# 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/XSLTi#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)*.

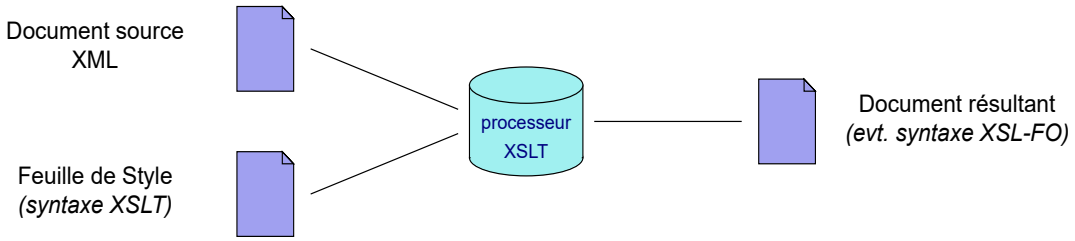
☞ Consulter la page principale sur XSL
[http://www.w3.org/Style/XSL/]

### 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

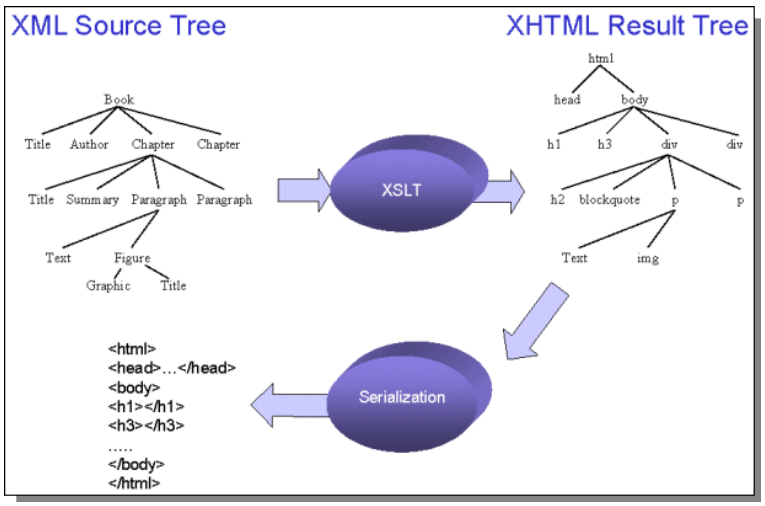
Une feuille de style **XSLT** contient des **modèles** *(templates)*.

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 le 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'autre 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'élt "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 formées 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]

```
<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 connait 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]

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]

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]

certains 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]

**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]

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]

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]

... 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]

## 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/> 
   <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"/> 
   <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="ancre"/></h2>
<xsl:apply-templates/>
```

et pour les blocs :

```
<xsl:when test="@titre">
 <h3><xsl:call-template name="ancre"/></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="ancre">
 <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="ancre">
 <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:apply-templates/>
 </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]

**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.xsl" type="text/xsl"?>
```

voici comment procéder :

```
<xsl:processing-instruction name="xml-stylesheet">
 <xsl:text>href="style.xsl" 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]