro>
</biblioteca>
```

Questo invece è un **esempio** di un semplice DTD (`biblioteca.dtd`) che definisce la struttura del documento:

```
<!-- biblioteca.dtd -->
 <!ELEMENT biblioteca (libro+)>
  <!ELEMENT libro (titolo, autore, anno_pubblicazione)>
   <!ELEMENT titolo (#PCDATA)>
   <!ELEMENT autore (#PCDATA)>
   <!ELEMENT anno_pubblicazione (#PCDATA)>
```

Analizziamo ciascun elemento del codice DTD precedente:

```
<!-- biblioteca.dtd -->
```

Questa riga è un commento XML e può essere ignorata dal parser XML. Serve solo a scopo di documentazione per descrivere il DTD.

<!ELEMENT biblioteca (libro+)>

- `**<!ELEMENT biblioteca (libro+)>**`: Questa riga definisce l'elemento `biblioteca`.
- `**<!ELEMENT>**` è una dichiarazione che indica che stiamo definendo un elemento XML.
- `**biblioteca**` è il nome dell'elemento che stiamo definendo.
- `**(libro+)**` indica che `biblioteca` può contenere uno o più elementi `libro`. Il simbolo `+` significa "una o più occorrenze". Quindi, `biblioteca` deve contenere almeno un elemento `libro`.

<!ELEMENT libro (titolo, autore, anno_pubblicazione)>

- `**<!ELEMENT libro (titolo, autore, anno_pubblicazione)>**`: Questa riga definisce l'elemento `libro`.
- `**<!ELEMENT>**` è una dichiarazione che indica che stiamo definendo un elemento XML.
- `**libro**` è il nome dell'elemento che stiamo definendo.
- `**(titolo, autore, anno_pubblicazione)**` indica che `libro` deve contenere gli elementi `titolo`, `autore` e `anno_pubblicazione` **nell'ordine specificato**.

<!ELEMENT titolo (#PCDATA)>

- `**<!ELEMENT titolo (#PCDATA)>**`: Questa riga definisce l'elemento `titolo`.
- `**<!ELEMENT>**` è una dichiarazione che indica che stiamo definendo un elemento XML.
- `**titolo**` è il nome dell'elemento che stiamo definendo.
- `**(#PCDATA)**` indica che `titolo` può contenere dati di carattere (Parsed Character Data), cioè testo.

<!ELEMENT autore (#PCDATA)>

- `**<!ELEMENT autore (#PCDATA)>**`: Questa riga definisce l'elemento `autore`.
- `**<!ELEMENT>**` è una dichiarazione che indica che stiamo definendo un elemento XML.
- `**autore**` è il nome dell'elemente che stiamo definendo.
- `**(#PCDATA)**` indica che `autore` può contenere dati di carattere **(Parsed Character Data)**, cioè testo.

<!ELEMENT anno_pubblicazione (#PCDATA)>

- `**<!ELEMENT anno_pubblicazione (#PCDATA)>**`: Questa riga definisce l'elemento `**anno_pubblicazione**`.
- `**<!ELEMENT>**` è una dichiarazione che indica che stiamo definendo un elemento XML.
- `**anno_pubblicazione**` è il nome dell'elemento che stiamo definendo.
- `**(#PCDATA)**` indica che `anno_pubblicazione` può contenere dati di carattere (Parsed Character Data), cioè testo.

In sintesi, questo codice DTD definisce la struttura di un documento XML che rappresenta una **"biblioteca".**
La `**biblioteca**` può contenere uno o più elementi `**libro**`, ciascuno dei quali deve contenere un `**titolo**`, un `**autore**` e un `**anno_pubblicazione**`, in quest'ordine.
Gli elementi `**titolo**`, `**autore**` e `**anno_pubblicazione**` possono contenere testo.

Questo DTD specifica la struttura e i vincoli che il documento XML deve rispettare per essere considerato valido rispetto a questa definizione.

3.3.2 Simboli utilizzati per definire le regole di struttura degli elementi nei documenti XML

Oltre al **simbolo `+`,** ci sono altri simboli utilizzati nelle dichiarazioni degli elementi nei documenti DTD per specificare diverse proprietà:

1. **`*` (Asterisco):** Indica che un elemento può avere zero o più occorrenze dell'elemento specificato. Ad esempio, `(elemento*)` significa che `elemento` può essere ripetuto zero o più volte all'interno dell'elemento genitore.

2. **`?` (Punto interrogativo):** Indica che un elemento può essere opzionale, ovvero può essere presente una volta o essere del tutto assente. Ad esempio, `(elemento?)` significa che `elemento` può essere presente una volta o essere assente.

3. **`|` (Barra verticale):** Utilizzata per definire una scelta tra gli elementi. Ad esempio, **`(elemento1 | elemento2)`** significa che è possibile includere uno tra `elemento1` e `elemento2`. Si possono anche inserire più elementi predefiniti **(elemento1 | elemento2 | elemento*).**

4. **`#PCDATA` (Parsed Character Data):** Indica che un elemento può contenere dati di carattere (testo). Ad esempio, `(elemento #PCDATA)` significa che `elemento` può contenere testo.

5. **`EMPTY`:** Indica che un elemento non ha contenuto. In altre parole, è un elemento vuoto. Ad esempio, `(elemento EMPTY)` significa che `elemento` non può contenere alcun contenuto.

6. **`ANY`:** Indica che un elemento può contenere qualsiasi tipo di contenuto, inclusi elementi, attributi e dati di carattere. È il tipo meno restrittivo. Ad esempio, `(elemento ANY)` significa che `elemento` può contenere qualsiasi cosa.

Esempi di come questi simboli possono essere utilizzati nelle dichiarazioni degli elementi:

- **`(elemento*)`:** `elemento` può essere ripetuto zero o più volte.
- **`(elemento?)`:** `elemento` è opzionale, può essere presente una volta o essere assente.
- **`(elemento1 | elemento2)`:** Puoi scegliere tra `elemento1` e `elemento2`.
- **`(elemento #PCDATA)`:** `elemento` può contenere testo.
- **`(elemento EMPTY)`:** `elemento` è vuoto, non contiene contenuto.
- **`(elemento ANY)`:** `elemento` può contenere qualsiasi cosa.

Questi simboli vengono utilizzati per definire le regole di struttura degli elementi nei documenti XML, nei documenti DTD e per specificare le restrizioni o le opzioni di contenuto per gli elementi XML.

3.4 - <!ELEMENT> e <!ATTLIST> (DTD)

<!ELEMENT>:

Dichiarazione utilizzata in un **DTD** per definire la struttura di un elemento XML. Essa specifica quali elementi figli possono essere contenuti all'interno di un elemento padre e in quale quantità (ad esempio, uno o molti).
La sintassi di base di **<!ELEMENT>** è la seguente:

<!ELEMENT nome_elemento contenuto>

- **nome_elemento** è il nome dell'elemento XML che stai definendo.
- **contenuto** definisce il tipo di contenuto che può essere presente all'interno dell'elemento e può essere uno dei seguenti:
  - **EMPTY:** L'elemento è vuoto e non ha elementi figli.
  - **ANY:** L'elemento può contenere qualsiasi contenuto, inclusi altri elementi e testo.
  - **(contenuto):** L'elemento può contenere un particolare tipo di contenuto definito tra parentesi, ad esempio (#PCDATA) per testo di caratteri o (elemento) per un altro elemento specifico.

Esempio:

1. Per definire un elemento **<persona>** che può contenere un elemento **<nome>** seguito da un elemento **<cognome>** possiamo definire il seguente:
```
<!ELEMENT persona (nome, cognome)>
```

2. Per definire un elemento **<paragrafo>** che può contenere testo di caratteri:
```
<!ELEMENT paragrafo (#PCDATA)>
```

Nel contesto delle dichiarazioni <!ELEMENT> di un DTD (Document Type Definition) XML, i termini "EMPTY" e "ANY" **specificano come un elemento può essere composto o se può contenere altri elementi.**

- **EMPTY:**

Quando si utilizza **"EMPTY"** in una dichiarazione **<!ELEMENT>,** si sta indicando che l'elemento specificato è vuoto e non può contenere alcun elemento figlio. In altre parole, l'elemento non ha contenuto interno, tranne eventualmente gli attributi. Questo è utile quando si desidera rappresentare un elemento che funge principalmente da marcatore o da contenitore per gli attributi, senza alcun contenuto strutturato.

Esempio:
<!ELEMENT linea EMPTY>

In questo esempio, l'elemento **<linea>** è dichiarato come vuoto, il che significa che non può contenere elementi figli, ma potrebbe contenere attributi.

- **ANY:**

Quando si utilizza **"ANY"** in una dichiarazione **<!ELEMENT>**, si sta indicando che l'elemento specificato può contenere qualsiasi contenuto, inclusi elementi figli, testo, commenti e altri nodi. In altre parole, l'elemento non è soggetto a restrizioni specifiche sul suo contenuto.

Questa dichiarazione è utilizzata quando si desidera consentire una vasta gamma di contenuti all'interno dell'elemento senza specificare una struttura dettagliata.

Esempio:
<!ELEMENT contenitore ANY>

In questo esempio, l'elemento **<contenitore>** è dichiarato come **"ANY"**, il che significa che può contenere qualsiasi contenuto, compresi elementi figli, testo, commenti, ecc.

L'uso di **"EMPTY"** e **"ANY"** nelle dichiarazioni **<!ELEMENT>** dipende dalle esigenze specifiche del tuo documento XML e delle regole di validità che desideri applicare. **"EMPTY"** è utile per gli elementi che fungono principalmente da contenitori di attributi, mentre **"ANY"** è utile quando si desidera massima flessibilità nella struttura dell'elemento.

<!ATTLIST>:

è una dichiarazione utilizzata in un DTD per definire gli attributi associati a un elemento XML.

La sintassi di base di <!ATTLIST> è la seguente:

```
<!ATTLIST nome_elemento nome_attributo tipo_valore default_valore>
```

- **nome_elemento** è il nome dell'elemento XML a cui l'attributo è associato.
- **nome_attributo** è il nome dell'attributo.
- **tipo_valore** specifica il tipo di valore accettato dall'attributo (ad esempio, "CDATA" per dati di carattere).
- **default_valore** specifica il valore predefinito dell'attributo, se applicabile.

Queste dichiarazioni <ELEMENT> e <ATTLIST> sono fondamentali per definire la struttura e gli attributi di un documento XML nel contesto di un DTD. Consentono di stabilire regole e restrizioni sulla struttura e la validità dei documenti XML rispetto al DTD specifico.

3.5 - #REQUIRED e #IMPLIED

Nei DTD (Document Type Definitions) e nei documenti XML, #REQUIRED e #IMPLIED sono valori che possono essere utilizzati per specificare se un attributo è obbligatorio o facoltativo all'interno di un elemento. Ecco cosa significano e alcuni esempi dettagliati:

#REQUIRED: indica che un attributo è obbligatorio all'interno di un elemento. Ciò significa che ogni istanza di quell'elemento deve includere l'attributo specificato, e se l'attributo manca, il documento non sarà valido secondo il DTD.

Esempio:

```
<!ELEMENT persona EMPTY>
<!ATTLIST persona
    nome CDATA #REQUIRED
    cognome CDATA #REQUIRED
>
```

In questo esempio, l'elemento <persona> deve includere gli attributi nome e cognome. Se uno o entrambi gli attributi mancano, il documento non sarà valido.

#IMPLIED: indica che un attributo è facoltativo all'interno di un elemento. Ciò significa che l'attributo può essere presente o assente, senza compromettere la validità del documento XML.

Esempio:

```
<!ELEMENT persona EMPTY>
<!ATTLIST persona
    nome CDATA #IMPLIED
    cognome CDATA #IMPLIED
>
```

In questo esempio, gli attributi nome e cognome sono facoltativi per l'elemento <persona>. Il documento sarà valido indipendentemente dal fatto che essi siano presenti o meno.

Definire l'elenco di attributi con ATTLIST

Si può anche definire un elenco di possibili attributi per uno specifico elemento della lista mediante la clausola **ATTLIST**, con la seguente sintassi:

```
<!ATTLIST elementname
      AttrName1 AttrType1 Value1
      AttrName2 AttrType2 Value2
      ...
>
```

Dove:

- **element-name** è il nome dell'elemento;
- **AttrNamen** è il nome dell'elemento n-esimo;

AttrTypen è il tipo dell'elemento n-esimo, che può assumere i seguenti valori:

- **CDATA:** testo;
- **(en1|en2|...|enn):** valore scelto da una lista di enumerazione;
- **ID:** identificatore univoco a livello di documento che non può iniziare con un numero e che viene generalmente indicato come obbligatorio **(#REQUIRED).**

Value nindica il valore di default dell'attributo n-esimo o modificatore di presenza che può essere:

- **"valore":** l'attributo ha valore di default pari a valore;
- **#REQUIRED:** l'attributo deve essere presente;
- **#IMPLIED:** l'attributo è opzionale e non è possibile stabilire un valore di default;
- **#FIXED "valore":** l'attributo deve avere un valore fi sso pari a valore.

Esempio:

*È dato lo schema che descrive un record anagrafico di una persona memorizzato in un file "**informatici.dtd**":*

```
<!ELEMENT persona (nominativo, professione*)>
<!ATTLIST persona nato CDATA #REQUIRED deceduto CDATA #IMPLIED>
<!ELEMENT nominativo (nome, cognome)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT cognome (#PCDATA)>
<!ELEMENT professione (#PCDATA)>
```

- *Vediamo il significato di ogni singola riga del DTD:*
- *la prima riga dichiara che l'elemento persona deve contenere esattamente un elemento fi glio nominativo e zero o più elementi figlio professione, in questo ordine;*
- *la seconda riga dice che lo stesso elemento persona deve avere un attributo nato e può avere un attributo deceduto (l'ordine non è rilevante);*
- *la terza riga dichiara che il nominativo deve avere due figli chiamati nome e cognome, riportati in questo ordine;*
- *le ultime tre righe specificano che gli elementi nome, cognome e professione devono contenere dati di carattere analizzati (PCDATA), ovvero testo non elaborato che potrebbe contenere riferimenti a entità e caratteri ma non contenente tag.*

*Ecco un **esempio completo** di un documento XML che utilizza queste dichiarazioni di attributi:*

```
<persona nome="Marco" cognome="Rossi" />
```

In questo caso, sia nome che cognome sono presenti, quindi il documento è valido.

```
<persona nome="Maria" />
```

In questo caso, solo l'attributo nome è presente, ma poiché entrambi gli attributi sono stati dichiarati come #IMPLIED, il documento è ancora valido.

```
<persona />
```

In questo caso, nessun attributo è presente, ma il documento è ancora valido perché entrambi gli attributi sono dichiarati come #IMPLIED.

In sintesi, **#REQUIRED indica che un attributo deve essere presente,** mentre **#IMPLIED indica che un attributo è facoltativo e può essere omesso senza invalidare il documento rispetto al DTD.**

5: Validità di un Documento XML

Un documento XML è considerato valido quando rispetta una specifica struttura e regole definite da uno schema XML (come un DTD o uno schema XSD) o quando segue le specifiche del linguaggio XML, comprese le regole di ben formazione come la corretta struttura dei tag, la chiusura adeguata dei tag, e così via. In altre parole, un documento XML è valido quando soddisfa tutte le condizioni e le restrizioni imposte dallo schema o dalle specifiche XML pertinenti.

Un documento XML è:

- **Ben formato** quando il documento segue le regole di base di XML (regole sintattiche).
- **Valido** quando il documento segue una specifica struttura definita da un DTD (Document Type Definition) ed è ache ben formattato.

Esercitazione 2:

- Crea un documento XML ben formato ma non valido, ad esempio, omettendo un elemento richiesto **[LIBRO].**

Soluzione 2:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<libro>
    <titolo>Il Codice Da Vinci</titolo>
    <!-- L'elemento 'autore' è omesso -->
    <anno_pubblicazione>2003</anno_pubblicazione>
</libro>
```

In questo esempio, l'elemento <autore> è stato omesso, rendendo il documento non valido se ci aspettiamo che tutti gli elementi siano presenti.

Domanda: *Ma come faccio a sapere quali elementi non devono essere omessi?*

Risposta:

Document Type Definition (DTD): Se il documento utilizza un DTD per definire la struttura, dovresti esaminare il DTD stesso. Il DTD contiene dichiarazioni che specificano quali elementi devono essere presenti e in che modo. Puoi trovare queste dichiarazioni all'interno del DTD, che è generalmente dichiarato nel documento XML utilizzando la direttiva <!DOCTYPE>.

5.1 - Crea un documento XML valido utilizzando un DTD.

Un **DTD (Document Type Definition)** è un documento utilizzato per definire la struttura e le regole di validità di un documento XML. In sostanza, un DTD stabilisce le regole per gli elementi e gli attributi che possono essere presenti in un documento XML specifico. Ecco una spiegazione dei principali elementi che caratterizzano un DTD e come si creano:

```xml
<!DOCTYPE libro [
    <!ELEMENT libro (titolo, autore, anno_pubblicazione)>
    <!ELEMENT titolo (#PCDATA)>
    <!ELEMENT autore (#PCDATA)>
    <!ELEMENT anno_pubblicazione (#PCDATA)>
]>
```

in questo caso, il DTD definisce che **l'elemento <libro>** deve contenere gli elementi **<titolo>, <autore> e <anno_pubblicazione>,** quindi nessuno di questi elementi può essere omesso per rendere il documento valido secondo questa definizione.

Ora, possiamo creare un documento XML valido basato su questo DTD:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE libro SYSTEM "libro.dtd">
<libro>
    <titolo>Il Codice Da Vinci</titolo>
    <autore>Dan Brown</autore>
    <anno_pubblicazione>2003</anno_pubblicazione>
</libro>
```

5.1.2 - CDATA e PCDATA

Sono due tipi di dati utilizzati nei Document Type Definitions (DTD) per specificare il tipo di contenuto consentito in un elemento XML.

Ecco cosa significano e come vengono utilizzati:

PCDATA (Parsed Character Data):
PCDATA è il tipo di dati predefinito nei documenti XML e rappresenta i dati di carattere analizzati (dati di carattere "parsati" o "interpretati" dall'XML parser).

Questo significa che il contenuto dell'elemento è analizzato dall'XML parser, e i caratteri speciali come <, >, &, ", e ' vengono interpretati come parte della sintassi XML e devono essere adeguatamente codificati (ad esempio, < per <).

Esempio:

```xml
<!ELEMENT testo (#PCDATA)>
```

```xml
<testo>Questo è un esempio di dati di carattere analizzati.</testo>
```

In questo esempio, l'elemento **<testo>** può contenere dati di carattere analizzati, che vengono interpretati dall'XML parser. La stringa "Questo è un esempio di dati di carattere analizzati." è un esempio di **PCDATA**.

CDATA (Character Data):
CDATA è un tipo di dati che rappresenta dati di carattere non analizzati. Il contenuto dell'elemento CDATA non viene interpretato dall'XML parser, il che significa che i caratteri speciali all'interno dell'elemento sono trattati come dati e non come parte della sintassi XML.

Esempio:

```xml
<!ELEMENT codice (#CDATA)>
```

```xml
<codice><![CDATA[<html><body><p>Questo è un esempio di dati di carattere non analizzati.</p></body></html>]]></codice>
```

In questo esempio, l'elemento <codice> contiene dati di carattere non analizzati, inclusi tag HTML e caratteri speciali come <, >, &, ", e '. Questi dati sono racchiusi in una sezione CDATA <![CDATA[...]]>.

Quindi:

- Usa **PCDATA** quando desideri che il contenuto dell'elemento sia interpretato come parte della sintassi XML standard. Questo è il caso più comune e viene utilizzato per testo normale all'interno di un elemento.

- Usa **CDATA** quando il contenuto dell'elemento deve contenere caratteri speciali, come tag HTML o codice sorgente, che non devono essere analizzati dall'XML parser. CDATA è utile quando si desidera includere blocchi di testo "grezzi" o dati non strutturati.

5.1.2.1 – Esempi Combinati di PCDATA e CDATA

In alcuni casi, potresti avere un elemento che contiene sia **PCDATA** che **CDATA**.

Ad esempio, potresti avere un elemento che rappresenta una descrizione generale **(PCDATA)** e un elemento separato che contiene dati di carattere non analizzati **(CDATA).**

```
<!ELEMENT prodotto (nome, descrizione, codice)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT descrizione (#PCDATA)>
<!ELEMENT codice (#CDATA)>
```

In questo esempio, l'elemento **<nome>** e l'elemento **<descrizione>** contengono **dati di carattere analizzati (PCDATA)**, mentre l'elemento **<codice>** contiene **dati di carattere non analizzati (CDATA).**

In conclusione, PCDATA e CDATA sono utilizzati per specificare come il contenuto di un elemento XML deve essere interpretato.

5.2: Esercizi DTD

Esercizio 1: DTD Semplice
Traccia: Crea un DTD per un documento XML che rappresenta un elenco di prodotti, ognuno con un nome.

Soluzione:

```
<!ELEMENT listaProdotti (prodotto+)>
<!ELEMENT prodotto (#PCDATA)>
```

Esercizio 2: DTD con Elementi Nidificati
Traccia: Estendi l'esercizio precedente aggiungendo un prezzo per ogni prodotto.

Soluzione:

```
<!ELEMENT listaProdotti (prodotto+)>
<!ELEMENT prodotto (nome, prezzo)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prezzo (#PCDATA)>
```

Esercizio 3: DTD con Elementi Ripetuti
Traccia: Aggiungi la possibilità di avere più prodotti dello stesso tipo nell'elenco.

Soluzione:

```
<!ELEMENT listaProdotti (prodotto*)>
<!ELEMENT prodotto (nome, prezzo)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prezzo (#PCDATA)>
```

Esercizio 4: DTD con Attributi
Traccia: Modifica il DTD precedente per consentire di aggiungere un attributo "codice" a ciascun prodotto.

Soluzione:

```
<!ELEMENT listaProdotti (prodotto*)>
<!ELEMENT prodotto (nome, prezzo)>
<!ATTLIST prodotto codice CDATA #REQUIRED>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prezzo (#PCDATA)>
```

Esercizio 5: DTD con Elementi Opzionali
Traccia: Modifica il DTD precedente aggiungendo la possibilità di includere una descrizione opzionale per ciascun prodotto.

Soluzione:

```
<!ELEMENT listaProdotti (prodotto*)>
<!ELEMENT prodotto (nome, prezzo, descrizione?)>
<!ATTLIST prodotto codice CDATA #REQUIRED>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prezzo (#PCDATA)>
<!ELEMENT descrizione (#PCDATA)>
```

Esercizio 6: DTD con Elementi Complessi Nidificati
Traccia: Modifica il DTD precedente per consentire di rappresentare un catalogo di prodotti suddivisi per categorie.

Soluzione:

```
<!ELEMENT catalogo (categoria+)>
<!ELEMENT categoria (nome, prodotto*)>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prodotto (nome, prezzo, descrizione?)>
<!ATTLIST prodotto codice CDATA #REQUIRED>
<!ELEMENT nome (#PCDATA)>
<!ELEMENT prezzo (#PCDATA)>
<!ELEMENT descrizione (#PCDATA)>
```

XML degli esercizi DTD svolti

Esercizio 1: DTD Semplice

```
<!-- listaProdotti.xml -->
<listaProdotti>
    <prodotto>Prodotto 1</prodotto>
    <prodotto>Prodotto 2</prodotto>
    <prodotto>Prodotto 3</prodotto>
</listaProdotti>
```

Esercizio 2: DTD con Elementi Nidificati

```
<!-- listaProdotti.xml -->
<listaProdotti>
    <prodotto>
        <nome>Prodotto 1</nome>
        <prezzo>10.99</prezzo>
    </prodotto>
    <prodotto>
        <nome>Prodotto 2</nome>
        <prezzo>19.99</prezzo>
    </prodotto>
</listaProdotti>
```

Esercizio 3: DTD con Elementi Ripetuti

```
<!-- listaProdotti.xml -->
<listaProdotti>
    <prodotto>
        <nome>Prodotto 1</nome>
        <prezzo>10.99</prezzo>
    </prodotto>
    <prodotto>
```

```
        <nome>Prodotto 2</nome>
        <prezzo>19.99</prezzo>
    </prodotto>
    <prodotto>
        <nome>Prodotto 3</nome>
        <prezzo>15.49</prezzo>
    </prodotto>
</listaProdotti>
```

Esercizio 4: DTD con Attributi

```
<!-- listaProdotti.xml -->
<listaProdotti>
    <prodotto codice="P001">
        <nome>Prodotto 1</nome>
        <prezzo>10.99</prezzo>
    </prodotto>
    <prodotto codice="P002">
        <nome>Prodotto 2</nome>
        <prezzo>19.99</prezzo>
    </prodotto>
</listaProdotti>
```

Esercizio 5: DTD con Elementi Opzionali

```
<!-- listaProdotti.xml -->
<listaProdotti>
    <prodotto codice="P001">
        <nome>Prodotto 1</nome>
        <prezzo>10.99</prezzo>
        <descrizione>Descrizione del Prodotto 1</descrizione>
    </prodotto>
    <prodotto codice="P002">
        <nome>Prodotto 2</nome>
        <prezzo>19.99</prezzo>
    </prodotto>
</listaProdotti>
```

Esercizio 6: DTD con Elementi Complessi Nidificati

```
<!-- catalogo.xml -->
<catalogo>
    <categoria>
        <nome>Elettronica</nome>
        <prodotto codice="E001">
            <nome>Smartphone</nome>
            <prezzo>499.99</prezzo>
        </prodotto>
        <prodotto codice="E002">
            <nome>Tablet</nome>
            <prezzo>299.99</prezzo>
        </prodotto>
    </categoria>
    <categoria>
        <nome>Abbigliamento</nome>
        <prodotto codice="A001">
            <nome>T-shirt</nome>
            <prezzo>19.99</prezzo>
            <descrizione>Comoda maglietta in cotone.</descrizione>
        </prodotto>
    </categoria>
</catalogo>
```

Conclusione

Il DTD è un documento opzionale che consiste in un insieme di regole specifi che per definire la struttura di un documento XML, in particolare definisce:

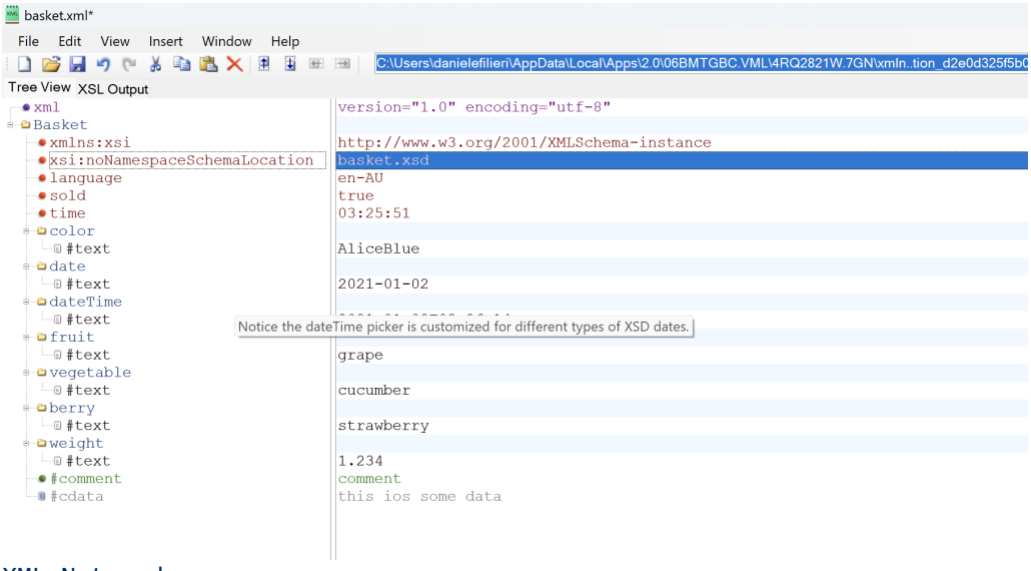
- quali elementi possono e/o devono essere usati nel documento;
- quali attributi si possono usare per ogni elemento;
- i vincoli sulle relazioni tra elementi.

Il DTD ha grande utilità per:

1. verificare la correttezza e la validità di un documento XML;
2. chi deve interpretare il documento XML, che sia il programmatore oppure un parser automatico;
3. definire i fogli di stile da associare ai documenti XML e/o HTML;
4. creare interfacce dinamiche per documenti XML;
5. l'interscambio di documenti.

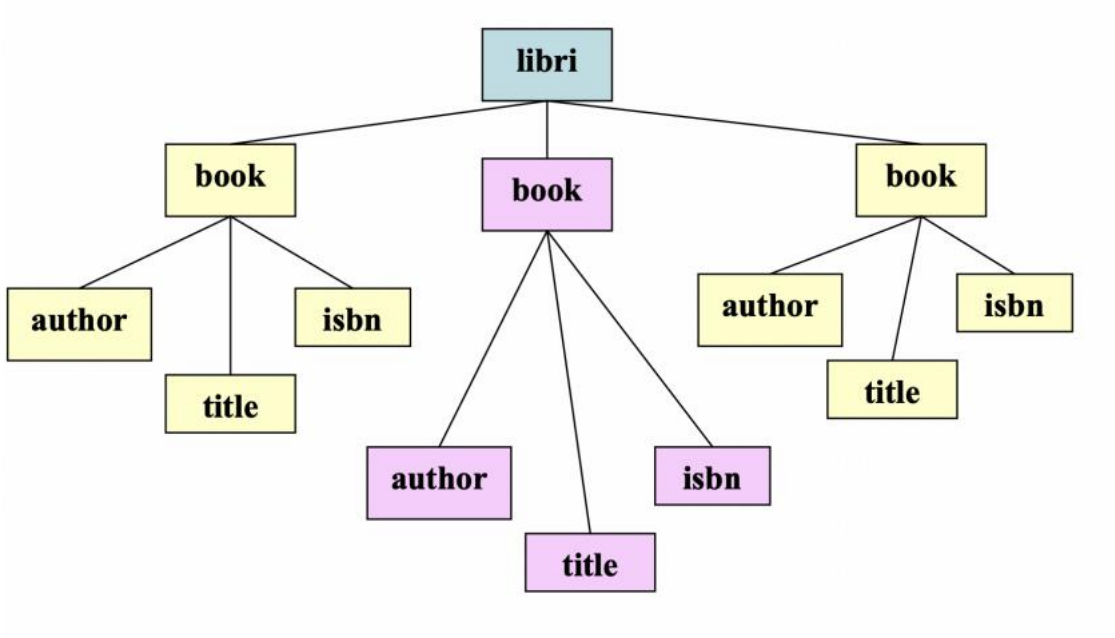
Il DTD ha alcuni svantaggi, evidenziamo i più importanti:

1. non è possibile defi nire il tipo degli elementi o attributi foglia che possono assumere come valore un qualsiasi dato carattere (o anche nessuno);
2. non è possibile limitare il numero di occorrenze di elementi ed è diffi cile specifi care contenuti non ordinati;
3. il meccanismo ID/IDREF è troppo semplice e non permette di limitare l'ambito di unicità degli attributi ID a un fram-mento ma solo all'intero documento;
4. le chiavi utilizzate per il meccanismo ID/IDREF devono essere solo attributi singoli;
5. dato che la sua defi nizione precede la raccomandazione del W3C non supporta gli spazi dei nomi;
6. fornisce un supporto limitato per la modularità, il riutilizzo e l'evoluzione degli schemi;
7. non è possibile verificare la correttezza della sua descrizione dato che non è scritto in notazione XML e, quindi, su di esso non si possono utilizzare gli strumenti che permettono di manipolare schemi con strumenti XML, come, per esempio, per verificare che sia ben formattato.



XML Notepad

5.3 Schema ad Albero



Uno schema ad albero è una struttura dati utilizzata per rappresentare la struttura gerarchica di un documento XML (e non solo). Questa rappresentazione ad albero è comunemente utilizzata per analizzare, manipolare e memorizzare dati XML in modo efficiente.

Nel mondo della programmazione, per lavorare con file XML, gli sviluppatori spesso utilizzano librerie o parser XML che convertono il documento XML in una struttura ad albero in memoria. Ciò consente loro di interrogare e manipolare facilmente i dati. Alcuni esempi comuni di parser XML includono DOM (Document Object Model) e SAX (Simple API for XML).

Ecco come potrebbe apparire la rappresentazione ad albero in memoria di un documento XML:

root

├── element1

│ └── Contenuto1

└── element2

├── attributo: Valore

└── Contenuto2

Ecco alcuni esempi di schemi ad albero per documenti XML:

Esempio 1:

```
<person>
    <name>John</name>
    <age>30</age>
</person>
```

Schema ad albero:

```
person
├── name
│   └── John
└── age
    └── 30
```

Esempio 2: Documento XML con attributi

```
<book title="Harry Potter" author="J.K. Rowling">
    <genre>Fantasy</genre>
    <price>20.00</price>
</book>
```

Schema ad albero:

```
book
├── title: Harry Potter
├── author: J.K. Rowling
├── genre
│   └── Fantasy
└── price
    └── 20.00
```

Esempio 3: Documento XML con elementi annidati

```
<address>
    <street>Main Street</street>
    <city>New York</city>
    <zipcode>10001</zipcode>
</address>
```

Schema ad albero:

```
address
├── street
│   └── Main Street
├── city
│   └── New York
└── zipcode
    └── 10001
```

Esempio 4: Documento XML con elementi multipli dello stesso tipo

```
<fruits>
    <fruit>Apple</fruit>
    <fruit>Banana</fruit>
    <fruit>Orange</fruit>
</fruits>
```

Schema ad albero:

```
fruits
├── fruit
│   ├── Apple
│   ├── Banana
```

```
        │   └── Orange
```

Esempio 5: Documento XML con elementi nidificati e attributi

```xml
<product category="Electronics">
    <name>Laptop</name>
    <price currency="USD">999.99</price>
</product>
```

Schema ad albero:

```
product
├── category: Electronics
├── name
│   └── Laptop
└── price
    ├── currency: USD
    └── 999.99
```

Questi esempi illustrano come la struttura ad albero può essere utilizzata per rappresentare la gerarchia dei dati in documenti XML, compresi gli attributi associati agli elementi. La struttura ad albero facilita la comprensione della struttura dei dati e l'accesso ai singoli elementi quando si lavora con documenti XML in programmazione o analisi dei dati.

6 XML Schema (cenni)

XML Schema (XSD), è un linguaggio utilizzato per definire la struttura e le regole di validazione per documenti XML. XML Schema è ampiamente utilizzato per garantire la coerenza e la validità dei dati XML in vari contesti, come il Web, la comunicazione tra sistemi e altro ancora.

6.1 Cos'è XML Schema?

XML Schema, abbreviato come XSD, è un linguaggio di definizione utilizzato per specificare la struttura e le regole di validazione per documenti XML.

In altre parole, XSD definisce quali elementi e attributi sono consentiti in un documento XML, insieme ai loro **tipi di dati**, ordinamenti e altre restrizioni.

6.2 Vantaggi di XML Schema:

- **Validazione:** Permette di controllare se un documento XML è conforme alle regole specificate nello schema.
- **Documentazione:** Fornisce una documentazione chiara sulla struttura del documento XML.
- **Interoperabilità:** Contribuisce a garantire che i documenti XML siano comprensibili e utilizzabili da diverse applicazioni.

6.3 Definire elementi in XML Schema:
Gli elementi sono i componenti principali di un documento XML.
In XML Schema, è possibile definire un elemento utilizzando l'elemento `**<element>**`.
Ecco un esempio di definizione di un elemento ***"persona":***

```
<xs:element name="persona" type="xs:string"/>
```

In questo esempio, abbiamo definito un elemento chiamato "persona" di tipo stringa.

6.4 Definire attributi in XML Schema:

Gli attributi forniscono informazioni aggiuntive sugli elementi. In XML Schema, è possibile definire un attributo utilizzando l'elemento `**<attribute>**`. Ecco un esempio di definizione di un attributo "età" per l'elemento "persona":

```
<xs:element name="persona">
  <xs:complexType>
    <xs:attribute name="età" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

In questo esempio, abbiamo definito un attributo "età" di tipo intero per l'elemento "persona".

Ecco come funziona:

- **<xs:element name="persona">:** Stiamo definendo un elemento chiamato "persona". Questo elemento rappresenta un oggetto di tipo persona in un documento XML.
- **<xs:complexType>:** All'interno dell'elemento "persona", stiamo specificando un tipo complesso utilizzando l'elemento <xs:complexType>. Un tipo complesso può contenere elementi e attributi.
- **<xs:attribute name="età" type="xs:integer"/>:** All'interno del tipo complesso, stiamo definendo un attributo chiamato "età". L'attributo "età" ha un nome ("età") e un tipo di dato (xs:integer), che indica che l'attributo "età" deve contenere un valore intero.

6.5 Validare un documento XML con XML Schema:

Per validare un documento XML utilizzando un XML Schema, è necessario fare riferimento allo schema nel documento XML stesso utilizzando un'istruzione `**xmlns**`.

Ecco un esempio:

```
<document xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="mio-schema.xsd">
  <!-- Contenuto del documento -->
</document>
```

In questo esempio, stiamo utilizzando l'attributo `***xsi:noNamespaceSchemaLocation***` per specificare la posizione del file XML Schema **"mio-schema.xsd".**

La validazione può essere effettuata utilizzando vari strumenti e librerie, a seconda dell'ambiente di sviluppo utilizzato.

XML Schema è uno strumento potente per definire e validare documenti XML in modo preciso e coerente. La sua struttura flessibile consente di definire regole dettagliate per i documenti XML e garantirne la conformità.

Ma l'aggiunta di tutte queste descrizioni comporta la complicazione della lettura dei file di formattazione prodotto in questo formato, anche da parte di programmatori esperti, rendendolo, di fatto, quasi inutilizzato.

7 XML Path Language

L'XML Path Language (XPath) è un linguaggio utilizzato per navigare e selezionare parti specifiche di un documento XML. È uno standard del World Wide Web Consortium (W3C) ampiamente utilizzato per interrogare documenti XML e recuperare dati da essi.

XPath fornisce un modo potente per definire percorsi nei documenti XML e individuare elementi, attributi, nodi di testo e altri componenti all'interno di un documento XML.

Ecco le principali caratteristiche e i principali utilizzi di XPath:

1. **Navigazione:** XPath consente di navigare all'interno di un documento XML in modo gerarchico. Si possono specificare percorsi che attraversano nodi padre, figli, fratelli, cugini, ecc., consentendo di accedere a parti specifiche del documento.

2. **Selezione:** Si può utilizzare XPath per selezionare elementi, attributi o nodi di testo in base a criteri specifici. Ad esempio, si possono selezionare tutti gli elementi `<p>` nel documento o solo quelli che hanno un attributo specifico.

3. **Filtro:** XPath supporta il filtraggio dei nodi in base a condizioni specifiche. Ad esempio, si possono selezionare tutti gli elementi **`<persona>` con un attributo "età" maggiore di 30.**

4. **Astrazioni:** XPath fornisce diverse astrazioni per rappresentare i nodi in un documento XML, ad esempio nodi di testo, elementi, attributi, nodi di commento, ecc.

5. **Funzioni:** XPath include una serie di funzioni incorporate che consentono di effettuare operazioni su dati recuperati dai documenti XML. Queste funzioni possono essere utili per eseguire operazioni matematiche, di manipolazione di stringhe o di data, ecc.

6. **Utilizzo diffuso:** XPath è utilizzato in molte tecnologie e standard, inclusi XSLT (Extensible Stylesheet Language Transformations), XQuery (un linguaggio di interrogazione XML), XML Schema e molti altri. Inoltre, è spesso utilizzato con linguaggi di programmazione come JavaScript e Python per elaborare dati XML.

Ecco un **esempio** di un'espressione XPath che seleziona tutti gli elementi **`<titolo>`** all'interno di un documento XML:

```
//titolo
```

Questa espressione selezionerà tutti gli elementi `<titolo>` indipendentemente dalla loro posizione nel documento. XPath è uno strumento estremamente utile per manipolare e recuperare dati da documenti XML in modo efficiente e accurato.

Alcuni siti Web consentono di caricare e/o copiare e incollare documenti XML e di immettere e valutare una stringa XPath:
ne riportiamo alcuni tra i più utilizzati

1. http://codebeautify.org/Xpath-Tester
2. http://www.freeformatter.com/xpath-tester.html
3. http://xpath.online-toolz.com/tools/xpatheditor.php
4. http://www.xpathtester.com/xpat

7.1 Gli elementi del linguaggio XPath

XPath consente di creare espressioni, dette espressioni XPath o pattern, che individuano i vari nodi dell'albero di rappresentazione di un documento XML, dove le espressioni XPath risultano essere costituite da token e delimi-tatori.

7.2 Elementi del linguaggio: i token

I token sono insiemi di caratteri Unicode che possono assumere due differenti significati:

1. di stringa se delimitati da virgolette;
2. di nodi-elemento se privi di virgolette.

In XPath è previsto l'utilizzo del carattere * come carattere jolly (wildcard) che permette di selezionare tutti i nodi del contesto indipendentemente dal loro nome.

ESEMPIO
L'espressione:
/ITIS2023/tpsit5/*

permette di selezionare tutti i nodi figlio degli elementi tpsit5 presenti sotto l'elemento radice ITIS2023 del documento XML.

Come nella sintassi dei percorsi del file system, il primo carattere **/** identifica il nodo **radice**, e un percorso aperto da **/** viene considerato **assoluto**.

7.3 Elementi del linguaggio: i nodi

Il modello di dati XPath rappresenta ogni documento XML come un albero di nodi, classificabili in tre tipologie fondamentali.

1. **Nodo Radice (Root Node):** il nodo radice è un nodo virtuale che non corrisponde a nessun componente nel documento XML ed è la radice che contiene tutti gli altri nodi. Il nodo radice non ha parent (fratelli) e il suo valore è il valore stringa del nodo dell'elemento documento.

2. **Nodo Elemento (Element Node):** ogni elemento XML del documento è un nodo elemento ed è etichettato con il tag dell'elemento XML;
   ha sempre un genitore (o il nodo radice o un nodo elemento) e può avere elementi di tipo elemento, commento, istruzione di elaborazione e testo. Il valore stringa di un nodo elemento è il testo contenuto tra i tag iniziale e finale dell'elemento, esclusi tutti i tag, i commenti e le istruzioni di elaborazione.

3. **Nodo Attributo (Attribute Node):** un nodo attributo corrisponde agli attributi di ciascun elemento; il suo genitore è il nodo dell'elemento che contiene l'attributo (ma *"non sono figli del loro genitore")* e il valore stringa di un nodo attributo è il valore dell'attributo.

Sono inoltre definiti altri quattro tipi di nodi:
Namespace Node: un nodo **namespace**, come i nodi degli attributi, i nodi **namespace** hanno un genitore ma "non sono figli del loro genitore" e il valore di stringa di un nodo è l›**URI** dello spazio dei nomi.

Istruzioni di elaborazione (Processing Instruction Node): un nodo dell'istruzione di elaborazione rappresenta un'istruzione di elaborazione; ha un genitore e nessun figlio

e il valore di stringa di un nodo dell'istruzione di elaborazione è il contenuto dell'istruzione esclusa la destinazione.

Testo (Text Node): rappresenta una stringa massima di testo tra tag, commenti e istruzioni di elaborazione; ha un nodo padre ma nessun figlio e il suo valore di stringa è il testo del nodo.

Commento (Comment Node): rappresenta un commento, ha un genitore e nessun figlio e il valore di stringa di un nodo di commento è il contenuto del commento.

7.4 Elementi del linguaggio: i tipi di dato e delimitatori

Il modello di dati XPath considera le dichiarazioni XML e DTD e tutti i risultati delle espressioni hanno come risultato uno dei seguenti tipi di dato:

- **node-set:** un insieme di nodi di un documento XML;
- **boolean:** un tipo di dato binario che può assumere il valore **true** o **false** ed è generalmente prodotto mediante operatori di confronto o logici;
- **number:** un tipo di dato floating point (compresi i valori speciali, Inf, –Inf e NaN);
- **string:** una sequenza di caratteri Unicode racchiusi tra apici singoli o doppi.

I **token** di un'espressione XPath sono separati da specifi ci caratteri delimitatori:

- **/** separa un passo/livello di localizzazione da quello a lui inferiore/interno;
- **[...]** definiscono un predicato;
- **=, !=, <, >, <=, >=** sono utilizzati all'interno di un predicato per verificare il valore **booleano** del suo contesto;
- **::** separa l'asse di un'espressione XPath dal nome di uno specifi co nodo;
- **//, @, ., ..** sono forme abbreviate per indicare rispettivamente "se stesso o discendente", "attributo", "se stesso", "genitore";
- **|** permette di definire un'espressione XPath composta dall'unione di più espressioni XPath;
- **(...)** raggruppano sottoespressioni XPath o delimitano gli argomenti di una funzione XPath;
- **+, –, *, div, mod** hanno la funzione degli operatori numerici ed eseguono le rispettive operazioni

7.5 Elementi del linguaggio: location step e path

XPath si compone di espressioni (o query) che vengono di volta in volta valutate in un determinato contesto restituendo uno o più nodi del documento XML.

Un'espressione è costituita da combinazioni di uno o più chunk di testo separati da caratteri speciali **'/'**, e vengono letti da sinistra a destra.

Le espressioni XPath individuano contenuti del documento XML o parte di questi: tale processo avviene in uno o più passi, detti location step, che messi insieme danno vita a un percorso completo di localizzazione, ossia un location path:

LocationPath = LocationStep/LocationStep/.../LocationStep

A ogni passo di valutazione dei location step (ossia procedendo da sinistra verso destra del location path) vengono esclusi alcuni nodi dalla selezione e vengono considerati i rimanenti.

Un location step identifica un nuovo node-set relativo al contesto nodeset corrente.
L'obiettivo di XPath è quello di individuare un nodo (o un insieme di nodi) o una porzione di documento XML utilizzando espressioni formate mediante un insieme di regole principali.
Il location path, che può essere assoluto o relativo, ammette due forme di scrittura per le espressioni XPath:

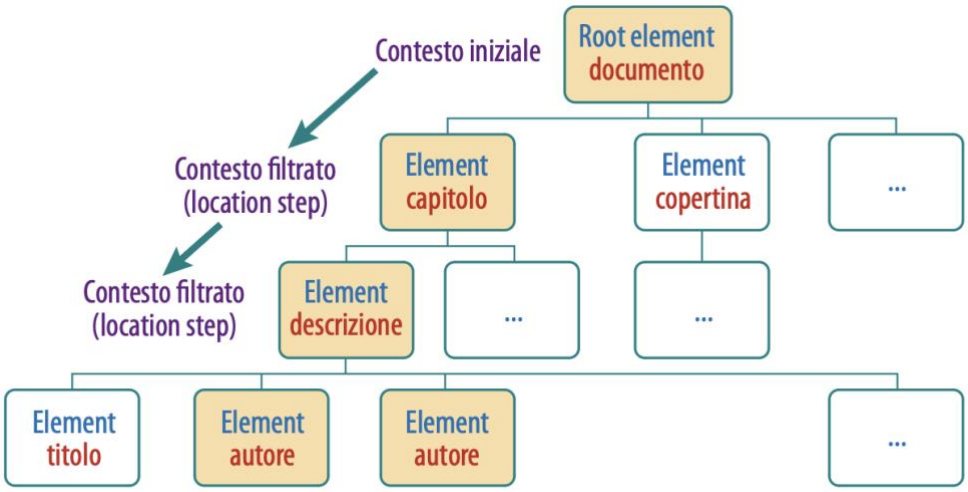
- la forma abbreviata che ricorda molto la sintassi della shell Unix con simboli come / . ..;
- la forma estesa che ha una sintassi particolare, e divide il suo costrutto in 3 parti:

axis::node_test[pred1][pred2]...[predN]

- un **axis** (asse), che individua la direzione di specifica del location step nell'albero: è opzionale;
- un **node_test** (test di nodo), che individua il tipo e il nome completo del nodo identificato dal location step;
- zero o più **predicati** che raffinano ulteriormente l'insieme di nodi selezionati dal location step.

La figura mostra il risultato della seguente location path:

```
/documento/capitolo/descrizione/autore
```



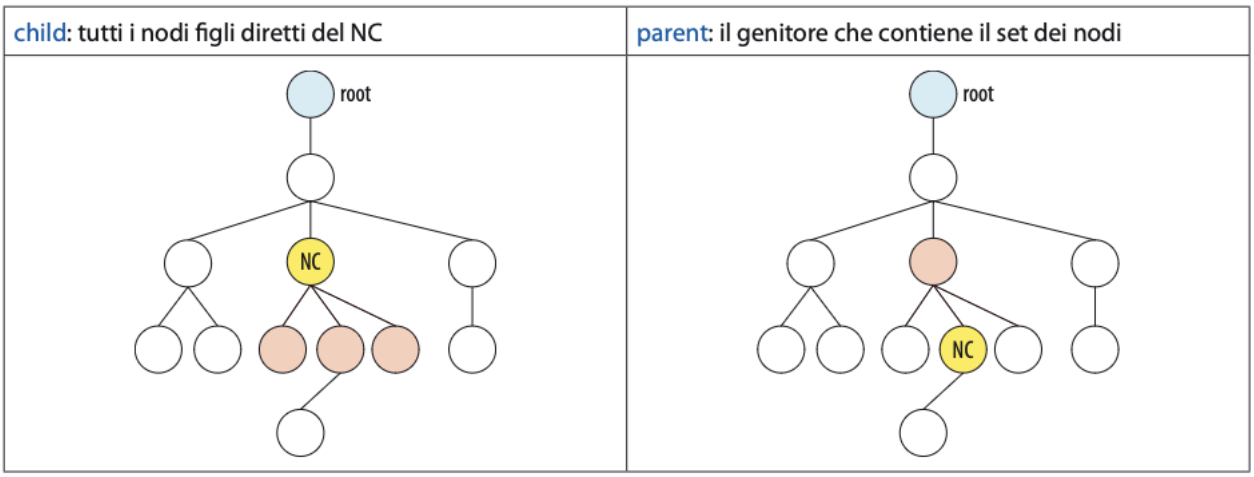
7.6 Elementi del location step: gli assi

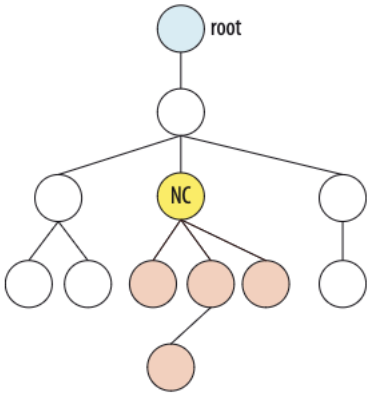
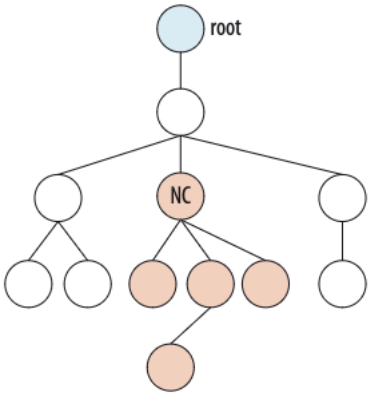
Gli assi identificano la direzione rispetto alla struttura del documento in cui andare a cercare l'oggetto da restituire rispetto al nodo contesto NC (o nodo corrente).

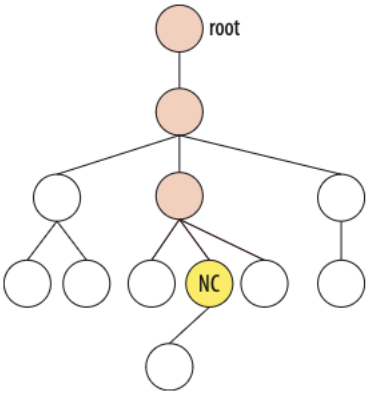
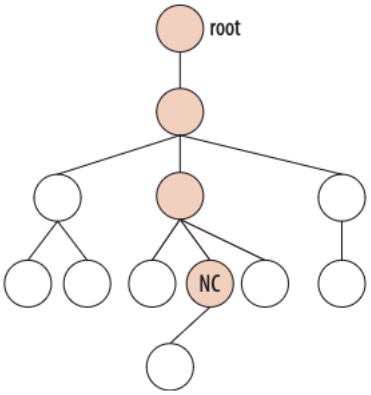
XPath definisce **13** tipi differenti di **assi**:

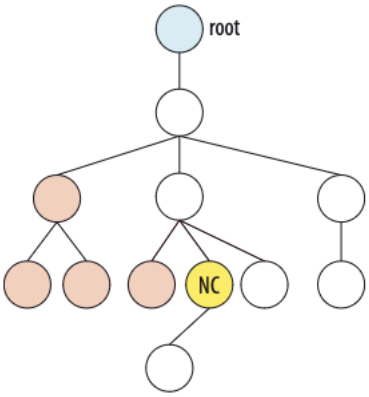
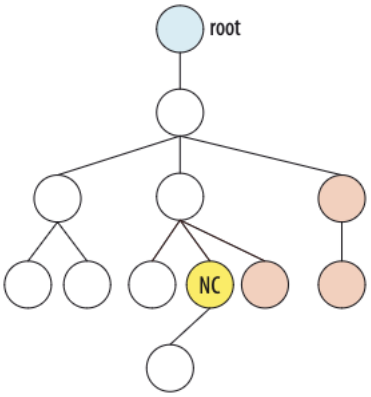
1. **child:** individua il nodo immediatamente discendente dal nodo contesto;
2. **parent:** individua il nodo genitore del nodo contesto;
3. descendant: individua tutti i nodi discendenti del nodo contesto a qualsiasi profondità (i figli, i figli dei figli ...);
4. **ancestor:** individua tutti gli antenati del nodo contesto fino all'elemento radice (il padre, il padre del padre ...);
5. **descendant-or-self:** individuano il nodo contesto stesso e tutti i suoi discendenti;
6. **ancestor-or-self:** individuano il nodo contesto stesso e tutti i suoi antenati;
7. **following:** individuano tutti i nodi che all'interno del documento XML seguono il nodo contesto esclusi i discendenti/antenati del nodo contesto;
8. **preceding:** individuano tutti i nodi che all'interno del documento XML precedono il nodo contesto esclusi i discen-denti/antenati del nodo contesto;
9. **following-sibling:** individuano tutti i nodi che seguono il nodo contesto e che condividono con questo lo stesso nodo parent;
10. **preceding-sibling:** individuano tutti i nodi che precedono il nodo contesto e che condividono con questo lo stesso nodo parent;
11. **self:** il nodo stesso (Nodo Corrente: NC);
12. **namespace:** il nodo namespace del set dei nodi NC;
13. **attribute:** gli attributi del set dei nodi di NC.

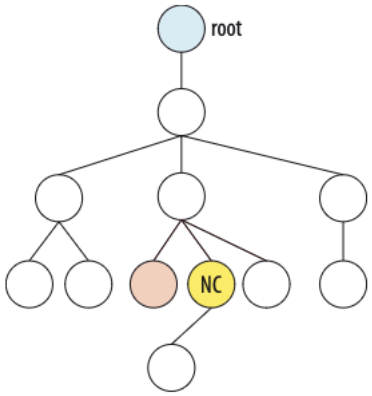
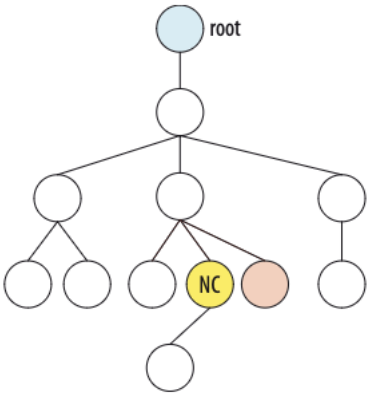
Di seguito illustriamo quelli per i quali la grafica ci viene in aiuto:



descendant: tutti i figli a qualunque livello del NC (figli, nipoti, ecc.)	descendant-or-self: il nodo contesto stesso e tutti i figli a qualunque livello del NC (esiste l'abbreviazione //)
	

ancestor: gli antenati di qualunque livello del NC	ancestor-or-self: il nodo contesto stesso e gli antenati di qualunque livello del NC
	

preceding: i nodi a qualunque livello (ma fuori al NC) che precedono il NC	following: i nodi a qualunque livello (ma fuori al NC) che seguono il NC
	

preceding-sibling: i nodi allo stesso livello (fratelli) ma non il NC	following-sibling: i nodi allo stesso livello (fratelli) ma seguenti il NC
	

Possiamo indicare l'asse utilizzando la forma estesa sopra indicata oppure la forma breve, preferita per la sua semplicità, anche se non permette di indicare tutti i tipi di assi, ma solo quelli relativi alla radice dell'albero oppure in riferimento relativo rispetto alla posizione corrente.

ABBREVIATA	ESTESA
name	child::name
@name	attribute::name
/	descendent::name
//	descendent-or-self::name
.	self::node()
..	parent::node()

7.7 Elementi del location step: il node_test

Il **node_test** indica quali nodi includere lungo l'axis specificato prima dei **::**, che può essere uno dei seguenti tipi possibili:

- **name** - ogni elemento o attributo con quel nome lungo l'asse specifi cato;
- - ogni elemento lungo l'asse specifi cato;
- **prefix**:* - ogni elemento o attributo con il namespace 'prefi x' lungo l'asse specifi cato;
- **comment()** - ogni commento lungo l'asse specifi cato;
- **text()** - ogni nodo di testo lungo l'asse specifi cato;
- **node()** - ogni nodo lungo l'asse specificato;
- **processing-instruction()** - ogni istruzione di elaborazione lungo l'asse specificato;
- **processing-instruction('target')** - ogni istruzione di elaborazione che si riferisce al relativo 'target' lungo l'asse specificato.

7.8 Elementi del location step: i predicati

Il **predicato** è solitamente racchiuso fra parentesi quadre ed espresso come un'espressione booleana con la seguente struttura:

```
[val1 operatore val2]
```

dove val1 e val2 sono delle espressioni XPath e operatore è uno degli operatori booleani descritti in precedenza.

Inoltre ogni location path che appare all'interno del predicato fa riferimento al contesto stabilito dai location step che precedono il predicato stesso.

ESEMPIO
Le seguenti espressioni sono differenti tra loro:

```
//descrizione/autore[@tipo="compositore"]
//descrizione[autore/@tipo="compositore"]
```

- Il **primo** predicato seleziona i nodi autorecon attributo tipo uguale a "compositore";
- il **secondo** predicato seleziona gli elementi descrizioneaventi almeno un elemento fi glio autore con attributo tipouguale a "compositore".

Il node_test indica il tipo di nodi che vogliamo selezionare; esistono due possibili approcci:

1. identificazione dei nodi in base al **nome** degli elementi;
2. identificazione dei nodi in base al **tipo** degli elementi.

L'utilità dei predicati sta nella possibilità di poter filtrare ulteriormente i nodi dal contesto corrente in base all'impo-stazione (e al verificarsi) di opportune condizioni.

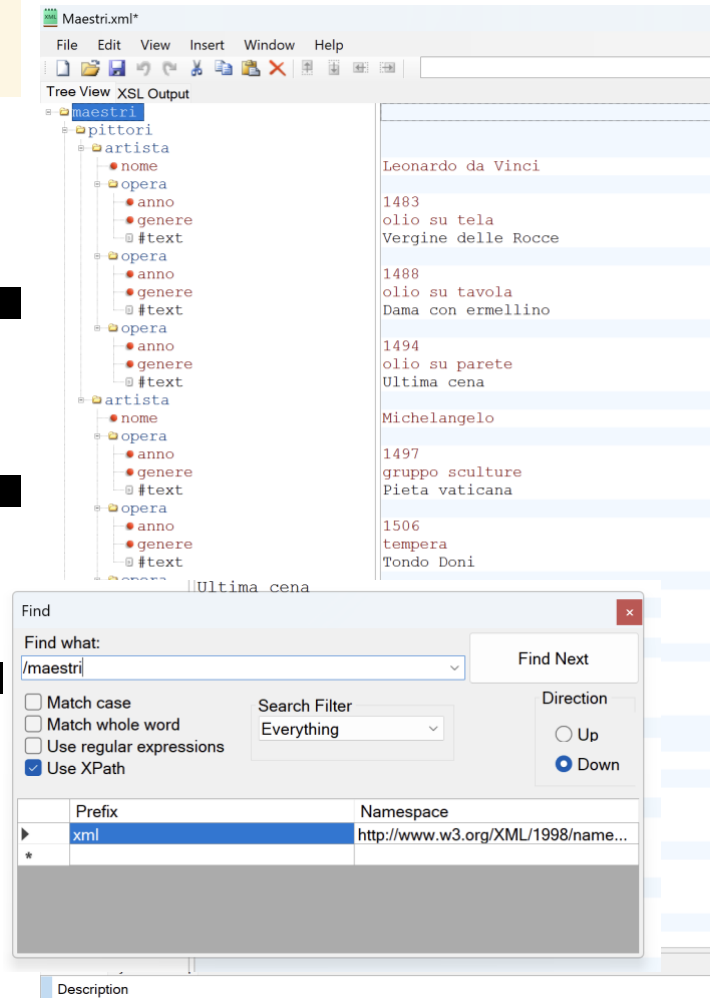
Riportiamo alcuni esempi di XPath.

XPATH	ELEMENTI SELEZIONATI
paragrafi [@tipo="riassunto"]	tutti i paragrafi figli del NC che abbiano l'attributo tipo uguale a "riassunto"
paragrafi [@tipo =" riassunto"][5]	il quinto paragrafo figlio di NC ad avere l'attributo tipo uguale a "riassunto"
paragrafi [5][@ tipo =" riassunto"]	il quinto paragrafo figlio di NC, ma solo se ha l'attributo tipo uguale a "riassunto"
capitolo[titolo]	il "capitolo" figlio del NC che contenga uno o più elementi "titolo"
//peso[@netto="11"]	tutti i nodi discendenti della root chiamati peso aventi attributo netto il cui valore è «11».

Per selezionare gli elementi con un particolare **nome** è sufficiente indicare come **node_test** il nome dell'elemento.

Esempio:

```
<maestri>
    <pittori>
        <artista nome= ”Leonardo da Vinci”>
            <opera anno= ”1483” genere= ”olio su tela”>Vergine delle Rocce</opera>
            <opera anno= ”1488” genere= ”olio su tavola”>Dama con ermellino</opera>
            <opera anno= ”1494” genere= ”olio su parete”>Ultima cena</opera>
        </artista>
        <artista nome= ”Michelangelo”>
            <opera anno= ”1497” genere= ”gruppo sculture”>Pieta vaticana</opera>
            <opera anno= ”1506” genere= ”tempera”>Tondo Doni</opera>
            <opera anno= ”1513” genere= ”statua”>Mose</opera>
            <opera anno= ”1537” genere= ”aff resco”>Giudizio universale</opera>
        </artista>
    </pittori>
    <musicisti>
        <artista nome= ”Giuseppe Verdi”>
            <opera anno= ”1871” atti= ”4”>Aida</opera>
            <opera anno= ”1851” atti= ”3”>Rigoletto</opera>
            <opera anno= ”1857” atti= ”3” prologo= ”1”>Simon Boccanegra</opera>
        </artista>
        <artista nome= ”Giacomo Puccini”>
            <opera anno= ”1896” atti= ”4”>La boheme</opera>
            <opera anno= ”1887” atti= ”3”>Tosca</opera>
            <opera anno= ”1926” atti= ”3”>Turandot</opera>
        </artista>
    </musicisti>
</maestri>
```



Scriviamo alcuni location path:

- per selezionare tutti gli elementi figli dell’elemento radice che hanno nome pari a “pittori” (sintassi con solo **node_test**) basta indicare semplicemente:
  `/maestri/pittori`
- per selezionare tutti gli elementi a prescindere dal loro nome è possibile utilizzare il carattere wildcard *: selezioniamo tutti i figli dell’elemento radice:
  `/*`
- per selezionare soltanto i nodi-testo degli elementi che hanno nome “autore” (fi gli a loro volta dell’elemento pittori della radice maestri):
  `/maestri/pittori/artista/text()`
  Analogamente si può utilizzare `node()` per selezionare tutti i nodi indipendentemente dal tipo, `comment()` per selezionare i nodi commento ecc.

7.9 Elementi del linguaggio: le funzioni

XPath non è un linguaggio di programmazione, ma in esso è definito un elenco completo di funzioni per elaborare/manipolare i node-set, ne ricordiamo le più utilizzate:

- **funzioni sul node-set:** last(), position(), count(), id(), local-name(), namespace-uri(), name();
- **funzioni sulle stringhe:** string(), concat(), starts-with(), contains(), substring-before(), substring-after(), substring(), string-length(), normalizespace(), translate();
- **funzioni booleane e numeriche:** boolean(), not(), true(), false(), lang();
- **funzioni numeriche:** number(), sum(), floor(), ceiling(), round().

Vediamo nel dettaglio le tre funzioni principali che operano sul node-set:

- **last()** restituisce il position-context dell'ultimo nodo del node-set corrispondente al contesto attualmente selezio-nato;
- **position()** restituisce il position-context di un determinato nodo all'interno del node-set corrispondente al contesto attualmente selezionato;
- **count(node-set)** restituisce il numero di nodi appartenenti a un node-set, agisce come **last()** ma a differenza di quest'ultimo accetta in ingresso un'espressione XPath che individua il node-set sul quale la funzione deve operare.

Riportiamo i comandi di location step più utilizzati, sempre facendo riferimento al file **maestri.xml** precedente:

COMANDO DI LOCATION STEP	NODI/ATTRIBUTI SELEZIONATI
child::maestri	figli dell'elemento maestri del nodo di contesto
child::*elementi	figlio del nodo di contesto
child::text()	figli del nodo di testo del nodo di contesto
child::node()	figli del nodo di contesto, qualunque sia il loro tipo di nodo
attribute::nomeattributo	nome del nodo di contesto
attribute::*	attributi del cenno del contesto
ancestor::pittori	antenati pittori del nodo di contesto
ancestor-or-self::pittori	antenati pittori del nodo di contesto ed, eventualmente, anche il nodo di contesto
descendant::pittori	discendenti dell'elemento pittori del nodo di contesto
descendant-or-self::pittori	discendenti dell'elemento pittori del nodo di contesto ed, eventualmente, anche il nodo di contesto
/	radice del documento (che è sempre il genitore dell'elemento documento)
/descendant:: pittori	elementi parametri nello stesso documento del nodo CN
self:: pittori	nodo di contesto se è un elemento pittori, e altrimenti non seleziona nulla
child::*/child:: pittori	pittori nipoti del nodo di contesto
child:: pittori [position()=1]	primo parametro secondario del nodo di contesto
child:: pittori [position()=last()]	ultimo parametro secondario del nodo di contesto
child:: pittori [position()=last()-1]	ultimo, tranne uno, fi glio secondario del nodo di contesto
child:: pittori [position()>]	figli secondari del nodo di contesto diversi dal primo figlio secondario del nodo di contesto

Esempio

Useremo il seguente documento XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="cooking">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
<book category="children">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="web">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>
<book category="web">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

Qui di seguito sono elencate alcune espressioni XPath e il risultato delle stesse:

espressioni XPath Risultato

- **/bookstore/book[1]** Seleziona il primo elemento libro che è il figlio dell'elemento libreria
- **/bookstore/book[last()]** Seleziona l'ultimo elemento libro che è il figlio dell'elemento libreria
- **/bookstore/book[last()-1]** Seleziona il penultimo libro elemento che è il figlio dell'elemento libreria
- **/bookstore/book[position()<3]** Seleziona i primi due elementi del libro che sono figli dell'elemento libreria
- **//title[@lang]** Seleziona tutti gli elementi del titolo che hanno un attributo denominato lang
- **//title[@lang='en']** Seleziona tutti gli elementi del titolo che hanno un "lang" attributo con un valore di "en"
- **/bookstore/book[price>35.00]** Seleziona tutti gli elementi del libro dell'elemento libreria che hanno un elemento di prezzo con un valore superiore a 35.00
- **/bookstore/book[price>35.00]/title** Seleziona tutti gli elementi del titolo degli elementi del libro dell'elemento libreria che hanno un elemento di prezzo con un valore superiore a 35.00

8. IL PARSING XML CON JAVA: LE SPECIFICHE JAXP

Un'operazione estremamente importante sui documenti XML è il **parsing**, che oltre a essere molto diffusa ha una notevole utilità.

In generale, eseguire il parsing di un documento significa costruire l'albero sintattico a esso associato, verificandone, eventualmente, la rispondenza con lo schema del documento stesso.

Il risultato dell'elaborazione è un albero che costituisce la base di ulteriori elaborazioni, come, per esempio, la trasformazione dello stesso in un altro formato (per esempio in CSV direttamente importabile all'interno dei database), oppure la ricerca veloce di particolari informazioni direttamente sul documento XML.

La realizzazione dell'albero richiede la valutazione del markup XML originale:
il parser deve elaborare tutti i documenti nella loro interezza per leggere ed estrarre i dati da ogni parte del testo corrispondente; la procedura più semplice è quella di leggere il documento XML, farne l'analisi sintattica e utilizzarlo all'interno di un programma.

L'analisi del documento avviene scandendo la sequenza dei simboli terminali, aggregandoli in entità più complesse sulla base dei tag secondo le semplici regole dell'XML.

Poiché le regole sintattiche dell'XML sono sempre le stesse, il codice che effettua l'analisi sintattica (parsing) può essere scritto una volta per tutte.

I principali controlli che il parser può eseguire sui documenti XML sono:

- verificare se il documento è **ben formato;**
- controllare che il documento **sia valido** in base a una definizione DTD oppure XML Schema.

Queste elaborazioni possono richiedere però molteplici risorse computazionali ed è quindi necessario utilizzare strumenti che ci permettano di scegliere quando attivarle o non eseguirle.

8.1 Approcci dei parser Java: DOM e SAX

In Java esistono sostanzialmente i seguenti due approcci.

8.1.1. Interfacce basate su eventi: Interfacce SAX (Simple API for XML).

Si tratta di un parser che, durante la lettura del fi le XML, ne individua i vari elementi e per ognuno di essi esegue le opportune azioni sfruttano un modello a callback; de facto è ora uno standard per il parsing di documenti XML.

Questo approccio si presta bene sia al trattamento di documenti molto semplici sia per le elaborazioni di documenti molto grandi che, per la loro lunghezza, non potrebbero essere contenuti nella memoria centrale.

8.1.2. Interfacce Object Model: W3C DOM Recommendation (Document Object Module).

Si tratta di un parser che legge l'intero documentoXML e lo trasforma in un albero che risiede nella memoria centrale; è quindi adatto nel caso di documenti che richiedano un'elaborazione complessa ma di dimensioni modeste.

Possiamo quindi affermare che SAX utilizza al meglio le risorse, ma non consente di navigare il documento al contrario né di modificare o creare documenti XML, operazioni queste invece facilmente realizzabili con DOM poiché tutto l'albero del documento viene caricato in memoria e si può lavorare su di esso come meglio preferiamo.

CSV

Il formato CSV (Comma-Separated Values) è un formato di file utilizzato per l'importazione ed esportazione di una tabella di dati in fogli elettronici o database: viene generato un semplice file di testo con separatori di campo e di record.

Callback

Con callback si intende una funzione (o metodo) che viene passato come parametro a un'altra funzione (o metodo).

8.2 Interfacce a eventi (SAX)

Nell'approccio a eventi l'applicazione implementa un insieme di metodi di callback che vengono invocati dal parser mentre elabora il documento al verificarsi di specifici eventi, cioè quando durante la scansione del documento vengono individuati particolari elementi, come il tag iniziale e finale ecc.;

al metodo callback vengono passati un insieme di parametri necessari a specificare e fornire ulteriori informazioni sull'evento, come per esempio

- il nome dell'elemento;
- il nome e valore assunto dagli attributi;
- il valore delle stringhe contenute ecc.

ESEMPIO
In SAX il metodo che individua il tag iniziale è **startElement()** e ha la seguente scrittura:

```
public void startElement(Stringnamespace, StringrawName, Stringdato, Attributesattributi)
```

dove i parametri sono:

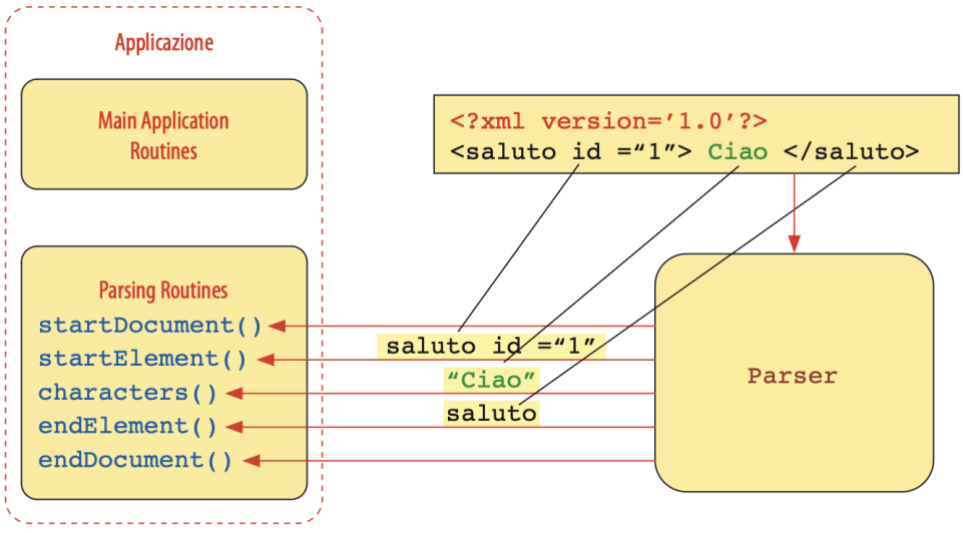
- **namespace**: URI del namespace associato all'elemento;
- **rawName**: nome nel formato XML 1.0;
- **dato**: nome dell'elemento;
- **attributi**: lista degli attributi.

Se si vuole "intercettare" ogni occorrenza del tag **<cognome>** nel corpo del metodo inseriamo il seguente codice:

```
public void startElement(Stringnamespace, StringrawName, Stringdato,Attributesattributi) {
    if (dato.equals("cognome"))
    System.out.println("Individuato l'inizio del tag <cognome>");
}
```

L'interfaccia a eventi è una modalità semplice ed efficiente, non mantiene in memoria una rappresentazione del documento ed è usato per grossi file XML o per accessi veloci: **è il modello adottato da SAX.**

Lo schema di parsing a eventi è riportato nella figura che segue applicato a un semplice esempio: possiamo osservare i momenti nei quali vengono richiamate le singole routine



8.3 Interfacce Object Model (DOM)

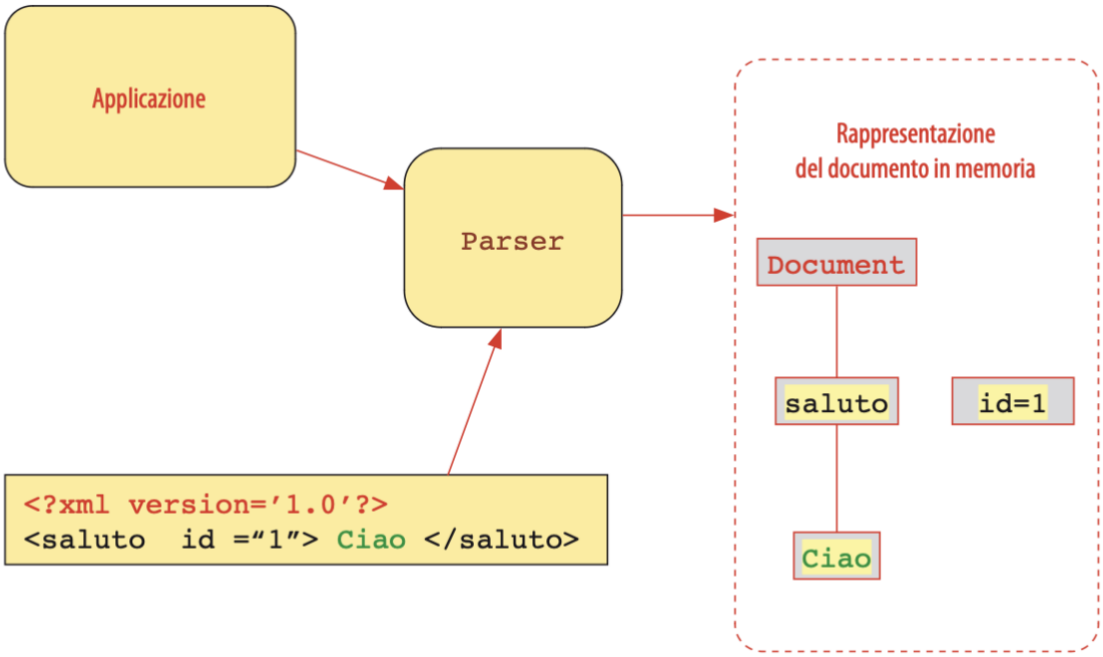
L'applicazione interagisce con una rappresentazione object-oriented del documento che viene interamente "ricostruito in memoria" in una struttura ad albero chiamata **parse tree**, nel quale **sono presenti oggetti per ciascun componente del file XML, cioè document, element, attribute, text ecc.**

La rappresentazione astratta realizzata mediante un albero favorisce la sua elaborazione anche grazie alle interfacce disponibili nel modello a oggetti: è facile visitare l'albero, accedere ai figli, ai fratelli ecc. di un elemento e/o aggiungere, rimuovere dinamicamente nodi;

L'unico limite che possiamo indicare per questo metodo è quello legato alla dimensione del file, che deve essere completamente contenuto nella memoria dinamica.

Il Document Object Model (DOM) è anche il modello adottato dal W3C (WWW Consortium) per le specifiche del Web.

Lo schema di **parsing object model** dell'esempio descritto in precedenza è riportato nella figura seguente



In **conclusione**, possiamo affermare che i parser XML rendono disponibile alle applicazioni la struttura e il contenuto dei documenti XML e si interfacciano mediante due tipologie di API:

1. **Event-based API (adottato da SAX):** le API di questa tipologia notificano alle applicazioni eventi generati nel parsing dei documenti usando poche risorse (parsing routine); non sono sempre comodissime da usare e non richiedono di avere tutto il documento in memoria;

2. **Object-model based (adottato da DOM):** questa tipologia di API realizza un **parse tree** che rappresenta il documento XML richiedendo maggiori risorse in termini di memoria dinamica in quanto il documento deve essere completamente presente in essa.

I parser più diffusi supportano sia SAX sia DOM: spesso i parser DOM sono sviluppati su parser SAX.

Nel seguito vedremo un esempio di parsing effettuato con SAX e uno con DOM, nei quali utilizzeremo come file da analizzare studenti.xml che contiene i seguenti dati:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<scuola>
    <studente ID="01">
        <nominativo>
            <nome> Paolo </nome>
            <cognome> Verdi </cognome>
        </nominativo>
        <indirizzo> via Verdi 15 </indirizzo>
        <citta> Genova </citta>
        <telefono> 339.1223344 </telefono>
    </studente>
    <studente ID="02">
        <nominativo>
            <nome> Paolo </nome>
            <cognome> Verdi </cognome>
        </nominativo>
        <indirizzo> via Roma, 22 </indirizzo>
        <citta> Genova </citta>
        <telefono> 339.1223355 </telefono>
    </studente>
    <studente ID="03">
        <nominativo>
            <nome> Stefano </nome>
            <cognome> Rossi </cognome>
        </nominativo>
        <indirizzo> via Milano 1 </indirizzo>
        <citta> Roma </citta>
    </studente>
</scuola>
```

Anticipazione
Esempio di codice Java per analizzare il file XML "scuola.xml", seguito da un possibile output del programma:

Questo programma Java è un esempio di come utilizzare JAXP (Java API for XML Processing) per analizzare un file XML chiamato "scuola.xml".
Il file XML rappresenta una serie di dati sugli studenti di una scuola, inclusi il loro ID, il nominativo, l'indirizzo, la città e il numero di telefono (se disponibile).
Il programma carica il file XML, estrae le informazioni sugli studenti e le visualizza sulla console.

```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class XMLParser {
    public static void main(String[] args) {
        try {
            // Crea una fabbrica per costruire il parser
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

            // Crea un costruttore per il documento XML
            DocumentBuilder builder = factory.newDocumentBuilder();

            // Analizza il documento XML
            Document document = builder.parse(new File("scuola.xml"));

            // Ottieni l'elemento radice del documento
            Element root = document.getDocumentElement();

            // Ottieni la lista di nodi "studente"
            NodeList studenti = root.getElementsByTagName("studente");

            // Itera attraverso i nodi "studente" ed estrai i dati
            for (int i = 0; i < studenti.getLength(); i++) {
                Element studente = (Element) studenti.item(i);
                String id = studente.getAttribute("ID");
                String nome =
studente.getElementsByTagName("nome").item(0).getTextContent().trim();
                String cognome =
studente.getElementsByTagName("cognome").item(0).getTextContent().trim();
                String indirizzo =
studente.getElementsByTagName("indirizzo").item(0).getTextContent().trim();
                String citta =
studente.getElementsByTagName("citta").item(0).getTextContent().trim();
                String telefono = "";

                NodeList telefoni = studente.getElementsByTagName("telefono");
                if (telefoni.getLength() > 0) {
                    telefono = telefoni.item(0).getTextContent().trim();
                }
```

```
                System.out.println("Studente " + id + ":");
                System.out.println("Nome: " + nome);
                System.out.println("Cognome: " + cognome);
                System.out.println("Indirizzo: " + indirizzo);
                System.out.println("Città: " + citta);
                System.out.println("Telefono: " + telefono);
                System.out.println();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Possibile output del programma:

```
Studente 01:
Nome: Paolo
Cognome: Verdi
Indirizzo: via Verdi 15
Città: Genova
Telefono: 339.1223344

Studente 02:
Nome: Paolo
Cognome: Verdi
Indirizzo: via Roma, 22
Città: Genova
Telefono: 339.1223355

Studente 03:
Nome: Stefano
Cognome: Rossi
Indirizzo: via Milano 1
Città: Roma
Telefono:
```

L'output del programma visualizza le informazioni sugli studenti estratte dal file XML, inclusi nome, cognome, indirizzo, città e, se disponibile, il numero di telefono.

8.5 Parser in Java: JAXP JAXP (Java API for XML Processing)

È un framework che ci consente di istanziare e utilizzare parser XML (sia SAX che DOM) nelle applicazioni Java: **JAXP 1.1 è inclusa nella JDK a partire dalla versione JDK 1.4**.

Fornisce vari package:

- **org.xml.sax:** interfacce SAX;
- **org.w3c.dom:** interfacce DOM;
- **javax.xml.parsers:** inizializzazione e uso dei parser;

quest'ultimo package mette a disposizione due classi astratte factory, una per SAX e una per DOM:

- SAXParserFactory;
- DocumentBuilderFactory.

In esse è presente il metodo statico **newInstance()** per creare un'istanza:

```
SAXParserFactoryspf =SAXParserFactory.newInstance();
DocumentBuilderFactorydbf =DocumentBuilderFactory.newInstance();
```

Creata una factory possiamo invocarla per creare un parser con interfaccia SAX o DOM:

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

La documentazione completa di JAXP è consultabile all'indirizzo:
http://download.oracle.com/javase/6/docs/technotes/guides/xml/jaxp/index.html.

Vediamo singolarmente i due metodi realizzando un'applicazione con SAX e una con DOM.

Interfacce SAX
Un parser SAX è orientato al flusso di dati, cioè legge il documento XML generando degli eventi quando incontra alcuni elementi strutturati;
le applicazioni interagiscono con parser conformi a SAX utilizzando interfacce stabilite, classificabili in tre categorie:

1. **Interfacce Parser-to-Application (callback):** consentono di definire funzioni di callback e di associarle agli eventi generati dal parser mentre questi processa un documento XML;
2. **Interfacce Application-to-Parser:** consentono di gestire il parser;
3. **Interfacce Ausiliarie:** facilitano la manipolazione delle informazioni fornite dal parser.

Un parser SAX non mantiene tutto l'albero del documento XML e gli eventi possono essere catturati e gestiti da rispettivi metodi Java: riportiamo quelli presenti nella versione 10

CARATTERISTICHE DELLE TRE TIPOLOGIE DI INTERFACCE
Interfacce Parser-to-Application (callback)
Vengono implementate dall'applicazione per imporre un preciso comportamento a seguito del verificarsi di un evento.
Sono quattro:

1. **ContentHandler:** metodi per elaborare gli eventi generati dal parser:
   a) 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);
   b) metodi per ricevere avvisi dell'inizio e fine documento: startDocument() endDocument()
   c) metodi per ricevere avvisi inizio e fine di un elemento: startElement(String namespaceURI, String localname, String rawName, Attributes atts) endElement(String namespaceURI, String localname, String qName).

   Possiamo osservare come i due ultimi metodi supportano i namespace.
   È presente un metodo che notifica la presenza di character data:
   characters(char ch[], int start, int length)

   È anche disponibile un metodo che notifica la presenza di whitespace ignorabili nell'element content:
   ignorableWhitespace(char ch[], int start, int length)

2. **DTDHandler:** metodi per ricevere notifiche su entità esterne al documento e loro notazione dichiarata in DTD.
3. **ErrorHandler:** metodi per gestire gli errori e i warning nell'elaborazione di un documento.
4. **EntityResolver:** metodi per personalizzare l'elaborazione di riferimenti a entità esterne.Se un'interfaccia non viene implementata, il comportamento di default è ignorare l'evento.

Interfacce Application-to-Parser

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

Interfacce ausiliarieAttributes: metodi per accedere a una lista di attributi.Locator: metodi per individuare l'origine degli eventi nel parsing dei documenti (p.e. systemID, numeri di linea e di colonna ecc.).