

1. Introduction

1.1 Problématique des Systèmes d'Information

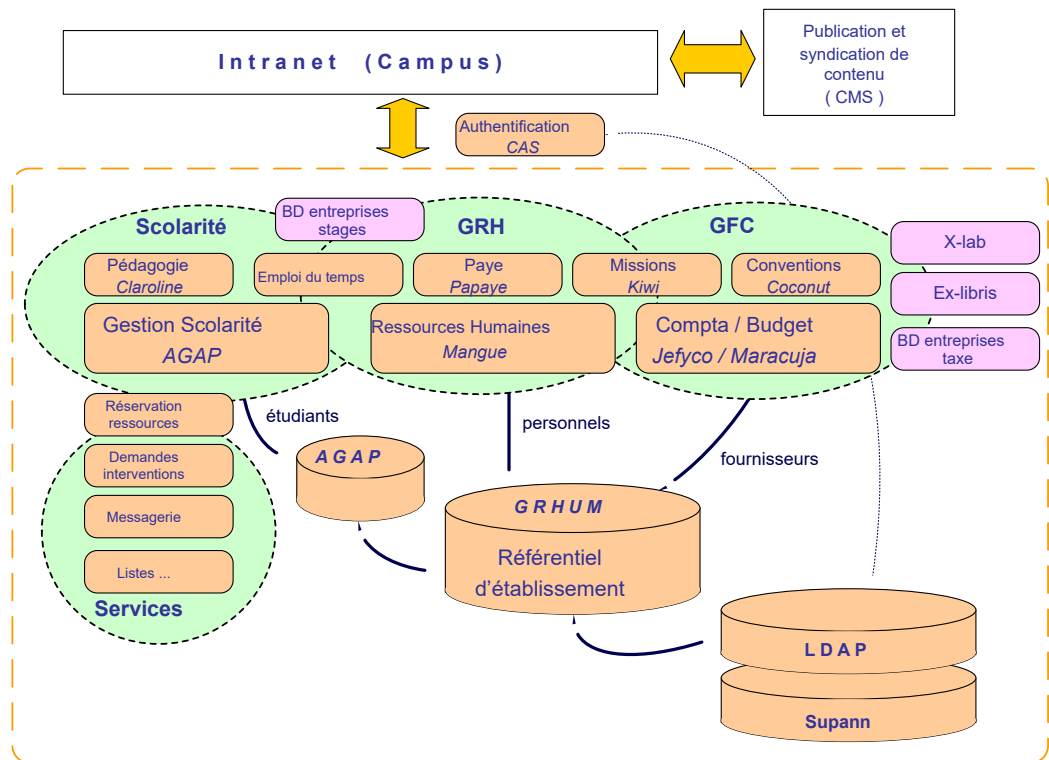
1.1.1 Qu'est-ce qu'un système d'information ?

D'après Wikipédia [http://fr.wikipedia.org/wiki/Syst%C3%A8me_d%27information], un **système d'information** (SI) représente l'ensemble des éléments participant à la gestion, au stockage, au traitement, au transport et à la diffusion de l'information au sein d'une organisation.

Cette définition englobe les éléments matériels (*serveurs, supports de stockage, ...*), logiciels (*applications, ...*), de communication (*réseau, ...*), les bases de données, les procédures (*workflow, ...*), voire le personnel.

Quelle que soit sa finalité (*production, gestion, ...*), un système d'information est donc un environnement **complexe** d'éléments en **interaction**.

Voici à titre d'exemple un schéma général du SI de l'Ecole Centrale fin 2006 :



Ce type de complexité est classique dans le domaine des **SI** d'entreprises.

1.1.2 Maîtriser la complexité

Au sein d'un **SI** se côtoient de nombreuses applications avec des finalités variées (*scolarité, gestion des ressources humaines, gestion financière et comptable, services*), qui communiquent entr'elles et partagent des données. Ces applications sont souvent basées sur des technologies diverses (*java, php, delphi, C, Cobol, ...*) et tirent leurs données de sources variées (*Oracle, Sybase, MySQL, LDAP, microservices...*).

Au niveau système la maîtrise de cette complexité repose sur les points suivants :

- les données sont fortement structurées,
- les données sont parfaitement identifiées (*cf. métadonnées*),
- les données sont toujours correctes (*vérifications d'intégrité*),
- les données ont une source unique (*transformation et non duplication*),
- les échanges se font via des formats publics (*architectures orientées services - SOA*).

On montrera dans la suite du cours comment **XML** et les technologies associées (*DTD, Schémas, Transformations, services Web*) répondent à cette problématique, et l'on comprendra dès lors pourquoi **XML** est omniprésent au cœur des **SI**.

1.1.3 Sommaire du cours

Voici donc les divers aspects qui seront abordés dans le cadre de ce cours, et comment ils se rattachent à la problématique générale des systèmes d'information :

- Introduction (*généralités, historique, tour d'horizon des standards et applications*),
- **XML**, éléments de syntaxe (*structuration des données*),
- **DTD**, schémas - validation (*intégrité des données*),
- **Namespaces** - espaces de nommage (*interopérabilité des applications*),
- **Xpath, Xquery** - recherche d'informations (*requêtage*),
- **XSLT** (*transformations*),
- **XML-RPC** - services Web (*échanges*),

1.2 Eléments d'histoire

1.2.1 A l'origine était SGML ...

SGML (*Standard Generalized Markup Language*) :

- est un langage créé chez IBM dans les années 70,
- ayant connu développement international,
- faisant l'objet du standard ISO 8879 en 1986.

SGML est :

- un langage sémantique et structurel à balises pour des documents "texte",
- destiné à la gestion de documentations techniques volumineuses (*plusieurs milliers de pages*).

SGML a entre autres donné lieu à des applications pour le gouvernement américain (*bibliothèque du congrès*), les militaires, le secteur aéronautique (*documentations techniques*).

1.2.2 De SGML à XML

> HTML : une application SGML

En 1992 Tim Berners-Lee invente le Web. **HTML** est conçu sous la forme d'une application **SGML**.

En 1995 la guerre des navigateurs fait rage (*HTML 2.0 - HTML 3.2*), sous la pression des utilisateurs et des concepteurs de navigateurs sans cesse à la recherche de nouvelles fonctionnalités.

> Une évolution de SGML pour le Web

Les évolutions incessantes de la syntaxe **HTML**, conduisent à rêver d'un langage qui permettrait d'inventer ses propres balises...

Malheureusement **SGML**, conçu dans les années 70 pour des applications batch, est tellement complexe et souvent redondant, qu'il n'est pas envisageable de pouvoir l'interpréter en temps réel au sein d'un navigateur.

En 1995, il n'existe aucune application implémentant l'ensemble des fonctionnalités de **SGML**.

En 1996 commencent les réflexions pour la définition d'une version simplifiée de **SGML** applicable au "temps réel" (*au sein d'un navigateur*).

Le produit de ces réflexions s'appellera **XML**.

> XML

Le développement s'est effectué avec les contraintes suivantes :

- **XML** doit pouvoir être facilement utilisé sur l'internet.
- **XML** doit pouvoir supporter des applications variées.
- **XML** doit être compatible avec **SGML**.
- Il doit être facile de développer des programmes qui traitent des documents **XML**.
- **XML** doit comporter un minimum d'éléments optionnels, idéalement zéro.
- Les documents **XML** doivent être lisibles et compréhensibles par des humains.
- Les spécifications de **XML** doivent être concises et précises.
- Les documents **XML** doivent être faciles à créer.
- La compacité des documents **XML** n'est pas d'une importance fondamentale.

Le premier draft **XML** a été présenté lors de la conférence **SGML 96** à Boston en **novembre 1996**.

La première recommandation (*XML 1.0*) a été émise en **février 1998**.

Les spécifications de **XML 1.0** ont subi plusieurs révisions. La quatrième date de **septembre 2006**.

🔗 Vers la recommandation XML 1.0 la plus récente en cours.

[\[http://www.w3.org/TR/REC-xml\]](http://www.w3.org/TR/REC-xml)

Sans remplacer la version 1.0 qui reste toujours recommandée, **XML 1.1** (*seconde version, septembre 2006*) prend en compte l'évolution du standard **Unicode** pour autoriser des caractères nouveaux pour les **noms XML** (*balises, attributs, ...*).

🔗 Vers la recommandation XML 1.1 la plus récente en cours.

[\[http://www.w3.org/TR/xml11\]](http://www.w3.org/TR/xml11)

1.2.3 En résumé

XML est une solution :

- qui s'appuie sur l'expérience **SGML**,
- constituant une simplification de **SGML**,
- faisant l'objet d'une recommandation [\[http://www.w3.org/TR/REC-xml\]](http://www.w3.org/TR/REC-xml) de la part du **W3C**,
- issue de la communauté "World-Wide-Web",
- enrichie depuis 1998 par de nombreuses extensions,
- suffisamment générique pour offrir des applications dans de nombreux domaines, qui n'ont souvent plus aucune relation directe avec le Web.

1.3 XML en 10 points

N.B. Le chapitre "XML en 10 points" a été élaboré d'après un document W3C.

[<http://www.w3.org/XML/1999/XML-in-10-points.fr.html>]

1.3.1 Une méthode pour structurer des données

XML n'est pas un langage ...

XML est un ensemble de règles pour la conception de formats texte permettant de structurer des données : *feuilles de calcul, carnets d'adresses, paramètres de configuration, transactions financières, dessins techniques...*

- permet la définition de formats non ambigus, extensibles, insensibles aux problèmes d'internationalisation/localisation,
- résout les problèmes de dépendance par rapport à certaines plates-formes,
- est conforme à Unicode.

1.3.2 XML ressemble à HTML

Comme **HTML**, **XML** identifie des éléments délimités par des balises de début et de fin, éventuellement affectés d'attributs :

```
<élément attribut="valeur">Contenu de l'élément</élément>
```

Toutefois :

- **HTML** possède un jeu de balises et d'attributs bien définis,
- avec **XML**, l'interprétation d'une balise dépend de l'application :

```
<article>
  <titre>
    Introduction à XML
  </titre>
  . . .
</article>
```

```
<personne>
  <titre>
    Altesse Sérénissime
  </titre>
  . . .
</personne>
```

1.3.3 XML : du texte pas forcément lu... par des humains

Un fichier **XML** est un fichier texte : l'édition et la correction sont donc possibles à l'aide d'un simple éditeur.

Ces opérations sont toutefois réservées à des experts car la syntaxe des applications est souvent très stricte (*bien plus que pour HTML*).

Les spécifications sont très explicites : une syntaxe approximative doit obligatoirement provoquer l'arrêt de l'application (*balise ouvrante non fermée, valeur d'un attribut sans guillemets, ...*).

1.3.4 XML est verbeux, mais ce n'est pas un problème

Un fichier **XML** n'est pas conçu pour optimiser l'espace de stockage (*fichier texte + balises*) :

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xml/intro/xml-10/point4-exemple.fr.html>]

Toutefois, cet aspect est pleinement assumé par les concepteurs.

En effet, les avantages du format texte sont évidents (*cf. point précédent*) et les inconvénients sont limités :

- l'espace disque est bon marché,
- il existe des possibilités de compression rapides et efficaces (*zip, gzip...*),
- certains protocoles de communication compressent à la volée (*modems, HTTP/1.1...*)

1.3.5 XML est une famille de technologies

XML définit ce qu'est une balise ou un attribut, mais ne possède pas de balises ni d'attributs en propre (*XML n'est pas un langage*).

XML a mis en place un cadre permettant le développement d'outils génériques et la spécification d'applications :

- parseurs, validateurs (*DTD, Schémas XML*), navigateurs (*CSS*),
- web (*XHTML*),
- moteurs de transformation XML vers XML (*XSL*), publication (*XSL-FO*),
- liens et sélecteurs (*XPath, XLink, XPointer*),
- protocoles de communication (*WebDAV, XMPP*), services Web (*XML-RPC, Soap*)...

1.3.6 XML est une technologie récente, mais éprouvée

Les premiers efforts de développement ont débuté en 1996.

Les premières spécifications datent de 1998.

Une technologie immature ?

Non, car basée sur l'expérience **SGML** (*années 70/80*) et **HTML** (*depuis 1990*).

1.3.7 XML fait passer HTML à XHTML

HTML 4.0 est une application **SGML**.

XHTML 1.0 est une application **XML**.

Les modifications syntaxiques sont mineures (*guillemets, balises fermantes, ...*).

Il est possible d'assurer la compatibilité avec les navigateurs pré-XML :

```

```

1.3.8 XML est modulaire

Un même document **XML** peut mêler des syntaxes provenant d'applications diverses :

en théorie un document web peut par exemple contenir à la fois des balises **XHTML**, des balises **MathML** (*formules mathématiques*), des balises **SVG** (*graphiques vectoriels*) et, pourquoi pas des balises **SMIL** (*animations*).

Les conflits potentiels entre les noms de balises et d'attributs provenant d'applications différentes sont gérés grâce aux espaces de noms (*XML Namespaces*).

1.3.9 XML est le fondement de RDF et du Web Sémantique

RDF (*Resource Description Framework*) est une recommandation du **W3C** pour les métadonnées au format **XML**.

Le **Web Sémantique** est une activité du **W3C** visant à permettre le développement d'applications réparties (*moteurs de recherche, e-commerce, annuaires...*) basées sur l'exploitation des métadonnées incluses au sein de documents **XML** disponibles en ligne.

1.3.10 XML est libre de droits, indépendant des plates-formes et correctement supporté

Les spécifications de **XML** sont publiques et libres de droit.

Chacune des technologies du **W3C** doit donner lieu à plusieurs implémentations indépendantes avant d'être recommandée (*i.e. publiée sous forme d'une recommandation*).

L'utilisation de technologies **XML** facilite le développement d'applications grâce à la disponibilité de modules logiciels standards, bien testés et maintenus, souvent eux-mêmes libres de droits (*parseurs, validateurs, moteurs de transformation, moteurs de recherche, éditeurs...*).

1.4 Un survol des standards

1.4.1 Définition de Type de Document

Dès l'origine, comme **SGML**, **XML** permet de spécifier la syntaxe d'une **application XML** (*nom des balises, des attributs, type de contenu, valeur par défaut..*) à l'aide d'une **DTD** (*Document Type Definition*).

La syntaxe d'une **DTD**, héritée de **SGML**, est particulière :

```
<!DOCTYPE classe [  
  <!ELEMENT classe (promo,etudiant+)>  
  <!ELEMENT etudiant (nom, prenom)>  
  <!ELEMENT nom (#PCDATA)>  
  <!ELEMENT prenom (#PCDATA)>  
  <!ELEMENT promo (#PCDATA)>  

```

Un **document XML** qui possède une **DTD** et dont le contenu est conforme à sa **DTD** est un **document XML valide** (*cf. sens anglo-saxon*).

Il existe des outils génériques permettant de **valider** (*i.e. vérifier la validité*) un document **XML** quelconque.

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/std/dtd-validation.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/std/dtd-validation.html)

 Vers le cours sur les DTD

[\[http://dmolinarius.github.io/demofiles/mod-84/dtd.pdf\]](http://dmolinarius.github.io/demofiles/mod-84/dtd.pdf)

1.4.2 Espaces de Noms - XML Namespaces

 Site de référence

[\[http://www.w3.org/XML/Core/\]](http://www.w3.org/XML/Core/)

Les espaces de noms (*XML Namespaces*) ont très tôt fait l'objet d'une recommandation du **W3C** (*janvier 1999*). Ils permettent d'éviter les conflits potentiels entre les noms de balises et d'attributs lorsqu'on désire construire des documents mêlant des vocabulaires provenant d'applications différentes :

```
<article>
  <titre>
    Introduction à XML
  </titre>
  . . .
</article>
```

```
<personne>
  <titre>
    Altesse Sérénissime
  </titre>
  . . .
</personne>
```

Pour cela, le domaine de validité de chaque nom (*éléments et attributs*) est précisé grâce à un préfixe :

```
<publication:article>
  <publication:titre>Introduction à XML</publication:titre>
  <publication:auteur>
    <personne:titre>Prof.</personne:titre>
    <personne:prenom>Daniel</personne:prenom>
    <personne:nom>Muller</personne:nom>
  </publication:auteur>
</publication:article>
```

N.B. Ce n'est pas le préfixe qui identifie l'espace de noms, mais un **URI** associé au préfixe. Ces aspects seront vus dans le chapitre sur les espaces de noms.

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/std/xmlns-exemple.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/std/xmlns-exemple.html)

 Vers le cours sur les espaces de noms

[\[http://dmolinarius.github.io/demofiles/mod-84/xmlns.pdf\]](http://dmolinarius.github.io/demofiles/mod-84/xmlns.pdf)

1.4.3 Schémas XML

 Site de référence

[\[http://www.w3.org/XML/Schema\]](http://www.w3.org/XML/Schema)

Espace de noms : <http://www.w3.org/2001/XMLSchema>

Préfixe courant : *xs*

Une **DTD** permet, aux fins de validation, de définir la syntaxe d'une application en indiquant le nom des balises et des attributs autorisés, le contexte dans lequel ils sont autorisés, ainsi que les valeurs autorisées pour certains attributs.

Toutefois, les **DTD** présentent plusieurs défauts, en effet : leur syntaxe est particulière (*non-XML, héritée de SGML*), elles décrivent uniquement la structure du document et non son contenu, et les données ne sont pas typées (*elles sont toutes considérées comme des chaînes de caractères*).

XML Schema est une **application XML** spécifiée par le **W3C** (*mai 2001*) qui comble ces lacunes.

Voici un extrait de schéma :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="page">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="titre" minOccurs="1" maxOccurs="1"/>
        <xs:group ref="contenuPage" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="pageID" use="required"/>
    </xs:complexType>
  </xs:element>
  . . .
</xs:schema>
```

Exemple détaillé :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/std/xsd-exemple.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/std/xsd-exemple.html)

📄 Vers le cours sur les schémas XML

[\[http://dmolinarius.github.io/demofiles/mod-84/xsd.pdf\]](http://dmolinarius.github.io/demofiles/mod-84/xsd.pdf)

1.4.4 Feuilles de style CSS

📄 Site de référence

[\[http://www.w3.org/Style/CSS/\]](http://www.w3.org/Style/CSS/)

Les navigateurs récents sont compatibles **XML**. Ceci signifie de prime abord qu'ils sont capables d'afficher un document en visualisant l'**arbre XML**.

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/std/css.xml\]](http://dmolinarius.github.io/demofiles/mod-84/xml/std/css.xml)

Toutefois, contrairement au cas de **HTML**, la signification des éléments et des attributs leur est totalement étrangère.

Pour afficher un tel document de manière plus ergonomique qu'un arbre il est nécessaire de disposer d'une **feuille de style** indiquant quel doit être le format d'affichage de **chacun des éléments** du document.

Le langage **CSS** (*Cascading Style Sheets*) est un langage de feuilles de style, toujours en évolution à l'heure actuelle (2004), et qui a fait l'objet de plusieurs recommandations du **W3C** depuis 1996.

CSS est compatible **XML** et correctement reconnu par les navigateurs :

```
classe {
    display: block;
    padding: 8px;
    line-height: 1.33;
    background-color: #FFCC66;
    font-family: Arial, Helvetica, Sans-serif;
    color: black;
}
```

Si le document **XML** référence la feuille de style, il est affiché correctement par le navigateur :

```
<?xml version="1.0" ?>
<?xml-stylesheet href="css-exemple.css" type="text/css"?>
<classe>
.
.
.
</classe>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/std/css-exemple.xml\]](http://dmolinarius.github.io/demofiles/mod-84/xml/std/css-exemple.xml)

Toutefois, les feuilles de style à mettre en oeuvre sont bien plus complexes que celles afférentes à un document **HTML** puisque contrairement à ce qui se passe pour ces dernières on ne peut pas se contenter de donner les écarts par rapport à un comportement par défaut du navigateur.

Les feuilles de style **CSS** qui s'appliquent à des documents **XML**, doivent préciser l'ensemble des caractéristiques permettant d'afficher chacun des éléments (*type, marges, couleurs, polices ...*).

A remarquer également que si le style est précisé par la feuille **CSS**, le contenu de éléments et leur ordre d'apparition à l'écran sont strictement imposés par le **document source**.

1.4.5 Feuilles de style XSL

📄 Site de référence

[\[http://www.w3.org/Style/XSL/\]](http://www.w3.org/Style/XSL/)


CSS possède des inconvénients : la syntaxe est très spécifique (*non-XML*) et surtout, il n'est pas possible de transformer le document (*changer l'ordre des éléments, choisir de ne pas en afficher certains, générer du contenu ...*).

Ces remarques ont conduit le **W3C** au développement d'un mécanisme de feuilles de style pour **XML** nommé **XSL** (*XML Style Sheets*).

XSL a été conçu en séparant ses deux fonctionnalités principales :

- **XSLT** (*XSL Transformation*), une **application XML** générique qui permet (*moyennant une feuille de style XSL*) de transformer tout **document XML** (*application x*) en un autre (*application y*),
- **XSL-FO** (*XSL Formatting Objects*), une **application XML** décrivant le contenu et l'aspect visuel d'un document, basée sur les fonctionnalités de **CSS-2** associées à une syntaxe **XML**.

1.4.6 Transformation de documents

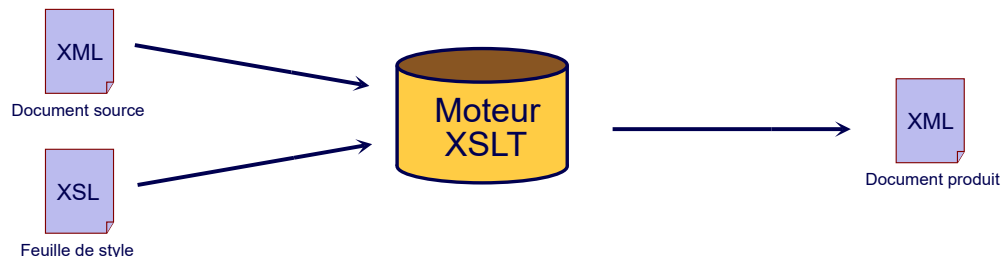
 Site de référence

[<http://www.w3.org/Style/XSL/>]

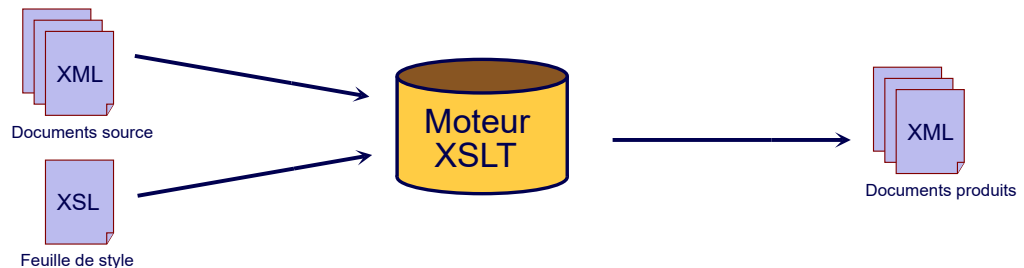
Espace de noms : <http://www.w3.org/1999/XSL/Transform>

Préfixe courant : *xsl*

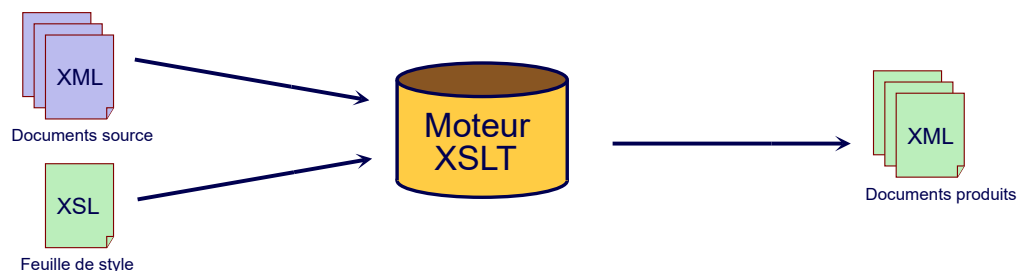
XSLT est une **application XML** recommandée par le **W3C** (*novembre 1999*) permettant de transformer un document **XML** en un autre document, **XML** ou non, à l'aide de règles spécifiées dans une feuille de style :



L'intérêt du procédé réside dans le fait qu'un nombre important de documents sources peut être transformé avec la même feuille de style, afin de produire, à des fins de publication, des documents possédant une identité visuelle commune :



Il suffit ensuite, sans toucher aux sources d'information, de modifier la feuille de style, pour modifier la présentation des documents produits :



1.4.7 XSL Formatting Objects

🔗 Site de référence

[\[http://www.w3.org/Style/XSL/\]](http://www.w3.org/Style/XSL/)

Espace de noms : <http://www.w3.org/1999/XSL/Format>

Préfixe courant : *fo*

XSL-FO est une **application XML** recommandée par le **W3C** (*version 1.0 en octobre 2001, 1.1 en décembre 2006*) basée sur les fonctionnalités de **CSS-2**, décrivant le contenu et l'aspect visuel d'un document :

```
<fo:root>
  <fo:layout-master-set>
    <fo:simple-page-master master-name="simple"
      margin-left="1.5cm" margin-right="1.5cm"
      margin-top="0.5cm" margin-bottom="0.5cm"
      page-width="21cm" page-height="29.7cm">
      . . .
    </fo:simple-page-master>
  </fo:layout-master-set>
  <fo:page-sequence master-reference="simple">
    <fo:static-content flow-name="xsl-region-after"
      font-family="Arial" font-size="8pt">
      . . .
    <fo:block text-align="left"
      space-before="1.0em" space-after="1.5em"
      color="#000088" font-family="Arial"
      font-weight="bold" font-size="16pt">
      1. Publication de cours avec XML
    </fo:block>
    . . .
  </fo:static-content>
</fo:page-sequence>
</fo:root>
```

Un document **XSL-FO** comprend toutes les informations nécessaires à la visualisation ou à l'impression d'un document. Sémantiquement, **XSL-FO** peut de ce fait être comparé à des langages comme **Postscript** ou **pdf**.

Il existe malheureusement très peu d'environnements logiciels capables de rendre **XSL-FO** de manière native, (*navigateurs ?*), et il n'existe en particulier aucune imprimante acceptant directement ce langage.

Les applications existantes permettent en général de transformer un document **XSL-FO** en **pdf** (*ou autres formats*) aux fins d'impression (*cf. fop*), éventuellement après pré-visualisation sur écran (*cf. Antenna House XSL Formatter*).

1.4.8 Sélecteurs XPath

🔗 Site de référence

[\[http://www.w3.org/Style/XSL/\]](http://www.w3.org/Style/XSL/)

Pour désigner les éléments auxquels doit s'appliquer une certaine règle de mise en forme, **CSS** utilise des **sélecteurs**.

De manière similaire, **XSLT** nécessite un mécanisme pour désigner les éléments à transformer. Ce mécanisme, recommandé par le **W3C** (*1.0 en novembre 1999, 2.0 en janvier 2007*), s'appelle **XPath** :

```
<xsl:apply-templates
  select="child::node()[starts-with(name(),'menu')]" />
<xsl:apply-templates select="logo" />
<xsl:apply-templates
  select="contenu|contenu_large|special" />
```

XPath a été conçu sous la forme d'un langage générique pour désigner des ensembles de noeuds (*nodeset*) d'un **arbre XML**. Ce langage est également utilisé par d'autres applications comme **XPointer**.

1.4.9 Développement SAX / DOM

 Site de référence SAX

[\[http://www.saxproject.org/\]](http://www.saxproject.org/)

 Site de référence DOM

[\[http://www.w3.org/DOM/\]](http://www.w3.org/DOM/)

Le module logiciel permettant de lire un **document XML** s'appelle un **parseur**.

Un **parseur XML** vérifie obligatoirement le caractère bien formé d'un document (*well-formedness*). La plupart des parseurs sont également capables de valider un document par rapport à une **DTD** ou un **schéma**.

Il y a de nombreux parseurs disponibles (*dont certains en open-source*) avec lesquels il est possible de s'interfacer lorsque l'on développe un logiciel qui travaille avec des documents **XML**.

Il existe deux façons différentes de s'interfacer avec un parseur : approche "événements" (*API SAX - simple API for XML*) ou travail sur l'arbre en mémoire (*API DOM - Document Object Model*). **SAX** est un standard de fait, **DOM** une recommandation du **W3C** (*level 1 oct. 1998, nov. 2000, level 2 nov. 2000, level 3 avril 2004*).

N.B. Il existe des parseurs **XML** pour tous les langages de programmation courants **C/C++**, **Java**, **perl**, **python**, **php**...

1.4.10 Pointeurs avec XPointer

 Site de référence

[\[http://www.w3.org/XML/Linking\]](http://www.w3.org/XML/Linking)

XPointer est une recommandation du **W3C** (*mars 2003*) qui définit le langage à utiliser au sein d'un **identifiant de fragment** pour référencer les ressources du type **text/xml** ou **application/xml**.

Cette méthode permet de désigner des parties de document en se basant sur une expression **XPath** :

```
#xpointer(id('table-principale')/tr[2]/td[1])
```

XPointer est un mécanisme générique utilisé par de nombreuses applications **XML** comme **XLink**, **XInclude**, **RDF** ou **SOAP**.

1.4.11 Liens avec XLink

 Site de référence

[\[http://www.w3.org/XML/Linking\]](http://www.w3.org/XML/Linking)

Espace de noms : <http://www.w3.org/1999/xlink>

Préfixe courant : *xlink*

XLink est une recommandation du **W3C** (*juin 2001*) qui décrit les éléments à insérer au sein de **documents XML** afin de mettre en place des liens entre ressources.

XLink est basé sur une **syntaxe XML** et autorise de simples liens unidirectionnels comme en **HTML** (*cf. élément A*), ou des liens plus sophistiqués (*bidirectionnels, multiples...*).

```
<etudiants
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:type="simple"
  xlink:href="promo_2005.xml"
  xlink:role="http://tic01.tic.ec-lyon.fr/annuaire_promo.xml"
  xlink:title="liste d'etudiants"
  xlink:show="new"
  xlink:actuate="onRequest">
  la promo 2005
</etudiants>
```

1.4.12 Avertissement

Le survol de quelques standards associés à **XML** qui vient d'être proposé est largement non exhaustif.

En effet, outre le fait que les technologies présentées sont pour la plupart encore en évolution (*nouvelles versions plus évoluées en cours de gestation*), il existe bien d'autres applications recommandées par le **W3C**, par d'autres organismes (*cf. OASIS*), voire normalisés.

Pour en suivre les évolutions, il est bon de connaître quelques sites de référence :

🔗 XML au W3C

[\[https://www.w3.org/standards/xml/\]](https://www.w3.org/standards/xml/)

🔗 Le consortium OASIS

[\[http://www.oasis-open.org/\]](http://www.oasis-open.org/)

🔗 www.xml.org

[\[http://www.xml.org/\]](http://www.xml.org/)

1.5 Exemples d'applications

1.5.1 Format Graphique X3D

🔗 Site de référence

[\[http://www.web3d.org/x3d/\]](http://www.web3d.org/x3d/)

Espace de noms : non

X3D est une **application XML** faisant l'objet d'un **Standard ISO** pour la représentation interactive d'objets et de scènes 3D.

X3D est développé par le consortium **Web3D**.

🔗 Vers le site du consortium Web3D

[\[http://www.web3d.org\]](http://www.web3d.org)

Les outils existants sont nombreux et vont de l'environnement auteur aux applications de rendu.

🔗 Ressources X3D

[\[http://www.web3d.org/x3d/content/examples/X3dResources.html\]](http://www.web3d.org/x3d/content/examples/X3dResources.html)

Les domaines d'application de **X3D** vont de la CAO mécanique au médical en passant par l'enseignement, les systèmes géographiques (*y compris extra-terrestres*), et les tutoriels de maintenance ou de sécurité.

1.5.2 Format Graphique SVG

🔗 Site de référence

[\[http://www.w3.org/Graphics/SVG/\]](http://www.w3.org/Graphics/SVG/)

Espace de noms : <http://www.w3.org/2000/svg>

*Préfixe courant : **svg***

SVG (*Scalable Vector Graphics*) est une **application XML** pour la description de **graphiques vectoriels** recommandée par le **W3C** (SVG 1.0 09/2001 remplacé par SVG 1.1 01/2003, et WD SVG 1.2 04/2004).

Historiquement, **SVG** a été fortement supporté par **Adobe** qui distribuait gratuitement un viewer sous forme de plugin pour la plupart des navigateurs.


A l'heure actuelle, les éléments **SVG** font partie intégrante de **HTML5** et sont nativement supportés par les navigateurs.

De nombreux outils sont par ailleurs disponibles pour **SVG** (*viewers dédiés, plugins navigateurs, viewers pour mobiles, éditeurs natifs, logiciels exportant SVG, logiciels de conversion de format, génération dynamique côté serveur*).

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide3-exemple1.svg>]

1.5.3 Format Multimédia SMIL

 Site de référence

[<http://www.w3.org/AudioVideo/>]

Espace de noms : <http://www.w3.org/2001/SMIL20/Language>


Préfixe courant : *smil*

SMIL (*Simple Multimedia Integration Language*) est une **application XML** pour l'intégration et la synchronisation de sources multimédia interactives (*vidéo, son, graphiques animés, texte, ...*) recommandée par le **W3C** (SMIL 1.0 06/1998 - SMIL 2.0 08/2001).

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <par dur="15s">
      
    </par>
  </body>
</smil>
```

SMIL est bien supporté par le marché et effectivement implémenté par de nombreux viewers (*Real Player G2, Internet Explorer, Adobe SVG Viewer, Apple Quicktime, ...*)

1.5.4 Format d'échange WebDAV

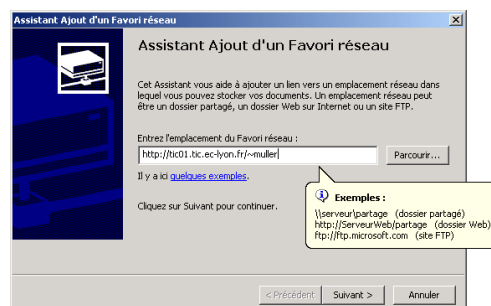
 Site de référence

[<http://www.webdav.org/>]

Espace de noms : DAV:

Préfixe courant : *D*

WebDAV est l'un des protocoles implémentés par l'outil "favoris réseau" de Microsoft Windows.



WebDAV (*Web-based Distributed Authoring and Versioning*) est une extension du protocole **HTTP** qui permet d'éditer et de maintenir un jeu de documents situé sur un serveur Web distant (RFC 2518 - 02/1999).

WebDAV s'appuie sur **XML** pour bénéficier de l'extensibilité qu'il procure, et le support des jeux de caractères **ISO 10646 (118N)**.

> Exemple de requête

```
PROPFIND /~muller/ HTTP/1.1
Host: tic01.tic.ec-lyon.fr
Depth: 1
Content-Type: text/xml; charset="utf-8"
Content-Length: 98
<?xml version="1.0" encoding="utf-8" ?>
<D:propfind xmlns:D="DAV:">
  <D:allprop/>
</D:propfind>
```

> Exemple de réponse

(noter au passage l'usage intensif des espaces de noms ...)

```
HTTP/1.1 207 Multi-Status
Date: Fri, 22 Nov 2002 13:05:43 GMT
Server: Apache/1.3.27 (Unix) PHP/4.2.3 DAV/1.0.3 mod_ssl/2.8.11 OpenSSL/0.9.6g
Transfer-Encoding: chunked
Content-Type: text/xml; charset="utf-8"
ec5
<?xml version="1.0" encoding="utf-8"?>
<D:multistatus xmlns:D="DAV:">
  <D:response xmlns:lp0="DAV:" xmlns:lp1="http://apache.org/dav/props/">
    <D:href>/~muller/images/</D:href>
    <D:propstat>
      <D:prop>
        <lp0:creationdate b:dt="dateTime.tz"
          xmlns:b="urn:uuid:c2f41010-65b3-11d1-a29f-00aa00c14882/">
          2002-10-09T10:58:02Z
        </lp0:creationdate>
        <lp0:getlastmodified b:dt="dateTime.rfc1123"
          xmlns:b="urn:uuid:c2f41010-65b3-11d1-a29f-00aa00c14882/">
          Wed, 09 Oct 2002 10:55:49 GMT
        </lp0:getlastmodified>
        <lp0:getetag>"32d61-1000-3da40b35"</lp0:getetag>
        <D:resourcetype><D:collection/></D:resourcetype>
        <D:getcontenttype>httpd/unix-directory</D:getcontenttype>
      </D:prop>
      <D:status>HTTP/1.1 200 OK</D:status>
    </D:propstat>
  </D:response>
  . . .
</D:multistatus>
```

1.5.5 Format d'échange XML RPC

 Site de référence

[<http://www.xmlrpc.com>]

XML RPC (*Remote Procedure Call*) est un mécanisme qui permet l'exécution de procédures distantes sur plates-formes hétérogènes via l'Internet, initié par Userland.

XML RPC s'appuie sur **HTTP** (pour le transport) et **XML** (pour le codage des données), et ouvre la porte aux **Web services**...

> Exemple de requête

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
<?xml version="1.0"?>
<methodCall>
  <methodName>NomDepartement</methodName>
  <params>
    <param>
      <value>69</value>
    </param>
  </params>
</methodCall>
```

> Exemple de réponse

```
HTTP/1.1 200 OK
Date: Tue, 14 Sep 2004 21:34:38 GMT
Server: Apache/1.3.28
X-Powered-By: PHP/4.3.3
Connection: close
Content-Length: 170
Content-Type: text/xml
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <string>Rh&#xF4;ne</string>
      </value>
    </param>
  </params>
</methodResponse>
```

1.5.6 Le Consortium OASIS

 Site de référence

[\[http://www.oasis-open.org\]](http://www.oasis-open.org)

OASIS (*Organization for the Advancement of Structured Information Standards*) est un "consortium global" à but non lucratif pour le développement, la convergence et l'adoption de standards e-business issu en 1998 de **SGML Open**, lui-même fondé en 1993.

 Vers la liste des membres

[\[http://www.oasis-open.org/about/\]](http://www.oasis-open.org/about/)

OASIS opère **www.xml.org** (référence pour les applications de XML & Web-services) ainsi que le site **xml.coverpages.org** (ressources en ligne sur les langages à balises).

 Vers le site www.xml.org

[\[http://www.xml.org\]](http://www.xml.org)

 Vers le site xml.coverpages.org


[\[http://xml.coverpages.org\]](http://xml.coverpages.org)

Voici quelques exemples de **Comités Techniques OASIS** : e-Government, Election and Voter Services, HumanMarkup, LegalXML,...

1.6 XHTML

1.6.1 Une reformulation de HTML 4.0 en XML 1.0

La spécification **XHTML 1.0** est une recommandation du **W3C** (janvier 2000 rev. août 2002).

 En savoir plus

[\[http://www.w3.org/TR/xhtml1/\]](http://www.w3.org/TR/xhtml1/)

Cette spécification définit **XHTML 1.0** comme une reformulation de **HTML 4** en une application **XML 1.0**, avec trois **DTD** (*Définitions de Type de Document*) correspondant à celles définies par **HTML 4** (*Strict, Transitional, Frameset*).

La compatibilité de **XHTML 1.0** avec les agents utilisateurs **HTML** (*navigateurs*) actuels ou plus anciens, est possible en suivant un ensemble raisonnable de règles.

1.6.2 Différences avec HTML 4.0

Un document **XHTML** doit être bien formé au sens de **XML** : tous les éléments doivent être correctement emboîtés.

XML et donc **XHTML** sont sensibles à la casse. Par convention les noms d'éléments et d'attributs **XHTML** sont en casse minuscule.

```
<html>
  <head>
    ...
  </head>
  <body>
    ... les noms d'éléments et d'attributs sont en minuscule ...
  </body>
</html>
```

La balise de fin est obligatoire.

```
<p>
  Les balises de fin sont obligatoires...
  même pour les éléments vides.
</p>
</img>
```

Les valeurs d'attributs doivent toujours être entre guillemets.

```
<table border="0">
  ...
</table>
```

En **HTML** certains attributs (*dits 'minimisés'*) ne prennent pas de valeur (cf. *CHECKED* et *NOWRAP*). **XML** ne supporte pas la minimisation de l'attribut.

```
<td nowrap="nowrap">
  Les attributs minimisés prennent leur nom pour valeur
</td>
```

La balise de fin des éléments vides peut être remplacée par un raccourci typographique. Les deux notations suivantes sont strictement équivalentes :

```
</img>

```

1.6.3 Problème des éléments **SCRIPT** et **STYLE**

En **XHTML**, les éléments **script** et **style** sont déclarés comme ayant un contenu de type **#PCDATA**, c'est à dire que le contenu des ces éléments est analysé par le client. Par suite, les caractères **<** et **&** seront traités comme le début d'un balisage.

Une première solution consiste à protéger ces caractères par des appels d'entité comme **<** ou **&** :

```
<script type="text/javascript">
  for ( i=0; i &lt; 10; i++ ) { ... }
</script>
```

La solution **XML** consisterait à emballer le contenu des éléments **script** à l'intérieur d'une section marquée **CDATA** ce qui évite l'interprétation de ces entités par les parseurs :

```
<script type="text/javascript">
  <![CDATA[
    ... Contenu de la script ...
  ]]>
</script>
```

Toutefois, si elle convient du point de vue de **XML**, cette solution pose problème car la déclaration **CDATA** n'est pas du code **Javascript** correct.

Démonstration et solution :

[\[http://dmolinarius.github.io/demofiles/mod-84/html/xhtml1/slide3-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/html/xhtml1/slide3-exemple1.html)

1.6.4 Problème des attributs **NAME** et **ID**

HTML 4 a défini l'attribut **name** (cf. éléments *A*, *applet*, *form*, *frame*, *iframe*, *img*, et *map*). De plus, **HTML 4** a également introduit l'attribut **id** qui peut s'appliquer entre autres à ces éléments-là.

En **XML**, les identificateurs partiels sont de type **ID**, et il ne peut y avoir qu'un seul identifiant de ce type par élément.

Un document **XHTML** bien formé doit utiliser de préférence l'attribut **id** et non pas **name** pour identifier les éléments listés ci-dessus.

```

```

Le cas des champs de formulaire (*input*, *select*, *textarea*...) est particulier puisque **id** et **name** ont un sens différent. Ces éléments peuvent porter les deux attributs, mais **XHTML** impose que leur valeur soit identique :

```
<input id="nom" name="nom" type="text"></input>
```

2. XML - Éléments de syntaxe

2.1 Généralités

2.1.1 Éléments, balises et contenu

Rappel :

Un fichier **XML** contient du texte, jamais de données binaires.

Exemple :

Le code ci-dessous est un document **XML** complet comportant un seul élément (*balises + contenu*) :

```
<etudiant>  
    Raymond Deubaze  
</etudiant>
```

L'élément 'etudiant' comporte une **balise de début**, une **balise de fin** et un **contenu**, constitué du texte 'Raymond Deubaze' et des espaces (*blancs, tabulations, retours à la ligne*) contenus entre les balises de début et de fin.

N.B. Même si la plupart des applications choisiront d'ignorer les espaces initiaux et finaux, ceux-ci sont significatifs pour **XML**.

2.1.2 Éléments vides

En **XML** les balises de fin sont obligatoires, y compris lorsqu'un élément ne possède pas de contenu :

```
<br></br>
```

Il existe toutefois une syntaxe particulière pour refermer la balise d'un élément vide :

```
<br />
```

Les notations ci-dessus sont autorisées toutes les deux et sont strictement équivalentes.

2.1.3 Sensibilité à la casse

Contrairement à **HTML**, **XML** est sensible à la casse.

Les trois exemples ci-dessous font donc appel à trois éléments distincts :

```
<etudiant>  
    Raymond Deubaze  
</etudiant>
```

```
<Etudiant>  
    Raymond Deubaze  
</Etudiant>
```

```
<ETUDIANT>  
    Raymond Deubaze  
</ETUDIANT>
```

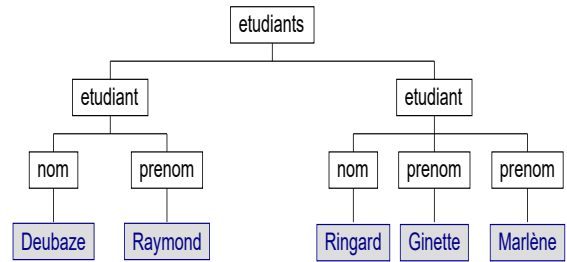
et le document ci-dessous comportera une erreur de syntaxe :

```
<etudiant>  
    Raymond Deubaze  
</Etudiant>
```

2.1.4 Arbre XML

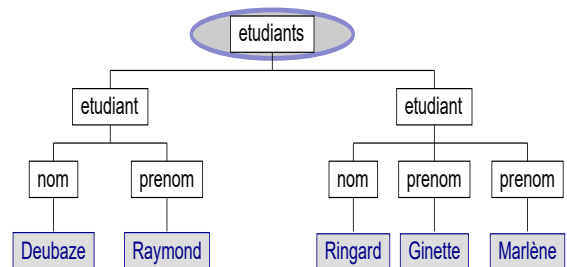
Les éléments d'un document **XML** forment une structure arborescente :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```



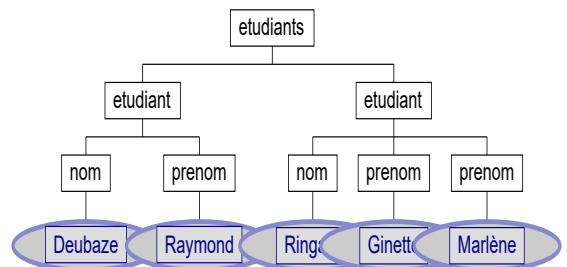
La description des éléments d'un document **XML** emprunte ainsi son vocabulaire aux arbres :
un document comporte donc un **élément racine** :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```



et certaines feuilles de l'arbre sont constituées par le **contenu** des éléments texte :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```



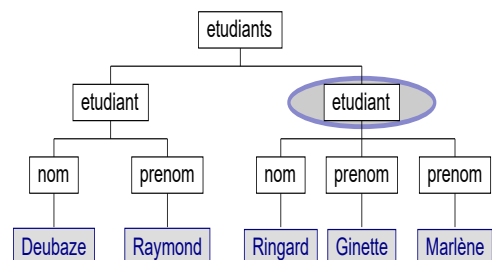
Connaissions-nous déjà d'autres éléments formant également des **feuilles de l'arbre** ?

2.1.5 La famille de l'élément courant

De la même manière qu'il est courant de se "déplacer" par l'esprit dans une arborescence de répertoires par exemple, certaines applications raisonnent en se déplaçant virtuellement dans l'arbre **XML** (*sélecteurs CSS, axes XPath, fonctions DOM, ...*)

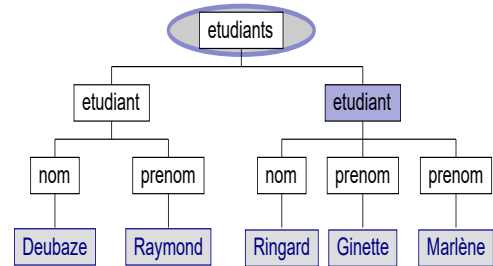
Lors d'un déplacement dans l'arbre, l'élément sur lequel on se trouve positionné à un instant donné du processus est appelé **l'élément courant** :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```



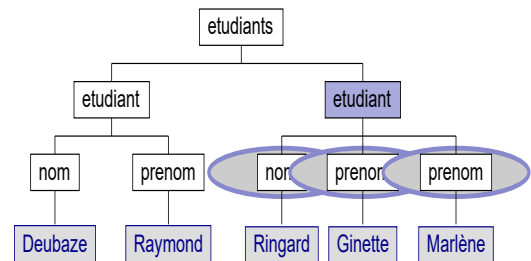
L'élément situé au-dessus de l'élément courant dans l'arbre est appelé son **élément parent** :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```

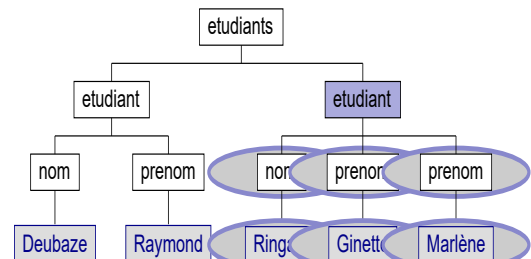
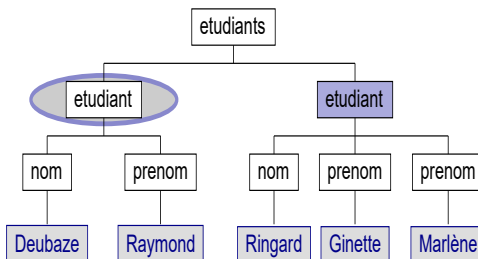


Toujours suivant la métaphore de l'arbre généalogique, les éléments situés à l'échelon juste inférieur à celui de l'élément courant sont appelés ses **enfants** :

```
<etudiants>
  <etudiant>
    <nom>Deubaze</nom>
    <prenom>Raymond</prenom>
  </etudiant>
  <etudiant>
    <nom>Ringard</nom>
    <prenom>Ginette</prenom>
    <prenom>Marlène</prenom>
  </etudiant>
</etudiants>
```



De la même manière, les éléments (*ou plus généralement les noeuds*) issus du même parent que l'élément courant seront appelés ses **frères**, l'ensemble de ceux situés plus bas dans la même branche ses **descendants** (*enfants, petits-enfants, ...*), etc...



2.1.6 Structure d'un document XML

Les spécifications **XML** imposent qu'un document bien formé doit forcément pouvoir se mettre sous la forme d'un arbre.

Il en résulte les contraintes suivantes :

- un tel document ne peut comporter qu'un **seul élément racine**,
- les balises doivent être **correctement imbriquées**.

Voici un exemple de construction mal formée :

```
<b>ceci est <i>un exemple</b> incorrect</i> en XML
```

> En résumé :

Tout document **XML** peut se mettre sous la forme d'un arbre.

De manière duale, tout arbre peut se représenter sous la forme d'un document **XML** !

Un informaticien blasé exprimera ceci en disant :

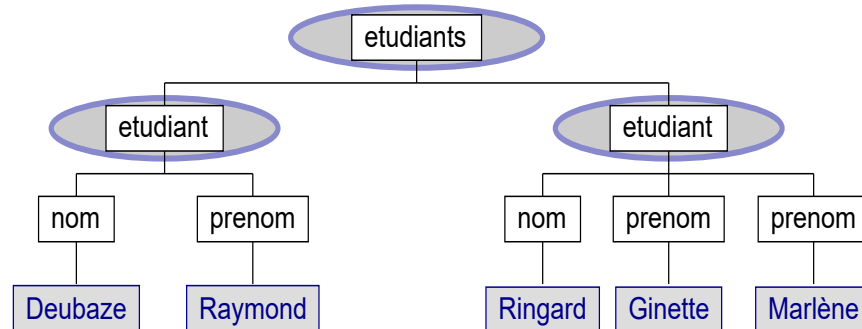
XML ? c'est juste un mécanisme pour sérialiser des arbres !

2.2 Eléments et attributs

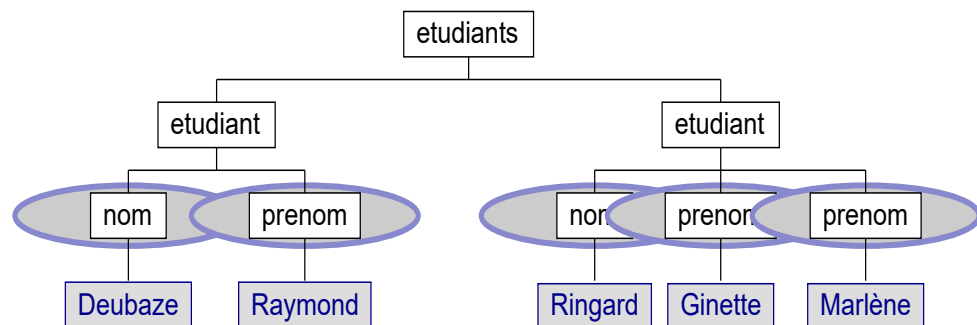
2.2.1 Contenu des éléments simples

L'arbre vu jusqu'à présent à titre d'exemple, fait apparaître deux types d'éléments simples :

- on dira que les éléments dont tous les enfants sont eux-mêmes des éléments ont un **contenu élémentaire** (*element content*) :



- tandis que les éléments qui ne contiennent que du texte seront dits à **contenu textuel** (*text content*) :



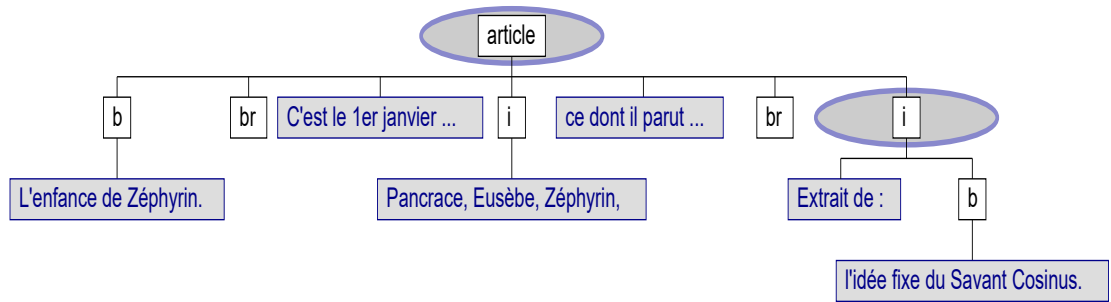
Cette structure, qui ne fait apparaître que des éléments à contenu élémentaire ou textuel est très fréquente dans le cas de **documents de données**.

2.2.2 Eléments à contenu mixte

Il existe un autre type d'éléments dits à **contenu mixte**, dont les enfants sont constitués à la fois par des éléments et du texte :

```

<article>
<b>L'enfance de Zéphyrin.</b><br/>
C'est le 1er janvier, à minuit une seconde sexagésimale de temps moyen,
que le jeune Brioché poussa ses premiers vagissements. A son baptême,
il reçut les prénoms harmonieux, poétiques et distingués de <i>Pancrace,
Eusèbe, Zéphyrin,</i> ce dont il parut se soucier comme un cloporte d'un
ophicléide.<br/>
<i>Extrait de : <b>l'idée fixe du Savant Cosinus.</b></i>
</article>
  
```



Les éléments à contenu mixte sont fréquents dans les **documents texte** à destination des humains (*articles, publications, pages web...*).

2.2.3 Attributs

Outre leur contenu (*élémentaire, textuel ou mixte*) les éléments **XML** peuvent également être munis d'**attributs**.

Un **attribut** est une propriété caractérisée par un **nom**, à laquelle le document associe une **valeur**.

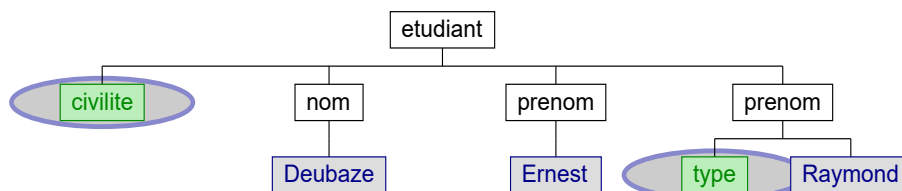
Dans l'exemple ci-dessous l'élément **etudiant** possède un attribut **civilite** dont la valeur est **M**, et le second élément **prenom** possède un attribut **type** dont la valeur est **usuel** :

```
<etudiant civilite="M">
  <nom>Deubaze</nom>
  <prenom>Ernest</prenom>
  <prenom type="usuel">Raymond</prenom>
</etudiant>
```

Ainsi qu'illustré par l'exemple ci-dessus, les règles de syntaxe pour affecter un ou plusieurs attributs à un élément au sein d'un **document XML** sont les suivantes :

- l'attribut apparaît sous la forme d'une affectation **nom="valeur"** associée à la **balise de début** de l'élément,
- le nom de l'attribut est séparé de sa valeur par le signe **=** et d'éventuels **espaces optionnels**,
- la valeur est obligatoirement délimitée par des **apostrophes** ou des **guillemets** (*simples ou double quotes*).

Au sein de l'**arbre XML** les attributs constituent une catégorie particulière de noeuds (*comme les noeuds texte*), et apparaissent comme des enfants de l'élément auquel ils s'appliquent :



La **valeur** de l'attribut, comme son nom, sont des propriétés du noeud lui-même qui constitue un élément terminal de l'arbre (*feuille*). (*N.B. contrairement à ce qu'on pourrait penser il n'y a pas de noeud texte, enfant du noeud attribut*)

2.2.4 Éléments ou attributs ?

Lors de la conception d'une application **XML** la question se pose souvent de savoir s'il faut modéliser telle ou telle information sous la forme d'un **élément** ou d'un **attribut**.

Une réponse générique n'est pas possible, et le problème est sujet à querelles de chapelles...

Ci-dessous se trouvent toutefois quelques éléments de réponse, basés sur des exemples.

> Exemple 1

Quelle solution ci-dessous faut-il par exemple privilégier lors de la conception d'une application de gestion des étudiants ?

```
<etudiant
  nom="Deubaze"
  prenom="Raymond"
/>
```

```
<etudiant>
  <nom>Deubaze</nom>
  <prenom>Raymond</prenom>
</etudiant>
```

Réponse

Un élément ne peut avoir qu'un seul attribut d'un nom donné. Il faut donc préférer les éléments aux attributs dans les cas où la propriété candidate (*ici le prénom*) serait susceptible d'avoir plusieurs valeurs.

La solution de droite est préférable, car elle laisse la porte ouverte à l'existence de plusieurs prénoms :

```
<etudiant>
  <nom>Deubaze</nom>
  <prenom>Raymond</prenom>
  <prenom>Ernest</prenom>
</etudiant>
```

> Exemple 2

Laquelle des deux solutions ci-dessous vaut-il mieux adopter ?

```
<article titre =
  "Eléments ou attributs ?"
/>
```

```
<article>
  <titre>
    Eléments ou attributs ?
  </titre>
</article>
```

Réponse

La valeur d'un attribut est forcément une chaîne de caractères, et ne pourra jamais être complétée par quoi que soit.

En s'appuyant sur un élément, celui-ci pourra posséder des attributs, et contenir lui-même d'autres éléments :

```
<article>
  <titre niveau="1"> Eléments ou attributs ? </titre>
  <titre niveau="2"> <i>(il faut choisir)</i> </titre>
</article>
```

Complément de réponse avancé

Cet aspect est également vrai pour les **instructions de traitement** (*vues plus loin*), couramment employées pour la génération de documents dynamiques (*cf. asp, php ou jsp*).

A noter que la construction suivante (*courante en PHP / HTML*) est **incorrecte** du point de vue de **XML** (*mais serait-il bien nécessaire qu'elle le soit ?*) :

```
<article titre="<?php print titre_dynamique(); ?>">
```

alors que si le titre a été conçu sous forme d'un élément, la construction suivante est possible :

```
<article>
  <titre><?php print titre_dynamique(); ?></titre>
</article>
```

2.3 Entités

2.3.1 Principe des entités prédéfinies

En **XML** le caractère **<** (*inférieur à*) est toujours considéré comme indiquant un début de balise.

Il est donc impossible de l'utiliser tel quel dans le texte d'un document : c'est un **caractère réservé**.

La solution consiste à insérer la construction **<** en lieu et place du caractère désiré. En verbiage **XML** on dira qu'on fait appel à une **entité prédéfinie**.

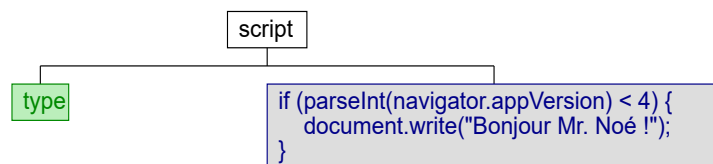
Exemple

```
<script type="text/javascript">
  if (parseInt(navigator.appVersion) &lt; 4) {
    document.write("Bonjour Mr. Noé !");
  }
</script>
```

Techniquement les spécifications précisent que le **parseur XML** (*i.e. le module logiciel qui lit le document dans un fichier*) doit remplacer un appel d'entité par le code caractère correspondant, aussitôt que possible avant de transférer l'information en mémoire.

Il n'a pas de sens de chercher trace de l'existence d'un appel d'entité dans l'**arbre XML**.

Dans le cas ci-dessus cet arbre comprendra un élément texte dont la valeur sera une chaîne de caractères qui contiendra quelque part le caractère **<** :



Ceci signifie en particulier que l'application qui aura à traiter l'information ne verra pas l'appel d'entité mais bel et bien le caractère concerné.

Réciproquement, si un élément texte contient le caractère **<**, alors l'opération de **sérialisation** (*qui consiste à retranscrire l'arbre XML dans un document*) fera apparaître un appel d'entité.

2.3.2 Les entités prédéfinies

Il existe en **XML** deux **caractères réservés** qui donnent **obligatoirement** lieu à appel d'entité :

Caractère réservé	Appel d'entité	Description
<	<	inférieur à
&	&	esperluète

ainsi que trois **caractères spéciaux** qui peuvent être employés tels quels ou donner lieu à appel d'entité en fonction du contexte :

Caractère spécial	Appel d'entité	Description
>	>	supérieur à
'	'	guillemet simple
"	"	guillemet double

Exemples

```
<auteurs>D'Alembert &amp; al.</auteurs>
```

```
<img src='photo.png' alt='D&apos;Alembert' />
```

Les cinq entités ci-dessus sont les seules à être spécifiées comme prédéfinies. Elles sont universellement reconnues par tous les outils et applications **XML**.

N.B. Il est possible de définir ses propres appels d'entités au sein d'une **DTD** (*Définition de Type de Document*).

2.3.3 Entités caractère

XML travaille en interne avec le codage de caractères Unicode (*ISO 10646*) (*i.e. les noeuds texte de l'arbre XML sont en Unicode*).

Il est possible d'inclure n'importe quel caractère Unicode (*même non accessible au clavier*) au sein d'un **document XML**, à l'aide d'un appel d'**entité caractère**.

La syntaxe d'une **entité caractère** ressemble à celle d'une **entité prédéfinie**, à cela près que le nom de l'entité est remplacé par la caractères **#** (*dièse*), suivi par le code du caractère considéré sans oublier le **;** (*point-virgule*) final :

Insérons un espace insécable entre guillemets : ** **

Quelques exemples :

Entité caractère	Appel d'entité	Description
	&#160;	espace insécable
©	&#169;	copyright
®	&#174;	marque déposée
Σ	&#931;	Sigma
β	&#946;	bêta
π	&#960;	pi
#	&#8704;	quel que soit
♠	&#9824;	pique

Sachant qu'Unicode répertorie à peu près 50000 caractères différents couvrant à peu près toutes les langues du monde (*y compris le phénicien et le klingon*), cette liste ne saurait évidemment être exhaustive...

Une variante permet d'utiliser le code caractère en hexadécimal (*au lieu de décimal*) en faisant précéder la valeur numérique par le caractère **x** :

Deux caractères α absoluments équivalents : **α** et **α**

N.B. Le traitement des **entités caractère** s'effectue exactement de la même manière que celle des **entités prédéfinies**.

Il n'a toujours aucun sens de chercher trace des appels d'entité au sein de l'**arbre XML**. Le noeud texte concerné contiendra simplement le caractère visé.

2.4 Autres sections

2.4.1 Sections CDATA

Il est possible d'indiquer au sein d'un **document XML** qu'une portion de texte ne doit pas être analysée (*parsed*) à la recherche de balises ou d'appels d'entités (*parsed character data ≠ character data*).

Cette opération s'effectue en isolant la portion de texte en question à l'aide d'une section **CDATA** :

```
<script language="JavaScript">
  <![CDATA[
    if (parseInt(navigator.appVersion) < 4) {
      document.write("Bonjour Mr. Noé !")
    }
  ]]>
</script>
```

Le texte d'une section **CDATA** se retrouvera tel quel au sein d'un noeud texte de l'**arbre XML**.

Cette construction est souvent utilisée pour intégrer des programmes (*cf. Javascript*) dans un document, ou du texte explicatif donnant des exemples de code source **XML** :

```
<p>
  Voici un exemple de code XSL :
  <exemple>
    <![CDATA[
      <xsl:template match="@email">
        <td>
          <a href="mailto:{.}">
            <xsl:apply-templates/>
          </a>
        </td>
      </xsl:template>
    ]]>
  </exemple>
  illustrant ceci.
</p>
```

Une section **CDATA** doit être correctement imbriquée avec les autres éléments.

2.4.2 Commentaires

Les **commentaires** inclus dans un document **XML** respectent la même syntaxe qu'en **HTML** :

```
<!-- ceci est un commentaire -->
```

Un commentaire ne peut pas contenir la chaîne **--**.

N.B. Un parseur **XML** n'est pas obligé de transmettre les commentaires à l'application. Il n'y a donc absolument aucune garantie qu'un commentaire se retrouvera au sein de l'**arbre XML**.

Il est par conséquent incorrect de baser le fonctionnement d'une application sur la présence (*ou l'absence*) d'un commentaire particulier.

Un commentaire doit être correctement imbriqué avec les autres éléments.

2.4.3 Instructions de traitement

Les **instructions de traitement** permettent de faire ce qu'il ne faut pas faire avec des commentaires : transmettre des informations à une application.

```
<?xml-stylesheet href="option.xsl" type="text/xsl" ?>
<?tic-appl-server OpticForm form init() ?>
<document>
  Votre navigateur est :
  <?php return $_SERVER["HTTP_USER_AGENT"]; ?>
</document>
```

Le nom qui suit les caractères `<?` ouvrant la balise indique l'application à laquelle les informations sont destinées.

Du point de vue de **XML** il n'y a aucune contrainte particulière sur le texte formant le corps de la balise (*sauf caractères réservés et chaîne fermante `>`*).

Une **instruction de traitement** doit être correctement imbriquée avec les autres éléments. La construction suivante (*courante en PHP / HTML*) est donc **incorrecte** :

```
<article titre="<?php print titre_dynamique(); ?>">
```

2.4.4 Déclaration XML

Un document **XML** peut commencer par une **déclaration XML** (*présence non obligatoire*).

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

La **déclaration XML** peut mentionner le jeu de caractères du document :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

D'après les spécifications, les seuls jeux de caractères obligatoirement reconnus par un parseur **XML** sont **UTF-8** et **UTF-16**.

Toutefois, les parseurs reconnaissent en général d'autres jeux de caractères (*point fort des parseurs commerciaux*), et en particulier **ISO-8859-1** qui nous a longtemps concernés en premier chef (*cf. caractères accentués français*), mais est actuellement remplacé par **UTF-8** par les systèmes récents.

2.5 Vers des documents bien formés

2.5.1 Noms autorisés

Les noms d'éléments et d'attributs peuvent contenir n'importe quel caractère alphanumérique (A-Z a-z 0-9) ainsi que quelques caractères de ponctuation : `_` (*souligné*), `-` (*tiret*), `.` (*point*) et `:` (*double-point*) qui sera réservé pour les espaces de noms (*namespaces*).

Un **nom XML** ne peut pas commencer par un chiffre, un tiret ou un point.

Tous les autres caractères (*espaces, apostrophes, \$, virgule, point-virgule...*) sont interdits.

Les caractères locaux (*cf. caractères accentués en français*) sont acceptés dans la mesure où la **déclaration XML** indique le type de codage utilisé pour le document et que celui-ci est reconnu par le parseur.

2.5.2 Noms autorisés ?

Saurez-vous localiser les erreurs ?

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<exemples>
  <Numéro-SS>30363112089</Numéro-SS>
  <adresse_postale>Av. Guy de Collongue</adresse_postale>
  <nom_d'emprunt>Johnny</nom_d'emprunt>
  <mois-année>12/99</mois-année>
  <jours/mois>01/02</jours/mois>
  <nb:>ne pas oublier</nb:>
  <2-vous-à-moi>petit à parté</2-vous-à-moi>
</exemples>
```

2.5.3 Documents bien formés

Un document qui respecte les règles de la syntaxe **XML** est appelé un **document XML bien formé**. Voici le récapitulatif de quelques règles simples participant à l'élaboration de documents **bien formés** :

- a toute balise de début doit correspondre une balise de fin,
- les balises doivent être correctement imbriquées,
- il ne peut y avoir qu'un seul élément racine,
- la valeur des attributs doit se trouver entre guillemets,
- un élément ne peut avoir deux attributs du même nom,
- les commentaires et instructions de traitement doivent être correctement imbriqués avec les balises,
- les caractères `<` et `&` ne doivent pas apparaître tels quels.

Un **parseur XML** conforme aux spécifications **ne doit pas** transmettre un document mal formé à l'application.

> Comment vérifier si un document est bien formé ?

Pour cela il suffit de l'ouvrir avec un navigateur de dernière génération (*Firefox, IE 6.0, Mozilla 1.5, NS 7.0 ...*) :

Exemple de document bien formé :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/xml/exemple_1.xml\]](http://dmolinarius.github.io/demofiles/mod-84/xml/xml/exemple_1.xml)

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/xml/exemple_erreur.xml\]](http://dmolinarius.github.io/demofiles/mod-84/xml/xml/exemple_erreur.xml)

3. Namespaces - Espaces de noms

3.1 Fiche d'identité

Les spécifications concernant les espaces de noms (*XML Namespaces*) sont issues du **XML Core Working Group** du **W3C**.

☞ Vers la page d'accueil du XML Core Working Group
[\[http://www.w3.org/XML/Core/\]](http://www.w3.org/XML/Core/)

A l'heure actuelle (oct. 2006) il existe deux spécifications : l'une pour **XML 1.0** et l'autre pour **XML 1.1** (révisions d'août 2006),

☞ Spécifications : Namespaces in XML
[\[http://www.w3.org/TR/REC-xml-names/\]](http://www.w3.org/TR/REC-xml-names/)

☞ Spécifications : Namespaces in XML 1.1
[\[http://www.w3.org/TR/xml-names11/\]](http://www.w3.org/TR/xml-names11/)

3.2 Pourquoi les espaces de noms ?

Les **espaces de noms** (*XML Namespaces*) permettent d'éviter les conflits potentiels entre les noms de balises et d'attributs lorsqu'on désire construire des documents mêlant des vocabulaires provenant d'applications différentes :

```
<article>
  <titre>
    Introduction à XML
  </titre>
  . . .
</article>
```

```
<personne>
  <titre>
    Altesse Sérénissime
  </titre>
  . . .
</personne>
```

Ils servent également à identifier les éléments et les attributs relatifs à une même **application XML** pour faciliter leur traitement par le processeur approprié :

```
<xsl:template match="back">
  <td align="left" width="25">
    <a>
      <xsl:attribute name="href">
        <xsl:call-template name="url-previous"/>
      </xsl:attribute>
      
    </a>
  </td>
</xsl:template>
```

3.3 Identifiants d'espaces de noms

Un espace de noms est identifié grâce à un **URI** (*Uniform Resource Identifier*).

N.B. Un **URI** n'est autre qu'une banale **chaîne de caractères**, conçue par l'organisme en charge de la ressource identifiée, qui garantit que cette chaîne ne désignera jamais autre chose que la ressource en question.

Exemple d'identifiants d'espaces de noms

application XML	URI d'espace de noms
XHTML	http://www.w3.org/1999/xhtml
XSLT	http://www.w3.org/1999/XSL/Transform
XSL-FO	http://www.w3.org/1999/XSL/Format
SVG	http://www.w3.org/1999/svg
SMIL	http://www.w3.org/TR/REC-smil
WebDAV	DAV:
RDF	http://www.w3.org/1999/02/22-rdf-syntax-ns#

N.B. Comme on le voit ci-dessus, un **URI** peut prendre la forme d'une **URL** (*Uniform Resource Locator*).

Toutefois, dans le cas particulier des **espaces de noms** cela n'implique nullement que cette **URL** corresponde à une quelconque ressource disponible en ligne (*spécifications, simple message, ou 404 Not Found*).

3.4 Préfixes d'espaces de noms

Pour indiquer au sein d'un document à quel espace de noms appartient un élément ou un attribut, on use d'un **préfixe** :

```
<xsl:template name="br">
  <fo:leader leader-pattern="space"/>
  <fo:inline linefeed-treatment="preserve" white-space-collapse="false">
    <xsl:text>&#x0A;</xsl:text>
  </fo:inline>
</xsl:template>
```

> Association préfixe - URI

L'association entre un préfixe et l'identifiant d'espace de noms se fait à l'aide de l'attribut spécial **xmlns** :

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

N.B. La portée de cette association est limitée à l'élément portant l'attribut **xmlns** et à ses descendants. C'est la raison pour laquelle ces déclarations sont très souvent effectuées au niveau de l'élément racine d'un document.

> Choix du préfixe

Bien que la plupart des applications possèdent un **préfixe usuel**, le choix du préfixe est totalement libre (*sauf le préfixe xml qui est réservé*), à la charge du concepteur d'un document, et limité à ce document :

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

N.B. Le processeur d'une application particulière s'appuie sur l'**URI** et non le préfixe, pour reconnaître les éléments et les attributs qui le concernent.

3.5 Espace de noms par défaut

Les éléments et attributs sans préfixe qui apparaissent dans un document sont attachés à l'espace de noms par défaut.

L'**URI** de l'espace de noms par défaut peut être spécifiée à l'aide d'un attribut **xmlns** ne mentionnant pas de préfixe :

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Un exemple de document XHTML</title>
    . . .
  </head>
  <body>
  </body>
  . . .
</html>
```

4. Définition de Type de Document

4.1 Déclaration de Type de Document

4.1.1 Déclaration de Type de Document

Un document **XML** fait référence à une **DTD** grâce à une **déclaration de type de document** placée dans le prologue du document (*i.e. après la déclaration XML, mais avant l'élément racine*).

La syntaxe générale d'une déclaration de type de document est :

```
<!DOCTYPE element_racine SYSTEM "URL_dtd">
```

> Exemple

La déclaration ci-dessous indique que le nom de l'élément racine est "*classe*" et que la **DTD** peut être trouvée à l'URL "*classe-primaire.dtd*".

```
<?xml version="1.0" ?>
<!DOCTYPE classe SYSTEM "classe-primaire.dtd">
<classe>
.
.
.
</classe>
```

4.1.2 DTD publique

Lorsqu'un document est muni d'une **DTD**, les applications qui accèdent au document sont susceptibles de devoir accéder également à la **DTD**, aux fins de validation, pour récupérer la valeur par défaut de certains attributs, ou obtenir la valeur de certaines entités.

Dans le cas d'une application standard (*i.e. très répandue*), on peut déclarer la **DTD** grâce à un **identifiant public** :

```
<!DOCTYPE element_racine PUBLIC "URI_public" "URL_dtd">
```

L'identifiant public est un **URI**, il s'agit donc d'une chaîne globalement unique qui identifie l'application.

Ceci permet éventuellement à un processeur qui reconnaît l'identifiant public de ne pas ouvrir de connexion réseau pour accéder à la **DTD**, car il la connaît par construction (*cf. navigateurs dans le cas de XHTML*).

> Exemple

La déclaration ci-dessous indique qu'il s'agit d'un document **XHTML** :

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

4.1.3 DTD interne

Une déclaration de type de document ne fait pas forcément référence à un document externe.

La **DTD** peut être développée directement au sein de sa déclaration :

```
<!DOCTYPE classe [
  <!ELEMENT classe (promo,etudiant+)>
  <!ELEMENT etudiant (nom, prenom)>
  <!ELEMENT nom (#PCDATA)>
  <!ELEMENT prenom (#PCDATA)>
  <!ELEMENT promo (#PCDATA)>
]>
```

Cette technique est particulièrement utile en phase de développement, lorsqu'on met au point à la fois le premier document d'une série et la **DTD** qui les décrit, de manière à n'avoir qu'un seul fichier source à maintenir.

Les deux formats (*DTD interne et externe*) peuvent être associés dans la déclaration :

```
<!DOCTYPE classe SYSTEM "promo-centrale.dtd" [
  <!ELEMENT classe (promo,etudiant+)>
]>
```

Dans cet exemple, la plupart des éléments sont déclarés au sein de la **DTD externe** "*promo-centrale.dtd*", alors que l'élément racine "*classe*" est déclaré localement.

N.B. Les sous-ensembles **externe** et **interne** de la **DTD** doivent être compatibles. On ne peut pas, de cette manière, surcharger des déclarations d'éléments ou d'attributs. Seules les déclarations d'entités peuvent être surchargées.

4.1.4 Déclaration standalone

Lorsqu'une déclaration de type de document fait référence à une **DTD** (ou un sous-ensemble) externe, il est conseillé de l'indiquer au sein de la **déclaration XML** à l'aide du pseudo-attribut "*standalone*" :

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE classe SYSTEM "exemple_2.dtd">
```

Cette mention sert à indiquer à un **parseur non validant** qu'il devrait tout de même ouvrir et analyser la **DTD** qui est susceptible de contenir par exemple la valeur par défaut de certains attributs.

N.B. De nombreux parseurs n'analysent pas les sous-ensembles de **DTD** externes, et il est toujours possible de configurer les parseurs (*même validants*) pour ne pas le faire. Dans ce cas, le résultat obtenu peut dépendre du parseur ou de sa configuration.

4.2 Principe d'une DTD

Une **DTD** permet de définir le vocabulaire (*noms des éléments, attributs et entités*) et la grammaire (*contexte dans lequel peuvent ou doivent apparaître ces noms*) d'un **document XML**.

Une **DTD** est essentiellement composée de **déclarations de balisage** qui se subdivisent en :

- déclarations de type d'éléments,
- déclarations de liste d'attributs,
- déclarations d'entités.

L'extrait ci-dessous donne un aperçu de déclaration de type d'élément, de liste d'attributs, et d'entité :

```
<!ELEMENT classe (promo,etudiant+)>
. . .
<!ATTLIST etudiant civilité ( M | F ) #REQUIRED>
. . .
<!ENTITY contact "Daniel.Muller@ec-lyon.fr">
```

4.3 Déclaration de type d'élément

4.3.1 Principe

La **déclaration de type** d'un élément précise le **modèle de contenu** de cet élément. Aucun type d'élément ne peut être déclaré plus d'une fois.

La syntaxe générale d'une déclaration de type d'élément est :

```
<!ELEMENT nom_element modele_de_contenu >
```

Le modèle de contenu d'un élément simple peut prendre l'une des formes suivantes :

EMPTY - l'élément est vide (*comme par exemple les balises HR, BR, IMG ... en langage XHTML*),

```
<!ELEMENT img EMPTY>
```

ANY - l'élément peut contenir n'importe quel autre élément déclaré (*très rarement employé*),

```
<!ELEMENT test ANY>
```

(#PCDATA) - l'élément est un élément texte (*Parsed Character Data*), il ne peut contenir de sous-élément,

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

(liste de sous-éléments) - la liste donne les noms (*séparés par des virgules et des blancs facultatifs*) et l'ordre des sous-éléments autorisés :

```
<!ELEMENT étudiant (nom,prénom)>
```

Dans le cas de l'exemple ci-dessus, un élément "*étudiant*" devra obligatoirement comporter un (*et un seul*) sous-élément "*nom*", suivi par un (*et un seul*) sous-élément "*prénom*", dans cet ordre.

4.3.2 Modèles de contenu élémentaire

Dans le cas des éléments simples à contenu élémentaire (*i.e. que des sous éléments, pas de texte*) on peut spécifier le type des sous-éléments, leur ordre et dans une certaine mesure le nombre d'occurrences autorisées.

Pour cela, le **modèle de contenu** peut comporter :

- **des listes** - avec l'opérateur ";",
- **des alternatives** - avec l'opérateur "|",
- **des groupes** - entre parenthèses "()".

Les éléments ou groupes d'éléments peuvent être post-fixés par un opérateur indiquant le nombre d'occurrences autorisées :

- **+** - une ou plusieurs occurrences,
- ***** - zéro ou plusieurs occurrences,
- **?** - zéro ou une occurrence.

> Exemples

La déclaration ci-dessous indique qu'une classe est composée de un ou plusieurs étudiants :

```
<!ELEMENT classe (étudiant+)>
```

Un étudiant doit avoir un nom, suivi par un ou plusieurs prénoms :

```
<!ELEMENT étudiant (nom,prénom+) >
```

Il est possible de mentionner son numéro de sécurité sociale :

```
<!ELEMENT étudiant (nom,prénom+,insee?) >
```

A défaut, on peut donner son identifiant interne ECL :

```
<!ELEMENT étudiant (nom,prénom+, (insee|eclid)?) >
```

Il est éventuellement inscrit à des cours :

```
<!ELEMENT étudiant (nom,prénom+, (insee|eclid)?,cours*) >
```

N.B. L'ordre des listes est obligatoire. Pour autoriser des ordres différents, la seule solution consiste à lister toutes les alternatives autorisées :

```
<!ELEMENT étudiant ((nom,prénom+) | (prénom+,nom)) >
```

Comment pourrait-on autoriser également le nom à apparaître entre deux prénoms ?

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/elt-elt-content-exemple.html>

4.3.3 Modèles de contenu mixte

Pour les éléments à contenu mixte (*texte mêlé à des sous-éléments*), il est possible de préciser le type des sous-éléments autorisés, mais pas leur ordre ni le nombre de leurs occurrences.

La syntaxe du modèle de contenu est la même que dans le cas des contenus élémentaires purs, sauf qu'il faut obligatoirement mentionner le texte en première position à l'aide du mot-clé **"#PCDATA"** (*Parsed Character Data*).

De plus, le modèle de contenu comporte obligatoirement l'opérateur **"*"** :

```
<!ELEMENT paragraphe (#PCDATA | gras | italiques)* >
<!ELEMENT gras (#PCDATA | italiques)* >
<!ELEMENT italiques (#PCDATA | gras)* >
```

4.4 Déclaration de liste d'attributs

4.4.1 Principe

La **déclaration de liste d'attributs** d'un élément sert à :

- définir un **jeu d'attributs** pour les éléments de ce type,
- établir un **modèle de contenu** de ces attributs,
- fournir une **valeur par défaut** pour certains attributs.

S'il existe plus d'une définition pour un même attribut d'un type d'élément donné, seule est prise en compte la première, les déclarations subséquentes sont ignorées.

La syntaxe générale d'une déclaration de liste d'attribut est :

```
<!ATTLIST nom_element (nom_attribut modele_de_contenu valeur_par_defaut)+ >
```

La déclaration ci-dessous indique par exemple que les éléments du type "*étudiant*" possèdent un attribut obligatoire nommé "*nom*", du type "*CDATA*" (*Character Data*) c'est à dire simplement composé de texte (*non analysé*) :

```
<!ATTLIST étudiant nom CDATA #REQUIRED >
```

4.4.2 Liste d'attributs

S'il existe plus d'une déclaration de liste d'attributs pour un type d'élément donné, le contenu de toutes les déclarations fournies est fusionné.

Lorsqu'un même élément possède plusieurs attributs, ceux-ci peuvent donc faire l'objet d'une seule ou de plusieurs déclarations :

```
<!ATTLIST étudiant
  nom CDATA #REQUIRED
  prénom CDATA #REQUIRED
  civilité ( M | F ) #REQUIRED
  statut ( célibataire | marié | mariée ) "célibataire" >
```

est équivalent à :

```
<!ATTLIST étudiant nom CDATA #REQUIRED >
<!ATTLIST étudiant prénom CDATA #REQUIRED >
<!ATTLIST étudiant civilité ( M | F ) #REQUIRED >
<!ATTLIST étudiant statut ( célibataire | marié | mariée ) "célibataire" >
```

N.B. Lorsqu'un même attribut est autorisé avec les mêmes caractéristiques pour plusieurs éléments (*cf. par exemple en XHTML, les attributs communs à TD et TH*), il doit être **répété** pour chacun des éléments.

```
<!ATTLIST étudiant nom CDATA #REQUIRED >
<!ATTLIST étudiant prénom CDATA #REQUIRED >
<!ATTLIST professeur nom CDATA #REQUIRED >
<!ATTLIST professeur prénom CDATA #REQUIRED >
```

Cette lourdeur peut être contournée par l'utilisation d'**entités paramètres** (*vues plus loin*).

4.4.3 Types d'attributs

Contrairement aux éléments, dont le contenu textuel ne peut être contraint à l'aide d'une **DTD**, il est possible dans une certaine mesure de restreindre le domaine des valeurs possibles pour un attribut.

Ci-dessous les modèles de contenu les plus courants :

CDATA - chaîne de caractères (*cf. l'attribut src de l'élément img en XHTML*),

```
<!ATTLIST img src CDATA #REQUIRED>
```

```

```

NMTOKEN (*Name Token*) - chaîne de caractères sans espaces ressemblant à un nom **XML**, à ceci près que tous les caractères autorisés sont permis pour l'initiale (*cf. par exemple l'attribut name des hyperliens en XHTML*),

```
<!ATTLIST a name NMTOKEN #IMPLIED>
```

```
<a name="2B3"> . . . </a>
```


NMTOKENS - liste de **NMTOKEN** séparés par des espaces. Soit par exemple un élément *"Allow"* qui admet un attribut *"host"* composé d'une liste de numéros IP séparés par des espaces :

```
<!ATTLIST Allow host NMTOKENS #REQUIRED>
```

```
<Allow host="156.18.10.1 156.18.10.2"/>
```

énumération - la valeur est à prendre parmi une liste de **NMTOKEN** fournie au sein de la déclaration (cf. l'attribut *align* de l'élément *img* en *XHTML*),

```
<!ATTLIST img align (top|middle|bottom) #IMPLIED>
```

```
<img align="middle" ... >
```

4.4.4 Identifiants uniques

Il existe une catégorie particulière d'attributs, dont la valeur doit correspondre à un identifiant unique au sein d'un même document. Ces attributs sont déclarés avec un modèle de contenu particulier :

ID - nom XML, identifiant unique (cf. les attributs *id* en *XHTML*) :

```
<!ATTLIST span id ID #IMPLIED>
```

```
<span id="drop_zone"> . . . </span>
```

N.B. Si, au sein d'un document, deux éléments ont la même valeur pour deux attributs (*même de nom différent*) déclarés du type **ID**, alors le document est non valide.

Il est également possible de contraindre certains attributs à correspondre à un identifiant, ou à une liste d'identifiants présents dans le document :

IDREF - nom XML donnant la valeur d'un attribut du type **ID** d'un élément du même document.

```
<!ATTLIST toc-entry link-to IDREF #IMPLIED>
```

```
<toc-entry link-to="chap1">Chapitre 1</toc-entry>
. . .
<chapter id="chap1">
. . .
</chapter>
```

IDREFS - une liste d'**IDREFs** séparés par des espaces :

```
<!ATTLIST login accesslist IDREFS #REQUIRED>
```

```
<role id="admin"> . . . </role>
<role id="webmaster"> . . . </role>
<role id="baseroor"> . . . </role>
. . .
<login accesslist="admin webmaster baseroor">administrateur</login>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/atr-id-exemple.html>

N.B. Il faut remarquer que ce mécanisme reste très limité, puisqu'il ne permet pas d'indiquer à quel type d'élément on veut faire référence ni si la référence doit être unique.

4.4.5 Valeur par défaut

Une déclaration de liste d'attributs, outre les noms et les modèles de contenu, indique également si l'attribut est obligatoire ou non, et quel comportement adopter lorsqu'il est absent.

Ci-dessous les quatre cas possibles :

#IMPLIED - l'attribut est optionnel, il n'y a pas de valeur par défaut,

```
<!ATTLIST img width CDATA #IMPLIED
             height CDATA #IMPLIED>
```

```
<img width="100%" ... >
```

#REQUIRED - la présence de l'attribut est obligatoire, il n'y a pas de valeur par défaut,

```
<!ATTLIST img src CDATA #REQUIRED>
```

```

```

"une valeur par défaut" - l'attribut est optionnel, mais lorsqu'il est absent le parseur doit se comporter comme s'il était présent avec la valeur par défaut fournie,

```
<!ATTLIST input type (text|password|checkbox|radio|hidden) "text">
```

```
<input ... />
<input type="text" ... />
<input type="radio" ... />
```

#FIXED "valeur" - l'attribut est optionnel, s'il est absent le parseur doit se comporter comme s'il était présent avec la valeur par défaut fournie, s'il est présent il ne peut prendre que la valeur fournie.

```
<!ATTLIST html xmlns CDATA #FIXED "http://www.w3.org/1999/xhtml">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
...
</html>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/atr-value-exemple.html>]

4.4.6 Quelques exemples

```
<!ATTLIST li type (disc|square|circle) #IMPLIED >
```

L'attribut **"type"** de l'élément **"li"** est optionnel. S'il est présent, il peut prendre une des valeurs **"disc"**, **"square"** ou **"circle"**. S'il est absent il n'y a pas de valeur par défaut (*i.e. il n'y aura pas d'attribut type dans l'arbre XML en mémoire*).

```
<!ATTLIST li type (disc|square|circle) "disc" >
```

L'attribut **"type"** de l'élément **"li"** est optionnel. S'il est présent, il peut prendre une des valeurs **"disc"**, **"square"** ou **"circle"**. S'il est absent **le parseur** doit se comporter **comme s'il était présent** et le transmettre à l'application avec la valeur **"disc"**.

```
<!ATTLIST html xmlns CDATA #FIXED 'http://www.w3.org/1999/xhtml'>
```

L'attribut *"xmlns"* de l'élément *"html"* est optionnel. S'il est présent, il ne peut prendre que la valeur précisée. Présent ou absent, **le parseur** doit se comporter strictement comme s'il était présent avec la valeur ci-dessus et le transmettre à l'application.

```
<!ATTLIST meta content CDATA #REQUIRED>
```

L'attribut *"content"* de l'élément *"meta"* est obligatoire. Il n'y a pas de valeur par défaut.

4.5 Déclarations d'entités

4.5.1 Entité générale

Une **entité** est une sorte de macro, pour représenter un caractère ou une chaîne de caractères par un nom.

Une **entité générale** est une macro qui s'emploie au sein du **document XML**.

La syntaxe d'une déclaration d'entité générale est :

```
<!ENTITY nom_entite "valeur_entite" >
```

Au sein du document l'utilisation se fait comme pour les **entités prédéfinies**, en faisant précéder le nom de l'entité par une esperluète *"&"* et suivre par un point-virgule *","* :

```
&nom_entite;
```

> Exemple 1

Ci-dessous un exemple de **déclaration d'entité** pour les caractères *"blanc insécable"* et *"e accent aigu"*.

```
<!ENTITY nbsp "&#160;"> <!-- no-break space -->
<!ENTITY eacute "&#233;"> <!-- small e with acute -->
```

```
<p>un&nbspbsp;&lt;&eacute;l&eacute;ment&gt;</p>
```

Tester un exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple1.html)

> Exemple 2

```
<!DOCTYPE page SYSTEM "doctype.dtd" [
  <!ENTITY contact "Daniel.Muller@ec-lyon.fr">
]>
```

```
<a href="mailto:&contact;">votre contact</a>
```

Remarquer ci-dessus comment l'entité est déclarée dans la partie interne de la déclaration de type de document, et appelée comme valeur d'un attribut.

Tester un exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple2.html)

> Exemple 3

Dans la mesure où le résultat n'est pas récursif, la déclaration d'une entité générale peut invoquer la valeur d'une autre entité générale :

```
<!ENTITY contact "Daniel.Muller">
<!ENTITY address ' <a href="mailto:&contact;">&contact;</a>' >
```

Tester un exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple3.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-general-exemple3.html)

4.5.2 Entité générale externe analysée

La valeur d'une entité générale peut être stockée dans un document à part. On parlera alors d'**entité générale externe analysée** (*parsed external general entity*).

La déclaration se fait sur l'un des modèles suivants (*entité système ou publique*) :

```
<!ENTITY nom_entite SYSTEM "URL_entite">
<!ENTITY nom_entite PUBLIC "URI_public" "URL_entite">
```

L'insertion de l'entité au sein du document se fait de manière classique :

```
&nom_entite;
```

N.B. Le contenu d'une entité externe analysée est très similaire à celui d'un document XML. Un tel document peut comporter (*ou non*) une **déclaration XML**.

La seule différence est qu'une entité externe analysée peut comporter **plusieurs éléments racine**.

> Exemple

Lorsque l'information contenue dans un même document **XML** devient très volumineuse, il est beaucoup plus pratique de la manipuler sous forme de fichiers multiples, correspondant par exemple aux divers chapitres (*voire aux sections*) d'un livre.

Les divers chapitres seront alors déclarés au sein du document principal en tant qu'entités externes :

```
<!DOCTYPE bouquin SYSTEM "livre.dtd" [
  <!ENTITY intro SYSTEM "introduction.ent">
  <!ENTITY chap1 SYSTEM "chapitre_1.ent">
  <!ENTITY chap2 SYSTEM "chapitre_2.ent">
  <!ENTITY concl SYSTEM "conclusion.ent">
]>
```

Et le contenu du document sera composé à l'aide d'appels d'entités :

```
<bouquin>
  &intro;
  &chap1;
  &chap2;
  &concl;
</bouquin>
```

Tester un exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-parsed-ext-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-parsed-ext-exemple1.html)

4.5.3 Entité paramètre

Contrairement à une entité générale qui s'emploie au sein du contenu XML lui-même, une **entité paramètre** s'emploie au sein de la **DTD**.

La définition d'une entité paramètre se fait conformément au modèle :

```
<!ENTITY % nom_entite "valeur_entite" >
```

La syntaxe permettant d'invoquer une entité paramètre (*au sein de la DTD*) est similaire à celle permettant d'appeler une entité générale (*au sein du code XML*). Il suffit de remplacer l'esperluète par un caractère "%":

```
%nom_entite;
```

> Exemple - Liste d'attributs

Les entités paramètre sont très souvent utilisées au sein des **DTD** pour définir des attributs ou des listes d'attributs communes à plusieurs éléments :

```
<!ENTITY % usemap "usemap CDATA #IMPLIED">
<!ATTLIST img %usemap; >
<!ATTLIST input %usemap; >
<!ATTLIST object %usemap; >
```

Elles servent également à des fins mnémotechniques, afin de rendre les **DTD** complexes plus lisibles (*noter comment la définition d'une entité peut faire elle-même appel à une entité*).

```
<!ENTITY % StyleSheet "CDATA">
  <!-- style sheet data -->
<!ENTITY % Text "CDATA">
  <!-- used for titles etc. -->
<!ENTITY % coreattrs
  "id          ID          #IMPLIED
   class       CDATA       #IMPLIED
   style       %StyleSheet; #IMPLIED
   title       %Text;      #IMPLIED"
  >
```

```
<!ATTLIST b %coreattrs >
<!ATTLIST i %coreattrs >
<!ATTLIST em %coreattrs >
. . .
```

> Exemple - Déclaration d'élément

```
<!ENTITY % head.misc
  "(script|style|meta|link|object|isindex) *">
<!ELEMENT head (%head.misc;,
  ((title, %head.misc;, (base, %head.misc;)?)) |
  (base, %head.misc;, (title, %head.misc;))))>
```

Le déclaration ci-dessus signifie que l'élément **"head"** contient un élément **"titre"** unique et un élément **"base"** optionnel dans un ordre quelconque avec une combinaison éventuellement vide des éléments de **"head.misc"**.

> Exemple - DTD complexe

Les **DTD** d'applications **"réelles"** qui atteignent une certaine complexité font un usage intensif des entités paramètres.

 Consulter la DTD de XHTML Strict

[\[https://www.w3.org/TR/xhtml1/DTDs.html#a_dtd_XHTML-1.0-Strict\]](https://www.w3.org/TR/xhtml1/DTDs.html#a_dtd_XHTML-1.0-Strict)

4.5.4 Entité paramètre externe

Les entités paramètres, comme les entités générales, sont susceptibles de faire l'objet d'un document séparé, public ou privé, connu par un **URI** et/ou accessible par une **URL** :

```
<!ENTITY % nom_entite SYSTEM "URL_entite">
<!ENTITY % nom_entite PUBLIC "URI_public" "URL_entite">
```

L'insertion de l'entité au sein de la **DTD** se fait toujours de la même manière :

```
%nom_entite;
```

Le code ci-dessous indique comment réutiliser les entités générales correspondant aux caractères **Latin1** naturellement accessibles en **XHTML** depuis un document **XML** de notre composition :

```
<!ENTITY % HTMLlat1 PUBLIC
    "-//W3C//ENTITIES Latin 1 for XHTML//EN"
    "xhtml-lat1.ent">
%HTMLlat1;
```

🔗 Voir l'entité externe xhtml-lat1.ent

[https://www.w3.org/TR/xhtml1/dtds.html#a_dtd_Latin-1_characters]

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xml/dtd/ent-param-ext-exemple.html>]

4.6 Espaces de noms

La **DTD** d'un document utilisant des espaces de noms doit définir les éléments à l'aide de leur nom qualifié (*Qname*) comportant le préfixe et la partie locale.

```
<!ATTLIST html xml:lang NMTOKEN #IMPLIED>
```

4.7 Validation

> Validateurs en ligne

Comme le montrent les exemples de ce cours, les navigateurs se contentent en général de documents bien formés, et n'effectuent pas l'étape de validation.

Pour un usage occasionnel il existe des services de validation en ligne :

🔗 <http://validator.w3.org/>

[<http://validator.w3.org/>]

🔗 <http://www.cogsci.ed.ac.uk/~richard/xml-check.html>

[<http://www.cogsci.ed.ac.uk/~richard/xml-check.html>]

> IDEs

Pour un usage plus intensif il sera plus utile d'utiliser un environnement de développement intégré, dédié à XML (*liste subjective et non exhaustive*) :

🔗 XML Copy Editor

[<http://xml-copy-editor.sourceforge.net/>]

🔗 XML Notepad

[<https://xmlnotepad.codeplex.com/>]

🔗 XML Spy

[<http://www.altova.com/simpliedownload2c.html>]

> APIs

Enfin, pour les plus geeks, il existe des APIs pour valider des documents XML depuis à peu près n'importe quel langage. Par exemple :

☞ java

[<http://docs.oracle.com/javase/7/docs/api/javax/xml/validation/package-summary.html>]

☞ Perl

[<http://search.cpan.org/~bbc/XML-Validate-1.025/lib/XML/Validate.pm>]

☞ Python

[<http://lxml.de/validation.html>]

5. Schémas XML

5.1 Introduction

5.1.1 Pourquoi des schémas XML ?

Les spécifications **XML** définissent ce qu'est un document **bien formé** et **valide**.

Le premier point signifie tout simplement que le document respecte la syntaxe **XML**, alors que le second impose qu'il possède une **DTD** et qu'il en respecte les contraintes.

Toutefois, certaines applications peuvent nécessiter la définition de structures plus riches et de contraintes différentes de celles qui sont à la portée d'une **DTD**.

D'autre part, il paraîtrait souhaitable que les contraintes soient exprimées à l'aide d'une syntaxe **XML** (*contrairement aux DTD*) ce qui permettrait de les traiter avec des outils standards.

Au regard de ces considérations, le **W3C** a jugé nécessaire de concevoir un langage simple et facile à utiliser, qui soit plus expressif qu'une **DTD**, qui utilise une syntaxe **XML** et soit compatible avec le reste de l'éco-système (*Namespaces, HTML, DOM, RDF, XSLT, ...*).

5.1.2 Les spécifications XML Schema

Les spécifications de **XML Schema** ont été découpées en trois documents :

XML Schema Part 0 : Primer (Rec. W3C - 2ème éd. Octobre 2004)

Un document non normatif qui décrit le langage afin d'en faciliter la compréhension et d'aider au développement de schémas.

 Consulter le document

<http://www.w3.org/TR/xmlschema-0/>

XML Schema Part 1 : Structures (Rec. W3C - 2ème éd. Octobre 2004)

Précise ce qu'est un schéma **XML**, référence les éléments permettant de construire des schémas, et définit comment appliquer des schémas aux documents **XML**.

 Consulter le document

<http://www.w3.org/TR/xmlschema-1/>

XML Schema Part 2 : Datatypes (Rec. W3C - 2ème éd. Octobre 2004)

Présente les types de données qui peuvent être utilisées dans un schéma **XML**. Ces données peuvent correspondre à la valeur d'un noeud texte ou d'un attribut. Peut être utilisée hors du contexte "*schémas*".

 Consulter le document

<http://www.w3.org/TR/xmlschema-2/>

5.1.3 Fiche d'identité

XML Schema est une application **XML**. L'extrait ci-dessous en fait apparaître l'**URI** public et l'**URL** privée permettant de localiser la **DTD**, ainsi que l'**URI** d'espace de noms et un préfixe usuel.

```
<?xml version="1.0"?>
<!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN"
        "http://www.w3.org/2001/XMLSchema.dtd">
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    .
    .
    .
</xs:schema>
```

 Consulter la DTD des schémas

<http://www.w3.org/2001/XMLSchema.dtd>

N.B. Il existe deux préfixes couramment associés aux schémas : "**xs**" et "**xsd**".

Outre la **DTD** dont l'**URL** est mentionnée ci-dessus, il existe également un schéma de **XML Schema**.

☞ Accéder au schéma des schémas

<http://www.w3.org/2001/XMLSchema.xsd>

5.1.4 Associer un schéma à un document

L'association d'un schéma à un document **XML** se fait à l'aide d'un attribut spécial, de préférence affecté à l'élément racine du document.

Cet attribut est rattaché à un espace de noms particulier, et mentionne l'**URL** du schéma associé au document :

```
<page id="xml/xsd/intro/usage.xml"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="syntax/cours.xsd">
  . . .
</page>
```

Cet exemple correspond au cas le plus simple, celui où le document n'utilise pas d'espaces de noms pour son vocabulaire propre.

> Espaces de noms

Dans le cas contraire, l'attribut associant le schéma au document doit indiquer quel est l'espace de noms du vocabulaire auquel correspond le schéma :

```
<dm:page id="xml/xsd/intro/usage.xml"
          xmlns:dm="http://tic.ec-lyon.fr/~muller/2002/cours"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://tic.ec-lyon.fr/~muller/2002/cours syntax/cours.xsd">
  . . .
</dm:page>
```

On remarquera que l'espace de noms est logiquement identifié par son **URI** et non par un quelconque préfixe...

N.B. Dans le cas où le document utilise plusieurs vocabulaires, avec chacun son espace de noms et un schéma associé, l'attribut "*schemaLocation*" prend pour valeur une liste de paires (*séparées par des blancs*) formées par un **URI** d'espace de noms suivi par l'**URL** du schéma correspondant.

> Des éléments indicatifs

Les spécifications disent que ces informations doivent être considérées comme indicatives.

Toute application travaillant avec un document et un schéma associé, doit offrir à l'utilisateur des méthodes alternatives (*fichier de configuration, ligne de commande, ...*) qui lui permettront d'utiliser un schéma autre que celui mentionné au sein du document.

5.1.5 A quoi sert un schéma XML ?

Un schéma donne un ensemble de règles décrivant la structure et le contenu d'une classe de documents **XML**. Ces informations sont susceptibles d'être exploitées à bien des fins différentes, rapidement évoquées ci-dessous.

> Validation

La validation est l'application la plus courante des schémas. Elle consiste à vérifier que le document est bien conforme à ce qui est décrit dans le schéma.

Cette validation vérifie la conformité structurelle du document (*arborescence des éléments et attributs*) ainsi que le type des données (*valeurs textuelles*) qui doivent correspondre à ce qui est autorisé par le schéma.

La validation permet de vérifier qu'un document reçu par une application est conforme à ce qu'elle attend, de manière à s'assurer que les traitements qui seront effectués ne soient pas erronés.

> Documentation

L'un des avantages essentiels d'un document **XML** est qu'il est lisible par des humains. Cette remarque s'applique bien sûr aux schémas.

Un schéma est une excellente méthode pour documenter un vocabulaire **XML** de manière précise, et impossible à atteindre en plein texte.

Une fois écrit, il suffit de tenter la validation d'une série de documents existants pour vérifier si la *"documentation"* est à jour !

Sachant qu'un schéma est lui-même un document **XML**, il est d'autre part tout à fait envisageable de le visualiser ou de le transformer à l'aide d'outils génériques (*éditeur d'arbre XML, XSLT, ...*) afin de le rendre encore plus lisible par un humain (*peut-être moins informaticien que d'autres*).

> Requêtage

Certaines applications travaillent en extrayant des parties de documents pour leur appliquer un traitement spécifique (*cf. XQuery ou XSLT via XPath...*).

A l'heure actuelle, ces applications sont totalement génériques, et travaillent *"en aveugle"*, sans connaître a priori la structure du document analysé.

Si la généricité fait évidemment la force de telles applications, il n'en demeure pas moins que certaines opérations comme le tri (*alphabétique, numérique, par date, ...*) pourraient être bien optimisées par la connaissance de la structure du document et de la nature des données.

Ces informations sont contenues dans les schémas. C'est la raison pour laquelle les spécifications en cours de gestation (*XPath 2.0, XSLT 2.0, XQuery 1.0*) s'appuieront sur les schémas quand ils seront disponibles.

> Edition

L'information disponible dans un schéma peut permettre à un éditeur générique intelligent de s'adapter à une application particulière et de ne proposer à l'utilisateur que les choix qui sont autorisés à tout instant en fonction du contexte (*éléments structurels*).

De plus, l'éditeur pourra également vérifier la valeur textuelle des noeuds, pour s'assurer qu'elle est compatible avec les types de données préconisés par le schéma.

5.2 Principes

5.2.1 Exemple de schéma XML

Considérons une application simple, qui liste l'ensemble des étudiants qui assistent à un cours :

```
<classe>
  <promo>2004</promo>
  <étudiant>
    <prénom>Raymond</prénom>
    <nom>Deubaze</nom>
  </étudiant>
  <étudiant>
    <prénom>Ginette</prénom>
    <nom>Ringard</nom>
  </étudiant>
  ...
</classe>
```

L'extrait de schéma suivant définit la liste des éléments de cette application :

```
<xs:element name="classe" type="type-classe"/>
<xs:element name="promo" type="xs:string"/>
<xs:element name="étudiant" type="type-etudiant"/>
<xs:element name="prénom" type="xs:string"/>
<xs:element name="nom" type="xs:string"/>
```

On remarque que les éléments texte sont déclarés comme étant du type `"xs:string"`. La présence du préfixe correspondant à l'espace de noms de **XML Schema** n'est pas fortuite : il indique qu'il s'agit d'un type prédéfini.

Le type des éléments à contenu élémentaire (*element content*) doit être défini par ailleurs :

- L'élément `"étudiant"` comportera un `"prénom"` et un `"nom"`, dans cet ordre :

```
<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

- L'élément classe comportera un élément **promo** suivi par un ou plusieurs éléments `"étudiant"` :

```
<xs:complexType name="type-classe">
  <xs:sequence>
    <xs:element ref="promo"/>
    <xs:element ref="étudiant" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/exemple-classe.html>]

5.2.2 Modèles de contenu courants

> Éléments texte

Le type des éléments à contenu purement textuel est dit **simple** :

```
<nom>Deubaze</nom>
```

XML Schema propose une bibliothèque de types simples réutilisables (*chaînes de caractères, nombres, dates, ...*) :

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

De plus, il est possible de définir ses propres types simples par dérivation à partir de types existants (*extension, restriction ou union*).

L'exemple ci-dessous impose que la valeur de l'élément `"nom"` soit un mot en minuscules, de 2 caractères au minimum, avec une majuscule initiale :

```
<xs:element name="nom" type="type-nom"/>
<xs:simpleType name="type-nom">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z] [a-z] +"/>
  </xs:restriction>
</xs:simpleType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/text-content-exemple.html>]

> Éléments texte avec attributs

Les éléments à contenu textuel pur munis d'attributs constituent un cas particulier :

```
<prénom usuel="oui">Raymond</prénom>
```

XML Schema parle ici d'éléments de **type complexe à contenu simple**.

On définit ce type d'éléments par dérivation à partir d'un type simple, ou d'un élément complexe à contenu simple, en ajoutant (*extension*) les attributs :

```
<xs:element name="prénom" type="type-prénom"/>
<xs:complexType name="type-prénom">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="usuel" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/text-attribute-content-exemple.html>]

Noter au passage que les attributs possèdent eux aussi un modèle de contenu simple référencé à l'aide de l'attribut "*type*" de l'élément *xs:attribute*".

> Éléments à contenu élémentaire

Les éléments ne contenant que des sous-éléments sont dits à contenu élémentaire (*element content*) :

```
<étudiant>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
</étudiant>
```

XML Schema considère que le type de tous les éléments à contenu autre que purement textuel est **complexe**. Il n'y a pas de types complexes prédéfinis.

Contrairement à ce qui se passe pour les nouveaux types simples, qui ne peuvent être créés mais uniquement obtenus par dérivation à partir de types existants (*prédéfinis ou non*), les types complexes peuvent être soit créés, soit dérivés.

Exemple de création d'un nouveau type complexe :

La création d'un type complexe se fait essentiellement en donnant la liste (*xs:sequence*) des sous-éléments autorisés. Cette liste peut éventuellement prendre en compte des alternatives mutuellement exclusives (*xs:choice*).

L'exemple ci-dessous définit un élément "*étudiant*" qui doit contenir exactement un "*nom*" et éventuellement plusieurs "*prénom*", dans un ordre quelconque.

```
<xs:element name="étudiant" type="type-étudiant"/>
<xs:complexType name="type-étudiant">
  <xs:choice>
    <xs:sequence>
      <xs:element ref="nom"/>
      <xs:element ref="prénom" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element ref="prénom" maxOccurs="unbounded"/>
      <xs:element ref="nom"/>
      <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/element-content-exemple.html>]

Exemple de dérivation à partir d'un type existant :

Soit à autoriser un attribut *"civilité"* pour l'élément *"étudiant"* :

```
<étudiant civilité="M.">
  <prénom>Jean</prénom>
  <nom>Aymard</nom>
</étudiant>
```

L'exemple ci-dessous s'appuie sur le type *"étudiant"* préalablement défini en ajoutant l'attribut par extension :

```
<xs:element name="étudiant" type="etudiant-civilise"/>
<xs:complexType name="etudiant-civilise">
  <xs:complexContent>
    <xs:extension base="type-etudiant">
      <xs:attribute name="civilité" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/element-content-exemple2.html>]

> Éléments à contenu mixte

Les éléments à contenu mixte peuvent contenir aussi bien du texte que des sous-éléments :

```
<prénom>Stanislas 1<sup>er</sup></prénom>
```

Les éléments à contenu mixte sont déclarés comme des éléments complexes, avec un attribut supplémentaire indiquant la présence éventuelle de texte :

```
<xs:element name="prénom" type="type-prénom"/>
<xs:element name="sup" type="xs:string"/>
<xs:complexType name="type-prénom" mixed="true">
  <xs:sequence>
    <xs:element ref="sup" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Voir un exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/mixed-content-exemple.html>]

Remarquer que ceci revient à traiter les éléments à contenu mixte comme un cas particulier des éléments à contenu élémentaire.

Il en résulte que les possibilités de contraintes sur la structure du document (*ordre, imbrication et cardinalité des éléments*) existent de la même manière pour les éléments à contenu mixte. Il est par contre tout à fait impossible de contrôler finement la position ou la nature du texte présent entre les éléments.

> Éléments vides

Les éléments vides ne peuvent contenir ni de texte, ni d'autres éléments, mais peuvent éventuellement être porteurs d'attributs :

```
<promo année="2005"/>
<étudiant>
  <prénom>Snoopy</prénom>
  <mascotte/>
</étudiant>
```

Pour définir le type de contenu d'un élément vide, on peut spécifier un contenu complexe sans sous-éléments, ou alors un contenu simple de longueur nulle.

Élément vide - modèle complexe :

```
<xs:complexType name="empty">
  <xs:sequence/>
</xs:complexType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/empty-content-exemple1.html>]

Élément vide - modèle simple :

```
<xs:simpleType name="empty">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/empty-content-exemple2.html>]

N.B. Un élément vide ne peut comporter ni de noeud texte, ni de sous-élément. Il peut par contre contenir des commentaires ainsi que des instructions de traitement.

5.2.3 Création de types par dérivation

Les exemples précédents ont montré qu'il était possible de créer de nouveaux types de contenu par extension ou dérivation d'un type existant :

```
<xs:complexType name="type-mascotte">
  <xs:simpleContent>
    <xs:extension base="empty">
      <xs:attribute name="source" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Exemple d'extension :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/derivation-exemple1.html>]

Les possibilités de dérivation sont la **restriction**, la **liste**, et l'**union**.

La restriction est un peu particulière, puisqu'elle permet de limiter le champ des possibles en agissant sur certaines caractéristiques du type modifié (*longueur d'une chaîne, valeur d'un nombre, ...*). Les paramètres sur lesquels il est possible d'agir par restriction sont appelés des **facettes**.

La déclaration d'un élément vide sur la base d'un contenu simple est un exemple de dérivation par restriction agissant sur la facette "*maxLength*" du type prédéfini "*xs:string*" :

```
<xs:simpleType name="empty">
  <xs:restriction base="xs:string">
    <xs:maxLength value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/derivation-exemple2.html>]

5.2.4 Définitions locales

Les définitions d'éléments et les définitions de type vues en exemple jusqu'à présent déclaraient des éléments et des types identifiés par un nom (*cf. attribut name*) d'une portée **globale**, accessibles depuis l'ensemble du schéma :

```
<xs:element name="étudiant" type="type-etudiant"/>
<xs:element name="prénom" type="xs:string"/>
<xs:element name="nom" type="xs:string"/>

<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

N.B. Les définitions globales servent souvent à constituer des bibliothèques de types réutilisables.

XML Schema offre également la possibilité de définir les éléments et les types localement, sans leur donner de nom.

Définition locale de type :

L'exemple ci-dessous montre l'élément *"étudiant"* dont le type (*non nommé*) est défini localement :

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

<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="prénom"/>
      <xs:element ref="nom"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Définition locale d'élément :

Comme le montre l'exemple ci-dessous, les éléments peuvent eux-mêmes être définis localement (*plutôt que référencés*) :

```
<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Aller au bout de la logique :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/localdefs-exemple.html>]

On parlera de **schémas plats** (*cf. structure de l'arbre XML*) lorsque ceux-ci sont fortement axés sur des définitions globales, et de **schémas profonds** lorsqu'ils font surtout usage de définitions locales.

5.2.5 Eléments fonction du contexte

Les spécifications de **XML** disent qu'une application comporte un certain nombre d'éléments identifiés par leur *"type"* (i.e. *par leur nom*), éventuellement discriminés par des espaces de noms.

Or les définitions locales, permettent de facto l'usage d'éléments dont la définition varie en fonction du contexte :

```
<cours>
  <nom>Technologies XML</nom>
  <étudiant>
    <nom>Deubaze</nom>
  </étudiant>
</cours>
```

Dans l'application ci-dessus, le *"nom"* du cours sera une chaîne générique, tandis que le *"nom"* de l'étudiant sera limité à un seul mot commençant par une majuscule :

```
<xs:complexType name="type-etudiant">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element name="nom">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="[A-Z] [a-z] +"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="type-classe">
  <xs:sequence>
    <xs:element name="nom" type="xs:string"/>
    <xs:element ref="promo"/>
    <xs:element ref="étudiant" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/principes/typecontext-exemple.html>

N.B. Cette approche était courante pour les attributs (*il est coutumier d'avoir des attributs portant le même nom attachés à des éléments différents*), mais est relativement nouvelle pour les éléments.

Elle est généralisée par l'application **XML Schema** elle-même qui réutilise dans des contextes différents des éléments avec des significations voisines mais pas toujours identiques.

5.3 Les types simples prédéfinis

5.3.1 Introduction

Avant de s'appesantir sur l'ensemble des types prédéfinis, et afin de bien comprendre certains aspects, il est nécessaire de détailler le processus allant de la lecture du document brut à l'obtention d'un **arbre XML** en mémoire.

> Espace physique

On conviendra d'appeler **espace physique** le niveau correspondant au texte brut tel qu'il apparaît dans un ou plusieurs documents physiques.

N.B. Lorsque les documents sont conservés sur un support quelconque (*disque, bande, mémoire flash, ...*) l'espace physique pourrait être appelé **espace de stockage**. Ce terme ne peut toutefois pas être employé dans le cas général pour tenir compte des documents purement dynamiques.

> Espace Unicode

La recommandation **XML 1.0** indique que la façon dont les documents sont stockés, segmentés ou transmis n'est pas significative pour les applications. Les parseurs conformes à **XML** doivent avant tout convertir la forme physique d'un document pour obtenir une version **normalisée**.

Une première étape consiste à convertir tous les caractères en **Unicode**, et ceci quel que soit l'encodage utilisé dans l'espace physique. Cette transformation concerne l'ensemble du document (*noms des éléments, des attributs, contenu textuel, ...*).

On conviendra dans ce cours d'appeler l'état obtenu après prise en compte des divers documents physiques et conversion des codes caractères l'**espace Unicode**.

> Espace normalisé

Dans un second temps, tous les espaces (*TAB* - #x09, *NL* - #x0A, *CR* - #x0D) sont convertis en blancs (#x20). Cette transformation est appelée la **normalisation**. Le nombre de caractères n'est pas modifié par cette opération.

Un analyseur validant normalise (*aux fins de validation*) tous les types de données prédéfinis sauf *"xs:string"*.

On dira après cette étape qu'une information se trouve dans l'**espace normalisé**.

> Espace lexical

Avant de pouvoir valider le document, les recommandations définissent une seconde transformation appelée **compactage** qui consiste à retirer les blancs initiaux et finaux, et à remplacer toutes les séquences de blancs consécutifs par un blanc unique.

Le **compactage** modifie le nombre de caractères. Il s'applique à tous les types de données prédéfinis sauf *"xs:string"* et *"xs:normalizedString"*.

La valeur obtenue après compactage éventuel appartient alors à l'**espace lexical**.

> Espace des valeurs

Dans l'espace lexical, on ne dispose encore que d'une représentation textuelle d'une valeur logique dont le sens est défini par le type qui lui est associé (*chaîne, nombre, date, ...*).

L'ensemble constitué des valeurs logiques, après prise en compte du type s'appelle l'**espace des valeurs**.

> Remarques

Chaque type de données a son propre espace lexical (*ensemble des représentations textuelles autorisées*) et son propre espace des valeurs. La correspondance entre ces deux espaces est spécifique à chaque type.

Il arrive souvent qu'une même valeur puisse avoir plusieurs représentations lexicales (*cf. 1, +1, 10e-1, ...*).

Cette subtilité est importante lors de comparaisons ou d'opérations de tri. En effet les valeurs associées aux représentations lexicales *"1"* et *"1.0"* seront les mêmes si leur type est *"xs:float"* mais différentes si c'est *"xs:string"*.

Enfin, il est important de noter que les informations transmises à une application sont celles situées dans un **espace normalisé** légèrement différent de celui présenté ici (*retours à la ligne #x0A, et compactage des valeurs d'attributs*). Toutes les opérations suivantes sont uniquement effectuées à des fins de validation et ne concernent que les analyseurs validants.

5.3.2 Chaînes brutes

> xs:string

Le type *"xs:string"* est le seul type prédéfini pour lequel il n'y a pas de normalisation (*et par suite, par de compactage non plus*) avant validation. L'**espace des valeurs** est confondu avec l'**espace Unicode**.

En théorie cet aspect ne concerne que la **validation** et ne préjuge donc pas de ce qui est transmis à l'application...

> `xs:normalizedString`

Comme son nom l'indique, une donnée du type "`xs:normalizedString`" est normalisée mais non compactée. L'**espace des valeurs** est confondu avec l'**espace normalisé**.

La seule différence avec "`xs:string`" est que les séparateurs non blancs sont transformés en blancs avant validation.

N.B. Il s'agit du seul type prédéfini normalisé mais non compacté.

5.3.3 Chaînes de caractères

> `xs:token`

Ce type est une version compactée de "`xs:normalizedString`". Les blancs initiaux et finaux ont été supprimés, et les blancs consécutifs réduits à une seule occurrence. L'**espace des valeurs** est confondu avec l'**espace lexical**.

```
<xs:attribute name="titre" type="xs:token"/>
```

```
<cours titre="XML, standards et applications"/>
```

N.B. En toute rigueur, et contrairement à ce que semble indiquer le nom de ce type, le résultat obtenu dans l'espace des valeurs ressemble plus à une liste de mots séparés par un espace, qu'à une unité lexicale unique...

> `xs:NMTOKEN`

Ce type correspond à un **nom XML** sans la restriction sur la nature du premier caractère. Seuls sont autorisés les caractères alphanumériques, et les caractères "*souligné*" (_), "*tiret*" (-), "*point*" (.) et "*double-point*" (:). Les blancs sont interdits.

```
<xs:attribute name="sigle" type="xs:NMTOKEN"/>
```

```
<produit sigle="2B3"/>
```

"`xs:NMTOKEN`" est dérivé de "`xs:token`" par restriction.

> `xs:NMTOKENS`

Une information du type "`xs:NMTOKENS`" est une liste de données du type "`xs:NMTOKEN`" séparées par des espaces.

```
<xs:attribute name="hosts" type="xs:NMTOKENS"/>
```

```
<Allow hosts="156.18.10.1 localhost tic01.tic.ec-lyon.fr"/>
```

"`xs:NMTOKENS`" est dérivé de "`xs:NMTOKEN`" par liste.

> `xs:Name`

Il s'agit ici d'un **nom XML** avec la restriction sur la nature du premier caractère qui ne peut être qu'alphabétique.

```
<xs:element name="prénom" type="xs:Name"/>
```

```
<prénom>Jean-Pierre</prénom>
```

"`xs:Name`" est dérivé de "`xs:token`" par restriction.

> `xs:NCName`

"`NCName`" signifie "*Non Colonized Name*" (nom sans double-points). Ce type correspond à "`xs:Name`" sans le droit au "*double-point*" (:).

Il s'agit du type le proche de ce que l'on pourrait appeler un nom (*quoique sans espaces*) dans le langage courant, ou un nom de variable en programmation, bien que son usage en tant que tel autorise encore des caractères tels le "*souligné*" (_), le "*tiret*" (-) et le "*point*" (.).

```
<xs:element name="HostName" type="xs:NCName"/>
```

```
<HostName>tic01.tic.ec-lyon.fr</HostName>
```

"*xs:NCName*" est dérivé de "*xs:Name*" par restriction.

> *xs:QName*

"*QName*" signifie "*Qualified Name*". Il s'agit d'un **nom XML** associé à son **espace de noms**. Au niveau de l'espace lexical est permis un unique caractère "*double-point*" (:) non situé en position initiale.

L'espace des valeurs est très particulier, puisque la valeur associée à un nom qualifié est constituée d'un couple comportant l'**URI de l'espace de noms** et la chaîne suivant le "*double-point*".

Dans le cas suivant (*déclaration d'un attribut type du type QName*) :

```
<xs:attribute name="type" type="xs:QName"/>
```

```
<variable type="xs:string"/>
```

et dans la mesure où le préfixe "*xs*" a été déclaré de la manière habituelle, alors la chaîne "*xs:string*" (dans l'espace lexical) aura pour valeur (dans l'espace des valeurs) le couple ("*http://www.w3.org/2001/XMLSchema*", "*string*").

Si le préfixe apparaissant dans l'espace lexical n'a pas été associé à un **URI** d'espace de noms, alors l'information ne sera pas validée.

N.B. En l'absence de préfixe, le terme correspondant à l'**URI** d'espace de noms dans l'espace des valeurs vaut "*NULL*".

"*xs:QName*" est un type primitif. Il n'est pas obtenu par dérivation.

5.3.4 Identifiants uniques

> *xs:ID*

L'espace lexical du type "*xs:ID*" est le même que celui du type "*xs:NCName*". En d'autres termes il s'agit là encore d'un **nom XML** privé du droit au "*double-point*" (:).

La différence se situe dans le fait que la valeur obtenue doit être unique au sein du document. Ceci est à rapprocher des attributs du type **ID** tels qu'il est possible de les déclarer à l'aide d'une **DTD** :

```
<xs:attribute name="insee" type="xs:ID"/>
```

```
<personne insee="I2631069210009">
  . . .
</personne>
```

"*xs:ID*" est dérivé de "*xs:NCName*" par restriction.

> *xs:IDREF*

Il s'agit là encore d'un type dont l'espace lexical est identique à celui de "*xs:NCName*".

La valeur obtenue doit correspondre à une valeur du type **xs:ID** trouvée ailleurs dans le même document.

```
<xs:attribute name="insee" type="xs:ID"/>
<xs:attribute name="prof" type="xs:IDREF"/>
```

```
<personne insee="I2631069210009">
  . . .
</personne>

<cours prof="I2631069210009">
  . . .
</cours>
```

"*xs:IDREF*" est dérivé de "*xs:NCName*" par restriction.

> *xs:IDREFS*

Une information du type "*xs:IDREFS*" est une liste de données du type "*xs:IDREF*" séparées par des espaces.

```
<xs:attribute name="élèves" type="xs:IDREFS"/>

<élève id="t1" nom="Deubaze" prénom="Raymond"/>
<élève id="t2" nom="Ringard" prénom="Ginette"/>
<élève id="t3" nom="Aymard" prénom="Jean"/>

<cours nom="XML, Standards et Applications" élèves="t1 t2 t3"/>
```

"*xs:IDREFS*" est dérivé de "*xs:IDREF*" par liste.

5.3.5 Types particuliers

> *xs:language*

"*xs:language*" est spécialement conçu pour accepter les noms de langues normalisés, tels que décrits par la **RFC 1766** "*Tags for the Identification of Languages*".

```
<xs:attribute name="lang" type="xs:language"/>
```

```
<texte lang="en-US"/>
```

🔗 Consulter la RFC 1766

[<https://www.ietf.org/rfc/rfc1766.txt>]

🔗 Voir aussi la RFC 3066

[<https://www.ietf.org/rfc/rfc3066.txt>]

"*xs:language*" est dérivé de "*xs:token*" par restriction.

> *xs:anyURI*

Ce type est construit pour autoriser tous types d'**URI** tels que définis par la **RFC 2396** "*Uniform Resource Identifiers (URI): Generic Syntax*" et la **RFC 2732** "*Format for Literal IPv6 Addresses in URL's*".

```
<xs:attribute name="tdm" type="xs:anyURI"/>
```

```
<cours tdm="http://tic01.tic.ec-lyon.fr/~muller/cours/xml"/>
```

🔗 Consulter la RFC 2396

[<https://www.ietf.org/rfc/rfc2396.txt>]

🔗 Consulter la RFC 2732

[<https://www.ietf.org/rfc/rfc2732.txt>]

La particularité de ce type est que l'**espace des valeurs** et l'**espace lexical** ne sont pas confondus, pour prendre en compte le fait qu'un **URI** ne peut être exprimé qu'à l'aide de caractères **ASCII**.

Voici un exemple :

Espace lexical : `http://www.exemple.fr/Page d'entrée.html`

Espace des valeurs : `http://www.exemple.fr/Page%20d%27entr%e9e.html`

"*xs:anyURI*" est un type primitif. Il n'est pas obtenu par dérivation.

> *xs:boolean*

"*xs:boolean*" est un type primitif dont l'espace lexical est limité aux quatre expressions "*true*", "*false*", "*1*" et "*0*". L'espace des valeurs n'en contient que deux, les valeurs logiques "*vrai*" et "*faux*".

```
<xs:attribute name="hidden" type="xs:boolean"/>
```

```
<circle hidden="true"/>
```

"*xs:boolean*" est un type primitif. Il n'est pas obtenu par dérivation.

> *xs:hexBinary*

Il s'agit d'un type permettant la représentation de flux de données binaires arbitraires. La représentation lexicale d'un octet quelconque consiste à le coder sous la forme de deux caractères hexadécimaux.

```
<xs:element name="data" type="xs:hexBinary"/>
```

```
<data>3F80</data>
```

La valeur d'une donnée du type "*hexBinary*" est la séquence d'octets obtenue par décodage. A titre d'illustration la valeur correspondant à l'exemple ci-dessus est le nombre binaire "*0011 1111 1000 0000*".

"*xs:hexBinary*" est un type primitif. Il n'est pas obtenu par dérivation.

> *xs:base64Binary*

"*base64Binary*" est encore un type permettant de représenter des données binaires. La différence avec le précédent se situe au niveau du codage utilisé.

Le codage "*base64*" décrit par la **RFC 2045** représente un groupe de 6 bits en série par un caractère, ce qui correspond à 4 caractères pour 3 octets (*i.e.* 24 bits).

```
<xs:element name="data" type="xs:base64Binary"/>
```

```
<data>aGVsbG8gV29ybGQgIQ==</data>
```

La valeur d'une donnée du type "*base64Binary*" est la séquence d'octets obtenue par décodage. A titre d'illustration la valeur correspondant à l'exemple ci-dessus est la chaîne de caractères "*hello World !*".

📖 Consulter la RFC 2045

[<https://www.ietf.org/rfc/rfc2045.txt>]

"*xs:base64Binary*" est un type primitif. Il n'est pas obtenu par dérivation.

5.3.6 Nombres

> *xs:decimal*

Le type "*xs:decimal*" permet de représenter n'importe quel nombre décimal. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un signe "+" ou "-" initial, les chiffres de "0" à "9" et le "*point*" (.) qui est obligatoire.

Les zéros initiaux (*avant la virgule, représentée par le point*) et finaux (*après la virgule*) présents dans l'espace lexical ne se retrouvent évidemment pas dans l'espace des valeurs, purement numériques.

```
<xs:attribute name="prix" type="xs:decimal"/>
```

```
<article désignation="baguette" prix="0.60" monnaie="euro"/>
```

"*xs:decimal*" est un type primitif. Il n'est pas obtenu par dérivation.

> xs:integer

Le type "*xs:integer*" permet de représenter n'importe quel nombre entier. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un signe "+" ou "-" initial, et les chiffres de "0" à "9".

Les zéros initiaux présents dans l'espace lexical ne se retrouvent évidemment pas dans l'espace des valeurs, purement numériques.

```
<xs:attribute name="promo" type="xs:integer"/>
<xs:attribute name="effectif" type="xs:integer"/>
```

```
<option sigle="TI" promo="2005" effectif="29"/>
```

"*xs:integer*" est obtenu par restriction à partir de "*xs:decimal*".

> xs:nonPositiveInteger

Ce type permet de représenter l'ensemble des entiers négatifs ou nuls. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont le signe "-" initial (*obligatoire sauf pour la valeur 0*), et les chiffres de "0" à "9".

```
<xs:attribute name="profondeur" type="xs:nonPositiveInteger"/>
```

```
<plongée conditions="apnée" profondeur="-162" unités="m"/>
```

"*xs:nonPositiveInteger*" est obtenu par restriction à partir de "*xs:integer*".

> xs:negativeInteger

Ce type permet de représenter l'ensemble des entiers strictement négatifs. Le nombre de digits n'est pas limité. La représentation lexical comporte un signe "-" obligatoire suivi par un ensemble de chiffres de "0" à "9".

```
<xs:attribute name="datation" type="xs:negativeInteger"/>
```

```
<fossile type="hominidé" nom="Lucy" datation="-3600000"/>
```

"*xs:negativeInteger*" est obtenu par restriction à partir de "*xs:nonPositiveInteger*".

> xs:nonNegativeInteger

Ce type permet de représenter l'ensemble des entiers positifs ou nuls. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un éventuel signe "+" suivi par un ensemble de chiffres de "0" à "9".

```
<xs:element name="âge" type="xs:nonNegativeInteger"/>
```

```
<âge>22</âge>
```

"*xs:nonNegativeInteger*" est obtenu par restriction à partir de "*xs:integer*".

> xs:positiveInteger

Ce type permet de représenter l'ensemble des entiers strictement positifs. Le nombre de digits n'est pas limité. Les seuls signes autorisés dans l'espace lexical sont un éventuel signe "+" suivi par un ensemble de chiffres de "0" à "9".

```
<xs:element name="promo" type="xs:positiveInteger"/>
```

```
<promo>2005</promo>
```

"*xs:positiveInteger*" est obtenu par restriction à partir de "*xs:nonNegativeInteger*".

> *xs:long*

Contrairement aux types précédents dont le nombre de digits n'était pas limité, le type "*xs:long*" admet l'ensemble des entiers signés dont la représentation binaire tient sur un mot de 64 bits, soit les valeurs allant de -9223372036854775808 à +9223372036854775807 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:long*" est obtenu par restriction à partir de "*xs:integer*".

> *xs:int*

Sur le même modèle que le type "*xs:long*", "*xs:int*" correspond aux entiers signés dont la représentation binaire tient sur 32 bits, soit les valeurs allant de -2147483648 à +2147483647 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:int*" est obtenu par restriction à partir de "*xs:long*".

> *xs:short*

Dans la même logique, "*xs:short*" s'adresse aux entiers signés dont la représentation binaire correspond à des mots de 16 bits, soit des valeurs de -32768 à +32767 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:short*" est obtenu par restriction à partir de "*xs:int*".

> *xs:byte*

"*xs:byte*" limite la représentation des entiers signés à 8 bits, soit des valeurs de -128 à +127 inclus.

Sont autorisés dans l'espace lexical un signe "+" ou "-" initial suivi de chiffres de "0" à "9".

"*xs:byte*" est obtenu par restriction à partir de "*xs:short*".

> *xs:unsignedLong*

Le type "*xs:unsignedLong*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 64 bits, soit les valeurs allant de 0 à +18446744073709551615 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedLong*" est obtenu par restriction à partir de "*xs:nonNegativeInteger*".

> *xs:unsignedInt*

Le type "*xs:unsignedInt*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 32 bits, soit les valeurs allant de 0 à +4294967295 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedInt*" est obtenu par restriction à partir de "*xs:unsignedLong*".

> *xs:unsignedShort*

Le type "*xs:unsignedShort*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 16 bits, soit les valeurs allant de 0 à +65535 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedShort*" est obtenu par restriction à partir de "*xs:unsignedInt*".

> *xs:unsignedByte*

Le type "*xs:unsignedByte*" admet l'ensemble des entiers positifs dont la représentation binaire tient sur un mot de 8 bits, soit les valeurs allant de 0 à +255 inclus.

Sont autorisés dans l'espace lexical le signe "+" optionnel, suivi de chiffres de "0" à "9".

"*xs:unsignedByte*" est obtenu par restriction à partir de "*xs:unsignedShort*".

5.3.7 Nombres à virgule flottante

> xs:float

Le type *"xs:float"* s'adresse aux nombres à virgule flottante. La représentation utilisée dans l'espace des valeurs sera conforme au standard **IEEE 754** simple précision.

La représentation standard **IEEE 754** en simple précision met tout d'abord le nombre à représenter sous la forme donnée ci-dessous, en réservant le bit de poids fort pour le signe, suivi par un exposant entier sur 8 bits et 23 bits pour la mantisse :

$$x = \pm m \cdot 2^e$$

Vue la représentation employée, l'exposant se verra compris dans l'intervalle *[-126, +127]*, tandis que la mantisse prendra des valeurs comprises entre *1* et *2-2⁻²³*. De fait, l'intervalle utile, exprimé en base 10, va approximativement de *10⁻³⁸* à *10³⁸* pour les nombres positifs ou négatifs.

N.B. Il est évident qu'il n'est pas possible de représenter de manière exacte n'importe quel nombre réel sous cette forme. La valeur retenue dans l'espace des valeurs pourra donc être uniquement une approximation de la représentation lexicale (*valeur la plus proche*).

Le standard **IEEE 754** prévoit des cas particuliers :


- Il y a de fait deux représentations possibles pour le zéro : *"0"* ou *"0"* pour le zéro positif, et *"-0"* pour le zéro négatif.
- Il existe de plus une représentation particulière pour un nombre plus grand que le plus grand des nombres que l'on peut représenter, c'est à dire *"plus l'infini"* noté *"INF"*. Il y a de même une représentation pour *"moins l'infini"* notée *"-INF"*.
- Enfin, lorsqu'un calcul renvoie une valeur non représentable (*comme 0 fois l'infini*) il existe un code particulier, noté *"NaN"* qui signifie *"Not a Number"*.

L'espace lexical d'un nombre du type *"xs:float"* admet donc les représentations suivantes :

- *"2.1"* : un nombre décimal (*point obligatoire*) précédé d'un signe optionnel,
- *"314.e-2"* : un nombre décimal précédé d'un signe optionnel suivi par la lettre *"e"* ou *"E"*, suivie par un exposant entier éventuellement précédé par un signe optionnel (*aucun espace permis*),
- l'une des valeurs *"INF"*, *"-INF"*, ou *"NaN"* (*attention, +INF est interdit*).

```
<xs:attribute name="valeur" type="xs:float"/>
```

```
<info desc="distance terre-soleil">
  <minimum valeur="147.1e6" unité="km">
  <maximum valeur="152.1e6" unité="km">
  <moyenne valeur="149.6e6" unité="km">
</info>
```

 Page de cours sur le codage IEEE 754

[\[http://dmolinarius.github.io/demofiles/microp/numeration/flp_ieee.html\]](http://dmolinarius.github.io/demofiles/microp/numeration/flp_ieee.html)

> xs:double

Le type *"xs:double"* concerne encore les nombres à virgule flottante. La représentation utilisée dans l'espace des valeurs sera conforme au standard **IEEE 754** double précision.

La seule différence se situe dans la taille de la zone mémoire requise pour stocker la valeur : 64 bits au lieu de 32 bits pour *"xs:float"*. Du coup la taille de l'exposant passe à 11 bits tandis que l'espace réservé à la mantisse comporte 52 bits.

L'exposant se verra par conséquent compris dans l'intervalle *[-1022, +1023]*, tandis que la mantisse prendra des valeurs comprises entre *1* et *2-2⁻⁵²*. L'intervalle utile, exprimé en base 10, va donc approximativement de *10⁻³⁰⁸* à *10³⁰⁸* pour les nombres positifs ou négatifs.

5.4 Création de types simples

5.4.1 Principe des dérivations

XML Schema permet de créer des types personnalisés à partir de types existants (*prédéfinis ou personnalisés*). Le mécanisme permettant de fabriquer ses propres types de données s'appelle la **dérivation**.

Il existe trois méthodes de dérivation permettant de créer de nouveaux types simples : la **restriction**, la **liste** et l'**union**.

La **restriction** consiste à augmenter les contraintes sur les données autorisées, ce qui conduit à diminuer la taille de l'espace lexical. Les facettes qui s'appliquent au type de données originel s'appliquent également au nouveau type restreint.

```
<xs:attribute name="promo" type="T_PROMO"/>
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1999"/>
    <xs:maxInclusive value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple1.html)

La **liste** permet de créer des listes de données appartenant toutes au même type. Les listes possèdent leurs propres facettes et perdent celles des membres qui les composent.

```
<xs:simpleType name="idCours">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:pattern value="[A-Z] [A-Z] ([1-9] | 10)"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
<xs:attribute name="cours" type="idCours"/>
```

```
<étudiant id="e34" cours="TI1 IF2 TI4 PH7 TI8 IF9"/>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple2.html)

L'**union** revient à fusionner les espaces lexicaux de divers types simples. La sémantique attachée à chacun des types d'origine est perdue. Le type résultant ne garde que les facettes spécifiques aux unions, à savoir "**xs:pattern**" et "**xs:enumeration**".

```
<xs:simpleType name="profId">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:pattern value="inconnu"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:IDREF"/>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

```
<cours id="IF3" profId="inconnu">
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/derivation-exemple3.html>]

5.4.2 Dérivation par restriction

La dérivation par **restriction** consiste à créer un nouveau type en renforçant les contraintes sur les données d'un type existant (*appelé type de base*). Ceci revient donc à ce que l'espace lexical du type créé soit un sous-ensemble de celui du type de base.

La restriction s'effectue en agissant sur certaines caractéristiques spécifiques du type de base appelées **facettes**. La grande majorité des facettes agissent sur l'espace des valeurs. Une facette de même nom peut avoir des effets différents selon le type de base auquel on applique la restriction.

L'exemple ci-dessous montre une restriction du type de base "*xs:unsignedShort*" agissant sur les facettes "*xs:minInclusive*" et "*xs:maxInclusive*" pour limiter l'intervalle des valeurs possibles :

```
<xs:simpleType name="type-promo">
  <xs:restriction base="xs:unsignedShort">
    <xs:minInclusive value="1999"/>
    <xs:maxInclusive value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/restriction-exemple1.html>]

Chaque type prédéfini possède une liste de facettes qui s'appliquent à lui. Cette liste est fixe, non modifiable et non extensible.

Quelle que soit la facette utilisée, la sémantique attachée au type de base est conservée, ce qui concerne en particulier les règles de passage de l'espace lexical à l'espace des valeurs, mais aussi la liste des facettes qui s'appliqueront au type créé.

```
<xs:simpleType name="pseudo-nombre">
  <xs:restriction base="xs:NMTOKEN">
    <xs:pattern value="[0-9] +"/>
  </xs:restriction>
</xs:simpleType>
```

L'exemple ci-dessus montre à titre d'illustration une restriction à partir du type "*xs:NMTOKEN*" qui résulte en un type n'acceptant que les chiffres. Pourtant, l'espace des valeurs de ce type nouveau sera tout à fait différent (*il s'agira de chaînes de caractères*) de celui de "*xs:integer*" (*des nombres*).

5.4.3 Facette xs:pattern

> xs:pattern

La facette "*xs:pattern*" est particulière, puisque c'est la seule qui exprime des contraintes sur l'espace lexical pour limiter l'espace des valeurs. Toutes les autres facettes utiles agissent sur l'espace des valeurs.

Cette facette s'applique à tous les types primitifs, ainsi que les types prédéfinis dérivés par restriction (*ce qui exclut les listes xs:NMTOKENS et xs:IDREFS*).

"*xs:pattern*" s'appuie sur une **expression régulière** pour indiquer quels sont les motifs que doivent respecter les formes lexicales du type résultant.

Les expressions régulières sont dérivées de celles du langage **Perl** (*Practical Extraction and Reporting Language*). Leur syntaxe est particulièrement puissante, mais aussi complexe et ardue à appréhender. C'est la raison pour laquelle nous nous contenterons ici de donner quelques exemples.

Liste de valeurs :

L'usage le plus simple de la facette "*xs:pattern*" consiste à créer des listes de possibilités :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999"/>
    <xs:pattern value="2000"/>
    <xs:pattern value="2001"/>
    <xs:pattern value="2002"/>
    <xs:pattern value="2003"/>
    <xs:pattern value="2004"/>
    <xs:pattern value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple1.html>]

L'intérêt par rapport à l'usage d'une restriction par énumération réside dans le fait que la facette "*xs:pattern*" agit sur l'**espace lexical**, alors que la facette "*xs:enumeration*" agit sur l'**espace des valeurs**. Cela signifie qu'une expression comme "*02000*" ne sera pas validée dans le cas présent, alors qu'elle le serait dans le cas d'une énumération.

Alternatives :

Le même effet que ci-dessus aurait pu être obtenu en utilisant l'opérateur "|" qui permet de signifier des alternatives :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999|2000|2001|2002|2003|2004|2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple2.html>]

Classes de caractères :

Il est possible de spécifier des intervalles de caractères admis à l'aide des opérateurs "[" et "]". L'exemple précédent devient :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:pattern value="1999|200[0-5]"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple3.html>]

Grouperment de sous-motifs :

Les sous-motifs peuvent être groupés à l'aide de parenthèses. Ainsi, L'exemple ci-dessous permet de représenter un sigle de la forme "*TI-8*", constitué de deux lettres majuscules, suivies par un tiret obligatoire, suivi par un nombre de *1* à *10* :

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z] [A-Z] - ([1-9] | 10)"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple4.html>]

On peut remarquer que vu le choix du type de base, les sigles appartenant au type ci-dessus garderont la sémantique d'un identifiant unique...

Opérateurs de répétition :

Les caractères individuels, classes, ou groupes (*délimités par des parenthèses*) peuvent être post-fixés par des opérateurs de répétition "?" (zéro ou un), "+" (un ou plus), "*" (zéro ou plus).

Le type défini ci-dessous accepte les sigles formés de 1 ou 2 caractères majuscules, suivi par un tiret et un nombre non nul d'un chiffre au moins, ne commençant pas par le chiffre 0.

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z] [A-Z] ?-[1-9] [0-9] *"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple5.html>]

Opérateur de répétition généralisé :

Dans le cas général, la notation "{n,m}" indique la possibilité d'accepter de n à m occurrences. La définition ci-dessus pourrait alors s'écrire :

```
<xs:simpleType name="T_SIGLE">
  <xs:restriction base="xs:ID">
    <xs:pattern value="[A-Z] {1,2} - [1-9] [0-9] {0,}"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/pattern-exemple6.html>]

5.4.4 Facette xs:enumeration

> xs:enumeration

La facette "*xs:enumeration*" est la seule à s'appliquer à la fois aux nombres, aux listes, et aux chaînes de caractères. Elle permet tout simplement de lister l'ensemble des valeurs autorisées :

```
<xs:simpleType name="T_PROMO">
  <xs:restriction base="xs:unsignedShort">
    <xs:enumeration value="1999"/>
    <xs:enumeration value="2000"/>
    <xs:enumeration value="2001"/>
    <xs:enumeration value="2002"/>
    <xs:enumeration value="2003"/>
    <xs:enumeration value="2004"/>
    <xs:enumeration value="2005"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple1.html>]

Il est important de se rendre compte que, même si l'expression qui apparaît comme valeur de l'attribut "*value*" le fait forcément par l'intermédiaire d'une valeur lexicale (*on est obligé de l'écrire*), la sélection se fait sur l'**espace des valeurs**. Dans l'exemple ci-dessus, la valeur *02005* est ainsi validée.

Cette remarque est particulièrement importante dans le cas des nombres à virgule flottante qui acceptent un grand nombre d'orthographes (*i.e. de valeurs lexicales*) différentes pour une même valeur (*dans l'espace des valeurs*).

Ceci s'applique aussi dans le cas de chaînes de caractères compactées du type *"xs.token"*. Dans ce cas, *"xs.enumeration"* permet de spécifier les items autorisés, aux espaces consécutifs près :

```
<xs:simpleType name="T_OPTION">
  <xs:restriction base="xs:token">
    <xs:enumeration value="Informatique"/>
    <xs:enumeration value="Technologies de l'Information et de la
Communication"/>
  </xs:restriction>
</xs:simpleType>
```

Dans ce cas, des chaînes comme " *Informatique* " ou "*Technologies*<TAB>*de l'Information*<CR><TAB>*et de la Communication*" sont validées.

Constater de visu :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple2.html>]

Il faut se méfier par contre du cas ci-dessous, où justement, les chaînes conformes aux espaces près ne seraient pas validées :

```
<xs:simpleType name="T_OPTION">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Informatique"/>
    <xs:enumeration value="Technologies de l'Information et de la
Communication"/>
  </xs:restriction>
</xs:simpleType>
```

Vérifier ce fait :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/enumeration-exemple3.html>]

N.B. La facette *"xs.enumeration"* ne s'applique pas au type *"xs:boolean"*.

5.4.5 Facettes xs:length et al.

> xs:length

Chaînes de caractères :

La facette *"xs:length"* s'applique au type *"xs:string"* et aux types qui en sont dérivés par restriction. Elle permet de spécifier le nombre de caractères requis dans l'**espace des valeurs**.

```
<xs:simpleType name="T_SIGLE_OPT">
  <xs:restriction base="xs:NCName">
    <xs:pattern value="[A-Z] *"/>
    <xs:length value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Dans l'exemple ci-dessus, les valeurs autorisées sont toutes les combinaisons de deux caractères majuscules.

Vérifier :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple1.html>]

N.B. Cette facette permet très simplement la création d'un type d'élément vide, puisqu'il suffit de la forcer à 0 :

```
<xs:simpleType name="T_VIDE">
  <xs:restriction base="xs:string">
    <xs:length value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Flux binaires :

"*xs:length*" s'applique également aux types "*xs:hexBinary*" et "*xs:base64Binary*" pour lesquels elle spécifie le nombre d'octets :

```
<xs:element name="Authorization" type="T_HTTP_PASSWORD"/>
<xs:simpleType name="T_HTTP_PASSWORD">
  <xs:restriction base="xs:base64Binary">
    <xs:length value="13"/>
  </xs:restriction>
</xs:simpleType>
```

Dans cet exemple, la longueur mentionnée correspond à l'**espace des valeurs** et valide effectivement la chaîne passée ci-dessous qui fait pourtant bien plus de **13** caractères, mais correspond à la valeur "*be-http:cool!*" (qui elle comporte bien 13 caractères).

```
<Authorization>YmUtaHR0cDpjb29sIQ==</Authorization>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple2.html>

Listes :

Enfin, "*xs:length*" s'applique à tous les types dérivés par liste (*prédéfinis ou non*). Cette facette sert dans ce cas à indiquer le nombre exact d'items de liste autorisés.

```
<xs:attribute name="élèves" type="T_MOLECULE_LIST"/>
<xs:simpleType name="T_MOLECULE_LIST">
  <xs:restriction base="xs:IDREFS">
    <xs:length value="6"/>
  </xs:restriction>
</xs:simpleType>
```

Le schéma ci-dessus permet de valider l'extrait suivant, car la liste d'élèves d'une molécule comporte effectivement **6** items :

```
<molécule id="M33" élèves="E262 E133 E486 E532 E122 E44"/>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple3.html>

N.B. En toute rigueur, "*xs:length*" s'applique également aux types "*xs:QName*" et "*xs:anyURI*". Toutefois, les spécifications sont plus que vagues sur la façon de compter des caractères dans les espaces de valeurs très particuliers de ces deux types de données.

> *xs:maxLength*

Cette facette est similaire à "*xs:length*", sauf qu'elle indique le nombre **maximal** de caractères et non pas le nombre **exact**. Elle s'applique aux mêmes types, de la même façon, et appelle les mêmes remarques que "*xs:length*".

> *xs:minLength*

De même, "*xs:minLength*" est en tous points comparable à "*xs:maxLength*", mis à part qu'elle permet de spécifier le nombre **minimal** et non pas le nombre **maximal** de caractères.

A titre d'exemple, la molécule de l'exemple suivant peut contenir soit **5** soit **6** élèves :

```
<xs:attribute name="élèves" type="T_MOLECULE_LIST"/>
<xs:simpleType name="T_MOLECULE_LIST">
  <xs:restriction base="xs:IDREFS">
    <xs:minLength value="5"/>
    <xs:maxLength value="6"/>
  </xs:restriction>
</xs:simpleType>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/length-exemple4.html>]

5.4.6 Facettes min et max

Les facettes décrites dans cette section s'appliquent aux types dont l'espace des valeurs est ordonné. C'est en particulier le cas des nombres, entiers, décimaux ou à virgule flottante, et de leurs types dérivés par restriction, mais pas des chaînes de caractères ni des types dérivés.

> **xs:maxInclusive**

"**xs:maxInclusive**" permet de fixer la borne supérieure de l'espace des valeurs. La valeur fournie fait partie de l'intervalle autorisé.

```
<xs:simpleType name="T_ALTITUDE">
  <xs:restriction base="xs:float">
    <xs:maxInclusive value="8848."/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple1.html>]

> **xs:maxExclusive**

"**xs:maxExclusive**" permet de fixer la borne supérieure de l'espace des valeurs. La valeur fournie ne fait pas partie de l'intervalle autorisé. Cette propriété est plus particulièrement utile pour les nombres décimaux ou à virgule flottante.

```
<xs:simpleType name="T_DECIMAL_NEGATIF">
  <xs:restriction base="xs:decimal">
    <xs:maxExclusive value="0."/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple2.html>]

> **xs:minInclusive**

"**xs:minInclusive**" permet de fixer la borne inférieure de l'espace des valeurs. La valeur fournie fait partie de l'intervalle autorisé.

```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minInclusive value="-273.15"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple3.html>]

> **xs:minExclusive**

"**xs:minExclusive**" permet de fixer la borne inférieure de l'espace des valeurs. La valeur fournie ne fait pas partie de l'intervalle autorisé. Cette propriété est plus particulièrement utile pour les nombres décimaux ou à virgule flottante.

```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minExclusive value="-273.15"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/minmax-exemple4.html>]

5.4.7 Facettes de digits

> xs:totalDigits

La facette "*xs:totalDigits*" s'applique au type "*xs:decimal*" et à ses types dérivés dont font partie tous les types entiers. Elle sert à fixer le nombre **maximum** de chiffres significatifs (*caractères 0 à 9*), ce qui inclut les chiffres après la virgule dans le cas des décimaux.

L'exemple ci-dessous définit un type permettant d'exprimer des années allant de l'an 0 à l'année 9999 :

```
<xs:simpleType name="T_ANNEE">
  <xs:restriction base="xs:unsignedShort">
    <xs:totalDigits value="4"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/digits-exemple1.html>]

On rappelle que les facettes agissent sur l'espace des valeurs ce qui explique qu'une forme lexicale comme "*02005*" soit validée pour le type de l'exemple ci-dessus.

> xs:fractionDigits

Cette facette ne s'applique qu'au seul type prédéfini "*xs:decimal*" (et aux éventuels types non prédéfinis dérivés par restriction). Elle fixe le nombre maximal de chiffres significatifs après la virgule.

```
<xs:simpleType name="T_PRIX">
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/digits-exemple2.html>]

Là encore, la facette agit sur l'espace des valeurs ce qui signifie que l'expression "*1.4200*" sera validée, contrairement à "*1.123*" par exemple.

5.4.8 Restrictions multiples

Il est possible d'obtenir un nouveau type dérivant par restriction d'un type lui-même obtenu à l'aide d'une dérivation par restriction.

Lorsque les restrictions portent sur des facettes différentes, les conditions s'ajoutent pour créer un type dont l'espace des valeurs est à chaque fois plus petit, puisqu'il est inclus (*en général strictement*) dans l'espace de valeurs du type de base.

Lorsqu'une restriction porte par contre sur une facette déjà restreinte par le type de base, alors certaines conditions sont à respecter :

- la facette "*xs:length*", une fois fixée, ne peut plus être modifiée par un type dérivé,
- la plupart des facettes peuvent être rédéfinies à condition d'être plus restrictives que pour le type de base (*facettes d'énumération, maximums, minimums, et contrôle des digits*),
- les applications consécutives de la facette "*xs:pattern*" imposent un respect simultané de tous les motifs. L'espace lexical résultant est donc l'intersection des espaces correspondant à chacun des motifs.

Lors de la création d'un type il est possible d'interdire toute modification ultérieure (*cf. types dérivés*) d'une facette à l'aide de l'attribut "*fixed*". Cet attribut est disponible pour l'ensemble des facettes à l'exception de "*xs:enumeration*" et "*xs:pattern*" :


```
<xs:simpleType name="T_TEMPERATURE">
  <xs:restriction base="xs:float">
    <xs:minExclusive value="-273.15" fixed="true"/>
  </xs:restriction>
</xs:simpleType>
```

Enfin, comme l'on montré quelques-uns des exemples déjà vus, il est également possible d'appliquer des restrictions qui agissent sur plusieurs facettes à la fois :

```
<xs:simpleType name="T_PRIX">
  <xs:restriction base="xs:decimal">
    <xs:pattern value="[0-9]*\.[0-9]{2}"/>
    <xs:maxInclusive value="10.0"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/restmult-exemple2.html>]

Le type ci-dessus pourrait s'appliquer aux prix d'un magasin dont tous les articles ont un prix inférieur ou égal à **10.0**. Les représentations lexicales autorisées comporteront obligatoirement deux chiffres après la virgule. Les zéros non significatifs au-delà du deuxième chiffre après la virgule, comme pour la chaîne **"9.540"**, sont interdits. Les zéros non significatifs en tête de nombre, comme pour **"04.50"** sont autorisés. Les signes (*et par conséquent, les valeurs négatives*) sont interdits.

5.4.9 Dérivation par liste

La dérivation par liste permet d'obtenir un type **"liste"** dont tous les items sont du même type. **"xs:IDREFS"** et **"xs:NMTOKENS"** sont deux exemples de listes prédéfinies, mais il est possible de créer des listes à partir de n'importe quel type simple, prédéfini ou non :

```
<xs:attribute name="coords" type="T_RECT_COORDS">
<xs:simpleType name="T_RECT_COORDS">
  <xs:list itemType="xs:decimal"/>
</xs:simpleType>
```

```
<rectangle coords="10.0 10.0 980.0 150.0"/>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple1.html>]

La définition ci-dessus aurait également pu être effectuée en embarquant une création de type simple comme contenu de l'élément **"xs:liste"** plutôt que de référencer un type global à l'aide de l'attribut **"itemType"** :

```
<xs:simpleType name="T_RECT_COORDS">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:decimal">
        <xs:pattern value="[0-9]{1,3}\.[0-9]?"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple2.html>]

> Facettes

Les types **"liste"** perdent les facettes du type de leurs items et possèdent leurs facettes propres **"xs:length"**, **"xs:minLength"**, **"xs:maxLength"** et **"xs:enumeration"**.

Pour appliquer une restriction sur une facette d'un type liste créé, ce type doit forcément être global. Le type défini ci-dessous comporte une liste comportant exactement 4 coordonnées :

```
<xs:simpleType name="T_RECT_COORDS">
  <xs:restriction base="T_COORDS">
    <xs:length value="4"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="T_COORDS">
  <xs:list itemType="xs:decimal"/>
</xs:simpleType>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple3.html>]

N.B. La raison imposant que le type des éléments de la liste soit global est qu'un type simple n'accepte qu'une seule méthode de dérivation à la fois (*restriction, liste, ou union*).

> Limitations

Les listes obtenues de cette manière sont très limitées. En effet, les items de liste sont obligatoirement séparés par un espace (*après compactage*), et il n'est pas possible de spécifier un autre caractère de séparation. Il n'est pas possible non plus de créer des listes dont les éléments n'appartiennent pas au même type simple, ou de créer des listes d'éléments complexes, ni même des listes de listes (*nommément interdit par les spécifications*)...

Lorsque les valeurs du type servant d'item de liste sont susceptibles de contenir des espaces (*comme c'est le cas de xs:string ou xs:token*), alors ces blancs seront pris en compte comme séparateurs de liste, ce qui fait que la facette "*xs:length*" compterait alors des "*mots*" (*en fait, des unités lexicales ne comprenant pas d'espace*)...

```
<xs:element name="summary" type="T_SUMMARY">
  <xs:simpleType name="T_SUMMARY">
    <xs:restriction base="T_WORDS">
      <xs:maxLength value="1000"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="T_WORDS">
    <xs:list itemType="xs:token"/>
  </xs:simpleType>
```

```
<summary>Ceci est un résumé limité à 1000 mots...</summary>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/list-exemple4.html>]

> Remarque

En terme de conception d'applications **XML**, il n'est pas très pertinent d'abuser des types "*liste*", qu'ils servent à recevoir des valeurs d'attributs ou d'éléments. En effet, les mécanismes classiques (SAX, DOM, XPath, ...) ne permettent pas facilement de récupérer les valeurs individuelles des items d'une liste, contrairement à ce qui serait possible avec une liste d'éléments :

```
<molécule id="M33" élèves="E262 E133 E486 E532 E122 E44"/>
```

La solution ci-dessous serait donc si possible préférable :

```
<molécule>
  <élève ref="E262"/>    <élève ref="E133"/>
  <élève ref="E486"/>    <élève ref="E532"/>
  <élève ref="E122"/>    <élève ref="E44"/>
</molécule>
```

5.4.10 Dérivation par union

L'union est une méthode de dérivation qui permet de fusionner les espaces de valeurs de types distincts.

L'exemple ci-dessous permet de rajouter au type "*xs:decimal*" la chaîne de caractère "*undefined*" :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:decimal"/>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:enumeration value="undefined"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Dans le cas ci-dessus, il est possible de raccourcir un peu le code par l'intermédiaire de l'attribut "*memberTypes*" qui permet de spécifier une liste de types membres de l'union :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union memberTypes="xs:decimal">
    <xs:simpleType>
      <xs:restriction base="xs:NCName">
        <xs:enumeration value="undefined"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Ou encore :

```
<xs:simpleType name="T_DECIMAL">
  <xs:union memberTypes="xs:decimal T_UNDEF"/>
</xs:simpleType>

<xs:simpleType name="T_UNDEF">
  <xs:restriction base="xs:NCName">
    <xs:enumeration value="undefined"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/simpleType/union-exemple3.html>

> Facettes

Les seules facettes qui peuvent s'appliquer aux types dérivés par union sont : "*xs:pattern*" et "*xs:enumeration*".

5.5 Création de types complexes

5.5.1 Introduction

Les types simples tels que tous ceux vus jusqu'ici décrivent tous le contenu d'un noeud texte (*élément ou attribut*). Ils n'ont aucun rapport avec un quelconque balisage, et peuvent mener une existence indépendante de **XML Schema** voire même de **XML**.

C'est la raison pour laquelle les spécifications de **XML Schema** comportent deux parties indépendantes, dont l'une est justement consacrée aux types de données prédéfinis.

A contrario, les types complexes décrivent la structure du balisage. Ils sont donc généralement plus spécifiques d'une application et moins réutilisables dans un cadre différent.

Toutefois, et comme pour les types simples, les types complexes peuvent être nommés et définis à l'échelon global, ou alors être anonymes et définis localement en fonction du besoin.

Exemple de définition globale

```
<xs:element name="étudiant" type="T_ETUDIANT"/>
<xs:element name="prénom" type="xs:token"/>
<xs:element name="nom" type="xs:token"/>

<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Exemple de définition locale

```
<xs:element name="étudiant">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="prénom" type="xs:string"/>
      <xs:element name="nom" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Rappelons également qu'on appelle *"complexe"*, le type de tout élément qui ne soit pas à contenu purement **textuel**. Ceci inclut les éléments à contenu textuel munis d'attributs qui seront dits *"éléments complexes à contenu simple"*, tous les autres étant dits *"à contenu complexe"* :

● élément de type simple :

```
<nom>Deubaze</nom>
```

● élément de type complexe à contenu simple :

```
<personne civilité="M.">Raymond Deubaze</personne>
```

● éléments de type complexe à contenu complexe :

```
<personne>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom> </
personne>
```

```
<personne civilité="M.">
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom> </
personne>
```

```
<personne>
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
  Un très bon élève... </
personne>
```

5.5.2 Création de type complexe à contenu simple

Un type complexe à contenu simple, obtenu en ajoutant un attribut à un type simple, est dit **créé par extension** :

```
<personne civilité="M.">Raymond Deubaze</personne>
```

```
<xs:complexType name="T_PERSONNE">
  <xs:simpleContent>
    <xs:extension base="T_NOM_PERSONNE">
      <xs:attribute name="civilité" type="T_CIVILITE"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="T_NOM_PERSONNE">
  <xs:restriction base="xs:token">
    <xs:pattern value="[A-Z][a-z]+ [A-Z][a-z]+">
  </xs:restriction>
</xs:simpleType>
```

L'attribut **"base"** est obligatoire. La définition du type simple ne peut pas se faire localement, comme contenu de l'élément **"xs:extension"**.

5.5.3 Dérivation de type complexe à contenu simple

Un type complexe à contenu simple obtenu en ajoutant un attribut à un autre type complexe à contenu simple est dit **dérivé** par extension :

Par rapport au cas précédent la nuance est subtile, d'autant plus que la syntaxe est strictement identique :

```
<xs:complexType name="T_PERSONNE_NATION">
  <xs:simpleContent>
    <xs:extension base="T_PERSONNE">
      <xs:attribute name="nationalité" type="T_NATION"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

```
<personne civilité="M." nationalité="F">Raymond Deubaze</personne>
```

La seule différence par rapport au cas précédent tient dans le fait que **"T_PERSONNE"** est un type complexe à contenu simple au lieu d'un type simple...

5.5.4 Restriction de type complexe à contenu simple

La dérivation par restriction d'un type complexe à contenu simple apporte une nouveauté par rapport à la restriction d'un type simple : elle agit non seulement sur le contenu textuel mais également sur les attributs.

La syntaxe permettant de restreindre le contenu textuel est la même que pour les éléments simples. Un attribut peut être restreint en modifiant son type pour un type restreint du précédent, ou tout simplement interdit.

Exemple : type de départ

```
<xs:complexType name="T_PERSONNE">
  <xs:simpleContent>
    <xs:extension base="xs:token">
      <xs:attribute name="nationalité" type="xs:NMTOKEN"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Restriction du contenu :

```
<xs:complexType name="T_NOM_PRENOM">
  <xs:simpleContent>
    <xs:restriction base="T_PERSONNE">
      <xs:pattern value="[A-Z][a-z]+ [A-Z][a-z]+" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Restriction du type de l'attribut :

```
<xs:complexType name="T_FRANCAIS">
  <xs:simpleContent>
    <xs:restriction base="T_NOM_PRENOM">
      <xs:attribute name="nationalité" type="T_NAT_F" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>

<xs:simpleType name="T_NAT_F">
  <xs:restriction base="xs:NMTOKEN">
    <xs:pattern value="F" />
  </xs:restriction>
</xs:simpleType>
```

N.B. la restriction de l'attribut aurait également pu être effectuée avec un type local :

```
<xs:complexType name="T_FRANCAIS">
  <xs:simpleContent>
    <xs:restriction base="T_NOM_PRENOM">
      <xs:attribute name="nationalité">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:pattern value="F" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Suppression de l'attribut :

La suppression totale de l'attribut s'effectue à l'aide de la valeur *"prohibited"* de l'attribut *"use"*, comme dans l'exemple ci-dessous :

```
<xs:complexType name="T_APATRIDE">
  <xs:simpleContent>
    <xs:restriction base="T_FRANCAIS">
      <xs:attribute name="nationalité" type="xs:NMTOKEN" use="prohibited" />
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/simpleContent-exemple3.html>

5.5.5 Création de type complexe à contenu complexe

Un **type complexe à contenu complexe** définit la liste des sous-éléments autorisés, l'ordre dans lequel ils peuvent ou doivent apparaître et le nombre d'occurrences autorisées.

> xs:sequence

La façon la plus simple de créer un type complexe consiste à lister les sous-éléments dans l'ordre dans lequel ils sont autorisés à apparaître, à l'aide d'un élément *"xs:sequence"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

N.B. On rappelle que les éléments listés par *"xs:sequence"* peuvent également être nommés (via l'attribut *name*) et définis localement au lieu d'être définis globalement puis référencés (via l'attribut *ref*) :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element name="prénom" type="T_NOM"/>
    <xs:element name="nom" type="T_NOM"/>
  </xs:sequence>
</xs:complexType>
```

Et la logique peut même être poussée encore plus loin, en définissant localement les types des éléments. (cf. ci-dessus *nom* et *prénom*)...

> Contrôle du nombre d'occurrences

Le nombre d'occurrences autorisées pour chacun des éléments peut être indiqué à l'aide des attributs *"minOccurs"* et *"maxOccurs"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom" maxOccurs="3"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Par défaut ces attributs valent **1**, ce qui rend obligatoire une unique occurrence de l'élément concerné.

Pour rendre un élément optionnel, il suffit de mettre la valeur *"minOccurs"* à **0**. Pour ne pas limiter le nombre d'occurrences, il existe la valeur particulière *"unbounded"* :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:element ref="prénom" maxOccurs="unbounded"/>
    <xs:element ref="prénom-usuel" minOccurs="0"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

> xs:choice

Comme avec les **DTD**, il est possible d'offrir des alternatives concernant les listes possibles de sous-éléments. Cette aspect est contrôlé par l'élément *"xs:choice"*, qui permet de proposer une liste d'éléments mutuellement exclusifs :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="prénom"/>
      <xs:element ref="prénom-usuel"/>
    </xs:choice>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Cet exemple impose l'usage d'un prénom ou (*exclusif*) d'un prénom usuel, suivi par un nom.

N.B. On remarque ici que l'élément `"xs:choice"` a pris la place d'un élément `"xs:element"` dans la liste spécifiée par `"xs:sequence"`. En effet, les éléments, les séquences, les choix, constituent des sortes de briques, ou **particules** (*particles*) qui peuvent apparaître chacune en lieu et place d'un élément comme membre d'une séquence ou d'un choix.

On peut mettre en pratique cette propriété pour imposer par exemple un prénom unique ou alors un prénom usuel éventuellement suivi par un ou plusieurs prénoms :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="prénom"/>
      <xs:sequence>
        <xs:element ref="prénom-usuel"/>
        <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:choice>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple3.html>]

> xs:group

En raisonnant sur notre exemple, le choix entre un prénom unique ou un prénom usuel suivi par une liste de prénoms, peut être une brique de base que nous aurions envie de réutiliser par ailleurs.

Or il ne s'agit pas là d'un élément que nous pourrions rendre global, ni même d'un type. C'est là qu'intervient l'élément `"xs:group"` qui consitue une autre brique ou particule réutilisable :

```
<xs:complexType name="T_ETUDIANT">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>

<xs:group name="G_PRENOMS">
  <xs:choice>
    <xs:element ref="prénom"/>
    <xs:sequence>
      <xs:element ref="prénom-usuel"/>
      <xs:element ref="prénom" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:choice>
</xs:group>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple4.html>]

> xs:attribute

La déclaration des attributs d'un type complexe à contenu complexe s'effectue à l'aide d'éléments `"xs:attribute"` placés **après** la brique (*ou particule*) constituant le type :

```
<xs:complexType name="T_PERSONNE">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>
```


De la même manière qu'il a pu être utile d'extraire des groupes structurels d'éléments sous forme de briques afin de pouvoir les réutiliser, il est intéressant, dès que la complexité d'une application devient moyenne, de pouvoir grouper des séries d'attributs couramment utilisés ensemble :

```
<xs:complexType name="T_PERSONNE">
  <xs:sequence>
    <xs:group ref="G_PRENOMS"/>
    <xs:element ref="nom"/>
  </xs:sequence>
  <xs:attributeGroup ref="A_COURANTS"/>
</xs:complexType>

<xs:attributeGroup name="A_COURANTS">
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="nationalité" type="xs:NCName"/>
</xs:attributeGroup>
```

La plupart du temps les attributs sont optionnels. Toutefois, il est possible de rendre un attribut obligatoire à l'aide de l'attribut **"use"** prenant la valeur **"required"** :

```
<xs:attributeGroup name="A_COURANTS">
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="nationalité" type="xs:NCName" use="required"/>
</xs:attributeGroup>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple5.html>]

5.5.6 Dérivation de type complexe à contenu complexe

> Dérivation par extension

La dérivation par extension d'un type complexe à contenu complexe permet d'ajouter des éléments ou des attributs au type de base (*noter l'élément **xs:complexContent** caractéristique d'une dérivation*) :

```
<xs:complexType name="T_ETUDIANT">
  <xs:complexContent>
    <xs:extension base="T_PERSONNE">
      <xs:sequence>
        <xs:element name="email" type="xs:token"/>
      </xs:sequence>
      <xs:attribute name="promo" type="xs:integer" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/complexContent-exemple6.html>]

Malheureusement, ce mécanisme reste très limité. En effet, il n'est possible de compléter le contenu du type de base qu'en rajoutant une brique (*ou particule*) **après le contenu existant**.

Tout se passe comme si on avait un nouveau type construit de la manière suivante :

```
<xs:complexType>
  <xs:sequence>
    <!-- contenu de l'élément de base -->
    <!-- contenu ajouté par extension -->
  </xs:sequence>
</xs:complexType>
```

Il n'est donc pas possible par extension de rajouter un élément à une alternative **"xs:choice"** ou de choisir un autre point d'insertion du contenu venant compléter le type de base.

5.5.7 Type complexe à contenu mixte

XML Schema traite les éléments complexes à contenu mixte comme un cas particulier des éléments complexes à contenu complexe.

Supposons que l'on veuille pouvoir ajouter des commentaires libres, hors éléments, pour qualifier les étudiants de notre exemple :

```
<étudiant>
  Un peu de pipotage ne lui ferait pas de mal !
  <prénom>Raymond</prénom>
  <nom>Deubaze</nom>
</étudiant>
```

Il suffirait tout simplement d'ajouter l'attribut *"mixed"* avec la valeur *"true"* à la déclaration de type :

```
<xs:complexType name="T_ETUDIANT" mixed="true">
  <xs:sequence>
    <xs:element ref="prénom"/>
    <xs:element ref="nom"/>
  </xs:sequence>
</xs:complexType>
```

Cette façon d'aborder le problème permet de conserver tous les mécanismes vus pour les éléments complexes à contenu complexe, mais interdit tout contrôle quant à la position du texte qui peut apparaître n'importe où entre les éléments :

```
<étudiant>
  <prénom>Ginette</prénom>
  <nom>Ringard</nom>
  Devrait se mettre à 1'heure d'Internet...
</étudiant>

<étudiant>
  <prénom>Jean</prénom>
  Se décourage facilement.
  <nom>Aymard</nom>
</étudiant>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/mixedContent-exemple1.html>

5.5.8 Type complexe à contenu vide

Selon **XML Schema**, les éléments vides peuvent être indifféremment considérés comme des éléments à contenu textuel de longueur nulle, ou comme des éléments à contenu élémentaire sans enfants.

Conformément à la seconde hypothèse, un type vide se définit donc de la manière suivante :

```
<xs:complexType name="T_VIDE">
  <xs:sequence/>
</xs:complexType>
```

Un type vide portant des attributs n'est guère plus compliqué à définir :

```
<xs:complexType name="T_PROMO">
  <xs:sequence/>
  <xs:attribute name="année" type="xs:integer"/>
</xs:complexType>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xsd/complexType/emptyContent-exemple1.html>

5.6 Validation

Il existe de nombreux outils compatibles avec **XML Schema** (*éditeurs, validateurs, ...*). Une bonne idée serait sans doute (*car un peu datée*) de consulter la page de référence sur le site du **W3C** qui indique notamment un certain nombre d'analyseurs validants open source et un module de validation pour divers langages comme **C/C++**, **Java**, **Perl** ou **Python** :

 <http://www.w3.org/XML/Schema#Tools>

[\[http://www.w3.org/XML/Schema#Tools\]](http://www.w3.org/XML/Schema#Tools)

Concernant la validation, pour un usage occasionnel il existe des services de validation en ligne, dont par exemple :

 <http://www.freeformatter.com/xml-validator-xsd.html>

[\[http://www.freeformatter.com/xml-validator-xsd.html\]](http://www.freeformatter.com/xml-validator-xsd.html)

Pour un usage plus intensif il est possible d'utiliser un plugin pour navigateur ou un **IDE** (*environnement de développement intégré*) dédié à **XML**. Il en existe quelques-uns qui sont gratuits, les plus perfectionnés sont des outils payants.

Pour les développeurs, il existe dans la plupart des langages une **API XML** proposant un parseur et des validateurs pour **DTD** ou **schéma XML**.

> Exemple

Pour la petite histoire, le mécanisme de validation utilisé dans le cadre de ce cours passe par un programme **PHP** :

```
$xml = new DOMDocument();
$xml->load($full_path);
$schema_path = $xml->documentElement-
>getAttribute("xsi:noNamespaceSchemaLocation");
libxml_use_internal_errors(true);
if ( ! $xml->schemaValidate($schema_path) ) {
    $errors = libxml_get_errors();
    $err_string = "";
    foreach ($errors as $error) {
        $err_string .= get_error($error);
    }
    libxml_clear_errors();
    return $err_string;
}
else return "OK";
```

Ce code est basé sur la classe **DOMDocument** et sa méthode **schemaValidate**, implémentées en **PHP** via la librairie **libxml**.

 La classe **DOMDocument**

[\[http://php.net/manual/fr/class.domdocument.php\]](http://php.net/manual/fr/class.domdocument.php)

 **DOMDocument::schemaValidate**

[\[http://php.net/manual/fr/domdocument.schemavalidate.php\]](http://php.net/manual/fr/domdocument.schemavalidate.php)

 **PHP libxml**

[\[http://php.net/manual/fr/book.libxml.php\]](http://php.net/manual/fr/book.libxml.php)

Ce type de code pourrait très facilement être transposé dans d'autres langages, qui proposent tous le même genre d'**API** pour la validation de documents **XML**.

N.B. Dans l'exemple ci-dessus, la mise en forme des messages d'erreurs est obtenue à l'aide de la fonction :

```
function libxml_get_errors($error){
    $return = "";
    switch ($error->level) {
        case LIBXML_ERR_WARNING:
            $return .= "<b>Warning $error->code</b>: ";
            break;
        case LIBXML_ERR_ERROR:
            $return .= "<b>Error $error->code</b>: ";
            break;
        case LIBXML_ERR_FATAL:
            $return .= "<b>Fatal Error $error->code</b>: ";
            break;
    }
    $return .= trim($error->message);
    if ($error->file) {
        $return .= " in <b>".basename($error->file)."</b>";
    }
    $return .= " on line <b>$error->line</b>\n";
    return $return;
}
```

6. XPath

6.1 Introduction

6.1.1 Fiche d'identité

Xpath est un langage **non-XML** pour identifier des sous-ensembles de documents **XML**.

XPath a fait l'objet de plusieurs recommandations du **W3C** :

📄 XPath 1.0 - novembre 1999

[\[http://www.w3.org/TR/xpath\]](http://www.w3.org/TR/xpath)

📄 XPath 2.0 - décembre 2010

[\[http://www.w3.org/TR/xpath20\]](http://www.w3.org/TR/xpath20)

📄 XPath 3.0 - avril 2014

[\[https://www.w3.org/TR/2014/REC-xpath-30-20140408/\]](https://www.w3.org/TR/2014/REC-xpath-30-20140408/)

📄 XPath 3.1 - mars 2017

[\[https://www.w3.org/TR/xpath-31/\]](https://www.w3.org/TR/xpath-31/)

Xpath fait entre autres partie des "**accessoires**" de **XSLT**. C'est la raison pour laquelle les spécifications de **XPath** sur le site du **W3C** sont notamment accessibles depuis la page principale consacrée à **XSL** (*eXtensible Stylesheet Language*) sur le site du **W3C**.

📄 Consulter la page principale sur XSL

[\[http://www.w3.org/Style/XSL/\]](http://www.w3.org/Style/XSL/)

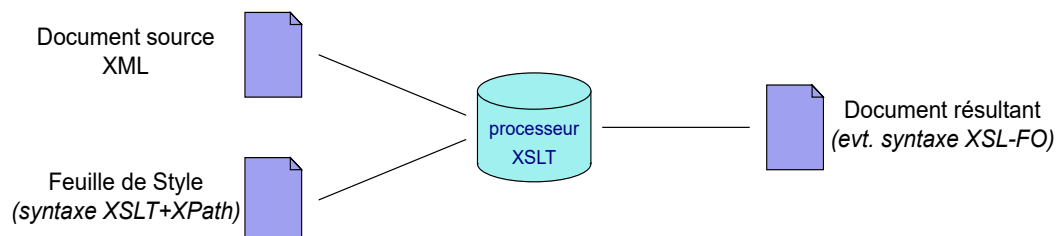
Comme **CSS** s'appuie sur des **sélecteurs** pour désigner certains éléments d'un document, **Xpath** est utilisé par **XSLT** pour sélectionner des éléments particuliers du document **XML** d'entrée et leur faire correspondre un modèle de traitement.

Toutefois, **Xpath** a été conçu comme un mécanisme générique. Il est également utilisé par **Xpointer** et **XQuery**.

6.1.2 Eléments de contexte

XPath est issu des besoins identifiés par le groupe de travail en charge de la définition de l'application de feuilles de styles **XSL** (*mi 1998*). Un an et quelques "**Working Drafts**" plus tard, la décomposition fonctionnelle de **XSL** était achevée.

On dispose depuis - de **XSLT** pour la transformation du document (*recommandation de novembre 1999*), - de **Xpath** (*utilisé par XSLT*) pour la localisation d'éléments dans un document (*novembre 1999*) - et de **XSL-FO** dont la recommandation a été finalisée plus tard, permettant la mise en forme d'un document, en vue notamment de l'impression (*octobre 2001*).



Remarquer que la feuille de style utilise une syntaxe propre à **XSLT**, qui s'appuie sur **XPath** pour désigner les éléments à "**mettre en page**" comme **CSS** s'appuie sur des sélecteurs.

Toutefois, **Xpath** ayant été conçu comme un langage générique, d'autres applications comme **XPointer** et **XQuery** bénéficient des fonctionnalités qu'il apporte.

6.1.3 Expressions XPath

Une expression **Xpath** peut renvoyer un noeud de l'arbre **XML** (*élément, attribut, ...*), un ensemble de noeuds (*node-set*), ou plus simplement une chaîne de caractères, un nombre ou un booléen :

`//étudiant[@id="E111"]` Renvoie un noeud

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/expressions-ex1.html>]

`//étudiant` Renvoie un ensemble de noeuds

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/expressions-ex2.html>]

`count(//étudiant)` Renvoie un nombre

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/expressions-ex3.html>]

`//étudiant[1]/nom = 'Deubaze'` Renvoie un booléen

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/expressions-ex4.html>]

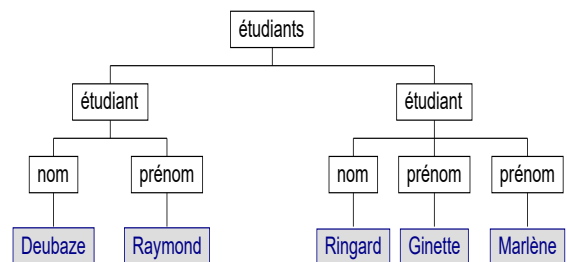
L'expression **XPath** toutefois la plus courante et la plus utile est le **chemin de localisation** (*location path - cf. exemples 1 et 2 ci-dessus*). Un chemin de localisation renvoie un noeud ou un ensemble de noeuds.

N.B. **XPath** travaille sur l'**arbre XML**. Certaines choses sont donc logiquement impossibles. Il n'est par exemple pas possible de savoir si du texte a été spécifié dans une section **CDATA** ou si un caractère particulier a été inséré avec un appel d'entité caractère.

6.1.4 Chemins de localisation

Reprenons le document **XML** bien connu à titre d'exemple :

```
<étudiants>
  <étudiant>
    <nom>Deubaze</nom>
    <prénom>Raymond</prénom>
  </étudiant>
  <étudiant>
    <nom>Ringard</nom>
    <prénom>Ginette</prénom>
    <prénom>Marlène</prénom>
  </étudiant>
</étudiants>
```



Un **chemin de localisation** désigne un élément particulier en navigant à travers l'arbre XML.

Syntaxiquement il ressemble au chemin d'accès d'une **URL** ou d'un fichier sur disque, permettant de naviguer à travers l'arbre des répertoires, à partir de la racine, pour atteindre un document particulier.

Un **chemin absolu** part de la racine du document. Au fur et à mesure de la progression vers l'élément recherché, le chemin de localisation se construit en séparant par des caractères **"/"** le nom des éléments traversés.

Voici par exemple le chemin de localisation permettant de désigner l'élément englobant :

`/étudiants`

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/locpath-ex1.html>]

6.1.5 Exemples de chemins de localisation

Si l'expression ci-dessous désigne bien l'élément englobant :

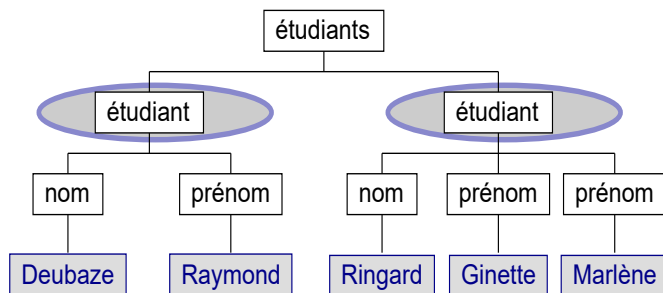
`/étudiants`

Que désigne l'expression suivante ?

`/étudiants/étudiant`

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/locpath-ex2.html>]

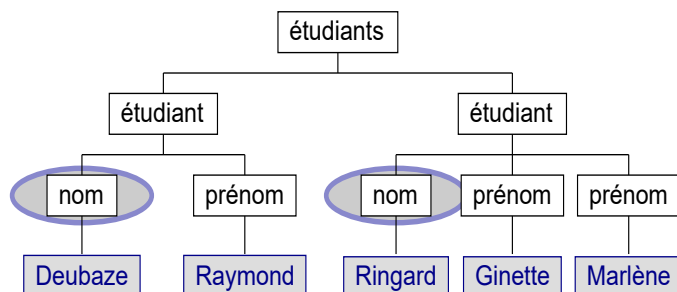


L'expression ci-dessous ?

`/étudiants/étudiant/nom`

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/locpath-ex3.html>]

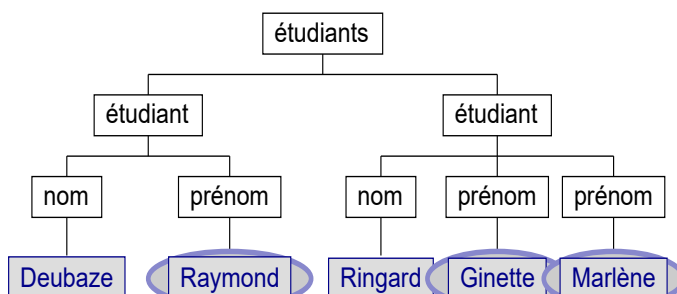


L'expression suivante fait appel à une fonction particulière :

`/étudiants/étudiant/prénom/text()`

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/locpath-ex4.html>]



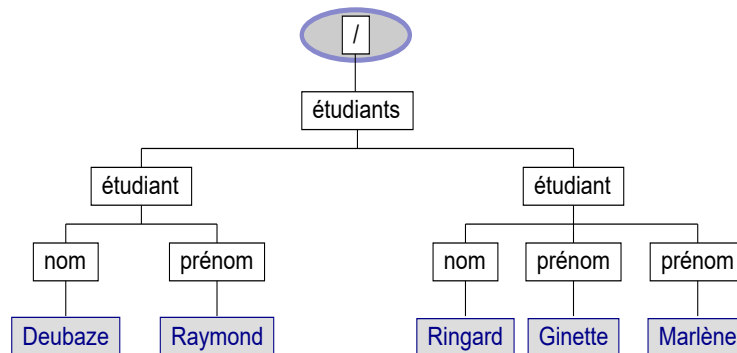
6.2 Chemins de localisation

6.2.1 Chemins de localisation absolus

Un **chemin de localisation absolu** permet de parcourir l'arbre **XML** à partir de la racine. L'expression qui représente un tel chemin commence toujours par le caractère **"/"**.

N.B. L'élément racine (*root*) est un élément **XML** abstrait. Il est représenté par le chemin :

/



En circulant le long d'un chemin de localisation le nom qui suit un caractère **"/"** désigne l'ensemble des éléments qui portent ce nom, descendants directs (*enfants*) de l'élément précédent du chemin.

/étudiants/étudiant/nom

6.2.2 Sélection des descendants

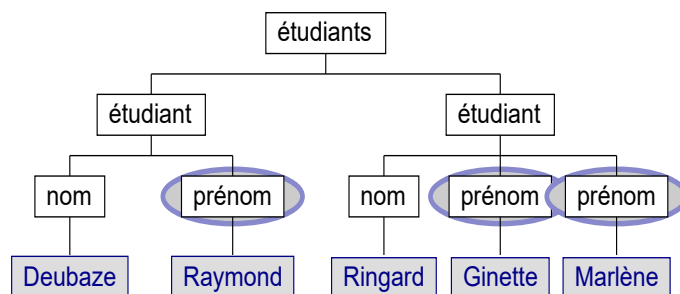
Il est possible de sélectionner un élément parmi l'**ensemble des descendants** à une certaine étape du chemin (*et non pas uniquement parmi les enfants directs*), en faisant précéder le nom de cet élément par un double **"/"**.

//prénom

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/abbrdesc-ex1.html>]

sélectionne l'ensemble des éléments **"prénom"** du document.



Cette notation peut apparaître à n'importe quelle étape du chemin :

/étudiants//prénom

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/abbrdesc-ex2.html>]

sélectionne l'ensemble des éléments **"prénom"**, descendants de l'élément **"étudiants"** (*enfants directs ou descendants lointains*).

6.2.3 Chemins de localisation relatifs

XPath permet également de travailler dans le contexte d'un élément particulier.

Les chemins de localisation partant de l'élément courant sont appelés des chemins **relatifs** et ne comportent pas de caractère "/" initial.

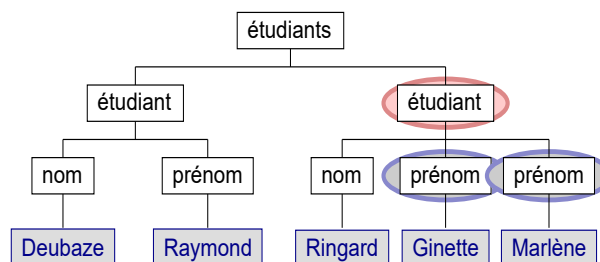
Supposons que l'élément courant soit le deuxième "*étudiant*" du document :

Que représente le chemin suivant ?

```
prénom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/relpath-ex1.html>]

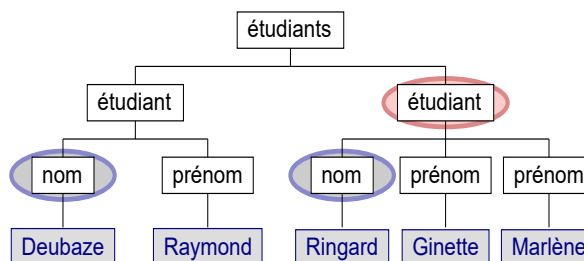


Et celui-ci ?

```
../nom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/relpath-ex2.html>]



6.2.4 Sélection des attributs

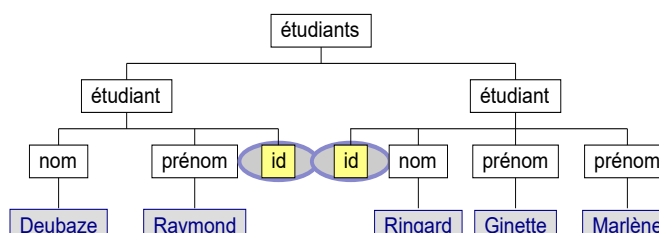
Dans l'arbre **XML**, les **attributs** sont des enfants du noeud correspondant à leur élément parent.

La syntaxe **XPath** pour sélectionner un attribut, consiste à préfixer son nom à l'aide du caractère "@" :

```
//étudiant/@id
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/abbrattr-ex1.html>]



6.3 Sélecteurs, axes et prédicats

6.3.1 Sélecteurs de noeuds

Pour désigner des noeuds dans l'arbre, **XPath** utilise des **sélecteurs de noeuds** (*node tests*).

Les seuls sélecteurs de noeuds utilisés jusqu'ici ont été des noms d'éléments. Toutefois, il y a bien d'autres façons de sélectionner des noeuds.

> élément

C'est le sélecteur vu jusqu'ici. Il permet de désigner les éléments de nom "*élément*".

```
//prénom
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex1.html)

> *

Il s'agit d'un joker qui désigne n'importe quel **élément**.

```
//étudiant[2]/*
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex2.html)

> text()

Cette expression a déjà été rencontrée, et permet de sélectionner les noeuds texte.

```
//étudiant[2]/prénom/text()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex25.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex25.html)

> comment()

Cette fonction permet de sélectionner les commentaires.

```
//étudiant[2]/comment()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex3.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex3.html)

> processing-instruction()

Désigne les instructions de traitement.

```
//étudiant[2]/processing-instruction()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex4.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex4.html)

> processing-instruction('proc')

Désigne les instructions de traitement à destination du processeur "*proc*".

```
//étudiant[2]/processing-instruction('test1')
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex5.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex5.html)

> node()

Sélectionne tous les noeuds, quel que soit leur type.

```
//étudiant[2]/node()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex6.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/node-test-ex6.html)

6.3.2 Prédicats

Les chemins de localisation vus jusqu'à présent désignaient toujours une classe de noeuds simplement désignés par leur nom. Les **prédicats** permettent de sélectionner les éléments qui répondent à une condition donnée, à une étape quelconque du chemin de localisation.

Tentons d'extraire du document ci-dessous le prénom usuel du deuxième étudiant :

```
<étudiants>
  <étudiant id="E111">
    <nom>Deubaze</nom>
    <prénom>Raymond</prénom>
  </étudiant>
  <étudiant id="E314">
    <nom>Ringard</nom>
    <prénom type="usuel">Ginette</prénom>
    <prénom>Marlène</prénom>
  </étudiant>
</étudiants>
```

L'expression suivante désigne l'ensemble de tous les prénoms usuels :

```
//prénom[@type='usuel']
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xpath/predicates-ex1.html>

Toutefois, cela ne répond que partiellement à la question. Le nième élément peut se désigner à l'aide d'une condition réduite à un nombre :

```
//étudiant[2]
```

La valeur du prénom usuel du deuxième étudiant se trouve donc à l'aide de l'expression :

```
//étudiant[2]/prénom[@type='usuel']/text()
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xpath/predicates-ex2.html>

6.3.3 Exemples de prédicats

L'expression qui décrit la condition de sélection des noeuds (*prédicat*) peut être arbitrairement complexe et faire appel à des fonctions **XPath** prédéfinies (*vues plus loin*).

De fait le prédicat réduit à un nombre, qui permet d'extraire le nième noeud d'un ensemble de noeuds, est une version simplifiée de l'expression :

```
element[position()=n] // équivalent à element[n]
```

Comment peut-on sélectionner à l'aide d'un chemin de localisation absolu le prénom de l'étudiant dont le nom est *"Deubaze"* ?

Solution :

<http://dmolinarius.github.io/demofiles/mod-84/xpath/predicates-ex3.html>

Comment sélectionner le nom de tous les étudiants ayant plus d'un prénom ?

Solution :

<http://dmolinarius.github.io/demofiles/mod-84/xpath/predicates-ex4.html>

Le prénom du dernier étudiant ?

Solution :

<http://dmolinarius.github.io/demofiles/mod-84/xpath/predicates-ex5.html>

6.3.4 Etapes de localisation non abrégées

Les chemins de localisation vus jusqu'à présent spécifiaient chacune des étapes en notation abrégée.

Voici la notation abrégée, puis non abrégée de quelques chemins de localisation simples :

```
/étudiants/étudiant/nom
/child::étudiants/child::étudiant/child::nom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/fullstep-ex1.html>]

```
../étudiant[1]/prénom
parent::node()/child::étudiant[1]/child::prénom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/fullstep-ex2.html>]

```
prénom/@type
child::prénom/attribute::type
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/fullstep-ex3.html>]

```
../étudiant[@id='E111']/prénom
parent::node()/child::étudiant[attribute::id='E11']/child::prénom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/fullstep-ex4.html>]

6.3.5 Axes

Le terme qui apparaît devant le caractère "." s'appelle un **axe**. Il sert à indiquer la direction dans laquelle il faut progresser pour l'étape suivante du chemin de localisation.

XPath propose **treize axes** différents dont certains possèdent une notation abrégée, d'autres non.

> Axes possédant une notation abrégée

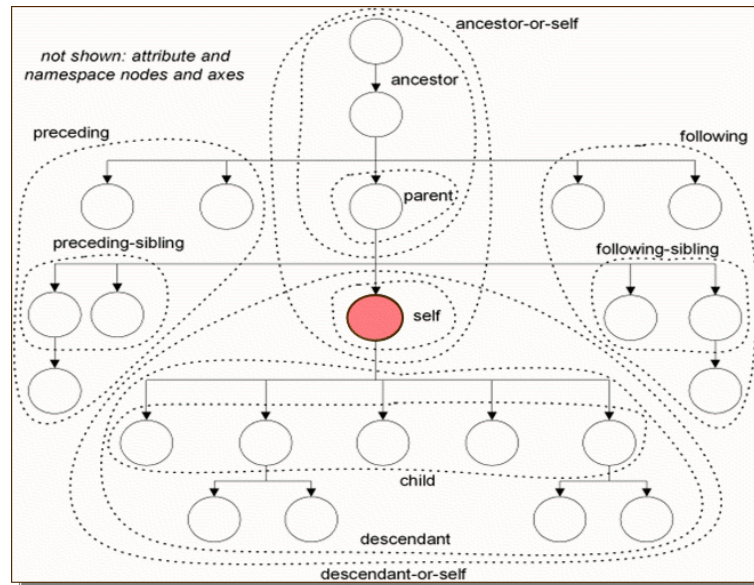
Axe	Ensemble concerné	Notation abrégée
child::	enfants directs du noeud courant	(axe par défaut)
self::	noeud courant	.
parent::	parent du noeud courant	..
attribute::	attributs du noeud courant	@
descendant-or-self::	descendants, noeud courant compris	//

> Autres axes

Axe	Ensemble concerné	Sens de parcours
ancestor::	ascendants du noeud courant	inverse
ancestor-or-self::	ascendants, noeud courant compris	inverse
descendant::	descendants du noeud courant	direct
following::	suivants dans l'ordre du document	direct
following-sibling::	frères suivants, dans l'ordre du doc.	direct
preceding::	précédents dans l'ordre du document	inverse
preceding-sibling::	frères précédents, dans l'ordre du doc.	inverse
namespace::	espaces de noms	-

6.3.6 Récapitulatif des axes

La figure suivante récapitule l'ensemble des axes :



Source : Crane Softwrights

6.3.7 Syntaxe généralisée des étapes de localisation

La syntaxe généralisée des étapes de chemins de localisation comprend un **axe**, un **sélecteur de noeuds** (*node test*) et un prédicat :

axe::sélecteur[prédicat]

Cette syntaxe concerne **chacune des étapes**.

Considérons à titre d'exemple, un document dont l'élément principal s'appelle *"option"* et comporte une liste de personnes :

```
<personnes>
  <personne id="P1" civilité="M.">
    <prénom>Jérôme</prénom>
    <nom>Gourdin</nom>
    <phone>06 54 32 10 98</phone>
    <email>J-Go@freedot.com</email>
    <adresse>6, impasse des fées</adresse>
    <ville>Maubeuge</ville>
  </personne>
  . . .
</personnes>
```

suivie par une liste de cours :

```
<cours intitulé="Technologies de validation" prof="P1">
  <étudiant id="P4">
    <note>12.5</note>
    <absences>1</absences>
    <comment>Devrait se mettre à jour</comment>
  </étudiant>
</cours>
. . .
```

- Comment trouver le prénom de la personne qui porte le nom "*Deubaze*" ?

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/syntax-ex1-option.html>]

- En supposant les personnes classées par ordre alphabétique du nom, quel est le nom de l'avant dernière personne ?

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/syntax-ex3-option.html>]

- Quel est l'intitulé des cours auxquels est inscrit "*Antoine Nette*" ?

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/syntax-ex4-option.html>]

- Comment obtenir la liste des noms des inscrits au premier cours ?

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/syntax-ex5-option.html>]

6.3.8 Chemins composés

Il est possible de désigner une liste d'objets composée à l'aide de plusieurs chemins de localisation. On parle alors de **chemin composé** (*compound location*).

La syntaxe générale est :

```
chemin_1 | chemin_2
```

N.B. Les exemples ci-dessous s'appuient sur le document vu précédemment.

- Comment trouver **le nom et le prénom** de la personne qui porte l'id "*P1*" ?

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/compound-ex2-option.html>]

- On désirerait l'intitulé du second cours, ainsi que le nom et le prénom du professeur.

Réponse :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/compound-ex3-option.html>]

N.B. L'ordre des objets renvoyés par une expression **XPath** est quelconque, non significatif, et notamment indépendant de l'ordre dans lequel sont arrangées les composantes d'une requête composée.

6.4 Expressions générales

6.4.1 Opérateurs

Outre les chemins de localisation qui renvoient un ensemble de noeuds (*node-set*), **Xpath** permet de spécifier des expressions qui renvoient une valeur du type numérique, booléenne ou chaîne de caractères.

Pour les opérations numériques, **Xpath** possède 5 opérateurs arithmétiques :

Op.	Opération	Exemple
+	addition	<code>1 + count(//cours[1]/étudiant)</code>
-	soustraction	<code>count(//personne) - count(//cours[1]/étudiant)</code>
*	multiplication	<code>0.5 * //étudiant[@id='P3']/note</code>
div	division	<code>sum(//cours[1]/étudiant/note) div count(//cours[1]/étudiant)</code>
mod	modulo	<code>count(//cours[1]/étudiant) mod 2</code>

Exemples :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex1-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex1-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex2-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex2-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex3-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex3-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex4-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex4-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex5-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex5-option.html)

Les opérateurs de comparaison renvoient un résultat booléen (*utile pour construire des prédicats*) :

Op.	Test	Exemple
=	égalité	<code>//personne[@id = 'P1']</code>
!=	non-égalité	<code>//cours[@prof != 'P1']/@intitulé</code>
<	inférieur	<code>//personne[@id=//étudiant[note < 10]/@id]</code>
>	supérieur	<code>//personne[@id=//étudiant[absences > 2]/@id]/nom</code>
<=	inférieur ou égal	<code>//personne[@id=//étudiant[absences <= 1]/@id]/nom</code>
>=	supérieur ou égal	<code>//personne[@id=//étudiant[absences >= 2]/@id]/nom</code>

Exemples :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex6-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex6-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex7-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex7-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex8-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex8-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex9-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex9-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex10-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex10-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex11-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex11-option.html)

Les opérateurs logiques sont classiques :

Op.	Opérateur	Exemple
and	ET logique	<code>//personne[@id=//étudiant[absences>=2 and note<11]/@id]/nom</code>
or	OU logique	<code>//personne[@id=//étudiant[absences>2 or note<10]/@id]/nom</code>

Exemples :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex12-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex12-option.html)
[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex13-option.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/operators-ex13-option.html)

6.4.2 Fonctions XPath


Xpath possède également un certain nombre de **fonctions**, utilisables notamment dans les prédicats.

Certaines des fonctions **XPath** (comme *position()* ou *last()*) ont déjà été rencontrées aux détours des exemples précédents :

 **last()** :


[cf. paragraphe 6.5.1.1]

renvoie l'index du dernier élément dans le contexte courant.

 **position()** :

[cf. paragraphe 6.5.1.2]

renvoie l'index de l'élément courant dans le contexte courant.

 **count(node-set)** :

[cf. paragraphe 6.5.1.3]

renvoie le nombre de noeuds de l'ensemble spécifié.

N.B. Les fonctions *last()* et *position()* sont souvent utilisées au sein de prédicats, bien que la seconde puisse également servir par exemple à générer une numérotation automatique.

6.5 Fonctions

6.5.1 Fonctions courantes

6.5.1.1 La fonction last()

> last()

Renvoie l'index du dernier élément dans le contexte courant.

```
last()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/last-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/last-ex1.html)

N.B. Cette fonction est en général utilisée pour la construction de prédicats comme celui-ci :

```
//personne[last()]
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/last-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/last-ex2.html)

qui est la version simplifiée de :

```
//personne[position()=last()]
```

6.5.1.2 La fonction position()

> position()

Renvoie l'index de l'élément courant dans le contexte courant.

```
position()
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/position-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/position-ex1.html)

N.B. Cette fonction est en général utilisée pour la construction de prédicats (*voir aussi la notation abrégée*) :

```
//personne[position()=3] ou plus simplement //personne[3]
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/position-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/position-ex2.html)

6.5.1.3 La fonction count()

> count(node-set)

Renvoie le nombre de noeuds de l'ensemble spécifié.

```
count(//personne)
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/count-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/count-ex1.html)

N.B. Cette fonction est très utile pour numéroter automatiquement les têtes de chapitre, sections, et autres paragraphes : il suffit de compter le nombre d'éléments identiques à l'élément courant qui le précèdent au sein du document...

```
count(preceding-sibling::personne)
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/count-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/count-ex2.html)

6.5.1.4 La fonction lang()

> lang(str)

Renvoie *"true"* si la langue du noeud courant (cf. *xml:lang*) est celle spécifiée par la chaîne passée.

```
<text xml:lang="en">
  This is an example of english text.
</text>
```

```
//text[lang('en')]
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/lang-ex1.html>]

L'avantage de cette fonction par rapport à un test visant à déterminer la valeur de l'attribut *"xml:lang"* est que la langue est héritée depuis les parents vers les enfants, même si ceux-ci ne spécifient pas cet attribut :

```
<exemple xml:lang="fr">
  <text>
    Voici un exemple de texte en français.
  </text>
</exemple>
```

```
//text[lang('fr')]
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/lang-ex2.html>]

Un autre avantage de cette fonction est que le standard des formats de langue permet de spécifier des variantes locales comme *"fr-FR"*, *"fr-BE"* ou *"fr-CA"*. Si une telle variante est présente, alors une requête vers la langue générique retiendra la variante :

```
<text xml:lang="en-UK">
  This is an example of english text.
</text>
```

```
//text[lang('en')]
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/lang-ex4.html>]

6.5.2 Fonctions renvoyant des noms

6.5.2.1 La fonction local-name()

> local-name(node-set)

Renvoie le nom local (i.e. sans le préfixe d'espace de noms) du noeud spécifié ou du premier noeud de l'ensemble.

```
local-name (//*[@id='P1'])
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/local-name-ex1.html>]

N.B. Pour obtenir le nom d'un élément avec son préfixe, voir la fonction *name()*...

6.5.2.2 La fonction name()

> name(node-set)

Renvoie le nom complet (*i.e. avec le préfixe d'espace de noms*) du noeud spécifié ou du premier noeud de l'ensemble.

```
name (//*[ @id='P1' ] )
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/name-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/name-ex1.html)

N.B. Pour obtenir le nom d'un élément **sans** son préfixe, voir la fonction *local-name()*...

6.5.2.3 La fonction namespace-uri()

> namespace-uri(node-set)

Renvoie l'URI d'espace de noms du noeud spécifié ou du premier noeud de l'ensemble.

```
namespace-uri (//*[ @id='P1' ] )
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/namespace-uri-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/namespace-uri-ex1.html)

6.5.3 Fonctions de chaînes

6.5.3.1 La fonction string()

> string(object)

Renvoie la valeur de l'objet sous forme de chaîne de caractères.

```
string (//*[ @id='P1' ] )
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-ex1.html)

N.B. Cette fonction s'applique à toutes sortes d'arguments : élément, noeud texte, attribut, ou ensemble de noeuds (*node-set*). Elle se contente de renvoyer le contenu textuel de l'ensemble des noeuds passés en argument.

Voici un exemple avec un attribut :

```
string (//*[ @id='P6' ]/civilité)
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-ex2.html)

6.5.3.2 La fonction string-length()

> string-length(string)

Renvoie le nombre de caractères de la chaîne "*string*".

```
string-length (//cours[1]/@intitulé)
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-length-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/string-length-ex1.html)

6.5.3.3 La fonction substring()

> substring(str, start, [len])

Renvoie la sous-chaîne de "str" commençant à la position "start," comportant "len" caractères ou jusqu'à la fin de "str".

```
substring(phone, 1, 2)
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/substring-ex1.html>]

N.B. Le dernier argument est optionnel. En son absence la sous-chaîne renvoyée comprend tous les caractères jusqu'à la fin de la chaîne source :

```
substring(@id, 2)
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/substring-ex2.html>]

6.5.3.4 La fonction substring-before()

> substring-before(str1, str2)

Renvoie la sous-chaîne de "str1" située avant la première occurrence de "str2", ou la chaîne vide si "str2" n'est pas contenue dans "str1".

```
substring-before(email, '@')
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/substring-before-ex1.html>]

6.5.3.5 La fonction substring-after()

> substring-after(str1, str2)

Renvoie la sous-chaîne de "str1" située après la première occurrence de "str2", ou la chaîne vide si "str2" n'est pas contenue dans "str1".

```
substring-after(email, '@')
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/substring-after-ex1.html>]

6.5.3.6 La fonction concat()

> concat(str1, str2, ...)

Renvoie la chaîne obtenue en concaténant les arguments passés.

```
concat(substring(prénom, 1, 1), substring(nom, 1, 1), substring(nom, string-length(nom)))
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/concat-ex1.html>]

N.B. L'exemple ci-dessus renvoie le fameux trigramme cher à certains, avec le défaut que la dernière lettre se trouve être en mibuscules ...

6.5.3.7 La fonction `normalize-space()`

> `normalize-space(str)`

Renvoie "*str*" compactée *i.e.* sans espace initial ou final et les espaces internes remplacés par un espace unique.

```
normalize-space(//exemple)
```

Exemple non normalisé :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/normalize-space-ex1.html>]

Exemple normalisé :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/normalize-space-ex2.html>]

6.5.3.8 La fonction `translate()`

> `translate(str1, str2, str3)`

Renvoie "*str1*" dont les caractères présents dans "*str2*" ont été remplacés par ceux de "*str3*".

```
concat (
  substring(prénom,1,1),
  substring(nom,1,1),
  translate(
    substring(nom,string-length(nom)),
    'abcdefghijklmnopqrstuvwxyz',
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  )
)
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/translate-ex1.html>]

N.B. L'exemple ci-dessus renvoie le fameux trigramme cher à certains, en prenant soin de mettre la dernière lettre en majuscules ...

6.5.3.9 La fonction `starts-with()`

> `starts-with(str1, str2)`

Renvoie "*true*" si "*str1*" commence par "*str2*".

```
//personne[starts-with(nom,'D')]/prénom
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/starts-with-ex1.html>]

N.B. Cette fonction est plus particulièrement employée dans le contexte de prédicats.

6.5.3.10 La fonction `contains()`

> `contains(str1, str2)`

Renvoie "*true*" si "*str1*" contient "*str2*".

```
//personne/adresse[contains(., 'place')]
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/contains-ex1.html>]

N.B. Cette fonction est plus particulièrement employée dans le contexte de prédicats.


6.5.4 Fonctions numériques

6.5.4.1 Fonctions XPath

> Fonctions numériques

number(object) :

renvoie la valeur numérique de l'objet (*conversion*).

 *sum(node-set)* :

[cf. paragraphe 6.5.4.2]

renvoie la somme des valeurs numériques de l'ensemble des noeuds passés en argument.

floor(number) :

renvoie le plus grand entier inférieur au nombre passé.

ceiling(number) :

renvoie le plus petit entier supérieur au nombre passé.

round(number) :

renvoie l'entier le plus proche du nombre passé.

6.5.4.2 La fonction sum()

> sum(node-set)

Renvoie la somme des valeurs numériques de l'ensemble des noeuds passés en argument.

```
sum(//cours[1]//note) div count(//cours[1]//note)
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xpath/functions/sum-ex1.html>]

6.5.5 Fonctions booléennes

6.5.5.1 Fonctions XPath

> Fonctions booléennes

boolean(object) :

renvoie la valeur booléenne de l'objet (*conversion*).

not(boolean) :

renvoie la négation de l'argument passé.

true() :

renvoie "true".

false() :

renvoie "false".

7. Transformations XSLT

7.1 Introduction

7.1.1 Spécifications

XSLT (*eXtensible Stylesheet Language Transformations*) est une application **XML** qui spécifie des règles par lesquelles un document **XML** peut être transformé en un autre.

Pour des raisons historiques, un document **XSLT** s'appelle une feuille de style, bien que le terme de "*feuille de transformation*" eusse été plus approprié.

XSLT 1.0 a fait l'objet d'une recommandation du **W3C** datée de novembre 1999.

☞ Consulter la recommandation XSLT 1.0

[<http://www.w3.org/TR/xslt>]

Il existe une seconde version, **XSLT 2.0**, recommandée en janvier 2007 :

☞ Consulter la recommandation XSLT 2.0

[<http://www.w3.org/TR/xslt20/>]

Les implémentations les plus courantes supportent en général **XSLT 1.0**. C'est le cas des navigateurs, de la librairie **MSXML** (*Microsoft*) et des bibliothèques open-source comme **libxslt** (*Gnome*) ou **Xalan** (*Apache*). Pour obtenir une compatibilité **XSLT 2.0** il faut utiliser **Saxon** ou des outils professionnels comme les produits de la suite **Altova** (source Wikipédia [https://en.wikipedia.org/wiki/XSLT#Processor_implementations]).

☞ MSXML - Supported XSLT features

[<https://msdn.microsoft.com/en-us/library/ms764682.aspx>]

☞ Libxslt - Introduction

[<http://xmlsoft.org/XSLT/index.html>]

☞ Xalan - Home Page

[<http://xalan.apache.org/>]

☞ Saxon feature map

[<http://www.saxonica.com/html/products/feature-matrix-9-6.html>]

☞ Altova XML Tools

[https://www.altova.com/xml_tools.html]

Les spécifications **XSLT** sont issues du groupe de travail du **W3C** sur les feuilles de style **XML**. C'est la raison pour laquelle les informations concernant l'état de l'art sur **XSLT** se trouvent sur la page concernant **XSL** (*eXtensible Stylesheet Language*).

☞ Consulter la page principale sur XSL

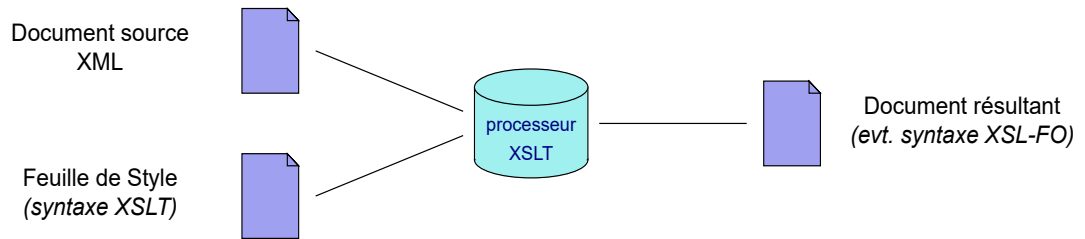
[<http://www.w3.org/Style/XSL/>]

7.1.2 Eléments de contexte

XSLT est issu des besoins identifiés par le groupe de travail en charge de la définition de l'application de feuilles de styles **XSL** (*mi 1998*). Un an et quelques "*Working Drafts*" plus tard, la décomposition fonctionnelle de **XSL** était achevée.

On dispose depuis :

- de **XSLT** pour la transformation du document (*recommandation de novembre 1999*),
- de **Xpath** (*utilisé par XSLT*) pour la localisation d'éléments dans un document (*novembre 1999*),
- et de **XSL-FO** dont la recommandation a été finalisée plus tard, permettant la mise en forme d'un document, en vue notamment de l'impression (*octobre 2001*).



La syntaxe des feuilles de style est une syntaxe **XML**. **XSLT** est l'**application XML** qui correspond à cette syntaxe.

On appelle **moteur XSLT** (*XSLT engine*) un logiciel capable d'interpréter une feuille de style **XSLT** et de produire un document **XML** à partir d'un document source et des instructions rencontrées au sein de la feuille de style.

XSLT est un exemple de plus de mécanisme conçu de manière totalement générique. Il est virtuellement possible de transformer n'importe quel document **XML** source, de manière à produire un document conforme à n'importe quelle autre application **XML**, ou plus généralement n'importe quel document au format texte.

La place de **XSLT** dans une chaîne de traitement du type "*feuille de style*" se justifie vraiment lorsque le format de sortie est un format de présentation comme **XHTML**, **SVG** ou **XSL-FO**.

7.1.3 Transformation et sérialisation

Une feuille de style **XSLT** contient des **modèles** (*templates*).

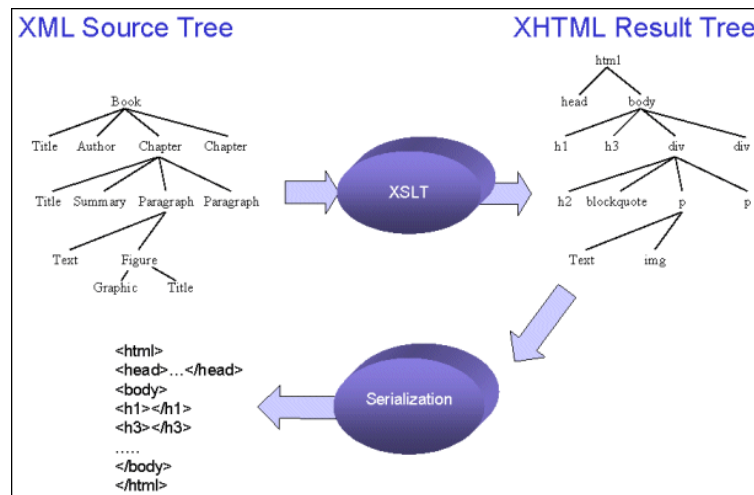
Lors du traitement du document source, un **moteur XSLT** remplace les éléments rencontrés par les modèles correspondants de la feuille de style.

Par exemple, le modèle ci-dessous s'interprète de la façon suivante : pour tout élément "*bloc*" du document source, générer un élément "*p*" dans le document de sortie, dont le contenu sera obtenu en traitant le contenu de l'élément "*bloc*" source :

```

<xsl:template match="bloc">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
  
```

Une fois ce travail effectué, le processeur dispose d'un **arbre XML** en mémoire. Il n'a plus ensuite qu'à sérialiser cet arbre pour obtenir le document de sortie :



Source : <http://www.nwalsh.com/docs/tutorials/xsl/xsl/frames.html>

N.B. En toute rigueur, un moteur **XSLT** conforme aux spécifications n'est pas forcément obligé d'implémenter la sérialisation. Ceci signifie que le *"label de conformité"* peut être délivré à des processeurs qui génèrent un arbre **XML** à *"consommer sur place"* (cf. navigateurs comme IE ou Firefox).

D'un autre côté, un processeur qui implémente la sérialisation doit être capable de sérialiser le résultat aux formats **texte**, **HTML**, ou **XML**.

7.2 Mise en place d'une feuille de style

7.2.1 Associer une feuille de style à un document

L'association d'une feuille de style à un document **XML** s'effectue en général depuis le document lui-même à l'aide d'une **instruction de traitement** qui s'adresse au processeur *"xml-stylesheet"*.

Voici un exemple de document très simple faisant référence à une feuille de style **XSLT** de cette façon (noter le *pseudo-attribut* donnant l'URL de la feuille de style) :

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="first-classe-ex1.xsl" type="text/xsl"?>

<classe>
  <étudiant>
    <nom>Deubaze</nom>
    <prénom>Raymond</prénom>
  </étudiant>
  .
  .
  .
</classe>
```

N.B. Noter ci-dessus le *"mime-type"* officiel d'une feuille de style **XSLT** qui est *"text/xsl"*. C'est grâce à ce *"détail"* qu'un processeur saura comment traiter ce document.

En effet, c'est la même instruction de traitement qui permet d'associer un document **XML** à une feuille de style **CSS**, à la différence près que celle-là serait du type *"text/css"*.

Remarque : Il existe bien d'autres façons d'indiquer à un **moteur XSLT** qu'il doit traiter un document **A** à l'aide d'une feuille de style **B**, ne serait-ce qu'à l'aide de paramètres de ligne de commande...

7.2.2 La feuille de style minimale

L'exemple ci-dessous correspond à la feuille de style **XSLT** la plus simple possible :

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

N.B. **XSLT** est une application dont le principe de fonctionnement est fondamentalement basé sur l'existence des espaces de noms. Noter donc ci-dessus l'**URI d'espace de noms** réservé à **XSLT**. Le préfixe usuel est *"xsl"*.

Bizarrement, l'élément principal peut indifféremment être appelé *"stylesheet"* ou *"transform"*. Ceci s'explique par la genèse d'**XSLT** qui tire ses racines du groupe de travail sur les feuilles de style, mais constitue une application générique de transformation de documents...

Anecdotiquement, la feuille de style vide ci-dessus est tout à fait fonctionnelle (les raisons en seront explicitées plus tard). Voici le résultat lorsqu'elle est appliquée au document vu précédemment :

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/start/first-ex1.html>]

7.2.3 Modèles recursifs

Considérons donc le document source rappelé ci-dessous, constitué d'une liste d'étudiants :

```
<classe>
  <étudiant >
    <prénom>Jean</prénom>
    <nom>Aymard</nom>
  </étudiant>
  . . .
</classe>
```

On désire transformer ce document à l'aide d'une feuille de style **XSLT** de manière à obtenir un document **XHTML** pour afficher la liste des étudiants (*nom et prénom*) dans un navigateur.

> Modèle de l'élément racine

La première étape consiste à créer un modèle pour l'élément racine du document source, qui met en place les contenants fondamentaux du document **XHTML** résultant :

feuille de style

```
<xsl:template match="/classe">
  <html>
    <body>
      <table border="2">
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
```

Ce modèle décrit le traitement à effectuer pour l'élément racine du document source (*N.B. la valeur de l'attribut match est un chemin de localisation XPath*) :

document source

```
<classe> <!-- contenu --> </classe>
```

arbre de sortie

```
<html> <body> <table border="2">
  <!-- contenu obtenu en appliquant
    les modèles correspondant
    au contenu de l'elt "classe" -->
</table> </body> </html>
```

Ce traitement revient à générer les éléments **XHTML** listés dans le modèle puis à poursuivre en appliquant au fur et à mesure les modèles des autres éléments rencontrés.

N.B. Le processeur **XSLT** différencie les éléments qui le concernent (*i.e. ses "instructions"*) de ceux à recopier vers l'arbre de sortie grâce à l'**URI** de l'espace de noms réservé à **XSLT**, associé ici au préfixe "**xsl**"

> Modèles des éléments de contenu

Le modèle ci-dessus a mis en place une "**table**" **HTML** qu'il s'agit maintenant de remplir de lignes.

Le modèle ci-dessous permet de générer une ligne "**tr**" pour chaque "**étudiant**" rencontré dans le document source :

```
<xsl:template match="étudiant">
  <tr>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```

Il n'y a plus ensuite qu'à générer une cellule de tableau "**td**" pour chacun des éléments "**nom**" et "**prénom**" du document source :

```
<xsl:template match="nom | prénom">
  <td>
    <xsl:apply-templates/>
  </td>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/start/recursive-ex1.html>]

7.2.4 Comportement par défaut

Le dernier modèle utilisé amène à réfléchir :

```
<xsl:template match="nom | prénom">
  <td>
    <xsl:apply-templates/>
  </td>
</xsl:template>
```

Il signifie que pour les éléments *"nom"* et *"prénom"* il faut générer un élément *"td"* dans l'arbre de sortie. Le contenu de *"td"* sera obtenu en fonction des modèles qui s'appliqueront au contenu des éléments source *"nom"* et *"prénom"*.

Or, il se trouve qu'il n'y a pas de modèle pour les contenus de ces éléments, qui sont des noeuds texte. De fait, le fonctionnement obtenu repose sur l'existence d'un **modèle implicite** :

```
<xsl:template match="text()">
  <xsl:value-of select="."/>
</xsl:template>
```

Ce modèle revient à transférer vers l'arbre de sortie la valeur de tous les noeuds texte du document source. En l'absence de règle plus précise fournie par la feuille de style, il confère à **XSLT** un comportement par défaut.

7.2.5 Modèles implicites

En fait, **XSLT** possède d'autres règles par défaut que celle qui s'applique aux noeuds texte. Voici l'ensemble des **modèles implicites** :

```
<xsl:template match="*/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*>
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="processing-instruction()|comment()"/>
```

Les modèles implicites reviennent à récursivement traiter les noeuds de tous les éléments, à transmettre la valeur des noeuds texte et des attributs vers le document de sortie, et à ignorer les instructions de traitement et les commentaires.

Le cas des attributs est toutefois particulier en ce sens que l'instruction *"apply-templates"* ne s'applique pas à eux. Pour que le modèle implicite d'un attribut puisse s'appliquer, il faut donc une règle qui le sélectionne explicitement.

Les modèles implicites confèrent un **comportement par défaut**. Les modèles spécifiés dans une feuille de style sont **prioritaires** par rapport aux modèles implicites.

N.B. Le comportement précédemment observé pour une **feuille de style vide** résulte entièrement des modèles implicites.

7.2.6 Validité et bonne forme

Une feuille de style doit être un document **XML bien formé**. Ceci implique en particulier que les *"instructions xsl"* et les éléments qui correspondent au modèle du document de sortie soient correctement emboîtés.

Une feuille de style mêle des éléments hétérogènes d'au moins deux espaces de noms (*l'espace xsl et celui du document de sortie*), dans un ordre et un emboîtement difficilement prévisible.

C'est la raison pour laquelle une feuille de style est très rarement décrite par une **DTD** ou un **schéma**. En général, une feuille de style est donc un document certes bien formé, mais non valide.

N.B. Dans la mesure où la sérialisation est effectuée au format **XML**, la bonne forme (*wellformedness*) du document de sortie est quasiment une conséquence mécanique du fait que la feuille de style elle-même est bien formée ou plus exactement du fait que l'on dispose en mémoire d'un **arbre XML** correct.

Pour être plus précis, le seul point qui ne puisse être garanti est l'unicité de l'élément racine au sein du document généré, qui peut être égal à zéro (*cf. résultat de l'application de la feuille vide à un document XML*), égal à 1, ou correspondre à plusieurs éléments concaténés.

La seule garantie que l'on puisse donc avoir est que le résultat d'une transformation est une **entité externe analysée** bien formée.

7.3 Méthodes de sérialisation

7.3.1 Contrôle de la sérialisation

La **sérialisation** de l'arbre de sortie est contrôlée par l'élément *"xsl:output"*. Cet élément doit être un enfant direct de l'élément racine (*top-level element*).

Les spécifications de **XSLT** préconisent de pouvoir générer un document au format **XML**, **HTML** ou texte :

```
<xsl:output method="text"/>
```

7.3.2 Sérialisation au format texte

La sérialisation au format **texte** consiste tout simplement à parcourir l'**arbre XML** en mémoire en recopiant vers la sortie le contenu des noeuds textuels.

Voici comment demander une sérialisation au format texte :

```
<xsl:output method="text"/>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/output/text-ex1.html>]

N.B. Ce mode de sérialisation peut être utile lorsque le document de sortie est un document texte non-XML.

7.3.3 Sérialisation au format HTML

Ce mode de sérialisation prévoit de pouvoir générer un document **HTML** à partir de l'arbre en mémoire.

```
<xsl:output method="html"/>
```

La syntaxe **HTML** possède un certain nombre de particularités non-XML comme le fait d'autoriser les éléments vides sans balises fermantes, et les attributs minimisés.

N.B. En l'absence d'élément *"output"*, un processeur **XSLT** sérialise par défaut en mode **HTML** si l'élément racine s'appelle *"html"* :

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex1.html)

```
<html>
<head>
  <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <link rel="stylesheet" href="exemple.css" type="text/css">
</head>
. . .
</html>
```

On peut remarquer grâce à l'exemple ci-dessus que le moteur **XSLT** génère automatiquement une balise *"META"* qui confirme le type de contenu, et que le document produit comporte effectivement des éléments vides sans balise fermante (*ce n'est donc pas un document XML bien formé*).

> Jeu de caractères du document de sortie

L'élément *"output"* permet de spécifier au moteur **XSLT** le jeu de caractères à utiliser pour la sérialisation. Ceci ne fonctionne bien sûr que dans la mesure où le processeur employé connaît le jeu de caractères préconisé :

```
<xsl:output method="html" encoding="utf-8"/>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex2.html)

Remarquer la nouvelle balise *"META"* qui indique que le processeur a bien pris en compte le jeu de caractères demandé :

```
<META http-equiv="Content-Type" content="text/html; charset=utf-8">
```

> Contrôle de la version de HTML

La version d'un document **HTML** est indiquée à l'aide d'une balise *"DOCTYPE"* (*similaire à celle des documents XML*).

Une balise *"DOCTYPE"* comprend une **URL** système (*obligatoire*) et un **URI** public (*optionnel*). La valeur de ces informations est encore spécifiée grâce à l'élément *"output"*.

Voici comment indiquer que le document de sortie est au format *"HTML 4.01 Strict"* :

```
<xsl:output
  method="html"
  encoding="utf-8"
  doctype-public="-//W3C//DTD HTML 4.01//EN"
  doctype-system="http://www.w3.org/TR/html4/strict.dtd" />
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex3.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html-ex3.html)

Et voici la balise générée par le processeur **XSLT** :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

> Cas particulier de HTML 5

La déclaration *"DOCTYPE"* de **HTML 5** est particulière, en ce sens qu'elle ne comporte ni **URI** publique, ni **URL** de la **DTD** :

```
<!DOCTYPE html>
```

Ce cas n'a pas été prévu par les concepteurs de **XSLT 1.0**.

La manière standard d'obtenir une déclaration de ce type consiste à la générer sous forme de texte (*ce qui est une hérésie sémantique*) :

```
<xsl:output method="html" encoding="utf-8"/>
<xsl:template match="/">
  <xsl:text disable-output-escaping='yes'>&lt;!DOCTYPE html&gt;</xsl:text>
  <html>
  </html>
</xsl:template>
```

Certains moteurs de transformation permettent néanmoins d'obtenir cette déclaration de manière sémantiquement plus satisfaisante, mais non standard :

```
<xsl:output
  method="html"
  encoding="utf-8"
  doctype-system="about:legacy-compat" />
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/output/html5-ex.html>]

cf. discussion à ce sujet sur stackoverflow

[<http://stackoverflow.com/questions/3387127/set-html5-doctype-with-xslt>]

7.3.4 Sérialisation au format XML

XML est le format de sérialisation le plus naturel, et correspond au mode par défaut en l'absence d'élément *"output"* et lorsque l'élément racine ne s'appelle pas *"html"* :

Illustration avec la feuille de style vide :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/output/xml-ex1.html>]

Exemple au format SVG :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/output/xml-ex2.html>]

certaines processeurs identifient les documents qui les concernent grâce aux indications de la balise *"DOCTYPE"*. Ces informations sont mises en place comme dans le cas du format **HTML** :

```
<xsl:output
  method="xml"
  encoding="utf-8"
  doctype-public="-//W3C//DTD SVG 20010904//EN"
  doctype-system="http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"/>
```

L'exemple ci-dessus génère la déclaration de type correspondant à un document **SVG** :

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/output/xml-ex3.html>]

7.4 Ordre de traitement des éléments

7.4.1 Traitement récursif ou procédural

Les exemples vus jusqu'à présent effectuent un traitement de type **récursif** :

- le modèle de chaque élément demande de poursuivre le traitement avec tous ses enfants,
- l'ordre de traitement des éléments est dépendant de l'ordre des éléments du document source,
- l'ordre des éléments du document produit dépend du document source.

```
<xsl:template match="étudiant">
  <tr>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```

Avec le modèle ci-dessus, par exemple, le prénom précède le nom dans le document de sortie. Cet ordre correspond à celui du document source. Si la syntaxe du document source permet de faire apparaître indifféremment le nom d'un étudiant avant ou après son prénom, alors ceci se retrouvera dans le document produit.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/order/apply-templates-ex1.html>]

A contrario, un traitement **procédural** permet d'imposer l'ordre des éléments dans le document produit, qui dépend alors de la feuille de style.

Pour ceci il suffit d'indiquer à l'instruction "*xsl:apply-templates*" quels sont les éléments à traiter (*l'attribut select admet une expression XPath qui renvoie un ensemble de noeuds*) :

```
<xsl:template match="étudiant">
  <tr>
    <td><xsl:apply-templates select="prénom"/></td>
    <td><xsl:apply-templates select="nom"/></td>
  </tr>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/order/apply-templates-ex2.html>]

Ce modèle s'applique au même document source que le modèle récursif précédent, mais produit un document dans lequel le prénom précède le nom, quel que soit leur ordre d'apparition dans le document source : la structure du document produit ne dépend pas du document source.

Ceci peut être un inconvénient plutôt qu'un avantage. Supposons par exemple que la structure du document source ait été modifiée par ajout de nouveaux éléments. Si le document est traité suivant une approche récursive, les nouveaux éléments pourront apparaître dans le document produit (*grâce aux modèles implicites*) sans modification de la feuille de style. Ils seront par contre ignorés en cas de traitement procédural.

N.B. Sauf intégrisme de l'auteur ou exercice de style pédagogique, il est courant de mêler dans une même feuille de style les approches récursive et procédurale.

7.4.2 Modèle local

Lorsque la structure du document source est maîtrisée, il existe une autre méthode de **traitement procédural** qui consiste à définir un modèle local :

```
<xsl:template match="classe">
  <html><body><table border="1">
    <xsl:for-each select="//étudiant">
      <tr>
        <th><xsl:value-of select="nom"/></th>
        <td><xsl:value-of select="prénom"/></td>
      </tr>
    </xsl:for-each>
  </table></body></html>
</xsl:template>
```

Cet exemple est a priori équivalent au code ci-dessous (*d'où le terme de modèle local*) :

```
<xsl:template match="classe">
  <html><body><table border="1">
    <xsl:apply-templates select="//étudiant">
  </table></body></html>
</xsl:template>

<xsl:template match="étudiant">
  <tr>
    <th><xsl:value-of select="nom"/></th>
    <td><xsl:value-of select="prénom"/></td>
  </tr>
</xsl:template>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/order/for-each-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/order/for-each-ex1.html)

N.B. Comme celui de "*xsl:apply-templates*", l'attribut "*select*" de l'instruction "*xsl:for-each*" admet une expression **XPath** renvoyant un ensemble de noeuds.

7.4.3 Tri

XSLT possède une instruction qui permet de trier l'ensemble de noeuds correspondant au contexte courant. L'instruction "*xsl:sort*" doit apparaître comme enfant direct d'un élément "*xsl:apply-templates*" ou "*xsl:for-each*".

```
<xsl:for-each select="//étudiant">
  <xsl:sort select="nom"/>
  <tr>
    <td><xsl:apply-templates select="id"/></td>
    <td><xsl:apply-templates select="prénom"/></td>
    <td><xsl:apply-templates select="nom"/></td>
  </tr>
</xsl:for-each>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex1.html)

Par défaut, "*xsl:sort*" trie les éléments par ordre alphabétique :

```
<xsl:for-each select="//étudiant">
  <xsl:sort select="id"/>
  ...
</xsl:for-each>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex2.html)

Le tri numérique se met en place de la façon suivante (*la valeur par défaut de l'attribut data-type est text*) :

```
<xsl:for-each select="//étudiant">
  <xsl:sort select="id" data-type="number"/>
  ...
</xsl:for-each>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex3.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex3.html)

... dans un ordre éventuellement inverse (*la valeur par défaut de l'attribut order est ascending*) :

```
<xsl:for-each select="//étudiant">
  <xsl:sort select="id" data-type="number" order="descending"/>
  ...
</xsl:for-each>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex4.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/order/sort-ex4.html)

7.5 Modes et modèles

7.5.1 Construction d'une table des matières

Soit un document **XML** dont l'élément racine est essentiellement composé d'éléments "*paragraphe*" possédant un attribut "*titre*", composés d'éléments "*bloc*" comportant éventuellement un attribut "*titre*" et un contenu texte :

```
<page>
  <titre>XML</titre>
  <sous-titre>From Wikipedia, the free encyclopedia</sous-titre>
  <bloc>In computing ... </bloc>
  . . .
  <paragraphe titre="Applications of XML">
    <bloc>As of 2009...</bloc>
    <bloc titre="Markup and content">The characters making up an XML document...</
  bloc>
  . . .
</paragraphe>
. . .
</page>
```

On désire concevoir une feuille de style produisant un document **XHTML** qui comporte tout d'abord une table des matières avec les titres des paragraphes et des blocs, suivie ensuite par le texte correctement présenté.

7.5.2 Modèles de mise en page

La seconde partie (*mise en page*) ne présente pas de difficulté majeure.

Le modèle de l'élément principal met en place le cadre **HTML** puis demande le traitement de ses descendants dans l'ordre souhaité (*approche procédurale*) :

```
<xsl:template match="page">
  ... entête document HTML ...

  <h1><xsl:value-of select="titre"/></h1>
  <div><em><xsl:value-of select="sous-titre"/></em></div>

  <xsl:apply-templates select="bloc"/>
  <xsl:apply-templates select="paragraphe"/>

  ... fin document HTML ...
</xsl:template>
```

Le modèle pour les paragraphes met en place le titre dans un élément "*h2*", puis poursuit le traitement avec les modèles qui s'appliquent dans le contexte de chacun des paragraphes (*approche récursive*) :

```
<xsl:template match="paragraphe">
  <h2><xsl:value-of select="@titre"/></h2>
  <xsl:apply-templates/>
</xsl:template>
```

Le modèle des blocs procède de même :

```
<xsl:template match="bloc">
  <h3><xsl:value-of select="@titre"/></h3>
  <xsl:apply-templates/>
</xsl:template>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslt/modes/exemple-1.html>

N.B. Contrairement à *"xsl:apply-templates"* qui poursuit récursivement le traitement en appliquant les modèles *ad'hoc*, *"xsl:value-of"* prend la valeur textuelle de l'ensemble de noeuds (*node-set*) renvoyé par l'expression **XPath** correspondant à son attribut *"select"*, ou la valeur de l'expression si celle-ci ne correspond pas à un ensemble de noeuds.

7.5.3 Modèles pour la table des matières

Pour la table des matières, comme pour l'affichage normal, il faut traiter les paragraphes et les blocs à l'aide de modèles, ne retenant cette-fois-ci que les titres.

Afin de différencier ces modèles des précédents, on les distingue en leur affectant un **mode** :

```
<div class="sommaire">
  <xsl:apply-templates mode="sommaire" select="paragraphe"/>
</div>
```

Le modèle de sommaire pour le paragraphe en retient le titre, puis appelle le modèle de sommaire (*cf. mode*) pour les éléments de contenu (*cf. blocs*) :

```
<xsl:template match="paragraphe" mode="sommaire">
<div><xsl:value-of select="@titre"/></div>
<div class="blocs">
  <xsl:apply-templates mode="sommaire" select="bloc"/>
</div>
</xsl:template>
```

Le modèle de sommaire pour les blocs en retient tout simplement le titre :

```
<xsl:template match="bloc" mode="sommaire">
<div><xsl:value-of select="@titre"/></div>
</xsl:template>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslt/modes/exemple-2.html>

7.6 Traitement conditionnel

7.6.1 Condition unique

En examinant bien le résultat des exemples précédents, on s'aperçoit que la feuille de style génère un élément *"h3"* vide pour chacun des blocs ne possédant pas de titre.

```
<body>
<h1>XML</h1>
<div><em>From Wikipedia, the free encyclopedia</em></div>
<h3></h3>
<p>In computing...</p>
. . .
</body>
```

Vérifier :

<http://dmolinarius.github.io/demofiles/mod-84/xslt/condition/exemple-2.html>

Pour éviter ceci, l'élément *"xsl:if"* évalue une expression **XPath** à résultat booléen, et permet de mettre en place une séquence conditionnelle :

```
<xsl:template match="bloc">
  <xsl:if test="@titre">
    <h3><xsl:value-of select="@titre"/></h3>
  </xsl:if>
  <p><xsl:apply-templates/></p>
</xsl:template>
```

Ce même raisonnement s'applique dans le cas des blocs en mode sommaire, pour éviter la génération d'éléments *"div"* vides :

```
<xsl:template match="bloc" mode="sommaire">
  <xsl:if test="@titre">
    <div><xsl:value-of select="@titre"/></div>
  </xsl:if>
</xsl:template>
```

N.B. Dans le contexte de l'attribut *"test"* qui attend une expression **XPath** à valeur booléenne, l'expression *@titre* est équivalente à *not(@titre='')*.

Voir cet exemple en oeuvre :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/condition/exemple-3.html>]

7.6.2 Conditions multiples

N.B. *"xsl:if"* ne possède pas de *"else"*.

Pour remédier à cela, **XSLT** propose un autre élément de traitement conditionnel permettant d'effectuer des choix en fonction d'une série de conditions (à rapprocher de l'instruction *switch* de certains langages de programmation) :

```
<xsl:choose>
  <xsl:when test="condition_1"> ... </xsl:when>
  <xsl:when test="condition_2"> ... </xsl:when>
  <xsl:otherwise> ... </xsl:otherwise>
</xsl:choose>
```

Cet élément a été utilisé dans l'exemple ci-dessous pour modifier les marges des blocs possédant un titre (cf. *modèle des blocs en mode normal*) :

```
<xsl:choose>
  <xsl:when test="@titre">
    <h3><xsl:value-of select="@titre"/></h3>
    <p style="margin-top:0; margin-left: 1.5em;"><xsl:apply-templates/></p>
  </xsl:when>
  <xsl:otherwise>
    <p><xsl:apply-templates/></p>
  </xsl:otherwise>
</xsl:choose>
```

Voir cet exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/condition/exemple-4.html>]

7.7 Numérotation

7.7.1 L'élément XSL number

Dans le cadre du sommaire, la numérotation des paragraphes peut être obtenue à l'aide de l'élément *"xsl:number"* :

```
<xsl:template match="paragraphe" mode="sommaire">
  <div>
    <xsl:number/>&#160;
    <xsl:value-of select="@titre"/>
  </div>
  ...
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/numero/exemple-5.html>]

N.B. Dans son utilisation la plus simple (comme ci-dessus) l'élément "*xsl:number*" renvoie la position de l'élément courant (ici "*paragraphe*") parmi ses frères et soeurs de même nom.

Tentons de numéroter les blocs de la même manière :

```
<xsl:template match="bloc" mode="sommaire">
  <xsl:if test="@titre">
    <div>
      <xsl:number count="paragraphe"/>.<xsl:number/>#160;
      <xsl:value-of select="@titre"/>
    </div>
  </xsl:if>
</xsl:template>
```

On s'appuie ici encore sur l'élément "*xsl:number*" pour compter d'une part les paragraphes, puis les éléments courants (i.e. les blocs), dans l'objectif de générer une numérotation du type 1.1, 1.2, 1.3....

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/numero/exemple-6.html>]

Malheureusement, dans ce cas particulier, cette approche est décevante. En effet, à chaque fois qu'un bloc ne possède pas de titre, la numérotation affichée (qui tient compte du bloc sans titre) n'est pas constituée de nombres consécutifs !

La solution consiste à compter uniquement les blocs titrés...

7.7.2 La fonction XPath count

La numérotation obtenue par "*xsl:number*" est simplement basée sur l'existence d'éléments d'un certain type. Dans des cas plus complexes il est possible de s'appuyer sur la fonction XPath "*count()*" :

```
<xsl:template match="bloc" mode="sommaire">
  <xsl:if test="@titre">
    <div>
      <xsl:number count="paragraphe"/>.
      <xsl:value-of select="count(preceding-sibling::bloc[@titre])+1"/>#160;
      <xsl:value-of select="@titre"/>
    </div>
  </xsl:if>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/numero/exemple-7.html>]

7.8 Modèles nommés, paramètres et constantes

7.8.1 Modèles nommés

Lorsqu'une même séquence d'éléments littéraux (i.e. à recopier vers l'arbre de sortie) revient plusieurs fois dans la feuille de style, on peut la "*mettre en facteur*" dans un **modèle nommé**, puis appeler le modèle à chaque fois que besoin.

Voici un modèle nommé pour générer automatiquement des ancres avec les attributs "*titre*" :

```
<xsl:template name="ancre">
  <span id="anchor-{count(preceding::*[@titre])+count(ancestor::*[@titre])+1}">
    <xsl:value-of select="@titre"/>
  </span>
</xsl:template>
```

Noter comment on a inséré le résultat d'une expression XPath entre accolades "{...}" pour calculer la valeur d'un attribut. L'expression utilisée ici permet de générer des identifiants du type "*anchor-nn*", où la valeur "*nn*" est unique pour chacun des titres.

Il suffit ensuite d'appeler le modèle partout où nous voulons générer un titre, pour les paragraphes :

```
<xsl:template match="paragraphe">
  <h2><xsl:call-template name="ancree"/></h2>
<xsl:apply-templates/>
```

et pour les blocs :

```
<xsl:when test="@titre">
  <h3><xsl:call-template name="ancree"/></h3>
  <p style="margin-top:0; margin-left: 1.5em;"><xsl:apply-templates/></p>
</xsl:when>
```

Pour exploiter ces ancres, il suffit d'insérer des liens autour des titres pour les paragraphes et les blocs en mode sommaire. Voici un second modèle nommé qui réalise cette opération :

```
<xsl:template name="lien">
  <a href="#anchor-{count(preceding::*[@titre])+count(ancestor::*[@titre])+1}">
    <xsl:value-of select="@titre"/>
  </a>
</xsl:template>
```

Noter ici encore l'insertion d'une expression **XPath** entre accolades, dont le résultat nous permet de calculer la valeur de l'attribut *"href"*.

Ce modèle s'appelle comme le précédent, au sein des modèles pour les paragraphes et les blocs en mode sommaire :

```
<xsl:call-template name="lien"/>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/named/exemple-8.html>]

7.8.2 Paramètres

Au-delà du simple appel d'un modèle nommé *"statique"* (*notion de macro*), il est possible de concevoir des modèles **paramétrés**, avec passage d'information lors de l'application du modèle (*notion de sous-programme*) :

```
<xsl:template name="ancree">
  <xsl:param name="nom"/>
  <span id="{ $nom }"><xsl:value-of select="@titre"/></span>
</xsl:template>
```

```
<xsl:template name="lien">
  <xsl:param name="nom"/>
  <a href="#{ $nom }"><xsl:value-of select="@titre"/></a>
</xsl:template>
```

Remarquer comment la valeur du paramètre est exploitée au sein des attributs *"id"* et *"href"*, à partir de son nom précédé d'un caractère *"\$"* (cf. *noms de variables dans certains langages comme PHP*). Les paramètres (*ainsi que les constantes vues plus loin*) peuvent apparaître au sein de n'importe quelle expression **XPath**.

L'appel du modèle nommé, précédemment vide, contient maintenant la valeur du paramètre passé :

```
<xsl:call-template name="ancree">
  <xsl:with-param name="nom"
    select="concat('bloc-', count(preceding::bloc[@titre])+1)"/>
</xsl:call-template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/named/exemple-9.html>]

Le modèle nommé peut également proposer une **valeur par défaut** pour un paramètre, qui sera prise en compte lorsque l'appel ne précise pas de valeur pour ce dernier :

```
<h2><xsl:call-template name="ancre"/></h2>
. . .
<xsl:template name="ancre">
  <xsl:param name="nom"
    select="concat(name(.), '-', count(preceding::*[@titre])+1)"/>
  <span id="{ $nom }"><xsl:value-of select="@titre"/></span>
</xsl:template>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslt/named/exemple-10.html>

7.8.3 Constantes

La définition d'une constante peut se rapprocher d'un modèle nommé non paramétré extrêmement simple :

```
<xsl:variable name="id">
  <xsl:value-of select="concat(name(.), '-', count(preceding::*[@titre])+1)"/>
</xsl:variable>
```

N.B. Bien l'élément s'appelle "*xsl:variable*", une variable **XSLT** se rapproche plus d'une constante que d'une variable telle qu'on l'entend généralement. En effet, une fois définie, la valeur d'une variable ne peut plus être modifiée.

La portée d'une variable est limitée à l'élément dans lequel elle a été définie. Dans ce contexte elle s'utilise au sein de n'importe quelle expression **XPath** en préfixant son nom du caractère "\$" :

```
<span id="{ $id }"><xsl:value-of select="@titre"/></span>
. . .
<a href="#{ $id }"><xsl:value-of select="@titre"/></a>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslt/named/exemple-11.html>

Remarque : la contrainte concernant la portée d'une constante est très restrictive. L'exemple ci-dessous constitue une erreur de débutant classique :

```
<xsl:choose>
  <xsl:when test="condition_1">
    <xsl:variable name="var">valeur 1</xsl:variable>
  </xsl:when>
  <xsl:when test="condition_2">
    <xsl:variable name="var">valeur 2</xsl:variable>
  </xsl:when>
</xsl:choose>
```

En effet, la portée de la variable est limitée à l'élément "*xsl:when*" à l'intérieur duquel elle a été définie, ce qui n'est *a priori* pas l'effet recherché. La bonne approche consiste à écrire :

```
<xsl:variable name="var">
  <xsl:choose>
    <xsl:when test="condition_1">
      <xsl:text>valeur 1</xsl:text>
    </xsl:when>
    <xsl:when test="condition_2">
      <xsl:text>valeur 2</xsl:text>
    </xsl:when>
  </xsl:choose>
</xsl:variable>
```

7.9 Construction de l'arbre de sortie

7.9.1 Insertion d'un élément

Ainsi que les exemples précédents l'on montré, la méthode triviale pour insérer des éléments dans l'arbre de sortie consiste tout simplement à les faire apparaître au sein de la feuille de style dans le corps d'un élément *"xsl:template"* :

```
<xsl:template match="/classe">
  <html>
    <body>
      <table>
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
```

Dans cet exemple, tous les éléments ne faisant pas partie de l'espace de noms associé au préfixe *"xsl"* apparaîtront tels quels dans l'arbre de sortie.

Il existe toutefois des cas particuliers dans lesquels on désire par exemple *"calculer"* le nom de l'élément. Pour cela, XSLT propose l'élément *"xsl:element"*.

A titre d'illustration, la feuille de style ci-dessous convertit le nom de tous les éléments du document source comportant des majuscules, de manière à obtenir des noms uniquement en minuscules :

```
<xsl:template match="*">
  <xsl:element name="{translate(name(.),
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'abcdefghijklmnopqrstuvwxyz')}">
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/element-ex1.html>]

"xsl:element" ouvre d'autres possibilités, comme de transformer des attributs en éléments, ou de générer des éléments en fonctions d'informations variées.

Le modèle ci-dessous génère ainsi un exemple de titre **HTML** dont le niveau (1 à 6) est passé sous forme de paramètre :

```
<xsl:template name="header">
  <xsl:param name="level" select="1"/>
  <xsl:element name="{concat('h', $level)}">
    Ceci est un exemple de titre de niveau
    <xsl:value-of select="$level"/>
  </xsl:element>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/element-ex2.html>]

7.9.2 Insertion d'un attribut

Comme pour les éléments, la méthode évidente pour insérer un attribut dans l'arbre de sortie consiste tout simplement à le faire apparaître de manière littérale :

```
<xsl:template match="/classe">
  <html>
    <body>
      <table border="2">
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
```

Toutefois, on peut là encore vouloir générer des attributs *"calculés"*, ou dont l'existence est liée à une condition.

Dans l'exemple précédent (*transformation des éléments majuscules en minuscules*) les attributs ont disparu corps et biens. Voici comment les faire apparaître :

```
<-- modèle des éléments -->
<xsl:template match="*">
  <xsl:element name="{translate(name(.),
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')}">
    <xsl:apply-templates select="@*" />
  </xsl:element>
</xsl:template>
```

```
<-- modèle des attributs -->
<xsl:template match="@*">
  <xsl:attribute name="{translate(name(.),
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')}">
    <xsl:value-of select="." />
  </xsl:attribute>
</xsl:template>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/attribute-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/attribute-ex1.html)

N.B. L'élément *"xsl:attribute"* est également très utile pour générer des attributs en fonction d'une condition :

```
<xsl:template match="animation">
  <span>
    <xsl:attribute name="style">
      <xsl:choose>
        <xsl:when test="@class='hidden'">visibility: hidden;</xsl:when>
        <xsl:otherwise>display: none;</xsl:otherwise>
      </xsl:choose>
    </xsl:attribute>
  </span>
</xsl:template>
```

... ou tout simplement des attributs dont la valeur est calculée :

```
<xsl:attribute name="OnClick">
  <xsl:text>animationStep('</xsl:text>
  <xsl:call-template name="url-next"/>
  <xsl:text>');</xsl:text>
</xsl:attribute>
```

7.9.3 Insertion d'un noeud texte

Le texte, comme les éléments ou les attributs, est recopié tel quel vers l'arbre de sortie lorsqu'il apparaît de manière littérale dans le contexte adéquat de la feuille de style :

```
<xsl:template match="/">
  <html>
    <head>
      <title>Une feuille de style mono-maniaque</title>
    </head>
    ...
  </html>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/text-ex1.html>]

Toutefois, lorsque cela est nécessaire, il est possible de générer expressément un **noeud texte** :

```
<xsl:attribute name="OnClick">
  <xsl:text>animationStep(')</xsl:text>
  <xsl:call-template name="url-next"/>
  <xsl:text>')</xsl:text>
</xsl:attribute>
```

L'intérêt vient ici de la gestion des espaces. En effet, par défaut, le moteur **XSLT** ignore tous les noeuds texte de la feuille de style qui ne contiennent que des espaces. Ces noeuds ne sont donc pas transférés vers le document généré.

En revanche, le texte effectivement transféré de la feuille de style vers le document de sortie n'est ni normalisé ni compacté.

Dans l'exemple ci-dessous, la valeur de l'attribut **"href"** dans le document généré comprendrait des espaces initiaux et finaux, ce qui n'est évidemment pas l'effet recherché et conduirait à un dysfonctionnement dans le cas où cet attribut servirait par exemple à un hyperlien **"a" HTML** :

```
<xsl:attribute name="href">
  http://www.ec-lyon.fr/
</xsl:attribute>
```

Une solution consisterait à supprimer les espaces indésirables :

```
<xsl:attribute name="href">http://www.ec-lyon.fr/</xsl:attribute>
```

... ou alors à isoler ces espaces dans des noeuds texte **"vides"** de manière à ce qu'ils soient ignorés :

```
<xsl:attribute name="href">
  <xsl:text>http://www.ec-lyon.fr/</xsl:text>
</xsl:attribute>
```

7.9.4 Insertion d'une instruction de traitement

Pour générer une instruction de traitement dans l'arbre de sortie, il faut recourir à l'élément **"xsl:processing-instruction"**. Pour mettre en place une référence à une feuille de style :

```
<?xml-stylesheet href="style.xml" type="text/xsl"?>
```

voici comment procéder :

```
<xsl:processing-instruction name="xml-stylesheet">
  <xsl:text>href="style.xml" type="text/xsl"</xsl:text>
</xsl:processing-instruction>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/pi-ex1.html>]

On peut reprendre une application déjà vue (*conversion des balises majuscules en minuscules*) pour lui faire également traiter les instructions de traitement (*ce qui n'est pas d'une pertinence évidente*) :

```
<xsl:template match="processing-instruction()">
  <xsl:processing-instruction name="{translate(name(),'ABCDEFGHIJKLMNOPQRSTUVWXYZ','abcdefghijklmnopqrstuvwxyz')}">
    <xsl:value-of select="."/>
  </xsl:processing-instruction>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/pi-ex2.html>]

7.9.5 Insertion d'un commentaire

Pour générer un commentaire dans l'arbre de sortie, il faut obligatoirement recourir à l'élément *"xsl:comment"* :

```
<xsl:comment>
  <xsl:text>Ceci est un commentaire</xsl:text>
</xsl:comment>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/comment-ex1.html>]

On peut compléter l'application qui convertit les noms de balises en majuscules pour lui faire traiter les commentaires :

```
<xsl:template match="comment()">
  <xsl:comment>
    <xsl:value-of select="."/>
  </xsl:comment>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/comment-ex2.html>]

7.9.6 Insertion d'une valeur textuelle

De manière générale, le résultat **textuel** d'une expression **XPath** est inséré dans l'arbre de sortie grâce à l'élément *"xsl:value-of"* :

```
<xsl:value-of select="count(preceding-sibling::*[normalize-space(@titre)])+1"/>
<xsl:value-of select="concat(' ',@titre)"/>
```

N.B. Lorsque le résultat de l'expression est un ensemble de noeuds (*node-set*), la valeur renvoyée par *"xsl:value-of"* correspond à la concaténation des valeurs textuelles de chacun des noeuds.

Si l'objectif est de transférer des informations existantes du document source vers l'arbre de sortie, cette méthode doit donc être restreinte aux attributs, ou aux éléments terminaux de l'arbre. Dans l'exemple ci-dessous, si le contenu textuel de l'élément *"exemple"* est bien récupéré, ce n'est pas le cas des balises *"b"* :

document source

```
<exemple>
  Du texte avec deux mots <b>en gras</b> !
</exemple>
```

feuille de style

```
<p> <xsl:value-of select="/exemple"/></p>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/value-of-ex1.html>]

Dans le cas général, il vaut mieux préférer *"xsl:apply-templates"* qui permet le cas échéant d'effectuer le traitement approprié :

feuille de style

```
<xsl:template match="/">
  <p><xsl:apply-templates select="/exemple"/></p>
</xsl:template>
<xsl:template match="b">
  <b><xsl:apply-templates/></b>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/value-of-ex2.html>]

7.9.7 Insertion d'un noeud

Dans l'exemple précédent, la copie du texte comprenant une partie en gras a nécessité un modèle pour le gras dont l'objectif est simplement de recopier le noeud vers l'arbre de sortie :

```
<xsl:template match="b">
  <b>
    <xsl:apply-templates/>
  </b>
</xsl:template>
```

Ce type d'opération peut être réalisé de manière générique, par une instruction qui permet de copier l'élément courant (*en fait, le noeud courant*) vers l'arbre de sortie :

```
<xsl:template match="b|i|tt|code">
  <xsl:copy>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/copy-ex1.html>]

N.B. *"xsl:copy"* ne copie qu'un seul noeud (*i.e. le noeud courant*). Pour copier un noeud et tous ses descendants, il faut utiliser *"xsl:copy-of"* présenté ci-dessous.

7.9.8 Insertion d'un ensemble de noeuds

Contrairement à *"xsl:copy"* qui ne copie qu'un seul noeud vers l'arbre de sortie, *"xsl:copy-of"* recopie récursivement un noeud ou un ensemble de noeuds, avec tout leur contenu (*sous-éléments, attributs, etc...*).

Cette instruction est très utile pour recopier tel quel un fragment du document source vers le document généré.

En supposant par exemple que le document source soit susceptible de contenir des tables présentant exactement la même syntaxe que les tables **HTML** (*ou un sous-ensemble compatible*), celles-ci peuvent être transférées telles quelles de la manière suivante :

```
<xsl:template match="table">
  <xsl:copy-of select="."/>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/copy-of-ex1.html>]

Une autre application de *"xsl:copy-of"* intervient lorsque l'on désire faire apparaître une même séquence à plusieurs endroits différents du document généré (*cf. par exemple entêtes et bas de pages, ...*).

Il suffit dans ce cas de générer la séquence *ad'hoc* dans une variable, puis de recopier récursivement le contenu de la variable vers l'arbre de sortie, à chaque fois que nécessaire.

```
<xsl:variable name="separ">
...
</xsl:variable>
<xsl:template match="bloc[position()>1]">
  <xsl:copy-of select="$separ"/>
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/tree/copy-of-ex2.html>]

7.10 Combinaison de feuilles de style

7.10.1 Inclusion d'éléments de style

XSLT prévoit la possibilité de construire des feuilles de style en assemblant des briques (*constituées de documents XML bien formés*).

La première possibilité consiste simplement à inclure le contenu d'un document externe, dont le contenu sera traité exactement comme s'il était présent à l'endroit considéré :

```
<xsl:include href="extrait.xsl"/>
```

L'exemple ci-dessous montre une feuille de style dont le modèle de l'élément racine est inclus à l'aide de l'instruction "*xsl:include*" :

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslt/include/include-ex1.html>]

7.10.2 Import d'éléments de style

La seconde possibilité offerte par **XSLT** pour combiner des feuilles de style permet d'importer une feuille dont le contenu peut être surchargé par la feuille importatrice.

```
<xsl:import href="feuille_source.xsl"/>
<xsl:template match="/">
  <-- Modèle surchargé -->
  ...
</xsl:template>
```

Exemple :

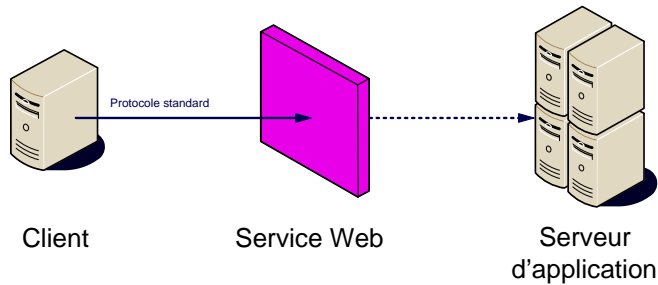
[<http://dmolinarius.github.io/demofiles/mod-84/xslt/include/import-ex1.html>]

8. Services Web

8.1 Qu'est-ce qu'un service Web ?

8.1.1 Principe

Un service web est une interface basée sur des technologies standards de l'internet permettant d'accéder depuis le réseau à des fonctionnalités applicatives.



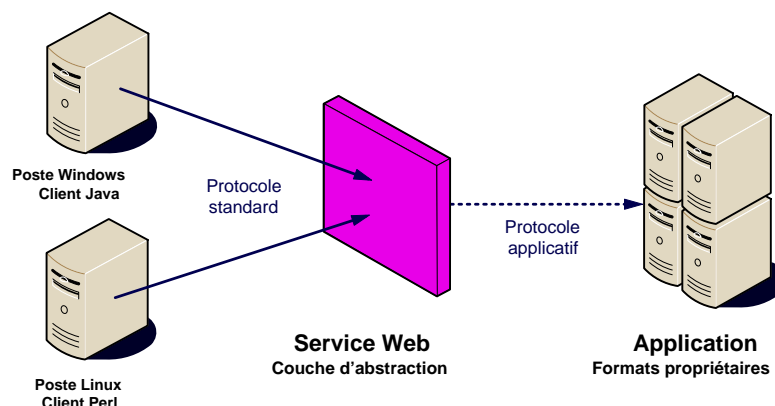
En résumé, dès lors qu'une application est accessible par le réseau à l'aide de protocoles et de formats tels que "*HTTP*", "*XML*", "*SMTP*"... on peut considérer qu'il s'agit d'un service Web.

Vus sous cet angle, les services Web n'apportent rien de particulièrement nouveau par rapport à des mécanismes comme les procédures **cgi** (*Common Gateway Interface*) qui existent depuis l'origine du Web, ou l'échange de documents électroniques (*EDI - Electronic Document Interchange*) tel qu'il est pratiqué par les entreprises depuis de nombreuses années.

Les services Web constituent toutefois la convergence et l'aboutissement de l'expérience accumulée dans les domaines de l'informatique et des réseaux pour l'appel de procédures distantes (*RPC - Remote Procedure Call*), et dans celui de l'industrie par des applications **EDI** basées sur des protocoles comme **CORBA** (*Common Object Request Broker Architecture*).

8.1.2 Interopérabilité

Dans son rôle d'interface, un service web constitue en fait une couche d'abstraction entre l'application et les clients qui accèdent à ses fonctionnalités :



L'interopérabilité est l'un des facteurs clés de succès des services Web.

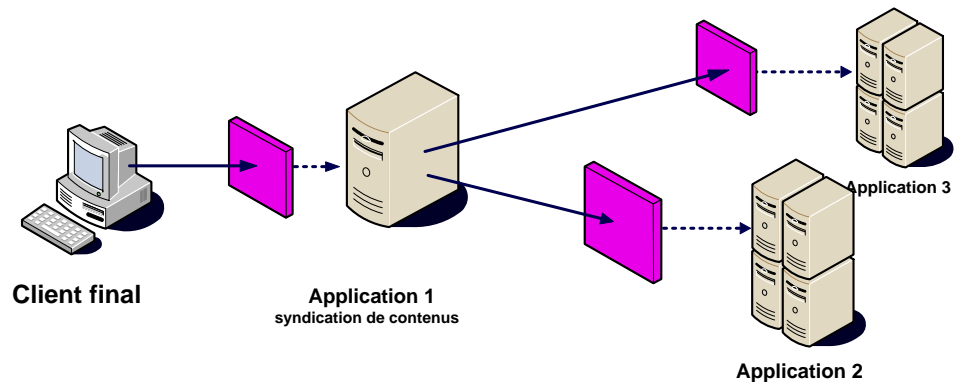
Comme illustré ci-dessus, le protocole standard et la publication des interfaces sont deux conditions qui permettent à des clients variés (*OS, plateformes matérielles, langages, ...*) d'accéder aux fonctionnalités publiques de l'application.

8.1.3 Applications réparties

En toute rigueur, on peut considérer qu'un serveur Web offre un service Web aux navigateurs qui lui adressent des requêtes **HTTP**.

Toutefois, les sites Web délivrent habituellement des informations formatées pour des humains, à la différence des services Web qui renvoient des données plus spécifiquement packagées à destination de procédures informatiques.

La syndication de contenus constitue un exemple classique où une application s'appuie sur des ressources disponibles ailleurs sur le réseau pour construire dynamiquement un ensemble cohérent à destination de l'utilisateur final :



Les services Web permettent ainsi l'existence d'applications réparties faiblement couplées, dont les divers modules peuvent être développés sur des plate-formes hétérogènes par des contributeurs indépendants.

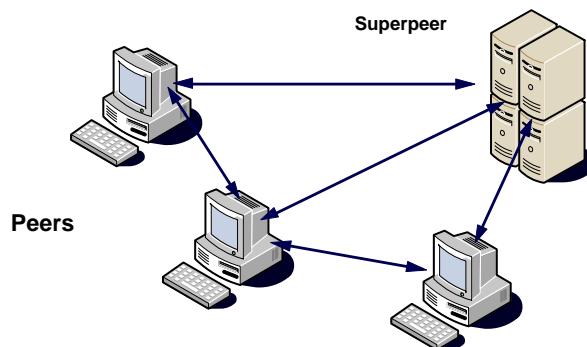
L'interopérabilité est encore une fois assurée par le respect de protocoles standards et la publication des interfaces offerts par chacun des services à ses clients.

8.1.4 Architectures

Les architectures suggérées jusqu'à présent respectaient peu ou prou le modèle client-serveur, à un seul niveau (*modèle classique*) ou multi-niveaux (*modèle n-tier*).

Cependant, la notion de client et de serveur n'est pas forcément statique, et une même machine (*et un même logiciel*) peut se retrouver alternativement dans le rôle de client ou celui de serveur (*architecture peer-to-peer*).

Ce type de fonctionnement est tout à fait compatible avec la notion générique de service Web (*la figure ci-dessous ne représente pas les interfaces*) :

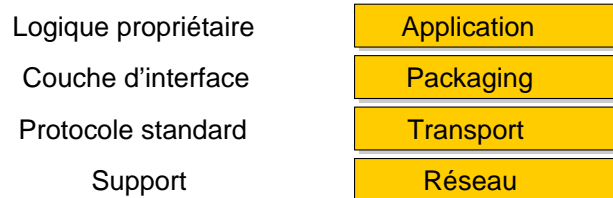


N.B. Ce type d'architecture est fréquent pour des applications comme la messagerie instantanée.

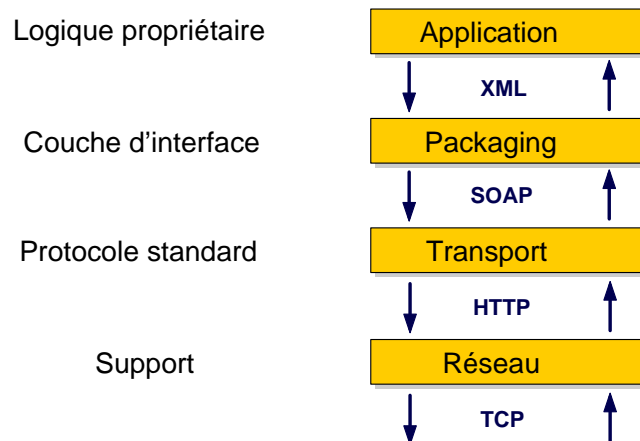
8.2 La pile revisitée

8.2.1 Fonctionnement d'un service Web

Les fonctions constitutives d'un service Web peuvent se décomposer en couches, organisées en pile. Le résultat ressemble aux piles réseau (comme OSI ou IP) :



L'intérêt de cette décomposition, et la clé de l'interopérabilité, sont que la communication entre couches s'effectue à chaque niveau à l'aide de protocoles bien établis, comme illustré ci-dessous.



Cet exemple montre un service Web basé sur l'échange de données **XML**, encapsulées par **SOAP** (*Simple Object Access Protocol*), transportées par **HTTP** sur un réseau **TCP**.

N.B. Ceci n'est bien sûr qu'un exemple. Il existe des formats concurrents pour chacun des niveaux de communication ci-dessus, sur lesquels peut se baser un service Web.

8.2.2 La fonction transport

> Alternatives

En théorie, un service Web peut virtuellement s'appuyer sur n'importe quel protocole de transport fiable.







Les protocoles existants comme **HTTP** (*Hypertext Transfer Protocol*) utilisé par les serveurs Web, ou **SMTP** (*Simple Mail Transfer Protocol*) conçu pour la messagerie, sont bien sûr de bons candidats...

Toutefois, certains services comme **Jabber** définissent leur propre protocole (*ici, XMPP - eXtensible Messaging and Presence Protocol*) qui est alors plus particulièrement adapté aux besoins de ce service (*ici, la messagerie instantanée*). **XMPP** s'appuie lui-même sur **TCP**.

> Critères de choix

Lors de la conception d'un service Web, le choix d'un protocole de transport parmi d'autres s'effectuera en fonction de critères comme la nécessité d'échanges synchrones, l'obligation de gérer des sessions ou la sensibilité aux pare-feux.

> Ressources

-  RFC 793 - Transmission Control Protocol
[\[https://tools.ietf.org/html/rfc793\]](https://tools.ietf.org/html/rfc793)
-  RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1
[\[https://tools.ietf.org/html/rfc2616\]](https://tools.ietf.org/html/rfc2616)
-  RFC 2821 - Simple Mail Transfer Protocol
[\[https://tools.ietf.org/html/rfc2821\]](https://tools.ietf.org/html/rfc2821)
-  RFC 3920 - eXtensible Messaging and Presence Protocol (XMPP): Core
[\[https://tools.ietf.org/html/rfc3920\]](https://tools.ietf.org/html/rfc3920)
-  RFC 3921 - XMPP: Instant Messaging and Presence
[\[https://tools.ietf.org/html/rfc3921\]](https://tools.ietf.org/html/rfc3921)
-  Jabber Software Foundation
[\[https://www.jabber.org/\]](https://www.jabber.org/)

8.2.3 La fonction packaging

> Position du problème

Afin de garantir l'interopérabilité au niveau applicatif, les données échangées par des applications hétérogènes (*OS, plateforme matérielle, langage ...*) doivent être soigneusement préparées et mises sous un format compris par tous.

Les questions qui doivent être résolues ici concernent le type des données échangées, les valeurs minimales et maximales, les codages utilisés, etc...

En langage informaticien on dira qu'il faut définir un format pour la sérialisation des données échangées par l'application.

> Alternatives







A l'heure actuelle (01-2005), **XML** constitue un candidat de choix pour la sérialisation de données.

En effet, comme son nom l'indique **XML** est eXtensible (*modularité, internationalisation...*), il existe énormément d'outils (*transformation, validation, présentation...*), il y a des environnements de développement pour tous les goûts (*commerciaux, publics, open source...*), et surtout, des parseurs **XML** sont disponibles dans tous les langages de programmation.

L'encapsulation de données **XML** aux fins de transport s'effectue alors elle aussi à l'aide d'éléments **XML**, dont la portée est en général cadrée par l'usage d'espaces de noms.

Dans ce contexte on peut considérer **XML-RPC**, plus spécifiquement conçu pour l'appel de procédures distantes (*Remote Procedure Call*), **XMPP** (*eXtensible Messaging and presence Protocol*) dédié au streaming de données **XML**, ou **SOAP** (*Simple Object Access Protocol*) recommandé par le **W3C**, qui s'appuie sur les types de données **XML Schema**.

> Ressources

-  XML-RPC Specification
[\[http://www.xmlrpc.com/spec\]](http://www.xmlrpc.com/spec)
-  RFC 3920 - eXtensible Messaging and Presence Protocol (XMPP): Core
[\[http://localhost/rfc/rfc3920.txt\]](http://localhost/rfc/rfc3920.txt)
-  RFC 3921 - XMPP: Instant Messaging and Presence
[\[http://localhost/rfc/rfc3921.txt\]](http://localhost/rfc/rfc3921.txt)
-  SOAP Version 1.2 Part 1: Messaging Framework
[\[http://www.w3.org/TR/soap12-part1/\]](http://www.w3.org/TR/soap12-part1/)
-  SOAP Version 1.2 Part 2: Adjuncts
[\[http://www.w3.org/TR/soap12-part2/\]](http://www.w3.org/TR/soap12-part2/)
-  XML Schema Part 2: Datatypes Second Edition
[\[http://www.w3.org/TR/xmlschema-2/\]](http://www.w3.org/TR/xmlschema-2/)

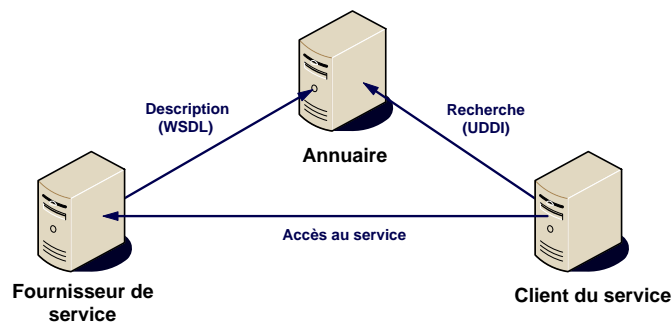
8.3 Environnement complet

8.3.1 Le paysage

Dans le monde du Web (*destiné aux humains*), il est courant de se servir d'un moteur de recherche ou d'un annuaire afin de pouvoir découvrir le site cherché.

D'autre part, afin de faire connaître un site plus rapidement, il est possible (*voire nécessaire*) d'effectuer une démarche afin de le référencer auprès des moteurs de recherche et des annuaires.

Dans le monde des services Web aussi, un service doit être décrit et référencé afin de pouvoir être découvert. Description et découverte sont des opérations qui doivent pouvoir s'effectuer de manière automatique, entre procédures de bonne compagnie :



Il existe des protocoles spécifiques pour chacune de ces étapes. La figure ci-dessus en suggère deux : **WSDL** (*Web Service Description Language*) et **UDDI** (*Universal Description, Discovery, and Integration*), mais il en existe d'autres (*vus plus loin*).

N.B. L'intérêt d'automatiser la découverte d'un service réside bien sûr dans la possibilité d'effectuer cette opération le plus tard possible, juste à temps, au moment où l'on ressent le besoin d'un service particulier (*just in time integration*).

8.3.2 Description d'un service Web

> Position du problème

Pour se référer encore à l'image du Web classique, destiné aux humains, le référencement d'un site nécessite de communiquer à l'annuaire l'**URL** du site, son nom, une liste de mots-clés, du texte décrivant son contenu, etc...

Dans le monde de services Web, où l'objectif est de pouvoir automatiser la découverte d'un service, puis son exploitation, les informations à fournir en plus de son adresse sont plus abstraites : type des données, nom des procédures, protocoles, liste des messages...

> Les solutions

WSDL (*Web Service Description Language*) constitue le standard de fait, pour la description de services Web. **WSDL** est actuellement (01/05) un standard en cours de mise place (*WD - Working Drafts*) auprès du **W3C**.

N.B. **WSDL** est conçu comme un mécanisme générique pour décrire toutes sortes de services Web. Toutefois, certains d'entre eux se prêtent plus facilement à cet exercice, et en particulier les services basés sur **SOAP** (*Simple Object access Protocol*).

Il existe également d'autres standards, moins employés, comme **RDF** (*Resource Description Framework*) ou **DAML** (*DARPA Agent Markup Language*). Ces deux langages sont nettement plus riches et plus puissants que **WSDL**, mais aussi plus complexes à mettre en oeuvre.

> Ressources

 W3C - Web Services Working Group

[\[http://www.w3.org/2002/ws/desc/\]](http://www.w3.org/2002/ws/desc/)

 W3C - XML Protocol Working Group

[\[http://www.w3.org/2000/xp/Group/\]](http://www.w3.org/2000/xp/Group/)

 The DARPA Agent Markup Language Homepage

[\[http://www.daml.org/\]](http://www.daml.org/)

 Resource Description Framework (RDF)

[\[http://www.w3.org/RDF/\]](http://www.w3.org/RDF/)

8.3.3 Recherche d'un service Web

> Position du problème

La description d'un service permet à ses clients potentiels de construire correctement les requêtes qui leur permettront d'en bénéficier. Cette étape est nécessaire mais pas suffisante : il manque toujours un annuaire où ces informations pourront être consultées afin de localiser les services disponibles.

Bien entendu, l'enregistrement d'un service, aussi bien que l'interrogation de l'annuaire devront être entièrement automatiques, et être accessibles sous forme de services Web.

> Les solutions

UDDI (*Universal Description, Discovery and Integration*) est un protocole de service qui répond à ce problème. **UDDI** est développé par le consortium **OASIS**.

L'enregistrement et la consultation constituent deux services Web qui s'appuient sur **SOAP** (*Simple Object Access Protocol*). Ils sont décrits à l'aide de documents **WSDL** (*Web Service Description Language*).

> Ressources

 UDDI Home page

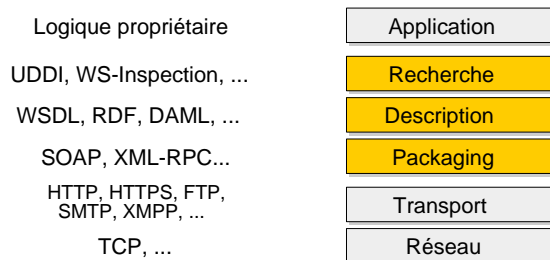
[\[http://uddi.xml.org/\]](http://uddi.xml.org/)

 W3C - Web Services Working Group

[\[http://www.w3.org/2002/ws/desc/\]](http://www.w3.org/2002/ws/desc/)

8.3.4 Retour sur la pile

La plupart des documents et auteurs chargent la pile des services Web avec l'étape de description et les mécanismes de publication et de recherche/découverte :



Cette vision est cohérente dans la mesure où chacune des couches sert d'abstraction à la suivante. La démarche associée consiste donc clairement à isoler les applications (*et leurs programmeurs*) de la mécanique **XML** sous-jacente.

> Remarques

Cette pile vient en général se compléter d'éléments concernant la sécurité, le routage, la gestion de transactions...

Il n'y a pas à l'heure actuelle de consensus sur l'ordre des opérations, les formats et protocoles ou l'intégration des mécanismes entre eux.

Les divers organismes et éditeurs en présence sur le marché des services Web (*Microsoft, IBM, W3C, ...*) proposent chacun leur vision et leur approche qui, si elles sont à peu près compatibles sur les fondamentaux (*HTTP, SOAP, WSDL*), ne le sont pas encore pour les éléments plus avancés.

 Web Services Architecture

[\[http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/\]](http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/)

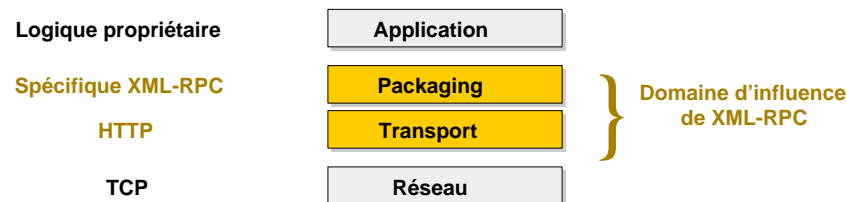
9. XML-RPC

9.1 Introduction

9.1.1 Principe

XML RPC (*Remote Procedure Call*) est un protocole de service Web pour l'exécution de procédures distantes sur plates-formes hétérogènes via l'Internet.

XML-RPC s'appuie sur **HTTP** (*Hypertext Transfer Protocol*) pour le transport et sur une syntaxe **XML** spécifique pour le packaging :



XML-RPC a été conçu au départ pour être d'une utilisation aussi simple que possible, de manière à favoriser sa dissémination en facilitant l'écriture de clients et de serveurs.

De ce fait il existe de nombreuses implémentations de **XML-RPC**, depuis la bibliothèque quasi-artisanale, aux modules officiels associés à des langages comme **php**, **Perl**, **Java**, **python**, **.NET**, ou même directement intégrées à des serveurs comme **Apache**.

9.1.2 Fiche d'identité

> Références

 Site de référence : www.xmlrpc.com
[<http://www.xmlrpc.com/>]

Les spécifications de **XML-RPC** ont été initialement développées en 1999 par [Dave Winer](#) de la société **Userland Software**, puis subi des modifications mineures dont la dernière date de juin 2003.

UserLand Software est un éditeur de logiciels pour la publication de sites Web et de [weblogs](#). Leur produit-phare est **Manila**.

 UserLand Software
[<http://www.userland.com/>]


 Manila
[<http://manila.userland.com>]

> Déclaration de type de document

Officiellement **XML-RPC** ne dispose pas d'un **URI** public, ni d'une **DTD**, et encore moins d'un schéma. Un document **XML-RPC** ne comporte donc généralement pas de déclaration "**DOCTYPE**" et ne peut être directement validé par les outils **XML** existants (*ce qui n'empêche pas de contrôler sa cohérence au niveau applicatif*).

Il existe toutefois des **DTD** et des schémas (*parfois partiels*) mis à disposition par la communauté.

 Exemple de DTD pour XML-RPC
[<http://code.haskell.org/haxr/xml-rpc.dtd>]

 Page de cours sur XML-RPC, avec DTD et schéma
[<http://www.cafeconleche.org/books/xmljava/chapters/ch02s05.html>]

> Espace de noms

XML-RPC ne gère pas les espaces de noms **XML**. Un document **XML-RPC** ne peut donc pas mêler des éléments provenant d'applications différentes, et tous les éléments d'un tel document doivent appartenir à l'espace de noms par défaut.

> Remarques

L'absence de DTD, de schéma, ou d'espace de noms ne sont pas les seules limitations de **XML-RPC**, dont les spécifications ne sont pas issues d'un organisme comme le **W3C** ou **OASIS** par exemple, mais ont été développées par une personne seule, sans vrai suivi ultérieur pour en améliorer la robustesse.

Nous pointerons par la suite quelques erreurs de conception de **XML-RPC**, non dans un but critique (*car XML-RPC est parfaitement utilisable - et utilisé - tel quel*), mais dans un but pédagogique afin de souligner la nécessité d'un travail de groupe en comité d'experts pour développer des spécifications robustes, interopérables et adaptées à leur écosystème.

9.1.3 Exemple d'échange

XML-RPC s'appuie sur le protocole **HTTP** pour l'échange de documents **XML**, dont le contenu décrit la requête et sa réponse.

Voici un exemple de requête, qui s'adresse à un service renvoyant le nom d'un département français lorsqu'on l'interroge en fournissant le numéro :

> Requête

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
```

```
<?xml version="1.0"?>
<methodCall>
  <methodName>NomDepartement</methodName>
  <params>
    <param>
      <value>69</value>
    </param>
  </params>
</methodCall>
```

> Réponse

```
HTTP/1.1 200 OK
Date: Tue, 14 Sep 2004 21:34:38 GMT
Server: Apache/1.3.28
X-Powered-By: PHP/4.3.3
Connection: close
Content-Length: 170
Content-Type: text/xml
```

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <string>Rh&#xF4;ne</string>
      </value>
    </param>
  </params>
</methodResponse>
```

9.2 Format de la requête

9.2.1 Entêtes HTTP

Voici l'entête **HTTP** d'une requête **XML-RPC** typique :

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
```

> Requête HTTP

```
POST /~muller/cours/xml/appl/xmlrpc-server.php HTTP/1.0
```

Une requête **XML-RPC** s'effectue obligatoirement avec la méthode **POST**.

L'**URI** de la requête est non significative pour **XML-RPC**. En ce qui concerne le service Web, cette **URI** pourrait aussi bien être vide et prendre la valeur **"/"**.

Toutefois, lorsque le serveur n'est pas uniquement un serveur **XML-RPC** et qu'il traite donc également d'autres requêtes **HTTP**, il est toléré que l'**URI** de la requête soit non vide, de manière à permettre au serveur de localiser précisément le service demandé (cf. exemple ci-dessus).

N.B. La version **HTTP** de la requête est obligatoirement **HTTP/1.0** (cf. justification dans la forme de la réponse).

> Directives HTTP

```
Host: tic01.tic.ec-lyon.fr
User-Agent: xmlrpc-client.php
Content-Type: text/xml
Content-Length: 153
Connection: Close
```

Les directives **Host** (nom du serveur virtuel), **User-Agent** (identification du client), **Content-Type** (type du corps de la requête) et **Content-Length** (taille du corps en octets) sont obligatoires.

La directive **Content-Type** prend obligatoirement la valeur **"text/xml"**.

La directive **Content-Length** est obligatoire, la valeur associée est obligatoirement correcte et doit correspondre au nombre d'octets du corps de la requête.

XML-RPC ne précise pas explicitement si d'autres directives (comme ici *Connection: close*) sont autorisées ou non. Toutefois, l'usage veut que de telles directives soient acceptées par les serveurs (ce qui va bien dans le sens de la philosophie **HTTP**).

N.B. Certaines implémentations utilisent cette possibilité pour compléter **XML-RPC** avec l'un des mécanismes d'authentification reconnus par **HTTP** (*Basic* ou *Digest*), et pour gérer des sessions à l'aide de **"cookies"**.

9.2.2 Corps de la requête

Le corps de la requête **XML-RPC** contient un document **XML** bien formé. L'élément racine doit être **"methodCall"** :

```
<?xml version="1.0"?>
<methodCall>
  <methodName>NomDepartement</methodName>
  ...
</methodCall>
```

Les spécifications ne précisent pas si la **déclaration XML** (1ère ligne du document) est obligatoire ou non, ni quels sont les codages de caractères autorisés.

Pour une interopérabilité maximale, il est donc conseillé pour un client de faire figurer la déclaration **XML** et de n'employer que les codages **UTF-8** ou **UTF-16**.

Un serveur par contre, pourra autoriser un document sans déclaration **XML** (*non obligatoire en XML*) et accepter tous les codages qui lui conviennent.

> Procédure

L'élément *"methodName"* doit contenir un élément *"methodName"* dont le contenu est une chaîne de caractères qui donne le nom de la procédure appelée.

Le nom de la procédure doit être composé uniquement de caractères alphanumériques parmi [a-z A-Z 0-9] et des 4 caractères de ponctuation [_ . : /]

Il appartient entièrement au service considéré d'interpréter le nom de la procédure, qui peut effectivement correspondre à un sous-programme qu'il s'agit d'appeler (*en Java, Perl, Python, php...*) ou bien à tout autre chose comme le nom d'une table dans une base de données ou bien le nom d'un fichier à analyser.

> Paramètres

Si la procédure appelée comporte des paramètres, alors l'élément *"methodName"* doit être suivi d'un unique élément *"params"*, qui contient autant d'éléments *"param"* qu'il y a de paramètres.

```
<methodCall>
  <methodName>NomDepartement</methodName>
  <params>
    <param>...</param>
    ...
  </params>
</methodCall>
```

Chacun des éléments *"param"* contient à son tour un unique élément *"value"* dont le contenu dépend du type du paramètre considéré :

```
<param>
  <value>Raymond Deubaze</value>
</param>
```

N.B. XML-RPC reconnaît divers types de données, scalaires, structures ou tableaux (*array*), qui seront abordés dans la suite du cours.

9.2.3 Paramètres

D'après les spécifications, on peut provisoirement résumer la syntaxe d'une requête **XML-RPC** à l'aide de l'extrait de **DTD** suivant :

```
<!ELEMENT methodCall (methodName, params?)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params (param+)>
<!ELEMENT param (value)>
```

Ces quelques règles font clairement apparaître que, conformément à une interprétation raisonnable des spécifications, l'élément *"params"* peut être omis (*procédure sans paramètres*), et que s'il existe, il contient au moins un élément *"param"*.

Exemple d'appel de procédure sans paramètres :

```
<methodCall>
  <methodName>ListeDepartements</methodName>
</methodCall>
```

> Problème d'interopérabilité

Certains clients procèdent de la manière qui vient d'être décrite.

Il se trouve malheureusement que certaines implémentations de serveurs ont interprété les spécifications de manière différente. Selon ces implémentations, l'élément *"params"* serait obligatoire, quitte à être vide :

```
<methodCall>
  <methodName>ListeDepartements</methodName>
  <params/>
</methodCall>
```

Lorsqu'un client envoie une requête sans *"params"* à un serveur qui considère que cet élément est obligatoire, nous sommes en présence d'un problème d'interopérabilité.

> Solution au problème

Il se trouve qu'un élément *"params"* vide ne semble poser problème à aucun serveur connu.

En cas d'absence de paramètres, l'interopérabilité maximale est alors obtenue pour un client en envoyant un élément *"params"* vide, et pour un serveur en acceptant indifféremment les deux constructions (*params absent ou vide*).

Cette démarche semble aujourd'hui avoir été adoptée par la plupart des implémentations (*python, php, ...*). Elle est résumée par les deux DTD ci-dessous :

Client :

```
<!ELEMENT methodCall (methodName, params)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params (param*)>
<!ELEMENT param (value)>
```

Serveur :

```
<!ELEMENT methodCall (methodName, params?)>
<!ELEMENT methodName (#PCDATA)>
<!ELEMENT params (param*)>
<!ELEMENT param (value)>
```

N.B. La DTD couramment trouvée sur Internet est la version *"Client"* ci-dessus. Implémentée côté serveur elle conduit au problème qui vient d'être évoqué.

9.2.4 Schéma d'une requête

Construisons un **schéma XML** qui traduit la syntaxe d'une requête **XML-RPC**.

L'élément *"methodCall"* contient un élément *"methodName"* suivi par un élément *"params"* optionnel :

```
<xs:element name="methodCall">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="methodName" type="T_NAME"/>
      <xs:element name="params" minOccurs="0" type="T_PARAMS"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

L'élément *"methodName"* contient une chaîne de caractères non vide, dont les caractères autorisés sont explicitement cités :

```
<xs:simpleType name="T_NAME">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Za-z0-9/._:~]+"/>
  </xs:restriction>
</xs:simpleType>
```

N.B. La chaîne est non normalisée (le type hérite de `"xs:string"`), ce qui signifie que les espaces sont interdits, y compris en tête et en fin de chaîne. Ceci signifie que la requête ci-dessous est incorrecte :

```
<methodCall>
  <methodName>
    getInfo
  </methodName>
</methodCall>
```

L'élément `"params"` contient une liste éventuellement vide d'éléments `"param"` :

```
<xs:complexType name="T_PARAMS">
  <xs:sequence>
    <xs:element name="param" minOccurs="0" maxOccurs="unbounded" type="T_PARAM">
    </xs:sequence>
  </xs:complexType>
```

L'élément `"param"` contient un unique élément `"value"` :

```
<xs:complexType name="T_PARAM">
  <xs:sequence>
    <xs:element name="value" type="T_VALUE"/>
  </xs:sequence>
</xs:complexType>
```

Le contenu de l'élément `"value"` sera décrit par ailleurs.

 Voir l'extrait de schéma complet

<http://dmolinarius.github.io/demofiles/mod-84/xrpc/request/request.xsd.html>

9.3 Valeurs scalaires

9.3.1 Nombres entiers

Les nombres entiers **XML-RPC** sont représentés par un élément `"int"` ou `"i4"` (notations équivalentes), dont le contenu est un entier signé codé sur 32 bits.

Vue la définition ci-dessus, les valeurs autorisées vont par conséquent de -2147483648 à +2147483647 inclus.

Exemple de valeurs entières :

```
<value><int>69</int></value>
<value><i4>2005</i4></value>
```

> Schéma XML

L'ensemble des valeurs autorisées pour les entiers **XML-RPC**, correspond à l'espace des valeurs du type prédéfini `"xs:int"`. Toutefois, `xs:int` est normalisé alors que les spécifications **XML-RPC** interdisent les espaces de tête et de fin.

En toute rigueur, si l'on veut absolument représenter exactement l'espace lexical autorisé pour les entiers **XML-RPC**, il faut définir (non sans malice) un type dérivé de `"xs:string"`. Cette opération sera détaillée à titre pédagogique dans le prochain paragraphe.

9.3.2 Vous avez dit normalisé ?

Les spécifications de **XML-RPC** précisent explicitement (commentaires du 21/1/99) que les espaces de tête et de fin sont interdits dans l'expression des valeurs scalaires (comme `int` par exemple).

Le schéma XML qui permet de représenter les entiers n'est pas simple à imaginer.

> Dérivation à partir de `xs:int`

L'espace des valeurs des entiers **XML-RPC** correspond au type prédéfini `"xs:int"`. Toutefois, ce type est normalisé, c'est à dire qu'il accepte les espaces de tête et de fin.

Afin de restreindre l'espace lexical de *"xs:int"*, on pourrait être tenté d'interdire les espaces grâce à la facette *"xs:pattern"* (qui agit bien sur l'espace lexical) :

```
<xs:simpleType>
  <xs:restriction base="xs:int">
    <xs:pattern value="\S+"/> <!-- limité aux non-espaces -->
  </xs:restriction>
</xs:simpleType>
```

Toutefois, cette méthode ne fonctionne pas, car la facette *"xs:pattern"* travaille sur l'espace lexical après normalisation.

> Dérivation à partir de xs:string

La seule solution consiste à dériver depuis *"xs:string"* (non normalisé), en n'autorisant que les expressions qui correspondent à des entiers :

```
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[+\-]?[0-9]+"

```

Malheureusement, cette méthode n'est pas satisfaisante non plus, car elle ne limite pas l'espace des valeurs pour correspondre exactement à celui des entiers 32 bits.

> Solution

La seule solution vraiment satisfaisante est assez laborieuse, puisqu'elle consiste effectivement à construire le type désiré à partir de *"xs:string"*, mais en listant l'ensemble des combinaisons autorisées.

La technique retenue ici revient à autoriser d'abord tous les nombres à 9 chiffres significatifs (les spécifications précisent que les zéros initiaux sont ignorés), puis à lister au fur et à mesure toutes les combinaisons autorisées de nombres à 10 chiffres.

Cette approche est appliquée au sein du modèle suivant dont l'espace lexical correspond effectivement à celui des entiers **XML-RPC** (les sauts de ligne ont été ajoutés uniquement pour faciliter la lecture) :

```
<xs:simpleType name="T_INT">
  <xs:restriction base="xs:string">
    <xs:pattern value="[+\-]? (0+|0* ([1-9] [0-9] {0,8}|1[0-9] {9}|2 (0[0-9] {8}|
      1 ([0-3] [0-9] {7}|4 ([0-6] [0-9] {6}|7 ([0-3] [0-9] {5}|4 ([0-7] [0-9] {4}|8 ([0-2] [0-9] {3}|3 ([0-5] [0-9] {2}|6 ([0-3]
      [0-9] |4 [0-7] ))) ) ) ) ) ) ) ) | -0*2147483648" />
  </xs:restriction>
</xs:simpleType>
```

 Voir un extrait de schéma détaillant le raisonnement employé

<http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/int.xsd.html>

9.3.3 Booléens

Les booléens **XML-RPC** sont représentés par l'élément *"boolean"*.

Les seules valeurs autorisées sont *"0"* et *"1"*.

Exemple de booléens :

```
<value><boolean>0</boolean></value>
<value><boolean>1</boolean></value>
```

> Schéma XML

Comme les spécifications interdisent les espaces de tête et de fin, il faut, là encore, recourir au type *"xs:string"* :


```
<xs:simpleType name="T_BOOLEAN">
  <xs:restriction base="xs:string">
    <xs:pattern value="0|1"/>
  </xs:restriction>
</xs:simpleType>
```

9.3.4 Nombres à virgule flottante

XML-RPC représente les nombres à virgule flottante par un élément *"double"*, dont le contenu est l'expression lexicale d'un nombre *"signé double précision"*.

Les spécifications (*commentaires du 21/1/99*) précisent toutefois que :

- il n'est pas possible de représenter *"NaN"* (*Not a Number*), ni les infinis positif ou négatif,
- la seule représentation lexicale autorisée prend la forme d'un nombre décimal (*pas d'exposant autorisé*),
- le nombre de chiffres est illimité (!),
- les valeurs maximales et minimales admises sont *"implementation dependant"*.

Exemple de valeurs à virgule flottante :

```
<value><double>3.141592</double></value>
<value><double>-273.29</double></value>
```

> Schéma XML

L'expression *"signé double précision"* tendrait de prime abord à indiquer un espace des valeurs correspondant à celui du type prédéfini *"xs:double"*. Pourtant, les commentaires indiquent qu'il s'agit plutôt de *"xs:decimal"*.

Cependant, les espaces étant toujours interdits, il faut encore une fois se rabattre sur *"xs:string"* :

```
<xs:simpleType name="T_DOUBLE">
  <xs:restriction base="xs:string">
    <xs:pattern value="[+\-]?[0-9]+\.[0-9]+" />
  </xs:restriction>
</xs:simpleType>
```

9.3.5 Dates et heure

Les spécifications **XML-RPC** prévoient également des valeurs de type date, dont l'élément nommé *"dateTime.iso8601"* fait référence au standard **ISO 8601**.

Le standard **ISO 8601** est plutôt complexe, et prévoit de nombreux formats de date, d'heure, et d'intervalles de temps.

Toutefois, et malgré l'abîme de flou laissé par les spécifications (*aucun commentaire, 1 exemple !*), les usages en cours au sein de la communauté tendent à indiquer que le seul format reconnu est **CCYYMMDDTHH:MM:SS**.

Ce format comporte 4 digits pour l'année, immédiatement suivis par le mois sur 2 digits, le jour sur 2 digits, la caratère *"T"* obligatoire, puis l'heure, les minutes et les secondes, chaque fois sur deux digits, séparées par des caractères *":"*.

Exemple de date :

```
<value><dateTime.iso8601>20050115T20:18:17</dateTime.iso8601></value>
```

La valeur ci-dessus signifie *"15 janvier 2005, 20 heures 18 minutes et 17 secondes"*.

Aucun autre format ne semble devoir être autorisé.

> Schéma XML

Bien que le seul format possible soit assez rigide, le schéma n'est encore une fois pas simple à définir.

En effet, et bien que **XML Schéma** dispose de types prédéfinis relatifs aux dates **ISO 8601**, l'interdiction des espaces de tête et de fin nécessite à nouveau le recours à *"xs:string"*, ce qui impose de contrôler manuellement le nombre de jours de chacun des mois, ainsi que les années bissextiles.

Une fois encore, le raisonnement consiste à juxtaposer toutes les solutions autorisées...

☞ Voir un extrait de schéma détaillant le raisonnement employé

[\[http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/date.xsd.html\]](http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/date.xsd.html)

Cette approche est appliquée au sein du modèle suivant (*les sauts de ligne ont été ajoutés uniquement pour faciliter la lecture*) :

```
<xs:simpleType name="T_DATE">
  <xs:restriction base="xs:string">
    <xs:pattern value="( [0-9]{4} ( (0(1|3|5|7|8)|1(0|2)) (0[1-9]|[1-2][0-9]|
      3[0-1])|(0(4|6|9)|11) (0[1-9]|[1-2][0-9]|30)|02(0[1-9]
      |1[0-9]|2[0-8]))|(( [0-9]{2} (0(4|8)|(1|3|5|7|9)(2|4|6)
      |(2|4|6|8)(0|4|8))|[0-9]000)0229))
      T((0(1)[0-9])|(2[0-3])):[0-5][0-9]:[0-5][0-9]" />
  </xs:restriction>
</xs:simpleType>
```

9.3.6 Chaînes de caractères

Les chaînes de caractère **XML-RPC** sont représentées par l'élément *"string"*.

Tous les caractères des chaînes **XML-RPC** sont significatifs (*espaces compris*). Conformément aux spécifications de **XML**, les caractères "<" et "&" doivent évidemment être remplacés par les appels d'entités "<" et "&".

La première version des spécifications **XML-RPC** restreignait les chaînes aux caractères **ASCII**, mais cet aspect a été modifié par la suite en autorisant tous les caractères (?)

La liste exacte des caractères autorisés par les spécifications **XML-RPC** a donné lieu à de nombreuses spéculations sur Internet. Il en découle qu'un serveur peut toujours accepter des caractères autres que **US-ASCII**, mais qu'un client ne pourra en profiter que s'il en est averti (*bien entendu, il n'existe aucun mécanisme de négociation des possibilités du serveur*).

Exemple de valeur chaîne :

```
<value>
  <string>
    Any technology distinguishable from magic is insufficiently advanced
  </string>
</value>
```

Lorsqu'un élément *"value"* contient directement du texte (*sans indication de type*), alors **XML-RPC** considère qu'il s'agit d'une chaîne de caractères :

```
<value>
  This is magic !
</value>
```

Afin d'optimiser l'interopérabilité il est conseillé d'accepter cette dernière syntaxe côté serveur, mais de ne pas l'employer côté client...

> Schéma XML

En toute rigueur les chaînes **XML-RPC** correspondent au type prédéfini *"xs:string"* limité aux caractères **ASCII** plus les espaces :

```
<xs:simpleType name="T_STRING">
  <xs:restriction base="xs:string">
    <xs:pattern value="(&#x9;|&#xD;|&#xA;| [&#x20;-&#x7F;])*" />
  </xs:restriction>
</xs:simpleType>
```

D'un autre côté, il est impossible d'exprimer dans un **schéma XML** la possibilité de ne pas faire figurer la balise "`<string>`" lorsqu'un élément "`value`" contient une chaîne.

Si l'on tient vraiment à autoriser ce comportement, la seule solution consiste à déclarer que l'élément "`value`" accepte un contenu mixte, ce qui conduit à valider de nombreux documents non conformes à **XML-RPC**...

9.3.7 Données binaires

Les documents **XML** sont des documents texte. Une solution pour intégrer des données binaires consiste à les encoder de manière à obtenir un flux de caractères autorisés.

A cet effet, **XML-RPC** prévoit explicitement les valeurs codées en base 64, spécifiées grâce à l'élément "`base64`".

Exemple de chaîne encodée en base 64 :

```
<value>
  <base64>RXh1bXBsZSBkZSBjaGHubmUgY29k6WUgZW4gYmFzZSA2NA==</base64>
</value>
```

> Schéma XML

Les valeurs "`base64`" de **XML-RPC** correspondent au type prédéfini "`xs:base64Binary`".

Ce type est un type normalisé. Toutefois, cet aspect n'est pas rédhitoire dans la mesure où la plupart des implémentations de l'algorithme base 64 autorisent (*et ignorent*) les espaces et les retours à la ligne au sein de la chaîne encodée.

9.4 Types composés

9.4.1 Tableaux

Outre des données scalaires, l'élément "`value`" de **XML-RPC** peut contenir un élément "`array`", formé d'un élément "`data`", qui contient lui-même des éléments "`value`".

Exemple de tableau :

```
<value>
  <array>
    <data>
      <value><int>69</int></value>
      <value><base64>Umj0bmU=</base64></value>
      <value><string>Lyon</string></value>
      <value><base64>Umj0bmUtQWxwZXNM=</base64></value>
    </data>
  </array>
</value>
```

L'exemple ci-dessus correspond à un tableau qui comprend quatre valeurs (*noter que "`base64`" a été ici utilisé pour coder les chaînes contenant des accents*). Remarquer que les valeurs contenues dans un tableau ne sont pas forcément toutes du même type.

Un tableau peut contenir des valeurs qui contiennent elles-mêmes des tableaux ou des structures.

> Schéma XML

Conformément à la définition ci-dessus, le schéma d'une valeur du type *"array"* est le suivant :

```
<xs:complexType name="T_ARRAY">
  <xs:sequence>
    <xs:element name="data" type="T_ARRAY_DATA"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="T_ARRAY_DATA">
  <xs:sequence>
    <xs:element name="value" type="T_VALUE" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

où l'on autorise les tableaux vides (*i.e. à zéro éléments*)...

9.4.2 Structures

Les éléments de tableaux ne sont pas nommés. A contrario, une structure contient des paires nom / valeur.

L'élément *"struct"* contient une liste d'éléments *"member"*, caractérisé chacun par un élément *"name"* et un élément *"value"*.

Exemple de structure :

```
<value>
  <struct>
    <member>
      <name>Numero</name>
      <value><int>69</int></value>
    </member>
    <member>
      <name>Departement</name>
      <value><base64>Umj0bmU=</base64></value>
    </member>
    ...
  </struct>
</value>
```

L'exemple ci-dessus correspond à une structure qui comporte deux membres. L'élément *"value"* peut être de n'importe quel type, et contenir récursivement un autre élément composé (*struct* ou *array*).

Remarquer comment, là encore, la valeur a été transmise en base 64, pour cause de caractères accentués. Il n'est toutefois pas possible d'utiliser la même approche pour le nom de la propriété.

> Schéma XML

Conformément à la définition ci-dessus, le schéma d'une valeur du type *"struct"* est le suivant :

```
<xs:complexType name="T_STRUCT">
  <xs:sequence>
    <xs:element name="member" maxOccurs="unbounded" type="T_STRUCT_MEMBER"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="T_STRUCT_MEMBER">
  <xs:sequence>
    <xs:element name="name" type="T_NAME"/>
    <xs:element name="value" type="T_VALUE"/>
  </xs:sequence>
</xs:complexType>
```

où l'on exige au minimum un membre dans une structure...

Devant le manque de précision des spécifications concernant les caractères autorisés pour le nom des membres de structures, on peut tenter une définition similaire à celle autorisée pour les noms de procédures :

```
<xs:simpleType name="T_NAME">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Za-z0-9/._:~]+" />
  </xs:restriction>
</xs:simpleType>
```

9.4.3 Schéma des valeurs XML-RPC

Le schéma de l'élément *"value"* fait apparaître le choix des divers types que cet élément est susceptible de contenir :

```
<xs:complexType name="T_VALUE" mixed="true">
  <xs:choice>
    <xs:element name="int" type="T_INT"/>
    <xs:element name="i4" type="T_INT"/>
    <xs:element name="boolean" type="T_BOOLEAN"/>
    <xs:element name="double" type="T_DOUBLE"/>
    <xs:element name="dateTime.iso8601" type="T_DATE"/>
    <xs:element name="string" type="T_STRING"/>
    <xs:element name="base64" type="T_BASE64"/>
    <xs:element name="array" type="T_ARRAY"/>
    <xs:element name="struct" type="T_STRUCT"/>
  </xs:choice>
</xs:complexType>
```

A noter que pour autoriser une valeur chaîne sans la balise *"string"* il faut que ce type soit à contenu mixte (*attribut mixed="true"*) ce qui a l'inconvénient de valider des documents non valides, mais moins grave que de ne pas valider des documents conformes...

Chacun des types fait ensuite l'objet d'une définition particulière détaillée par ailleurs.

 Voir l'extrait de schéma complet

[\[http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/value.xsd.html\]](http://dmolinarius.github.io/demofiles/mod-84/xrpc/values/value.xsd.html)

9.5 Format de la réponse

9.5.1 Entêtes HTTP

La réponse à une requête **XML-RPC** prend la forme d'une réponse **HTTP** classique :

```
HTTP/1.1 200 OK
Date: Mon, 17 Jan 2005 13:01:08 GMT
Server: Apache/1.3.28
X-Powered-By: PHP/4.3.3
Content-Length: 793
Connection: close
Content-Type: text/xml
```

Le code de retour doit obligatoirement être *"200 OK"*, sauf erreur bas niveau côté serveur, ce qui laisse la porte ouverte pour les codes de la famille *"500"* (*Internal Server Error*). De fait, les codes de redirection (*famille 300*) sont sans objet pour **XML-RPC**, et les erreurs client (*codes 400*) sont traitées dans le corps de la réponse.

D'après les spécifications **XML-RPC** les directives *"Content-Type"* et *"Content-Length"* sont obligatoires, la valeur de *"Content-Type"* est forcément *"text/xml"*, et la valeur de *"Content-Length"* doit être correcte (*taille du corps en octets*).

Il est possible que la réponse contienne d'autres directives, comme ici *"Connection"*, *"Server"*, ou *"X-Powered-By"*. Cette possibilité est exploitée par certaines implémentations qui sont par exemple amenées à gérer des sessions basées sur le mécanisme des *"Cookies"*.

> La directive Content-Length

Le fait que *"Content-Length"* soit obligatoire est très contraignant. En effet, ceci oblige le serveur à connaître la longueur de la réponse (*et donc à en disposer en entier*) au moment où il compose les entêtes.

D'autre part, cette directive n'est pas obligatoire en **HTTP/1.0** (*c'est la coupure de connexion qui indique la fin du message*).

En **HTTP/1.1** la connexion n'est pas coupée, et le serveur utilise donc traditionnellement un autre mécanisme (*Content-Encoding: chunked*) qui est doublement contraire aux spécifications **XML-RPC** : d'un côté la directive *"Content-Length"* est alors absente, et d'autre part le corps de la réponse est découpé *"en tranches"* délimitées par des informations numériques sur la taille des *"tranches"*, ce qui conduit à *"polluer"* le corps :

```
HTTP/1.1 200 OK
Date: Mon, 17 Jan 2005 13:59:31 GMT
Transfer-Encoding: chunked
Content-Type: text/xml

319
<?xml version="1.0"?>
<methodResponse>
...
</methodResponse>

0
```

La solution consiste évidemment à soigneusement configurer le serveur de manière à ce qu'il génère la directive *"Content-Length"* quelle que soit la version **HTTP** de la requête, et qu'il ne se mette jamais en mode *"Content-Encoding: chunked"*.

9.5.2 Corps de la réponse

Le corps de la réponse est, comme dans le cas de la requête, composé d'un document **XML** bien formé. L'élément racine est cette fois *"methodResponse"*.

L'élément principal contient un élément *"params"*, dont le contenu autorisé est le même que pour l'élément du même nom vu dans le cadre de la requête :

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    ...
  </params>
</methodResponse>
```

> Schéma XML

Le schéma XML d'une réponse standard est donc assez simple :

```
<xs:element name="methodResponse" type="T_RESPONSE"/>
<xs:complexType name="T_RESPONSE">
  <xs:sequence>
    <xs:element name="params" minOccurs="0" type="T_PARAMS"/>
  </xs:sequence>
</xs:complexType>
```

où le type *"T_PARAMS"* est celui défini pour la requête.

9.5.3 Messages d'erreur

En cas d'erreur, le serveur **XML-RPC** peut renvoyer une réponse qui comporte un élément **"fault"** en lieu et place de l'élément **"params"** :

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    ...
  </fault>
</methodResponse>
```

Le contenu de l'élément **"fault"** est forcément une seule et unique valeur (*value*), qui correspond à une structure (*struct*) à deux membres, dont le premier s'appelle **"faultCode"** et a pour valeur un entier (*int*), et le second s'appelle **"faultString"** et a pour valeur une chaîne de caractères.

Exemple de message d'erreur :

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value>
            <int>4</int>
          </value>
        </member>
        <member>
          <name>faultString</name>
          <value>
            <string>Syntax Error : found empty element value</string>
          </value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

> Schéma XML

L'élément **fault** est très contraint, puisque le seul degré de liberté concerne la valeur numérique du code d'erreur et la chaîne de caractères associés à l'erreur.

La rédaction d'un schéma traduisant les contraintes portant sur le contenu de la structure pose pourtant problème : il n'est pas possible de déclarer dans un même contexte deux éléments portant le même nom (*ici "member"*) et dont le contenu doit être différent.

Le schéma se réduit donc à ceci :

```
<xs:complexType name="T_FAULT">
  <xs:sequence>
    <xs:element name="value" type="T_FAULT_VALUE"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="T_FAULT_VALUE">
  <xs:sequence>
    <xs:element name="struct" type="T_FAULT_STRUCT"/>
  </xs:sequence>
</xs:complexType>
```



```
<xs:complexType name="T_FAULT_STRUCT">
  <xs:sequence>
    <xs:element name="member" type="T_STRUCT_MEMBER"/>
    <xs:element name="member" type="T_STRUCT_MEMBER"/>
  </xs:sequence>
</xs:complexType>
```

où les contraintes sur le contenu de la structure ne peuvent pas être explicitées plus avant (*tous les documents conformes seront validés, mais certains documents non conformes le seront aussi*).

9.6 Conclusion

9.6.1 Conclusion

XML-RPC est une application **XML** conçue en 1999 pour autoriser les appels de procédures distantes via le réseau internet.

> Le pire

Elle a été conçue par une personne seule et comporte un certain nombre d'éléments dont l'implémentation aurait mérité d'être un peu mieux réfléchi. Entre autres :

- les échanges se font obligatoirement en **HTTP/1.0** ("*Content-Length*" obligatoire, et "*Content-Encoding: chunked*" interdit,)
- **XML-RPC** s'insère mal dans l'écosystème **XML** (*absence d'espace de noms, impossibilité de transmettre des fragments XML autrement qu'en les codant en base64*),
- les chaînes de caractères, ont été initialement limitées aux caractères **ASCII**, avant d'autoriser "*tous les caractères*" sans plus de détails concernant les encodages autorisés ni la façon de spécifier l'encodage utilisé,
- les valeurs (*int, boolean, double, date, base64*) ne doivent pas comporter d'espaces initiaux et finaux, alors qu'il est très facile depuis n'importe quel langage de supprimer ces espaces, et ce d'autant plus que le type de chacune des valeurs est explicitement indiqué dans les messages,
- les nombres à virgule flottante sont forcément écrits en notation décimale (*les valeurs avec exposant comme 3.14E-1 sont interdites*) et certaines valeurs décrites par les spécifications **IEE 754** sont impossibles à exprimer ("*NaN*", "*INF*", "*-INF*"),

Par ailleurs, de nombreux points des spécifications manquent de précision (*caractères autorisés pour les noms de procédures, directives autorisées au sein des entêtes HTTP, codages de caractères autorisés...*) ce qui laisse aux différentes implémentations le soin de résoudre les problèmes d'interopérabilité.

XML-RPC constitue à ce titre un excellent exemple de ce qu'il faut éviter de faire, et illustre parfaitement la nécessité de réfléchir en comité avant d'émettre des spécifications à vocation globale.

> Le meilleur

Toutefois, **XML-RPC** a survécu à d'autres technologies ayant tenté de résoudre le même problème (*CORBA, Java RMI, ...*) avec moins de succès car trop complexes, plus opaques, moins ouvertes et moins portables.

La force d'**XML-RPC** vient essentiellement d'**XML** qui lui garantit ouverture et portabilité, même si tous les avantages d'**XML** n'ont pas été exploités (*internationalisation, interopérabilité*).

Le second avantage d'**XML-RPC** est lié à sa simplicité : les spécifications tiennent en une page, même si on aurait aimé parfois quelques détails supplémentaires...

> L'état de l'art

Il est aujourd'hui tout à fait possible d'utiliser **XML-RPC**, en particulier dans un contexte très orienté **XML**. Toutefois, les choix technologiques pour concevoir des appels de procédure distante, notamment dans le contexte du Web, ont actuellement tendance à privilégier le format **JSON**, plus léger qu'**XML**.

 Le format JSON

[<http://www.json.org/json-fr.html>]

10. Graphiques Vectoriels SVG

10.1 Introduction

10.1.1 Qu'est-ce que SVG ?

SVG (*Scalable Vector Graphics*) est une application **XML** de graphiques vectoriels ayant fait l'objet de plusieurs recommandations du **W3C** :

● **SVG 1.0** (04/09/2001)

● **SVG 1.1** (14/01/2003 - 2^{ème} édition 16/08/11) - une modularisation de **SVG 1.0**,

🔗 W3C SVG Home Page

[<http://www.w3.org/Graphics/SVG/>]

SVG a initialement été promu par **Adobe**, qui mettait gratuitement à disposition un viewer sous forme de plugin pour les navigateurs les plus répandus de l'époque (*Netscape, Internet Explorer, Opera*) tout en commercialisant **Illustrator** permettant de créer des graphiques ou images dans ce format.

Heureusement, les navigateurs ont peu à peu fini par implémenter nativement les fonctionnalités de **SVG**, pour arriver finalement aux spécifications récentes qui intègrent directement les balises **SVG** au vocabulaire **HTML5**.

SVG est compatible **CSS** (*i.e. on peut spécifier le style des graphiques grâce à une feuille de style*) et scriptable à l'aide de **Javascript**.

10.1.2 Un format graphique pour le Web

> Documents SVG autonomes

Un document **SVG** est avant tout un document **XML** sur lequel il est possible de faire pointer un hyperlien. En ouvrant ce document dans un navigateur, celui-ci affichera le document **SVG** comme il le ferait d'une page **HTML** ou d'une image classique :

```
<a href="tigre.svg">Voir un tigre</a>
```

Voir un tigre :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple1-papier.svg>]

> La balise OBJECT

HTML possède d'autre part la balise **OBJECT**, qui est une balise historique pour l'inclusion de divers types de documents au sein d'une page. Cette balise pouvait être utilisée pour les images, les vidéos ou les sons. Elle s'applique également aux documents **SVG**, avec la syntaxe suivante :

```
<object type="image/svg+xml"
      data="tigre.svg" width="180" height="180" />
```

Remarquer ci-dessus le type mime d'un document **SVG** : **image/svg+xml**.

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple2.html>]

> La balise EMBED

EMBED est une autre balise historique, introduite par **Netscape** et non standard :

```
<EMBED TYPE="image/svg+xml"
      WIDTH="180" HEIGHT="180" SRC="tigre.svg"
      PLUGINSOURCE="http://www.adobe.com/svg/viewer/install/">
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple3.html>]

On a pu voir des constructions pour n'utiliser **EMBED** qu'en dernier recours pour les navigateurs qui ne reconnaissent pas **OBJECT** :

```
<object type="image/svg+xml"
      data="tigre.svg" width="180" height="180">
  <embed type="image/svg+xml"
        width="180" height="180" src="tigre.svg"
        pluginspage="http://www.adobe.com/svg/viewer/install/">
</object>
```

> La balise IMG

La possibilité la plus propre, implémentée le plus tardivement par les navigateurs, consiste à intégrer un graphique **SVG** directement dans une page **HTML** à l'aide de la classique balise **IMG**, dont l'attribut **SRC** pointe vers un document **SVG**.

```

```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple4.html>]

> Les espaces de noms XML

Moyennant une gestion correcte des espaces de noms **XML**, il est possible d'intégrer directement du code source **SVG** au sein d'un document **XML** quelconque. Cette remarque s'applique bien entendu aux documents **XHTML**.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<h1>Scalable Vector Graphics</h1>
...
<svg:svg xmlns:svg="http://www.w3.org/2000/svg" ... >
  <svg:rect x="-600" y="-600" width="1800" ... />
  ...
</svg:svg>
<hr class="bottom"/>
...
</body>
</html>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple5.xml>]

> HTML5

A l'heure actuelle, la méthode la plus simple consiste encore à faire confiance à **HTML5**, dont les spécifications intègrent nativement les balises **SVG**.

```
<!DOCTYPE html>
<h1>Scalable Vector Graphics</h1>
<hr class="top"/>
<svg viewBox="0 0 600 600" width="360" height="360" style="margin: auto;">
  <rect x="-600" y="-600" width="1800" height="1800" stroke="none"
  fill="#f0f0ea"/>
  <g transform="translate(200,200)" style="fill-opacity:1; fill:none;">
    ...
  </g>
</svg>
<hr class="bottom"/>
<address>JohnDoe@mycompany.com</address>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/intro/slide2-exemple6.html>]

10.2 Principes généraux

10.2.1 SVG : une application XML

SVG est avant tout une application **XML**. A ce titre, un certain nombre d'éléments décrits ci-dessous peuvent être nécessaires.

> Déclaration XML

La déclaration **XML** est en toute rigueur optionnelle vis à vis des spécifications. Toutefois, elle peut être utile pour préciser le codage utilisé :

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

Dans le cas d'un document **HTML5** contenant un ou plusieurs graphiques **SVG**, la déclaration **XML** est uniquement présente pour le document global, lorsque celui-ci est sérialisé en **XHTML5**.

> Déclaration de type de document

La déclaration de type de document indique la **DTD** (*Définition de Type de Document*) à laquelle se conforme le document. La déclaration ci-dessous correspond à un document **SVG 1.0** (noter les identificateurs **PUBLIC** et **SYSTEM**) :

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
    "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
```

N.B. Dans le cas d'un document **HTML5** contenant un ou plusieurs graphiques **SVG**, la déclaration de type de document serait différente :

```
<!DOCTYPE html>
```

> Élément racine

Comme l'indique la déclaration de type de document de **SVG** vue ci-dessus, l'élément racine d'un document **SVG** est **svg**.

Vis-à-vis de **XML**, **SVG** possède son propre espace de nom, qu'il est opportun de mentionner au niveau de l'élément racine :

```
<svg xmlns="http://www.w3.org/2000/svg">
...
</svg>
```

N.B. L'attribut **xmlns** ne doit pas être utilisé lorsque la balise **svg** est utilisée dans un document **HTML5**.

> Exemple complet

La structure générale d'un document **SVG** autonome est conforme à :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
    "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
...
</svg>
```

Vers un exemple complet :

[\[http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide1-exemple1.html\]](http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide1-exemple1.html)

10.2.2 Echelles et coordonnées

> Coordonnées utilisateur

En théorie, **SVG** permet de représenter des éléments sur un plan de coordonnées allant de moins l'infini à plus l'infini suivant les deux axes **X** et **Y**.

En pratique, il est plus commode de préciser une fenêtre de coordonnées (*viewBox*) à travers laquelle on désire contempler le document. Cette fenêtre définit ce qu'il est convenu d'appeler le **système de coordonnées utilisateur** à l'aide de l'attribut **viewBox** de l'élément racine **svg**.

```
<svg viewBox="-500 -500 1000 1000">
...
</svg>
```

N.B. l'axe vertical **Y** est compté positivement de haut en bas. La syntaxe de l'attribut **viewBox** est :

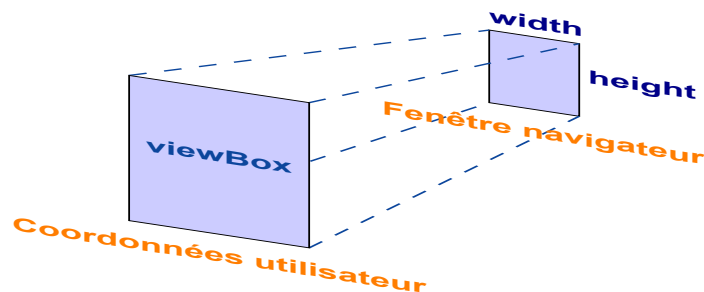
```
viewBox = "x_min y_min largeur hauteur"
```

Exemples :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide2-exemple1.html>]

> Préconisations d'échelle

Le passage des coordonnées utilisateur à un système de coordonnées réelles (*pixels, cm, ...*) est effectué par le dispositif de visualisation (*navigateur, convertisseur vers format papier*) en faisant coïncider la zone utilisateur avec l'espace disponible pour la représentation du graphique :



Les dimensions préconisées pour l'affichage peuvent être indiquées au sein du document lui-même :

```
<svg viewBox="-500 -500 1000 1000" width="400" height="400">
...
</svg>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide2-exemple2.html>]

> Dimensions de la zone d'affichage

Toutefois, l'expérience prouve qu'il vaut mieux préciser les dimensions souhaitées du graphique au sein du document appelant, par exemple grâce aux attributs **width** et **height** de la balise **OBJECT** :

```
<object width="180" height="180"
        type="image/svg+xml" data="slide2-exemple3-1.svg">
</object>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide2-exemple3.html>]

Remarque : Il n'est pas interdit de tenter d'afficher un document **SVG** en modifiant le rapport hauteur / largeur. Toutefois, le résultat obtenu peut dépendre du document lui-même, qui doit en principe explicitement autoriser un tel comportement :

```
<svg xmlns="http://www.w3.org/2000/svg"
    viewBox="0 0 180 180" preserveAspectRatio="none">
...
</svg>
```

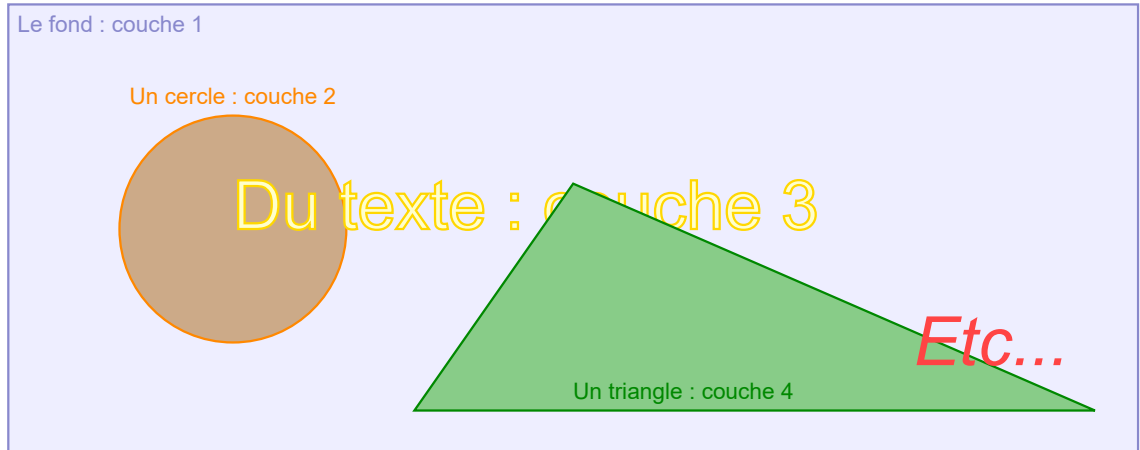
Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/generalites/slide2-exemple4.html>]

10.2.3 Troisième dimension

> L'algorithme du peintre

Pour gérer la troisième dimension (*quel est l'objet représenté à l'avant-plan lorsque plusieurs éléments se trouvent au même endroit ?*) **SVG** utilise l'algorithme du peintre: les éléments recouvrent les précédents, au fur et à mesure de leur tracé :



Le code **SVG** : (*noter surtout l'ordre des éléments...*)

```
<svg viewBox="0 0 500 200">
  <!-- le fond -->
  <rect x="1" y="1" width="498" height="198" ... />
  <!-- le cercle -->
  <circle cx="100" cy="100" r="50" ... />
  <!-- le texte -->
  <text x="100" y="100">Du texte : couche 3</text>
  <!-- le triangle -->
  <polygone points="180,180 250,80 480,180" .../>
  <!-- etc... -->
  <text x="400" y="160">Etc...</text>
</svg>
```

10.3 Formes de base

10.3.1 Attributs communs aux formes graphiques

L'ensemble des formes graphiques de **SVG** texte compris, partagent de nombreux attributs concernant leur représentation graphique, dont certains sont décrits ci-après.

> stroke

L'attribut **stroke** décrit la couleur d'un trait. Les valeurs possibles les plus couramment rencontrées sont **none** (*pas de tracé*), le nom d'une couleur prédéfinie, ou la valeur d'une couleur dans l'un des modes **CSS 2** :

```
<rect stroke="none" ... /> <!-- bords non tracés -->
<rect stroke="ivory" ... /> <!-- couleur prédéfinie -->
<rect stroke="#F00" ... /> <!-- #rgb -->
<rect stroke="#008800" ... /> <!-- #rrggbb -->
<rect stroke="rgb(255,127,0)" ... /> <!-- rgb(r,g,b) -->
<rect stroke="rgb(100%,50%,0%)" ... /> <!-- rgb(r%,g%,b%) -->
```

Voir les couleurs prédéfinies :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/couleurs.svg>]

La valeur par défaut de l'attribut **stroke** (si non spécifié) est **none**.

> stroke-width

La valeur de l'attribut **stroke-width** détermine la largeur du trait. Cette valeur peut être dépourvue d'unité (*coordonnées utilisateur, cf. viewBox*), être exprimée en % (*de la largeur de la fenêtre viewBox*) ou utiliser l'une des unités reconnues par **CSS 2** (*px = coords utilisateur, em, ex, pc, cm, mm ...*).

```
<line stroke-width="2" ... >
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/attr-stroke-width-papier.svg>]

La valeur par défaut de l'attribut **stroke-width** (si non spécifié) est **1**.

> stroke-opacity

Exprimé à l'aide d'une valeur comprise entre **0.0** (*ligne totalement invisible*) et **1.0** (*ligne totalement visible*) cet attribut contrôle la transparence du tracé.

```
<line stroke-opacity="0.5" ... >
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/attr-stroke-opacity-papier.svg>]

La valeur par défaut de l'attribut **stroke-opacity** (si non spécifié) est **1.0**.

> stroke-dasharray

Avec **SVG** il est possible de concevoir le pointillé de ses rêves... L'attribut **stroke-dasharray** peut prendre la valeur **none** (*ligne continue*) ou une liste de longueurs séparées par des virgules, donnant le dessin du pointillé :

```
<line stroke-dasharray="1,1,1,1,1,3" ...>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/attr-stroke-dasharray-papier.svg>]

La valeur par défaut de l'attribut **stroke-dasharray** (si non spécifié) est **none**.

> fill

L'attribut **fill** décrit la couleur du remplissage d'une surface. Là encore, les valeurs les plus courantes sont **none** (*pas de tracé*), le nom d'une couleur prédéfinie, ou la valeur d'une couleur dans l'un des modes **CSS 2** :

```
<rect fill="orange" .../>
```

Revoir les couleurs prédéfinies ? :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/couleurs.svg>]

La valeur par défaut de l'attribut **fill** (si non spécifié) est **black**.

> fill-opacity

Comme **stroke-opacity**, cet attribut contrôle la transparence du remplissage, depuis un tracé totalement visible (*valeur 1.0*), jusqu'à un remplissage totalement transparent (*valeur 0.0*).

```
<rect fill-opacity="0.5" ... >
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/attr-fill-opacity-papier.svg>]

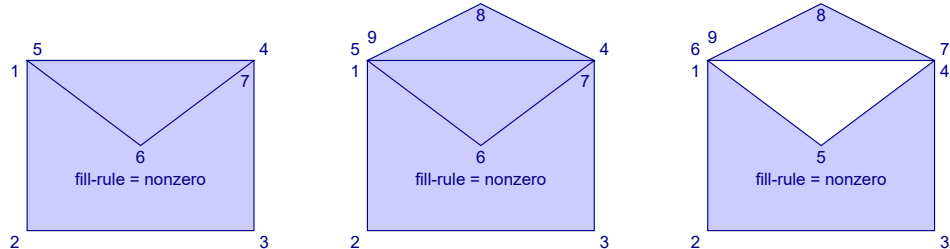
La valeur par défaut de l'attribut **fill-opacity** (si non spécifié) est **1.0**.

> fill-rule

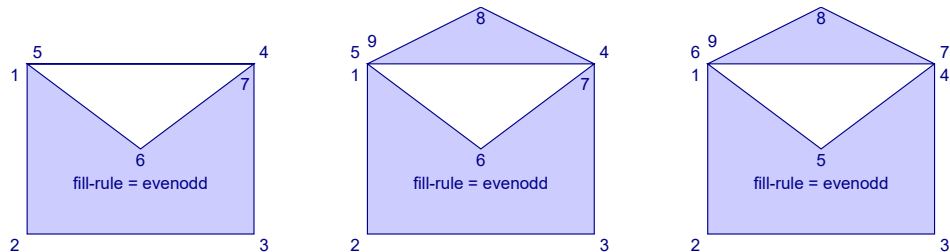
L'attribut **fill-rule** permet de spécifier comment se comporte le remplissage dans le cas de surfaces comportant des trous ou dont les bords présentent des intersections. Les valeurs autorisées sont **nonzero** ou **evenodd** en fonction de l'algorithme de remplissage désiré.

```
<polygon points="..." fill-rule="evenodd"/>
```

L'algorithme **nonzero** cherche à déterminer la position d'un point donné en lançant un rayon depuis ce point vers l'infini. Le long de ce rayon et partant de zéro, on compte +1 chaque fois qu'on croise un bord orienté de la gauche vers la droite, -1 pour un bord orienté de la droite vers la gauche. Le point est à l'intérieur (*donc coloré*) si le résultat final est non nul :



L'algorithme **evenodd** procède un peu de même, en comptant simplement le nombre d'intersections entre le rayon et les bords de la surface. Si ce nombre est impair le point est à l'intérieur (*i.e. coloré*), sinon il est à l'extérieur :



La valeur par défaut de l'attribut **fill-rule** (si non spécifié) est **nonzero**.

10.3.2 Segments et lignes

> line

line est la forme de base la plus simple.

Cet élément correspond à un simple segment, dont il suffit de donner les coordonnées à l'aide des attributs **x1**, **y1** (point de départ), et **x2**, **y2** (point d'arrivée) :

```
<line x1="10" y1="10" x2="100" y2="100" />
```

Remarque : en cas d'attribut non spécifié la valeur par défaut est : **0.0**

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur et le style de trait s'appliquent à l'élément **line**.

Exemple de figures à base de segments :



Un motif de la frise ci-dessus est ainsi tracé à l'aide des éléments suivants :

```
<line x1="0" y1="130" x2="0" y2="20"/>
<line x1="0" y1="20" x2="120" y2="20"/>
<line x1="120" y1="20" x2="120" y2="102.5"/>
<line x1="120" y1="102.5" x2="60" y2="102.5"/>
<line x1="60" y1="102.5" x2="60" y2="75"/>
<line x1="60" y1="75" x2="90" y2="75"/>
<line x1="90" y1="75" x2="90" y2="47.5"/>
<line x1="90" y1="47.5" x2="30" y2="47.5"/>
<line x1="30" y1="47.5" x2="30" y2="130"/>
<line x1="30" y1="130" x2="150" y2="130"/>
```

> polyline

Pour la représentation de graphiques du type de celui ci-dessus, il est évidemment assez fastidieux (*et volumineux en terme de taille de fichier*) de répéter deux fois les coordonnées de chacun des points (*point d'arrivée du segment n, et point de départ du segment n+1*).

L'élément **polyline** permet de représenter une ligne continue constituée de segments jointifs consécutifs :

```
<polyline points="0,130 0,20 120,20 120,102.5 ..."/>
```

Le motif élémentaire de la frise ci-dessus, se verra donc représenté par l'extrait de code suivant, qui est éventuellement un peu moins lisible, mais indéniablement plus concis !

```
<polyline points="0,130 0,20 120,20 120,102.5
60,102.5 60,75 90,75 90,47.5
30,47.5 30,130 150,130"/>
```

La frise avec polyline :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/formes/line-polyline.svg>]

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur et le style de trait, ainsi que ceux relatifs au remplissage s'appliquent à l'élément **polyline**.

N.B. Pour remplir une forme tracée avec **polyline**, **SVG** ferme le tracé avec un segment virtuel joignant le dernier point au premier...

10.3.3 Rectangles

> rect

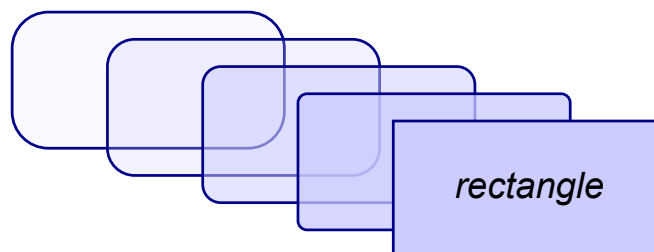
L'élément **rect** correspond à un rectangle. Il est décrit par les coordonnées de son coin haut gauche (*x min, ymin*) et par ses dimensions (*longueur et largeur*) :

```
<rect x="0" y="0" width="300" height="100"/>
```

La valeur par défaut des coordonnées **x** et **y** (*si non spécifiées*) est **0**. Si l'une des dimensions n'est pas spécifiée, le rectangle n'est pas tracé.

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur, le style de trait et le remplissage s'appliquent à l'élément **rect**.

Exemple de rectangles :



Ainsi qu'illustré par l'exemple ci-dessus, le rectangle **SVG** permet d'arrondir les angles !

Les attributs **rx** et **ry** donnent respectivement le rayon horizontal et le rayon vertical des portions d'ellipse servant à arrondir les angles :

```
<rect x="50" rx="40" ry="30" width="300" height="150"/>
```

Si aucun des attributs **rx** et **ry** n'est spécifié, leur valeur par défaut est **0** (*coins non arrondis*). Si l'un seulement est donné, l'autre est supposé avoir la même valeur.

La valeur de ces attributs ne peut être supérieure à la demi-dimension correspondante du rectangle, si c'est le cas elle sera limitée à cette valeur.

10.3.4 Polygones

> polygon

L'élément **polygon** représente un polygone. Un polygone n'est autre qu'une polyligne fermée.

Si la distinction **polygon** / **polyline** n'est pas significative pour ce qui concerne le remplissage, elle l'est pour le tracé des contours : un polygone est toujours fermé.

Pour le reste, les attributs d'un polygone sont les mêmes que ceux d'une polyligne :

```
<polygon points="0,130 0,20 120,20 120,102.5 ..."/>
```

Exemples de polygones :



10.3.5 Cercles

> circle

L'élément **circle** représente un cercle.

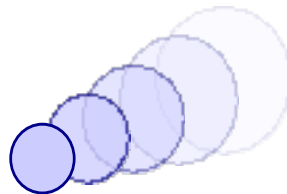
Un cercle est classiquement donné par les coordonnées de son centre **cx** et **cy**, ainsi que par la valeur de son rayon **r** :

```
<circle cx="150" cy="150" r="100"/>
```

La valeur par défaut des coordonnées **cx** et **cy** (si non spécifiées) est 0. Si le rayon **r** n'est pas spécifié, le cercle n'est pas tracé.

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur, le style de trait et le remplissage s'appliquent à l'élément **circle**.

Exemples de cercles :



Attention ! Le cercle est tracé dans le système de coordonnées utilisateur (cf. attribut *viewBox* de l'élément *racine svg*). Si le rapport hauteur / largeur n'est pas conservé pour la représentation finale, alors le cercle apparaîtra comme une ellipse...

10.3.6 Ellipse

> ellipse

L'élément **ellipse** représente une ellipse.

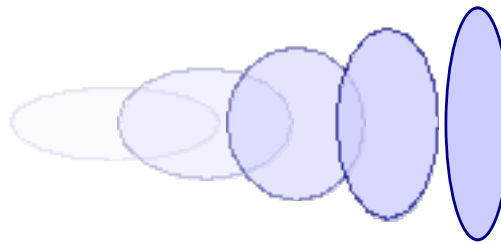
Une ellipse est donnée par les coordonnées **cx,cy** de son centre et par les rayons **rx** et **ry**.

```
<ellipse cx="150" cy="150" rx="100" ry="50"/>
```

La valeur par défaut des coordonnées **cx** et **cy** (si non spécifiées) est 0. Si l'un des rayons **rx** ou **ry** n'est pas spécifié, l'ellipse n'est pas tracée.

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur, le style de trait et le remplissage s'appliquent à l'élément **ellipse**.

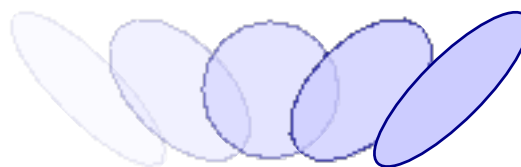
Exemples d'ellipses :



Remarque : Vue la manière dont elle spécifiée, en première approche les axes de l'ellipse ne peuvent évidemment qu'être horizontaux et verticaux.

SVG possède néanmoins des **transformations** (*vues plus loin*) qui permettront de faire tourner les ellipses...

Exemples d'ellipses tournées :



10.4 Texte

10.4.1 Élément texte

> text

L'élément **text** permet d'intégrer du texte aux graphiques **SVG**.

Les attributs permettant de positionner le texte sont ses coordonnées **x** et **y**. Le texte lui-même correspond au contenu de l'élément :

```
<text x="10" y="20">Un exemple de texte</text>
```

La valeur par défaut des coordonnées **x** et **y** (*si non spécifiées*) est **0**.

Les attributs communs [cf. paragraphe 10.3.1] aux formes graphiques concernant la couleur, le style de trait et le remplissage s'appliquent à l'élément **text** :

```
<text y="21" x="10">Un exemple de texte</text>
<text y="21" x="170" fill="#CCF" stroke-width="0.7" stroke="#008">SVG</text>
```

Un exemple de texte SVG

10.4.2 Polices de caractères

Les propriétés relevant de la description de la police de caractères sont à rapprocher de celles vues avec **CSS 2**.

> font-family

L'attribut **font-family** permet de spécifier la police à utiliser, à l'aide d'un nom (ou d'une liste de noms) de police générique (*serif, sans-serif, cursive, fantasy, monospace*) ou de police spécifique à la plateforme du client.

La valeur par défaut (*attribut non spécifié*) dépend du client.

:

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-font-family-generic-papier.svg>]

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-font-family-specific-papier.svg>]

> font-size

L'attribut **font-size** donne la taille de la police.

La valeur est une taille absolue à prendre dans une table gérée par le client (*xx-small, x-small, small, medium, large, x-large, xx-large*), une taille relative à celle héritée de l'élément parent (*smaller, larger*), une longueur (avec unité, *px = coordonnées utilisateur*), ou un pourcentage de la taille héritée de l'élément parent.

La valeur par défaut (*attribut non spécifié*) est **medium**.

> font-weight

Cet attribut permet de spécifier la graisse de la police.

Comme en **CSS 2** les valeurs possibles de l'attribut **font-weight** sont : **normal, bold, bolder, lighter** ou une valeur parmi **100, 200, ... 900**.

La valeur par défaut (*attribut non spécifié*) est **normal**.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-font-weight-papier.svg>]

> font-style

L'attribut **font-style** indique s'il s'agit d'une police italique.

Les valeurs possibles sont **normal, italic** ou **oblique**.

La valeur par défaut (*attribut non spécifié*) est **normal**.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-font-style-papier.svg>]

> font-variant

L'attribut **font-variant** possède deux valeurs : **normal** et **small-caps**.

La valeur par défaut (*attribut non spécifié*) est **normal**.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-font-variant-papier.svg>]

> text-decoration

L'attribut **text-decoration** prend les valeurs suivantes : **none, underline, overline, line-through** ou **blink**.

La valeur par défaut (*attribut non spécifié*) est **none**.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-text-decoration-papier.svg>]

> letter-spacing

L'espace entre les caractères peut être réglé grâce à l'attribut **letter-spacing**. Les valeurs possibles sont : **normal** ou une longueur munie d'une unité ou sans unité (*coordonnées utilisateur*)

La valeur par défaut (*attribut non spécifié*) est **normal**.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-letter-spacing-papier.svg>]

> word-spacing

L'espace entre les mots peut être réglé indépendamment du précédent à l'aide de l'attribut **word-spacing**. Les valeurs sont **normal** ou une longueur.

La valeur par défaut (*attribut non spécifié*) est **normal**.

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-word-spacing-papier.svg\]](http://dmolinarius.github.io/demofiles/mod-84/svg/texte/attr-word-spacing-papier.svg)

10.4.3 Attributs de l'élément texte

> x,y

Outre le fait de préciser la position de la chaîne de caractères, les attributs **x** et **y** peuvent être employés sous forme de liste donnant la position individuelle de chacun des caractères de la chaîne :

```
<text x="0,5.26,10.52,15.78,21.05, ..."
      y="0,4.75, 8.37, 9.96, 9.15, ...">
  Ceci est un exemple de texte chahuté !
</text>
```

Exemple :

Ceci est un exemple de texte chahuté !

> dx,dy

Plutôt que de donner une liste de positions absolues pour les caractères, il est possible de spécifier plutôt une position relative, par rapport à la position normale des caractères. Cette opération s'effectue à l'aide des attributs **dx** et **dy** :

```
<text dx="0, 0, -3, -3, 0, 0, 0, 0, 0, -3, 0"
      dy="0, 0, 4, -7, 3, 0, 0, 0, 0, 4, -4">
  axo2 + bxo + c = 0
</text>
```

Le texte correspondant à l'exemple ci-dessus :

$$ax_o^2 + bx_o + c = 0$$

N.B. Une fois le texte déplacé (*i.e. dx ou dy non nul*), la chaîne de caractères repart de la nouvelle position. Si l'effet recherché est de mettre comme ci-dessus un caractère en indice ou en exposant, il est nécessaire de refaire un décalage de signe opposé au caractère suivant pour revenir sur la ligne normale...

> rotate

L'élément **text** admet un autre attribut appelé **rotate** permettant de spécifier l'angle des caractères :

```
<text x="0" y="0" rotate="20">
  Autant en emporte le vent !
</text>
```

Autant en emporte le vent !

Comme les attributs précédents, **rotate** accepte également une liste s'appliquant séquentiellement à chacun des caractères de la chaîne.

Pour réorienter l'ensemble des caractères de la chaîne avec éventuellement un angle différent pour chacun d'eux, on peut écrire :

```
<text x="0" y="0" rotate="20, 20, 20, 20, 20, 20, 20, 20, 20 ...">
  Autant en emporte le vent !
</text>
```

Résultat :

Autant en emporte le vent !

N.B. Il est bien entendu possible de compléter l'attribut **rotate** avec les effets obtenus à l'aide des listes de coordonnées vues plus haut, afin d'obtenir du texte dont la position et l'angle varient de caractère en caractère.

10.4.4 Sous-chaînes

> tspan

Il existe de nombreux cas où l'on désire pouvoir affecter des caractéristiques particulières à une sous-chaîne d'un texte (*changement de police, couleur, indice, exposant...*).

L'obtention d'un tel résultat uniquement basé sur l'élément **text** serait difficile, car elle nécessiterait à chaque fois de calculer très exactement la position de départ (*attribut x*) de chaque nouvelle sous-chaîne.

Afin d'éviter ceci il existe un élément nommé **tspan** qui, apparaissant comme sous-élément d'un élément **text** ou d'un autre élément **tspan**, garde le compte de la position du caractère suivant :

```
<text>
  Du texte
  <tspan fill="#000088">en couleur</tspan>
  dans une phrase...
</text>
```

Exemple :

Du texte en couleur dans une phrase...

L'élément **tspan** est très utile pour la gestion du texte sur plusieurs lignes :

L'enfance de Zéphyrin.

C'est le 1er janvier, à minuit une seconde sexagésimale de temps moyen, que le jeune Brioché poussa ses premiers vagissements. A son baptême, il reçut les prénoms harmonieux, poétiques et distingués de Pancrace, Eusèbe, Zéphyrin, ce dont il parut se soucier comme un cloporte d'un ophicléide.

Consultée à son sujet, une somnambule extralucide, phrénologue distinguée, pédicure de nombreuses têtes couronnées, lui découvrit la bosse du mouvement perpétuel. D'où elle conclut logiquement qu'il serait un grand voyageur ou un grand mathématicien, à moins qu'il ne fût affligé de la danse de Saint-Guy.

Conformément à la prédiction de la somnambule, Zéphyrin montra, dès sa plus tendre enfance, des dispositions étonnantes pour les sciences expérimentales. Livré à lui-même, il s'ingéniait avec beaucoup de persévérance à résoudre les problèmes les plus compliqués.

Extrait de : l'idée fixe du Savant Cosinus.

L'exemple ci-dessus est obtenu à l'aide d'un seul élément **text** comportant un élément **tspan** par ligne :

```
<text>
  <tspan fill="#000088">L'enfance de Zéphyrin.</tspan>
  <tspan x="0" dy="6">C'est le 1er janvier ...</tspan>
  <tspan x="0" dy="3">poussa ses premiers ...</tspan>
  <tspan x="0" dy="3">et distingués de ...</tspan>
  ...
</text>
```

N.B. SVG ne gère pas les passages à la ligne. Il est du ressort de l'application générant le code source **SVG** de régler les points de césure et les coordonnées de début de ligne.

SVG possède également des mécanismes permettant de créer du texte justifié à droite (*attributs `textLength` et `lengthAdjust`*).

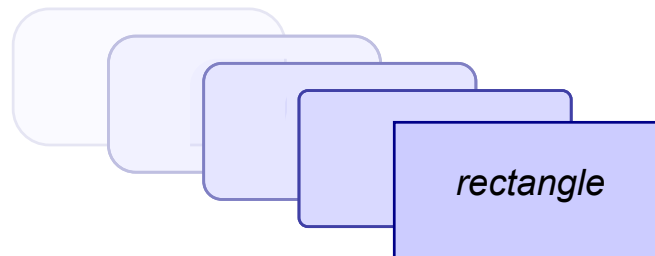
10.5 Groupes

> g

L'élément **g** est un élément structurel permettant de regrouper d'autres éléments **SVG** afin de les manipuler ensemble, en leur affectant par exemple des caractéristiques de présentation communes (*couleurs, traits, polices de caractères ...*) ou, comme il sera vu plus tard, de les positionner et de les orienter ensemble dans l'espace à l'aide d'une même transformation.

N.B. Un groupe peut contenir à peu près n'importe quel autre élément **SVG**, y compris un autre groupe.

Exemple :



Dans l'exemple ci-dessus, les rectangles héritent leur couleur et leur bordure de l'élément **g** englobant :

```
<g fill="#CCCCFF" stroke="#000050" stroke-width="3">
  <rect width="30" height="15" x="9" y="2" rx="4" opacity="0.1"/>
  <rect width="30" height="15" x="19" y="5" rx="3" opacity="0.25"/>
  <rect width="30" height="15" x="30" y="8" rx="2" opacity="0.5"/>
  <rect width="30" height="15" x="40" y="11" rx="1" opacity="0.75"/>
  <rect width="30" height="15" x="51" y="14" opacity="1.0"/>
</g>
```

Il faut toutefois noter que l'élément **g** ne transmet que les propriétés de présentation, mais pas les propriétés dimensionnelles (*width, height...*) ou de position (*x, y ...*).

10.6 Transformations

10.6.1 Principe des transformations

Soit à tracer la frise ci-dessous, visiblement composée d'un même motif plusieurs fois répété :



Cette frise est effectivement implémentée à l'aide d'éléments **polyline** correspondant au motif élémentaire, tous décrits à l'aide de la même liste de coordonnées (*sauf le premier et le dernier motif dont le segment d'extrémité est différent*).

La juxtaposition spatiale des motifs élémentaires est obtenue en spécifiant pour chacun d'eux une **translation** qui l'amène à la position souhaitée :

```
<polyline transform="translate(18.75,0) "
  points="0,130 0,20 120,20 120,102.5 60,102.5 60,75 ..."/>
<polyline transform="translate(168.75,0) "
  points="0,130 0,20 120,20 120,102.5 60,102.5 60,75 ..."/>
<polyline transform="translate(318.75,0) "
  points="0,130 0,20 120,20 120,102.5 60,102.5 60,75 ..."/>
```

La translation n'est qu'un exemple parmi l'ensemble des transformations géométriques dont dispose **SVG** (*translation, modification d'échelle, rotation, repères non orthogonaux, transformations quelconques...*).

Ainsi qu'illustré ci-dessus, ces transformations sont appliquées aux divers éléments **SVG** à l'aide de l'attribut **transform**.

N.B. Outre le texte et les divers éléments graphique, l'élément **g** est bien entendu une cible de choix pour les transformations, puisqu'il permet en une seule opération de modifier les caractéristiques géométriques de l'ensemble des éléments qu'il regroupe...

10.6.2 Translations

La translation est la plus simple des transformations. Elle consiste simplement à déplacer l'élément concerné dans la fenêtre de vue vers un nouvel emplacement.

La fonction **translate** spécifiée à l'aide de l'attribut **transform** prend deux arguments **tx** et **ty**, correspondant aux coordonnées du déplacement à effectuer :

```
transform="translate(tx,ty) "
```

N.B. le paramètre **ty** est optionnel. S'il est omis il est supposé valoir **0**.

La translation est une opération très utile pour représenter plusieurs fois un même objet éventuellement complexe car composé de multiples éléments :



Ainsi, le panneau **STOP** ci-dessus regroupe un polygone et un texte, groupés à l'aide d'un élément **g** puis représentés deux fois à des positions différentes grâce à une translation appliquée au groupe :

```
<g transform="translate(100,80) ">
  <polygon points="69.29,28.7 28.7,69.29 -28.7,69.29 -69.29,28.7 ..."/>
  <text dy="10">STOP</text>
</g>
<g transform="translate(900,80) ">
  <polygon points="69.29,28.7 28.7,69.29 -28.7,69.29 -69.29,28.7 ..."/>
  <text dy="10">STOP</text>
</g>
```

10.6.3 Modification d'échelle

> Modification d'échelle uniforme

La modification d'échelle s'effectue à l'aide de la fonction **scale**. Elle consiste à grossir ou à rapetisser l'objet considéré.

```
transform="scale (sx) "
```

Si **sx** est supérieur à 1, l'objet sera agrandi, dans le cas contraire il sera rapetissé.



L'image ci-dessus a été obtenue en réutilisant le groupe d'éléments correspondant au panneau (*polygone* + *texte*) et en lui faisant subir d'abord une diminution d'échelle (*scale*) puis une translation (*translate*) pour l'amener à l'endroit souhaité :

```
<g class="stop" transform="translate(800,300)">
  <polygone points="69.29,28.7 28.7,69.29 -28.7,69.29 -69.29,28.7 ..."/>
  <text dy="10">STOP</text>
</g>
<g class="stop" transform="translate(580,28)">
  <g transform="scale(0.3)">
    <polygone points="69.29,28.7 28.7,69.29 -28.7,69.29 -69.29,28.7..."/>
    <text dy="10">STOP</text>
  </g>
</g>
```

> Déformation

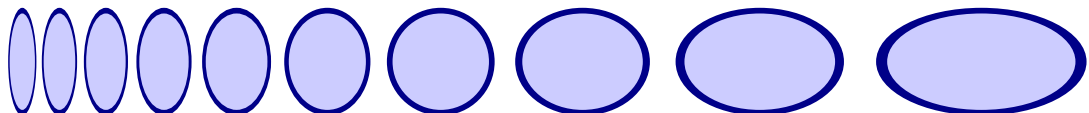
Il est également possible de spécifier un deuxième paramètre à la fonction **scale** :

```
transform="scale (sx, sy) "
```

De cette manière l'objet sera déformé d'un facteur différent suivant les deux dimensions **x** et **y**.

N.B. Lorsque **sy** est omis, il est de fait supposé égal à **sx**.

Exemple d'un cercle transformé en ellipses par redimensionnement non uniforme :



Une partie du code correspondant :

```
<g transform="scale(0.8,1)">
  <circle r="45" cx="500" cy="50"/>
</g>
<circle r="45" cx="500" cy="50"/>
<g transform="scale(1.25,1)">
  <circle r="45" cx="500" cy="50"/>
</g>
```


Noter en observant le graphique ci-dessus comment **SVG** redimensionne chacune des ellipses **y compris la largeur du trait**, qui n'est plus uniforme suivant les directions horizontales et verticales...

10.6.4 Rotations

> Rotation autour de l'origine

Une rotation s'effectue avec la fonction **rotate** :

```
transform="rotate(a)"
```

Cette opération aura pour effet de faire tourner la représentation de l'objet concerné de **a** degrés autour de l'origine du système de coordonnées utilisateur.

Exemple :



Afin de ne pas perdre l'objet dans l'espace, il est recommandé de le définir avec des coordonnées au voisinage de l'origine, de lui appliquer la rotation désirée, puis de le traduire ensuite à sa position finale :

```
<g transform="translate(100,100)">
  <g transform="scale(2)">
    <g transform="rotate(-45)">
      <circle transform="scale(1,0.25)" r="50"/>
      <text y="5.5">ellipses</text>
    </g>
  </g>
</g>
```

N.B. Comme l'axe vertical est compté du haut vers le bas, le sens de rotation positif correspond au sens horaire et non pas trigonométrique comme on pourrait s'y attendre...

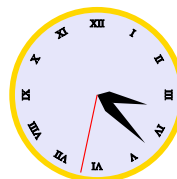
> Rotation autour d'un point arbitraire

La fonction **rotate** accepte en plus de l'angle **a** les coordonnées **cx,cy** du centre de rotation :

```
transform="rotate(a,cx,cy)"
```

Cette transformation est équivalente à une translation *translate(-cx,-cy)* pour amener le point (cx,cy) à l'origine, suivie par une rotation *rotate(a)* autour de l'origine, puis à nouveau par une translation *translate(cx,cy)* pour revenir à la position initiale.

Exemple :
(heure de création de ce document)



Les chiffres de cette horloge ont été tracés à l'aide d'une rotation autour du centre du cadran :

```
<circle cx="500" cy="100" r="75"/>
<text x="500" y="40">XII</text>
<text x="500" y="40" transform="rotate(30,500,100)">I</text>
<text x="500" y="40" transform="rotate(60,500,100)">II</text>
<text x="500" y="40" transform="rotate(90,500,100)">III</text>
...
```

10.6.5 Déformations

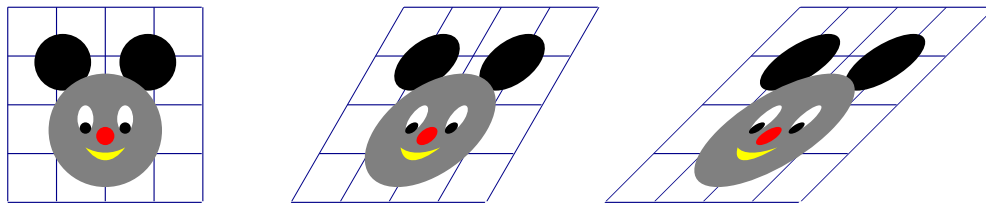
> Déformation suivant X

Une autre transformation élémentaire proposée par **SVG** est la déformation. Les déformations consistent à tracer les objets dans des repères non orthogonaux.

La déformation suivant **X** s'effectue à l'aide de la fonction **skewX** :

```
transform="skewX(a) "
```

La transformation **skewX** conserve l'axe **X**, tandis que l'axe **Y** est réorienté d'un angle **a** :



L'exemple ci-dessus correspond à des déformations de -30 puis -45 degrés.

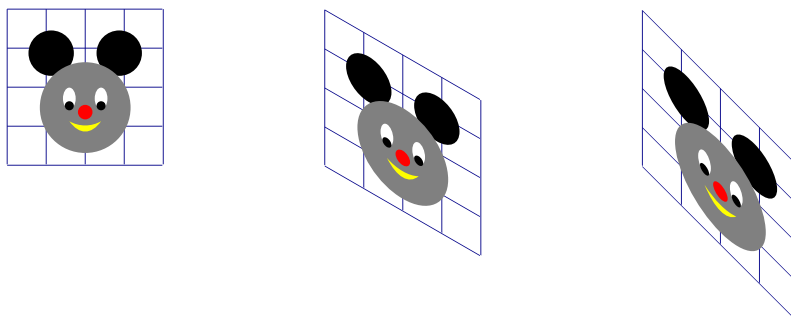
```
<g transform="skewX(-30)">
...
</g>
<g transform="skewX(-45)">
...
</g>
```

> Déformation suivant Y

De même, la déformation suivant **Y** s'effectue à l'aide de la fonction **skewY** :

```
transform="skewY(a) "
```

La transformation **skewY** conserve l'axe **Y**, tandis que l'axe **X** est réorienté d'un angle **a** :



L'exemple ci-dessus correspond à des déformations de 30 puis 45 degrés.

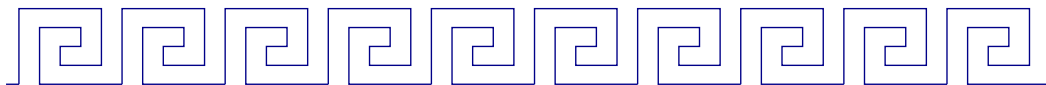
```
<g transform="skewY(30)">
...
</g>
<g transform="skewY(45)">
...
</g>
```

10.7 Réutilisation d'éléments

10.7.1 Réutilisation d'éléments

> use

Revenons une fois de plus à notre frise :



La répétition in extenso du motif élémentaire, déplacé d'intervalle en intervalle, conduit à un document dont le volume pourrait être réduit si l'on était capable de réutiliser simplement le motif.

Pour ceci, il faut lui donner un nom à l'aide d'un attribut **id**, puis y faire référence à l'aide d'un élément **use** pointant sur le motif à l'aide d'un fragment d'URL :


```
<polyline id="motif"
  transform="translate(-281.25,0)"
  points="0,130 0,20 120,20 120,102.5 ..."/>
<use x="150" xlink:href="#motif"/>
<use x="300" xlink:href="#motif"/>
<use x="450" xlink:href="#motif"/>
...
```

N.B. L'élément **use** utilise un lien **xml** défini par une spécification nommée **XLink**. Qu'il suffise ici de constater la nécessité de déclarer l'espace de noms correspondant, au niveau de l'élément **use** lui-même ou de l'un de ses parent, par exemple l'élément racine :

```
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  viewBox="-300 18 1650 150">
...
</svg>
```

 Les spécifications de XLink

<http://www.w3.org/TR/xlink/>

 Les espaces de noms en XML

<http://www.w3.org/TR/REC-xml-names/>

> symbol

L'inconvénient de l'exemple ci-dessus est que l'élément servant de modèle est visible sur la page, à sa position propre.

Il paraît tentant de pouvoir définir le motif élémentaire de manière générique, en l'absence de toute représentation graphique, puis de l'utiliser en le positionnant pour chacune des instances désirées.

Ce fonctionnement est possible, en définissant le modèle dans une section spéciale du document **SVG** précédant les éléments graphiques, appelée **defs**. Le modèle sera défini à l'aide de l'élément **symbol** :

```
<defs>
  <symbol id="motif" viewBox="0 0 150 150">
    <polyline points="0,130 0,20 120,20 120,102.5 60,102.5 ..."/>
  </symbol>
</defs>
```

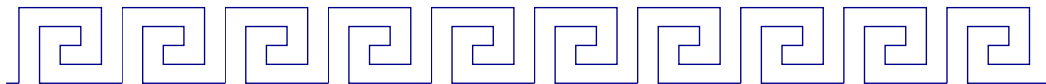
Remarquer l'attribut **viewBox** permettant de définir la zone utile du symbole.

La réutilisation d'un symbole se fait encore une fois à l'aide d'un élément **use** :

```
<use x="-281.25" width="150" height="150" xlink:href="#motif"/>
<use x="-131.25" width="150" height="150" xlink:href="#motif"/>
<use x="18.75" width="150" height="150" xlink:href="#motif"/>
...
```

Noter la nécessité d'indiquer l'échelle du symbole à l'aide des attributs **width** et **height**, en plus de sa position. Ces informations servent à recadrer la zone utile définie par l'attribut **viewBox** de l'élément **symbol**.

Le résultat est évidemment :



10.8 Feuilles de style CSS

10.8.1 SVG et CSS

> Exemple

Les avantages de séparer le fond de la forme, le contenu d'un document du style de la présentation ne sont plus à démontrer. On peut par exemple citer **HTML** et **CSS** ou dans une approche différente **XML** et **XSL**.

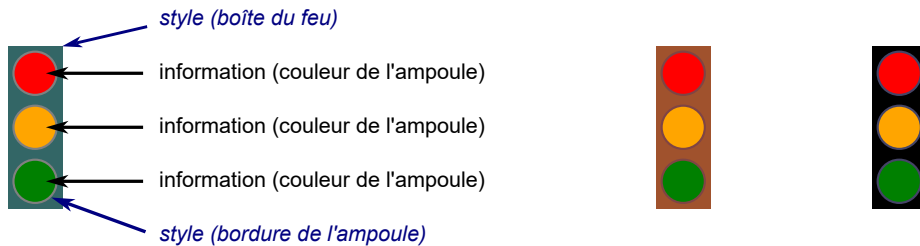
Ce raisonnement peut-il s'appliquer dans le domaine graphique ? La réponse est oui. **SVG** est compatible **CSS** :



Les trois ours ci-dessus ont été tracés avec les mêmes primitives graphiques, faisant simplement référence à trois feuilles de style différentes.

> Position du problème

Le problème consistant à séparer le fond de la forme, l'information de sa présentation, est toutefois un peu plus délicat à régler pour un document graphique que pour un document texte.



En effet, la police de caractère est un attribut de style lorsqu'il s'agit de l'appliquer à un paragraphe, mais devient de l'information dans le cas d'un logo. La couleur relève du style dans le cas d'un graphe de données (*courbes, camemberts...*) mais est chargée d'information sur les représentations cartographiques (*routes, chemins, voies ferrées...*) ou dans le cas d'un feu de signalisation.

> La solution SVG

C'est pour cette raison qu'en **SVG** une propriété peut être spécifiée indifféremment à l'aide d'un **attribut** (*information*) ou d'une règle **CSS** (*style*) :

Code **SVG** :

```
<rect width="50" height="150"/>
<circle fill="red" cx="25" cy="25" r="20"/>
<circle fill="orange" cx="25" cy="75" r="20"/>
<circle fill="green" cx="25" cy="125" r="20"/>
```

Règles **CSS** :

```
rect { stroke:none; fill:#336666; }
circle { stroke:grey; stroke-width:2 }
```

N.B. La possibilité de spécifier des valeurs à l'aide d'une feuille de style est limitée aux **propriétés de présentation**.

10.8.2 Feuille de style interne ou externe

> Élément style

La méthode la plus simple pour adjoindre des règles **CSS** à un document **SVG** consiste, comme avec **HTML**, à utiliser une feuille de style interne.

L'élément **style** devra se trouver dans la section des définitions **defs** (*comme l'élément symbol*).

```
<defs>
  <style type="text/css"><![CDATA[
    rectangle { stroke:none; fill:#336666; }
    circle { stroke:grey; stroke-width:2 }
    text { stroke:none; font-family: Arial; font-size:20; }
  ]]></style>
</defs>
```

N.B. Comme on peut le noter ci-dessus il sera utile, par acquis de conscience, de protéger le contenu de la balise **style** l'aide d'un section **CDATA** (*ou pas*)...

L'usage d'un feuille de style interne n'est pas idéal. En effet, si elle permet de partager un même style pour de nombreux éléments d'un même document, elle ne peut être réutilisée dans un autre document que par recopie...

> Feuille de style externe

La seconde méthode consiste à faire référence à une feuille de style externe, localisée dans un document **CSS** à part entière.

La syntaxe à utiliser est la syntaxe classique permettant de faire référence à une feuille de style depuis un document **XML** :

```
<?xml-stylesheet href="style.css" type="text/css"?>
```

Cette instruction de traitement est à placer au sein du document **SVG** dans la zone des entêtes, avant l'élément racine **svg** :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
    "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<?xml-stylesheet href="smile1.css" type="text/css"?>
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 180 180">
...
</svg>
```

10.8.3 Sélection d'éléments

> Sélection par famille

Avec **SVG** comme avec **HTML**, les sélecteurs **CSS** permettent de désigner tous les éléments d'une famille donnée :

```
rect { fill:blue; stroke:yellow; }
circle { stroke-width: 2; }
```

Dans l'exemple ci-dessus tous les éléments **rect** auront un fond bleu et une bordure jaune, tandis que les cercles auront une bordure de largeur 2.

> Eléments identifiés

Les éléments identifiés de manière unique à l'aide de l'attribut **id**, peuvent être désignés de la même façon que lorsque **CSS** s'applique à un document **HTML** :

```
#face { fill:black; }
#eyes ellipse { fill:white; stroke:black; }
```

Dans ce exemple, l'élément dont l'attribut **id** vaut *face* sera rempli en noir, et les ellipses enfant de l'élément identifié par *eyes* seront remplies en blanc avec une bordure noire.

> Classes d'éléments

Les concepteurs de **SVG** ont également repris la possibilité apparue avec **HTML** d'attacher un attribut **class** à l'ensemble des éléments graphiques.

Une série d'éléments peut ensuite être sélectionnée grâce à la valeur de leur attribut **class** :

```
.body { stroke:none; fill:#336666; }
.lamp { stroke:grey; stroke-width:2 }
```

Tous les éléments de la classe *body* seront remplis d'une couleur cyan foncé, sans bordure, et les éléments de la classe *lamp* auront une bordure grise d'une largeur de 2.

> Sélecteurs CSS

Pour le reste, la syntaxe des sélecteurs est celle spécifiée par **CSS 2**.

11. XSL Formatting Objects

11.1 Introduction

11.1.1 Fiche d'identité

XSL-FO (*XSL Formatting Objects*) est une application **XML** qui décrit la présentation (*mise en page*) d'un document texte, généralement à des fins d'impression. La sémantique de l'application **XSL-FO** est par conséquent orientée **présentation**.

XSL 1.0 (aka. *XSL-FO*) a fait l'objet d'une première recommandation émise par le **W3C** en octobre 2001, suivie par **XSL 1.1** en décembre 2006. **XSL-FO 2.0** a fait l'objet d'un document de travail (*Working Draft*) en janvier 2012, qui n'a pas évolué depuis.

📄 Recommandation XSL 1.0

[<https://www.w3.org/TR/2001/REC-xsl-20011015/>]

📄 Spécifications XSL 1.1

[<http://www.w3.org/TR/xsl11/>]

📄 Document de travail sur XSL 2.0

[<https://www.w3.org/TR/xslfo20/>]

📄 L'ensemble des spécifications XSL au W3C

[<http://www.w3.org/Style/XSL/>]

XSL-FO constitue l'aboutissement du groupe de travail du **W3C** sur les feuilles de style. C'est pour cette raison que le terme **XSL** fait parfois référence à **XSL-FO**, notamment dans les spécifications.

> Place dans l'écosystème

XSL-FO a été conçu pour émettre des documents de complexité simple à moyenne (*rapports, factures, documents techniques*) et n'est malheureusement pas en mesure de produire des documents typographiquement parfaits comme ceux issus des outils de publication professionnels comme **QuarkXPress** ou **InDesign**, ou générés via **LaTeX**.

📄 cf. Wikipedia

[https://en.wikipedia.org/wiki/XSL_Formatting_Objects]

Il existe également un document de travail pour permettre la création de documents paginés via **CSS**. Toutefois, ce document émis en 2013 n'a pas évolué depuis :

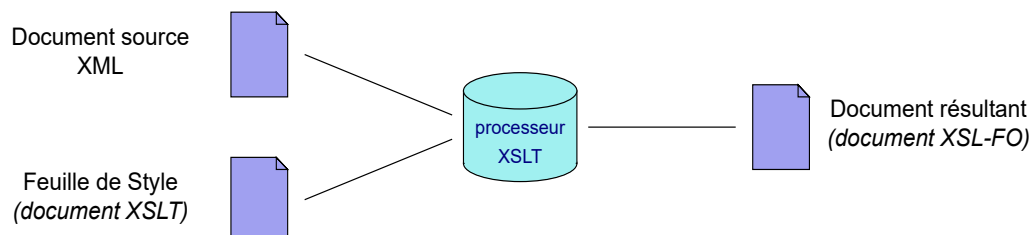
📄 CSS Paged Media Module Level 3

[<https://www.w3.org/TR/css3-page/>]

Attention toutefois à ne pas sous-estimer **XSL-FO** qui permet très facilement de générer des documents professionnels de qualité, à condition de maîtriser **XML** et **XSLT** et de ne pas viser la perfection typographique d'un ouvrage papier. Pour information, la version pdf de ce cours a été générée via XSL-FO.

11.1.2 Rappel des principes

XSL-FO est issu des besoins identifiés par le groupe de travail en charge de la définition de l'application de feuilles de styles pour **XML**. Le principe retenu pour la visualisation d'un document, consiste à le transformer à l'aide d'une feuille de style **XSLT** afin d'obtenir un document **XSL-FO** décrivant la mise en page :



Contrairement à **XHTML**, **XSL-FO** comprend la notion de **pages**, et permet de gérer des pages d'entêtes, de titre, des hauts de page, bas de page, la numérotation, des pages paires et impaires, *etc...*

Bien que rien dans le principe n'interdise l'idée qu'un navigateur puisse (*un jour*) visualiser directement les documents **XSL-FO**, leur exploitation est plus particulièrement pertinente sur support papier.

Là encore, rien n'interdirait que le format **XSL-FO** soit directement interprété par des imprimantes compatibles, comme c'est le cas pour **Postscript** par exemple. Malheureusement, il n'existe à l'heure actuelle (*fin 2016*) aucune imprimante de ce type.

Comment alors exploiter un document **XSL-FO** ?

- Il existe des applications spécialisées permettant de visualiser un document **XSL-FO** sur écran (*généralement utilisées en phase de développement*).
- D'autres applications permettent de traduire la syntaxe **XSL-FO** en un format d'impression (*pdf, pcl, postscript, rtf, svg, mif-framemaker...*) :

🔧 Antenna House Formatter V6

[<https://www.antennahouse.com/product/ahf63/index.html>]

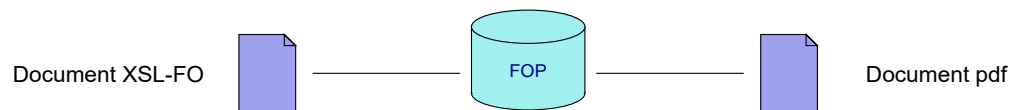
🔧 Apache FOP

[<http://xmlgraphics.apache.org/fop/>]

🔧 Render-X XEP engine

[<http://www.renderx.net/Content/tools/xep.html>]

FOP est une application de ce type, gratuite et open-source, développée et maintenue par la fondation Apache, qui permet l'impression de documents **XSL-FO** en les transformant en documents au format **pdf**



FOP est l'application utilisée pour obtenir la version pdf de ce cours.

11.1.3 Avertissement

XSL-FO est une application relativement complexe.

Les spécifications comportent plus de 400 pages. Certaines fonctionnalités, bien que simples à énoncer sont très difficiles à implémenter, et il n'existe pas d'implémentation exhaustive de l'ensemble des fonctionnalités décrites par les spécifications.

Par suite, ce cours est lui aussi loin d'être exhaustif et ne doit en aucun cas être considéré comme une information de référence sur **XSL-FO**. Le parti-pris a été choisi d'aborder cette technologie sous un aspect essentiellement pragmatique, à travers les fonctionnalités offertes par **FOP** V2.0.

11.2 Les éléments d'un document XSL-FO

11.2.1 Document XSL-FO

XSL-FO est une application compatible avec les espaces de noms **XML**. L'**URI d'espace de noms** réservé est "<http://www.w3.org/1999/XSL/Format>", et le préfixe le plus couramment employé est "**fo**".

L'élément racine d'un document **XSL-FO** s'appelle "**root**".

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  ...
</fo:root>
  
```

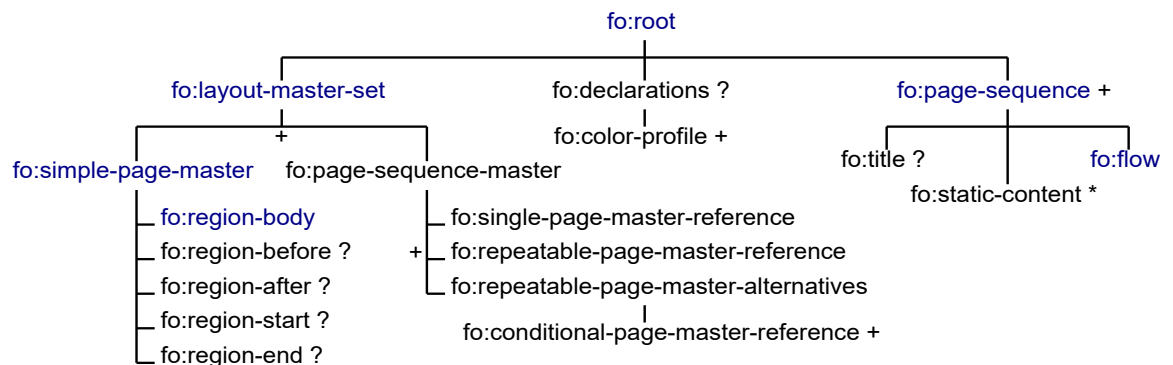

11.2.2 Structure d'un document XSL-FO

L'élément racine d'un document **XSL-FO** comporte toujours :

- une unique occurrence de l'élément **"fo:layout-master-set"**,
- un élément **"fo:declarations"** optionnel,
- un ou plusieurs éléments **"fo:page-sequence"** :

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    ...
  </fo:layout-master-set>
  <fo:page-sequence master-reference="...">
    ...
  </fo:page-sequence>
</fo:root>
```

La figure ci-dessous illustre la structure générale d'un document **XSL-FO** telle qu'imposée par les spécifications. Les éléments en couleur sont ceux qui permettent de constituer un document simple **"à moindre frais"**.



L'élément **"fo:layout-master-set"** définit un ou plusieurs modèles de pages (*pages de titre, pages courantes, paires, impaires, ...*), appelés **"page-masters"** qui décrivent le découpage (*entête avec le titre, zone de contenu, zone de bas de page, marges...*), ainsi que la taille et la position des différentes zones.

Les éléments **"fo:page-sequence"** mettent en place le contenu des pages, texte, et attributs de mise en forme.

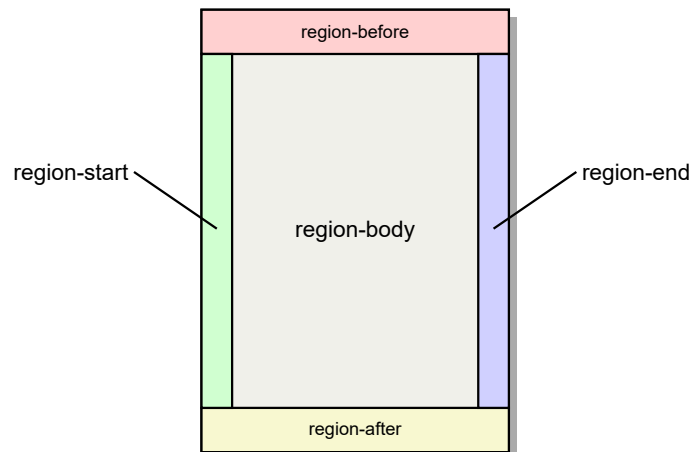
11.2.3 Modèle de page

L'élément **"fo:simple-page-master"** est utilisé par le processeur d'impression lors de la génération de pages de texte. Il décrit la géométrie des pages.

Une page peut être subdivisée en plusieurs zones : le corps de la page (*region-body*), l'entête (*region-before*), le bas de page (*region-after*), la marge gauche (*region-start*), la marge droite (*region-end*). La seule zone obligatoire est **"region-body"**.

N.B. Les zones portent ces noms assez étranges pour cause d'internationalisation. En effet, lorsque l'écriture se fait de droite à gauche par exemple, **"region-start"** se trouve à droite, contrairement à ce qui serait le cas d'une zone qui serait appelée **"region-left"**...

La figure suivante illustre la position respective de ces cinq zones, dans le cas le plus classique (*orientation portrait, écriture de gauche à droite*) :



L'exemple ci-dessous met en place les éléments structurels minimaux nécessaires à un document **XSL-FO** :

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="page-unique"
    page-height="29.7cm" page-width="21cm"
    margin-top="1.5cm" margin-bottom="2cm"
    margin-left="2.5cm" margin-right="1cm">
    <fo:region-body background-color="#CCCCCC"/>
  </fo:simple-page-master>
</fo:layout-master-set>

<fo:page-sequence master-reference="page-unique">
  <fo:flow flow-name="xsl-region-body">
    <fo:block>Hello World !</fo:block>
  </fo:flow>
</fo:page-sequence>
```

Tester l'exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/start/simple-page-master-ex1.html>]

Noter les attributs qui décrivent la géométrie de la page. Les noms des attributs, et les principes sous-jacents attachés aux propriétés qu'ils recouvrent sont très massivement dérivés de **CSS**.

Remarquer également comment l'élément *"fo:page-sequence"* référence le modèle de page à utiliser à l'aide de la valeur de l'attribut *"master-reference"*, ainsi que l'attribut *"flow-name"* de l'élément *"flow"* qui indique dans quelle zone se trouvera le texte concerné.

N.B. Les marges s'appellent bien *"margin-left"*, *etc...* et non pas *margin-start...* comme les zones, car elles ne sont pas effectuées par le sens de l'écriture.

11.2.4 Contenu informationnel

Chacun des éléments *"fo:page-sequence"* possède un unique élément *"fo:flow"* qui contient lui-même le texte qui sera affiché après pagination par le processeur d'impression :

```
<fo:flow flow-name="xsl-region-body">
  <fo:block>Hello World !</fo:block>
  ...
</fo:flow>
```

Comme illustré par l'exemple ci-dessus, le gros du texte est traditionnellement affiché dans la zone qui correspond au corps de la page.

Lorsque d'autres zones ont été définies, elles contiennent habituellement des informations répétitives d'une page à l'autre. Ces informations sont mises en place grâce à des éléments *"fo:static-content"* :

```
<fo:static-content flow-name="xsl-region-before" font-weight="bold">
  <fo:block text-align="center" space-before="0.5em" font-size="12pt">
    Exemple de document XSL-FO
  </fo:block>
</fo:static-content>

<fo:static-content flow-name="xsl-region-after" text-align="center">
  <fo:block space-before="0.5em">
    Document XSL-FO © Daniel Muller
  </fo:block>
</fo:static-content>
```

On peut observer sur cet exemple que **XSL-FO** a repris le concept d'**héritage des propriétés** introduit par **CSS** : les attributs de présentation (*marges, style, police, couleurs, ...*) sont transmis de père en fils dans l'arbre **XML**, ce qui évite d'attacher les attributs correspondants à chacun des éléments.

Les zones ci-dessus venant prendre place **dans les marges de la zone principale**, il faut par ailleurs donner au corps de la page des marges suffisantes pour abriter l'entête et le bas de page :

```
<fo:region-before extent="1.5cm"/>
<fo:region-after extent="1.5cm"/>
<fo:region-body margin-top="1.6cm" margin-bottom="1.6cm"/>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/start/page-sequence-ex2.html>]

N.B. Dans l'exemple ci-dessus on aurait tout aussi bien pu spécifier les marges à l'aide des propriétés "*space-before*" et "*space-after*" (*plutôt que margin-top et margin-bottom*)

Vérifier :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/start/page-sequence-ex3.html>]

11.2.5 Blocs de texte

Les éléments "*fo:flow*" et "*fo:static-content*" contiennent du texte formaté en blocs. Les blocs s'empilent dans la direction principale de progression du texte (*i.e. de haut en bas, conformément aux habitudes locales*).

Les éléments "*fo:block*" sont à rapprocher des **éléments blocs** (*block-level elements*) de **HTML**, et servent pour la mise en page des titres, des paragraphes, des légendes de tables et de figures, etc...

En général, chaque élément "*fo:block*" porte les propriétés de mise en forme qui lui sont spécifiques, et profite de celles qui sont "*mises en facteur*" par héritage :

```
<fo:flow flow-name="xsl-region-body" font-size="10pt"
  text-align="justify" start-indent="5pt" end-indent="5pt">
  <fo:block font-weight="bold" space-before="0.5em">
    L'enfance de Zéphyrin.
  </fo:block>
  <fo:block space-before="0.5em">
    C'est le 1er janvier, à minuit une seconde sexagésimale de temps moyen,
    que le jeune Brioché poussa ses premiers vagissements. A son baptême,
    il reçut les prénoms harmonieux, poétiques et distingués de Pancrace,
    Eusèbe, Zéphyrin, ce dont il parut se soucier comme un cloporte
    d'un ophicléide.
  </fo:block>
  <fo:block space-before="0.5em" font-style="italic">
    Extrait de : l'idée fixe du Savant Cosinus.
  </fo:block>
</fo:flow>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/start/block-ex1.html>]

N.B. On pourrait être tenté de reporter la propriété *"space-before"* commune à l'ensemble des éléments sur leur parent *"fo:flow"* de manière à les en faire hériter. Malheureusement *"space-before"* n'est pas une propriété héritée, contrairement à *"start-indent"* par exemple.

Remarque : *"fo:block"* n'est pas le seul élément bloc. Il en existe d'autres comme *"fo:table"* ou *"fo:list-block"* qui seront étudiés en leur temps.

11.2.6 Eléments texte

On peut souvent être amené à vouloir modifier la présentation du texte *"en ligne"*, sans pour autant changer de bloc. Cette fonctionnalité est assurée par l'élément *"fo:inline"* :

```
<fo:block space-before="0.5em" font-style="italic">
  Extrait de :
  <fo:inline font-weight="bold">
    l'idée fixe du Savant Cosinus
  </fo:inline>.
</fo:block>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/start/inline-ex1.html>]

11.3 Présentation des blocs

11.3.1 Alignement du texte

L'alignement des blocs de texte dans la direction perpendiculaire à la direction de progression des blocs (i.e. de gauche à droite dans notre cas) peut s'effectuer à l'aide des propriétés *"start-indent"* (gauche) et *"end-indent"* (droite) :

```
<fo:block start-indent="2em" end-indent="2em">
  ...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/text-align-ex1.html>]

N.B. La valeur par défaut est *"0"* : le texte est aligné sur les bords gauche et droit du bloc.

L'espacement entre blocs se règle préférentiellement par les propriétés *"space-before"* (haut) et *"space-after"* (bas). La valeur par défaut est *"0"* (pas d'espace supplémentaire).

```
<fo:block font-weight="bold" space-before="1em" space-after="1em">
  ...
</fo:block>
<fo:block space-before="0.5em">
  ...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/text-align-ex2.html>]

A noter que **XSL-FO** possède des règles complexes pour le calcul des espaces entre blocs, qui sont en première approximation déterminés de manière à satisfaire la plus importante des demandes entre la marge bas du bloc supérieur et la marge haut du bloc inférieur.

11.3.2 Bordures

Les bordures des blocs sont invisibles par défaut. Chacun des bords (*haut*, *bas*, *gauche*, *droite*) de chacun des blocs peut être rendu visible sous des formes diverses. Les propriétés relatives aux bordures sont la couleur, le style et la largeur.

> Propriétés des bordures

La couleur s'exprime à l'aide de la propriété **"border-color"** avec des valeurs conformes aux standards introduits par **CSS**, notation hexadécimale **"#RRGGBB"** ou fonction **"rgb(r,g,b)"**. La valeur par défaut est la couleur du fond.

La propriété **"border-width"** donne la largeur, qui peut prendre l'une des valeurs prédéfinies **"thin"**, **"medium"** ou **"thick"**, ainsi qu'une valeur numérique exprimée avec l'une des unités reconnues (**cm**, **mm**, **pt**, **pc**, **em**, **ex** ...). La valeur par défaut est **"medium"**.

Les styles exprimés par la propriété **"border-style"** sont prédéfinis. Les valeurs autorisées sont : **"none"** (pas de bordure), **"hidden"** (bordure cachée, effet identique au précédent), **"dotted"** (pointillés), **"dashed"** (pointillés longs), **"solid"** (cadre), **"double"** (cadre double), **"groove"** (cadre en relief rentrant), **"ridge"** (cadre en relief sortant), **"inset"** (ligne en relief rentrant) et **"outset"** (ligne en relief sortant). La valeur par défaut est **"none"**.

```
<fo:block border-color="#008800" border-width="0.3mm" border-style="solid">
...
</fo:block>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/border-ex1.html>

> Côtés individuels

Le réglage individuel des diverses bordures se fait à l'aide des propriétés dont le nom générique peut se noter **"border-x-y"**, où **"x"** identifie le côté concerné et **"y"** est l'une des propriétés vues ci-dessus : **"color"**, **"width"**, ou **"style"**.

L'identification du côté peut prendre l'une des valeurs **"top"** (haut), **"bottom"** (bas), **"left"** (gauche), **"right"** (droite), **"before"** (précédent, dans le sens de progression des blocs), **"after"** (suivant, dans le sens de progression des blocs), **"start"** (précédent, dans le sens de progression des caractères), ou **"end"** (suivant, dans le sens de progression des caractères).

```
<fo:block border-left-style="solid" border-left-color="#CC0000">
<!-- bloc accompagné d'un trait dans la marge gauche -->
</fo:block>

<fo:block border-bottom-style="solid" border-bottom-color="#000088">
<!-- souligné -->
</fo:block>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/border-ex2.html>

> Raccourcis typographiques

Conformément à ce qui avait déjà été introduit par **CSS**, **XSL-FO** permet de simplifier les spécifications relatives aux bordures grâce à des raccourcis syntaxiques.

Outre les attributs **"border-width"**, **"border-color"** et **"border-style"** introduits en début de section qui permettent de fixer d'un coup une propriété pour les quatre côtés, il existe des raccourcis permettant de fixer les trois propriétés **"width"**, **"style"** et **"color"** (dans cet ordre) pour un côté donné.

Ces attributs prennent alors la forme **"border-x"** (où **x** représente comme plus haut le nom d'un côté), avec la propriété super-raccourcie **"border"** qui permet de fixer d'un coup les trois propriétés des quatre côtés :

```
<fo:block border-bottom="thin solid #000088">
...
</fo:block>
<fo:block border="2pt double black">
...
</fo:block>
```

11.3.3 Couleur et motif du fond

La décoration des blocs ne s'arrête pas à leurs bordures. Il est également possible de colorer leur fond, ou de l'illustrer à l'aide d'une image.

> Couleur de fond

La couleur du fond est indiquée à l'aide de la propriété *"background-color"*. La valeur indiquée peut être une couleur (*au sens de CSS*) ou le mot-clé *"transparent"*. La valeur par défaut est *"transparent"*.

L'exemple suivant, en colorant le fond des paragraphes, permet de visualiser l'espace vertical entre blocs :

```
<fo:block space-before="0.5em" background-color="#CCCCCC">
...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/background-ex1.html>]

> Motif du fond

De manière tout à fait similaire à ce qui se fait dans le domaine du web, **XSL-FO** permet de paver le fond des blocs à l'aide d'une image. La propriété qui permet d'attacher l'image au bloc est *"background-image"*. Les valeurs possibles sont une **URL** ou le mot-clé *"none"*. La valeur par défaut est *"none"*.

```
<fo:block space-before="0.5em" background-image="papier-peint.gif">
...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/background-ex2.html>]

Le comportement par défaut consiste à paver le fond du bloc concerné en répétant l'image horizontalement et verticalement. Il est possible de contrôler ce comportement à l'aide de la propriété *"background-repeat"* qui peut prendre les valeurs *"repeat"*, *"repeat-x"*, *"repeat-y"*, et *"no-repeat"*. La valeur par défaut est *"repeat"*.

Lorsque l'image n'est pas répétée, il est logique de pouvoir la positionner. Ceci est assuré à l'aide des propriétés *"background-position-horizontal"* et *"background-position-vertical"*. Les valeurs possibles s'expriment en pourcent de la dimension correspondante du bloc, à l'aide d'une dimension munie d'une unité, ou à l'aide d'un mot clé parmi *"left"*, *"center"* et *"right"* (*position horizontale*) ou *"top"*, *"center"* et *"bottom"* (*position verticale*).

Une application possible des images de fond consiste à mettre un logo en *"signe d'eau"* sur le fond de la page. L'élément auquel appliquer l'image de fond est alors *"fo:region-body"* :

```
<fo:region-body background-image="logo.gif"
background-repeat="no-repeat"
background-position-horizontal="center"
background-position-vertical="center"/>
```

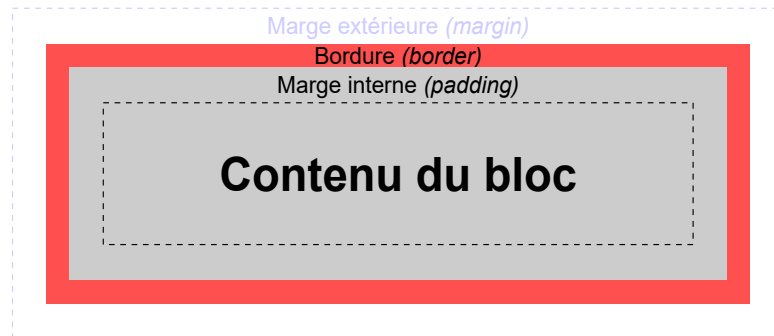
Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/background-ex3.html>]

11.3.4 Marges internes

Le comportement normal des blocs consiste à aligner le texte au plus près des bordures (*i.e. sans espace*). Lorsque ceux-ci comportent un fond coloré ou lorsque les bordures sont matérialisées, il peut être intéressant de disposer d'un marge dite *"interne"* entre le texte et la bordure.

De fait, **XSL-FO** s'appuie sur un modèle identique à celui introduit par **CSS** (*Cascading Style Sheets*) pour la gestion des marges et des bordures :



Comme pour **CSS**, les marges internes sont globalement fixées par la propriété *"padding"*. La valeur par défaut est *"0"*.

```
<fo:block background-image="image.gif" padding="0.5em">
...
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/padding-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/padding-ex1.html)

Attention à cette propriété qui, conformément aux recommandations, résulte en un comportement initialement surprenant (*comme illustré par l'exemple précédent*).

Pour remédier à l'effet observé, on peut compenser la marge interne en réglant les propriétés *"start-indent"* et *"end-indent"* :

```
<fo:block space-before="0.5em" background-image="image.gif"
padding="0.5em" start-indent="0.5em" end-indent="0.5em">
...
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/padding-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/padding-ex2.html)

N.B. Les marges internes peuvent également se régler individuellement à l'aide des propriétés *"padding-x"*, où *"x"* prend l'une des valeurs *"top"*, *"bottom"*, *"left"*, *"right"*, *"before"*, *"after"*, *"start"* ou *"end"*, avec les significations habituelles.

11.3.5 Inclusion de blocs

Les blocs sont susceptibles de contenir du texte, mais aussi d'autres blocs.

Cette propriété permet de mettre en facteur un certain nombre de propriétés qui peuvent être héritées par les blocs internes, mais aussi d'obtenir des effets de présentation comme par exemple celui qui consiste à inclure plusieurs blocs consécutifs dans une même boîte :

```
<fo:block border="0.3mm #000088 solid" space-before="1em"
padding="0.5em" start-indent="0.5em" end-indent="0.5em">
<fo:block>
...
</fo:block>
<fo:block>
...
</fo:block>
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/include-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/block/include-ex1.html)

11.4 Présentation du texte

11.4.1 Polices de caractères

Les propriétés qui permettent de déterminer la police et la forme des caractères sont calquées sur les propriétés **CSS** équivalentes.

> font-family

La propriété **"font-family"** sélectionne la police de caractère à partir de son nom. Comme pour **CSS**, il est possible de spécifier une liste de polices par ordre de priorité décroissante. La valeur par défaut de cette propriété dépend du processeur considéré et de sa version (*cf. système d'exploitation*).

Outre les polices classiques, il est possible de spécifier une **police générique** (*en général en fin de liste*). Les polices génériques sont **"serif"**, **"sans-serif"**, **"cursive"**, **"fantasy"**, et **"monospace"**.

```
<fo:page-sequence master-reference="simple"
                  font-family="Georgia, Times New Roman, serif">
...
</fo:page-sequence>
```

Noter ci-dessus comment la police a été attachée à l'élément **"page-sequence"**, qui la transmet en héritage à tous ses enfants, assurant par là-même une police unique pour l'ensemble du document.

N.B. Le processeur d'impression **FOP** ne connaît pas les polices génériques **"cursive"** et **"fantasy"**. Toutefois, cette limitation n'est pas très grave. En effet, contrairement aux pages web, les documents **XSL-FO** sont en général exploités dans un environnement parfaitement contrôlé.

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/font-ex1.html>]

> font-size

"font-size" permet de fixer la taille de la police. Celle-ci peut être spécifiée sous forme absolue, relative, sous forme d'une dimension, ou d'un pourcentage.

Les valeurs absolues sont données sous forme de mots-clés. Les mots-clés autorisés sont : **"xx-small"**, **"x-small"**, **"small"**, **"medium"**, **"large"**, **"x-large"** et **"xx-large"**.

Les valeurs relatives permettent de modifier la taille de la police par rapport à celle de l'élément parent (*i.e. la taille "normale"*). Les mots-clés autorisés sont **"smaller"** et **"larger"**.

La taille de la police peut être donnée directement à l'aide d'un nombre associé à une unité comme **"10pt"**.

Une taille exprimée en pourcent est comptée par rapport à la taille **"normale"**, c'est à dire celle héritée de l'élément parent.

```
<fo:block font-size="12pt">
...
</fo:block>
```

La valeur par défaut de cette propriété est **"medium"**.

N.B. Contrairement à ce qui peut être préconisé pour le web, l'accessibilité d'un document destiné à être imprimé n'est pas forcément affectée par l'usage de dimensions absolues.

> font-style

Les valeurs autorisées pour cette propriété sont : **"normal"**, **"italic"**, **"oblique"** et **"backslant"**. Toutefois, le résultat obtenu dépend de la disponibilité de la police.

```
<fo:block font-style="italic">
...
</fo:block>
```

Outre **"normal"**, la valeur la plus couramment utilisée est **"italic"**. La valeur par défaut est **"normal"**.

> font-weight

Cette propriété admet des valeurs absolues, relatives ou numériques. Toutefois, là encore, le résultat obtenu est lié à la disponibilité de la police. L'expérience montre que les valeurs à retenir sont *"normal"* (valeur par défaut), et *"bold"*.

```
<fo:block font-weight="bold">
...
</fo:block>
```

> Autres propriétés

Les spécifications **XSL-FO** prévoient d'autres propriétés, déjà introduites par **CSS**, qui permettent d'agir sur les caractéristiques de la police de caractères. Ces propriétés sont : *"font-stretch"*, *"font-size-adjust"* et *"font-variant"*.

N.B. Ces propriétés ne sont pas supportées par le processeur d'impression **FOP**.

11.4.2 Rendu des caractères

Le rendu des caractères est affecté par les propriétés décrites ci-dessous.

> letter-spacing

La modification de l'espacement des caractères permet de donner plus ou moins de *"volume"* à certains textes comme des titres ou des légendes.

L'avantage de cette propriété est qu'elle ne modifie pas la forme des caractères eux-mêmes, ce qui fait que son fonctionnement ne dépend pas de l'éventuelle disponibilité d'une police. Sa mise en oeuvre est d'autre part plus facile pour les processeurs d'impression qu'une propriété comme *"font-stretch"*.

Les valeurs autorisées sont le mot-clé *"normal"* ou une dimension munie d'une unité qui indique le supplément d'espacement par rapport à la normale. La valeur par défaut est *"normal"*.

```
<fo:block font-family="Haettenschweiler" letter-spacing="0.2em">
...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/decoration-ex1.html>]

N.B. Il est très fortement suggéré de spécifier l'espacement des caractères à l'aide de l'unité *"em"*, proportionnelle à la taille de la police...

> text-decoration

Les valeurs autorisées pour la propriété *"text-decoration"* sont : *"none"* (texte normal), *"underline"* (souligné), *"overline"* (surligné) et *"line-through"* (barré). La valeur par défaut est *"none"*.

```
<fo:inline text-decoration="underline">
...
</fo:inline>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/decoration-ex2.html>]

> color

La couleur du texte est spécifiée à l'aide de la propriété *"color"*. Les valeurs autorisées sont conformes aux standards introduits par **CSS**, notation hexadécimale *"#RRGGBB"* ou fonction *"rgb(r,g,b)"*. La valeur par défaut dépend du processeur.

```
<fo:block font-size="12pt" color="#000088" space-after="1.0em">
...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/decoration-ex3.html>]

11.4.3 Alignement du texte

> text-align

Au sein de la boîte virtuelle qui délimite les paragraphes, le texte peut être aligné à gauche, à droite, centré ou justifié. Les valeurs correspondantes de la propriété *"text-align"* sont : *"start"* (début de ligne), *"end"* (fin de ligne), *"left"* (gauche), *"right"* (droite), *"center"* (centré) et *"justify"* (justifié). La valeur par défaut est *"start"*.

```
<fo:block text-align="justify">
...
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/justify-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/justify-ex1.html)

> text-indent

La propriété *"text-indent"* donne l'indentation de la première ligne de texte d'un paragraphe. Les valeurs autorisées sont une dimension munie d'une unité, ou une valeur en pourcent de la largeur du bloc.

La valeur spécifiée peut être négative. Dans ce cas, la première ligne n'est pas en retrait mais en avant. La valeur par défaut est *"0pt"*.

```
<fo:block text-indent="2em">
...
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/justify-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/justify-ex2.html)

11.4.4 Alignement vertical

> line-height

Le réglage de l'interligne intra-paragraphe s'effectue à l'aide de la propriété *"line-height"*. Les valeurs possibles sont le mot-clé *"normal"*, une dimension munie d'une unité ou un pourcentage relatif à la taille de la police de caractères. La valeur par défaut est *"normal"*.

```
<fo:block line-height="1em">
...
</fo:block>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/vertical-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/vertical-ex1.html)

> vertical-align

La propriété *"vertical-align"* permet de déplacer verticalement une séquence de texte en cours de ligne. Les valeurs les plus utiles sont : *"baseline"* (normal), *"sub"* (indice), *"super"* (exposant), une dimension affectée d'une unité ou une valeur en pourcent relative à la valeur de l'interligne. La valeur par défaut est *"baseline"*.

```
ax<fo:inline vertical-align="super" font-size="80%">2</fo:inline> + bx + c = 0
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/vertical-ex2.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/vertical-ex2.html)

11.4.5 Traitement des espaces

> white-space-collapse

Le comportement naturel d'une application XML est en général de ne pas tenir compte des espaces surnuméraires, et de travailler avec des chaînes de caractères compactées. Cette propriété permet de contrôler cet aspect du comportement d'un processeur d'impression.

Les valeurs autorisées sont : **false** et **true**. La valeur par défaut est **"true"**.

```
<fo:block white-space-collapse="false">
...
</fo:block>
```

> wrap-option

Lorsque le texte disponible dans un bloc ne peut tenir sur une seule ligne, un processeur d'impression introduit automatiquement des sauts de ligne aux endroits adéquats. La propriété **"wrap-option"** permet éventuellement d'empêcher ceci.

Les valeurs autorisées sont **"wrap"** et **"no-wrap"**. La valeur par défaut est **"wrap"**.

```
<fo:block wrap-option="no-wrap">
...
</fo:block>
```

N.B. Cette propriété est à manier avec précautions, car elle peut facilement conduire à des lignes de texte qui dépassent la largeur de la page. Un contrôle fin du comportement du processeur concernant le contenu de boîtes relativement petites en taille comme des cellules de tableau, semble être son domaine de prédilection.

> line-feed-treatment

Dans l'optique de créer un bloc de texte préformaté (*comme l'élément PRE en HTML*), il faut tenir compte de tous les espaces, ne pas introduire de saut de ligne supplémentaires (*ce que permettent les propriété ci-dessus*), mais aussi respecter les sauts de ligne contenus dans le texte du bloc.

Cette dernière fonctionnalité est apportée par la propriété **"line-feed-treatment"**. Les valeurs autorisées sont : **"ignore"**, **"preserve"**, **"treat-as-space"** et **"treat-as-zero-width-space"**. La valeur par défaut est **"treat-as-space"**.

```
<fo:block linefeed-treatment="preserve">
...
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/space-ex1.html>]

11.4.6 Ligne de caractères

> fo:leader

L'élément **"fo:leader"** permet de tracer une ligne formée de caractères répétés, comme dans une table des matières, ou pour obtenir un séparateur horizontal. Cet élément est muni d'une série de propriétés qui conditionnent son comportement et son rendu visuel.

Il s'agit d'un **élément texte** (*inline element*) dont le comportement consiste a priori à remplir l'espace disponible dans la ligne courante.

> leader-pattern

La propriété **"leader-pattern"** permet de spécifier la nature de la ligne à tracer. Les valeurs possibles sont : **"space"**, **"rule"**, **"dots"** ou **"use-content"**. La valeur par défaut est **space**.

L'espace disponible peut être rempli de diverses façons :

- par des espaces (*valeur "space"*),
- par une ligne (*valeur "rule"*),
- par des pointillés (*valeur "dots"*),
- par une séquence de caractères arbitraire (*valeur "use-content"*).

```
<fo:block text-align-last="justify" linefeed-treatment="preserve">
  Chapitre premier <fo:leader leader-pattern="dots" /> p. 12
  Chapitre 2 <fo:leader leader-pattern="dots" /> p. 24
  Chapitre trois <fo:leader leader-pattern="dots" /> p. 3564
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/leader-ex1.html>]

N.B. Pour que "*fo:leader*" fonctionne avec **FOP** il est nécessaire, comme illustré ci-dessus, que le paragraphe contenant les leaders possède un attribut "*text-align-last*" valant "*justify*".

 FOP FAQ

[<https://xmlgraphics.apache.org/fop/faq.html#leader-expansion>]

> leader-alignment

Cette propriété s'applique lorsque "*leader-pattern*" vaut "*dots*" ou "*use-content*". Elle permet d'aligner les motifs entre deux lignes générées par "*fo:leader*".

La valeur "*reference-area*" aligne les motifs sur le bord du bloc parent, la valeur "*page*" sur le bord de la page. La valeur par défaut est "*none*" (*pas d'alignement*).

```
...<fo:leader leader-pattern="dots" leader-alignment="reference-area"/>...
```

N.B. Cette propriété n'est pas (*plus*) supportée par le processeur d'impression **FOP**.

> leader-pattern-width

En imposant la largeur totale d'un motif, la propriété "*leader-pattern-width*" permet en fait de régler l'espace entre deux motifs consécutifs. Elle s'applique lorsque "*leader-pattern*" vaut "*dots*" ou "*use-content*".

Les valeurs possibles sont "*use-font-metrics*", une longueur munie d'une unité, ou un pourcentage de la largeur de la boîte parent. La valeur par défaut est "*use-font-metrics*".

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/leader-ex3.html>]

> rule-style

Lorsque "*leader-pattern*" vaut "*rule*", le style de la ligne est indiqué par la propriété "*rule-style*". Les valeurs possibles sont : "*none*" (*pas de ligne - intérêt limité ?*), "*dotted*" (*ligne de tirets courts*), "*dashed*" (*ligne de tirets longs*), "*solid*" (*ligne continue*), "*double*" (*ligne double*), "*groove*" (*ligne dont la moitié inférieure est blanche*), "*ridge*" (*ligne dont la moitié supérieure est blanche*). La valeur par défaut est "*solid*".

```
... <fo:leader leader-pattern="rule" rule-style="double"/> ...
```

> rule-thickness

La largeur de la ligne tracée lorsque "*leader-pattern*" vaut "*rule*", est réglable grâce à la propriété "*rule-thickness*". La valeur doit être une dimension munie d'une unité. La valeur par défaut est "*1pt*".

```
... <fo:leader leader-pattern="rule" rule-thickness="3pt"/> ...
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/inline/leader-ex5.html>]

11.5 Listes

11.5.1 Modèle des listes

> Structure

Le modèle des listes implémenté par **XSL-FO** repose sur quatre éléments distincts qui, une fois correctement emboîtés et configurés, permettent de mettre en place l'ensemble des listes habituellement rencontrées, et en particulier les listes **HTML** "*UL*", "*OL*", et "*DL*".

Si on devait décrire la structure d'une liste **XSL-FO** en utilisant la syntaxe **DTD**, on écrirait :

```
<!ELEMENT fo:list-block      (fo:list-item+)>
<!ELEMENT fo:list-item      (fo:list-item-label, fo:list-item-body)>
<!ELEMENT fo:list-item-label (%block;)+>
<!ELEMENT fo:list-item-body (%block;)+>
```

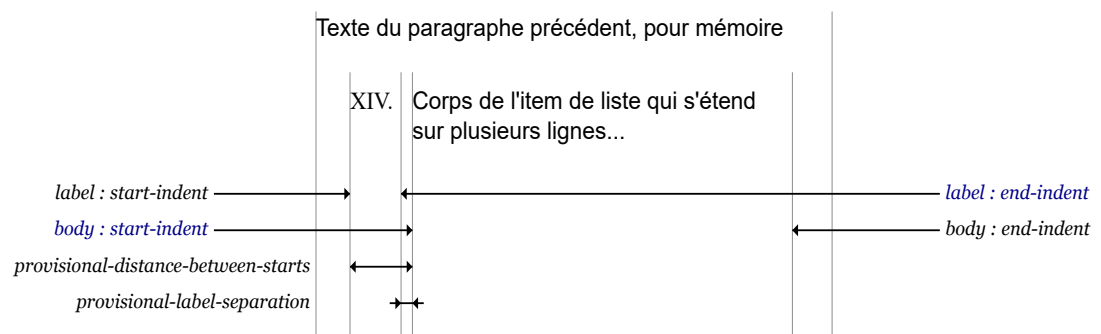
Autrement dit, une liste est constituée d'un élément **"fo:list-block"**, contenant une liste d'items matérialisés par des éléments **"fo:list-item"**, comportant chacun une entête **"fo:list-item-label"** et un corps **"fo:list-item-body"**.

Sémantiquement, l'entête d'item de liste correspond à la puce, le numéro, ou la définition de la liste, et contient techniquement un bloc, voire une liste de blocs (*cas inhabituel*).

Le corps de l'item est classique et peut contenir un ou plusieurs blocs d'information.

> Configuration

Le réglage fin de l'allure visuelle désiré pour la liste nécessite de comprendre les interactions entre les marges de l'entête et du corps d'un item de liste :



Par défaut, les marges **"start-indent"** et **"end-indent"** de l'entête et du corps d'item de liste sont alignés sur ceux des paragraphes normaux. Si la liste doit être mise en retrait par rapport à eux, il est loisible de modifier les propriétés **"start-indent"** de l'entête, et **"end-indent"** du corps.

La géométrie de la liste impose par contre obligatoirement de fixer de manière appropriée les propriétés **"start-indent"** du corps et **"end-indent"** de l'entête.

XSL-FO préconise de fixer ces valeurs grâce à deux fonctions, qui calculent les valeurs requises à partir des deux propriétés **"provisional-distance-between-starts"** et **"provisional-label-separation"**. Ces deux grandeurs sont des propriétés de la liste, et sont respectivement initialisées par défaut avec les valeurs **"24pt"** et **"6pt"**.

```
<fo:list-block provisional-label-separation="2pt"
               provisional-distance-between-starts="8pt">
  <fo:list-item>
    <fo:list-item-label end-indent="label-end()">
      <fo:block>1. </fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()">
      <fo:block>2 cuillères à soupe de matignon</fo:block>
    </fo:list-item-body>
  </fo:list-item>
  ...
</fo:list-block>
```

11.5.2 Liste à puces

Pour construire une liste à puces à l'aide du modèle de listes **XSL-FO**, le bloc d'entête d'item de liste contiendra simplement la puce, alors que le corps comprendra le texte de l'item de liste.

De fait, l'exercice le plus laborieux consistera sans doute à découvrir un caractère satisfaisant pouvant servir de puce, disponible dans la police de caractères désirée.

Voici un exemple d'item de liste à puces :

```
<fo:list-item>
  <fo:list-item-label end-indent="label-end()">
    <fo:block font-size="300%" line-height="12pt">
      <fo:inline>&#xB7;</fo:inline>
    </fo:block>
  </fo:list-item-label>
  <fo:list-item-body start-indent="body-start()">
    <fo:block>2 cuillères à soupe de matignon</fo:block>
  </fo:list-item-body>
</fo:list-item>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/list/ul-ex1.html>]

11.5.3 Liste ordonnée

Les listes ordonnées se construisent de manière tout à fait similaire aux listes à puce.

La difficulté consiste à gérer le fait que la longueur des entêtes est susceptible de varier en fonction du nombre de caractères du numéro de l'item. Il suffit pour cela de prévoir une largeur suffisante, et d'aligner les labels à droite dans le bloc qui leur est réservé :

```
<fo:list-block provisional-label-separation="5pt"
  provisional-distance-between-starts="3em">
  ...
  <fo:list-item>
    <fo:list-item-label end-indent="label-end()" text-align="end">
      <fo:block>3.</fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()">
      <fo:block>
        Placer les bols dans un four préalablement chauffé à 200°C.
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
  ...
</fo:list-block>
```

Exemple complet :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/list/ol-ex1.html>]

11.5.4 Liste descriptive

Les listes descriptives posent un problème particulier si on désire que l'entête puisse avoir une longueur relativement importante.

L'astuce consiste d'une part à ne pas réduire la marge droite *"end-indent"* de l'entête, et d'autre part à positionner la description elle-même une ligne plus bas :

```
<fo:list-item space-before="0.5em">
  <fo:list-item-label><!-- pas de modification de end-indent -->
    <fo:block font-weight="bold">
      Gratinée lyonnaise
    </fo:block>
  </fo:list-item-label>
  <fo:list-item-body start-indent="body-start()">
    <fo:block margin-top="1.2em">
      Bol à soupe allant au four.
    </fo:block>
  </fo:list-item-body>
</fo:list-item>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/list/dl-ex1.html>

11.5.5 Listes imbriquées

Les listes peuvent évidemment être imbriquées. Ceci revient à retrouver un (voire plusieurs) éléments *"fo:list-block"* comme contenu d'un élément *"fo:list-item-body"* :

```
<!-- liste principale -->
<fo:list-block space-before="0.5em" provisional-label-separation="5pt"
  provisional-distance-between-starts="2em">
  <fo:list-item>
    <fo:list-item-label end-indent="label-end()" text-align="end">
      <fo:block font-weight="bold">A.</fo:block>
    </fo:list-item-label>
    <fo:list-item-body start-indent="body-start()">
      <!-- premier item de liste principale -->
      <fo:block font-weight="bold">
        Définitions :
      </fo:block>

      <!-- liste descriptive imbriquée -->
      <fo:list-block space-before="0.5em" provisional-label-separation="5pt"
        provisional-distance-between-starts="1em">
        ...
      </fo:list-block>

    </fo:list-item-body>
  </fo:list-item>

  <!-- autres items de liste principale -->
  ...
</fo:list-block>
```

Exemple complet :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/list/embed-ex1.html>

11.6 Tables

11.6.1 Modèle des tables

Le modèle des tables adopté par **XSL-FO** s'appuie fortement sur celui de **HTML**, plus précisément décrit par **CSS2**. Dans ce modèle une table est essentiellement constituée de lignes, elles-mêmes constituées de cellules.

L'élément principal s'appelle **fo:table**, et contient un (ou plusieurs) éléments **fo:table-body**. Le corps de table est constitué de lignes **fo:table-row**, comportant elles-même des cellules **fo:table-cell**.

```
<fo:table text-align="center" space-before="1em">
  <fo:table-body>
    <fo:table-row>
      <fo:table-cell><fo:block>Nabuchodonosor</fo:block></fo:table-cell>
      <fo:table-cell><fo:block>16 litres</fo:block></fo:table-cell>
      <fo:table-cell><fo:block>20</fo:block></fo:table-cell>
    </fo:table-row>
  </fo:table-body>
</fo:table>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/principe-ex1.html>]

> Description explicite de la largeur des colonnes

Les tables disposent d'une propriété nommée **table-layout** dont la valeur peut être **auto** ou **fixed**.

La valeur par défaut est **auto**. Elle signifie que le processeur d'impression doit calculer automatiquement la largeur des colonnes de la table en fonction du contenu des cellules (*comme le font les navigateurs HTML par exemple*).

La valeur **fixed** signifie que la largeur des colonnes est explicitement indiquée dans le document. Cette information est mise en place à l'aide d'éléments **fo:table-column** :

```
<fo:table table-layout="fixed" text-align="center" space-before="1em">
  <fo:table-column column-width="3cm"/>
  <fo:table-column column-width="3cm"/>
  <fo:table-column column-width="3cm"/>
  <fo:table-body> . . . </fo:table-body>
</fo:table>
```

Lorsque plusieurs colonnes consécutives ont la même largeur, il est possible de ne pas les répéter, en précisant le nombre de colonnes identiques à l'aide de la propriété **number-columns-repeated** :

```
<fo:table table-layout="fixed" text-align="center" space-before="1em">
  <fo:table-column column-width="3cm" number-columns-repeated="3"/>
  <fo:table-body> . . . </fo:table-body>
</fo:table>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/principe-ex2.html>]

Pour obtenir des colonnes proportionnelles (*comme avec l'élément COL et la notation 1*, 2*, 3* en HTML*), il est possible de calculer leur largeur à l'aide de la fonction **proportional-column-width()** :

```
<fo:table table-layout="fixed" width="10cm">
  <fo:table-column column-width="proportional-column-width(2)"/>
  <fo:table-column column-width="proportional-column-width(1)"/>
  <fo:table-column column-width="proportional-column-width(3)"/>
  <fo:table-body> . . . </fo:table-body>
</fo:table>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/principe-ex3.html>]

Noter toutefois que lorsque *"table-layout"* vaut *"fixed"*, ceci implique comme ci-dessus de spécifier la largeur de la table à l'aide de la propriété *"width"*.

N.B. Le processeur d'impression **FOP** n'implémente pas la valeur *"auto"* de la propriété *"table-layout"*. Il impose systématiquement une description explicite des colonnes.

11.6.2 Visualisation des tables

Come pour tous les éléments **XSL-FO** (*Formatting Objects*), les bordures de tables ainsi que les montants (*cf. bordures de cellules*) ne sont pas visibles par défaut.

Pour les rendre visibles, il faut jouer sur les propriétés des bordures des cellules ainsi que de la table elle-même.

```
<fo:table table-layout="fixed" border="0.5pt solid #000088">
  <fo:table-column column-width="3.3cm" number-columns-repeated="3"/>
  <fo:table-body start-indent="0pt">
    <fo:table-row>
      <fo:table-cell border-bottom="0.5pt solid #000088" padding="2pt 5pt">
        <fo:block>Nom</fo:block>
      </fo:table-cell>
      ...
    </fo:table-row>
    ...
  </fo:table-body>
</fo:table>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/border-ex1.html>]

11.6.3 Position horizontale d'une table

N.B. Les limitations citées dans cette section , sont spécifiques au processeur d'impression **FOP**.

> Positionnement horizontal

FOP implémente le positionnement des tables de manière très imparfaite. Outre le fait qu'il est incapable de calculer lui-même les largeurs de colonnes (*cf. table-layout="auto"*), le positionnement horizontal d'une table est assez difficile à obtenir.

En effet, une table vient normalement se positionner à gauche le long de la marge de la page. Pour la décaler, on peut penser à utiliser les propriétés *"start-indent"* ou *"margin-left"* :

```
<fo:table table-layout="fixed" text-align="center" space-before="1em"
  width="9cm" margin-left="5cm" border="1px solid">
  . . .
</fo:table>
```

Résultat obtenu :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/position-ex1.html>]

Dans ce cas, **FOP** émet des erreurs et on obtient un résultat inattendu, visiblement dû au fait que le décalage imposé par la marge gauche est *"hérité"* par les cellules. Or si les spécifications indiquent que la valeur de *"start-indent"* est transmise par héritage, ce n'est pas le cas de *"margin-left"*.

☞ XSL start-indent

[<https://www.w3.org/TR/xsl/#start-indent>]

☞ XSL margin-left

[<https://www.w3.org/TR/xsl/#margin-left>]

La solution consiste à s'assurer que la marge gauche des cellules est nulle :

```
<fo:table-cell text-align="left" margin-left="0">
  <fo:block>Salmanazar</fo:block>
</fo:table-cell>
```

Résultat :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/position-ex2.html>]

> Centrage horizontal

Pour centrer une table horizontalement dans la page, la méthode **FOP** consiste à inclure la table en question dans une autre table, possédant trois colonnes, dont les colonnes latérales sont proportionnelles :

```
<fo:table table-layout="fixed" width="100%">
  <fo:table-column column-width="proportional-column-width(1)"/>
  <fo:table-column column-width="9cm"/>
  <fo:table-column column-width="proportional-column-width(1)"/>
  <fo:table-body>
    <fo:table-row>
      <fo:table-cell><fo:block/></fo:table-cell><!-- noter la cellule vide -->
      <fo:table-cell>
        <!-- table affichée -->
        <fo:table table-layout="fixed" border="1px solid">
          </fo:table>
        </fo:table-cell>
      <fo:table-cell><fo:block/></fo:table-cell><!-- noter la cellule vide -->
    </fo:table-row>
  </fo:table-body>
</fo:table>
```

Résultat :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/position-ex3.html>]

Dans l'exemple ci-dessus on fixe la largeur de la table, et on adapte les marges latérales en fonction de l'espace restant. Il est également possible de raisonner en fixant la largeur des marges et en laissant l'espace restant à la table :

```
<fo:table table-layout="fixed" width="100%">
  <fo:table-column column-width="2cm"/>
  <fo:table-column column-width="from-parent(width) -4cm"/>
  <fo:table-column column-width="2cm"/>
  <fo:table-body>
    <fo:table-row>
      <fo:table-cell><fo:block/></fo:table-cell><!-- noter la cellule vide -->
      <fo:table-cell>
        <!-- table affichée -->
        <fo:table table-layout="fixed" border="1px solid">
          </fo:table>
        </fo:table-cell>
      <fo:table-cell><fo:block/></fo:table-cell><!-- noter la cellule vide -->
    </fo:table-row>
  </fo:table-body>
</fo:table>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/position-ex4.html>]

Remarquer ci-dessus l'usage de la fonction **XSL-FO "from-parent()"** qui renvoie la valeur d'une propriété de l'élément parent.

Noter également que la valeur de l'attribut **"column-width"** de la colonne centrale est **"calculée"** (cf. *présence d'une fonction et d'un opérateur*). Une approche similaire a été mise en oeuvre pour la définition de la géométrie des items de listes. Ceci est une caractéristique de **XSL-FO** qui offre cette possibilité de manière générique pour tous ses attributs.

11.6.4 Tables complexes

La mise en place de cellules couvrant plusieurs lignes ou plusieurs colonnes se fait à l'aide des propriétés *"number-columns-spanned"* et *"number-rows-spanned"* de l'élément *"fo:table-cell"*.

```
...
<fo:table-row>
  <fo:table-cell><fo:block>1</fo:block></fo:table-cell>
  <fo:table-cell><fo:block>2</fo:block></fo:table-cell>
  <fo:table-cell><fo:block>3</fo:block></fo:table-cell>
  <fo:table-cell number-rows-spanned="2"><fo:block>=</fo:block></fo:table-cell>
</fo:table-row>
<fo:table-row>
  <fo:table-cell number-columns-spanned="2"><fo:block>0</fo:block></fo:table-cell>
  <fo:table-cell><fo:block>.</fo:block></fo:table-cell>
</fo:table-row>
...
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/colspan-ex1.html>]

Noter au passage dans l'exemple ci-dessus le rendu des montants de table *"larges"* obtenus grâce à la valeur *"separate"* de la propriété *"border-collapse"*, et de la largeur de montant fixée par *"border-separation"* :

```
<fo:table table-layout="fixed" width="5cm" border="0.5pt solid #000088"
  border-separation="3pt" border-collapse="separate" >
  ...
</fo:table>
```

11.6.5 Entête et bas de table

La mise en page des tables inclut la possibilité de spécifier des informations présentes en haut et en bas de table, et qui seront automatiquement rappelées en haut et en bas de chacune des pages, dans le cas où la table serait éclatée sur plusieurs pages.

Ces informations sont à inclure dans des éléments *"fo:table-header"* et *"fo:table-footer"* qui s'intègrent au même niveau que le corps de la table *"fo:table-body"* et admettent le même contenu :

```
<fo:table-header>
  <fo:table-row>
    <fo:table-cell number-columns-spanned="2" color="#880000">
      <fo:block>Introduction à XML</fo:block>
    </fo:table-cell>
  </fo:table-row>
</fo:table-header>
```

```
<fo:table-footer>
  <fo:table-row>
    <fo:table-cell number-columns-spanned="2" font-size="8pt">
      <fo:block>
        <xsl:text>Cours "</xsl:text>
        <xsl:value-of select="../@titre"></xsl:value-of>
        <xsl:text>" © 2004 Daniel Muller</xsl:text>
      </fo:block>
    </fo:table-cell>
  </fo:table-row>
</fo:table-footer>
```

Exemple avec entêtes et bas de tables :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/table/header-ex1.html>]

N.B. En principe, lorsqu'une même table contient plusieurs occurrences de *"fo:table-header"* ou de *"fo:table-footer"*, le comportement résultant est laissé à l'appréciation du processeur d'impression. Dans le cas de **FOP**, une table ne peut comporter qu'une seule instance de chacun de ces éléments.

N.B. XSL-FO prévoit également un mécanisme de table avec légende basé sur les éléments *"fo:table-and-caption"* et *"fo:table-caption"* qui ne sont pas supportés par **FOP**. Comme illustré par l'exemple ci-dessus, les entêtes et/ou bas de table peuvent constituer une alternative intéressante à cette lacune.

11.7 Graphiques

11.7.1 Graphiques externes

Comme **HTML**, **XSL-FO** permet d'intégrer des images dans une page à l'aide d'un élément nommé *"fo:external-graphic"*.

L'**URL** de la ressource est classiquement spécifiée via une propriété nommée *"src"*, charge au processeur de la récupérer lors de la construction de la page.

```
<fo:block>
  <fo:external-graphic src="url(VGE.jpg)" />
</fo:block>
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/image/external-ex1.html>

Les formats d'image reconnus dépendent bien sûr du processeur d'impression.

Le processeur **FOP** par exemple, supporte de manière native les formats : *"BMP"* (Microsoft Windows Bitmap), *"EPS"* (Encapsulated Postscript), *"GIF"* (Graphics Interchange Format), *"JPEG"* (Joint Photographic Experts Group), et *"TIFF"* (Tag Image Format File). Il peut être également configuré pour les formats *"PNG"* (Portable Network Graphic) et *"SVG"* (Scalable Vector Graphics).

> Taille de l'image

La taille de l'image obtenue dans le cas d'une image bitmap dépend du nombre de pixels, et de la résolution par défaut adoptée par le processeur d'impression qui est par exemple de **72 dpi** pour **FOP**.

Cette taille peut être modifiée grâce aux propriétés *"content-width"* et *"content-height"* de l'élément *"fo:external-graphic"*. Les valeurs de ces propriétés doivent être munies d'une unité :

```
<fo:external-graphic content-width="6cm" src="url(VGE.jpg)" />
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/image/external-ex2.html>

Lorsqu'une seule dimension est spécifiée, la deuxième dimension est modifiée de sorte à ne pas étirer l'image. Lorsque deux dimensions sont indiquées, l'image peut être déformée.

> Emplacement de l'image

Comme en **HTML**, l'image est un élément directement inséré dans le flot du texte (*inline element*). Cette propriété est en général mise à profit pour la génération de puces, ou autres pseudo-caractères du même type.

```
Les images ne servent pas uniquement de puces. Elles peuvent s'insérer
<fo:external-graphic src="url(triangle-right.gif)" />n'importe
où<fo:external-graphic src="url(triangle-left.gif)" />
dans le texte !
```

Exemple :

<http://dmolinarius.github.io/demofiles/mod-84/xslfo/image/external-ex3.html>

Les images *"inline"* de l'exemple ci-dessus sont alignées verticalement au sein de la ligne grâce à la propriété *"alignment-baseline"* :

```
<fo:external-graphic
  content-height="1em"
  alignment-baseline="mathematical"
  src="url(triangle-right.gif)" />
```

Les blocs flottants (élément *"fo:float"*) sont partiellement implémentés par **FOP**. La solution la plus sûre pour positionner une image par rapport à du texte reste toutefois l'emploi de tables.

 **FOP** floats

[<https://xmlgraphics.apache.org/fop/fo.html#floats>]

Revoir l'exemple 2 ci-dessus :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/image/external-ex2.html>]

11.7.2 Graphiques internes

"fo:instream-foreign-object" est un autre élément qui permet d'insérer des objets *non-XSL* dans un document **XSL-FO**.

La différence essentielle avec *"fo:external-graphic"* est que l'élément n'est pas référencé mais **inclus** dans le code source du document. La conclusion immédiate est qu'il doit a priori s'agir d'un format **XML** reconnu par le processeur d'impression visé.

Le seul format éligible à l'heure actuelle (*quel que soit le processeur d'impression*) est **SVG** :

```
<fo:block text-align="center">
  <fo:instream-foreign-object width="15cm">
    <svg xmlns="http://www.w3.org/2000/svg"
      width="15cm" height="3cm" viewBox="0 0 1000 200">
      ...
    </svg>
  </fo:instream-foreign-object>
</fo:block>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/image/instream-ex1.html>]

Bien entendu, l'usage d'un format graphique **XML** apporte d'insignes avantages, comme le fait de pouvoir générer le graphique lui-même à l'aide de **XSLT** ou d'autres outils compatibles **XML** (*et sous cet aspect, l'exemple ci-dessus n'est pas exempt de magie...*).

11.8 Gestion des pages

11.8.1 Numérotation des pages

L'élément *"fo:page-number"* permet d'insérer le numéro de la page courante.

Il est couramment utilisé pour la numérotation automatique des pages, et de ce fait en général inséré dans la section *"fo:static-content"* qui correspond au bas de page :

```
<fo:static-content flow-name="xsl-region-after"
  text-align="center" font-style="italic" font-size="10pt">
  <fo:block font-weight="bold">
    - Page <fo:page-number/> -
  </fo:block>
</fo:static-content>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/number-ex1.html>]

11.8.2 Référence à une page

L'élément `<fo:page-number-citation>` permet d'insérer le numéro de la page contenant l'élément dont l'identifiant correspond à la valeur de l'attribut `ref-id`. Cet élément est destiné à la génération de tables des matières, de références croisées et d'index.

Une astuce permet également de l'utiliser pour obtenir le nombre total de pages d'un document. Il suffit pour cela d'insérer en fin de document un bloc (*même vide*) portant un identifiant connu :

sonde en fin de document

```
<fo:flow flow-name="xsl-region-body">
  ...
  <fo:block id="last-page"/>
</fo:flow>
```

puis de faire référence à cet élément à chaque fois que nécessaire :

référence

```
Page <fo:page-number/>
sur <fo:page-number-citation ref-id="last-page"/>
```

Exemple :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/citation-ex1.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/citation-ex1.html)

11.8.3 Gestion des sauts de page

Le fonctionnement normal d'un processeur d'impression consiste à insérer automatiquement des sauts de page aux endroits adéquats. Des nécessités de présentation peuvent toutefois inciter l'auteur à vouloir contrôler la manière dont ils s'effectuent, voire à imposer certains sauts de page.

> Contrôle des sauts de page

On désire parfois interdire les sauts de page entre certains éléments, comme par exemple entre un titre et le premier paragraphe d'une section. Cette possibilité est offerte grâce aux propriétés `keep-with-next.within-page`, `keep-with-previous.within-page` et `keep-together.within-page`.

```
<fo:block font-weight="bold" space-before="1em" keep-with-next.within-page="1">
  Calme précurseur d'événements orageux.
</fo:block>
```

La valeur par défaut de ces propriétés est `auto`, ce qui autorise un saut de page entre éléments. La valeur `always` interdit le saut de page, tandis qu'une valeur numérique entière décourage d'autant plus le saut de page que la valeur est élevée.

Bien utilisé, ce mécanisme est très puissant pour prioriser les endroits où les sauts de page sont autorisés, et permet de générer des documents de manière automatique sans quasiment aucune retouche manuelle.

Exemple sans contrôle des sauts de page :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex5.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex5.html)

Le même avec contrôle :

[\[http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex6.html\]](http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex6.html)

> Insertion d'un saut de page

Les propriétés `break-after` et `break-before` servent à forcer un saut de page. Les valeurs possibles sont `auto`, `column`, `page`, `even-page`, et `odd-page`. La valeur par défaut est `auto`.

Ces diverses valeurs permettent de provoquer à l'endroit adéquat, un simple saut de page (*valeur page*), ou d'imposer que l'élément concerné (*i.e. l'élément courant ou le suivant*) se trouve sur une page paire (*even-page*) ou impaire (*odd-page*), ou dans la colonne suivante en cas d'impression multi-colonne (*vue plus loin*).

Voici comment introduire un saut de page à l'aide d'un bloc vide :

```
<fo:block break-after="page"/>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex1.html>]

> Veuves et orphelines

La propriété **"orphans"** spécifie le nombre minimum de lignes d'un paragraphe coupé en bas de page. De manière similaire, la propriété **"widows"** impose un nombre minimum de lignes pour un paragraphe coupé en haut de page.

```
<fo:flow flow-name="xsl-region-body" orphans="3" widows="3">
```

Exemple avec les valeurs par défaut (orphans=2, widows=2) :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex2.html>]

Exemple avec orphans=3 :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex3.html>]

Exemple avec widows=3 :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/page/break-ex4.html>]

11.9 Mise en page avancée

11.9.1 Impression multi-colonnes

La génération d'un document dont le texte est disposé sur plusieurs colonnes est élémentaire.

Il suffit pour cela de préciser le nombre de colonnes désiré grâce à la propriété **"column-count"** de l'élément **"fo:region-body"**. La valeur par défaut de cette propriété est bien sûr **1**.

Une fois le texte s'affichant sur plusieurs colonnes, l'espace entre celles-ci est ajusté à l'aide de la propriété **"column-gap"**, dont la valeur par défaut est **"12pt"**.

```
<fo:simple-page-master master-name="page-unique" ... >
  <fo:region-before extent="1.5cm"/>
  <fo:region-after extent="2cm"/>
  <fo:region-body ... column-count="2" column-gap="1cm"/>
</fo:simple-page-master>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/column-ex1.html>]

11.9.2 Séquence avec page de titre

Si de nombreux documents simples (*bons de commande, factures, articles, ...*) peuvent se contenter d'un unique modèle de page, il en est d'autres (*thèses, cours, livres, ...*) qui nécessitent d'enchaîner des pages dont la présentation varie en fonction du contexte.

Le modèle développé ci-dessous permet d'enchaîner une page de titre, suivie par une série de pages identiques. La page de titre possède des marges différentes des pages normales, et ne comporte pas d'entête ni de bas de page. Le contenu des pages normales est présenté sur deux colonnes.

La première chose à faire consiste à décrire les divers modèles de page qui seront utilisés par la suite :

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="premiere-page" ... >
    <fo:region-body margin-top="10cm" margin-bottom="2.1cm"/>
  </fo:simple-page-master>
  <fo:simple-page-master master-name="page-normale" ... >
    <fo:region-before extent="1.5cm"/>
    <fo:region-after extent="2cm"/>
    <fo:region-body ... column-count="2" column-gap="1.5cm"/>
  </fo:simple-page-master>
  ...
</fo:layout-master-set>
```


La seconde étape conduit à mettre en place la séquence suivant laquelle on passe d'un modèle de page à un autre :

```
<fo:layout-master-set>
...
<fo:page-sequence-master master-name="document">
  <fo:repeatable-page-master-alternatives>
    <fo:conditional-page-master-reference
      master-reference="premiere-page" page-position="first"/>
    <fo:conditional-page-master-reference
      master-reference="page-normale" page-position="rest"/>
  </fo:repeatable-page-master-alternatives>
</fo:page-sequence-master>
</fo:layout-master-set>
```

Il n'y a plus ensuite qu'à faire référence au modèle de document tel que défini ci-dessus lors de la génération d'une séquence de pages :

```
<fo:page-sequence master-reference="document" ... >
...
</fo:page-sequence>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/cover-ex1.html>]

Pour obtenir un document comportant plusieurs séquences introduites par des pages de titre (cf. *chapitres d'un livre par exemple*), il suffit de faire se succéder plusieurs éléments *"fo:page-sequence"* (la condition *page-position="first"* s'adresse à la première page d'une séquence).

11.9.3 Pages paires et impaires

> Séquence simple

Le principe permettant de générer un document avec un modèle différent pour les pages paires et impaires est exactement le même. La seule différence se situe au niveau du test associé à l'élément *"fo:conditional-page-master-reference"* :

```
<fo:repeatable-page-master-alternatives>
  <fo:conditional-page-master-reference
    master-reference="page-paire" odd-or-even="even"/>
  <fo:conditional-page-master-reference
    master-reference="page-impair" odd-or-even="odd"/>
</fo:repeatable-page-master-alternatives>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/evenodd-ex1.html>]

> Entêtes et bas de page adaptés

Pour disposer d'entêtes ou de bas de pages différents suivant les modèles de page, il faut commencer par nommer les régions spécifiques au sein des modèles (les valeurs *"xsl-region-before"* et *"xsl-region-after"* ne sont que des valeurs par défaut) :

modèle pages impaires

```
<fo:simple-page-master master-name="page-impair" ... >
  <fo:region-before extent="1.5cm" region-name="haut-impair"/>
  <fo:region-after extent="2cm"/>
  <fo:region-body ... column-count="2" column-gap="1.5cm"/>
</fo:simple-page-master>
```


modèle pages paires

```
<fo:simple-page-master master-name="page-paire" ... >
  <fo:region-before extent="1.5cm" region-name="haut-pair"/>
  <fo:region-after extent="2cm"/>
  <fo:region-body ... column-count="2" column-gap="1.5cm"/>
</fo:simple-page-master>
```

Il faut ensuite définir le contenu des zones ainsi évoquées à l'aide d'éléments *"fo-static-content"* :

entête pages impaires

```
<fo:static-content flow-name="haut-impair">
  <fo:block text-align="right" font-size="8pt">
    Exemple de document XSL-FO
  </fo:block>
</fo:static-content>
```

entête pages paires

```
<fo:static-content flow-name="haut-pair">
  <fo:block text-align="left" font-size="8pt">
    Exemple de document XSL-FO
  </fo:block>
</fo:static-content>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/evenodd-ex2.html>]

11.9.4 Séquences de pages complexes

> Séquences pair/impair avec page de titre

La logique permettant d'enchaîner des séquences de pages comportant à chaque fois une page de titre, suivie par des pages paires et impaires dont le modèle diffère, est plus complexe. En effet, on désire en général que le titre se trouve sur une page impaire

Cette contrainte est matérialisée par la propriété *"initial-page-number"* de l'élément *"fo:page-sequence"* concerné. Les valeurs possibles sont *"auto"*, *"auto-odd"*, *"auto-even"* ou un numéro de page. La valeur par défaut est *"auto"*.

```
<fo:page-sequence ... initial-page-number="auto-odd">
  ...
</fo:page-sequence>
```

Le modèle des pages à enchaîner dans une même séquence est :

```
<fo:repeatable-page-master-alternatives>
  <fo:conditional-page-master-reference
    master-reference="premiere-page" page-position="first"/>
  <fo:conditional-page-master-reference
    master-reference="page-paire" odd-or-even="even"/>
  <fo:conditional-page-master-reference
    master-reference="page-impair" odd-or-even="odd"/>
</fo:repeatable-page-master-alternatives>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/complex-ex1.html>]

> Pages blanches

L'exemple ci-dessus a montré qu'un processeur d'impression peut être amené à insérer des pages blanches pour satisfaire certaines conditions d'enchaînement des séquences. Une dernière subtilité consiste à spécifier un modèle différent pour les pages blanches.

En prenant en compte cette dernière possibilité, les propriétés conditionnelles qui s'appliquent aux éléments *"fo:conditional-page-master-reference"* pour sélectionner le modèle de page approprié sont donc :

- *"page-position"* : valeurs possibles *"first"*, *"last"* ou *"rest"*,
- *"odd-or-even"* : valeurs possibles *"even"*, *"odd"* ou *"any"*,
- *"blank-or-not-blank"* : valeurs possibles *"blank"*, *"not-blank"* ou *"any"*.

Le modèle de la séquence de pages devient :

```
<fo:repeatable-page-master-alternatives>
  <fo:conditional-page-master-reference
    master-reference="premiere-page" page-position="first"/>
  <fo:conditional-page-master-reference
    master-reference="page-blanche" blank-or-not-blank="blank"/>
  <fo:conditional-page-master-reference
    master-reference="page-paire" odd-or-even="even"/>
  <fo:conditional-page-master-reference
    master-reference="page-impaire" odd-or-even="odd"/>
</fo:repeatable-page-master-alternatives>
```

et pour compléter le tout, les sauts de page après la page de titre ont été modifiés de manière à ce que la première page de contenu soit elle aussi une page impaire :

```
<fo:block break-after="odd-page"/>
```

Exemple :

[<http://dmolinarius.github.io/demofiles/mod-84/xslfo/adv-page/complex-ex2.html>]