

XML & XSLT

(avec xsd , svg, ...)

Table des matières

I - XML (présentation générale).....	3
1. Présentation de XML & XSLT.....	3
2. XML - Limitations/précautions/conseils.....	10
3. XML face aux autres formats de données.....	11
II - Document Xml bien formé (syntaxiquement).....	12
1. Structure d'un document XML.....	12
2. Entités internes et externes.....	15
III - Namespace XML.....	16
1. Objectifs des namespaces xml.....	16
2. Syntaxes.....	17
3. Sémantique des namespaces.....	18

IV - Schéma XML / Validation.....	20
1. Présentation des schémas W3C.....	20
2. Lien entre un document XML et un schéma.....	21
3. Structure d'un schéma.....	22
4. Aspects avancés des schémas.....	29
V - Quelques dérivés d'XML (SVG, MathML, ...).	35
1. Présentation & affichage.....	35
2. Applications diverses (...).	39
3. E.A.I.....	40
4. Services WEB.....	40
VI - XPath et XQuery.....	41
1. XPath.....	41
2. XQuery.....	44
VII - XSLT et XSL-FO.....	50
1. Présentation de XSLT.....	50
2. Applications des templates.....	52
VIII - "Parsing XML" (Stratégies/API).....	60
1. Différents types d'analyse.....	60
2. DOM (Document Object Model).....	61
IX - Annexe – Entités XML et DTD (Has been).....	67
1. Entités XML.....	67
2. Documents validés via une DTD.....	70
3. Lien entre la DTD et le document XML.....	71
4. Structure d'une DTD.....	73
X - Annexe – Bibliographie, Liens WEB + TP.....	78
1. Bibliographie et liens vers sites "internet".....	78
2. TP.....	78

I - XML (présentation générale)

1. Présentation de XML & XSLT

1.1. XML

XML signifiant *eXtended Markup Language* est un **langage standardisé (w3c)** permettant d'**encoder des données structurées** de *manière hiérarchique*.

XML est **eXtensible** car il permet (contrairement à HTML) d'utiliser des noms de balises quelconques (ex: chapitre , region ,) .

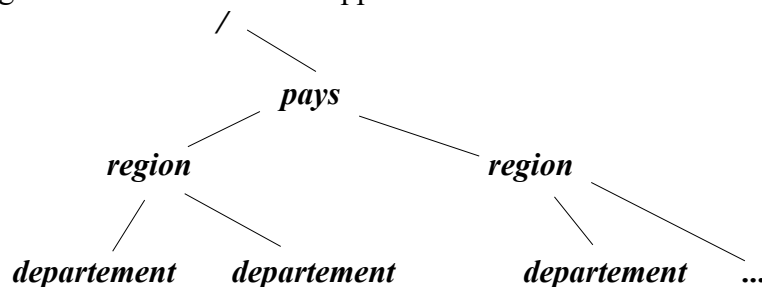
Exemple:

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<?xml-stylesheet type="text/xslt" href='fic_style.xslt' ?>
<!-- commentaire -->
<pays nom='France' capitale='Paris' >
  <region nom='Haute-Normandie' >
    <departement nom='Eure' num='27' prefecture='Evreux' />
    <departement nom='Seine Maritime' num='76' prefecture='Rouen' />
  </region>
  <region nom='Picardie'>
    <departement nom='Oise' num='60' prefecture='Beauvais' /> ...
  </region> ...
</pays>
```

Remarques:

Ce **document xml** comporte des **données** dont **les significations** sont **clairement indiquées** par les **attributs** ou par les **noms des balises englobantes**.

La **structure** globale de ce document s'apparente à celle de l'**arbre** suivant:



Aucune mise en forme (grs, italique , couleur , alignement, ...) n'apparaît directement dans le fichier xml. Par contre l'**application d'une feuille de style externe (CSS ou XSL)** permet d'obtenir facilement une présentation de ce genre:

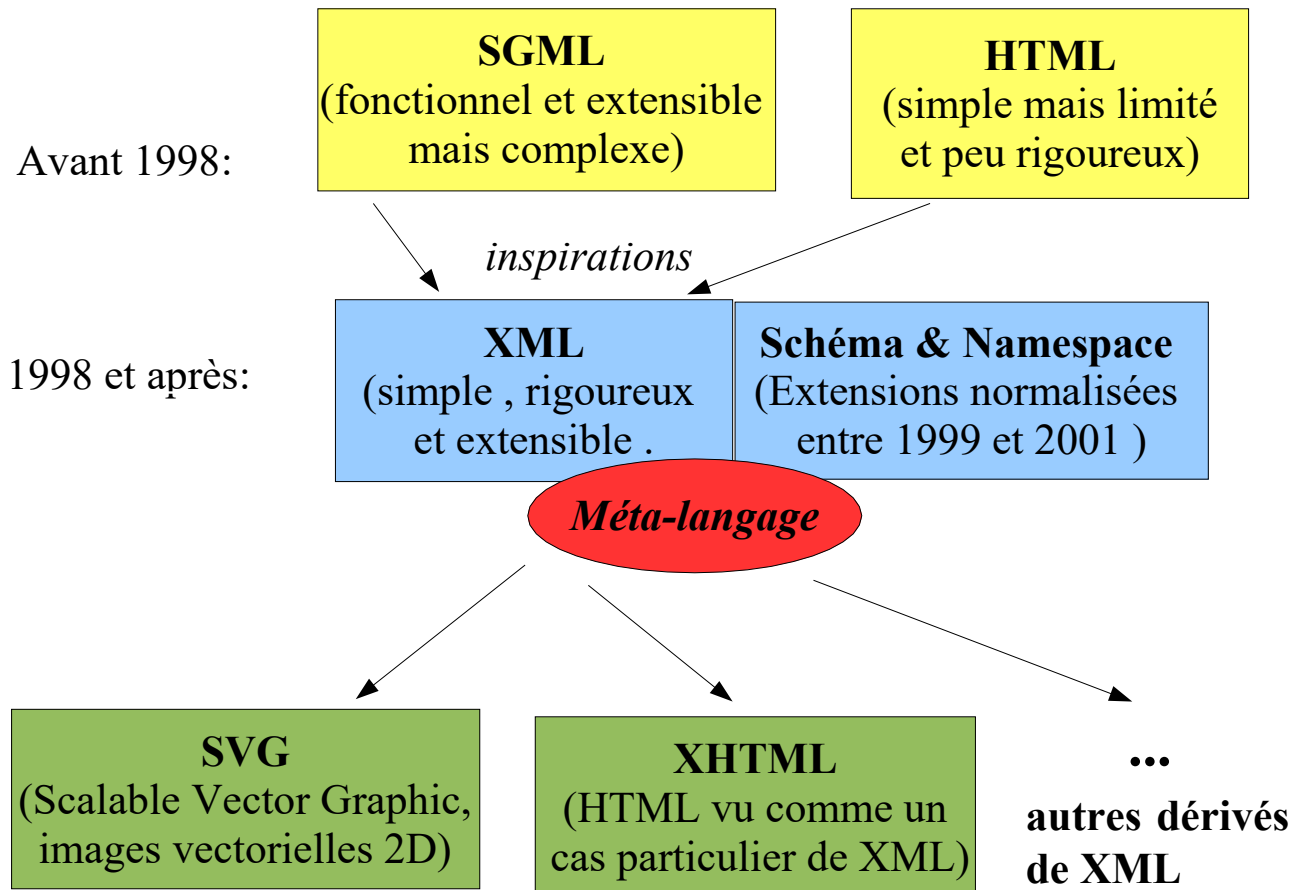
France

Haute-Normandie

num	departement	prefecture
27	Eure	Evreux
76	Seine Maritime	Rouen

1.2. XML: Historique et évolution

XML – historique & évolution



Principales technologies :

DTD (Document Type Definition)	Dès 1998	Mécanisme permettant de valider un document en comparant la structure de celui par rapport à une structure type (arborescence imposée de balises).
Namespace XML	Depuis	Identifiant de langage dérivé d'XML (porté par l'attribut spécial xmlns). Les préfixes locaux associés permettent d'encoder des documents composés dont les différentes parties sont exprimées avec des langages différents.

<i>Schéma XML (xsd)</i>	Depuis 2001	Mécanisme de validation plus évolués que les DTD. Les <u>2 principaux apports</u> sont : * syntaxe basée sur XML (==> homogène) * typage fin des données (string,integer, ...)
<i>XHTML</i>	Depuis	HTML vu comme un cas particulier de XML. XHTML 1.0 est le successeur officiel de HTML 4.

<i>SVG (Scalable Vector Graphic)</i>	Depuis	Langage permettant d'encoder des images vectorielles en XML (balises <i>line</i> , <i>circle</i> , ... avec <i>coordonnées</i>) .
<i>XSLT</i>	Depuis fin 1999	Partie Transformation des feuilles de styles XSL . Permet entre autre de transformer un arbre XML (document sans mise en forme) en un arbre (X)HTML .

XML a été normalisé en 1998 par le *World Wide Web Consortium (W3C)* .

Le site web de référence (pour consulter les normes officielles) est www.w3.org

1.3. Principales caractéristiques d'XML

XML – principales caractéristiques

XML est un **META-LANGAGE** à partir duquel on peut *dériver* tout un tas de *langages particuliers (XHTML, SVG, MathML, ...)*.

Les langages dérivés d'XML (et normalisés) sont généralement *identifiés par des namespaces* (ex: <http://www.w3.org/1999/xhtml>) .

- C'est un **LANGAGE DE DESCRIPTION DE DONNEES** comportant non seulement des **valeurs textuelles** mais également **les sémantiques de celles-ci** . Les **significations des données** sont renseignées au niveau des *noms de balises* d'encadrement (ou bien au niveau des noms de certains attributs).

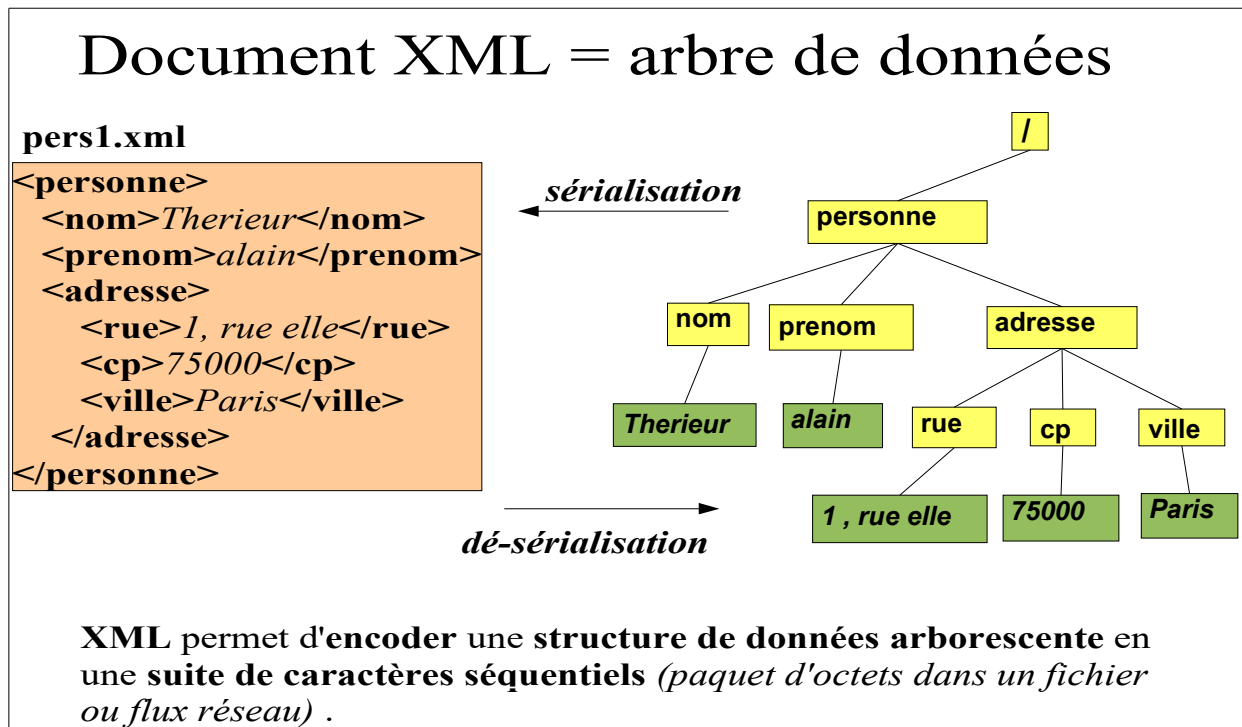
- *Simple à encoder et à ré-interpréter* sur tout environnement (*Unix, Windows, ...*) et depuis n'importe-quel langage (*C/C++, Java, Perl/Php, ...*) , XML est une technologie clef permettant de garantir une bonne **interopérabilité** entre différents systèmes informatiques.

(X)HTML est un plutôt un langage de présentation (pour affichage à l'écran) .

XML est plutôt un **langage de communication entre différentes applications** (des re-traitement

sont presque toujours nécessaires).

XML est une **technologie transverse** que l'on retrouve partout (Présentation via feuilles de styles, Echange de données , Services WEB , B2C, B2B , Bureautique , ...).



1.4. Namespaces XML

Namespace XML et documents composés

Un **namespace XML** est un **identificateur de langage dérivé d'XML**

Cet ID prend généralement la forme d'un **URI** (*Exemple:*

'<http://www.yyy.com/Année/Norme>') [sans téléchargement].

Les **namespaces** permettent d'encoder des **documents XML composés** dont les différentes parties sont exprimées dans des langages distincts .

```

<h:html xmlns:h='http://www.w3.org/1999/xhtml'
  xmlns:b='http://www.yyy.com/2004/biblio' >
  <h:head>
    <h:title> bibliographie sur XML </h:title>
  </h:head>
  <h:body>
    <h:p> <b:title> XML et XSLT </b:title> </h:p>
  </h:body>
</h:html>

```

*Différentes
interprétations*

Syntaxe générale: **xmlns:prefixe_local='nom_du_namespace'**

1.5. Documents XML bien formés

XML – documents bien formés

Un document XML est dit « **bien formé** » s'il est **syntactiquement correct** ; il doit pour cela vérifier les principales règles suivantes:

- **Toute balise ouverte doit être fermée:**

~~<p> paragraphe~~ mais <p> paragraphe </p>
 ligne ~~
~~ mais ligne
</br> ou ligne

 sachant que `<elt_vide a1='v1' ></elt_vide> <==> <elt_vide a1='v1' />`

- **Une seule balise de premier niveau englobant toutes les autres:**

~~<pers>....</pers> <pers> </pers>~~ mais
 <liste_pers><pers>....</pers> <pers>...</pers></liste_pers>

- **Pas de chevauchement de balises mais des imbrications claires:**

~~<i><g>blabla</i></g>~~ mais <i><g>blabla</g></i>

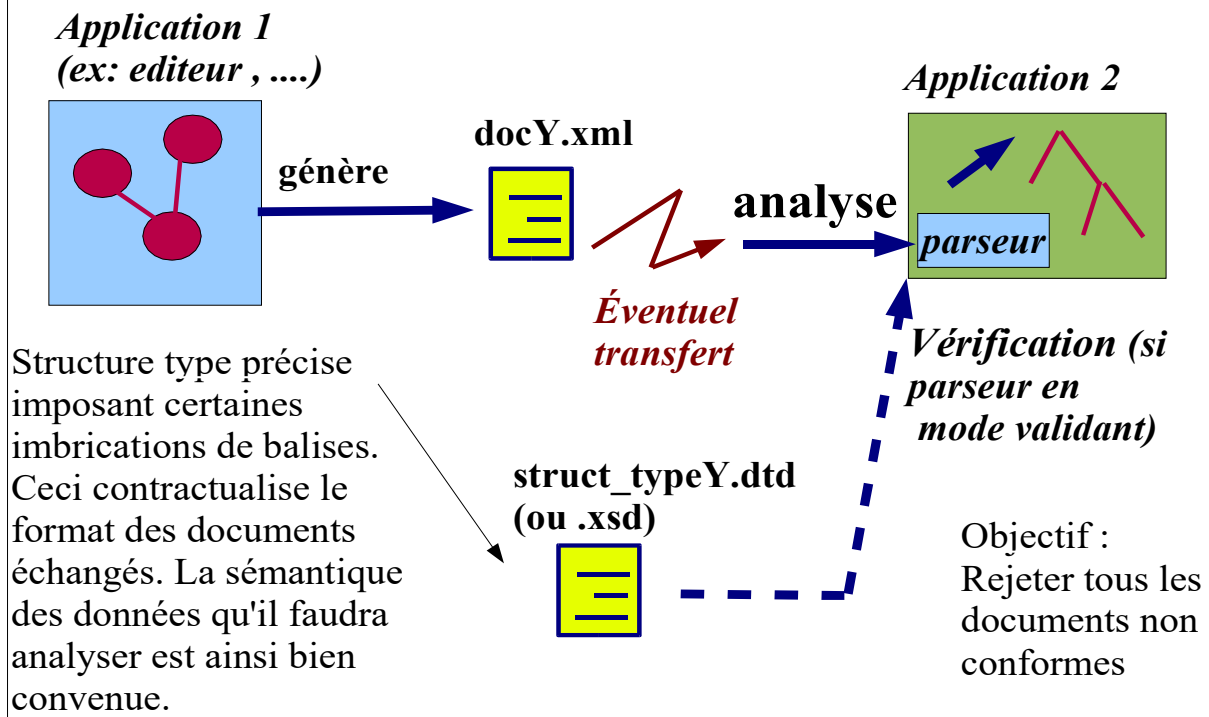
- **Les valeurs des attributs sont obligatoirement entre simple ou double quote:**

~~<pers age=35>...~~ mais <pers age='35'>... ou <pers age="35">...

NB: Un document XML est par défaut encodé avec le jeu de caractères **UTF-8**.

1.6. Documents valides

XML – Validation (via DTD ou Schema)



1.7. DOCTYPE (DTD)

DOCTYPE (entités & validation)

personne.dtd

```

<!ELEMENT personne (titre?,nom,prenom+,adresse*) >
<!ATTLIST personne age CDATA #IMPLIED>
<!ELEMENT titre (#PCDATA) >
<!ELEMENT nom (#PCDATA) >
<!ELEMENT prenom (#PCDATA) >
...

```

VERIFICATION:
conforme à la structure
type attendue ?

personne.xml

```

<?xml version='1.0'?>
<!DOCTYPE personne SYSTEM "personne.dtd"
[
  <!ENTITY Mr "Monsieur" >
  <!ENTITY adr SYSTEM "./adresse.xml" >
]>
<personne age="35">
  <titre>&Mr;</titre><nom>Defrance</nom><prenom>Didier</prenom>
  &adr;
</personne>

```

ENTITY
==> jeux de
remplacements

1.8. Schéma W3C

Schéma XML (.xsd) plus évolués que DTD

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name='pays' type='PaysType' />
  <xsd:complexType name="PaysType">
    <xsd:sequence>
      <xsd:element name="region" type="RegionType" maxOccurs='22' />
    </xsd:sequence>
    <xsd:attribute name="nom" type="xsd:string"/>
    <xsd:attribute name="capitale" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="RegionType">...</xsd:complexType>
  <xsd:complexType name="DepartementType">
    <xsd:attribute name="nom" type="xsd:string"/>
    <xsd:attribute name="num" type="xsd:positiveInteger"/> ....
  </xsd:complexType>
</xsd:schema>

```

france.xsd

Types précis

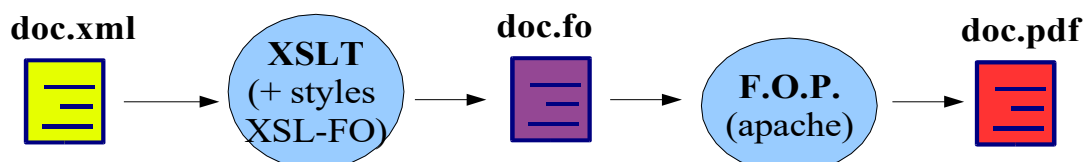
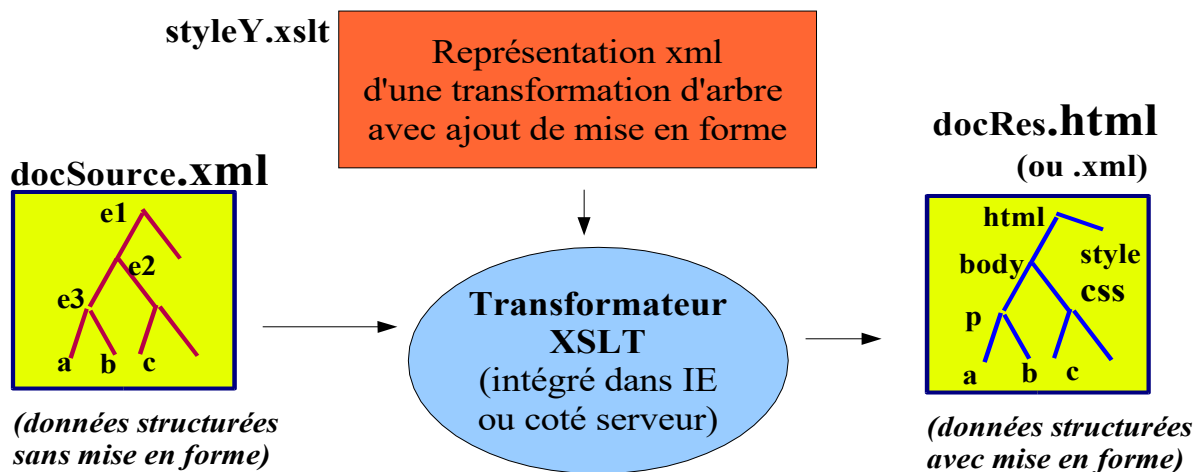
```

<pays xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="france.xsd"
  nom='France' capitale='Paris' >
  <region nom='Haute-Normandie' > <departement nom='Eure' num='27' ... />
...</region> ... </pays>

```

1.9. Feuilles de styles (CSS & XSL)

XML & feuilles de styles (CSS, XSL)



2. XML - Limitations/précautions/conseils

XML – quelques limitations

Nécessitant quelques traitements non négligeables (vérifications , analyse de texte) pour être ré-interprété , un document XML ne peut être rapidement traité dans sa globalité que s'il n'est pas trop volumineux.

Certains parseurs (dont XML-DOM normalisé par le W3C) créent des arbres de données en mémoire. Il faut veiller à ce que ceux-ci ne deviennent pas énormes !

==> **Attention aux performances !!!**

==> **préférer une collection de petits documents à un gros document.**

XML – documents fédérateurs (chapeaux)

fichier_global.xml

```
<ensemble>
  <texte>blabla</texte>
  <img
    src= 'image1.jpeg' />
  ...
  <sound
    href='musique1.xxx' />
  ...
</ensemble>
```

Données de haut niveau
en XML (assemblage,
synchronisation ,,
méta-données, ...)

image1.jpeg
(binaire)

musique1.xxx
(fichier binaire)

**autre(s)
fichier(s)
binaires(s)**

Entités externes
(fichiers annexes)
NON XML

3. XML face aux autres formats de données

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

Le format JSON (beaucoup plus simple , plus efficace) est devenu un concurrent très sérieux de XML en tant que format pour l'interopérabilité .

Le format de données JSON est plus compact (moins verbeux) que XML.

JSON est également plus rapide à encoder et re-décoder .

D'autre part , certaines technologies des années 2000-2012 ont exagéré sur l'utilisation XML (fichiers trop longs , trop complexes, ...) .

En 2020, 2021, ... , XML est à considérer comme une technologie "dèjà ancienne" et qui n'évolue quasiment plus.

Cependant , beaucoup d'existants sont basés sur XML et XML fait donc partie des incontournables à connaître et maîtriser un minimum .

II - Document Xml bien formé (syntaxiquement)

1. Structure d'un document XML

1.1. Prologue facultatif

3 versions possibles (selon l'implicite des valeurs par défaut):

```
<?xml version="1.0" ?>
```

ou bien

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

ou bien

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

NB1: l'attribut **encoding** permet de préciser le jeu de caractères qui sera utilisé pour encoder les morceaux de texte du document . **UTF-8** est la valeur par défaut pour un fichier XML.

Le jeu de caractères **UTF-8** est issu d'une transformation appliquée sur les caractères "unicode" (UCS-2 / UCS-4) . Un caractère UTF-8 est ainsi codé sur un nombre variable d'octets (de 1 à 6). **Tous les caractères de base de la table ASCII sont codés de la même façon en UTF-8 c'est à dire sur un seul octet** . D'autres caractères (é, è, à, , japonais , arabes , ...) sont quelquefois codés sur 2 octets ou plus .

L'encodage ISO-Latin "**ISO-8859-1**" permet d'encoder "tel quel" certains caractères accentués (é , è, à , ù) mais ne sera pas toujours disponible au niveau du logiciel qui devra lire et interpréter le fichier XML .

NB2: L'attribut **standalone** (vaut "**no**" par défaut) . il n'a de signification qu'avec une DTD et si standalone vaut "yes" cela signifie que la DTD n'est utilisée que pour la validation (et pas pour des entités ou attributs par défaut)

NB3: Une **instruction de traitement (processing instruction)** et un commentaire quelconque sont souvent présents en fin de prologue.

Une référence à une feuille de style est l'exemple le plus courant d'instruction de traitement:

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="styleFun.xslt" ?>
<!-- commentaire XML -->
```

1.2. Arbre d'éléments

Synonymes à connaître dans le monde XML : Tag = Balise = Markup = Element

Un document XML comporte obligatoirement une seule balise de premier niveau englobant toutes les autres . Il y a un seul point d'entrée dans le document :

```
<livre> .... </livre>
<livre>.... </livre>
```

est un contenu impossible dans un fichier XML
et doit être ré-écrit avec une nouvelle balise englobante :

```
<liste-livre>
  <livre> .... </livre>
  <livre>.... </livre>
</liste-livre>
```

Toute balise ouverte doit absolument être ultérieurement fermée .

Les balises doivent être proprement imbriquées les unes dans les autres mais ne doivent surtout pas se chevaucher:

~~<i>xxxxxx</i>~~ est par exemple **interdit**.

Une syntaxe spéciale peut être utilisée pour les **éléments vides** (ne comportant aucun texte entre l'ouverture et la fermeture de la balise) :

</br> peut s'écrire
 <!-- le / final permet d'auto-fermer la balise -->

Une balise vide peut éventuellement comporter des attributs:

```
<conf>
  <parameter name="couleur" value="red" />
</conf>
```

1.3. Règles syntaxiques

Le langage XML fait la différence entre les *minuscules* et les *MAJUSCULES* ==> de la rigueur s'impose !!!

Le *nom d'une balise* doit respecter les *contraintes suivantes*:

- pas de blancs (<Nom Societe> interdit)
- pas de chiffre en première position (<007-Bond> interdit)
- pas de sous-chaîne "xml" , "Xml" ou "XML" au début (<Xml_MaBalise> interdit)
- pas de tiret ou de point en première position (< X.a> et <X-a> autorisés)

Sections littérales (non interprétées par XML)

<![CDATA[texte non interprété **<BaliseNonInterprétée> ...fin]]>**

1.4. Attributs

Suivant les mêmes contraintes lexicographiques que les balises, les **attributs** (**nom="valeur"** ou bien **nom='valeur'**) sont très pratiques pour ajouter quelques caractéristiques sur certaines données.

....

Attribut prédéfini **xml:lang** (indication de la langue)

exemples:

`<p xml:lang = 'fr'> texte francais </p>`

`<p xml:lang = 'en-GB'> colour </p>`

`<p xml:lang = 'en-US'> color </p>`

Attribut prédéfini **xml:space** (gestion des caractères blancs)

L'attribut **xml:space** (ayant la valeur '**default**' par défaut) fait que les caractères blancs avant ou après un texte sont ignorés.

S'il l'on souhaite qu'un parseur tienne compte de tous les caractères blancs présents dans une portion du document XML, il faut fixer la valeur de l'attribut **xml:space** à "**preserve**".

NB: Les valeurs des attributs **xml:lang** et **xml:space** sont héritées par les éléments fils de l'élément auquel ils sont appliqués, tant qu'il n'y a pas de modifications explicites.

1.5. Caractères spéciaux

Certains caractères spéciaux nécessitent l'utilisation d'entités lorsqu'ils doivent être inclus dans les données textuelles d'un élément.

Caractères spéciaux	&	<	>	"	'
entités d'encodage	&amp;	&lt;	&gt;	&quot;	&apos;

Exemple :

`<P> un est < trois et > deux </P>`

INCORRECT

`<P> un est < trois et > deux </P>`

CORRECT

Tous les autres caractères (é,@,...) ne peuvent être universellement encodés que via leurs valeurs numériques UNICODE :

& <==> **&** <==> **&**
 décimal hexadécimal

Quelques exemples:

<i>A</i>	&#0065;		€	&#x20AC;
&	&#0038;		£	&#x00A3;
@	&#0064;		©	&#x00A9;
~	&#0126;		∞	&#x2219;
à	&#x00E0;		Σ	&#x2211;
é	&#x00E9;		½	&#x00BD;
è	&#x00E8;		«	&#x00AB;
ê	&#x00EA;		»	&#x00BB;

==> consulter une table des caractères UNICODE pour connaître les codes numériques des autres caractères.

NB: ** ** est l'équivalent XML du * * de HTML (*non-break space*) .
** ** permet ainsi d'obtenir **3 caractères blancs consécutifs** .

2. Entités internes et externes

- Jeux de remplacements (texte , abréviations)
- inclusion de sous fichiers XML

--> intéressant mais assez lié à la technologie DTD
 Compatibilité avec schéma / XSD et interprétation à tester au cas par cas !!:

--> voir l'éventuelle annexe sur le sujet pour approfondir (si besoin).

III - Namespace XML

1. Objectifs des namespaces xml

De façon à :

- **identifier** sans ambiguïté un **langage dérivé d'XML** (tel que XHTML , SVG , ...)
- autoriser la **coexistence de plusieurs parties exprimées dans des langages différents** au sein d'un même **document composé**

un "namespace XML" (*espace de noms "XML"*) est une **information complémentaire** qui va **qualifier le nom d'une balise ou d'un attribut** (de la même manière qu'un *préfixe*).

Quelques analogies:

- espace de noms C++
- package JAVA
- package UML

Namespace XML et documents composés

Un **namespace XML** est un **identificateur de langage dérivé d'XML**. Cet **ID** prend généralement la forme d'un **URI** (*Exemple:*

'<http://www.yyy.com/Année/Norme>') [sans téléchargement].

Les **namespaces** permettent d'encoder des **documents XML composés** dont les différentes parties sont exprimées dans des langages distincts .

```
<h:html xmlns:h='http://www.w3.org/1999/xhtml'
        xmlns:b='http://www.yyy.com/2004/biblio' >
  <h:head>
    <h:title> bibliographie sur XML </h:title>
  </h:head>
  <h:body>
    <h:p> <b:title> XML et XSLT </b:title> </h:p>
  </h:body>
</h:html>
```

Différentes
interprétations

Syntaxe générale: `xmlns:prefixe_local='nom_du_namespace'`

2. Syntaxes

2.1. Association entre préfixe local et nom de namespace

Pour avoir une bonne chance d'être unique et bien compréhensible, un nom de namespace doit être relativement long .

Un namespace long directement utilisé en tant que préfixe aurait pour conséquence d'augmenter considérablement la taille des fichiers:

~~<nom_namespace_xx:balise_yy></nom_namespace_xx:balise_yy>~~

La solution qui a été retenue consiste à **associer un préfixe local très court à un nom relativement long de namespace** :

```
<prefix1:elementEnglobant xmlns:prefix1 = "nom_du_namespace_1" >
  <prefix1:sousElement ...>
    ...
```

L'association entre un préfixe local et un nom de namespace s'effectue via l'attribut spécial **xmlns** (signifiant "*XML NameSpace*") présent au niveau d'une balise englobante (qui est généralement la balise de premier niveau du document). Ce préfixe sera alors valable dans tous les sous éléments.

2.2. Exemple classique (avec préfixes)

Des préfixes locaux différents (associés à des namespaces différents) autorisent la construction de document alternant à loisir des éléments provenant de différents langages:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format" >

<xsl:template match="/" >
...
  <xsl:for-each select="...">
    <fo:block color='red' font-size='12' ...>
      <xsl:value-of select="title" />
    </fo:block>
  </xsl:for-each>
...
</xsl:template>

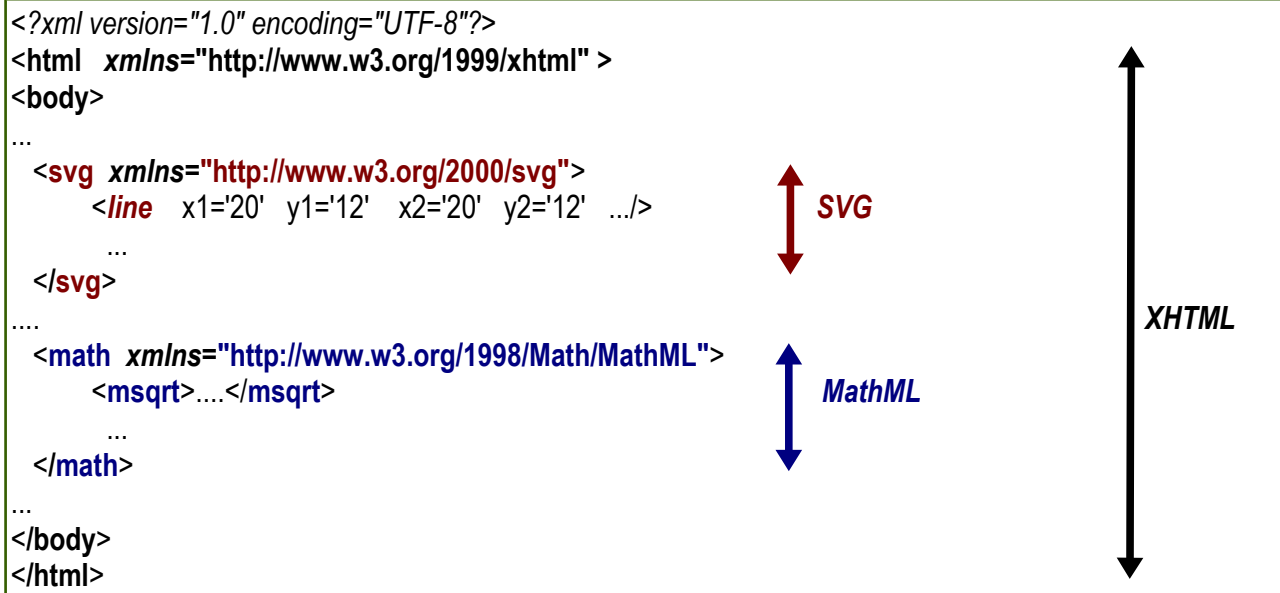
</xsl:stylesheet>
```

(NB: cet exemple utilisant *XSLT* et *XSL-FO* est la trame d'une génération de *pdf* via *FOP*).

2.3. Exemple plus rare (sans préfixe) ==> Namespace par défaut

Si les zones du document qui sont exprimées dans différents langages sont clairement séparées, on peut éventuellement ne **pas mentionner de préfixe local** => Un **namespace par défaut** est valable dans tous les sous-éléments tant que celui-ci n'a pas été ré-explicité :

Exemple à tester avec *amaya.html*



3. Sémantique des namespaces

3.1. Noms des namespaces

Un **nom de namespace** est une simple **chaîne de caractères** qui est sensée **identifier un vocabulaire XML** (un ensemble de balises XML ayant une signification bien précise) .

Il s'agit d'un(e) **URI (Uniform Resource Identifier)** et non pas d'une URL (Uniform Resource Locator). Autrement dit , **il n'y a aucun accès réseau ni aucun téléchargement** .

En règle générale , un nom de namespace prend l'une des 2 formes suivantes:

- *urn:un_nom_de_namespace*
- *http://www.yyy.com/AnneeNorme/PartieFinaleDuNomDuNamespace*

L'intérêt d'une **URI** commençant par un nom de domaine du genre "*http://www.yyy.com*" est de ne normalement pas entrer en conflit avec une autre URI provenant d'une autre entreprise ou d'un autre organisme.

Quelques **namespaces** associés à des **langages normalisés** (et dérivés d'XML):

http://www.w3.org/1999/xhtml	XHTML (successeur de HTML 4)
http://www.w3.org/2000/svg	SVG (Scalable Vector Graphics)
http://www.w3.org/2001/XMLSchema	Schémas W3C (.xsd) / alternative aux DTD
http://www.w3.org/1999/XSL/Transform	XSLT (Transformations)
http://www.w3.org/1999/XSL/Format	XSL-FO (Mise en forme "fo" ==> pdf)
...	

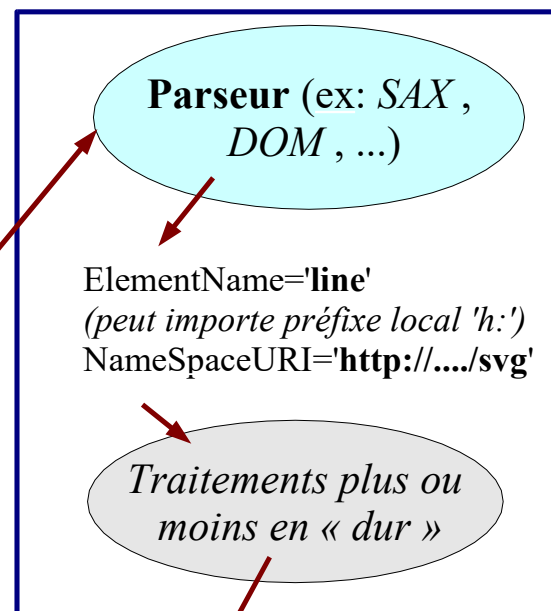
3.2. Interprétation des namespaces

Interprétation des namespaces

FichierComposé.xml

```
<h:html xmlns:h='.../xhtml'
        xmlns:s='.../svg'>
<h:header>
  <h:title> Titre page </h:title>
</h:header>
<h:body>
  <h:h3> titre qui va bien </h:h3>
  <s:svg>
    <s:line x1='12' y1='13'
            x2='50' y2='68' />
  </s:svg>
</h:body>
</h:html>
```

Application (ex: Amaya)



Action : dessiner une ligne à l'écran

Le préfixe local ("s:" ou "svg:") n'a aucune importance. Par contre le **namespaceURI** sera rigoureusement comparé à ce qu'attendait l'application pour déclencher tel ou tel traitement.

IV - Schéma XML / Validation

1. Présentation des schémas W3C

Les **schémas W3C** constituent une **alternative aux DTD**.

Cette technologie proposée à l'origine par Microsoft, a été normalisée (après de nombreux ajustements) par l'organisme W3C en 2001 (soit assez longtemps après les DTD).

Beaucoup de logiciels récents utilisent aujourd'hui les schémas (ex: J2EE 1.4, JEE 5 & 6) alors que d'anciens programmes ou d'anciennes versions utilisaient les DTD (ex: J2EE 1.3).

Les schémas W3C offrent les principaux avantages suivants:

- **syntaxe homogène vis à vis d'XML** (un schéma est codé comme un cas particulier de fichier XML)
- On peut attacher des **types de données très précis** (string, integer, date, ...) aux éléments et aux attributs
- Certains aspects des schémas sont assez proches des concepts objets.

Schéma XML (.xsd) plus évolués que DTD

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name='pays' type='PaysType' />
  <xsd:complexType name="PaysType">
    <xsd:sequence>
      <xsd:element name="region" type="RegionType" maxOccurs="22" />
    </xsd:sequence>
    <xsd:attribute name="nom" type="xsd:string"/>
    <xsd:attribute name="capitale" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="RegionType">...</xsd:complexType>
  <xsd:complexType name="DepartementType">
    <xsd:attribute name="nom" type="xsd:string"/>
    <xsd:attribute name="num" type="xsd:positiveInteger"/> ....
  </xsd:complexType>
</xsd:schema>
```

france.xsd

*Types
précis*

```
<pays xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="france.xsd"
  nom="France" capitale="Paris" >
  <region nom="Haute-Normandie" > <departement nom="Eure" num='27' ... />
...</region> ... </pays>
```

2. Lien entre un document XML et un schéma

Un **schéma** correspond à un fichier ayant l'extension **xsd** (*Xml Schema Definition*).

Un tel fichier (.xsd) décrit la structure d'une **classe de documents** .

Un **fichier XML conforme au schéma** sera considéré comme une **instance** .

De façon à associer un schéma à un document XML, on aura recours à l'une des deux syntaxes suivantes:

```
<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schema_yyy.xsd" >
...
</rootElement>
```

ou bien

```
<p:rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.yyy.com/namespaceYYY schema_yyy.xsd"
  xmlns:p="http://www.yyy.com/namespaceYYY">
...
</p:rootElement>
```

La localisation d'un schéma (.xsd) s'introduit donc via de nouveaux attributs que l'on rattache à la balise de premier niveau .

L'attribut "**schemaLocation**" (ou bien "**noNamespaceSchemaLocation**") sera pris en compte par **parseur en mode validant** (qui de plus doit être *paramétré pour tenir compte des schémas W3C*).

Pour effectuer rapidement quelques tests de conformité, on pourra utiliser un des outils suivants:

- **XML-Spy** (sous Windows) (*bon produit commercial avec version d'évaluation*)
- **XSD** (Xml Schema Validator) (*freeware en mode texte*)
- **ValidApp** (petit programme utilisant JAXP à écrire en Java)
- **Plugin eclipse**
- ...

3. Structure d'un schéma

3.1. Terminologie (types simples et complexes)

Au sein de la définition d'un schéma, on distingue:

- les **types complexes** correspondant à des éléments qui ont *au moins un sous élément ou bien un attribut*.
- les **types simples** correspondant à des *valeurs atomiques* (qui ne se décomposent pas).

Beaucoup de **types simples** sont **prédéfinis** (ex: `xsd:string`, `xsd:integer`, `xsd:decimal`, ...).

Ces *types simples* seront utilisés pour caractériser des *attributs* ou des *balises feuilles* (terminales) du genre "titre", "prenom", "age",

On peut en définir de nouveaux en précisant de nouvelles contraintes (motif à respecter pour une chaîne de caractères, plage de valeurs obligatoire [ex: `type_age = positiveInteger` ayant 150 comme valeur maximale])

Au sein d'un document XML, toute **balise xxx** qui comporte des sous balises doit clairement être associée à un **type complexe** (souvent nommé *xxxType*). A un arbre XML comportant *N niveaux de profondeur*, correspond généralement un schéma comportant *N types complexes principaux*.

Petite subtilité:

`<valeur>1200</valeur>` est considérée comme étant de **type simple**

`<prix monnaie="euro">1200</prix>` est considérée comme étant de **type complexe**

3.2. Structure globale d'un schéma (.xsd)

1. Un bloc `<annotation>` fait généralement office de commentaire et permet d'indiquer la raison d'être du schéma.
2. Figure ensuite la définition d'**un ou plusieurs éléments** (balises) que l'on peut potentiellement trouver en tant que **point d'entrée du document**.
On trouve également à ce **niveau global** des définitions de sous éléments récurrents qui seront plus tard référéncés via `<xsd:element ref="nom_element_global"/>`.
3. La plus grande partie du schéma est ensuite structurée par une série de définitions de types complexes (chaînés entre eux via `<xsd:element ... type="type_sous_élément"/>`).

3.3. Exemple complet (pour vue d'ensemble)

personne.xml

```
<?xml version='1.0'?>
<personne age="35"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="personne.xsd">
  <nom>Defrance</nom>
  <prenom>Didier</prenom>
  <adresse>
    <rue>10 rue Elle</rue>
    <cp>75000</cp>
    <ville>Paris</ville>
  </adresse>
</personne>
```

est conforme au schéma ci-après

personne.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>Schéma pour personne avec adresse</xsd:documentation>
  </xsd:annotation>

  <xsd:element name="personne" type="persType" />

  <xsd:complexType name="persType">
    <xsd:sequence>
      <xsd:element name="nom" type="xsd:string"/>
      <xsd:element name="prenom" type="xsd:string" maxOccurs="3" />
      <xsd:element name="adresse" type="adrType" />
    </xsd:sequence>
    <xsd:attribute name="age" type="myLittelInteger"/>
  </xsd:complexType>

  <xsd:complexType name="adrType">
    <xsd:sequence>
      <xsd:element name="rue" type="xsd:string"/>
      <xsd:element name="cp" type="xsd:string" />
      <xsd:element name="ville" type="xsd:string" />
      <xsd:element name="pays" type="xsd:string" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="myLittelInteger">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxInclusive value="149"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

3.4. Types simples prédéfinis

3.4.a. Principaux types prédéfinis:

xsd:string	Chaîne de caractères quelconque
xsd:integer	Nombre entier (ex: -126789, -1, 0, 1, 126789)
xsd:positiveInteger	Nombre entier positif ou nul (ex: 0, 1, 126789)
xsd:decimal	Nombre décimal (à virgule éventuelle) (ex: -1.23, 0, 123.4, 1000.00)
xsd:date	Date au format international / US (aaaa-mm-dd) (ex: 1999-05-31)
xsd:boolean	Booléen (ex: true , false , 0 , 1)
xsd:hexBinary	Valeur binaire codé en hexa (base 16) (ex: 0BF7)
xsd:long	Entier long
xsd:float , xsd:double	Nombre réel en simple précision (32bits) et double précision (64 bits) (ex: -INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN)
xsd:time	heure au format HH:MM:SS.mmm (ex: 13:20:00.000)
xsd:anyURI	URI (ou URL) quelconque (ex: http://www.yyy.com/page1.html#p1)
xsd:language	Code de langue (ex: en-GB , en-US , fr ,)
xsd:ID	Identificateur unique au sein d'un document
xsd:IDREF	Référence (renvoi) vers un ID

3.4.b. Autres types prédéfinis:

(xsd:) token , byte , normalizedString, unsignedByte , base64Binary, negativeInteger, short, int, unsignedInt, unsignedLong , unsignedShort , dateTime, duration, gMonth , gYear , gDay , Name , QName , NCName , ENTITY , NOTATION , ...

==> consulter la documentation de référence pour approfondir le sujet .

3.5. Types simples personnalisés

Ex1: (entier restreint à l'intervalle [1,99])

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="99"/>
  </xsd:restriction>
</xsd:simpleType>
```

Ex2:(chaîne de caractères devant se conformée à un certain format / expression régulière)

Format ici imposé : 2 chiffres , suivis d'un tiret , lui même suivi de 3 lettres majuscules.

```
<xsd:simpleType name="RefProdType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{2}-[A-Z]{3}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Ex3: (Enumération)

```
<xsd:simpleType name="Mois">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Janvier"/>
    <xsd:enumeration value="Février"/>
    <xsd:enumeration value="Mars"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Ex4: (TypeSimple "liste de ...")

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

validant un élément XML du type suivant:

```
<listOfMyInt> 20003 15037 95977 95945</listOfMyInt>
```

Voir la documentation de référence pour plus de détails.

3.6. Types complexes

3.6.a. Syntaxe générale (élément composé de sous élément(s) et d'attribut(s))

```
<xsd:complexType name="persType">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string" maxOccurs="3" />
    <xsd:element name="adresse" type="adrType" />
  </xsd:sequence>
  <xsd:attribute name="age" type="xsd:positiveInteger"/>
</xsd:complexType>
```

La **liste des sous éléments** doit absolument (si elle existe) être englobée par l'une des trois balises suivantes:

<xsd:sequence></xsd:sequence> impose un **ordre** pour les sous éléments.
 <xsd:all> ... </xsd:all> signifie tous les sous éléments dans **n'importe quel ordre** .
 <xsd:choice> </xsd:choice> signifie **un** sous élément **parmi les n** possibilités (*XOR*)

Les attributs **minOccurs** (ex: ="0") et **maxOccurs** (ex: ="unbounded") permettent d'indiquer le nombre d'exemplaires d'un sous élément (**0 à n** ,).

Par défaut, **minOccurs** et **maxOccurs** sont fixés à "1" .

3.6.b. Détails sur les attributs

Outre le fait de comporter un nom et un type , un attribut peut être caractérisé par:

- une valeur par défaut (à préciser via **default**="valeur_defaut" dans le schéma)
- une présence éventuellement obligatoire (à indiquer via **use**="required")

exemples:

```
<xsd:attribute name="prix" type="xsd:decimal" use="required" />
```

```
<xsd:attribute name="couleur" type="xsd:string" use="optional" default="noir" />
```

Remarques:

- Par défaut la valeur de **use** est fixée à "optional". **Tout attribut est donc par défaut facultatif.**
- Une application analysant un document XML via un parseur (ex: Jaxp/DOM) pourra récupérer une valeur par défaut dans le cas où un attribut n'a pas été renseigné.

3.7. Syntaxes possibles mais un peu moins lisibles

*Les éléments de syntaxe suivants peuvent parfois être pratiques.
Une utilisation abusive est toutefois fortement déconseillée.*

3.7.a. Type complexe sans nom et décrit d'une façon imbriquée

Lorsqu'une balise isolée est un type bien particulier qui a très peu de chance d'être ré-utilisé, on peut éventuellement définir le type (simple ou complexe) de celle-ci de manière imbriquée:

```
<element name="quantite">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="99"/>
    </xsd:restriction>
  </xsd:simpleType>
</element>
```

3.7.b. Référence vers un élément défini au niveau global

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>Schéma pour livre, chapitre et paragraphe</xsd:documentation>
  </xsd:annotation>

  <xsd:element name="livre" type="livreType" />
  <xsd:element name="titre" type="xsd:string" />

  <xsd:complexType name="livreType">
    <xsd:sequence>
      <xsd:element ref="titre" />
      <xsd:element name="chapitre" type="chapType" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="chapType">
    <xsd:sequence>
      <xsd:element ref="titre" />
      <xsd:element name="paragraphe" type="paraType" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
  ...
</xsd:schema>
```

3.8. Types complexes particuliers

3.8.a. Dériver un type complexe d'un type simple ...

... en lui ajoutant un attribut :

```
<xsd:element name="prix" type="prixInternational"/>

<xsd:complexType name="prixInternational">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="monnaie" type="xsd:string"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

==> `<prix monnaie="Euro" >12.5</prix>`

3.8.b. type complexe pour élément mixte

En précisant **mixed="true"** au niveau de la définition d'un type complexe, les éléments de ce type pourront comporter des blocs de texte à côté des sous éléments.

```
<xsd:element name="paragraphe" type="paraType"/>

<xsd:complexType name="paraType" mixed="true">
  <xsd:all>
    <xsd:element name="note" type="noteType" minOccurs="0" maxOccurs="unbounded" />
  </xsd:all>
</xsd:complexType>
```

==>

```
<paragraphe>
  Debut texte
  <note comment="important"> Texte de la note </note>
  Fin texte
</paragraphe>
```

3.8.c. type complexe pour élément vide

Un élément vide (sans texte ni sous balise) mais avec des éventuels attributs tel que

```
<param name="nom_param" value="valeur_param" />
```

... peut se schématiser de l'une des 2 façons suivantes:

```
<xsd:complexType>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
```

```
<xsd:complexType name="parameterType">
  <xsd:complexContent>
    <xsd:restriction base="xsd:anyType">
      <xsd:attribute name="name" type="xsd:string"/>
      <xsd:attribute name="value" type="xsd:string"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

4. Aspects avancés des schémas

4.1. Prise en compte des namespaces au sein d'un schéma

Exemple:

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

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:bi="http://www.yyy.com/my_namespace"
  targetNamespace="http://www.yyy.com/my_namespace"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified" >

  <xsd:annotation>
    <xsd:documentation>Base bibliographique avec namespace</xsd:documentation>
  </xsd:annotation>

  <xsd:element name="bibliographie" type="bi:bibliographieType"/>

  <xsd:element name="titre" type="xsd:string"/>

  <xsd:complexType name="bibliographieType">
    <xsd:sequence>
      <xsd:element ref="bi:titre"/>
      <xsd:element name="sujet" type="bi:sujetType" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="sujetType">
    <xsd:sequence>
      <xsd:element ref="bi:titre"/>
      <xsd:element name="livre" type="bi:livreType" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="livreType">
    <xsd:sequence>
      <xsd:element ref="bi:titre"/>
      <xsd:element name="auteur" type="xsd:string"/>
      <xsd:element name="editeur" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="prix" type="xsd:decimal"/>
    <xsd:attribute name="parution" type="xsd:date"/>
  </xsd:complexType>

</xsd:schema>
```

Au sein de l'exemple précédent, l'attribut **targetNamespace**="*http://www.yyy.com/my_namespace*" étant placé sur la balise englobante **<xsd:schema>**, toutes les parties imbriquées (*éléments*, *types complexes*, ...) qui sont habituellement de niveau global sont ici implicitement liées au namespace précisé par l'attribut **targetNamespace** hérité.

==> *il n'est donc pas nécessaire d'écrire <complexType name="bi:xxxType"> d'autant plus que cette écriture n'est pas autorisée au niveau d'une définition de type.*

Par contre, au sein d'une référence vers un type ou un élément, il faut préciser (via un préfixe local) le namespace de l'entité référencée (*ex: ref="bi:titre"*).

NB: Le préfixe local (*ex: bi*) utilisé pour encoder le schéma n'a aucune importance, il peut d'ailleurs être différent de celui qui sera utilisé pour encoder le document XML.

Les attributs **elementFormDefault**="*qualified*" et **attributeFormDefault**="*unqualified*" permettent respectivement de préciser que l'on souhaite trouver au sein des documents qui seront validés:

- des balises préfixées (avec des noms qualifiés)
- des attributs non préfixés (pas qualifiés)

Le fichier xml suivant est conforme au schéma précédent :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<b:bibliographie xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.yyy.com/my_namespace schema_biblio.xsd"
  xmlns:b="http://www.yyy.com/my_namespace" >

  <b:titre> Bibliographie sur certains langages de programmation </b:titre>
  <b:sujet>
    <b:titre>C++</b:titre>
    <b:livre prix="30.5">
      <b:titre>Mieux développer en C++</b:titre>
      <b:auteur>auteur francais</b:auteur>
      <b:editeur>Inter-Editions</b:editeur>
    </b:livre>
    <b:livre prix="50">
      <b:titre>Numerical recipes in c++</b:titre>
      <b:auteur>auteurs américains</b:auteur>
      <b:editeur>????</b:editeur>
    </b:livre>
  </b:sujet>
  ....
</b:bibliographie>
```

Le préfixe local choisi **b** (*et non pas bi*) n'a aucune importance. Ce qui compte c'est qu'il soit associé au bon namespace : celui précisé au niveau du schéma via **targetNamespace**.

4.2. 1.1.1 Notion de groupe

Un `<xsd:group>` est une *construction* constituée de plusieurs éléments (ou attributs) qui servira ultérieurement à définir une *partie* d'un (ou plusieurs) élément(s) complexe(s).

Contrairement à un Type complexe, **un groupe ne correspond pas à une basile XML**.

L'intérêt d'un regroupement est la réutilisation multiple d'une partie commune que l'on cherche à factoriser sans vraiment imposer une délimitation via un niveau de balise supplémentaire.

exemple1 (groupe d'éléments):

```
<xsd:complexType ...>
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="Adresses_livraison_et_facturation"/>
      <xsd:element name="adresse" type="AdresseType"/>
    </xsd:choice>
    <xsd:element name="xxx" type="..." />
  </xsd:sequence>
</xsd:complexType>
...
<xsd:group name="Adresses_livraison_et_facturation">
  <xsd:sequence>
    <xsd:element name="adresse_livraison" type="AdresseType"/>
    <xsd:element name="adresse_facturation" type="AdresseType"/>
  </xsd:sequence>
</xsd:group>
```

exemple 2 (groupe d'attributs):

```
<xsd:complexType ...>
  <xsd:sequence>
    ...
  </xsd:sequence>
<!-- attributeGroup replaces individual declarations -->
  <xsd:attributeGroup ref="Prix_et_Qte"/>
</xsd:complexType>
...
<xsd:attributeGroup name="Prix_et_Qte">
  <xsd:attribute name="prix" type="xsd:decimal" use="required"/>
  <xsd:attribute name="qte" type="xsd:integer"/>
</xsd:attributeGroup>
```

4.3. 1.1.1 Inclusion ou importation d'un schéma dans un autre

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>Schema biblio (englobant) </xsd:documentation>
  </xsd:annotation>
  <!-- inclusion du "sous-schéma" décrivant la structure d'un livre -->
  <!-- à placer absolument dès le début (avant element et complexType) -->
  <xsd:include schemaLocation="livre.xsd" />
  ...
</xsd:schema>
```

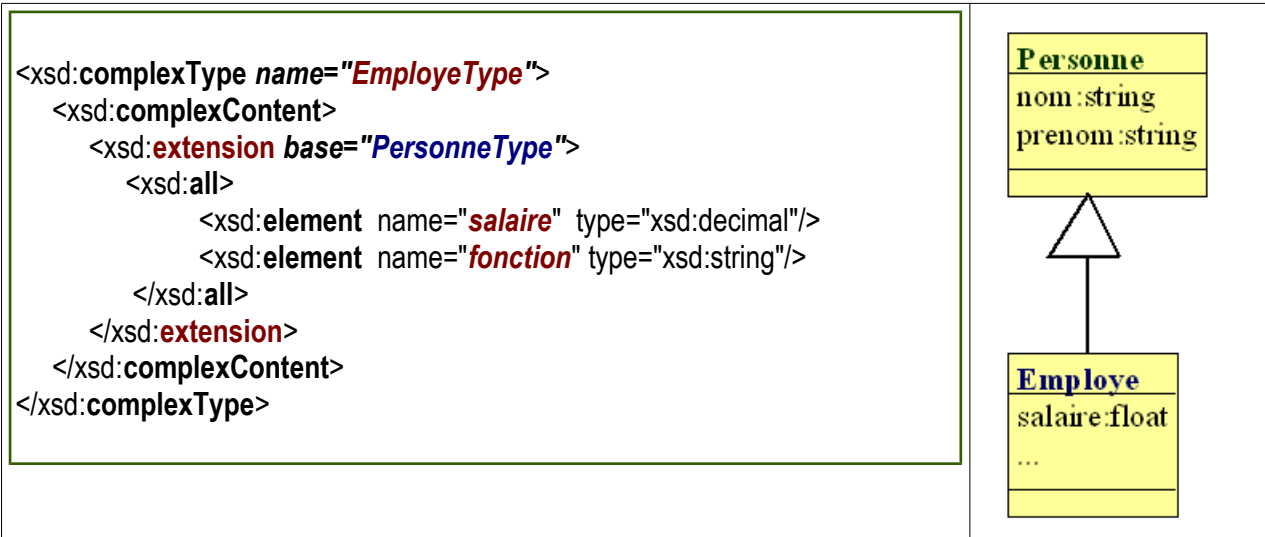
NB: à la place de `<xsd:include>` on utilise assez souvent

```
<xsd:import namespace="nomAutreNameSpace" schemaLocation="sousSchema.xsd" />
```

car `<xsd:import />` permet d'importer un sous-schéma qui peut être d'un autre namespace.

4.4. Structures de données avancées

4.4.a. Héritage entre types complexes (extensions)



```
<element name="personne" type="EmployeType" substitutionGroup="PersonneType" />
```

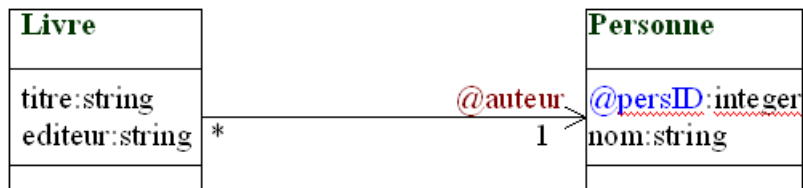
4.4.b. Références (liaisons)

On peut reproduire au sein d'un document XML des liaisons entre éléments qui sont très proches des jointures du modèle relationnel (FK ==> PK).

Afin d'atteindre cet objectif, il faudra:

- Préciser des informations de type "`<xsd:keyref>`" et "`<xsd:key>`" au sein du schéma de façon à encoder les *liaisons* "référence --> clef" et pour que **la phase de validation** puisse **vérifier les contraintes d'intégrités**.
- Qu'une application tienne compte de ces liaisons (via par exemple des requêtes XPath / XQuery)

Exemple:



schema_id_ref.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="biblio" type="biblioType">
  <!-- attribut id définit en tant que clef primaire sur personne -->
  <xsd:key name="PersonneKey">
    <xsd:selector xpath="personne" />
    <xsd:field xpath="@id" />
  </xsd:key>

  <!-- contrainte d'intégrité : livre/@auteur vers personne/@id -->
  <xsd:keyref name="LivreToPersonne" refer="PersonneKey">
    <xsd:selector xpath="livre" />
    <xsd:field xpath="@auteur" />
  </xsd:keyref>
</xsd:element>

<xsd:complexType name="biblioType">
  <xsd:sequence>
    <xsd:element name="livre" type="livreType" minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="personne" type="personneType" minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="livreType">
  <xsd:sequence>
    <xsd:element name="titre" type="xsd:string"/>
    <xsd:element name="editeur" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="auteur" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="personneType">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" />
</xsd:complexType>
</xsd:schema>
  
```

livre_auteur.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<biblio xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="schema_id_ref.xsd">

  <livre auteur="1">
    <titre>Livre A</titre>
    <editeur>Eyrolles</editeur>
  </livre>

  <livre auteur="1">
    <titre>Livre B</titre>
    <editeur>Masson</editeur>
  </livre>

  <livre auteur="2">
    <titre>Livre C</titre>
    <editeur>InterEditions</editeur>
  </livre>

  <personne id="1">
    <nom>durant</nom>
    <prenom>luc</prenom>
  </personne>

  <personne id="2">
    <nom>dupond</nom>
    <prenom>julie</prenom>
  </personne>

</biblio>

```

Si par erreur , *auteur="3"* ==>



The keyref '3' does not resolve to a key for the Identity Constraint 'PersonneKey'.

Remarque:

<xsd:key ...> permet de préciser un champ clef dont la valeur doit être unique pour un type donné (2 personnes ne peuvent pas avoir le même id).

<xsd:ID> (issu des **DTD**) permet de préciser un identificateur unique au sein d'un document (fichier) complet (une personne ne peut pas avoir le même ID qu'un éditeur) .

V - Quelques dérivés d'XML (SVG, MathML, ...)

1. Présentation & affichage

1.1. XHTML

XHTML (*HTML vu comme un cas particulier du rigoureux XML*) est le successeur officiel de HTML 4.01 .

Principales contraintes:

- toutes les balises doivent être fermées (`
 ==>
`) .
- les noms de balises doivent être écrites en minuscules
- les valeurs des attributs doivent être encadrées par des simples ou doubles quotes.

XHTML 1.0 existe en 3 variantes:

Stricte	version très stricte (structure du document dans le fichier XHTML , mise en forme dans les pages de styles)
Transitionnelle	version plus souple permettant de coder une grande partie de la mise en forme directement en HTML (sans obliger à utiliser des styles)
supportant les frames	version étendue autorisant le découpage du navigateur en différentes zones (frames).

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

Namespace associé à XHTML 1.0 : `xmlns="http://www.w3.org/1999/xhtml"`

Maintenant que HTML5 a remplacé HTML4 , XHTML est plutôt à considérer comme un HTML plus rigoureux où toutes les balises ouvertes doivent se fermer .

1.2. MathML

MathML permet d'encoder en XML la représentation d'équations mathématiques.

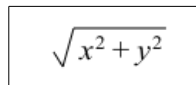
MathML 1.0 est une recommandation W3C du 7 juillet 1999

MathML 2.0 est une recommandation W3C du 21 Octobre 2003 .

Navigateurs internet supportant MathML:

- **Amaya** (navigateur expérimental du W3C)
- **Mozilla** (Firefox , Netscape7,)
- **IE** avec plugin "*mathPlayer*" ou "*techExplorer*" .

Exemple:



$$\sqrt{x^2 + y^2}$$

S'encode de la façon suivante:

ex.mml

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
    "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">

<math xmlns="http://www.w3.org/1998/Math/MathML">
  <msqrt>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <msup>
      <mi>y</mi>
      <mn>2</mn>
    </msup>
  </msqrt>
</math>
```

Ce fichier s'affiche à peu près correctement au sein de Mozilla Firefox .

1.3. SVG (Scalable Vector Graphics)

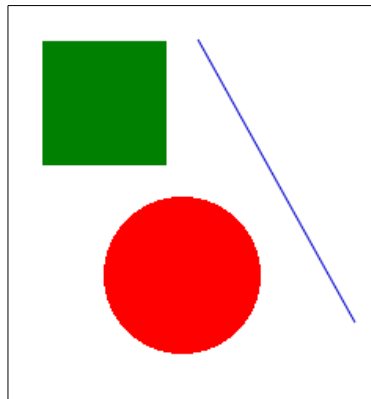
Signifiant *Scalable Vector Graphics* , SVG est un langage dérivé d'XML qui permet d'encoder des images vectorielles en 2D .

Correspondant au type MIME "*image/svg+xml*" , un fichier SVG peut maintenant être bien affiché par n'importe quel navigateur récent (ex: Firefox , Chrome , Edge , ...) car SVG fait partie du standard HTML5 .

Principales fonctionnalités de SVG:

- Coder des figures, schémas et diagrammes de façon assez compacte sans pour autant sacrifier la qualité (l'image reste nette quelque soit le facteur d'agrandissement).
- Simple à encoder par programmation (la génération d'un fichier SVG est quasiment aussi simple que celle d'un fichier texte).
- Possibilité d'animation simple en rajoutant du code JavaScript

Exemple:



peut être encodé de la façon suivante:

scene.svg

```
<svg xmlns="http://www.w3.org/2000/svg" width='500' height='350' >
  <rect x='50' y='50' width='80' height='80' style='fill:green' />
  <circle cx='140' cy='200' r='50' style='fill:red' />
  <line x1='150' y1='50' x2='250' y2='230' style='fill:blue;stroke: mediumblue;' />
</svg>
```

NB:

Une image SVG peut éventuellement être intégrée au sein d'une page HTML via l'une des façons suivantes:

```
<object align="middle" data="scene.svg"
  border="0" name="SVGEmbed3"
  width="300" height="300"
  type="image/svg+xml">
</object>
```

```

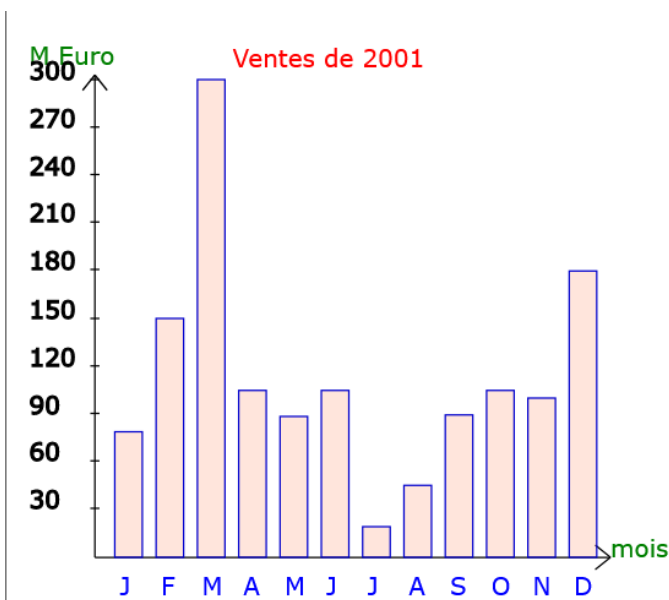
```

```
<svg xmlns="http://www.w3.org/2000/svg" width='300' height='350' >
  <rect x='50' y='50' width='80' height='80' style='fill:green' />
  <circle cx='140' cy='200' r='50' style='fill:red' />
  <line x1='150' y1='50' x2='250' y2='230' style='fill:blue;stroke: mediumblue;' />
</svg>
```

NB : le logiciel open source et gratuit "**inkscape**" permet de dessiner de jolies images vectorielles au format SVG .



NB : Etant simple à générer, beaucoup de technologies WEB "coté serveur" (ex : Servlet/JSP , ASP, PHP , python, ...) peuvent facilement générer de manière dynamique des fichiers **.svg** correspondant à des **graphiques** en fonction de valeurs puisées dans des bases de données .



2. Applications diverses (...)

2.1. Fichier de configuration encodés en XML

Les données d'un fichier de configuration sont souvent arborescentes et se prêtent très bien à un encodage au format XML.

Par exemple, tous les **fichiers de configuration** de la plate-forme **J2EE** sont au format XML .

exemple: WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app id="WebApp">
  <display-name>tp_web</display-name>

  <context-param>
    <param-name>PROPERTIES_FILENAME</param-name>
    <param-value>WEB-INF/fic1.properties</param-value>
  </context-param>

  <servlet>
    <servlet-name>B2cMvc2Servlet</servlet-name>
    <servlet-class>tp.B2cMvc2Servlet</servlet-class>
  </servlet>
  ...
  <servlet-mapping>
    <servlet-name>B2cMvc2Servlet</servlet-name>
    <url-pattern>/Mvc2B2c</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

2.2. Ant

ANT est un projet open source de la communauté Apache qui permet d'encoder en XML des **fichiers de scripts** (équivalents à des "Makefile").

==> *ANT* est très utilisé par les développeur Java car il permet d'automatiser facilement des compilations , des créations d'archives et d'autres opérations diverses (copies de fichiers , déploiements,) .

Le grand atout d'un script ANT par rapport à des fichiers ".bat" ou ".sh" est d'être *indépendant de la plate-forme* (Unix, Windows) et donc très *portable*.

2.3. XUL

Xml based User Interface Language (IHM événementielle en mode interprété)

==> technologie utilisée au sein de *Mozilla / Firefox* .

2.4. Echanges de données entre applications

Il s'agit de la principale utilisation d'XML.

==> se rapporter au chapitres "DTD" et "Schéma" pour se remémorer le principe de validation des fichiers échangés.

==> se rapporter également au protocole "SOAP with attachment" pour entrevoir un moyen (parmi d'autres) de véhiculer des fichiers à travers le réseau .

==> La principale difficulté est de parvenir à un accord sur le format des données qui seront échangées (==> normes française , européennes , mondiales , inter-galactiques).

3. E.A.I.

3.1. Concepts d' E.A.I.

3.2. Bonnes aptitudes d'XML et limitations

4. Services WEB

VI - XPath et XQuery

1. XPath

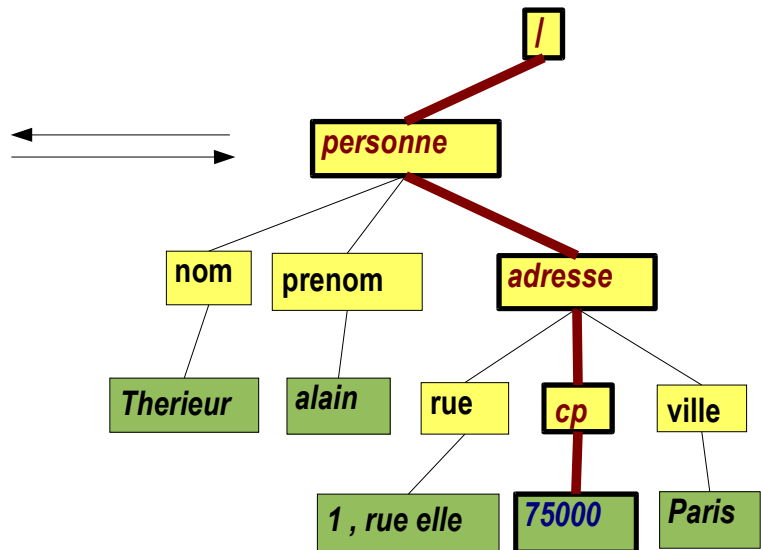
1.1. Présentation de XPath

Le mini langage XPath permet d'exprimer des chemins au sein d'un arbre XML.

XPath = chemin dans un arbre XML

pers1.xml

```
<personne>
  <nom>Therieur</nom>
  <prenom>alain</prenom>
  <adresse>
    <rue>1, rue elle</rue>
    <cp>75000</cp>
    <ville>Paris</ville>
  </adresse>
</personne>
```



Le chemin **XPath** menant au code postal est :

/personne/adresse/cp ou **/personne/adresse/cp/text()**

Autre exemple de chemin XPath: **/bibliographie/sujet[1]/livre[1]/auteur**

La recommandation **XPath 1.0** du W3C date du 16/11/1999 .

Le texte de référence est accessible via l'URL : <http://www.w3.org/TR/xpath>

Principales utilisations de XPath :

- Localiser des parties à extraire via **XQuery**
- Localiser des parties à récupérer et à transformer via **XSLT**
- Localiser des parties à atteindre avec des liens **XLink** / **XPointer**

NB: Il existe une version "développée" (ou "longue") de la syntaxe XPath qui n'est cependant jamais utilisée en pratique.

1.2. Syntaxe abrégée (réellement utilisée)

De façon très similaire aux chemins d'accès (*pathname*) menant à des sous répertoires ou fichiers d'un système de fichiers, un **chemin XPath** comporte les principales caractéristiques suivantes:

- il peut s'exprimer soit en **relatif** (par rapport au niveau courant) , soit en **absolu** (par rapport à la racine du document).
- il est constitué d'une **succession de pas élémentaires** correspondant généralement à des **balises à traverser** au sein de l'arbre XML

Chaque pas élémentaire (*LocationStep*) peut éventuellement être de la forme:

node-test[*prédicat*] (*ex: sujet*[1] <=> *sujet*[*position()* = 1])

Le test de noeud (**node-test**) est généralement un **nom de balise** ou une **fonction prédéfinie** telle que *text()* , *comment()* , *node()* et ceci permet d'écarter tous les noeuds qui ne sont pas du type indiqué.

Principaux éléments de syntaxe de XPath:

<i>livre</i>	sélectionne tous les éléments fils de nom " <i>livre</i> "
*	éléments fils quelconques (tout nom de balise)
@ <i>titre</i>	sélectionne l' attribut <i>titre</i> de l'élément courant
@*	sélectionne tous les attributs de l'élément courant
.	noeud courant (contextuel)
..	noeud parent
<i>livre/editeur</i>	sélectionne le ou les élément(s) " <i>editeur</i> " directement situé(s) sous un élément " <i>livre</i> " (fils direct)
<i>biblio//livre</i>	sélectionne tous les éléments " <i>livre</i> " situés 1 , 2 ou n niveaux en dessous d'un élément " <i>biblio</i> " (descendants directs ou indirects)
<i>chapitre</i> [1]	sélectionne l'élément fils " <i>chapitre</i> " situé en première position
<i>chapitre</i> [<i>last()</i>]	sélectionne l'élément fils " <i>chapitre</i> " situé en dernière position
<i>log</i> [@ <i>level</i> ="warning"]	sélectionne les éléments fils " <i>log</i> " dont l'attribut <i>level</i> vaut " <i>warning</i> "
<i>para</i> [@ <i>note</i>]	sélectionne les éléments fils " <i>para</i> " qui ont un attribut " <i>note</i> "

Fonctions prédéfinies de XPath:

text()	sélectionne le texte de l'élément courant
node()	sélectionne tous les noeuds
comment()	sélectionne les noeuds de type commentaire (<!-- -->)
processing-instruction()	sélectionne les noeuds de type instructions de traitement (<? ... ?>)
position()	récupère la position de l'élément courant (selon l'ordre des éléments fils qui apparaissent directement sous le parent)
last()	récupère la position du dernier élément
count(...)	compte le nombre d'éléments correspondant au paramètre indiqué. <i>Ex: <code> sujet[count(livre) > 5] ==> sujet ayant plus de 5 livres</code></i>
element()	sélectionne tous les éléments (noeuds liés à des balises / tags)

Quelques exemples de chemins XPath:

- `/bibliographie//livre[@titre='java 5']/auteur`
- `/bibliographie/sujet[last()]/livre[@prix < 10]/@titre`

1.3. Extensions de XPath 2

XPath 2.0 est un **sur-ensemble** de XPath 1.0 .

Nouveaux éléments ajoutés:

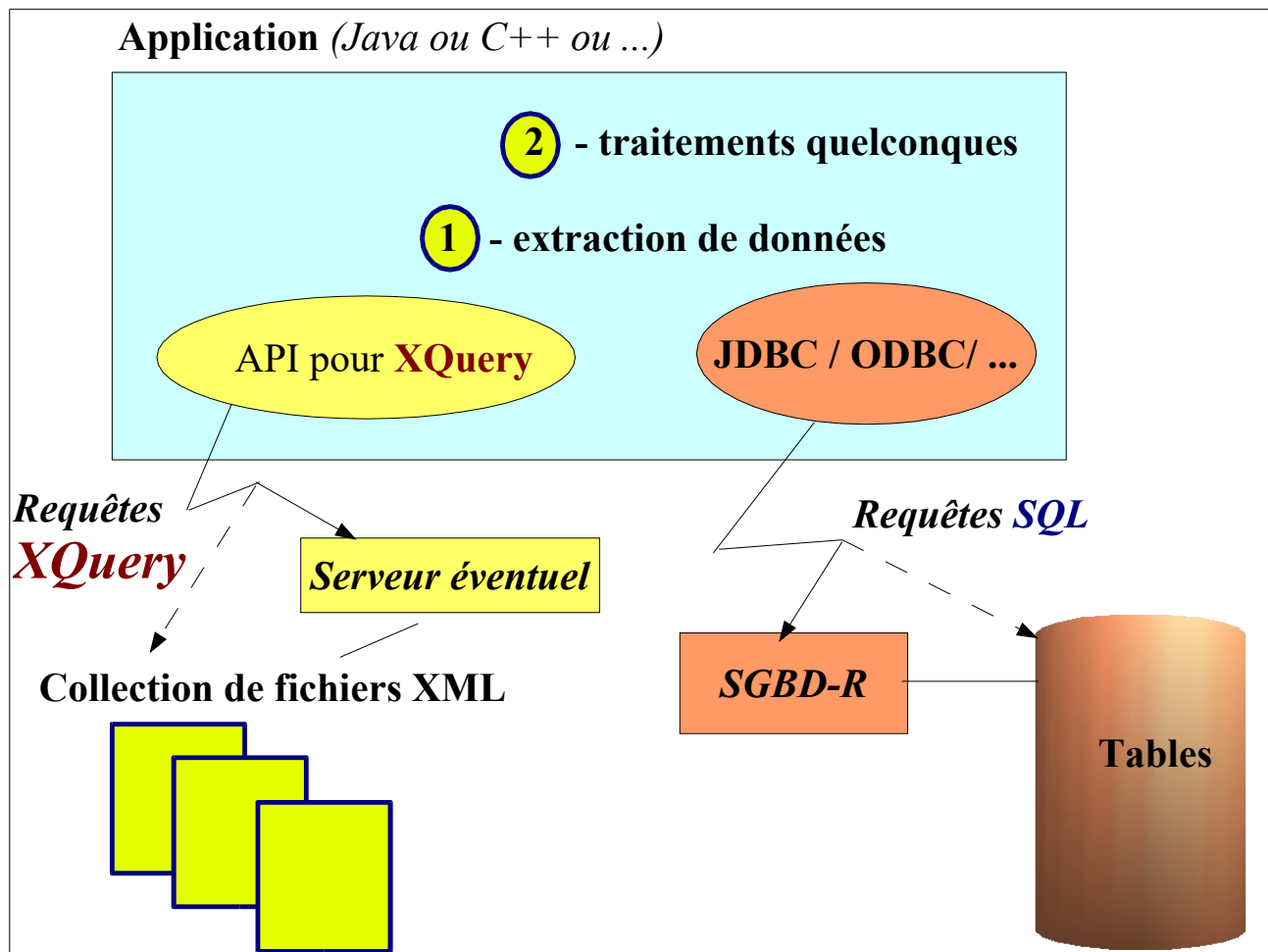
- Riche ensemble de types de données (identiques à ceux des schémas xsd).
- Beaucoup plus de fonctions prédéfinies pour exprimer les prédicats et retraiter certaines données.
- Modèle de données commun avec XQuery 1.0

2. XQuery

2.1. Présentation de XQuery

XQuery est un langage d'extraction de données XML.

D'un point de vue fonctionnel, *XQuery* est dans le monde XML, ce que la partie "SELECT ..." du langage SQL est dans le domaine des bases de données relationnelles.



Contrairement à XSLT, XQuery est un langage qui s'appuie sur des types de données très précis (ex: *xs:string*, *xs:integer*, *element()*, ...).

Ceci permet de faire en sorte que les données extraites soient directement transposables (sans aucune ambiguïté) dans des variables d'un langage fortement typé tel que Java, C++ ou C#.

D'un point de vue plus formel, XQuery est :

- Un **modèle de données** (avec types précis) pour les **documents XML**.
- Un **ensemble d'opérateurs** (arithmétiques, logiques, ensemblistes, ...) basés sur ce modèle.
- Un **langage d'interrogation** basé sur ces opérateurs.

2.2. Modèle de données de XQuery

- Basé sur "*XML Infoset*"
- Modèle de données proche de celui utilisé par XSLT .
- Exploitant les types de données des Schémas (**xsd**).
- Prenant en compte les namespaces
- Capable (en théorie) de gérer des références inter- (et intra-) document.

2.3. Langage d'interrogation basé sur des expressions

XQuery est un *langage fonctionnel* au sein duquel une **requête** est **vue comme une expression** .

Les expressions (ou requêtes) peuvent être emboîtées les unes dans les autres.

En entrée et en sortie : des instances du modèle de données de XQuery .

2.4. Opérateurs et expressions de XQuery

2.4.a. Expressions de chemins (Path):

reprise de **XPath 1.0**

exemple: `document("biblio.xml")//sujet[2]//livre[@titre = "java 5"]`

avec quelques suppléments:

- un opérateur permettant de suivre des références/liaisons (`->`)
- un prédicat de plage (*range*) pour exprimer certaines séquences: (1 **to** 4)

2.4.b. Opérateurs arithmétiques:

`+` , `-` , `*` , `/` , ...

exemples: `3+4` `=> 7`

```
let $x := 5
let $y := 8
return $x+$y
```

`=> 13`

2.4.c. Séquences:

```
let $paire1 := (3,4)
let $paire_de_paire := ($paire1, $paire1)
let $element := 99
let $vide := ()
return (count($paire_de_paire) , count($paire1), count($element), count($vide))
```

⇒ génère (4, 2, 1, 0).

2.4.d. Itérations:

```
for $x in (1 to 3) return ($x,5*$x)
```

⇒ génère 1, 5, 2, 10, 3, 15

```
<html>{
let $sujet := document("biblio.xml")//sujet[1]
for $livre in $sujet/livre
return <h2>{$livre/titre/text()}</h2>
}</html>
```

=> génère :

```
<?xml version='1.0' encoding='UTF-8'?>
<html>
  <h2>Mieux développer en C++</h2>
  <h2>Numerical recipes in c++</h2>
</html>
```

2.4.e. Opérateurs logiques:

or , and , ...

2.4.f. Opérateurs ensemblistes:

union, intersect, ...

2.4.g. Expressions conditionnelles :

exemples:

```
let $x :=5
return if ( $x mod 2 = 0) then "pair" else "impair"
```

```
for $b in document("biblio.xml")//livre order by ($b/titre/text())
return
<livre titre='{ $b/titre/text()}'
  gamme_prix='{ if($b/@prix < 30) then "prix_vert" else "prix_orange" }' />
```

2.4.h.

2.4.i. XQuery FLWR (For ... Let ... Where ... Return ...) Expression

La construction FLWR est très semblable à l'instruction SELECT du SQL.

La boucle **for** permet d'itérer sur un ensemble de valeurs ou de noeuds
pour chaque itération:

- l'instruction **let** peut éventuellement être utilisée pour calculer la valeur d'une variable qui sera utilisable au niveau des parties "**where**" et "**return**"
- la clause **where** permet d'effectuer un *filtrage*
- la partie **return** permet d'indiquer ce qu'il faut *générer*

Exemples:

```
for $b in document("biblio.xml")//livre
  where $b/editeur = "Eyrolles" and $b/@prix < 40
return $b/@titre
```

```
for $e in distinct-values(document("biblio.xml")//editeur/text())
let $moyenne := avg(document("biblio.xml")//livre[editeur = $e]/@prix)
return
<editeur>
  <nom> { $e } </nom>
  <prix_moyen> { $moyenne } </prix_moyen>
</editeur>
```

2.4.j. Fonctions

XQuery est un langage fortement typé qui nécessite la précision (via le mot clef "*as*") des types de données pour les paramètres et les valeurs de retour des fonctions .

exemples:

```
declare function local:addition($x as xs:double , $y as xs:double) as xs:double
{
  $x+$y
}

local:addition(2,3)
```

==> 5

```
define function local:liste_sujet($biblio as element() ) as element()
{
  <html><body><ul>
  {
    for $s in $biblio/sujet
    return <li>{$s/titre/text()}</li>
  }
  </ul></body></html>
}

local:liste_sujet(document("biblio.xml")/bibliographie)
```

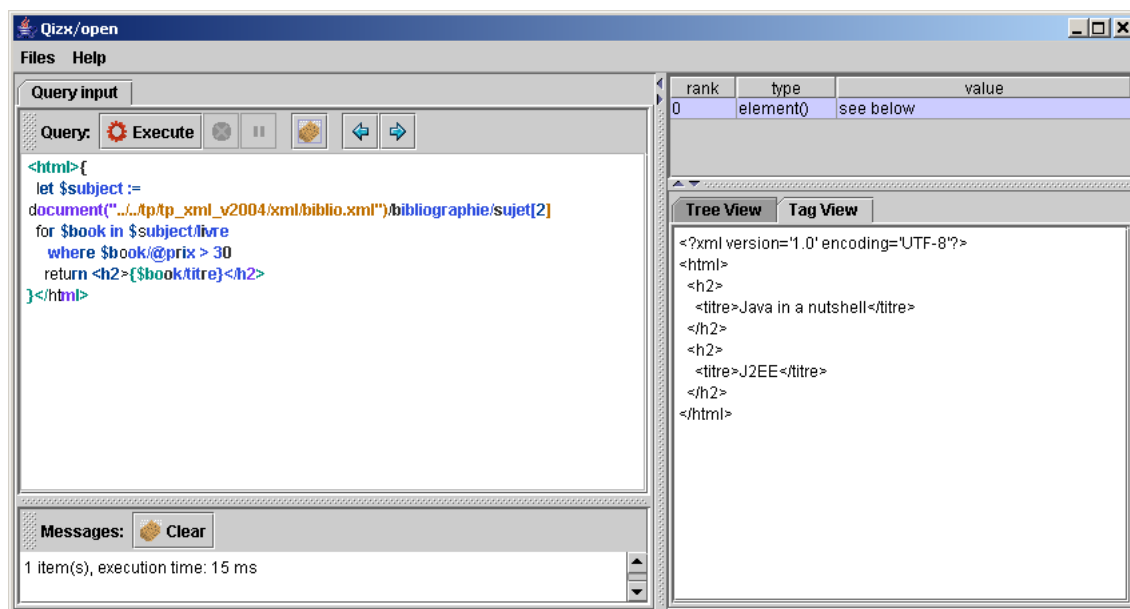
==>

```
<?xml version='1.0' encoding='UTF-8'?>
<html>
  <body>
    <ul>
      <li>C++</li>
      <li>Java</li>
      <li>XML</li>
    </ul>
  </body>
</html>
```


2.5. Quelques produits "XQuery"

QizX/open	http://www.xfra.net/qizxopen/root.html	bonne documentation très simple à utiliser
Qexo / Kawa	GNU product	précurseur évoluant peu. latest release : 2003 !!!
eXist	http://exist.sourceforge.net/	respect des standards , ... base de données XML
...		
XML:DB API	http://xmldb-org.sourceforge.net/xapi/	api utilisé en interne par beaucoup de moteurs XQuery programmés en Java

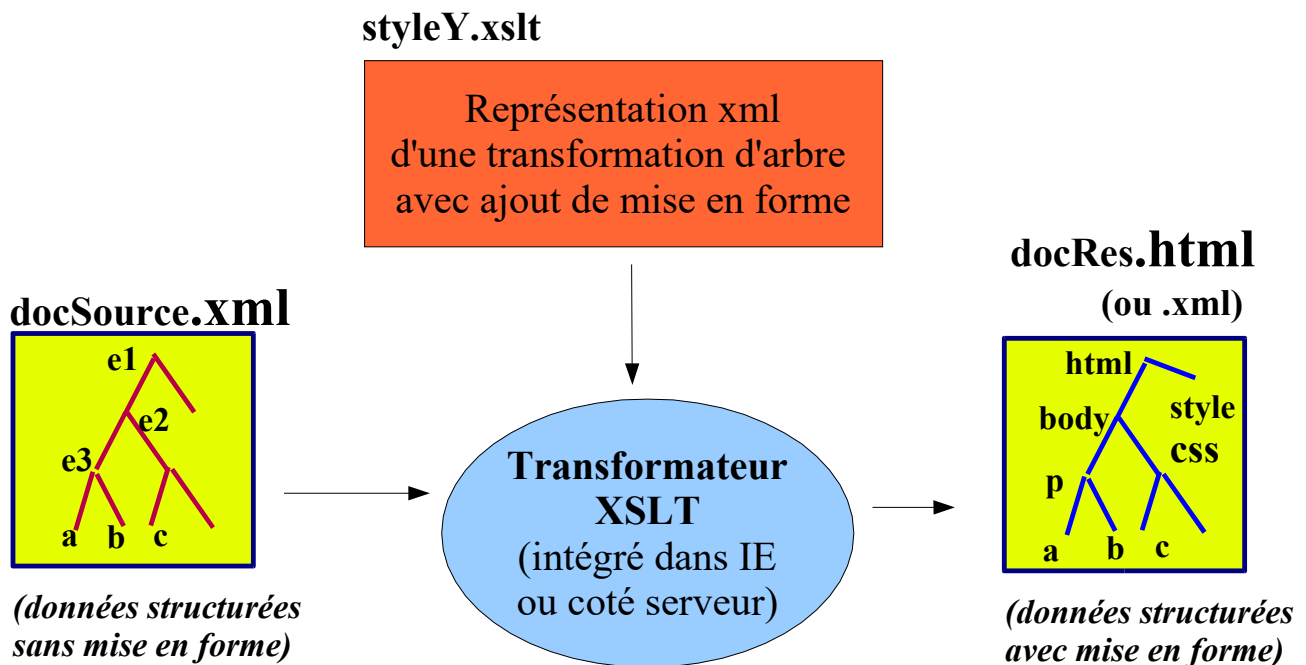
2.6. QizX/open en mode GUI :



VII - XSLT et XSL-FO

1. Présentation de XSLT

Transformations XSLT



La partie d'XSL qui traite des transformations d'arbres s'intitule **XSLT** (*XSL Transformation*).
XSLT 1.0 est une recommandation de W3C datant du *16 novembre 1999*.

XSLT permet de transformer un arbre xml source en un autre arbre résultat.

On peut ainsi générer une présentation HTML à partir d'une source XML.

On peut également prendre des éléments de l'arbre source dans un certain ordre pour les placer dans un autre ordre dans l'arbre résultat. Des duplications sont en outre possibles. On peut ainsi générer automatiquement une table des matières avec XSLT.

Le namespace associé à **XSLT** est <http://www.w3.org/1999/XSL/Transform>

D'autre part, la mention d'un numéro de **version** est obligatoire au niveau de la feuille de style XSLT :

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0" >
  <xsl:template match="/"> ...</xsl:template> ...
</xsl:stylesheet>
```

EXEMPLE**Source.xml**

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<?xml-stylesheet type='text/xsl' href='fic_style.xslt' ?>
<pays nom='France' capitale='Paris' >
  <region nom='Haute-Normandie' >
    <departement nom='Eure' num='27' prefecture='Evreux' />
    <departement nom='Seine Maritime' num='76' prefecture='Rouen' />
  </region>
  ...
</pays>
```

fic_style.xslt

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
  <xsl:template match="/pays">
    <html>
      <body>
        <h3>...<xsl:value-of select="@nom" /> ...</h3>
        <xsl:apply-templates select="region" />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="region">
    <h5> <font color='blue'><xsl:value-of select="@nom" /> </font> </h5>
    <table border='1'> ...
      <xsl:for-each select="departement">
        <tr>
          <td> <xsl:value-of select="@num" /> </td>
          ...
        </tr>
      </xsl:for-each>
    </table>
  </xsl:template>
</xsl:stylesheet>
```

Processeur XSLT**fic_res.html**

```
<html> ... <body>
<h3>France</h3>

<h5>Haute-Normandie</h5> ....
<table border="1">
  <tr> <td>27</td> <td>Eure</td> <td>Evreux</td> </tr>
  <tr> <td>76</td> <td>Seine Maritime</td> <td>Rouen</td> </tr>
</table>
...
</body> </html>
```

France

Haute-Normandie

num	departement	prefecture
27	Eure	Evreux
76	Seine Maritime	Rouen

2. Applications des templates

Une feuille de style XSLT est **structurée** en un *ensemble* de **templates** .

Chaque **template** correspond à un **modèle de (sous) arbre** qui sera **généré dans le fichier résultat**.

Le **processeur XSLT** va **automatiquement instancier (appliquer) un template principal** dès qu'il trouvera dans l'*arbre source* un élément accessible via le **chemin XPath** renseigné au niveau de l'attribut **match** du template .

Dans toute feuille de style XSLT on trouve au moins un **template principal** dont l'attribut **match** correspond à un *chemin XPath absolu (ou relatif à la racine)* menant généralement *au sommet de l'arbre source*.

Exemples:

```
<template match="/" > ..... </template>
```

est appliqué systématiquement puisque tout fichier source comporte forcément une racine .

```
<template match="/nom_balise_de_premier_niveau" > .... </template>
```

est une autre forme classique de template principal .

Les autres **(sous) templates** de la feuille de style doivent avoir un attribut **match** ne comportant que le **nom** terminal **d'un élément** de l'arbre source (*simple nom de balise*).

Ces **sous templates** (générant chacun une sous partie de l'arbre résultat) seront **déclenchés** via l'instruction **xsl:apply-templates** d'un template parent .

2.1. apply-templates et liaisons entre templates

L'instruction **<xsl:apply-templates select="chemin_XPath" />** donne l'ordre de **déclencher** immédiatement (et récursivement s'il le faut) **tous les sous templates sélectionnés** par le chemin XPath renseigné par l'attribut **select**.

xsl:apply-templates est une instruction XSLT qui **boucle** donc **sur chacun des sous éléments de l'arbre source** et tous les sous arbres alors générés seront accrochés à cet endroit dans le haut de l'arbre résultat.

Dans l'exemple précédent , le template principal (lié à la balise pays) utilise <apply-templates select="region" /> pour boucler automatiquement sur toutes les régions .

2.2. Templates nommés / call-template

Dans certains cas (relativement rares) , il peut être plus pratique d'**appeler un template par son nom** plutôt qu'en fonction d'une correspondance avec une partie du fichier source à traiter.

Ceci peut être le cas d'un **template générique** du genre "*entête*" qui comporte des éléments que l'on a envie d'appliquer systématiquement (indépendamment des données du fichier d'entrée).

exemple:

```
<xsl:template name="entete">
  <table border="5" width="100%">
    <tr> <td> ENTETE DE PAGE --- LOGO --- Date et Heure </td> </tr>
  </table>
</xsl:template>
```

```
<xsl:template match="/">
<html>
  ...
  <body>
    <xsl:call-template name="entete"/>
  ...
</body>
</html>
</xsl:template>
```

2.3. Boucle for-each

La boucle **for-each** permet de **parcourir un ensemble d'éléments de même niveau** (*sélectionnés par un chemin XPath*) est très pratique pour construire des tableaux ou d'autres listes d'éléments:

```
<xsl:template match="region">
  <h5> <font color='blue'><xsl:value-of select="@nom" /> </font> </h5>
  <table border='1'>
    <tr> <th> num </th> <th> departement </th> <th> prefecture </th> </tr>
    <xsl:for-each select="departement">
      <tr>
        <td> <xsl:value-of select="@num" /></td>
        <td> <xsl:value-of select="@nom" /></td>
        <td> <xsl:value-of select="@prefecture" /></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
```

NB: à l'intérieur d'une boucle **<for-each>** , le niveau courant change et correspond à l'élément sur lequel porte l'itération. Il faut alors utiliser des chemins XPath exprimés **relativement** à ce niveau.

Rappel des principales syntaxes de XPath:

@titre	sélectionne l'attribut <i>titre</i> de l'élément courant
.	noeud courant (contextuel)
..	noeud parent
livre/editeur	sélectionne le ou les élément(s) " <i>editeur</i> " directement situé(s) sous un élément " <i>livre</i> " (<i>fil direct</i>)
biblio//livre	sélectionne tous les éléments " <i>livre</i> " situés 1 , 2 ou n niveaux en dessous d'un élément " <i>biblio</i> " (<i>descendants directs ou indirects</i>)
chapitre[1]	sélectionne l'élément fils " <i>chapitre</i> " situé en première position

2.4. Modes (variantes d'un template)

On peut éventuellement définir **plusieurs variantes d'un template** ayant un même motif de correspondance (*renseigné au niveau d'attribut **match***) .

Ceci s'effectue en donnant différentes valeurs à l'attribut optionnel **mode** que l'on retrouve sur les éléments `<xsl:template ...>` et `<xsl:apply-templates>` .

Exemple:

```
<xsl:template match="/">
<html>
  <body>
    ...
    <!-- table des matières -->
    <xsl:apply-templates select="bibliographie/sujet" mode="tm"/>
    <hr/>
    <xsl:apply-templates select="bibliographie/sujet" mode="content"/>
  </body>
</html>
</xsl:template>
```

```
<!-- - mode="tm" table des matières (que les titres) -->
<xsl:template match="sujet" mode="tm">
  <h4><xsl:number count="sujet" format="I - "/> <xsl:value-of select="titre"/></h4>
</xsl:template>
```

```
<!-- - mode="content" contenu développé -->
<xsl:template match="sujet" mode="content">
  <h4><xsl:number count="sujet" format="I - "/> <xsl:value-of select="titre"/></h4>
  <table border='1'> ... <xsl:for-each select="livre"> .... </xsl:for-each> </table>
</xsl:template>
```

2.5. Numérotation automatique

Pour simplement récupérer un **numéro d'ordre** correspondant à la **position courante** de la boucle actuelle, on peut se contenter d'utiliser `<xsl:value-of select="position()" />`.

Pour obtenir une **numérotation plus évoluée**, on pourra utiliser l'instruction suivante:

```
<xsl:number level="..." count="..." format="..." />
```

- L'attribut *level* permet de préciser le niveau de numérotation ("*single*" ou "*multiple*").
- L'attribut *count* permet de renseigner quels sont les nœuds de l'arbre XML qui doivent être comptés (*Attention*: il ne faut pas utiliser à cet endroit une syntaxe *XPath* mais une syntaxe spécifique : "*Elt_Niveau1|Elt_Niveau2*")
- L'attribut *format* permet de préciser le type de numérotation (A,B,C, 1,2,3 ,I,II,III, ...)

Exemples:

```
<xsl:template match="Chapitre">
  <xsl:number count="Chapitre" level="single" format="1. " />
  <xsl:value-of select="@titre"/><br/>
  <xsl:apply-templates select="Paragraphe"/>
</xsl:template>

<xsl:template match="Paragraphe">
  <xsl:number count="Chapitre|Paragraphe" level="multiple" format="1.a " />
  <xsl:value-of select="."/> <br/>
</xsl:template>
```

2.6. Ordonnancement (tri) des noeuds

L'instruction `<xsl:sort .../>` permet de modifier l'ordre de parcours au sein d'une boucle `<xsl:for-each>` ou dans `<xsl:apply-templates>`.

Syntaxe générale :

```
<xsl:sort select="IndexDeTri" order="OrdreDeTri" data-type="TypeDeTri">
```

- L'attribut *select* est obligatoire. Il permet de préciser quel est le critère du tri.
- L'attribut *order* indique l'ordre de tri. Il peut prendre les valeurs "*ascending*" (valeur par défaut) ou "*descending*".
- L'attribut *data-type* indique s'il s'agit d'un tri alphabétique ou numérique. IL peut prendre les valeurs "*text*" ou "*number*".

exemples:

```
<xsl:for-each select="departement">
  <xsl:sort select="@num" order="ascending" data-type="number"/>
  <tr>
    <td> <xsl:value-of select="@num" /></td>
    <td> <xsl:value-of select="@nom" /></td>
    <td> <xsl:value-of select="@prefecture" /></td>
  </tr>
</xsl:for-each>
```

```
<xsl:apply-templates select="sujet/livre">
  <xsl:sort select="@titre" order="ascending" data-type="text"/>
</xsl:apply-templates>
```

2.7. Création de balises à attributs variables

Si l'on souhaite générer un lien hypertexte HTML dont l'URL portée par l'attribut *href* est variable (#p1, #p2, #p3, ...) et devant par exemple être calculée en fonction de la position courante, alors l'écriture suivante (pourtant intuitive) ne fonctionne pas:

~~<a href="#p<xsl:value select="position()"/>" /> > ~~

Il est en effet impossible de placer une balise XML à l'intérieur de l'ouverture d'une autre .

De façon à contourner ce problème récurrent, le langage XSLT nous offre deux instructions sophistiquées `<xsl:element ...>` et `<xsl:attribute ...>` qui une fois combinées entre elles, permettent de générer des balises dont les attributs sont variables .

Exemple:

```
<xsl:element name="a">
  <xsl:attribute name="href">#p<xsl:value-of select="position()" /> </xsl:attribute>
  vers xxxx
</xsl:element>
```

permet de générer la ligne suivante dans le document résultat:

```
<a href="#p1" > vers xxxx </a>
```

2.8. Tests conditionnels


```

<xsl:for-each match="departement">
...
<xsl:if test="position() mod 2 = 0">
...
</xsl:if>
...

```

`<xsl:if>` ne comportant pas de partie *else*, on préfère souvent utiliser :

```

<xsl:choose>
  <xsl:when test="@prix < 20" >
    ...
  </xsl:when>

  <xsl:when test="@prix < 50" >
    ...
  </xsl:when>

  <xsl:otherwise>
    ...
  </xsl:otherwise>
</xsl:choose>

```

Exemple:

```

<xsl:for-each select="departement">
  <xsl:element name="tr">
    <xsl:attribute name="bgcolor">
      <xsl:choose>
        <xsl:when test="position() mod 2 = 0">yellow</xsl:when>
        <xsl:otherwise>green</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
    <td> <xsl:value-of select="@num" /></td>
    <td> <xsl:value-of select="@nom" /></td>
    <td> <xsl:value-of select="@prefecture" /></td>
  </xsl:element>
</xsl:for-each>

```

2.9. Passage de paramètres aux sous templates

Un sous template peut éventuellement être considéré comme une sous fonction.

On peut alors lui passer des paramètres de façon à moduler le contenu qu'il doit générer.

Exemple:

```
<xsl:template match="sujet">
...
<xsl:apply-templates select="livre">
  <xsl:with-param name="nom_sujet" select="titre"/>
</xsl:apply-templates>
...
</xsl:template>
```

```
<xsl:template match="livre">
  <xsl:param name="nom_sujet" select="sujet_inconnu" />
  <p>
    Livre <xsl:value-of select="@titre"/> [ sujet = <xsl:value-of select="$nom_sujet"/> ]
  </p>
</xsl:template>
```

NB: La valeur 'sujet_inconnu' de l'attribut *select* de la balise *<xsl:param>* correspond à une *valeur par défaut* .

2.10. inclusion de sous fichiers ".xslt"

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

<xsl:import href="generic_html.xslt" />

<xsl:output method="html" version="1.0" indent="yes" />

<xsl:template match="/">
  <html>... </html>
</xsl:template>

</xsl:stylesheet>
```

2.11. Transformation identité et reformulation (xml -> xml)

Transformation identité :

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Le template ci-dessus recopie tel quel l'élément courant de l'arbre source via l'instruction `<xsl:copy>` et se rappelle de façon réursive pour recopier également tous les attributs et les sous-éléments.

Une petite transformation (reformulation):

Ex: Laisser le document source quasi inchangé et ne remplacer que `<titre>` par `<title>` :

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" indent="yes"/>

  <xsl:template match="titre">
    <title>
      <xsl:apply-templates select="@*|node()"/>
    </title>
  </xsl:template>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

VIII - "Parsing XML" (Stratégies/API)

1. Différents types d'analyse

Grandes catégories de parseurs	caractéristiques
au fil de l'eau (événementiel/push ou pull) <i>exemple: SAX , StAx</i>	Déclenchant des <i>traitements simples</i> au fur et à mesure de la lecture des lignes du fichier xml à analyser , cette méthode a le mérite d'être <i>efficace</i> et ne <i>consomme que très peu de mémoire</i> au sein de l'application.
Arbre de noeuds en mémoire <i>exemple : DOM</i>	<i>Créant en mémoire un arbre complet contenant toutes les données du document xml</i> à analyser, cette méthode <i>consomme plus de ressources</i> mais permet en contre-partie d'effectuer des <i>traitements plus élaborés qui peuvent être basés sur la globalité du document</i> .
Liaison/mapping " <i>Java / XML</i> " via l'API " <i>JAXB2</i> "	Certaines API sophistiquées (telles que JAXB2 de java6) sont capables d'effectuer une correspondance quasi directe entre des objets en mémoire et des parties d'un document XML . Ceci nécessite un petit paramétrage (sous forme d'annotations Java). <u>NB:</u> JAXB comporte un générateur de code java (qui se base sur un schéma XML (.xsd)).
...	

NB: Dans le monde **Java** , **SAX** signifie *Simple Api for Xml* .

Cette API très simple (et assez rudimentaire) peut s'avérer pratique dans le cas où l'on recherche de très bonnes performances ou bien si l'on souhaite traiter de gros fichiers sans pour autant consommer trop de mémoire.

La nouvelle API java "**StAx**" (Stream Api for XML) fonctionne également au fil de l'eau (sans consommer beaucoup de mémoire) et elle permet en plus de programmer l'analyse d'un document XML de façon plus naturelle :

- au lieu que le parseur pousse les données XML vers le code événementiel (mode "push"),
- l'application peut demander au parseur d'aller chercher les données XML (mode "pull").

2. DOM (Document Object Model)

2.1. Présentation

DOM signifie *Document Object Model* .

Cette **API** normalisée par l'organisme **W3C** constitue un véritable **standard** .

2.1.a. *Universalité - versions dans quasiment tout langage*

Reposant sur un modèle orienté objet indépendant de tout langage de programmation, l' **API DOM** (*initialement spécifiée via le langage neutre IDL de Corba*) est utilisable dans un très grand nombre de langages informatiques:

Java , C++ , JavaScript , VBScript , PHP , ...

Seules diffèrent quelques adaptations liées à certains langages:

.documentElement en *JavaScript*

.getDocumentElement() en *Java*

2.1.b. *Xml-DOM et DOM niveaux 1,2 et 3 pour XHTML*

Le coeur de l'api DOM (**Xml-DOM**) permet de traiter n'importe quel fichier XML.

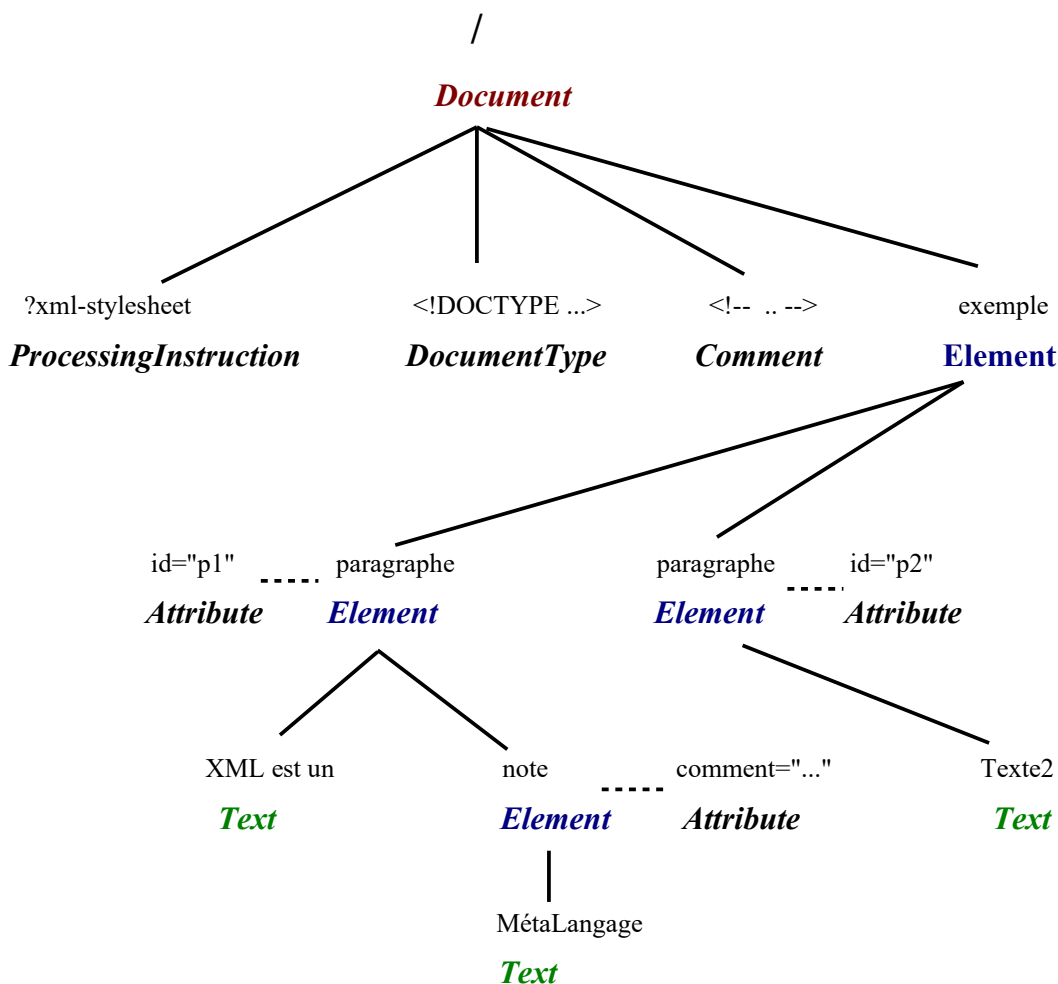
Des extensions liées au langage (X)HTML (considéré comme un cas particulier de XML) permettent d'écrire des petits morceaux de scripts (*ex: JavaScript*) qui vont pouvoir opérer sur la structure d'un document (*ex: changement dynamique de style , développement / contraction d'une zone, permutation d'images , autres animations ou effets dynamiques, ...*).

2.2. Modèle d'arbre en mémoire

Le fichier xml suivant

```
<?xml version="1.0" ?>
<?xml-stylesheet href="/style1.css" type="text/css" ?>
<!DOCTYPE exemple
[
  <!ENTITY eacute "&#x00E9;" >
]>
<!-- commentaire -->
<exemple>
  <paragraphe id="p1">XML est un
    <note comment="important"> M&eacute;taLangage</note>
  </paragraphe>
  <paragraphe id="p2">texte2</paragraphe>
</exemple>
```

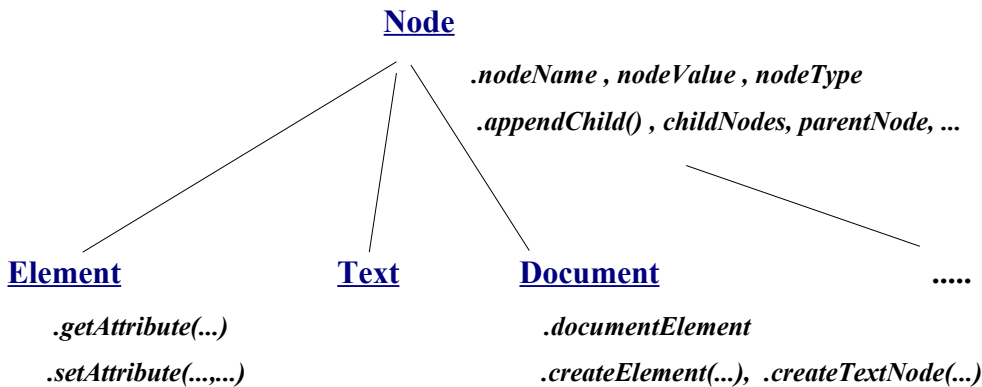
est représenté via un arbre DOM de ce type:



Dans l'arbre précédent, les noeuds ne sont pas tous du même type:

- Le noeud racine est de type **Document**
- Les noeuds liés aux balises sont de type **Element**
- Les noeuds comportant un morceau de texte sont de type **Text**

Cependant, ces différents types de noeuds héritent tous d'un type générique : "**Node**".



Certaines fonctions (associées au type générique *Node*) sont accessibles depuis n'importe quel noeud :

- **nodeName** retourne le nom d'une balise ou null .
- **nodeValue** retourne la valeur d'un texte ou null .
- **nodeType** retourne une constante indiquant le type de noeud (ELEMENT_NODE , TEXT_NODE, ...)
- **childNodes** retourne une liste de noeuds fils sous la forme d'un objet ensembliste de type **NodeList** .
- **parentNode** retourne une référence sur le noeud père

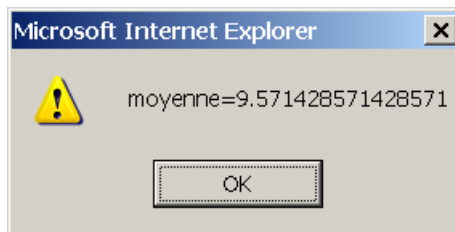
D'autres fonctions ne sont disponibles que sur certains types précis de noeuds:

- La fonction **getDocumentElement()** que l'on appelle sur le noeud racine du document retourne **l'unique noeud de type Element qui correspond à la balise de premier niveau** .
- Seul le noeud racine (de type *Document*) comporte des fonctions [**createElement("nombalise")** , **createTextNode("valeurTexte")**] permettant de créer de nouveaux noeuds qui devront ultérieurement être accrochés sous un des noeuds de l'arbre via la méthode **appendChild()** .
- Les méthodes **setAttribute("nomAttribut","valeur")** et **getAttribute("nomAttribut")** doivent être appelée sur un noeud de type *Element* .

2.3. Exemple (DOM / javascript) :

serie.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<serie>
  <val>5</val>
  ...
  <val>9</val>
</serie>
```



```
<html>
</head>

<script>
function btn_onClick(){
  calculer_moyenne(thisForm.ficXml.value);
}

function calculer_moyenne(xmlFileName){
  var xhr = new XMLHttpRequest();

  xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && xhr.status == 200) {
      parse_xml_doc(xhr.responseXML);
    }
  }
  xhr.onerror = function() {
    console.log("Error while getting XML.");
  }

  xhr.open("GET", xmlFileName, true);
  xhr.send();
}

function parse_xml_doc(docXml){
  var noeudSerie = docXml.documentElement;
  //console.log("documentElement.nodeName="+docXml.documentElement.nodeName);
  var listeNoeudsVal = docXml.getElementsByTagName("val");
  var nbVal = listeNoeudsVal.length;
  var s=0;
  for (i=0;i<nbVal;i++){
    var noeudVal = listeNoeudsVal[i];
    var noeudTexte = noeudVal.firstChild;
    if(noeudTexte){
```



```

        s= s + parseInt(noeudTexte.nodeValue);
    }
}
//alert("Moyenne=" + (s/nbVal));
var moyenne = s/nbVal;
document.getElementById("spanMoyenne").innerHTML="<b>" +moyenne+"</b>";
}
</script>

</head>

<body>
page html contenant un script XML-DOM (en javascript) permettant de calculer la
moyenne des valeurs du fichier xml <br/>
Avec un navigateur récent , ça ne fonctionne qu'en http://...
avec une url relative menant au fichier .xml à analyser (restrictions CORS)<br/>
via lite-server et http://localhost:3000 ça fonctionne.
<hr/>
<form name="thisForm">
Fichier xml source:<input type="text" id="ficXml" name="ficXml" value="serie.xml"/> <br/>
<input type="button" id="btn" name="btn" value="calcul" onClick="btn_onClick()">
</form>
moyenne=<span id="spanMoyenne"></span>
</body>
</html>

```

ANNEXES

IX - Annexe – Entités XML et DTD (Has been)

1. Entités XML

1.1. Terminologie

Une **ressource** est une unité ou un service d'information **désignée par un identificateur globalement unique**.

Une ressource pourra éventuellement être disponible en plusieurs exemplaires (copies exactes).

Ex1: numéro ISBN pour les ouvrages imprimés , *Ex2*: norme ISO ,

Une **entité** est un **objet physique** (fichier , enregistrement d'une base de données ,).

Une entité peut éventuellement est référencé par une référence de portée locale.

Un **document** est une **oeuvre composite** qui **forme un tout**. Un document peut être copié et traduit sur plusieurs supports différents.

Type d'entité	Interne	Externe
XML	Abréviations , jeu de remplacements	sous documents, DTD , ...
non - Xml	interdit	Images, graphiques, sons, ...

1.2. Entités internes

Une **entité interne** correspond à une sorte d' **alias** (ou **abréviation**) pour une chaîne de caractères .

Pour définir des entités (ayant des valeurs de remplacement) qui seront utilisées dans une partie du document, la syntaxe générale est :

```
<!DOCTYPE NomSousArbre
[
  <!ENTITY NomEntite "valeur de remplacement" >
]>
```

Ensuite , dans une partie textuelle du document , on fera référence à l'entité via la syntaxe:

&*NomEntité*; ==> ceci provoquera un remplacement automatique .

exemple:

```
<?xml version="1.0" ?>
```

```
<!DOCTYPE exemple [
<!ENTITY politesses "Veuillez agreer , &mme_m; .... salutations distinguees" >
<!ENTITY mme_m "Madame, Monsieur" >
]>
<exemple>
    <p> mon texte </p>
    <p> &politesses; </p>
</exemple>
```

NB:

On peut également placer les lignes <!ENTITY> dans un fichier externe (*ex*: xxx.ent) qui sera référencé par le fichier de données XML via **SYSTEM "url"** ou **PUBLIC "nom public" "url"**.

example:

```
<?xml version="1.0" ?>  
!DOCTYPE exemple SYSTEM "xhtml.ent" >  
<exemple>  
    <p> mon texte </p>  
    <p> Debut&nbsp;&nbsp;&nbsp;&Fin </p>  
</exemple>
```

xhtml.ent

```
<!ENTITY nbsp "&#160;" >
```

==> Cette façon de procéder introduit cependant une dépendance qui tend à complexifier la gestion globale des fichiers . Ceci n'a d'intérêt que dans le cadre d'une compatibilité avec **HTML** .

==> Autant utiliser directement ** ** dans un **texte XML**.

1.3. Entités externes XML

Une **entité externe XML** correspond à un sous fichier XML qui sera inclus dans un document XML de plus haut niveau.

Syntaxe générale:

```
<!ENTITY NomEntitéExterne SYSTEM "url_entite_externes">
```

avec le remplacement déclenché par **&NomEntitéExterne;**

Example:

adresse.xml

```
<adresse>
  <rue> rue elle </rue>
  <cp> 75000 </cp>
  <ville> Paris </ville>
</adresse>
```

personne.xml

```

<?xml version="1.0" ?>
<!DOCTYPE personne
  [
    <!ENTITY adresse_principale SYSTEM "adresse.xml">
  ]>
<personne>
  <prenom> Alex </prenom> <nom> Therieur </nom>
  &adresse_principale;
</personne>

```

1.4. Entités externes non XML

Une **entité externe non XML** peut être un **fichier binaire** tel qu'une image , un enregistrement sonore ou encore une vidéo . Celle-ci ne pourra être traitée que par un logiciel comportant le module d'extension (plugin) adéquat.

La **norme générale XML** prévoit d'un point de vue **théorique**:

- La déclaration d'une **NOTATION** correspondant à un module d'extension.
- L'association d'une entité externe (ayant un certain type de contenu) avec une notation via l'information **NDATA** .

En pratique , une entité externe non XML est:

- représentée par une balise spécifique à un langage dérivé d'XML (ex: **<object type="..." ...>** ou **** en **xhtml**)
- référencée par un attribut (ex: **href="xxx.yyy"**) précisant l'**URL** de cette ressource.

2. Documents validés via une DTD

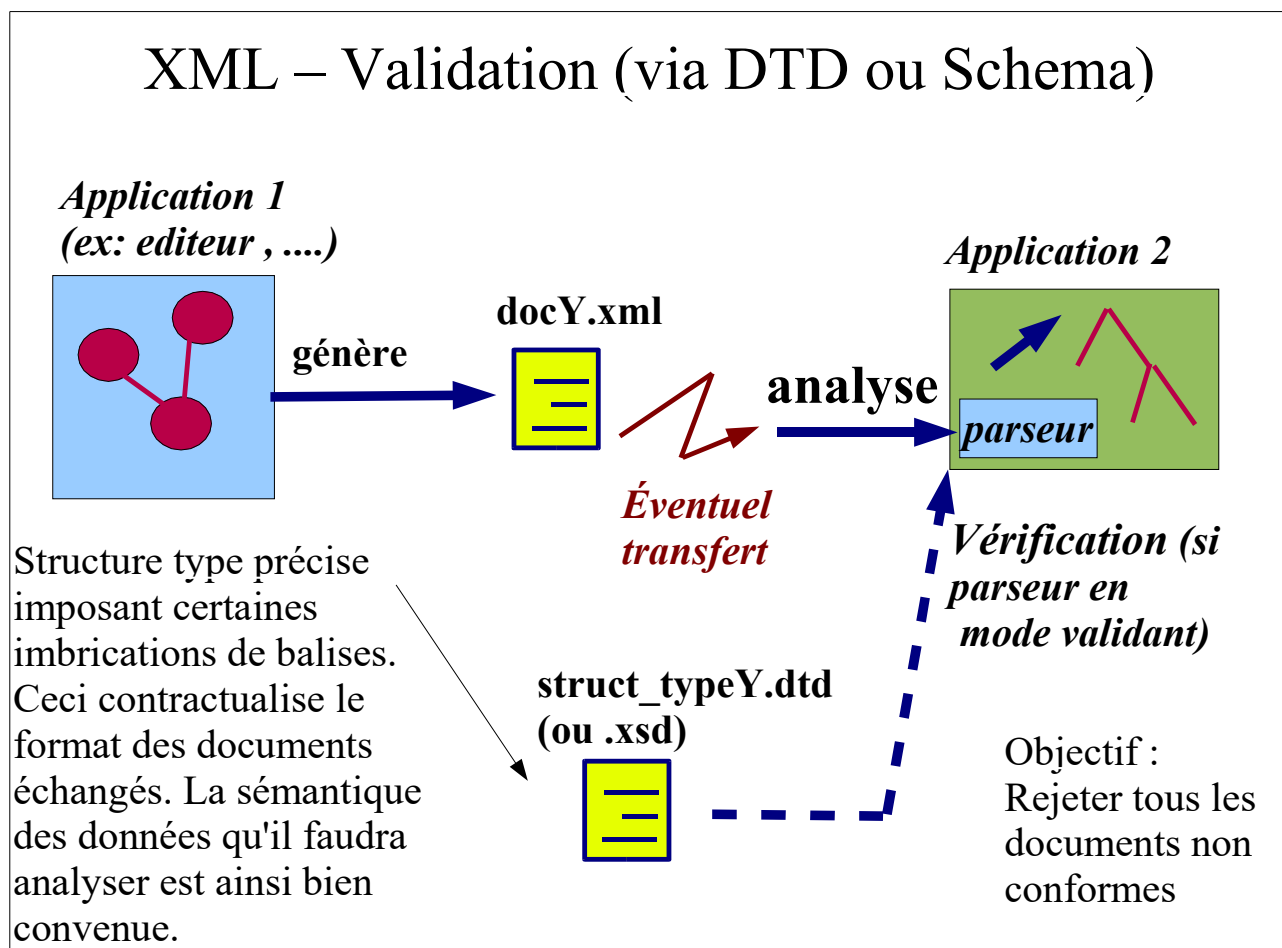
Une **DTD** (*Document Type Definition*) est un petit fichier qui correspond à la **structure type** d'une catégorie de documents qui seront échangés entre différentes applications.

Cette **structure type** s'exprime essentiellement en termes de **noms et d'imbrications de balises imposés**.

Un fichier XML bien formé (syntaxiquement correct) sera de plus **valide** s'il est conforme à la structure type décrite dans la DTD .

En pratique, une application vérifie cette conformité en utilisant un **parseur en mode validant** au moment où elle doit lire et charger le fichier XML en mémoire.

Si un élément du document est détecté comme "**non valide**" un **traitement d'exception** permet de **remonter un message d'erreur** et d'**arrêter la suite des traitements** qui conduiraient alors à des erreurs d'interprétations de données fausses ou altérées .



3. Lien entre la DTD et le document XML

3.1. DTD externe adressée par URL

address.dtd

```
<!ELEMENT address (street,zip,town,country?) >
<!ELEMENT street (#PCDATA) >
<!ELEMENT zip (#PCDATA) >
<!ELEMENT town (#PCDATA) >
<!ELEMENT country (#PCDATA) >
```

adresse.xml

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "../address.dtd">
<address>
  <street> 12, rue Elle </street>
  <zip> 75000 </zip>
  <town> Paris </town>
  <country> France </country>
</address>
```

NB: l'URL précisée après le mot clef *SYSTEM* peut être *relative* (ex: "../address.dtd") ou *absolue* (ex: <http://www.yyy.com/zzz/address.dtd>) et un accès internet est dans ce dernier cas obligatoire.

3.2. DTD externe adressée par FPI

Au lieu d'utiliser le mot clef *SYSTEM* suivi d'une simple URL , on peut éventuellement utiliser le mot clef *PUBLIC* suivi d'une double information:

- un nom logique (identifiant formel public [*FPI*])
- une *URL* (généralement absolue)

Exemple (fichier de configuration de J2EE):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>Mvc2B2cServlet</servlet-name> <servlet-class>b2c.Mvc2B2cServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Mvc2B2cServlet</servlet-name> <url-pattern>/Mvc2B2c</url-pattern>
  </servlet-mapping>
</web-app>
```

==> L'application qui va interpréter le fichier xml va alors essayer de trouver une version locale du fichier DTD en se basant sur le nom logique et n'utilisera l'URL distante (absolue) qu'en tant que solution de replis.

Ceci permet ainsi d'accéder à une version de la DTD même si l'on ne dispose pas d'un accès internet.

Pour que la prise en compte des noms logiques (FPI) puisse fonctionner, il faut (en règle générale) paramétrer l'application pour indiquer une liste de correspondances entre des noms logiques (FPI) et des URL. On parle quelquefois de fichier "catalogue" .

Exemples dans le monde "Java" :

* La tâche *ANT* **XmlValidate** peut être paramétrée pour qu'elle puisse trouver une version locale d'un fichier DTD:

```
<target name="valider_web_xml">
  <xmlvalidate file="${web_content}/WEB-INF/web.xml" warn="false">
    <dtd publicId="-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
      location="c:/repXxx/dtd/web-app_2_2.dtd"/>
    <dtd publicId="-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
      location="c:/repXxx/dtd/web-app_2_3.dtd"/>
  </xmlvalidate>
</target>
```

* Au sein d'un programme JAVA utilisant SAX , la méthode **resolveEntity**(String *publicId*, String *systemId*) permet d'encoder la façon dont on veut localiser le fichier DTD en fonction du nom logique (FPI) .

3.3. DTD interne et mixte

Les lignes introduites par **<!DOCTYPE nomSousArbre ...>** peuvent éventuellement être placées en interne dans le fichier XML lui même .
On utilisera pour cela des [] en tant que délimiteur de regroupement :

exemple (très rare):

adresse_v0.xml

```
<?xml version="1.0"?>
<!DOCTYPE address
[
  <!ELEMENT address (street,zip,town) >
  <!ELEMENT street (#PCDATA) >
  <!ELEMENT zip (#PCDATA) >
  <!ELEMENT town (#PCDATA) >
]>
<address>
  <street> 12, rue Elle </street>
  <zip> 75000 </zip>
  <town> Paris </town>
</address>
```

Finalement, on peut également trouver un **encodage mixte** (une **partie externe** et une **partie**

interne):

adresse v2.xml

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd"
[
  <!ENTITY eacute "&#x00E9;" >
]>
<address>
  <street> 12, rue &eacute;t&eacute;; </street>
  <zip> 75000 </zip>
  <town> Paris </town>
</address>
```

4. Structure d'une DTD

4.1. Structures et imbrications des éléments (balises)

```
<!ELEMENT elementPere (premier_ElementFils , ... , n_eme_elementFils) >
```

Chaque élément fils *f* pourra apparaître un certain nombre de fois selon les éventuelles précisions suivantes:

f	une et une seule fois
f?	0 ou 1 fois (<i>optionnel</i>)
f*	0 ou n fois (<i>en nombre quelconque</i>)
f+	1 ou n fois (<i>au moins 1 fois</i>)

D'autre part la **virgule** qui sépare deux éléments fils consécutif détermine un **ordre imposé**.

Le trait vertical (ou pipe) | correspond à un **ou exclusif**.

Toutefois la construction **(a | b)*** autorisera l'apparition d'une *suite quelconque de a et de b*.

Quelques exemples:

```
<!ELEMENT personne (nom , prenom+ , adresse?) >
<!ELEMENT adresse (rue , cp , ville , pays?) >
```

```
<!ELEMENT livre (preface? , introduction? , chapitre+ , conclusion?) >
<!ELEMENT chapitre (titre , paragraphe+) >
```

Pour indiquer qu'une **balise terminale (ou feuille)** ne **comportera que du texte**, on utilise le mot clef **#PCDATA** signifiant *Parsed Character Data* (c'est à dire du texte pouvant comporter des

portions qui seront analysées (ex: é ==> é).

exemple:

```
<!ELEMENT titre (#PCDATA) >
```

D'autre part, certaines portions de texte peuvent éventuellement comporter des sous balises (telles que `<note ... />` ou ``). On parle alors d'**éléments mixtes** .

exemple:

```
<!ELEMENT p (#PCDATA | note)* >
```

autorise la construction mixte suivante:

```
<p> XML est un langage <note text="ou meta langage" /> pour encoder .... <p>
```

Le mot clef **EMPTY** permet d'indiquer qu'une **balise** doit absolument rester **vide** (pas de texte ni de sous balise, seuls des attributs sont autorisés [ex: `
`]).

exemple:

```
<!ELEMENT br EMPTY >
```

Finalement , le mot clef **ANY** signifiant un *sous élément quelconque* peut éventuellement être utilisé pour autoriser localement un contenu tout à fait libre:

exemple:

```
<!ELEMENT partie_libre (#PCDATA | ANY)* >
```

4.2. Liste d'attributs

De façon à préciser les **attributs possibles** de certaines balises, une **DTD** peut comporter les blocs introduits par `<!ATTLIST >` .

La syntaxe générale est la suivante:

```
<ATTLIST  nom_élément
           nom_attribut_1  type_attribut_1  caractéristique_attribut_1
           nom_attribut_N  type_attribut_N  caractéristique_attribut_N >
```

Les **types d'attributs** possibles sont les suivants:

CDATA	chaîne de caractères quelconque (ex: "ok" , "12" ,)
ID	Identifiant unique au sein du document (ex: <i>id</i> ="livre1" ou <i>name</i> ="livre2")
IDREF	Référence (renvoi) vers un ID (ex: <i>href</i> ="#livre2")

NMTOKEN	"Named Token" (nom pour entité du monde réel) (<i>ex.</i> 'fr' , 'de' , 'us' ,)
('Lu' 'Ma' ...)	Enumération de valeurs possibles
...	<i>Autres types: NOTATION , ENTITY , ENTITIES , NMTOKENS peu utilisés</i>

Les **caractéristiques (déclaration de valeurs par défaut)** sont les suivantes:

#IMPLIED	attribut facultatif (<i>pas obligatoirement présent</i>) et sans valeur par défaut
#FIXED 'Valeur'	attribut devant obligatoirement prendre une valeur fixe (constante)
#REQUIRED	attribut obligatoire (<i>doit absolument être renseigné</i>)
'Valeur_par_défaut'	valeur par défaut d'un attribut implicitement facultatif

exemples:

<!ATTLIST personne			
<i>name</i>	ID	#REQUIRED	
<i>age</i>	CDATA	#IMPLIED	
<i>id_adr</i>	IDREF	#IMPLIED	>

<!ATTLIST doc			
<i>xml:lang</i>	NMTOKEN	'fr'	
<i>xml:space</i>	('default','preserve')	'default'	>

4.3. Entités paramètres et inclusions de "sous DTD"

Une **entité paramètre** est un **alias** (référence à une valeur) qui **ne peut être utilisé dans la DTD elle même**.

Une entité paramètre est toujours déclarée après le symbole % :

```
<!ENTITY % DEFAULT "0" >
```

```
<!ENTITY % sous_dtd_adresse SYSTEM "../adresse.dtd" >
```

Pour faire référence à la valeur associée à l'entité paramètre, il faut faire appel à une construction du type **%NomEntiteParam;** (*ex.* **%DEFAULT;** **%sous_dtd_adresse;**)

NB: Une entité paramètre ne peut pas être utilisée au niveau de la valeur d'un attribut.

4.4. Sections conditionnelles (IGNORE & INCLUDE)

Les constructions

```
<![ INCLUDE [ <!ELEMENT elt (e1,e2) > ..... ]]>
```

et

```
<![ IGNORE [ <!ELEMENT elt (e1,e2,e3) > ... ]]>
```

permettent respectivement d'**inclure** ou d'**exclure** une définition d'élément(s) dans la dtd courante.

On aura généralement recours à des entités paramètres pour basculer d'une version à une autre:

Exemple:

AdresseAGeometrieVariable.dtd

```
<!ENTITY % V1 "IGNORE" >
<!ENTITY % V2 "INCLUDE" >
...
<![ %V1; [ <!ELEMENT address (#PCDATA) > ]]>
<![ %V2; [ <!ELEMENT address (street,zip,town) >
    <!ELEMENT street (#PCDATA) >
    <!ELEMENT zip (#PCDATA) >
    <!ELEMENT town (#PCDATA) > ]]>
```

adresse_v2.xml:

```
<?xml version="1.0" >
<!DOCTYPE address SYSTEM "AdresseAGeometrieVariable.dtd" >
<!-- Mon adresse conforme à la version V2 -->
<address>
    <street>12 , rue Elle</street>
    <zip>75000</zip>
    <town>Paris</town>
</address>
```

adresse_v1.xml:

```
<?xml version="1.0" >
<!DOCTYPE address SYSTEM "AdresseAGeometrieVariable.dtd"
[
    <!ENTITY % V1 "INCLUDE">
    <!ENTITY % V2 "IGNORE" >
]>
<!-- Mon adresse conforme à la version V1 -->
<address>12 , rue Elle - 75000 Paris </address>
```

NB: Les définitions **internes** (dans le fichier XML) des entités paramètres sont **prioritaires** par rapport à celles indiquées au sein du fichier dtd **externe**.

4.5. Exemple de DTD

monnaies.ent

```
<!ENTITY euro "&#8364;"> <!-- euro sign, U+20AC NEW -->
<!ENTITY livre "&#x00A3;">
```

produit.dtd

```
<?xml version="1.0" ?>
<!ENTITY % Monnaies SYSTEM "monnaies.ent">
%Monnaies;
<!ELEMENT produit (num, designation , prix) >
<!ELEMENT num (#PCDATA) >
<!ELEMENT designation (#PCDATA) >
<!ELEMENT prix (#PCDATA) >
```

produit.xml

```
<?xml version="1.0" ?>
<!DOCTYPE produit SYSTEM "produit.dtd" >
<produit>
  <num>123</num>
  <designation>Cahier - grands carreaux</designation>
  <prix>2 &euro;</prix>
</produit>
```

4.6. Points forts et limitations des DTD

Points forts:

- Utilisant un **formalisme très proche des grammaires** utilisées pour décrire la syntaxe des langages informatiques , les DTD sont assez **compactes** et **lisibles** .
- Normalisées dès 1998** en tant que partie interne de la version "1.0" d'XML les DTD sont supportées par des programmes anciens (et non pas seulement par des "*parseurs*" récents).

Limitations:

- La syntaxe portant sur les attributs est assez ésotérique (pour le non initié)
- Les DTD n'ont rien prévu pour imposer des types de données précis** (ex: String , Integer ,) au niveau des *valeurs textuelles* encadrées par des balises ou portées par les attributs.
- Les DTD sont quasi **incompatible** avec les **namespaces** car elles imposent une valeur précise sur les préfixes locaux alors que ceux n'ont aucune importance.

X - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP