

XSLT

eXtensible Stylesheet Language Transformations

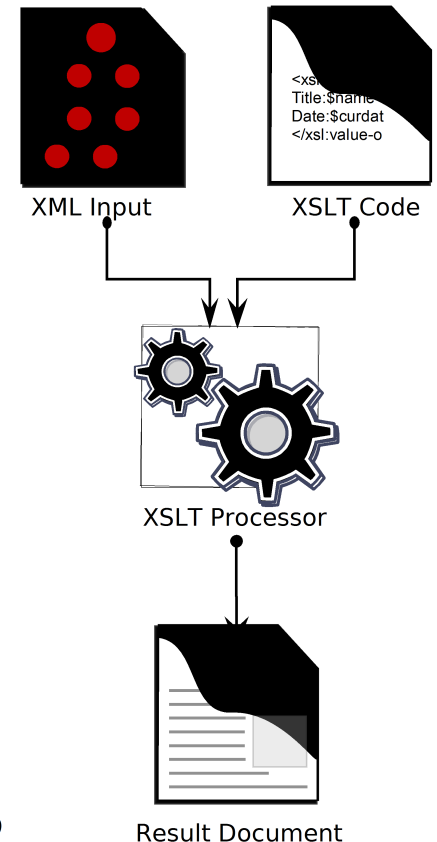
Didier Le Foll

Sources

- Ce cours est largement inspiré des sites suivants :
 - <http://www.formation.jussieu.fr/~perrot/EPITA/XMLA/Cours2/XSLT>
 - <http://www.liafa.univ-paris-diderot.fr/~carton/Enseignement/XML/Cours/XSLT>
 - <http://tecfa.unige.ch/guides/tie/pdf/files/xml-xslt.pdf>
 - <http://en.wikipedia.org/wiki/XSLT>

Généralités

- *XSLT* (*eXtensible Stylesheet Language Transformations*) est un langage de programmation à part entière,
 - spécialisé dans la génération de texte à partir de XML,
 - muni d'une syntaxe en XML, pesante, mais qui a l'intérêt majeur d'autoriser la génération et l'analyse par les outils standard de la technologie XML,
 - exécutable par divers procédés, notamment à travers d'autres langages de programmation, ce qui lui confère une situation originale dans le paysage informatique.



Généralités (2)

- La mise en œuvre (exécution) d'une feuille de style (=programme) XSLT peut se faire :
 - directement **dans un navigateur** : tous les navigateurs récents sont capables d'afficher, directement (pas de fichier résultat généré), le résultat d'une transformation XSLT vers HTML d'un fichier source XML.
 - explicitement **via un interpréteur** (un fichier résultat est généré):
 - *xsltproc* est l'interpréteur de la bibliothèque *libxml2*. Il peut être appelé en tant que commande *Linux*.
 - *saxon* est un interpréteur disponible sous un environnement Java et .NET (*Windows* et *Linux*). C'est le seul qui en 2012 supporte XSLT 2.0.
 - **à travers un langage de programmation** : *PHP*, *Java*, *Javascript* (entre autres) possèdent des fonctions ou des classes permettant d'exécuter des transformations XSLT.

Généralités (3)

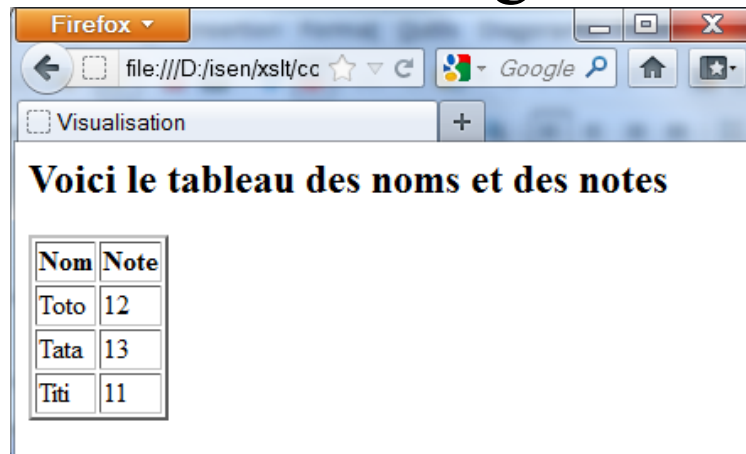
- XSLT est une partie du projet *XSL (eXtensible Stylesheet Language)* du W3C. Ce projet a aussi produit *XSL-FO*, *XPath* et *XQuery*.
- **XSLT 1.0** est une recommandation W3C de 1999. La nouvelle version, **XSLT 2.0**, est une recommandation W3C depuis 2007. Cependant XSLT 1.0 est encore largement utilisé, du fait que les implémentations XSLT natives des navigateurs ne supportent pas XSLT 2.0.
- XSLT est très lié à **XPath** qui permet de définir des sous-ensembles de l'arbre du document source. XPath fournit aussi une gamme de fonctions utilisable par XSLT. XSLT 1.0 utilise XPath 1.0 et XSLT 2.0 utilise XPath 2.0.

Un exemple simple (1)

- A partir du fichier XML suivant (*notes.xml*) :

```
<?xml version="1.0" ?>
<liste>
  <eleve>
    <nom>Toto</nom><note>12</note>
  </eleve>
  <eleve>
    <nom>Tata</nom><note>13</note>
  </eleve>
  <eleve>
    <nom>Titu</nom><note>11</note>
  </eleve>
</liste>
```

on veut obtenir l'affichage suivant dans un navigateur :



Un exemple simple (2)




- Pour cela, il faut transformer l'arbre XML de départ en un texte HTML interprétable par le navigateur. La présentation désirée (cf. p. précédente) correspond à :

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>Visualisation</title>
</head>
<body>
  <h2> Voici le tableau des noms et des notes</h2>
  <table border="2">
    <tr><th>Nom</th><th>Note</th></tr>
    <tr><td>Toto</td><td>12</td></tr>
    <tr><td>Tata</td><td>13</td></tr>
    <tr><td>Titi</td><td>11</td></tr>
  </table>
</body>
</html>
```

Un exemple simple (3)

- C'est ce que fait le programme XSLT suivant (*notes.xsl*) :

```
<?xml version='1.0'?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version='1.0'>
  <xsl:output method="html" indent="yes"/>
  <xsl:template match="/"> Règle 1
    <html><head><title>Visualisation</title></head>
    <body>
      <h2> Voici le tableau des noms et des notes</h2>
      <table border="2">
        <tr><th>Nom</th><th>Note</th></tr>
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>
  <xsl:template match="liste/eleve"> Règle 2
    <tr><td><xsl:value-of select="nom"/></td>
      <td><xsl:value-of select="note"/></td></tr>
  </xsl:template>
</xsl:stylesheet>
```

	Partie purement XSLT
	Partie fixe HTML
	Expressions XPath

Un exemple simple (4)

- Si on veut que **le résultat s'affiche directement dans le navigateur** à partir du fichier XML source (*notes.xml*), il faut ajouter dans son prologue la ligne suivante :

```
<?xml-stylesheet type="text/xsl" href="notes.xsl"?>
```

Dans ce cas, il n'y a aucun fichier HTML généré. Si on clique sur « Code source de la page », c'est le fichier XML de départ qui s'affiche.

- Si on veut un **fichier HTML généré**, il faut utiliser un interpréteur. Ex. avec *saxon* (.NET) et *xsltproc* (Linux) :

```
C:\Program Files\Saxonica\SaxonHE9.4N\bin\Transform.exe  
-s:notes.xml -xsl:notes.xsl -o:notes.html
```

```
xsltproc notes.xsl notes.xml > notes.html
```

Dans ces deux cas il n'est pas nécessaire d'ajouter une instruction de traitement `<?xml-stylesheet>` dans le prologue du fichier *notes.xml*.

Un exemple simple : explications (1)

- Dans le fichier XSLT :
 - Les balises qui relèvent de l'espace de noms XSLT (préfixées par "*xsl:*") sont interprétées par le processeur XSLT.
 - Les autres balises (<*html*>, etc...) sont considérées comme du texte et envoyées directement en sortie.
- Les éléments <*xsl:template*> représentent des règles. Leur attribut "*match*" détermine leur domaine d'application : la valeur donnée à cet attribut doit être un motif *XPath*. Les motifs *XPath* sont des expressions *XPath* simplifiées (des chemins) permettant de sélectionner des nœuds dans un document (cf. p. suivante).

Motifs simples (chemins) XPath

Élément syntaxique	(Type de chemin)	Exemple d'un chemin	Exemple d'un match réussi par rapport au chemin indiqué à gauche
balise	nom d'élément	project	<project> </project>
/	sépare enfants directs	project/title	<project><title> ... </title>
		/	(correspond à l'élément racine)
//	descendant	project//title	<project><problem><title>....</title>
		//title	<racine>...<title>..</title> (n'importe où)
*	"wildcard"	*/title	<bla><title>..</title> et <bli><title>...</title>
 	opérateur "ou"	title head	<title>...</title> <i>ou</i> <head> ...</head>
		* / @*	(tous les éléments: les enfants, la racine et les attributs de la racine)
.	élément courant	.	
../	élément supérieur	../problem	<project>
@	nom d'attribut	@id	<xyz id="test">...</xyz>
		project/@id	<project id="test" ...> ... </project>
@attr='type'		list[@type='ol']	<list type="ol"> </list>

Une variation de l'exemple : attributs

- On veut obtenir le même résultat à partir d'un fichier XML où les noms et les notes sont représentés par des **attributs** et non par les éléments-fils :

```
<?xml version="1.0" ?>
<liste>
  <eleve nom="Toto" note="12"/>
  <eleve nom="Tata" note="13"/>
  <eleve nom="Titi" note="11"/>
</liste>
```

- Il suffit de **modifier la seconde règle** ainsi :

```
<xsl:template match="liste/eleve">
  <tr>
    <td><xsl:value-of select="@nom"/></td>
    <td><xsl:value-of select="@note"/></td>
  </tr>
</xsl:template>
```

Les mécanismes (1) : les règles

- L'unité de base de la programmation XSLT est **la règle** :
 - Une règle s'applique à un nœud et insère le texte résultat dans le texte en construction. La **donnée d'une règle** doit donc spécifier :
 - à quels nœuds elle s'applique (partie *filtre*),
 - comment calculer son résultat à partir du nœud en question (partie *forme*),
 - éventuellement des indications de priorité et de désignation.
 - La **syntaxe d'une règle** est celle d'un élément *xsl:template* avec un attribut *match* représentant le filtre et un contenu représentant la forme :

```
<xsl:template match="le filtre">  
    la forme  
</xsl:template>
```
 - Les **règles** sont **toutes déclarées au même niveau**, directement dans l'élément-racine du fichier XSL, à savoir *<xsl:stylesheet>*.
 - Le programme XSLT (ou "feuille de style") apparaît donc essentiellement comme une **collection "plate" de règles**, chacune étant activée (le plus souvent) par *<xsl:apply-templates/>* . Il contient aussi quelques éléments accessoires (*xsl:output*, *xsl:strip-space*, etc..., cf. p.22) qui modifient le détail du traitement sans en changer le principe.

Les mécanismes (2) : la récursivité

- À chaque étape du calcul, l'interpréteur XSLT gère une **pile de nœuds candidats** : au commencement, cette pile contient un seul nœud qui est le **nœud racine** ("/") qui représente le fichier XML tout entier, dont l'élément-racine est un fils, le seul qui nous intéresse.
- La **collection de toutes les règles est essayée sur le nœud qui est au-dessus de la pile**. Appelons P ce nœud. Normalement, au plus une seule règle doit se révéler applicable (via l'évaluation de son attribut *match*). Sinon, un mécanisme de priorités entre en jeu pour sélectionner une règle entre plusieurs possibles.
- **Si aucune règle** ne s'avère applicable à P, un **mécanisme récursif** se déclenche **automatiquement** (règles implicites, *default rules*).
 - le contenu textuel (éventuel) de P est envoyé en sortie (Règle 1),
 - la liste de ses nœuds-enfants est empilée sur la pile des nœuds candidats (Règle 2),
 - et on passe au premier de cette liste (c-à-d. on essaie toutes les règles sur ce nœud).Lorsque le nœud P n'a pas d'enfants, on passe au suivant dans la pile. Le calcul s'arrête lorsque la pile des candidats est épuisée.
- **Si une règle est applicable** au nœud considéré, elle s'applique (voir p. suivante). Cette application va d'une part produire du texte en sortie, d'autre part faire apparaître de nouveaux nœuds candidats, qui sont empilés sur la pile gérée par l'interpréteur (mécanisme récursif).

Les mécanismes (3) : application d'une règle

- **Application d'une règle à un nœud** de l'arbre XML :
C'est par rapport à ce nœud que sont évaluées les expressions qui apparaissent dans les attributs *select* (exemples ci-dessus) et *test* (voir plus loin) du corps de la règle.
- Quelques exemples dans les pages suivantes :

Les mécanismes (3) : ex. 1

- L'application de la règle 2 du fichier *notes.xsl* (p. 8)

```
<xsl:template match="liste/eleve">
  <tr><td><xsl:value-of select="nom"/></td>
    <td><xsl:value-of select="note"/></td></tr>
</xsl:template>
```

au nœud (élément) <eleve> suivant :

```
<eleve>
  <nom>Toto</nom><note>12</note>
</eleve>
```

s'explique ainsi : la chaîne "nom" dans select="nom" s'évalue comme le premier nœud-fils de l'élève en question dont la balise est "nom", et l'opérateur *xsl:value-of* renvoie la chaîne "Toto".

De même pour "note" dans select="note" et l'opérateur *xsl:value-of* renvoie la chaîne "12".

Après quoi on passe au prochain nœud à traiter, qui en l'occurrence sera le suivant dans l'arbre XML, à savoir :

```
<eleve>
  <nom>Tata</nom><note>13</note>
</eleve>
```


Les mécanismes (3) : ex. 2

- L'application de la règle 1 du fichier *notes.xsl* (p. 8), sur le nœud-racine ("*/*"), engendre d'abord du texte en sortie, à savoir :

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Visualisation</title>
  </head>
  <body>
    <h2> Voici le tableau des noms et des notes</h2>
    <table border="2">
      <tr><th>Nom</th><th>Note</th></tr>
```

Après quoi, l'exécution de `<xsl:apply-templates />` provoque l'activation du mécanisme récursif :

- la liste des nœuds-enfants du nœud courant est empilée sur la pile des nœuds-candidats : ici le nœud-racine a un seul fils, l'élément `<liste>`,
- on essaie toutes les règles sur ce nœud : aucune ne s'applique.
- Par conséquent les règles par défaut s'enclenchent : R1) tous les enfants de `<liste>` sont empilés, R2) comme `<liste>` n'a pas de contenu textuel, rien n'est envoyé en sortie.

Les mécanismes (3) : ex. 2 suite

- on passe au premier nœud candidat, qui sera :

`<eleve>`

`<nom>Toto</nom><note>12</note>`

`</eleve>`

- on essaie toutes les règles sur ce nœud : ici la (seule) règle applicable est la règle 2, puisqu'il s'agit bien d'un `<eleve>` qui est fils immédiat d'une `<liste>`. Ce qui se produit alors a été décrit en ex. 1 (p. 16).
- Après quoi on passe au deuxième fils de `<liste>` , etc... jusqu'à épuisement de la liste.
- après que le dernier fils de `<liste>` ait été traité, la pile de nœuds créée par le `<xsl:apply-templates />` de la première règle est épuisée. On revient donc à l'exécution de cette première règle, en séquence, et on envoie en sortie les balises HTML fermantes :

`</table>`

`</body>`

`</html>` et le calcul se termine.

- Remarque : on obtiendrait exactement le même résultat :

- en donnant comme filtre à la règle 1 : `match="/liste"` au lieu de `match="/"`,
- en donnant comme filtre à la règle 2 : `match="/liste/eleve"` ou bien `match="eleve"` au lieu de `match="liste/eleve"`

Les mécanismes (4) : retour sur les règles implicites (ou par défaut)

- Les **règles implicites**, qui s'appliquent en l'absence de règle applicable à un nœud, provoquent un **parcours récursif de l'arbre en profondeur d'abord**, envoyant en sortie le contenu textuel de l'arbre tout entier (concaténation des contenus de tous ses nœuds, dans l'ordre du parcours).
- Par conséquent, la **feuille de style vide** fait quelque chose : elle extrait fidèlement le contenu textuel du fichier source (avec tous les blancs et sauts de ligne) :

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

- Pour obtenir l'inaction totale, il faut donc inhiber ce fonctionnement par défaut, et pour cela écrire une règle applicable au nœud-racine (filtre = "/"), et qui ne produise rien (forme vide). En effet, lorsqu'une règle est applicable à un nœud, elle inhibe l'effet de la "propagation par défaut" aux fils du nœud en question.

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

Les mécanismes (5) : priorité des règles

- Quand plusieurs règles peuvent s'appliquer à un même élément, le processeur XSLT en choisira **une seule**. La règle choisie sera celle qui a **la plus haute priorité** parmi les règles applicables.
- L'attribution d'une priorité aux règles se fait selon **deux mécanismes** : implicite et explicite.
- **Priorités implicites** :
 - Les règles exprimées (*templates*) sont prioritaires par rapport aux règles implicites.
 - Les règles "plus spécifiques" sont prioritaires par rapport aux règles "plus générales". Ceci dépend de la **forme du motif** contenu dans l'attribut *match*. Par exemple, un motif "eleve" est prioritaire par rapport à un motif "*", un motif "liste/eleve" est prioritaire par rapport à un motif "eleve". Le processeur calcule une priorité (un nombre décimal entre -0.5 et +0,5) en se basant sur la forme du motif.
- **Priorités explicites** : la priorité peut être fixée, sous forme d'un nombre décimal, par un attribut *priority* de l'élément `<xsl:template>`. Par ex :

```
<xsl:template match="unElement" priority="2">
```

pour forcer une priorité plus élevée à cette règle par rapport aux priorités implicites.

Pilotage des règles par `<xsl:sort.../>`

- Les règles lancées par `<xsl:apply-templates/>` s'appliquent aux nœuds **dans l'ordre où ils apparaissent** dans le document, et par conséquent leurs produits figurent dans le même ordre en sortie.
- Dans de nombreux cas, typiquement dans le traitement de listes, on souhaite obtenir un **résultat trié** (ordre alphabétique ou numérique). On utilise pour cela l'instruction `<xsl:sort ... />`, à l'intérieur d'un `<xsl:apply-templates>` (ou d'un `<xsl:for-each>`, cf. p.29).
- Ex. simple d'utilisation :

```
<xsl:apply-templates>  
  <xsl:sort select="nom"/>  
</xsl:apply-templates>
```
- En plus de *select*, `<xsl:sort ... />` peut porter les attributs :
 - *order* : ascending (par défaut) ou descending,
 - *data-type* : text (=ordre alphabétique, par défaut) ou number.
- On peut indiquer **plusieurs critères de tri** : il suffit pour cela de mettre, dans l'ordre, plusieurs `<xsl:sort ... />` à l'intérieur d'un `<xsl:apply-templates>`.

Autres éléments XSLT de premier niveau

- Au même niveau que les *templates*, directement dans l'élément-racine `<xsl:stylesheet>`, on peut trouver d'autres éléments :
 - L'élément `<xsl:output />`, comme son nom l'indique, gouverne la production du fichier-résultat. Il peut porter comme principaux attributs :
 - *method* : `xml` (par défaut), `html` ou `text`,
 - *encoding* : `utf-8` (par défaut), `iso-8859-1`,...
 - *indent* : `no` (par défaut) ou `yes` (génère automatiquement des indentations si on veut améliorer la lisibilité du fichier résultat),

Ex : `<xsl:output method="html" indent="yes"
encoding="iso-8859-1"/>`

- Les éléments `<xsl:strip-space />` et `<xsl:preserve-space />` permettent de contrôler les nœuds "blancs" (nœuds textuels contenant uniquement des caractères d'espacement) du document source avant de commencer la transformation (par défaut, les nœuds "blancs" ne sont pas supprimés). Ex : suppression de tous les les nœuds "blancs" du document, sauf ceux des éléments B et C :

`<xsl:strip-space elements="*" />`

`<xsl:preserve-space elements="B C" />`

Les fonctions et opérateurs (1)

- Le langage *XPath* fournit des **opérateurs** :
 - **Numériques** : +, - (attention : espace obligatoire de chaque côté), *, div (division), mod (modulo),
 - **Booléens** : and, or
- Le langage *XPath* fournit aussi de **nombreuses fonctions**, complétées par d'autres fournies par XSLT. Elles sont utilisables dans toute expression XPath apparaissant dans un programme XSLT. Les fonctions les plus utilisées sont :
 - **name()** : renvoie le nom (chaîne de caractères) du nœud courant (par ex. le nom de la balise si c'est un nœud de type "élément"),
 - **position()** : renvoie le rang du nœud courant au sein du contexte (en général dans sa "fratrie"), le nœud le plus à gauche a le rang 1,
 - **last()** : renvoie le rang du dernier nœud dans le contexte (en général la "fratrie" du nœud courant) = nombre de nœuds du contexte),
 - **count(*expression XPath = ensemble de noeuds*)** : renvoie le nombre de nœuds dans l'ensemble de nœuds passé en argument.

Les fonctions et opérateurs (2)

- Il y a aussi un jeu de **fonctions de base** sur les **numériques** : `sum()`, `round()`, `floor()`,... sur les **chaînes** : `concat()`, `contains()`, `substring()`,... sur les **booléens** : `not()`, `true()`, `false()`,...
- Ex. d'**utilisation des fonctions** :

```
<xsl:if test="not(position()=last())">... </xsl:if>
```

équivalent à :

```
<xsl:if test="position() != last()">... </xsl:if>
```

Exemple de calcul (calcul de moyenne) :

```
<xsl:value-of select="sum(/liste/eleve/note) div  
count(/liste/eleve)" />
```

ou bien, arrondi à l'entier le plus proche :

```
<xsl:value-of select="round( sum(/liste/eleve/note) div  
count(/liste/eleve))" />
```


Création dynamique de noeuds XML (1)

- Les **noeuds XML** de type élément, texte, commentaire, attribut, instruction de traitement peuvent être générés à l'aide, respectivement, des instructions XSLT : `<xsl:element>`, `<xsl:text>`, `<xsl:comment>`, `<xsl:attribute>`, `<xsl:processing-instruction>`.
- `<xsl:text>` est notamment utilisé pour produire des noeuds texte contenant uniquement des blancs (ceci est impossible autrement). Ex:

```
<xsl:value-of select="nom"/><xsl:text> </xsl:text>
<xsl:value-of select="prenom"/>
```
- `<xsl:element>` et `<xsl:attribute>` ont un attribut "name" obligatoire. En fait tous les exemples montrés précédemment, où des éléments "fixes" étaient créés (ex: `<table border="2">`) pouvaient être écrits :

```
<xsl:element name="table">
  <xsl:attribute name="border">2</xsl:attribute>
  .... contenu de l'élément "table" ....
</xsl:element>
```

Cette écriture, beaucoup plus lourde, n'a pas d'intérêt dans ce cas.

Création dynamique de noeuds XML (2)

- L'utilisation de `<xsl:attribute>` permet de **générer dynamiquement un attribut** dont le contenu et/ou le nom ne sont connus qu'à l'exécution (c'est-à-dire non connu à l'avance). Ex. : transformation d'un élément "xyz" en attribut d'un élément "abc" (ici, le nom est connu à l'avance, mais pas le contenu) :

```
<abc><xsl:attribute name="xyz">  
    <xsl:value-of select="xyz"/>  
</xsl:attribute></abc>
```

qui peut s'écrire plus simplement :

```
<abc xyz="{xyz}"/>
```

Remarque : " { xyz } " est appelé « **attribut interpolé** ». Les accolades indiquent au processeur de remplacer l'expression par sa valeur.

- L'utilisation de `<xsl:element>` permet de **générer dynamiquement un élément** dont le nom n'est connu qu'à l'exécution. Ex. : transformation de la valeur d'un attribut "xyz" en élément :

```
<xsl:element name="{@xyz}">  
    <xsl:value-of select="."/>  
</xsl:element>
```

Copie de nœuds XML

- XSLT permet de copier des nœuds du document source vers le document résultat. La copie peut concerner **un seul nœud** (c'est-à-dire sans ses descendants) avec `<xsl:copy>` (copie superficielle), ou bien elle peut se faire **en profondeur** (c'est-à-dire avec tous ses descendants) avec `<xsl:copy-of>`.
- Une utilisation courante de `<xsl:copy>` consiste à recopier un document en excluant certains éléments ou certains types de nœuds. Ex. : recopie du document *"livres.xml"* en excluant l'attribut *"editeur"* :

```
<xsl:template match="@*|*">
  <xsl:copy>
    <xsl:apply-templates select="@*|*" />
  </xsl:copy>
</xsl:template>
<xsl:template match="@editeur" />
```

- Ex. d'utilisation de `<xsl:copy-of>` (recopie des livres de langue française):

```
<xsl:template match="INVENTAIRE">
  <xsl:copy>
    <xsl:copy-of select="LIVRE[@lang='fr']" />
  </xsl:copy>
</xsl:template>
```

Deux styles de programmation en XSLT

- Le langage XSLT permet de programmer selon deux styles différents (qu'on peut mélanger) :
 - **Déclaratif** : on donne des **règles** (*templates*), et le processeur XSLT fait le travail. C'est le style "natif" du langage, préconisé par ses promoteurs : XSLT est un **langage à base de règles**.
 - **Impératif** : on utilise les structures habituelles d'un langage de programmation classique (tests, boucles, variables, fonctions).
- Dans les pages suivantes sont développées les structures du style "impératif" de XSLT : `<xsl:if>`, `<xsl:for-each>`, `<xsl:variable>`, l'écriture de fonctions par des *templates* "nommés", `<xsl:param>`,...

Les structures de contrôle

- la **boucle** : `xsl:for-each`, de forme générale :

```
<xsl:for-each select="unFiltre">  
    une action  
</xsl:for-each>
```
- la **conditionnelle simple** ("*si*") : `xsl:if`, de forme générale :

```
<xsl:if test="expression XPath à valeur booléenne">  
    une action  
</xsl:if>
```

Remarque : il n'existe pas de "*sinon*" (*else*) en XSLT.
- la **conditionnelle multiple** ("*selon*" ou "*switch*") : `xsl:choose`, de forme générale :

```
<xsl:choose>  
    <xsl:when test="expression XPath à valeur booléenne">  
        une action</xsl:when>  
    <xsl:when test="..autre..">une autre action</xsl:when>  
        .....  
    <xsl:otherwise>tous les autres cas...</xsl:otherwise>  
</xsl:choose>
```

Les variables

- XSLT connaît la notion de **variable** mais sa valeur est fixée au moment de la déclaration par l'élément `<xsl:variable>` et ne peut plus changer ensuite. C'est donc plutôt à une constante! Une variable peut être de n'importe quel type *XPath* (*number*, *string*, *node-set*,...).
- Le **nom de la variable** est donné par l'attribut *name*.
- La **valeur de la variable** est donnée soit par l'attribut *select*, soit (à défaut) par le contenu de l'élément `<xsl:variable>`.
- La **portée de la variable** est l'élément XSLT qui la contient. Les variables dont la déclaration est enfant de l'élément `<xsl:stylesheet>` sont donc globales.
- **Utilisation** d'une variable : son nom doit être **préfixé** par "\$".
- Ex :

```
<xsl:variable name="moyenne" select="12"/>
<xsl:if test="note < $moyenne">Redouble</xsl:if>
```

Les *templates* "nommés"

- On peut définir des éléments `<xsl:template>` sans attribut *"match"*, mais avec un attribut *"name"* : ce sont des « règles nommées » ou « *named templates* ». On peut les appeler directement avec une instruction `<xsl:call-template>`, en leur passant des paramètres. Cela ressemble aux *fonctions* d'un langage de programmation classique.
- Un *paramètre* est défini, dans une règle, par un élément `<xsl:param>`, ayant un attribut *"name"* obligatoire et un attribut *"select"* optionnel (valeur par défaut). Utilisation d'un paramètre : *préfixé par "\$"*.
- Ex : définition de la « fonction » :

```
<xsl:template name="affiche">
  <xsl:param name="chaîne"/>
  Chaîne :<xsl:value-of select="$chaîne"/>
</xsl:template>
```

appel de la « fonction » :

```
<xsl:call-template name="affiche">
  <xsl:with-param name="chaîne" select="'bonjour'"/>
</xsl:call-template>
```